EFFICIENT PROGRAMMING OF MASSIVE-MEMORY MACHINES

A Dissertation
Presented to
The Academic Faculty

by

Alexander M. Merritt

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2017

Copyright © 2017 by Alexander M. Merritt

EFFICIENT PROGRAMMING OF MASSIVE-MEMORY MACHINES

Approved by:

Professor Ada Gavrilovska, Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Professor Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Moinuddin K. Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Dejan S. Milojicic
Hewlett Packard Labs
Hewlett Packard Enterprise

Date Approved: July 28, 2017

To my wife, who does everything with me.

PREFACE

The latest git version identifier used for the Nibble source code in this dissertation is `dd1adf23d9dbc3c64174668512dff7a79f8dc4d5` on July 14, 2017.

ACKNOWLEDGEMENTS

The work in this dissertation was supported by Karsten Schwan and Ada Gavrilovska, both of my advisors whom I would like to thank, for pushing me and always being available for sharing their guidance.

Many students of our lab shared their time, feedback, and coffee breaks with me, all of which served valuable moments of my life during my PhD at the university. Vishakha Gupta-Cledat was my senior mentor during my early years; Naila Farooqui is a valuable friend, who studied for the qualifier exam with me; Sudarsun Kannan, Jeff Young, Jai Dayal, Hrishikesh Amur, Vishal Gupta, Mukil Kesavan, Priyanka Tembey, Min Lee gave me their time during many random moments; many other colleagues I shared my time with I would like to thank for various aspects. Susie McClain was (and is) utmost supportive in assisting with all aspects surrounding the lab and support for the students (e.g. when needing more paper for printing, or to subsequently shred it).

Many courses were valuable in fostering my knowledge of Computer Science and surrounding disciplines. I particularly enjoyed courses with Moin Qureshi, Nick Feamster, Hyesoon Kim, and opportunities to both be a teaching assistant for and independently teach the undergraduate course CS 3210 Design of Operating Systems (together with Jai Dayal) supported by Matt Wolf and Karsten Schwan. An independent study with Sudhakar Yalamanchili gave me the opportunity to work together with Jeff Young on an alternative aspect of cluster computing.

I would like to specially thank Dejan Milojicic for having a multi-faceted role in my PhD – as a mentor, committee advisor, colleague, and supporter – without whom I would not have learned as much as I know now. He facilitated the effort during one summer to work closely with two colleagues – and now friends – Gerd Zellweger and Izzat El Hajj, all of whom I valued working with very much.

The Graduate Student Counsel provided me with much-needed funding many times to visit conferences throughout the years. H. Venkateswaran I would like to thank for his oft-sought advice.

Not least, I wish to thank each of my PhD committee members – Dejan, Taesoo, Moin, Ada, and Kishore – for their time and feedback to review my work, early drafts of papers, and advice.

TABLE OF CONTENTS

DEDICATION . . . . **iii**

PREFACE . . . . **iv**

ACKNOWLEDGEMENTS . . . . **v**

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

SUMMARY . . . . **xiv**

I INTRODUCTION **1**

1.1 Thesis Statement 4

1.2 Contributions 5

II BACKGROUND AND MOTIVATION **6**

2.1 The Advent of Large Memory Multi-core Machines 6

2.2 Software Challenges in Programming Large Systems 12

2.2.1 Key-Value Stores for Managing Large Datasets 12

2.2.2 Virtual Memory Interfaces in the Operating System are Limiting 16

2.2.3 Summary 19

III SCALABLE MEMORY EFFICIENT KEY-VALUE STORES **20**

3.1 Background and Motivation 20

3.2 Scaling Log-Structured Systems 22

3.2.1 The Index - Enabling Concurrent Operations 22

3.2.2 Supporting Scalable Asynchronous Compaction 23

3.2.3 Prioritizing High, Sustained Write Throughputs 24

IV NIBBLE - A CONCURRENT LOG-STRUCTURED KEY-VALUE STORE **26**

4.1 System Overview 26

4.2 Overview - Index, Objects, and Segments. 26

4.3 Optimistic Concurrency - The Index 28

4.4 Multi-Head Log Allocation 29

4.5 Distributed Hardware Epochs For Coordination 33

4.6 Hardware Assumptions 35

V NIBBLE - EVALUATION **36**

5.1 Breakdown of Components 39

5.2 Memory Overhead 41

5.3 Dynamic Workloads — Postmark 42

5.4 Data-Serving Workloads — YCSB 45

VI AN ANALYSIS OF COMPACTION **49**

6.1 Breakdown of Compaction 49

6.2 Relaxing the nature of the log-structured design 53

6.3 Evaluation – Nibble-2 60

6.4 Summary 63

VII SYSTEMS CHALLENGES WITH IN-MEMORY COMPUTING **65**

7.1 Memory Sizes > Address Bits 65

7.2 Limited control over in-memory data organization 71

7.3 Summary 77

VIII ADDRESS SPACES - NEW ABSTRACTIONS FOR MEMORY **79**

8.1 SpaceJMP - an Operating System Model 80

8.2 Summary 87

IX SPACEJMP - REFERENCE IMPLEMENTATION IN DRAGONFLY BSD **88**

X EVALUATION – PROGRAMMING WITH LARGE ADDRESS SPACES **94**

10.1 Micro-benchmarks 94

10.2 GUPS: Addressing Large Memory 96

10.3 Discussion 98

10.4 Other Applications 99

10.5 Summary 99

XI RELATED WORK **101**

11.1 Memory Management and Allocation 101

11.2 Scalable Key-Value Stores 102

11.3 Log-Structured Systems 102

11.4 Memory Reclamation and Concurrent Data Structures 104

11.5 Garbage collection systems 105
11.6 Operating Systems and Address Spaces 106
11.7 Hardware Address Space Support 109
XII CONCLUSION 111
12.1 Extensions and Opportunities 112
12.1.1 Load-Balancing in Nibble 112
12.1.2 Index Optimizations in Nibble 113
12.1.3 Hybrid DRAM + NVM Machines 114
12.1.4 Hardware Extensions 115
12.1.5 Miscellaneous 116
12.2 Closing Remarks 116
REFERENCES 118

LIST OF TABLES

1 Example single-system large-memory machines. Total Memory bandwidth is a sum of bandwidth to local memory across sockets. *64 TiB is a limit of the CPU, not the platform, able to address only 2^{46} bytes of physical memory. 7

2 Example large-memory networked machines. “The Machine”is a datacenter system, and Summit an HPC machine. 7

3 Summary of compared systems. 37

4 Summary of workloads. 37

5 Large-memory platforms used in this study. 94

6 Breakdown of *context switching*. Measurements on M2 in cycles. Numbers in **bold** are with the use of CPU cache TLB tagging. 94

LIST OF FIGURES

1 HPE Superdome X system design – a custom controller connects to each processor via QPI channels, establishing a large load/store domain to all of memory. Reproduced from [9]. 9

2 Figure showing the effect of data placement within a 16-socket machine on the performance of a concurrent implementation of a hash table (Cuckoo). 10

3 Figure showing a measure of memory bloat in common general-purpose heap allocators. P1-P6 are object sizes. P1-P6 are explained in Figure 14b. 13

4 Figure illustrating a high-level view of log-structured allocation. 14

5 Data region cannot be mapped into the current address space because of a conflict with an existing library mapping. 17

6 Cost of mapping virtual memory using 4 KiB page sizes to preallocated physical memory. Costs are directly a result of page table modifications. 18

7 High-level design of Nibble. A global index is partitioned across sockets, pointing to objects anywhere within the system. Logs, segments, and compaction activities are isolated to the cores and memory within each socket. 27

8 A segment – a container of blocks holding objects. 27

9 The hash table with optimistic concurrency. Buckets are in-lined and entries searched for via linear probing. Nibble allocates 15 entries and one version counter per bucket. Update operations will atomically increment the version to an odd value, blocking new traversals into the bucket, and forcing in-progress reads to restart. Once complete, the version is bumped again, allowing read operations to scan the bucket concurrently. 28

10 Decomposition of one multi-head log in Nibble. 29

11 Illustration of synchronization between application threads recording their current epochs, and compaction threads determining minimum epoch for recycling segment memory (left), and updating the live bytes information for segments (right). 33

12 Performance breakdown of various designs in Nibble: uniform YCSB 240 threads (Mix is a 50:50 ratio of PUT and GET). 39

13 Performance breakdown of use of multiple log heads in Nibble: uniform YCSB 240 threads (Mix is a 50:50 ratio of PUT and GET). With PUT to local memory, more heads provide the greatest initial performance gains. Selecting the core-specific head, we avoid further contention, gaining an additional 15%. 40

14 Measured memory overhead (bloat) 41

15 Characteristics of our trace captured from Postmark. Overall, the workload presents a dynamic behavior: most objects are accessed infrequently, yet a consistently high churn in objects is present. 20% of all operations will add or remove objects, putting stress on each system's object allocator logic. 43

16 Weak scaling of our Postmark trace, measured in time- to-completion, at a consistent 17% memory utilization: as each thread executes the entire trace on a private working set, we increase available memory as appropriate (except for Masstree). Nibble completes the trace within a few minutes, whereas the other systems require upwards of an hour or more with more threads. RAMCloud was able to run with only 2% utilization until system memory was insufficient. 44

17 From Figure 16 we measure time-to-completion for Nibble and MICA at 135 threads, and progressively decrease available memory. At 65% utilization, Nibble's runtime doubles due to compaction costs. MICA seems unaffected, however any such effect is masked by contention on its per-core heap allocators. 44

18 Uniform YCSB throughput measurements on 1 TiB of data – ca. 230 1 KiB objects. "95% RD 5% WR" is a workload with GET composing 95% of all operations and 5% with PUT. No configuration uses DEL. Approximately 8 TiB of storage capacity was provided to ensure a low utilization rate of ca. 12%. 45

19 Zipfian YCSB throughput measurements on 1 TiB of data – ca. 230 1 KiB objects. "95% RD 5% WR" is a workload with GET composing 95% of all operations and 5% with PUT. No configuration uses DEL. Approximately 8 TiB of storage capacity was provided to ensure a low utilization rate of ca. 12%. 46

20 Throughput of MICA and Nibble driven with YCSB (uniform) using all 240 cores, steadily increasing the capacity used. 47

21 Effect of varying the block and segment size in Nibble on the throughput, executing YCSB (uniform) with **1-KiB objects** on smaug-3. 224 client threads in total, with 8 compaction threads per socket, on 1 TiB of data at 90% fill capacity (1.1 TiB total memory allocated to Nibble). Three runs executed, showing the median, and standard deviation. 51

22 Using **500-byte objects**. 51

23 Using **64-byte objects**. 51

24 Effect of varying the number of compaction threads in Nibble (per socket) on the throughput, executing YCSB (uniform) on smaug-3. 224 client threads in total, on 1 TiB of data at 90% fill capacity (1.1 TiB total memory allocated to Nibble). Three runs executed, showing the mean, and standard deviation. 52

25 Latency of memory copy mechanism as the size of the copy region varies. Source and destination locations are in local memory only. 54

26 Performance as a function of memory copy mechanism. Client threads copying in/out objects use the x86 `rep movsb` mechanism; only the compaction's use of memory movement was modified. Shown are four object sizes, executed with 224 client threads at 90% capacity utilization, running YCSB uniform. 54

27 Operations that update existing objects by appending new instances to the log heads create unnecessary work for compaction to perform: object versioning occurs, and is unused in Nibble. Dashed box outlines indicate values that are made stale due to an update. 55

28 By allowing update operations, such as used by PUT, to overwrite existing object locations (subject to constraints such as the new size is ≤ existing allocation size) we can avoid creating unnecessary cleanup work for compaction threads. Light-green boxes indicate objects overwritten with new data (operation sequence is identical to the prior figure). When new objects are inserted (their keys do not exist in the index) do we perform a bump allocation in the log head, and append the item. Only when such operations — insertion and deletion — occur frequently, will compaction be necessary. *Workloads that never insert or delete objects (e.g., after loading the initial working set) will not incur overheads due to compaction.* 57

29 Throughput comparison of Nibble and Nibble-2 on YCSB (uniform) with 240 threads on the HPE Superdome X (smaug-1). As the amount of available memory decreases, compaction costs draw performance down in Nibble. The dip at 50% is due to compaction threads triggered into polling in attempts to reduce fragmentation. Nibble-2 experiences zero compaction, as all PUT operations overwrite existing objects. YCSB does not delete or insert objects. 61

30 Comparison of Nibble and Nibble-2 for YCSB Uniform distribution accesses of 1 KiB objects. '50' means 50:50 ratio between PUT (update) and GET; '0' means a pure-PUT (update) workload configuration. Nibble-2 shows a slight drop in throughput for the pure-PUT workload, as it breaks the "local-write global-read" policy, updating objects wherever they happen to reside in memory; Nibble always updates by append to local memory. 62

31 Comparison of Nibble and Nibble-2 for YCSB Zipfian distribution accesses of 1 KiB objects. '50' means 50:50 ratio between PUT (update) and GET; '0' means a pure-PUT (update) workload configuration. 62

32 Latency for PUT between Nibble and Nibble-2 for a single thread running without contention at low utilization. Insert behavior for PUT shows no difference in performance, as both systems behave the same (append). Update for Nibble-2 shows roughly a 15% reduction in latency for a PUT of 1-KiB objects to local memory. . 63

33 Latency for PUT between Nibble and Nibble-2 for a single thread running without contention at low utilization. Insert behavior for PUT shows no difference in performance, as both systems behave the same (append). Update for Nibble-2 shows roughly a 25% reduction in latency for a PUT of 64-byte objects to local memory. . 63

34 Linear-address translation using 5-level paging. From Intel's white paper *5-Level Paging and 5-Level EPT*, document number 335252-002, revision 1.1 [12]. 66

35 Diagram of The Machine as a memory-centric system, reproduced from [99]. . . . 68

36 Figure illustrating how to access an alternate memory region in (1) that either (a) does not fit, or (b) conflicts with one or more existing regions (e.g., if the new region requires a specific base virtual address). (2a)-(2b) illustrate unmapping the conflicting regions; (3) if they cannot be unmapped (e.g., the application may have stored references into such regions, which we cannot track), then the new region cannot be mapped at the required location (e.g., otherwise pointers must be swizzled). 72

37 Cost of creating a mapping in a page table with 4-KiB pages, as a function of the size of the region. Measured on Linux. 74

38 Figure illustrating how sharing in-memory data among multiple processes with varying access rights to disjoint subsets of the data. An individual file is insufficient to express relationships such as these, and programmers must use system administration access policies on multiple files to accurately express and enforce them. 76

39 Example segment. Segments are a contiguous region of virtual memory, composed of physical memory pages, with a base address and a length. The segment shown contains a symbol table and heap allocator context that a process may use to create and manipulate memory-resident data structures, e.g. a linked list. 81

40 API in SpaceJMP for managing address spaces and memory segments. 81

41 Examples of virtual address space types and their uses by a process. 83

42 Process Alice creating a memory segment with shared data for other processes to attach to. 86

43 Process Bob attaching to the "shared" named address space and accessing the common data within. 86

44 Illustration of a virtual address space in DragonFly BSD, VM objects. The implementation of a segment and address space in SpaceJMP are simply wrappers around each. 89

45 Figure illustrating modifications made to the BSD process and thread structures. Green shapes are additions. A process maintains a set of references to attached address spaces, and each thread an additional reference to its active address space, if not currently in the primary. 90

46 Diagram to show the private, shared, reserved segments. 90

47 Latency to access one cache line within a random page between context switches – simply modifying the base address of the active translation page table in the processor, held in CR3. When CR3 is modified, the CPU will automatically (for x86-based processors) flush the TLB. With a small number of pages, the translation state will fit within the TLB, illustrating the direct impact of flushing the TLB on memory load latencies. As the working set size increases, we see the effect of the TLB capacity to translate for a random-access workload, where tagging exhibits diminished benefits. 95

48 Comparison of three designs to program large memories with GUPs M3. Update set sizes 16 and 64. 96

49 Rate of VAS switching and TLB misses for GUPS executed with SpaceJMP, averaged across 16 iterations. 97

SUMMARY

New and emerging memory technologies combined with enormous growths in data collection and mining within industry are giving rise to servers with massive pools of main memory — terabytes of memory, disaggregated bandwidth across tens of sockets, and hundreds of cores. But, these systems are proving difficult to program efficiently, posing scalability challenges for all layers in the software stack, specifically in managing in-memory data sets. Larger and longer-lived data sets managed by key-value stores require minimizing over- commitments of memory, but current designs trade off performance scalability and memory bloat. Furthermore, opaque operating system abstractions like virtual memory and ill-matched, non-portable interfaces used to manipulate them make the expression of semantic relationships between applications and their data difficult: sharing in-memory data sets requires careful control over internal address mappings, but mmap, ASLR, and friends remove this control.

To explore and address these challenges, this dissertation is composed of two pieces: (1) We introduce and compare a new design for key-value stores, a multi-head log-structured allocator whose design makes explicit use of a machine's configuration to support linear scalability of common read- and write-heavy access patterns. Our implementation of this design, called Nibble, is written in 4k lines of Rust. (2) Going beyond key-value stores, the second part of this dissertation introduces new general support within the operating system enabling applications to more explicitly manage and share pointer-based in-memory data: we introduce explicit control over address space allocation and layout by promoting address spaces as an explicit abstraction. Processes may associate with multiple address spaces, and threads may arbitrarily switch between them to access infinite data set sizes without encountering typical bottlenecks from legacy mmap interfaces. Our implementation of this design is in DragonFly BSD.

CHAPTER I

INTRODUCTION

Within the last decade, application data sets have grown to enormous sizes [35, 52, 70, 173] frequently exceeding the capacities of main-memory machines [83]. Numerous software systems, such as key-value stores [123, 115, 72, 152, 27, 143], databases [92, 120], file systems [165, 63], and operating systems [163, 96, 14], have thus introduced clever designs that balance the performance of main memory with the capacity provided via cheaper persistent storage, such as disks and SSDs [137].

In response to the continued growth of application data sets, and the demand for faster access to such data [134, 62, 143, 100, 86] computer manufacturers have introduced very large multi-socket machines with (now) terabytes of byte-addressable main memory. Keeping entire working sets in main memory provides fast access to data sets — at DRAM latencies, 100-400 nanoseconds for one cache line — even under random-access patterns; a design property that proves extremely advantageous for web applications [86], databases [101, 70], etc. In many domains, low latency is vital for revenue and customer retention [154, 86]. As analytics applications increasingly demand real-time or near real-time processing of large data sets, a new class of in-memory big data systems (e.g., key-value/object stores, in-memory databases and in-memory analytics engines) has arisen to store and process data in memory. The demand for in-memory processing of large data sets and advances in hardware such as newer more dense memory technologies [137] accelerate the emergence of large-memory systems with a large number of cores [8, 82, 7, 6, 11, 20]. Compared to distributed systems to scale computation, large single machines prove to be more effective, for example in graph computations [173, 121], where individual large data sets are accessed in parallel (or if it cannot be independently partitioned). Designs of software systems executing on such machines have increasingly incorporated concurrent algorithms and data structures [44, 116, 125, 68, 37] to achieve the multi-core performance available on such hardware.

Key-value stores are important in building large-scale applications, mapping arbitrary keys to locations in memory that hold objects (structureless, opaque memory blobs). Ignoring a machine's

underlying topology, a key-value store may initialize all memory on a single socket; when supporting many threads, performance often plateaus due to insufficient inter-socket link bandwidth [95, 108]. And, common synchronization primitives, such as those which rely on atomic compare-and-swap to guard shared resources, quickly become bottlenecks [126, 127, 129, 59], using only a fraction of the total compute resources within such machines. Alternative methods for handling such scale, such as concurrent methods [162, 91, 84, 60], are necessary.

A further consideration within such data management layers concerns the effectiveness of memory allocation to ensure the use of available capacity is efficient — as low a ratio of used memory to allocated memory as possible. Just as in file systems or other persistent storage where data sets live for extended periods [106], fragmentation becomes a concern for main-memory data storage, as well [170, 151]. Often, key-value stores incorporate common general-purpose heap allocators or techniques derived from their designs [42, 152, 72, 123], given their excellent SMP scalability. As applications' use of the heap can often be described as regular – reusing a small number of specific object sizes, derived from structs or other memory objects [42] – the management, then, of *arbitrary data* leads to situations of high fragmentation. Such fragmentation, or "bloat", can result in as high as twice or three times as much memory used for a given data set [151], making very little of the immense capacity of large-memory machines available. Situations like these result in degenerate behaviors, where software swaps to storage mediums, or arbitrary failures.

As very large memory SMP machines are characterized by designs that resemble distributed systems where resources are disaggregated and have non-uniform characteristics [141, 108], the design of the machine more directly impacts higher-level software performance; having the operating system present resources entirely transparent presents limits in how far software systems can scale or program their data. Specifically, building large applications or runtimes that take advantage of long-lived, complex data sets is met by a lack of control from the operating system to manage these data sets, such as interfaces for virtual memory management [65, 138], or physical memory management [105] (memory interfaces in the operating system are the most heavily used, according to a recent survey of POSIX [30]). Data often takes on two forms: the in-memory representation, and the persistent (or on-disk) format, with additional code developed to convert between the two (e.g., Protobuf [16]). Given the size and complexity, it becomes increasingly beneficial to maintain

the in-memory representation, as serialization to and from alternative mediums can increasingly become costly [71] – a burden on recovery, maintenance, and persistence. This burden is directly imposed on software by the virtual memory interface presented by operating systems. The fact it remains a transparent "resource", and hides knowledge of where and how physical memory is provided to software presents a "death by a thousand cuts" that makes designs and performance of software managing large data sets impractical. For example, page faulting, page migration, and the translation granularities make it challenging for user-level software to determine how much or where their physical memory is located — information important for ensuring performance and scalability on large machines.

With a memory-centric focus on the design of applications [69, 46], where data is maintained as their in-memory representation, the sharing of large data sets becomes a burden without an effective means of controlling its access. For example, rebuilding an index from its persistent format takes time, but more demanding is the lack of control over how user-level software can set up access to such data: as virtual-to- physical memory translation is required, it remains a challenge with current low-level APIs to ensure that a specific range of addresses can be made available to map in a data set [46, 65]; if address ranges are unavailable, costly swizzling techniques, for example, must be used before such information can be used [168]. Libraries, heap regions, devices memory, etc. all require modifying the address space, where their base offsets are determined by disjoint subsystems within the kernel — address space layout randomization ("ASLR"), device drivers, and the state of the memory map at any point in time — making it difficult or impossible for user- level software to acquire address ranges necessary for accessing large data sets. This problem becomes exacerbated with increasing physical memory sizes, exceeding the translation mechanisms' capacity within commodity processors [77]. x86 processors use radix trees to implement virtual memory, and with four layers, can translate a maximum of 2^{48} bytes (256 terabytes) of memory; individual machines today already have 2^{46} bytes of physical DRAM (e.g., SGI UltraViolet machines) – the maximum addressable by current commodity processors. Data center machines based on disaggregated memory designs, such as HPE's "The Machine", will easily exceed this threshold.

Given the scope of this area of research, this dissertation specifically aims to address the following questions concerning the design and implementation of scalability bottlenecks within user-level and

system software on very large-memory SMP machines:

- What key-value store designs exist today, and how is their performance impacted by such machines? How well do they handle operation sequences that may lead to fragmented memory, thus consuming most of a machine's capacity?
- Log-structured systems, as used in key-value stores, allow for 'cleaning' memory to reclaim unused memory holes; are such designs amenable to scaling, and how do they compare to existing alternatives? What are the main overheads and challenges, and how can we overcome them?
- A disadvantage of log-structured systems is the need to "compact" memory. What can we do to reduce this overhead?
- What motivates the need for improving memory interfaces within operating systems today, and are challenges motivating this need specific only to such large machines?
- What does the future of these machines look like, and how relevant are the insights in this dissertation to them?

In exploring these questions, this dissertation discusses the design and evaluation of two systems: (1) a user-level, scalable, log-based key-value store, and (2) an alternative approach to memory management within an operating system where virtual memory is managed with explicit address space abstractions, as would be files.

1.1 Thesis Statement

Very-large-memory machines pose many challenges for runtime systems and system software to make the underlying platform scale and capacity available to applications: concurrent access and terabyte-size data sets become fragmented over time, and mechanisms for organizing memory access, such as memory allocation techniques underlying key-value stores or abstractions presented by operating systems, result in designs that do not scale to many cores, support high levels of concurrency, or make addressing enormous sizes of physical memory both cumbersome and inefficient. The thesis of this dissertation is:

To reduce memory bloat and attain high scalability on massive-memory SMP machines, user-level data management tools, such as key-value stores, must incorporate use of multi-head log allocation, concurrent indexes, and scalable synchronization for memory reclamation, and be built on top of operating systems that expose explicit abstractions for managing and switching between large virtual address spaces.

1.2 Contributions

The specific contributions made in this dissertation are as follows:

1. Characterization of challenges on large-memory extreme-SMP machines for key-value stores: limitations, programmability, scalability, and efficiency of current methods; and challenges of operating systems for supporting applications in general which work with large, rich in-memory data. Chapters 2, 3, 7.

2. Design of a scalable log-structured key-value store using concurrent data structures, state partitioning, use of scalable hardware primitives, called *Nibble*. Chapter 4.

3. A view into design parameters that influence efficiency of compaction in Nibble, and a design adaptation of Nibble meant to reduce compaction work itself via a relaxation of strict log-based behavior, *overwriting* objects on updates, instead of appending them, called *Nibble-2*. Chapter 6.

4. The design of a new operating system memory subsystem exposing virtual address spaces as a model for supporting memory-centric computing, where processes may allocate, compose, and arbitrarily switch between address spaces, called *SpaceJMP*. Chapter 8.

5. Implementation and empirical evaluation of Nibble and Nibble-2 in Rust on an HPE Superdome X machine with 240 CPU cores on terabyte-sized data sets, compared with alternative state-of-the-art scalable key-value stores. Chapters 4, 5, 6.

6. Implementation and empirical evaluation of SpaceJMP in DragonFly BSD on a half-terabyte parallel machine, and demonstration of its use for scalable micro-benchmarks. Chapters 9, 10.

7. Public release of source code for Nibble.

CHAPTER II

BACKGROUND AND MOTIVATION

This chapter provides background and context of the topic of this dissertation, as well as to give definition to the types of machines evaluated.

2.1 The Advent of Large Memory Multi-core Machines

Beyond the evolution into multi-core processors from the prior decade, a large growth in data collected by corporations of their customers, financial transactions, and social behavior online together with very tight budgets on processing or analyzing such data has led to machines with enormous memory sizes and parallelism. Different from other high-performance machinery – specialized for high-performance computing on custom chips and networks, such as China's Sunway [22], IBM's Sequoia [19], and Fujitsu's Kcomputer [17] – many more machines today are built from (mostly or entirely) commodity equipment and run standard operating systems, such as HPE's Superdome X [82, 8], or SGI's UltraViolet products [20] Their cost and familiarity enables a large community to take advantage of their capabilities for database processing, analytics of customer purchases and preferences, as well as leveraging the availability of large pools of memory for quickly serving information backing major websites. Often, the argument made in favor of such machines is the extremely close proximity of enormous single data sets to a large pool of computing elements, simplifying programming designs by avoiding networking layers and communication protocols.

It becomes important to understand how to achieve high performance on larger machines, and avoid pitfalls when programming for smaller-scale machines. Typical data centers deploy the most cost-effective machine configurations, with dual-socket multi-core processors, at the scale of thousands of such machines connected over TCP/IP. While software executing on these machines increasingly benefit from parallelism — around 12-24 processing cores — solutions for leveraging many cores by individual applications or runtimes face bottlenecks on terabyte-class machines where many tens to hudreds of processor cores are available [43, 133]. This quantity of hardware parallelism arises not just from an increasing core count on processors – a trend which is not halting, as lower

Table 1: Example single-system large-memory machines. Total Memory bandwidth is a sum of bandwidth to local memory across sockets. *64 TiB is a limit of the CPU, not the platform, able to address only 2^{46} bytes of physical memory.

Machine	Sockets	Cores	Memory Size	Total Memory BW
SGI UV 3000	256	4096	64 TiB*	1 TiBps
SGI UV 300	64	1536	64 TiB*	-
HPE Superdome X	16	384	48 TiB	1 TiBps
Dell R930	4	96	12 TiB	240 GiBps
Cavium Thunder X2	2	96	1 TiB	-

Table 2: Example large-memory networked machines. “The Machine”is a datacenter system, and Summit an HPC machine.

		Per Node		System-wide	
Machine	CPU	DRAM	NVM	Memory Size	Network
HPE “The Machine”	ARM	512 GiB+	2-4 TiB	1 PiB+	Gen-Z
IBM Summit	POWER9	512 GiB+	800 GiB	2.6 PiB	InfiniBand

power processors such as ARM-based CPUs are also entering the mainstream server market – but also due to the greater quantity of processing sockets deployed in these machine configurations. Current SGI UV machines, for example, are able to scale up to 256 Intel Xeon processor sockets [20].

We noticed that certain design techniques leveraging simple synchronization primitives, while sufficiently scalable on such smaller machines, quickly became sources of contention once thread counts increased further. In the work within this dissertation, we observed that spin locks, or use of atomic instructions to modify counters, for example, are simply insufficient to guard shared resources when scheduling highly parallel applications. These primitives are often found lurking within the depths of various software libraries that provide concurrent services; one must use more scalable primitives that one might otherwise have considered overkill, or which have higher per-operation latencies (e.g., MCS locks [129]). Two specific examples we stumbled upon in the implementation of our key- value store – Nibble – were in Rust [135] libraries specifically developed for multi-threaded software:

- The crate `Crossbeam` [136], containing data structures for concurrent programming. The use of epochs is implemented to allow for asynchronous deallocation of memory (i.e., mark a time stamp on the object, and later check for a quiescent period after which to release it). Data structures such as a multi- consumer multi-producer lock-free Michael-Scott queue [132]

use this mechanism. The problem therein is that the epoch mechanism within the library is shared among all instances of queues used in software, even among disjoint sets of threads working on independent queues. The epoch itself is an atomic counter: each time the memory backing an object removed from a queue is released, the counter is incremented. Obviously with increasing numbers of threads, this method quickly becomes a bottleneck, unbeknown to the programmer, when seemingly no threads are actually sharing any data.

- Use of reader-writer locks (either part of `std::sync`, or in the crate `parking_lot`, the latter being used in Nibble). While more trivial of an example (and perhaps not specifically a result of using Rust), reader-writer locks are used in scenarios where the programmer assumes some resource is more frequently accessed by many readers and at most one or few mutators. When a mutator wishes to lock the resource, it must (a) prevent new readers from entering and (b) not itself use the resource until all existing readers or a mutator have relinquished it. To accomplish the latter, an atomic reference counter is used to keep track of in-progress readers. Unfortunately, it is this mechanism in reader-writer locks which limits scalability: with increasing numbers of readers — even with no mutator attempts — read access is bottlenecked, as all threads entering must lock the shared cache line and either increment or decrement the counter, a cost that is greater across large off-chip network distances.

Our experiences from situations like the above have guided us to look for solutions where we minimize the need to actually synchronize operations unless it becomes absolutely necessary. Nibble uses a scalable implementation of an epoch we designed specifically for high-concurrency execution, and optimistic concurrency control within its index (both discussed in more detail in later chapters).

Just as in distributed systems where individual machines can become overloaded due to high request rates across its network, so can individual sockets within these machines. Off-chip network bandwidth is not increasing with time as the amount of hardware parallelism available (Table 1), leading to a growing gap in performance per thread to arbitrary memory. Understanding this is important, as simple workloads that have skewed characteristics, such as in key-value stores, can quickly create bottlenecks. Thus, where data is allocated to — which specific socket memory — can have a large impact on performance. Unlike prior work which uncovered discrepancies in bandwidths

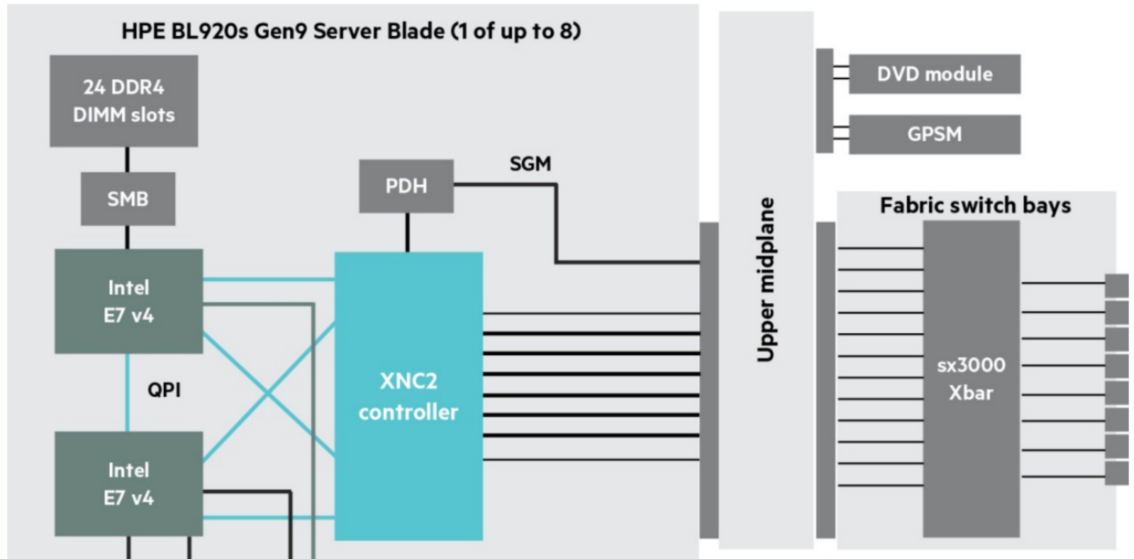


Figure 1: HPE Superdome X system design – a custom controller connects to each processor via QPI channels, establishing a large load/store domain to all of memory. Reproduced from [9].

and even directionality of such bandwidth between sockets on 8-socket AMD machines [108], machines like shown in Figure 1 aim to provide as uniform of memory bandwidth to remote memories across all sockets. While this helps avoid needing solutions in prior work to match application bandwidths to placement across each socket, large machines like these have much more hardware parallelism. What does this imply? Highly multi-threaded software can easily become bottlenecked by saturating a socket's off-chip bandwidth.

We illustrate this problem with a concurrent hash table [111], varying how many threads perform lookups on the data, shown in Figure 2. Allocating physical memory pages backing the hash table to (a) one socket (blue circles), (b) evenly across all sockets with lookups by threads to key only found in local memory (light-blue star), and (c) the same but lookups performed randomly across the entire memory (green square). If we look closely at the behavior under 50 threads, we see all three methods scale well; with all data on a single socket, performance is highest. More threads shows no increase in performance when memory is placed on one socket – we saturated its cross-chip network. Such large machines are expected to be nearly or fully utilized, thus the discrepancies seen at 150+ cores becomes most important.

Thus, a challenge in managing memory requires understanding data placement, as well as data access, and ensuring the aggregate bandwidth available across all memories is balanced to ensure

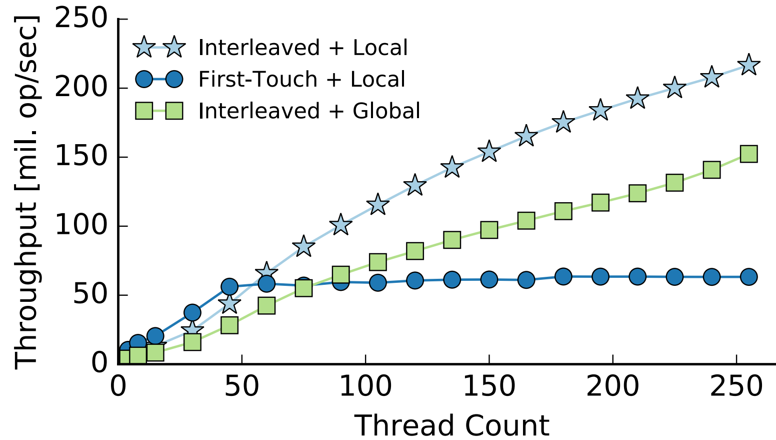


Figure 2: Figure showing the effect of data placement within a 16-socket machine on the performance of a concurrent implementation of a hash table (Cuckoo).

achieving high overall bandwidth to large in-memory data sets.

Looking at absolute performance in terms of the latency of individual operations on large machines, another characteristic which must be considered is the much larger overall cache line access latencies. On smaller 2-socket systems and a randomized access pattern, 50% of operations will access remote memory, but on larger machines, depending on how numerous their socket configurations are, will see nearly all accesses be to remote memories. For the system evaluated in this dissertation, there are 16 sockets, leading to $\frac{15}{16}$ or 94% of all operations which touch remote memory. A tool developed as part of this dissertation measures the individual unloaded cache line latency. On a 2-socket machine, remote memory accesses average around 180 nanosecond, and on an HPE Superdome X they average 320 nanoseconds – a 70% increase in latency.

Placement of memory thus must be carefully balanced with scaling operations.

Aside from available hardware parallelism, another aspect concerns the scope of memory sizes, and the capabilities of the processors and operating systems to effectively make this memory accessible to software. We will see an enormous growth in the capacities of memory systems in the future, not only in part due to alternative memory technologies such as NVM, Memristor, 3D XPoint, STT-RAM, etc. but also due to new processor connectivity networks, such as Gen-Z [5]. The latter, Gen-Z, is a memory fabric network technology touted to enable connecting many machines together with custom network processors that connect general CPUs to a unified remote memory.

Effectively, one can create enormous load-store domains, much larger than what current commodity processors are capable of addressing. Today's Intel and AMD processors, although "64-bit enabled", allow for 52 bits of memory addressing in their specifications. Actual implementations (processors purchasable today) are often limited to 46 bits of physical addressing and 48 bits of virtual memory. Such limits are imposed by the implementation of the virtual memory within the hardware, in the form of virtual-to-physical page translations: a fixed 512-child radix tree four levels deep provides translation to, 9 bits (= 512 entries) · 4 levels + 12 (bits in lowest entry, a 4 KiB page) = 48 bits of translation, or 256 TiB of memory. Without the ability to address large regions of memory, or where fragmentation of the address space itself prevents mapping in large regions, applications and system software must resort to frequent modifications of their page tables, which as we will see in later chapters, is an ill-performing workaround.

As of this writing, Intel has updated their processor manuals to include a 5th level in their virtual memory page tables. While there is precedence to do so (earlier 32-bit systems had introduced PAE to increase the address range by a few bits, to use more than 4 GiB of system memory), each additional level in the page table impacts overall performance, as an additional memory fetch is required in translation, as well as increases pressure within translation buffers to cache translations for application, buffers that are already limited in capacity.

Summary. Terabyte- and future petabyte-scale machines present different performance and programming characteristics to applications and system software that are not as visible in typical data center servers: memory sizes are larger, and so are the processor configurations. Total system memory bandwidth is available across a greater number of discrete processor sockets where bottlenecks may easily appear without careful placement of data, and memory access latencies are much higher to accommodate ever-increasing memory capacities. The latter encourages system software to take advantage of more scalable methods for designing concurrent mechanisms.

By presenting an overview of specific hardware challenges here, the intended idea is not to claim this dissertation presents a comprehensive analysis of their impact, or a set of solutions for overcoming them, but rather to convey a general mindset for programmers working towards achieving high performance at or near full capacity on these large machines. A solution to each aspect above is incorporated within the introduced systems in this dissertation. We dive into the design space

of key-value stores on these machines to explore how the above machine characteristics influence their implementations, and examine more closely what limitations exist within system software in designing and developing such services, specifically in virtual memory interfaces.

2.2 Software Challenges in Programming Large Systems

We follow with discussion on challenges seen within the software layers deployed on these machines, and briefly introduce the research ideas developed to explore them.

2.2.1 Key-Value Stores for Managing Large Datasets

For managing arbitrary data for applications that require very fast lookups, one may use for this purpose special data structures to organize it, such as hash tables or trees. "Arbitrary data" is meant here as data not limited to the memory required to enable the application to run, but objects representing web caches, media, graph information, arbitrary structs, or opaque binary blobs. Key-value stores make management of such arbitrary data easier, as the interfaces are simple, using keys. Implementations typically consist of two pieces: an index and a memory allocator, and on top of this is an interface to add, remove, or retrieve an individual object. Often used on machines with large quantities of memory serving cached data over a network, many recent systems have been developed (1) for use on individual machines to manage persistent data, such as with LevelDB [78], and (2) to exploit available hardware parallelism to maximize performance of individual machines, such as with MICA [116], cLSM [80], and others.

Given the idea behind a key-value store is to provide high-performance access to cached data (instead of resorting to costly database operations or similar), the main priority in their designs has been performance — scalability [80, 123], network efficiency [134, 62], and reducing latencies. Secondary to this is how well such a system manages the memory allocated to hold objects, how much exposure to internal or external fragmentation it presents, and whether it can do so well without sacrificing end-to-end performance. Many of the recent key-value store projects describe designs which introduce new techniques for scalable or concurrent processing — multiple threads [72], concurrent implementations for trees [123], hash tables [143, 116], or partitioning state across cores [130, 36]. In this dissertation, we flip this perspective, and argue that *better use of available memory capacity must be the main priority* (i.e., reducing memory bloat), and that *devising scalable*

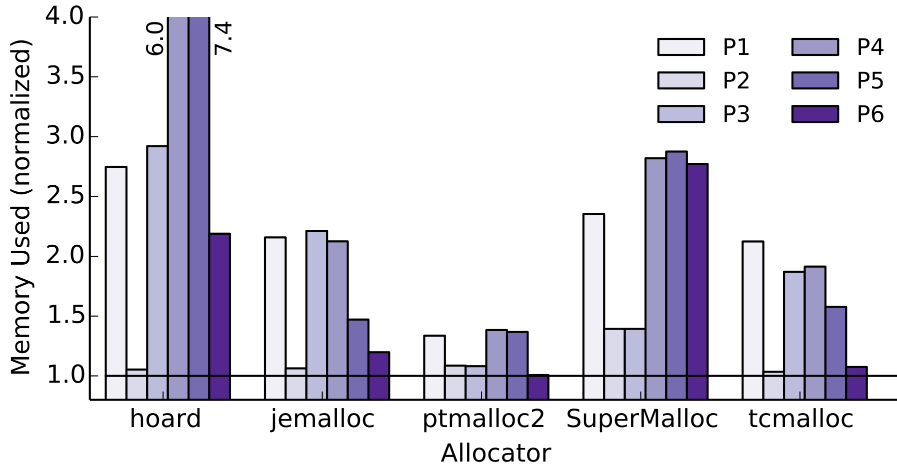


Figure 3: Figure showing a measure of memory bloat in common general-purpose heap allocators. P1-P6 are object sizes. P1-P6 are explained in Figure 14b.

methods thereof become the focus of state-of-the-art designs. We next briefly discuss reasons behind this perspective.

Nearly every kind of application uses some means of heap memory management, and is typically a general-purpose allocator. Examples of these include jemalloc [68], tcmalloc [2], ptmalloc [1], as well as various custom implementations that use slab allocation for specific optimizations. Arguably, general-purpose heap allocators perform well, applying common techniques such as thread-local allocator pools and some form of slab allocation [42] to minimize fragmentation. Given their generality, many key-value stores rely on their performance in managing the heap, for backing application objects (e.g., Redis with jemalloc). We observe that large-scale, dynamic in-memory data sets will be long-lived; such precedence has been established with other systems such as RAMCloud [143], which use DRAM as the primary reference location for structured and unstructured persistent data, or TableFS [148] which uses the persistent key-value store LevelDB [78] to implement a file system. Long-lived data sets have greater exposure to allocation patterns that, over time and across numerous clients, may create fragmentation within the allocators [106]. File system implementations are keenly aware of this, as de-fragmentation tools have been around for a long time (e.g., in Windows, or the research project LFS — Log-structured File System [149]).

We demonstrate how severe this can become by measuring the memory usage of multiple general-purpose heap allocators. Allocating ca. 10 GiB of objects of a fixed size, randomly releasing 90%

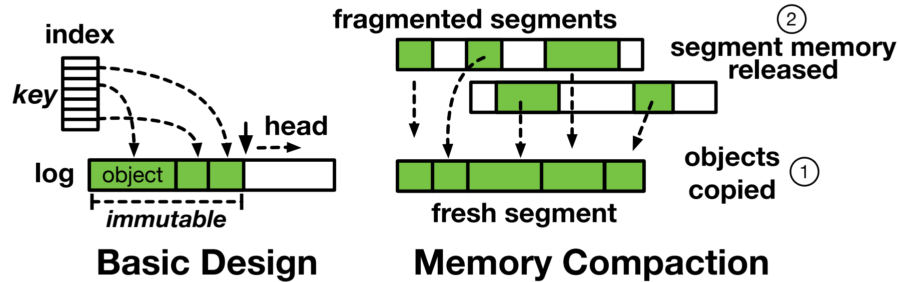


Figure 4: Figure illustrating a high-level view of log-structured allocation.

of them, then allocating objects of a second size until we have again reached 10 GiB of aggregate objects, we then record the resulting resident set size (RSS) reported by Linux — total resident memory in DRAM used by the test program. Shown in Figure 3 is the ratio of RSS over the aggregate size of objects actually allocated. One can immediately see that memory bloat can be extreme in some cases, as is for Hoard with up to 6-7x overhead, but otherwise in our measurements is around 2-3x across allocators. The allocation pattern itself does not dictate if or how much bloat results, as P2-P3 shows very large differences for Hoard, jemalloc, and tcmalloc, but not so for ptmalloc or SuperMalloc [104]. Bloat of 2x means half of actual memory is unavailable. On machines with terabytes of memory, this overhead implies also that terabytes of memory are wasted — a non-negligible amount. The impact of memory bloat can be significant. With low memory availability, the operating system may trigger swapping, or if the key-value store acts as an object cache, it may begin unnecessarily evicting objects from memory. Both are behaviors that result in degenerate situations and destroy performance.

Key-value stores have explored log-structured designs, as well. A log in this sense contains a head offset into memory, and all new objects and object updates are appended in whole to a 'head' offset (see Figure 4). Deleted objects create holes, and eventually require compaction by relocating live entries to reclaim empty areas of memory. The most prevalent design is a log-structured merge tree [142], which is a two-piece tree, one in memory and another on disk, meant to balance the large performance gap between main memory and storage. New items are appended, and when a specific child node becomes full, it is rolled into a lower layer, eventually reaching disk. The ability to rearrange memory allows for very efficient use of memory capacity (and thus also disk capacity). Another approach, the design used within Nibble, has been demonstrated in RAMCloud [151, 150]

and does not use a merge tree. Instead, all of memory is broken into pieces called 'segments'; full segments are closed and made immutable, and compacted segments are eventually released to be reused for the head offset. Compared to heap allocators, log-structured designs allow for relocating objects – a characteristic that gains resistance to allocation patterns that may lead to fragmentation – and thus exhibit very low memory overheads. We describe in more detail the design and functioning of log-structured allocation in key-value stores in Chapter 4.

The most significant cost in a log-structured design is in the compaction process. Each time an object is updated, or erased, work is created for compaction threads, which involves significant amounts of memory copying. Prior work, e.g., RAMCloud and subsequent research to LFS [124], have examined methods for appropriate selection of fragmented segments to compact in order to reduce extraneous work (such as avoiding repeatedly compacting segments that frequently become fragmented). Despite this trade-off, such designs are the most advantageous in managing long-lived large data sets in memory due to their memory efficiency.

On highly parallel machines, performance bottlenecks become apparent within each component of common log-structured designs. Scaling log-structured key-value stores is thus the focal point of the first part of this dissertation. In general designs, there exists a single log head, multiple compaction threads, one index, and one segment allocator; no awareness of the machine's memory topology exists. Each of these components becomes a scalability bottleneck, and requires incorporating an understanding of the aforementioned large-system characteristics into new designs in order to scale.

In further investigating the log-structured behavior, we ask whether requiring append for all object mutations, and maintaining immutable segments – a strict log-structured nature – is always necessary for storing volatile data sets and combating fragmented states. This is the subject of Chapter 6, where we explore disabling append and further study its impact on performance using our key-value store Nibble. We additionally explore alternative heuristics for improving the process of compaction when we are absolutely unable to avoid it, e.g., in scenarios where the storage system experiences high 'churn' — objects are frequently inserted and erased.

Next, we look at a broader perspective of efficient general-purpose programming on such large machines: we dive one layer down into the core system software that enables applications and services beyond just key-value stores to be implemented and run, and what challenges exist at

this level that are introduced with large physical memories, or in implementing memory-centric semantics for interacting with data sets — keeping pointer-based data in persistent memory instead of serializing them to files, or sharing such data among multiple processes.

2.2.2 Virtual Memory Interfaces in the Operating System are Limiting

Current data sharing semantics, within modern system software and also in recent research proposals, is based on the de-facto UNIX philosophy that everything is a file: applications create files in a common name space, assign permission to the entire file based on administrator-assigned users, groups, etc., persist relevant data to such files by serializing it to a linearized format, with subsequent de-serialization methods returning it to a usable in-memory representation. Shared memory semantics are similar (e.g., SysV or POSIX shared memory interfaces), but require that participating processes are either able to map memory to exactly the same base addresses in order to reuse pointers (which is not guaranteed, as we discuss later in Chapter 7, where even the semantics for accomplishing this are not portable across operating systems), or must implement alternative offset-based semantics over the data to work with unequal base addresses. Alternative methods for sharing data between processes within a machine require the use of UNIX domain sockets, which forces endless data serialization, buffer copying, system calls, and involvement with the operating system thread scheduler to ensure both end points of a pipe are attended to when data is ready to be moved. With the future availability of new memory fabrics, such as Gen-Z, and dense persistent memory technologies, applications and services are poised to change how they interface with memory and share information. Data accesses will be expressed in terms of memory-based abstractions, in contrast to the current use of files: being able to allocate physical memory directly, rearrange the virtual address space to accommodate data structures, to more easily share memory regions among processes, and to allow for finer-grained access permissions based on underlying page tables instead of with a system's administrator-defined users and/or groups.

We review example scenarios to demonstrate the specific focus in this dissertation. The primary cause of the following challenges is a result of virtual memory being a transparent resource managed by the operating system, with weak controls over mapping, permissions, and physical memory allocation given to user-level software. While the discussion here remains high-level, we go into

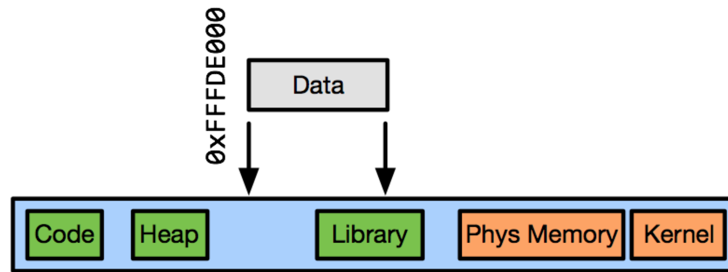


Figure 5: Data region cannot be mapped into the current address space because of a conflict with an existing library mapping.

more detail in Chapter 7 to elaborate on the specific details.

There is no recourse for applications to map large regions of physical memory, as illustrated in Figure 5, when:

- an address space either itself is too small;
  - due to insufficient translation bits supported by the processor as discussed earlier, or,
  - limits on available address space imposed by the operating system.
- an address space becomes fragmented and an available range of virtual memory cannot be established

The result is that processes must take alternate actions: (1) break their data into separate mapped regions, in an attempt to make all data accessible; (2) or, if possible, unmap existing regions from the address space to make room, such as unnecessary libraries, heap areas, etc.; (3) or, map in the data and *swizzle* all contained pointers to reflect the new base address of the mapping. None of these makes for an appropriate solution, and at best they are temporary and broken workarounds.

Accessing a large pointer-based data structure, for example, such as a search tree, would no longer function if broken up and mapped into separate regions. Memory in each region would need its pointers rewritten (swizzled) to reflect the alternate base address used that contains them. The second option – dynamically remapping regions based on the application's access patterns – imposes a very high runtime cost: mapping requires editing the underlying page table, flushing TLBs (on many-socket systems, this requires broadcasting interprocessor interrupts to do so), and installing new mappings.

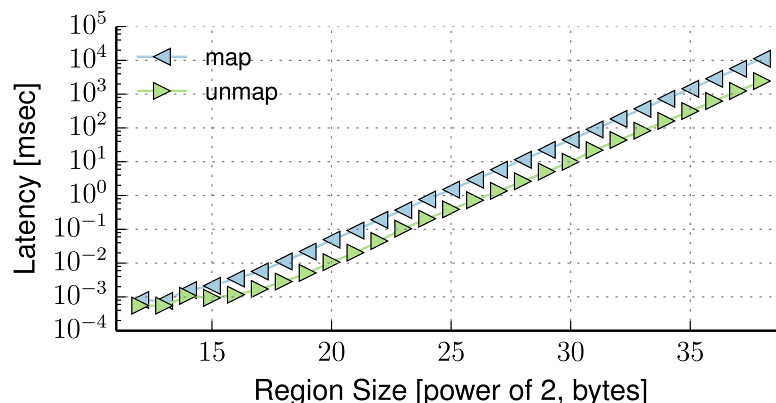


Figure 6: Cost of mapping virtual memory using 4 KiB page sizes to preallocated physical memory. Costs are directly a result of page table modifications.

How costly can repeated mapping become? Figure 6 illustrates costs associated with updating page tables to reflect new mappings onto existing physical memory. 32 GiB regions already consume over one second of time; going up to terabyte-sized regions would incur unacceptable latencies. This cost arises from page-table modifications (for large regions), and from system call and TLB flushing costs (for smaller regions). On machines with a large number of cores, the process of TLB shoot downs can itself be a challenge to implement well, as was

Managing memory holding shared data structures (which often live beyond the lifetime of the creating processes) without using a storage medium requires doing so via memory-based file systems: data is serialized to files, and when needed are mapped in and converted to an in-memory representation. Ideally, data structures would be stored directly with no serialization or transformation, but we are not given strong controls over the ability to specify the layout of a process' address space to ensure we can obtain the necessary base addresses to use the data structures. In addition to this, the transparency of address spaces makes physical memory allocation guess-work, as the interfaces are coarse and fragmented: a process cannot allocate physical memory without a virtual memory map (unless file APIs are used), and even then explicit page faulting must be programmed into software to force the operating system to locate free memory (where it is allocated from also depends on complex memory binding policies and heuristics).

We explore these challenges within the system software executing on these large machines via a new memory management design within the operating system called **SpaceJMP**, based on the

DragonFly BSD operating system [14]. Our specific contributions are new memory abstractions and the ability to directly use and manipulate address spaces as first-class abstractions in the operating system: processes can explicitly create and compose multiple address spaces, and arbitrarily switch between them at runtime to modify their views to physical memory without resorting to dynamically re-mapping memory. This design allows applications to store in-memory data structures directly into memory regions without dealing with expensive workarounds when a region of memory is partially or fully occupied by existing mappings; processes simple create a new address space, map in the physical memory at the offsets they require, and switch into the address space to access their data.

2.2.3 Summary

Performance of data management on massive-memory machines has become the primary focus for applications. Focusing on one high-level tool for data management – key-value stores – we argue that current and past designs have prioritized multi-core scalability or performance-oriented optimizations for use of network or storage devices, over a 'capacity-efficiency-first' approach, and optimizing these. Our goal is to flip the trend of current system design requirements, prioritizing instead memory capacity efficiency, and to introduce designs that present scalable alternatives to log-structured systems.

For supporting the wide range of software that will run on these machines, we briefly highlighted constraints of working with fully in-memory data structures based on pointers, and the control afforded to applications by the operating system. Lack of control over address space layout, which is essential for working with and sharing pointer-rich data, motivates our second part of the dissertation, to design a new memory subsystem that gives applications the ability to program infinite physical memory spaces, and to make it easier to work with and share data structures without resorting to costly workarounds, such as pointer swizzling, use of file abstractions, or costly data partitioning.

CHAPTER III

SCALABLE MEMORY EFFICIENT KEY-VALUE STORES

In this chapter we motivate the first component introduced in this dissertation — designing a concurrent log-structured key-value store for parallel large-memory machines, and present designs used by alternative state-of-the-art key-value stores as they pertain to scaling and efficient use of memory capacity. We begin with an overview of the motivation behind this work, discussing memory fragmentation as the main motivation of this part of the dissertation, followed by an approach to leverage log-structured systems for scalable data management.

3.1 Background and Motivation

Memory-Centric Platforms. Recent shifts in the design of platforms towards "memory-centric" computing [69] are characterized by enormous pools of byte-addressable memory shared by hundreds to thousands of concurrent processors. Implications on the designs of applications and runtimes (and even compilers [95]) are to incorporate more explicit awareness about the underlying platform's performance characteristics. Capturing thread access patterns, available memory bandwidth paths and bottlenecks becomes increasingly important for these platforms [58, 146].

The level of concurrency expected on such systems changes the designs of data structures to handle lower tolerances on performance. For example, maintaining a large global data structure across large arrays of memory and high levels of concurrency requires careful use of hardware atomics across processors [84, 59] to design data structures to scale.

Memory Fragmentation. Long-lived data sets face the challenge of becoming fragmented over time [106]. We see this characteristic in file systems [159] where allocation patterns may result in memory holes throughout storage, leading to wasted space, or decreased performance [56]. In-memory systems face similar challenges where allocation patterns are subject to client workloads characteristics, such as seen in object- or key-value stores. Memcached, for example, uses slab allocation within pools of memory [140] to attempt to reduce memory waste, and Redis uses a standard heap allocator – jemalloc [68] – which similarly uses forms of slab allocations called runs.

However, in-memory storage methods that are unable to relocate memory objects remain at-risk to fragmentation [170].

We measured this effect on five separate general-purpose heap allocators, shown earlier in Figure 3, using simple allocation patterns and observe up to 2-3x additional memory is consumed. The degree of memory bloat is unpredictable, as it varies on the underlying design, and on the pattern.

Ill-chosen heuristics can lead to large bloat in memory consumption, often as much as 3× or more [151]. With a memory overhead of 2×, the available memory is reduced to 50%, leading to greater costs to over-provision. But, for systems which are at their maximum physical capacity, applications experience performance drops, or outright termination. The same fragmentation issues also arise in higher level memory management systems and object stores [116].

With data sets growing larger – into the TiB or PiB ranges – the longevity of data sets will become a primary concern. Sheer sizes of memory alone create pressure to keep data sets alive as long as possible: regenerating or reloading memory from other sources becomes inconvenient. Data storage methods are increasingly used to manage long-lived data sets, making fragmentation management a first-class priority.

Log-Structured Systems. In-memory systems such as object stores have found benefit from log-structured allocation[1] with origins from earlier research file systems [149]. In such designs (the earlier Figure 4), memory is arranged linearly (logically) with an offset referred to as the *head*; modifications are copied in whole to the head, and the head is incremented. Allocations are fast, as the location of the head already points to a free area, and bumping it is a few instructions. Memory "behind" the head is immutable to clients; background workers typically rearrange this memory to reclaim holes for the head. Their work proceeds at the granularity of *segments*, which are typically contiguous regions of memory of some size. When the head reaches the end of its current segment, it is *rolled over* to a new, unused, segment. Empty segments are provided by these background threads, and released to some free pool of segments. While more intricate of a design, it enables resistance to fragmented memory at the cost of not allowing updating objects in-place.

Key-value store designs match well with log-structured allocation: they present an interface to clients where objects have a key, value, and miscellaneous meta-data, where access to objects is

[1]Not to be mistaken for log-structured *merge trees*.

often performed via copying them in their entirety. The indirection between keys and their memory location is already abstracted by the interface and the use of an index (typically a hash table, or a tree). Worker threads can rearrange objects, but must be sure that their work is consistent with the clients' view into the system, typically guarded by the index (e.g., an object exists if the index contains an entry, and the latest version must be seen by all).

Log-structured systems have demonstrated practical benefits in main-memory object stores, such as in RAMCloud [151] for persistent data or in MICA [116] for caching, both serving within high-performance environments. Much of the efficiency within a log-structured system comes from the careful choice of heuristics – picking low-activity segments for cleaning [149], or the segment size [124] in their impact on cleaning efficiency or disk bandwidth utilization. As platforms evolve to become increasingly parallel, designs affecting efficiency shift to other components of the software.

3.2 Scaling Log-Structured Systems

Log-structured systems can efficiently address memory fragmentation, but scalability remains a challenge. Contention on centralized components such as a shared log and index structures, which are inherent in traditional log-structured design, as well as ignoring the underlying hardware features such as NUMA, can cause severe performance bottleneck when deployed on large scale systems with hundreds of cores and massive memories.

The design of a *scalable, in-memory* log-structured system hence necessitates operations that impose little to no synchronization constraints between operations, but doing so poses many challenges, as implementing a log-structured design requires making careful modifications in all aspects of the system. In some prior work to scale log-structured merge trees, the primary bottlenecks are in the interactions with storage devices; solutions for scale encompass designs which focus heavily on moving data to disk [155]. In our work, we focus solely on DRAM-based storage where latency constraints are much tighter.

3.2.1 The Index - Enabling Concurrent Operations

As every operation – GET, PUT, DEL– requires accessing the index, the first challenge is enabling concurrent lookup operations to be both low latency, and scalable. With DRAM acting as the main source for data, latencies to information become constrained under even tighter bounds. Data

structure design becomes more challenging for concurrent scalability [60]; even with read-centric workloads (common in data-serving environments [29]), concurrent hash table designs, for example, that use mutexes to guard each bucket can scale poorly, as each lookup must acquire the lock before reading the bucket. Scalable lock designs, such as the ticket or MCS locks, are best applied to common mutable resources experiencing high demand, such as in highly skewed write workloads. Fully concurrent data structure designs are often complex to design and implement. Lastly, scalability on larger platforms is sensitive to the distribution of memory across processor sockets; the bandwidth on processor chip interconnects can quickly become a scalability bottleneck, even for simple index data structures, due to the large number of CPUs present.

3.2.2 Supporting Scalable Asynchronous Compaction

Another challenge is the coordination of activities between client and background compaction operations, as both sides will be concurrently mutating the system state. The mechanisms to relocate an object within the log require ensuring the location of an object remains consistent in the index, and that observing partial state is not possible.

(1) We may associate reader-writer locks with segments: compaction threads would acquire a writer lock on a segment it wanted to compact, and client operations would only read from a segment if the writer lock was not currently held. Such a design simplifies consistency between threads, but limits performance; locks will always be in the critical path (which become costly compared to reading small objects), read operations will stall during periods of high compaction activity, and may be blocked entirely for data recently placed into head segments (until they roll over to a new segment).

(2) An asynchronous design, used in RAMCloud based on ideas from the earlier K42 operating system [34], uses the notion of *epochs* to record the version of an object as its state changes over time. Lookup operations and compaction activities can thus proceed in parallel; segments that are ready to be recycled are placed into a holding queue until any references into them, held by in-flight operations, are released. The current design in RAMCloud uses a single epoch value, implemented as an atomic unsigned 64-bit counter, to *tag* segments that have been compacted, as well as new operations that enter the system, placing descriptors of the latter into "in-flight operation" queues

together with the value of the epoch at the time an operation began. When a waiting segment's time stamp is lower than any existing concurrent operation, it may be immediately recycled into the free pool.

Keeping track of in-flight operations can pose challenges. One design is to keep track of each operation using descriptors, inserting them into a data structure once their execution has started, and removing them upon completion. If a global queue is maintained (used by RAMCloud), a single mutex guard will quickly become a bottleneck; with queues per thread, compaction threads will need to iterate over each such queue. Furthermore, with main memory as the primary backing store, throughputs will be much higher, creating greater load on the data structures used to maintain these in-flight operations.

3.2.3 Prioritizing High, Sustained Write Throughputs

Log-structured allocation methods often design for a single log head. In write-intensive scenarios, it can become a bottleneck, especially with hundreds of concurrent threads where disk or network latencies are no longer in the critical path. One design strategy is to batch writes before they appear in the log. Doing so is undesirable, as it delays when objects become visible to other threads. On the other hand, leveraging distribution of logs gains advantages in parallelism, such as in database systems when combined with NVM [167]. However, coordination is still required among each log when committing transactions; in log- structured systems the use of distribution may cause compaction work to move data across the entire system, increasing contention for shared interconnects.

In key-value stores, a similar technique – partitioning – has shown some benefit, especially for alternative multi-core platforms like the Tilera [36], or to better provision for last-level caches [130], albeit for smaller working set sizes. In recent network-based key-values stores such as MICA, partitioning shows great promise where requests can be routed to specific cores for processing, avoiding shared lock accesses; in fully shared-memory systems, requests may come from any core in the system, making shared lock access unavoidable.

Summary. With the push towards using main memory as the primary storage location for information, latency tolerances become much lower. Existing techniques for balancing the asynchrony of operations in log-structured stores are proving difficult to scale. Our work introduces new designs

around the components of log- structured allocation that enable continued scalability on larger shared-memory platforms, discussed next.

CHAPTER IV

NIBBLE - A CONCURRENT LOG-STRUCTURED KEY-VALUE STORE

Nibble is a scalable log-structured key-value store intended for main-memory object allocation in highly concurrent and shared environments, maintaining low memory overhead. In this section, we review the design of individual components and how they combine to form the overall system, and its implementation in Rust [135].

4.1 System Overview

The basic design of Nibble consists of three key pieces (Figure 7):

- **Optimistically-concurrent index**. A partitioned, resizable hash table for mapping application keys to virtual memory locations — it reduces contention for highly concurrent operations, and allows memory compaction to run concurrently with application threads.
- **Distributed hardware-based epochs.** Thread-private epochs accessible from a shared table enable full concurrency be- tween operations and with background compaction to quickly identify quiescent periods for releasing memory.
- **Partitioned multi-head logs.** Per-socket log-structured memory managers with per-core log heads and socket-isolated compaction support concurrent writes and concurrent allocation of memory.

Collectively, these three components enable Nibble to scale on extreme-SMP platforms and to support high capacity use for dynamic workloads. We continue with a summary of the key memory abstractions in Nibble, and then describe each of the above components in greater detail.

4.2 Overview - Index, Objects, and Segments.

In Nibble, we organize memory into indivisible contiguous chunks of memory called *blocks* (see Figure 8). Their length is a multiple of the size of a virtual memory page, each represented by a structure that encodes its starting virtual address, length, and segment information if it has been

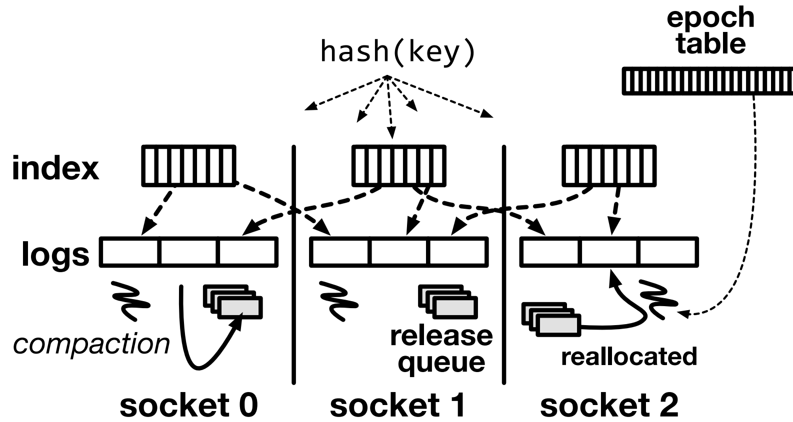


Figure 7: High-level design of Nibble. A global index is partitioned across sockets, pointing to objects anywhere within the system. Logs, segments, and compaction activities are isolated to the cores and memory within each socket.

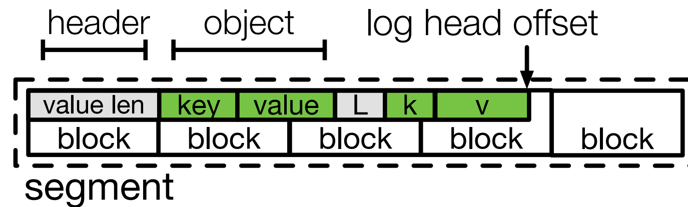


Figure 8: A segment – a container of blocks holding objects.

allocated to one. Segments are merely container objects, allowing the rest of the system to identify a specific collection of blocks, and apply operations on them in some linear order, such as writing new data. A set of segments forms a *log*, one per memory node or socket, and all segments and blocks of a log are managed by a single allocator.

Objects provided by application threads are placed next to each other within blocks, prefixed by a 32-bit header, encoding the length of the object, and a 64-bit key. To ensure optimal memory usage, we allow objects to be split across block boundaries within a segment; objects may not span segments. Storing the key within the segment allows us to determine whether this instance of the object is the live copy via a lookup in the *index*, a data structure that provides a mapping between object keys and their current locations.

A *head segment* is used for appending new objects, the exact location determined by the *head offset* (and then bumping it). When full, head segments are rolled over - closed and replaced with a new segment from free blocks. If the block pool size crosses below some threshold, threads will

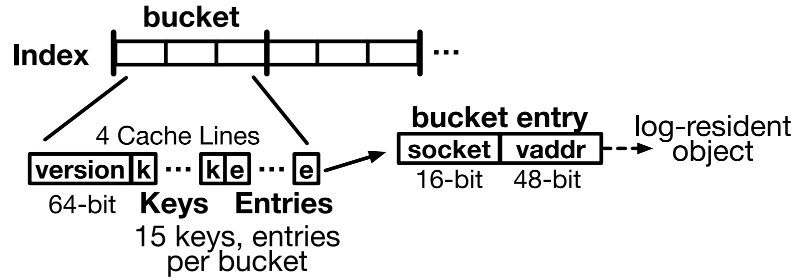


Figure 9: The hash table with optimistic concurrency. Buckets are in-lined and entries searched for via linear probing. Nibble allocates 15 entries and one version counter per bucket. Update operations will atomically increment the version to an odd value, blocking new traversals into the bucket, and forcing in-progress reads to restart. Once complete, the version is bumped again, allowing read operations to scan the bucket concurrently.

compact existing fragmented segments; the latter are held until references within them cease to exist, then their blocks are released back to the pool.

4.3 Optimistic Concurrency - The Index

As every operation requires interacting with the index, we must choose a design that allows enough concurrency between operations as possible. Given our target to run on extreme-SMP systems, we additionally must ensure there is su cient bandwidth for scaling lookups. The former we address with *optimistic concurrency*, and the latter by *distributing the entire index* across sockets in the system.

The index implements an open-addressing hash table, illustrated in Figure 9. We compose the index of multiple *subtables*, each resident within the memory of a specific socket. Lookups then use the key's hash to determine (i) the subtable to index, and (ii) the location within the subtable to extract the entry. The main goal is achieving extreme-SMP scalability. Thus, while distribution hurts per-operation latencies by accessing remote memory, each lookup is just 1-2 cache lines. Ensuring buckets do not cross cache line boundaries, we reduce false sharing between unrelated lookups.

Optimistic concurrency is implemented as follows. Each bucket has a 64-bit version. Mutators lock the bucket by atomically incrementing the version by one (to become odd), and again incrementing upon completion (to become even). When odd, subsequent operations will spin on the version until it is even, ensuring serialization of updates. Read operations will enter when the version is even, but will record the version to the thread stack. Upon extracting an entry, the version is read again; if both are equal, it means no update modified the bucket contents, otherwise the read restarts.

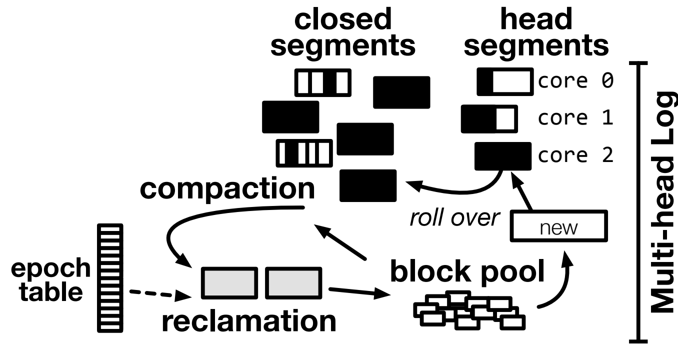


Figure 10: Decomposition of one multi-head log in Nibble.

On each socket are multiple subtables, affording the ability to perform concurrent resizes on subtables, where individual buckets may overflow, and to avoid having to over-provision memory for the index when the number of keys desired is unknown in advance. Resizing occurs by incrementally locking each bucket, then iterating over each key to compute its new location, moving it if required. To avoid encountering scalability bottlenecks within the operating system, we allocate 64 GiB of *virtual* address space per subtable, but only fault in the pages needed to hold the current length of buckets. Upon resizing, we do not invoke mmap (as it is serialized within the OS kernel [55]) and instead simply write to the virtual memory areas immediately following the existing buckets to cause physical page allocation (we bind the memory regions to sockets at the time the subtable is created, to avoid the "first-touch" policy, which would allocate pages potentially from other sockets).

4.4 Multi-Head Log Allocation

As shown earlier in Figure 4, traditional log-structured designs maintain a single head as a reference into unused memory for new updates, which is incrementally bumped with each append. Data becomes immutable within the log, and any subsequent operations to change the value will again append a whole copy. Doing so invalidates the prior area of memory, and requires reclaiming. Background-scheduled threads operate on such data, relocating objects to make recently released regions accessible to the head again.

The above design is sufficient for systems which interact with much slower hardware than DRAM, such as storage or networks, but will not scale to hundreds of cores: (1) contention to mutually lock the head would increase, and (2) without considering platform topology, many threads would quickly

saturate the bandwidth available on one socket.

Nibble introduces a hierarchical log design, shown in Figure 10, to solve both problems: on each memory socket we instantiate an instance of the log, and within each, numerous head segments. When client threads wish to append data in Nibble, a log instance is selected, then a specific head, before copying data and bumping the offset. Nibble holds to a *"write local read global"* policy where append operations are always performed to local memory nodes, to avoid high memory write costs and contention on the shared interconnect. Local- memory log-based allocation allows Nibble to gracefully handle skewed workloads by restricting contention to exist only within the index; threads will not attempt to overwrite the same location in memory for the object (explored in Section 5.4).

While simply instantiating multiple head segments per socket allows for some scale, on highly multi-core processors, the effect on performance of the specific method used to select a segment for appending becomes increasingly pronounced. How many heads should be allocated on a socket? How should we choose which head to append? We explore these questions in Section 5.1.

At high thread counts, the probability of eliminating contention with random selection becomes exceedingly small, as threads must choose unique segments with each operation. One way to overcome this is to assign each thread a head segment, however, we chose to avoid this option and instead instead assign a head segment to each CPU core. When an append operation is issued, we identify the specific core a thread is on, and use this information to lookup a given head segment. To ensure such an operation is as fast as possible, we directly leverage the hardware: on all modern x86 processors the rdtscp instruction (or rdpid on very recent implementations) returns both the socket and core ID within 30 cycles.[1]

Supporting Read Operations. When storing new objects in Nibble, we must be able to later identify the location of the object quickly. Appending to the log returns a virtual address of the object header, and this we store together with the ID of the specific log segment into our index (shown in Figure 9), we use the remaining 16 bits of our 64-bit entry to store this ID, as virtual addresses on x86 systems are a maximum of 48 bits). When executing read operations, if the head, key, or value lie within a block, we directly copy them out using the instructions rep movsb to minimize the

[1]Data returned are the user-defined bits in the `IA32_TSC_AUX` machine state register (MSR), initialized for every core by the OS upon first boot.

implementation of the copy routine.

As blocks within a segment may not be contiguous, for objects that span these boundaries, we must identify the subsequent block to copy from: we map the virtual address to a segment by (1) aligning the address to the next-lowest block boundary and using this to index into our vector of blocks, then (2) extract the segment information to locate the address of the next block to read from.

Scalable Asynchronous Compaction. Log-structured allocation requires objects never to be overwritten once committed to the log. Each mutation of an object creates a new copy at the head, making the prior instance stale (and thus contribute to memory bloat); such holes in the log may also arise from deleted objects. To reclaim unused memory such as this, immutable segments must be scanned, and any live objects copied to a fresh segment – a (private) head segment of a size just large enough to hold the live data to be moved. General designs of such background operations may naively consider any such segments across the entire system. Nibble restricts such threads to only operate on segments within a memory node, to both reduce cross-socket noise and ensure compaction operations remain as fast as possible.

Once segments have rolled over from being a head, they become immutable and are candidates for compaction. Segment selection in Nibble follows prior work [151] where segments are sorted by a cost-benefit ratio of their age and amount of reclaimable bytes:

$$goodness_{seg} = \frac{(1 - util_{seg}) \times age_{seg}}{1 + util_{seg}} \quad \text{where} \quad util_{seg} = \frac{live_{seg}}{length_{seg}}$$

Accurate live state information for each segment is maintained within a SegmentInfo table, described in Section 4.5, below. Essentially, each time an object is added to or invalidated from a segment, we atomically increment or decrement this size within this table.

Compaction threads in Nibble cache this metric (*goodness*) along with a record of their live size in order to determine when recomputation of this metric becomes required. This cache, with a reference to its associated segment, is kept within a vector; periodically (when available capacity reduces below 20%), compaction threads will iterate over each and if the cached live bytes metadata differs from the current, that segment's metric is updated. The vector is sorted in-place, and the segments with the highest cost-benefit ratio are selected for compaction.

To manage the sheer size of memory available on platforms studied in this work, we leverage

multiple compaction threads. When heads roll, each is placed round-robin into a queue destined for a specific thread, ensuring an even distribution of load among compaction threads at all times. Each thread greedily maintains a private set of candidates, allowing us to make further use of scale by having such threads perform candidate selection and sorting in parallel, instead of maintaining an enormous vector with all segments, using complex parallel algorithms to sort. Finally, as compaction threads are bound to a memory node, they are unaware of and do not synchronize with thread or memory of other sockets, keeping the amount of work per thread low.

The actual process of compaction is straight-forward. First, the sum of live bytes is calculated from the set of candidates, using the SegmentInfo table (discussed below), and then a new segment allocated. Should there not be sufficient blocks, a set of special blocks reserved just for compaction are allocated to perform the work. Each candidate segment is scanned and keys found are verified with the index: lookup failures or mappings which do not point to the current location indicate the object is stale, and is thus skipped over. A live object is individually locked using the index while the object is copied to a private head segment – a minimal, required synchronization with client threads. Nibble implements copying using the compact `rep movsb` instruction, which is one of the most efficient instructions for arbitrary copying of byte sequences, regardless of alignment (we do not attempt to align objects within the log, as this would complicate scanning). Once completed, the new segment is added to the set of candidates, and the old is pushed into a reclamation queue.

Segments in the reclamation queue must wait for *quiescent* periods, where no application thread may hold references into the segment. Such is required as we allow segments to be read concurrently by both compaction and client threads. Once this is determined (using *scalable epochs*, discussed below in Section 4.5), the segment is deconstructed, and its blocks released. If the new segment was constructed with blocks from the reserved pool, any released blocks obtained are first used to fill the reserve pool, before going back to the general allocator.

Summary. Scaling memory management on terabyte-class platforms in Nibble is accomplished by partitioning and distributing work, localizing operations to maintain the best possible performance. Nibble minimizes interference and overhead on application threads by leveraging multiple log heads in parallel, using efficient hardware routines in the critical path to ensure minimal contention when appending new objects.

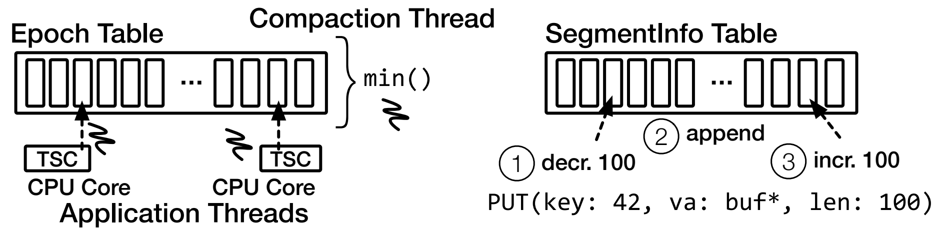


Figure 11: Illustration of synchronization between application threads recording their current epochs, and compaction threads determining minimum epoch for recycling segment memory (left), and updating the live bytes information for segments (right).

4.5 Distributed Hardware Epochs For Coordination

Two components in Nibble require synchronization between client threads and compaction logic: fast methods to determine the live size of a segment kept in the *SegmentInfo* table, and the set of active threads currently performing a GET or PUT operation, kept in an *Epoch Table* (Figure 11).

SegmentInfo Table. In order for compaction to determine the current set of live bytes within a segment, it would normally have to scan the segment, checking each key one-by-one against the index. This becomes increasingly prohibitive with greater numbers of segments, or smaller objects. Instead, we create a metadata table called the SegmentInfo table as a vector of cache-line-sized unsigned values that store the current sum of live objects within that segment; the index into this vector is the same as the index into the segment vector maintained by the log on a given socket. To distribute the work of maintaining the live bytes per segment, each application thread that performs a PUT or DEL will look up the associated segments and atomically increment or decrement the information directly.

We would not expect such operations to typically become bottlenecks: incrementing this value for a segment means a thread has chosen to append to that segment, but this process is made unique via the head selection method described earlier; decrementing a value means an object has become stale or was removed. On very small data sets where access patterns are highly localized, objects may be updated while the original is still located within a head segment (competing with the current appending thread). Larger data sets make this increasingly unlikely, rolling the segment sooner.

Scalable Epoch Tracking. As Nibble allows concurrent read-only access to segments, quiescent phases must be determined before releasing its memory, to avoid actively locking segments for each read, write, or compaction operation. One solution to this is to keep track of all active operations by

application threads using epoch and global participant lists: each thread that begins a PUT or GET will record a global epoch and add itself to a "participant set". The epoch will be incremented after each segment has finished being compacted, and a snapshot of the epoch stored with the segment in a reclamation queue. To release a segment, we would scan the participant set for the minimum epoch value; all waiting segments with epochs less than this minimum may be immediately released. PUT/GET operations that complete will remove themselves from this participant set. This approach exists in other systems, such as K42 [34], Keir Fraser's PhD thesis [73], and RAMCloud [150].

The way a global epoch is implemented in the above systems is via a single unsigned value, monotonically atomically incremented. On platforms with hundreds of active cores, such a design quickly becomes a bottleneck: adding and removing from the participant set requires global locking, and is too slow to support on a per-operation basis; and incrementing a global counter does not scale beyond one or two sockets. We evaluate this design in Section 5.1.

In Nibble, all threads register themselves upon first access to our API in an Epoch Table – as presented in Figure 11, left part. Each entry is a cache-line-sized unsigned value that records the current epoch. The set of participants is then determined by scanning this table for valid values (epoch > 0).

To avoid mutating a shared global value as the epoch, we leverage a known, global counter provided by the processor: the time stamp counter (TSC). As the TSC increments with each tick, there is no longer a need to manually increment the epoch. Prior work has used the TSC for similar global synchronization operations, such as ordering writes in update-heavy data structures and more recently in operating system data structures to scale concurrent accesses. In Nibble, prior to performing a GET or PUT, each thread will read the local core's TSC and store that value into its slot within the Epoch Table. Reading the TSC is fast – 30 cycles on an Intel Haswell processor. As each thread receives its own dedicated record for the epoch, no synchronization occurs between threads. When completing an operation, each thread will record a value that represents immediate quiescence, to indicate it has "left" the participant set of active operations (the value zero).

When compaction periodically checks the reclamation queue for segments, it will scan the Epoch Table to find the minimum. No atomic operations are required, and each read costs a cache line miss to pull in the value from the cache where an executing thread is currently residing.

Summary. By fully decentralizing the global data structures needed for synchronization between compaction and application threads, and enabling threads to act entirely on per-thread cache lines, we eliminate synchronization to support linearly scalable operations to hundreds of cores.

4.6 Hardware Assumptions

Nibble relies on a few mechanisms available in hardware. The first and most obvious is the use of the per-core timestamp counters. Each processor core contains a clock that increments at some rate. On receiving a RESET signal (propagated to all, e.g. upon reboot) the clock is initialized to zero. Modern processors all have some counter available for software to use, and is accessible to software to read even in restricted-privilege modes (e.g., ring 3 in x86 CPUs). A more critical assumption we make is that the counters are synchronized among all cores; that the RESET signal is received at the same time. Whether or not this is in fact precisely accurate remains an open question. Prior work [44, 166] has empirically studied timestamp values and determined that offsets are not noticeable or if offsets exist, one can measure them and add the maximum difference between clocks to each point used in software where comparisons are made (e.g., establishing global ordering).

We also rely on cache-coherent access to all of memory. It is not expected this feature will consistently remain available going forward [33, 6, 11], but rather exist in the form of coherence domains, e.g., among cores within a socket, but not across sockets. Shared memory data structure access and synchronization points would need to be modified to account for this: the global index, the epoch table, and the segment information table. Developing data structures in shared-memory across coherence domains is an open area of research; some techniques may utilize light-weight message passing [179, 144, 147].

CHAPTER V

NIBBLE - EVALUATION

The main focus of this work is to evaluate our methods for scaling log-based allocation as a memory management design within in- memory key-value stores. We compare our design in Nibble with both a competing log-based key-value store, RAMCloud, in two alternate configurations, and other systems that leverage varied memory allocation strategies (see Table 3), such as general-purpose heap allocators, or custom methods (slabs, segregated fits, etc.). Supporting this, we address the following questions:

- How does the performance of Nibble compare to competitive systems for both static and dynamically changing workloads?
- How much does memory pressure affect performance?
- How well does Nibble's log allocator avoid fragmentation?
- How well does Nibble handle high insertion workloads?
- What role does the use of multiple log heads and our distributed epoch play in overall performance?

The more general question we aim to address from the entire evaluation is, given the added complexity in Nibble to scale log- based allocation, in which situations is our design most useful?

Evaluation Platform. We performed all measurements on an HPE Superdome X computer with 16 cache-coherent Intel Xeon E7-2890 v2 CPUs, each with 15 physical cores @ 2.80 GHz, for a total of 240 cores. Each CPU addresses 768 GiB of DDR3 ECC DRAM, for a total of 12 TiB of system memory. SMT is enabled, but alternate threads were unallocated across all experiments; all default CPU prefetchers were enabled. The host operating system is Red Hat Enterprise Linux 7.3, kernel 3.10.0-327.28.3.

Table 3: Summary of compared systems.

System	Version	Description
Redis [152]	`3.2.8`	In-memory data structure store allocator: jemalloc [68]
RAMCloud [151]	`d802a969`	In-memory log-structured KVS
Masstree [123]	`15edde02`	Concurrent optimistic B^{+}-tree allocator: Streamflow [153]
MICA [116]	`b8e2b4ca`	Concurrent partitioned KVS allocator: custom segregated-fits [170]

Table 4: Summary of workloads.

Workload	Configuration(s)
Fragmentation benchmark	Benchmark derived from prior work [151]: Given a pair of sizes, allocate objects of first size, randomly delete 90%, allocate of second size until allocation fails (or until a specific capacity has been reached).
Dynamic *Postmark [98]*	2 million files, 500-4096 bytes, 32 million transactions, 10k subdirectories
As a trace: *per thread:*	18 million objects, 183 million operations 5.7 GiB working set size
Data-serving *YCSB [57]*	1 million 1 KiB objects, 100/95/50% `GET`, 100% `PUT`, uniform and zipfian

Our evaluation focuses on SMP performance across all systems, thus there are no networking components considered in this study.

Workloads. Table 4 summarizes the workloads used in our study. We evaluate each system with workloads which stress scalability, variable access patterns, the level of stress on the underlying memory management, and susceptibility to memory fragmentation. With computing becoming increasingly memory-centric, we value evaluating each system with a more dynamic workload at scale. Using Postmark [98] – a well-known file system benchmark – we expose each system to a working set with variable object sizes and a fairly high *churn* in objects – frequent insertion and deletion. In order to capture this behavior for key-value stores, we use TableFS [148] that converts I/O

operations into equivalent key-value store operations (it implements a file system with LevelDB [78] within a FUSE [15] mount point).

Using YCSB we measure overall system throughput in terms of operations per second. YCSB models typical data-serving environments, where data is often consumed at large scales. Note that with YCSB, the in-memory data set does not change in size, nor are objects ever created or deleted throughout our experiments, thus we label this as static workload behaviors.

Compared Systems. The compared systems are summarized in Table 3. We modified systems when necessary to isolate the core functionality and package them as a shared library for evaluation; where systems may act either as a cache (i.e., items may be evicted) or a store, we select the latter mode. We do not compare the advanced features of each system, using the simplest common subset – PUT, GET, and DEL operations. All persistent storage features of these systems (if present) were not enabled, and all evaluation was performed with client workloads executing on the same machine.

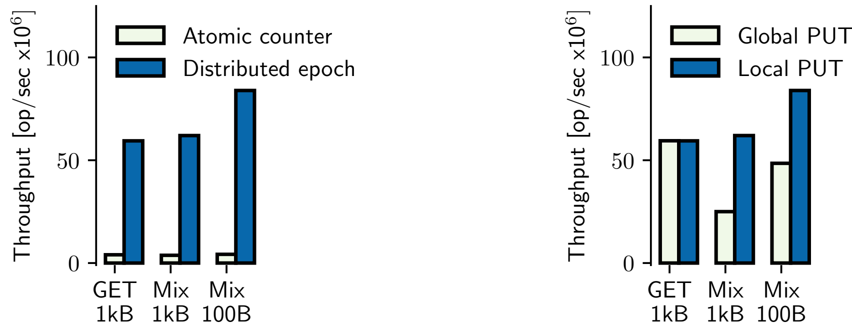
Redis: No modifications. Its implementation does not permit creating multiple instances within a single address space, thus we create multiple servers and use UNIX domain sockets with the hiredis client library. We use the hash to assign a key to an instance.

RAMCloud: We extracted the internal `ObjectManager` class, which encapsulates its key index, log allocator, and cleaner threads, allowing us access to invoke PUT, GET, and DEL methods directly. As it is the only recent key-value store implementing log allocation, we provide two configurations for comparison: using a *single* instance of `ObjectManager` (the default RAMCloud configuration) and one with an *array of instances*, to model a trivially parallelized log-based key-value store. Statistics collection was disabled. A key is assigned an instance based on its hash.

Masstree: We extracted the primary radix tree data structure, creating a single instance we invoke PUT, GET, and DEL on directly from client threads. MICA: The 'EREW' mode is enabled to support shared-memory configuration, allowing application threads to read and update objects located on any core; the segregated fits allocator was configured with 8192 size classes, offset by 256 bytes to accommodate the range of object sizes in our workloads; additional code was added to object deletion to invoke region coalescing[1] and the index hash uses SipHash [10, 31].

Nibble: We enable eight compaction threads per socket, segment and block sizes of 32 MiB and

[1]Provided by the authors of MICA from personal correspondance.



(a) Our distributed epoch provides nearly zero contention for PUT and GET — 20x greater performance over an atomic counter.

(b) Nibble restricts writes to local memory, resulting in a 1.75x-2.48x improvement in overall performance for mixed workloads.

Figure 12: Performance breakdown of various designs in Nibble: uniform YCSB 240 threads (Mix is a 50:50 ratio of PUT and GET).

64 KiB, respectively, and 1-2% of memory is reserved for compaction.

5.1 Breakdown of Components

To better understand the impact of each design feature within Nibble, we experiment with alternative designs, shown in Figure 12.

Epoch. We compare our distributed epoch design with that of a single atomic counter, shown in Figure 12a, and measure a large impact on performance — a 20x gain in throughput. While the epoch is required primarily by the compaction logic, to enable concurrent function with front-facing client operations GET, PUT, and DEL, it has the most impact on the latter. Releasing segments back to the free pools requires keeping track of in-flight operations, and the "time" at which the operation began. Use of an atomic counter requires threads to manually ensure forward progress. The hardware-based TSC increments autonomously, however, and can be read independently from any core in ca. 30 nanoseconds; the onus is on hardware designers to ensure all core TSCs are synchronized.

Furthermore, having threads remain as independent as possible by keeping track of their own operations and time stamps in a private cache line (their slot within the Epoch Table) allows for scalability. Only when compaction threads must interpret a global value will they scan the entire table and record the minimum; no locking is necessary for consistency, as we rely on the CPU coherence mechanism. The impact of our design is apparent regardless of the mix of operations.

Local vs Remote Memory. One design strategy to distribute work across all resources is to

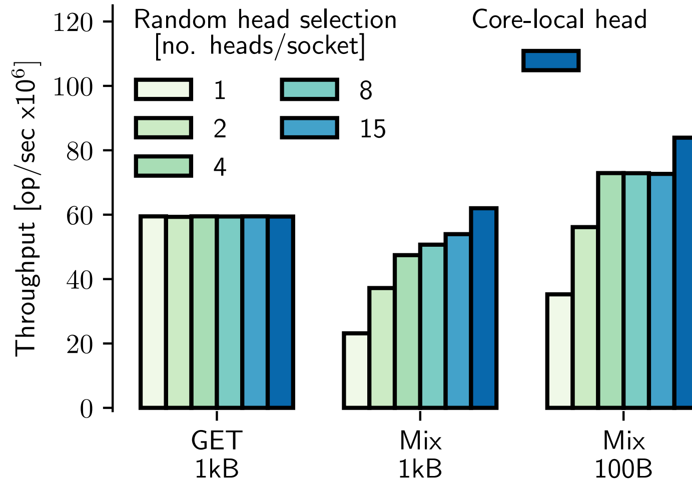
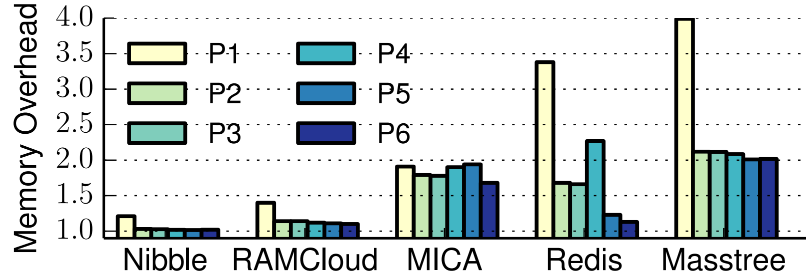


Figure 13: Performance breakdown of use of multiple log heads in Nibble: uniform YCSB 240 threads (Mix is a 50:50 ratio of PUT and GET). With PUT to local memory, more heads provide the greatest initial performance gains. Selecting the core-specific head, we avoid further contention, gaining an additional 15%.

use the key hash. On small platforms with two sockets, this results in only half of all operations to traverse remote-socket distances. Our large test platform, with 16 sockets, exaggerates this ratio, resulting in 15/16 accesses (94%!) to remote memory. When compared with a strategy of always writing to local memory (Figure 12b), we measure an improvement in throughput of 1.75x and 2.48x, for large and small objects, respectively. Furthermore, use of a hash to determine the log head may result in increased contention between threads during append operations. We next look at how to reduce contention on selecting a log head when implementing PUT.

Log Head Design. In systems like RAMCloud with a single log head, supporting parallel write-heavy workloads becomes strenuous. We evaluate the effect of having multiple log heads, and furthermore the method of choosing a head when implementing PUT (Figure 13). With threads writing only to local memory, we increase the number of heads available on each socket from 1 to 15, and select the head (i) randomly, or (ii) with 15 heads, based on the core the request is executing on. As discussed in Section 3, we accomplish the latter using the rdtscp instruction to determine socket and core IDs.

More heads results in greater, but diminishing, throughput: a maximum gain of ca. 2x over just one log head on each socket. For smaller objects, a performance plateau is reached with only four heads. There is room for further improvement, as random selection, and very high rates of operations,



(a) Memory overhead of each system. Reported is the ratio of total resident memory over aggregate size of allocated objects.

Label	Pattern	Label	Pattern
P1	60 → 70 B	P4	1 → 10 KiB
P2	1000 → 1024 B	P5	10 → 100 KiB
P3	1000 → 1030 B	P6	500 → 600 KiB

(b) Memory allocation patterns. 8 GiB of objects of the first size are allocated, 90% randomly removed, then again allocated of the second size until (i) insertion fails, or (ii) we allocate a total of 8 GiB again.

Figure 14: Measured memory overhead (bloat)

creates some contention on the local log heads. We designed Nibble to select a head based on the core; as long as threads do not migrate during a transaction, or cores are not over-subscribed, this results in zero contention for PUT, gaining an additional 15% in throughput.

Summary. Scaling log-structured designs have resulted in uncovering bottlenecks within a multitude of components. Solutions must ensure minimal synchronization and use of local resources as much as possible. To scale on extreme-SMP platforms, Nibble implements a write local, read global policy, contention-less log head selection, and an epoch that minimizes synchronization.

5.2 Memory Overhead

Ensuring efficient use of available memory alleviates applications from the responsibility of handling failed allocations that result from fragmentation when a dataset is known to fit within the system's physical memory. Additionally, long-running services must remain resistant to changes in both access and allocation patterns that may cause performance loss.

We visit the challenge of ensuring the client use of available memory by subjecting each system

to a series of object allocation and release cycles, modeled after prior work [151] shown in Figure 14b. Both RAMCloud and Nibble are log-allocating systems, and are able to accommodate destructive allocation patterns by relocating memory to reclaim holes. Nibble consumes less than 10% additional memory for all but the smallest objects; all systems have greater pressure when managing small objects, due to the fixed-cost of meta-data needed to manage their greater numbers (as is expected).

Non-copying allocation methods are unable to readjust allocated memory to reclaim holes, resulting in over-use of available memory. Our measurements show MICA and Masstree consistently using up to 2x the amount of memory actually allocated. How much memory is consumed also depends: Redis' use of memory varies significantly, depending on the size of objects used.

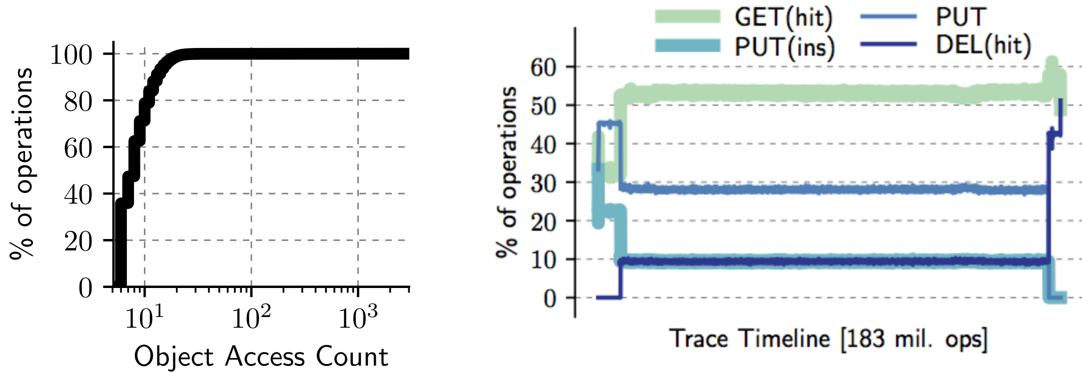
Summary. The ability to relocate objects in response to fragmented states provides the ability to keep memory pressure low. Non-copying allocators are unable to correct fragmentation after-the-fact, and the amount of memory bloat varies between designs.

5.3 Dynamic Workloads — Postmark

In this set of experiments, we opted to evaluate each system in how well it can support workloads which operate on more dynamic data sets – objects are destroyed, created, and are of a variety of sizes. To accomplish this, we capture a trace from a standard file system benchmark called Postmark. We execute an instance of Postmark within a file system created by TableFS, a storage layer that uses a key-value store within FUSE to implement file system operations.

Trace characteristics. Postmark the benchmark creates a set of files across a number of subdirectories, then performs transactions on them, such as append or deletion (refer to Table 4 for our configuration of Postmark). The trace we captured has 183 million operations and 18 million objects (see Table 4) with, on average, 5.7 GiB of active objects allocated in memory at any given moment. Object access frequency is show in in Figure 15a. The majority of objects are accessed a small number of times – 18 individual operations or fewer per object for 80% of the entire trace. Such objects are created, accessed a minimal number of times, then removed. With little reuse of objects, most of the work in this trace is in the *insertion and removal of unique objects* and responding to GET requests.

Figure 15b shows a breakdown of the types of operations. There is a brief setup phase with a high



(a) Objects accessed a total of 18 times or less comprise 80% of all operations – very little reuse.

(b) The set of accessible objects changes over time: 10% of operations delete existing objects and another 10% insert new objects, DEL(hit) and PUT(ins), respectively, and 20% of operations overwrite existing objects (=PUT-PUT(ins)).

Figure 15: Characteristics of our trace captured from Postmark. Overall, the workload presents a dynamic behavior: most objects are accessed infrequently, yet a consistently high churn in objects is present. 20% of all operations will add or remove objects, putting stress on each system's object allocator logic.

portion of writes and object insertions – PUT and PUT(ins) respectively. Throughout the majority of the trace, a consistent 10% of operations insert new objects, and 10% remove existing objects – DEL(hit). At the end of the trace, nearly 50% of operations are deletions, as Postmark cleans up all objects.

Scalability. We measured the ability to sustain executing the trace, with a thread executing the trace independently on its own set of objects. As we add more threads, we are adding more work to the system (weak scaling), but as each thread is independent, no thread coordination occurs. Figure 16 shows our results. To ensure a consistent capacity utilization (ca. 17%), we increase the total amount of memory allocated to each system as more threads are added. Runtime increases very little for Nibble, as each thread can independently (over-)write objects without synchronization with other threads, and having created all objects on the local socket, object accesses are fast. Within MICA, runtime is higher due to multiple factors: the majority of PUT and GET traverse the chip interconnect, as an object's location is determined by its key's hash, and, there is high contention on the per-core heap allocators, which arises when objects are newly allocated or destroyed. The latter is greatly increased with greater numbers of threads.

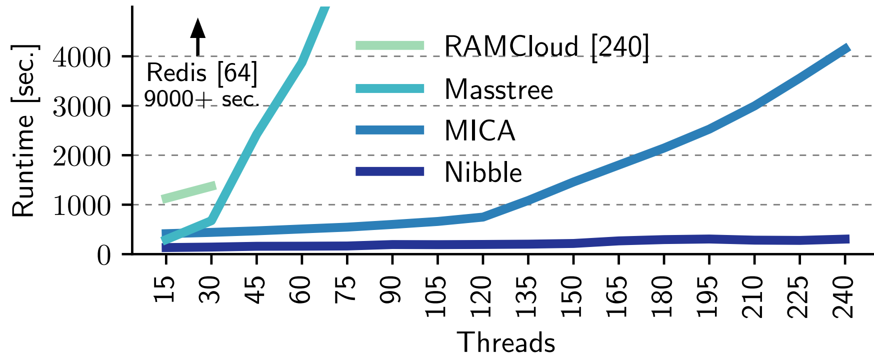


Figure 16: Weak scaling of our Postmark trace, measured in time- to-completion, at a consistent 17% memory utilization: as each thread executes the entire trace on a private working set, we increase available memory as appropriate (except for Masstree). Nibble completes the trace within a few minutes, whereas the other systems require upwards of an hour or more with more threads. RAMCloud was able to run with only 2% utilization until system memory was insufficient.

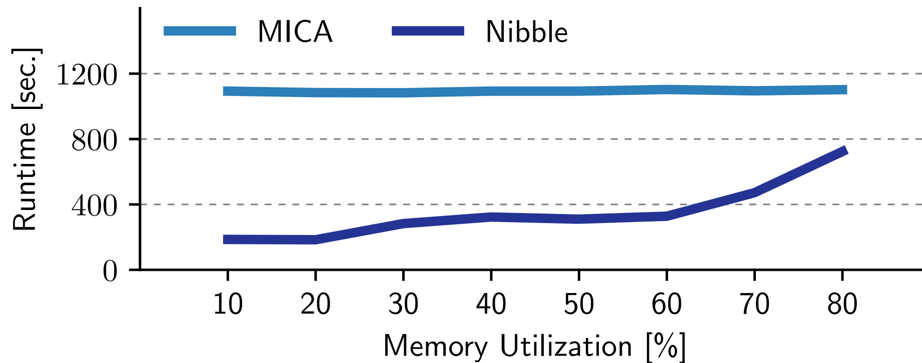


Figure 17: From Figure 16 we measure time-to-completion for Nibble and MICA at 135 threads, and progressively decrease available memory. At 65% utilization, Nibble's runtime doubles due to compaction costs. MICA seems unaffected, however any such effect is masked by contention on its per-core heap allocators.

With high rates of insertions and deletions, Masstree unfortunately suffers from frequent rebalancing of its trees, resulting in very high completion times. We were able to execute RAMCloud only by reducing the utilization to 2% overall, with 240 instances. At 45 threads, this would require more memory than exists in our test platform.

Impact of capacity utilization. Using the two highest-performing systems from the prior experiment – MICA and Nibble – we measure the effect of memory pressure on overall performance. With 135 threads, we decrease available memory such that the utilization increases to 80%, shown in Figure 17. MICA's measured performance is as explained earlier: per-core heap allocators are

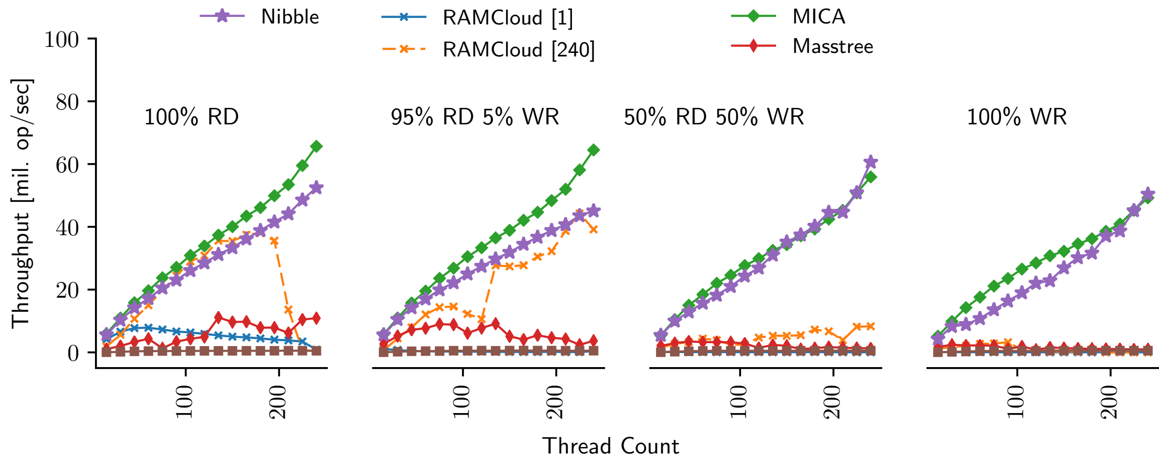


Figure 18: Uniform YCSB throughput measurements on 1 TiB of data – ca. 230 1 KiB objects. "95% RD 5% WR" is a workload with GET composing 95% of all operations and 5% with PUT. No configuration uses DEL. Approximately 8 TiB of storage capacity was provided to ensure a low utilization rate of ca. 12%.

guarded each by a mutex, creating contention for the 20% of operations which create and destroy objects throughout the trace. Any effects due to memory pressure may be masked by this bottleneck. As Nibble does not experience this bottleneck, it spends more time compacting segments, as expected, more than doubling the execution time at 80% utilization, but Nibble still outperforms MICA under 80% memory utilization.

Summary. Workload patterns with churn place high stress on the memory management of key-value stores. Nibble scales by ensuring object insertion and deletion are independent operations, completing Postmark in 80% less time than MICA at 135 threads.

5.4 Data-Serving Workloads — YCSB

We implemented an internal YCSB workload driver, measuring the scalability of each system as a function of throughput and number of client threads. Figures 18 and 19 show our YCSB measurements of each system. The notation [*N*] indicates *N* instances of the associated system used for that configuration. All YCSB experiments do not perform DEL and all objects are instantiated before throughput is measured; thus, PUTs never require insertion of new objects.

Uniform access. With uniform key distribution (Figure 18), both Nibble and MICA are able to scale with increasing threads. For mostly-GET configurations, MICA demonstrates up to 25% and 43% greater throughput for 100% and 95% GET, respectively. With high percentages of PUT,

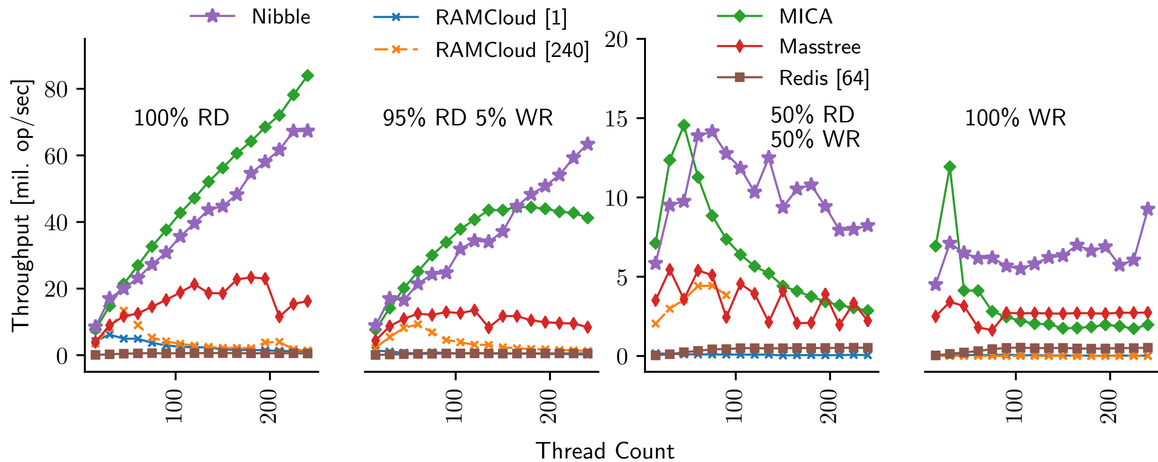


Figure 19: Zipfian YCSB throughput measurements on 1 TiB of data – ca. 230 1 KiB objects. "95% RD 5% WR" is a workload with GET composing 95% of all operations and 5% with PUT. No configuration uses DEL. Approximately 8 TiB of storage capacity was provided to ensure a low utilization rate of ca. 12%.

both systems show roughly equivalent performance; Nibble executes PUT to local socket memory, whereas MICA distributes them based on the key hash. Scalability for both systems is achieved via optimistically concurrent indexes, and the ability to overwrite (and for Nibble, append) independently.

Single-instance RAMCloud scales up to ca. 30 threads, then declines. Threads in YCSB quickly saturate the off-chip bandwidth of a single socket: by default, RAMCloud's internal memory allocator will fault in all pages backing the log and index, restricting memory to a single socket. Any application that has a fault-then-use pattern will exhibit this type of bottleneck at scale, but less so on much smaller platforms.

Multi-instance RAMCloud plateaus much later (after 135 threads), but drops precipitously thereafter. With 240 instances, the operating system scheduler will spread out all initialization tasks, enabling pages backing the log and index to be faulted in across all memories (there is significant variability in measured performance for RAMCloud, as memory allocation by the operating system is indeterministic). The drop may be attributed to growing contention on the index. Each bucket in the hash table is guarded by a mutex, thus any two lookup operations will serialize. Even for the tree index in Masstree, which has greater lookup costs, its optimistic design supports further scaling.

High percentages of PUT create additional bottlenecks for RAMCloud and Masstree. Single-instance RAMCloud is limited to a single log head, which becomes an immediate bottleneck, and multi- instance RAMCloud, nevertheless with 240 log heads, is limited by append operations that

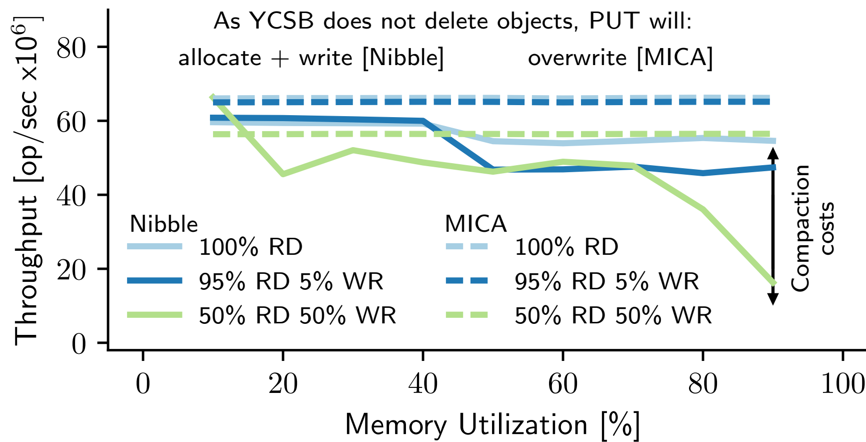


Figure 20: Throughput of MICA and Nibble driven with YCSB (uniform) using all 240 cores, steadily increasing the capacity used.

frequently chose the same head, as objects are assigned based on their hash.

We ran Redis with 64 instances. Each server contains a single thread to service requests. The use of UNIX domain sockets adds additional buffer copying, as well as system call overhead into the critical path, limiting scalability, a 6× or more difference in throughput.

Zipfian access. Both MICA and Nibble scale with the number of threads in a pure read configuration, for a maximum difference in throughput of 25% at 240 threads. Overall throughput is slightly higher than with a uniform distribution, as zipfian results in a smaller effective working set size. Masstree shows some scalability, and along with MICA and Nibble, it utilizes optimistic concurrency within their indices, enabling contention-less access for read operations. Due to the smaller working set size, the bucket locks within RAMCloud's index limit its scalability across all configurations.

With greater ratios of PUT, a greater portion of operations will overwrite a subset of objects in the dataset. As hashes determine which instance in RAMCloud objects are written to, a zipfian distribution will create skew in the load across the 240 log heads – more contention on a subset of all log heads. This skew also affects MICA in a similar fashion: frequent writes to a subset of objects places contention on the *memory* backing those objects, as objects are overwritten with each PUT.

In contrast, writes in Nibble *are not* directed to a shared subset of memory (the log-structured nature ensures this) and where an append occurs is not determined by the hash, but instead by the

per- core log heads. Thus, threads will copy objects to private memory areas in local memory, but contention still exists within the index to reflect the new location created for the objects.

Impact of capacity utilization. Figure 20 examines the impact on overall throughput when the total system capacity is increased to near full. Given that MICA and Nibble are the highest-performing systems in our YCSB study, we limited this analysis to just these two. At high utilizations ($\geq$ 70%), compaction in Nibble becomes the dominant overhead. For systems which do not utilize log-based allocation, such as MICA, no allocation logic is exercised in these experiments (as the YCSB workload does not delete or create new objects). At high capacity utilizations, log-based allocation becomes prohibitive under moderate to substantial write configurations.

Summary. Nibble supports scalable execution, within 25%- 43% of the best non-log-structured system (MICA) for read workloads, supported by the use of a scalable epoch implementation. Our use of multiple log heads supports write-dominated workloads and out-performs all compared systems by 2x for skewed workloads at high thread counts, demonstrating our write-local read-global policy is advantageous. With static data sets and high memory pressure, use of a non-log-structured system proves better for write-dominated workloads, due to lack of compaction costs.

CHAPTER VI

AN ANALYSIS OF COMPACTION

In log-structured systems, a significant cost that can impact overall performance is the work needed for compaction — relocating items to reclaim fragments of unused memory. We have seen in the prior chapter that such costs can be significant when a machine's memory capacity is nearly fully utilized by application data (an expected behavior seen in other systems under similar circumstances, such as for SSDs [23] and file systems [149]). As there are many applications whose performance is correlated with available memory — key-value stores, databases [83], etc. — we believe there is opportunity to improve on how in-memory log-structured systems manage their data when little available capacity remains, to avoid perturbing application performance. In this chapter, we dive into the mechanisms and logic of compaction, using Nibble, and discuss two main points:

1. How can compaction costs be reduced, and what parameters is the work of compaction most sensitive to? Do the mechanisms, such as copying, or the policies, have the most impact?
2. The strict nature of appending (and operations such as deletion) necessarily create work for compaction operations; how much can this be avoided entirely?

Except where indicated, all measurements in this chapter are performed on smaug-3, a system similar to smaug-1: an HPE SuperDome X machine with 16 Intel Xeon E7-8890 v3 18-core cache-coherent processors (a total of 288 physical cores) with 12 TiB of aggregate ECC DRAM. SMT is enabled, but alternate threads were unallocated across experiments.

6.1 Breakdown of Compaction

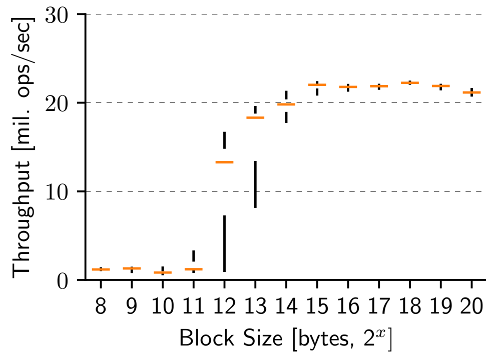
In this section, we explore what costs and parameters affect compaction the most, tweaking parameters within Nibble's design, and observe how each affects end-to-end performance. We run YCSB on 1 TiB of data with uniform random access and under very high capacity utilization – 90% – thus, changes we make to Nibble's design will affect compaction performance, and allow us to study its characteristics.

The following questions we aim to address in this section:

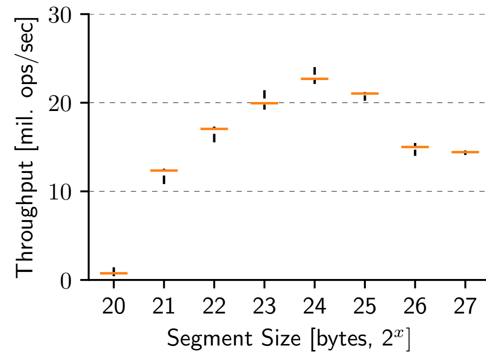
- Two obvious candidates are the segment and block sizes. Does their specific size have any effect on compaction performance? Does this vary across object sizes used by the application?
- Compaction is easily parallelized, and given these threads occupy CPU cores themselves, they might compete with application threads. How few can we utilize to still achieve best performance?
- The bulk of work performed by compaction is in just moving memory around. Which mechanisms are available? How do they perform across data sizes? Does the underlying memory move mechanism have a significant effect on overall performance?

Segment and block sizes. Figures 21, 22, and 23 illustrate the effects of varying the block and segment sizes on overall performance when memory utilization is high. For 1 KiB and 500 byte objects, it is apparent that small block sizes reduce performance to nearly a crawl, whereas 'large' block sizes saturate performance, but no gains result from greater block sizes. Two reasons explain why small block sizes may reduce performance: (1) objects are chopped up into smaller pieces, thus copying them from segment to segment requires many more individual copies, or (2) smaller blocks means segments contain a larger number of them, increasing their construction and destruction costs, which compaction does continuously. As the drop in performance for 1 KiB objects seems to occur at 8 or 16 KiB block sizes (similarly for 500 byte objects) it would suggest reason (2) is more likely the cause here, as objects are less likely to be divided among many blocks.

Examining segment sizes, a different trend is visible. A bell shape curve in the performance suggests there is a 'sweet spot' for an appropriate segment size. When segments are too small, compaction threads must iterate over a larger number of segments to reclaim the same amount of free memory, increasing the logistics work of updating segment metrics (the 'goodness' metric from earlier), sorting, destruction and construction costs. When too large, none of the memory from a segment can be reclaim until all objects within the entire segment are relocated, creating larger delays in recycling memory.

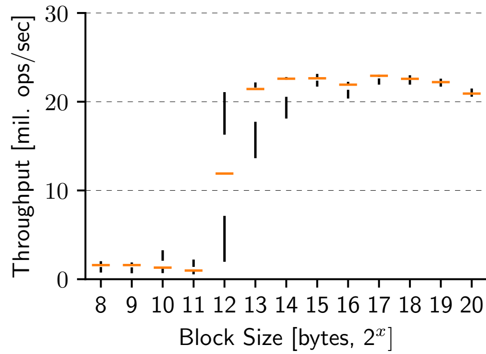


(a) Fixed 32 MiB segment size.

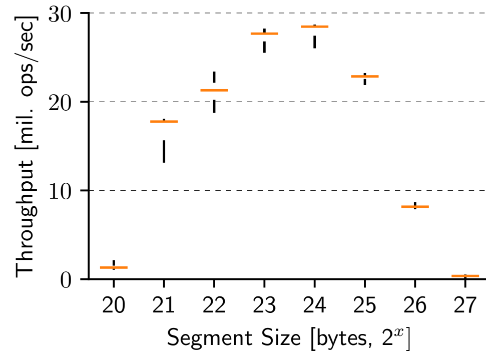


(b) Fixed 16 KiB block size.

Figure 21: Effect of varying the block and segment size in Nibble on the throughput, executing YCSB (uniform) with **1-KiB objects** on smaug-3. 224 client threads in total, with 8 compaction threads per socket, on 1 TiB of data at 90% fill capacity (1.1 TiB total memory allocated to Nibble). Three runs executed, showing the median, and standard deviation.

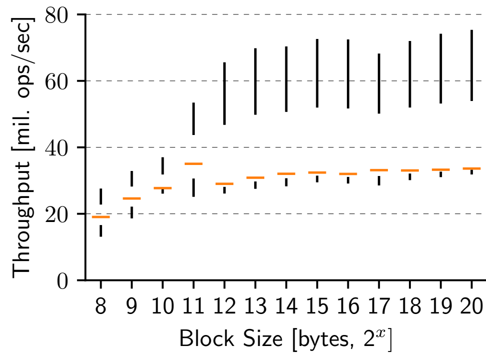


(a) Fixed 32 MiB segment size.

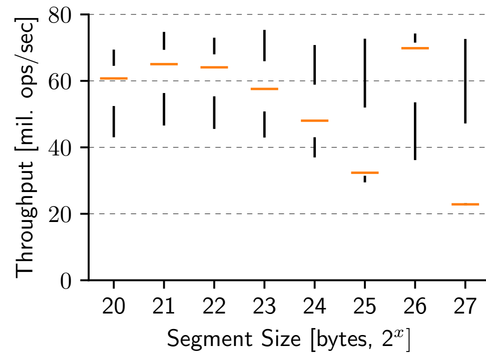


(b) Fixed 16 KiB block size.

Figure 22: Using **500-byte objects**.

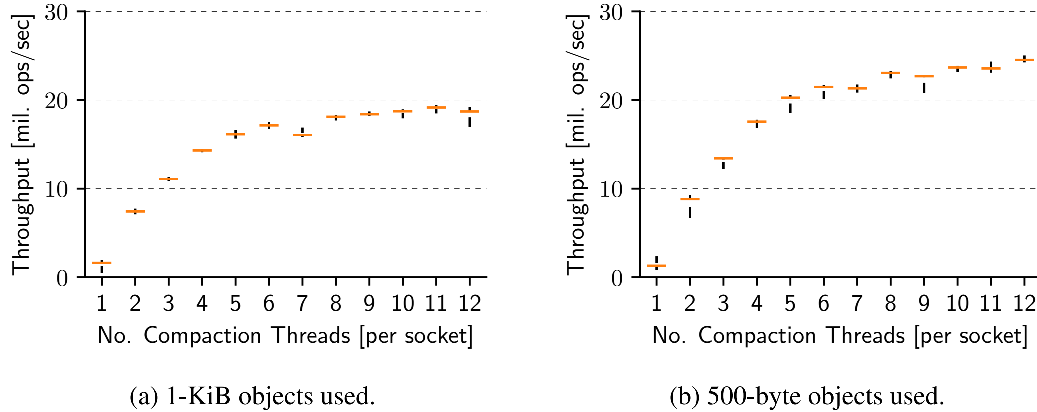


(a) Fixed 32 MiB segment size.



(b) Fixed 16 KiB block size.

Figure 23: Using **64-byte objects**.



(a) 1-KiB objects used.

(b) 500-byte objects used.

Figure 24: Effect of varying the number of compaction threads in Nibble (per socket) on the throughput, executing YCSB (uniform) on smaug-3. 224 client threads in total, on 1 TiB of data at 90% fill capacity (1.1 TiB total memory allocated to Nibble). Three runs executed, showing the mean, and standard deviation.

For very small objects (Figure 23), measurements are not definitive as to the impact of block and segment size selection on overall perforamnce. Just looking at the median values, it would suggest that larger block sizes help, and segment sizes of a few megabytes provide maximum throughput.

Compaction parallelism. Compaction in Nibble is a parallel process; each thread manages a unique set of segments to observe for compaction, and can independently allocate new segments into which to move objects. Figure 24 highlights the effects of using more threads per socket. A greater gain in throughput results, but with diminishing returns, with approximately a 20× gain from use of a single thread, and see a similar effect for smaller object sizes. We must balance the number of threads allocated for compaction so as to avoid contending for CPU time with application threads, as compaction threads will periodically poll the block allocators to determine utilization and recalculate their metadata describing segments to determine viable candidates. Nibble by default uses eight compaction threads per socket.

Copying mechanism. As segment compaction primarily relies on memory copying mechanisms, especially for moving larger objects, we wondered if the *choice in mechanism* would have an impact on overall compaction performance. For a brief comparison, we considered two popular tools: the memcpy(2) routine implemented within GNU libc, and the x86 CISC instruction sequence `reb movsb`. The former is able to take advantage of advanced instructions in newer architectures for

moving data very quickly, such as streaming SIMD registers, whereas the latter exploits micro-architectural features present in the processor to optimize data movement that other instructions may not have access to. Figure 25 shows the latency of copying randomized locations within local-socket memory, as the region size varies. Figure 25a shows absolute latency, and Figure 25b shows the percent gain in latency of memcpy(2) over `rep movsb`. For most data sizes, memcpy(2) shows roughly 10-25% lower in latency, except for a curious reversal for sizes in the range of 500 bytes and 4 KiB – sizes used in most of our evaluation – where the performance is either equivalent or memcpy(2) has up to 15% greater latencies.

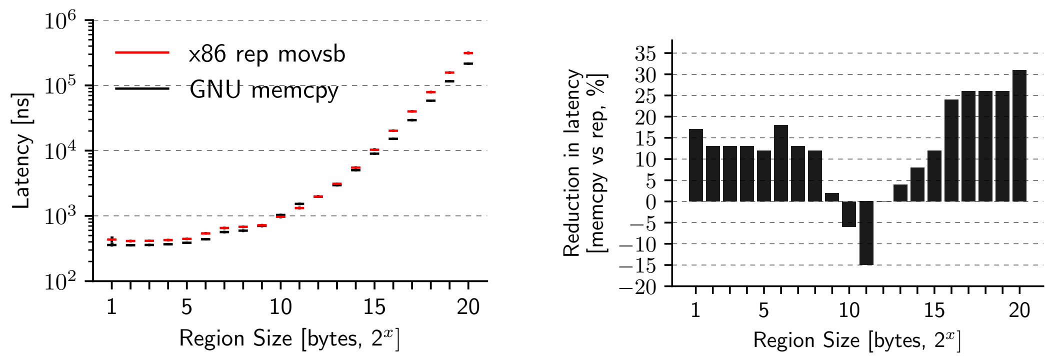
When we examine the impact of memory copying mechanisms on overall performance of Nibble running the YCSB workload, shown in Figure 26, we see a negligible impact across numerous object sizes. This result suggests that the choice in tool plays a small role in overall performance, where most bottlenecks may be within the machinery of Nibble itself.

Summary. Numerous design configurations and build parameters can influence the performance of compaction. Our measurements illustrate that the machinery of managing memory in blocks and segments can have a significant impact on overall performance – the rights sizes must be used. Parallelism also has significant influence, but too many threads do not provide increasing benefit. When it comes to the actual copying of memory itself, we do not see significant gains or loss on end-to-end performance, suggesting the mechanisms surrounding compaction play a more significant role.

6.2 Relaxing the nature of the log-structured design

In this section we look at how we can *entirely avoid creating unnecessary work* for compaction in the first place by avoiding appending to the log for certain operations. An advantage of this is an opportunity for sustained higher performance experienced by clients, by eliminating expensive background compaction activities that consume memory bandwidth and CPU time.

Certain file systems, for example NetApp WAFL, Oracle ZFS, etc. leverage copying semantics, like what we see in log-structured designs, to provide better recovery properties and consistency of information. Retaining prior instances of objects provides the ability to version it over time — in case you wish to roll an object back due to corruption, compare changes over time, or to



(a) GNU C `memcpy` and x86 `rep movsb` routines on local-node memory, single-threaded, unloaded, with 8 runs per data point.

(b) To more clearly visualize the differences in latency, we show by how much lower the GNU C `memcpy` is compared to `rep movsb`, in percent. For example, `memcpy` can move 2^6 bytes over 15% faster than `rep movsb`; at 2 KiB, it is 15% slower.

Figure 25: Latency of memory copy mechanism as the size of the copy region varies. Source and destination locations are in local memory only.

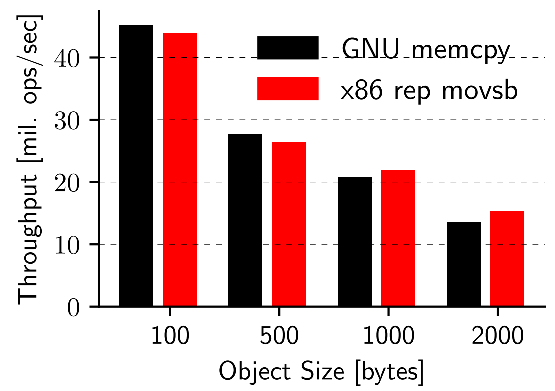


Figure 26: Performance as a function of memory copy mechanism. Client threads copying in/out objects use the x86 `rep movsb` mechanism; only the compaction's use of memory movement was modified. Shown are four object sizes, executed with 224 client threads at 90% capacity utilization, running YCSB uniform.

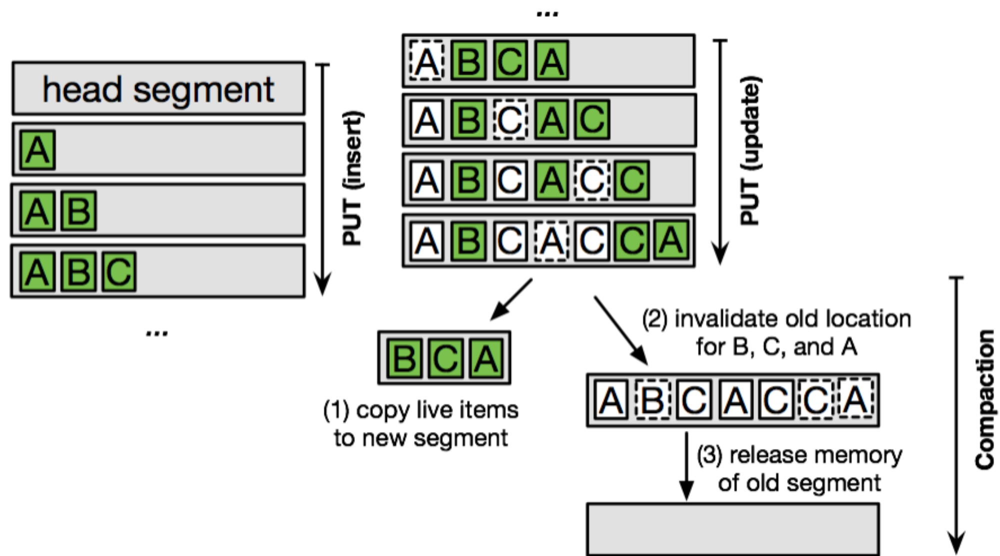


Figure 27: Operations that update existing objects by appending new instances to the log heads create unnecessary work for compaction to perform: object versioning occurs, and is unused in Nibble. Dashed box outlines indicate values that are made stale due to an update.

increase availability of a service. In the case of key-value stores where such features are not made available to clients, as is the case in Nibble, log-structured management creates stale versions that must be routinely cleaned up. In the illustration in the above Figure, update operations invalidate prior versions, but the lifetimes of such versions are extremely short – *stale versions of objects are alive just long enough to be reclaimed and recycled for new head segments*. Routine cleaning is burdensome and creates unnecessary work.

The act of *appending* an item to a log is actually two operations: (1) an *allocation* for new space, followed by (2) a *write* to that location. With Nibble, these steps occur separately; both are used in sequence to implement PUT. We can, however, differentiate between whether just the former or both are required: during a PUT, if a key exists, we may overwrite the current location, else perform a normal insertion.

Understanding the purpose of appending What is then the point of appending? Before we go further into evaluating our proposed approach, we will discuss the advantages and disadvantages of appending itself. The main goal with Nibble's design overall is to avoid fragmented memory states, and additionally provide higher performance than existing designs. Let us make explicit the

benefits afforded by the use of appending to allocate memory, keeping in mind the use of multiple head segments:

1. As appending creates an instance of an object, we can implement policies that decide exactly *where* to instantiate it (e.g., local or remote memories), giving us some control over performance, which we saw some impacts of in our earlier Postmark measurements;

2. Appending allows multiple threads updating a common set of objects *to avoid writing to the same location in memory*, reducing contention on shared memories and on cross-socket cache-coherence from, for example, skewed workload behaviors, and only the key within the index would become a source of contention within Nibble — to update the reference to the latest thread's stored copy within the log;

3. Allocation is very fast — simple 'bump' arithmetic, or, an increment.

Aside from the mentioned disadvantages of appending (more compaction work), appending for each object means we must indeed copy the entire object. If objects are large (e.g., a few hundred KiB or more) and updates to such objects impact merely a few bytes therein, repeatedly copying unchanged data also becomes costly, consuming memory bandwidth.

Reducing fragmentation is afforded via use of compaction (and keys for relocating them), not strictly through the use of appending. Unlike traditional heap allocators, which do use free-lists or some variant thereof for allocation, Nibble does not give out explicit virtual addresses of objects in memory to clients; if it did, we would be bound to holding an object at its first location in memory throughout its lifetime, and be subject to potential fragmenting allocation patterns. By using keys and having background tasks relocate objects, we can reclaim otherwise fragmenting memory holes.

Our proposed changes to Nibble. For many workloads, writes are frequent. Given Nibble makes no use of prior versions of objects, currently, we propose to only append object instances to the log when they are first created, and when objects are updated, we directly update their existing copy in-place. Figure 28 illustrates an example of this in action.

To be more explicit, our changes to Nibble, illustrated in the above Figure, are the following: PUT operations where the specified key *misses a lookup in the index* are classified as *inserts* and

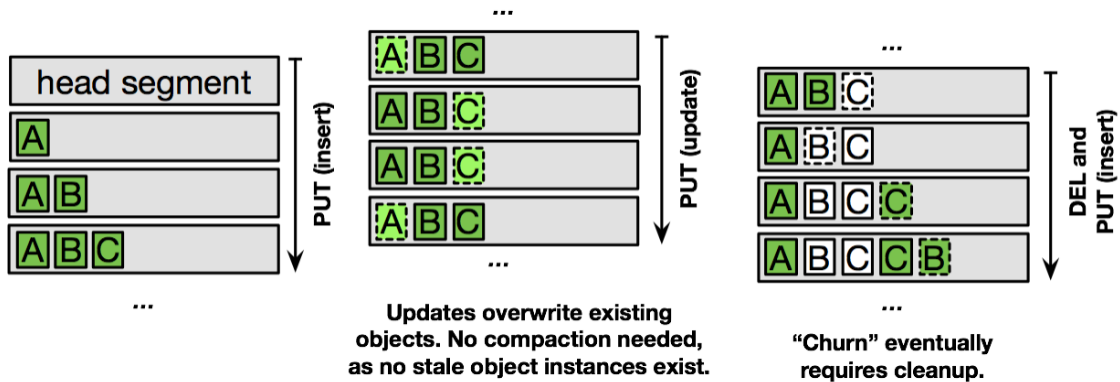


Figure 28: By allowing update operations, such as used by PUT, to overwrite existing object locations (subject to constraints such as the new size is ≤ existing allocation size) we can avoid creating unnecessary cleanup work for compaction threads. Light-green boxes indicate objects overwritten with new data (operation sequence is identical to the prior figure). When new objects are inserted (their keys do not exist in the index) do we perform a bump allocation in the log head, and append the item. Only when such operations — insertion and deletion — occur frequently, will compaction be necessary. *Workloads that never insert or delete objects (e.g., after loading the initial working set) will not incur overheads due to compaction.*

thus will result in an append; PUT operations where the specified key results in a *successful lookup in the index* are thus classified as *updates* and an object's memory will thus be overwritten, instead. Segments and objects stored within the log are no longer immutable. We discuss the implications and correctness of this next.

We label the version of Nibble with these modifications as **'Nibble-2'**. The term **'Nibble'** henceforth refers strictly to the version of our system not incorporating changes discussed within this chapter.

Nibble-2: Correctness and implications. We originally described segments in Nibble to become 'closed' and are thus immutable. By allowing threads to overwrite existing objects, we are breaking this property. What impact does this have on other aspects of Nibble's design? How do we ensure competing operations, such as reading, writing, and compaction, do not result in data corruption within the logs, or corrupt data being returned to clients? As we show, correctness is in fact not affected, however there is a reduction in concurrency between operations when operating on common objects. Below are scenarios of concurrent activities between threads on common objects, and how they remain correct.

(1) One thread reading, another writing the same object. Correctness is preserved through the protocol implemented within the concurrent index: readers will wait until the bucket version is even, then record the version to the stack, use the reference to copy the object out, then again read the bucket version and determine if the version has changed. If the version has changed, what they copied is potentially corrupt due to a concurrent writer, and they must restart. Writers will atomically increment the version to an odd value (and if already odd, they wait), mutate the reference to point to the new object instance, then again atomically increment the bucket version.

There is a subtle difference in the order of steps performed that leads to variation in the latency of reading and updating objects: In Nibble-2, as writers overwrite the existing instance of an object, they must do so *while holding the bucket lock*. Within the design of Nibble, a writer thread may append the new object to its own head segment's memory, and only after that point, acquire the lock on the bucket to update the reference. Thus, update operations in Nibble-2 have slightly lower concurrency with readers, than compared to Nibble.

(2) Competing threads updating the same object. Correctness is preserved via the bucket version protocol within our index — no two writers may update an object instance at the same time as only one may atomically increment a version to be odd. Again, performance variations exist: as writers overwrite object instances, they can only do so while holding the bucket lock (Nibble-2) instead of being able to make a copy to their log's head segment first, then very quickly updating the reference within the index (Nibble).

(3a) Compaction copying an object to a new location, while another thread is reading it. A compaction thread pretends to behave like a writer when copying an object to a new location: it locks the bucket holding the key and reference, then copies the object before unlocking (incrementing the version twice in the process). Concurrent readers will be forced to restart, and join new readers, which will then spin on the version until it becomes even, indicating a compaction thread has finished relocating the item. Readers will be released on an even version, and (with correct use of memory barriers) will see the new reference stored for that key by the compaction logic.

In this situation, for both Nibble and Nibble-2, the object is only relocated while the corresponding bucket lock is held, thus performance experienced by either the readers or the compaction threads will remain the same.

(3b) Compaction copying an object to a new location, while another thread is writing it. By the same principles described in (3a), with compaction threads pretending to act like writers, only one compaction thread or client writer thread will be able to mutate a given object's state at any time.

As in (1) and (2) above, objects can only be overwritten with new state while the corresponding bucket lock is held, thus in Nibble-2, concurrency is reduced: client writer threads will not append to a log head segment prior to attempting to lock the bucket, and instead must perform their copy after acquiring the lock. Operations will experience higher latencies, as a result.

Other log-structured systems which have similar conceptual designs to Nibble have much stronger requirements on segments remaining immutable. RAMCloud [151], for example, ensures data remains persistent by mirroring its in-memory logs onto disk and implements protocols to ensure that, during recovery, the order of recorded object deletions and updates are maintained. Interactions with concurrent network requests via RDMA protocols (and other forms of DMA engines) also require segments to remain immutable, to allow greater levels of concurrency between updates, segment compaction, and a high number of read operations. In his PhD dissertation, Steve Rumble [150] (the designer of log-structured memory within RAMCloud) specifically states the following reasons:

> §4.4 — *Parallel cleaning in RAMCloud is greatly simplified [. . .] since segments are immutable after they are created, the cleaner never needs to worry about objects being modified while the cleaner is copying them.*

Nibble and Nibble-2 avoid this issue by requiring updates and compaction threads to hold the corresponding bucket lock representing an object, in effect serializing operations. Parallel compaction (cleaning) is still possible in either design, and the complexity in Nibble and Nibble-2, we argue, is not greatly increased.

> §6.1.5 — *A major problem with updating objects in place is that segments would no longer be immutable. This makes concurrent access to segments much more complicated and expensive. For example, when the Infiniband [sic] transport layer is used, RAM-Cloud will not copy objects out of the log to service read requests. Instead, the network*

interface is instructed to DMA the data directly out of the log in order to reduce latency and conserve memory bandwidth. Making segments immutable ensures that the data does not change out from under the network card. Allowing segments to be modified would require addressing this concurrency problem with mutual exclusion, which would add undesirable latency to both read and delete operations.

Network devices would need to participate in locking protocols with CPU execution, thus complicating the design. Within Nibble and Nibble-2 there are no interactions with disk or asynchronous networking hardware that would otherwise require challenging protocols for handling locking to be implemented, allowing us instead to relax the log-structured behavior of appending for each update to an object. In the next section we examine the impact on performance of Nibble-2, highlighting both the advantages and disadvantages.

6.3 Evaluation – Nibble-2

We now revisit earlier experiments and compare the end-to-end performance of our modifications with the original design. The questions we aim to address are as follows:

- Does the design in Nibble-2 significantly improve throughput performance for write-heavy random-access workloads?
- What can we tell for low capacity utilized scenarios on experiments using the entire machine – does the act of appending to a local head segment matter?
- Is the latency of PUT affected, and is there a noticeable difference when objects are inserted versus updated?

In Figure 29 we revisit the earlier MICA vs Nibble measurement using YCSB and varying the capacity utilization, substituting MICA for Nibble-2. It becomes apparent that the avoidance of appending does not create stale object versions, eliminating work for compaction threads. The result is high performance regardless of capacity utilization.

We compare performance using YCSB at low capacity utilization, in Figures 30 and 31. Uniform access patterns show little performance variation; a slight drop with pure-write (‘Uni 0’) is due to

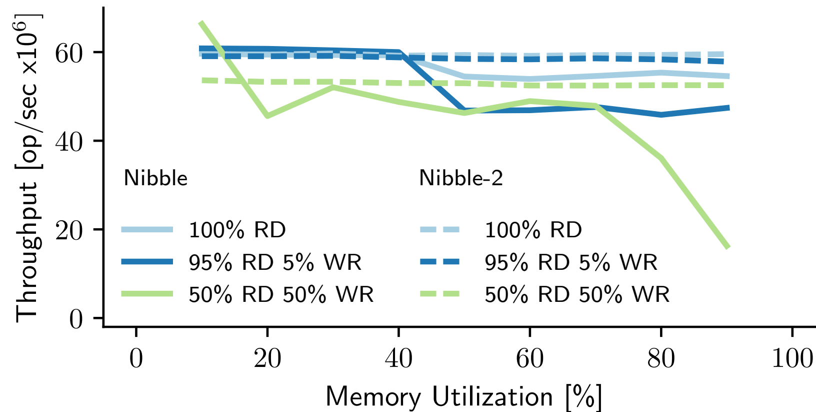


Figure 29: Throughput comparison of Nibble and Nibble-2 on YCSB (uniform) with 240 threads on the HPE Superdome X (smaug-1). As the amount of available memory decreases, compaction costs draw performance down in Nibble. The dip at 50% is due to compaction threads triggered into polling in attempts to reduce fragmentation. Nibble-2 experiences zero compaction, as all PUT operations overwrite existing objects. YCSB does not delete or insert objects.

Nibble-2 overwriting objects during PUT at their current location, determined at the time the object was instantiated. In contrast, threads in Nibble always write to local memory, providing slightly higher performance. Otherwise, performance between systems under this workload remain the same.

In zipfian distributions (Figure 31) we expect performance to be limited by the contention on bucket locks and memory backing the small shared subset of objects defined by the distribution characteristics. Nibble-2, curiously, exhibits higher performance. We assume this is attributed to the workload benefiting from the aggregate last-level cache capacity from all CPUs in the system – over 700 MiB – as objects are simply overwritten without moving them to new locations (as appending would require).

We measured the latency of updates and insertions, shown in Figures 32 and 33. We pin a single thread to a core on socket 0 and measure the latency for each type of PUT to memory on each socket: the first bar is local memory (socket 0 to 0), and each of the subsequent 15 bars are writes from socket 0 to that remote socket. Insertions perform the same for both Nibble and Nibble-2, as the machinery is identical (append+copy). For updates, Nibble-2 latencies are roughly 25% lower, due mainly to the avoidance of having to allocate new space (and potentially roll the head segment). The same observation is made for both 1-KiB and 64-byte object sizes.

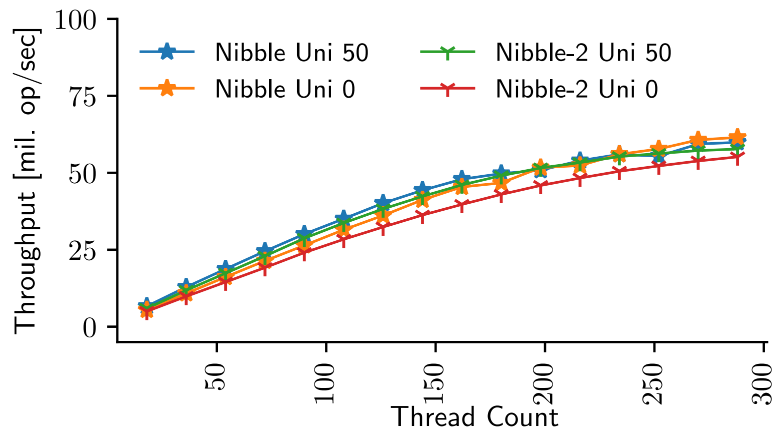


Figure 30: Comparison of Nibble and Nibble-2 for YCSB Uniform distribution accesses of 1 KiB objects. ‘50’ means 50:50 ratio between PUT (update) and GET; ‘0’ means a pure-PUT (update) workload configuration. Nibble-2 shows a slight drop in throughput for the pure-PUT workload, as it breaks the “local-write global-read” policy, updating objects wherever they happen to reside in memory; Nibble always updates by append to local memory.

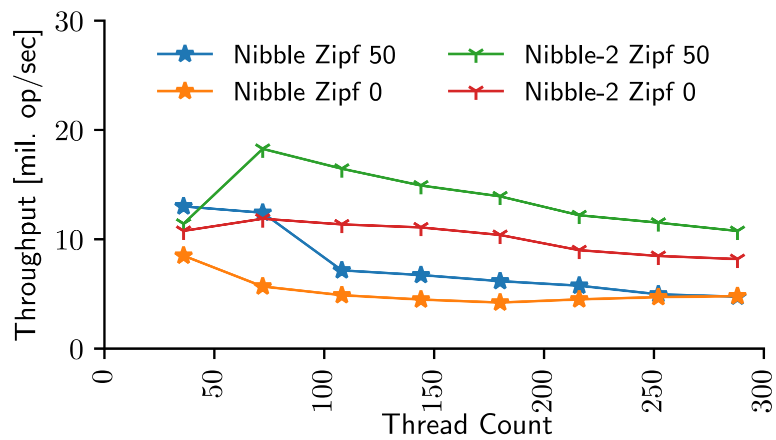
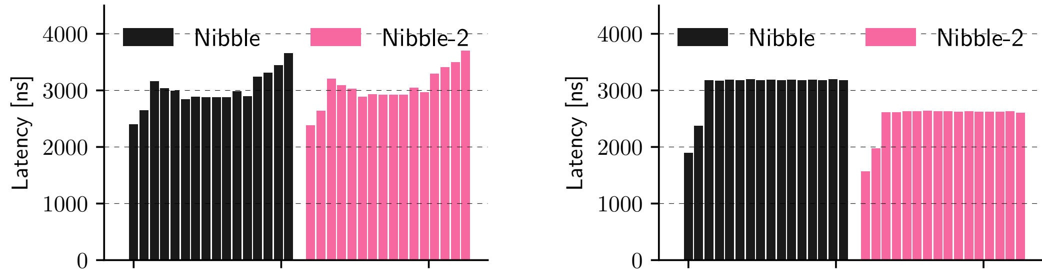


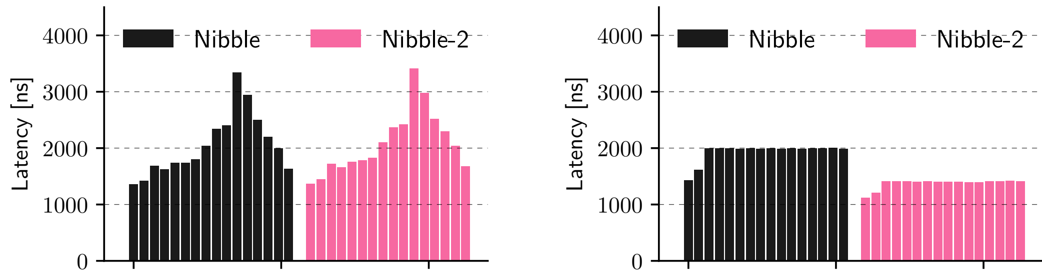
Figure 31: Comparison of Nibble and Nibble-2 for YCSB Zipfian distribution accesses of 1 KiB objects. ‘50’ means 50:50 ratio between PUT (update) and GET; ‘0’ means a pure-PUT (update) workload configuration.



(a) Latency of inserts. YCSB Uniform 1KiB objects.

(b) Latency of updates. YCSB Uniform 1KiB objects.

Figure 32: Latency for PUT between Nibble and Nibble-2 for a single thread running without contention at low utilization. Insert behavior for PUT shows no difference in performance, as both systems behave the same (append). Update for Nibble-2 shows roughly a 15% reduction in latency for a PUT of 1-KiB objects to local memory.



(a) Latency of inserts. YCSB Uniform 64-byte objects.

(b) Latency of updates. YCSB Uniform 64-byte objects.

Figure 33: Latency for PUT between Nibble and Nibble-2 for a single thread running without contention at low utilization. Insert behavior for PUT shows no difference in performance, as both systems behave the same (append). Update for Nibble-2 shows roughly a 25% reduction in latency for a PUT of 64-byte objects to local memory.

6.4 Summary

Compaction of memory is an important tool for reclaiming memory holes, otherwise unused, potentially contributing to higher consumption of memory. In our log-structured key-value store, we demonstrate that parallelism of compaction operations and appropriate size parameters for block and segments have the greatest influence on overall performance when memory capacity utilization is high[1].

For application environments that do not require interaction with asynchronous hardware reading

[1]Appropriate segment selection is also very important in determining overall compaction overheads, and significant effort in understanding this has been published in prior work [151, 149].

and writing memory to log memory, one may elect to relax the requirement that log memory remain immutable. Doing so may provide some gains in overall performance, especially for write-heavy workloads under high memory capacity utilization. Doing so still retains the memory efficiency, due to compaction of objects either inserted or deleted from the system.

CHAPTER VII

SYSTEMS CHALLENGES WITH IN-MEMORY COMPUTING

In these next set of chapters, we dive into the system software and explore existing limitations for building large-memory software, identifying opportunities for improvement. We follow with a proposed design and implementation of an operating system that presents virtual memory address space APIs to processes as a building block for working efficiently with, manipulating, and sharing in-memory data, beyond the use of key-value stores discussed earlier.

Some of the challenges currently unmet by state-of-the-art or legacy programming methods for managing and organizing large data machines leave software to overcome them on their own. Without proper support from system software, higher-level software designs result in inefficient execution at best. We outline these challenges next.

7.1 Memory Sizes > Address Bits

The first and most apparent challenge is based on our observation that main memory sizes will grow beyond the addressing capacity of virtual memory hardware in modern CPUs. Commodity system software, such as operating system kernels, do not provide higher-level software with the means of easily managing and manipulating data sizes beyond what can be mapped into an address space.

It is curious to notice that many features typically associated with a supercomputer, such as custom networks and large scales, are becoming available and prevalent in data center machines not just used for high-performance computing.

Where is all this memory coming from? A few combined trends are leading to machines with greater memory capacities:

(1) Custom-designed ASICs allow for greater numbers of socket configurations, enabling individual machines to be provisioned with more DRAM overall, as can be seen in Table X in the Motivation Section. The Ultraviolet™ product line from SGI, for example, supports immense configurations of

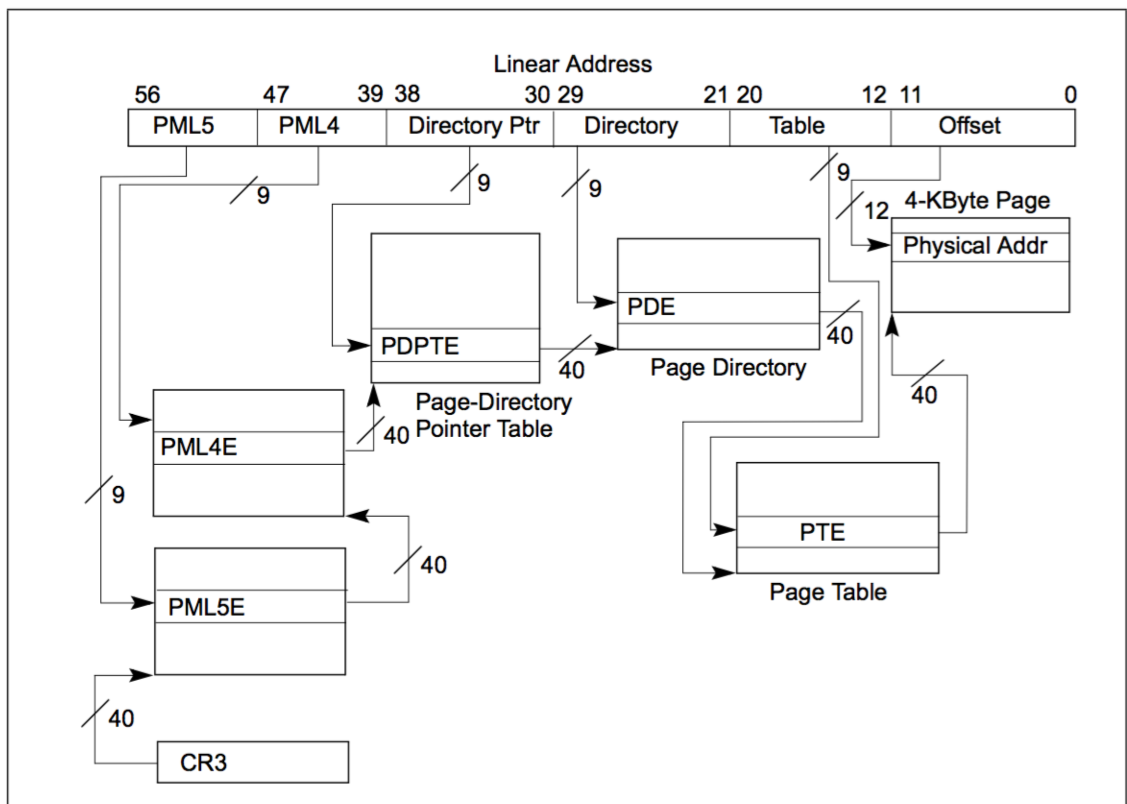


Figure 34: Linear-address translation using 5-level paging. From Intel's white paper *5-Level Paging and 5-Level EPT*, document number 335252-002, revision 1.1 [12].

up to 256 processors and a total of 64 TiB of DRAM[1], and HPE's Integrity Superdome X machines support up to 16 processors, each with 3 TiB of DRAM, for a total of 48 TiB. Custom interconnects developed by the respective system manufacturers (implemented within these ASICs) enables so many processors to be joined into a cache-coherent network to function as a single machine; SGI has NUMAlink® and HPE has their own chipsets to do the same, sx3000 and XNC2 (shown in Figure X in the Motivation Section). It further enables a large amount of memory bandwidth to be made available, from the cores within each socket to the locally attached memory.

(2) New memory technologies are coming to market that are more dense than DRAM, such as PCM, STT-RAM, Memristor, and current technologies such as Intel's 3DXPoint. Multi-level cells further allow a single memory cell to encode two or more bits, enabling additional density. Future machines are poised to include a multitude of such memories together; notable examples include the "Summit" supercomputer developed jointly by IBM and NVIDIA, to be released in 2017, and HPE's "The Machine" concept machine. The former will include half a terabyte of DRAM and nearly one terabyte of non-volatile memory, with individual machines communicating over an InfiniBand interconnect. The latter, in contrast, departs from the traditional "shared-nothing" architecture, and will use the newly introduced Gen-Z network fabric to connect all non-volatile memories as a single byte-addressable pool of persistent memory, projected to be an aggregate size in the tens to hundreds of petabytes (see Figure below).

The typical data center machine will likely have a deep hierarchy of memories and storage, squashing non-volatile memories between DRAM and some form of non-byte-addressable persistent storage, such as flash memory (SSD) or magnetic storage (HDD). Thus, the treatment of such memories will become more like typical memory heaps, rather than through other intermediary interfaces, such as via file systems.

There is interest for use of such technologies in the low-power and embedded space, too. As DRAM becomes increasingly costly to drive as capacities grow (in refresh energy), non-volatile memories become attractive for these devices, such as sensors, as non-volatile memories require no power when not in engaged. While the total capacities on embedded or low-power machines may not be as large as found on data center machines, they nevertheless will benefit from a large increase in

[1]Notably, more memory is likely possible, but Intel CPUs can only address 2^{46} bytes, currently.

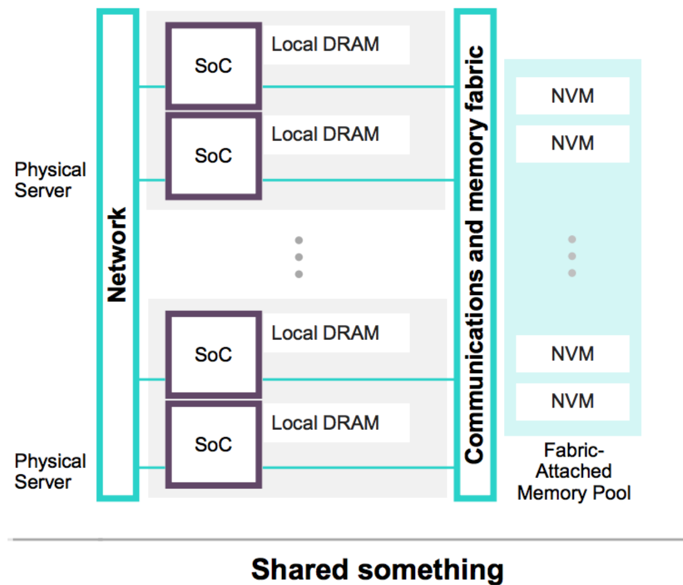


Figure 35: Diagram of The Machine as a memory-centric system, reproduced from [99].

overall memory due to the use of non-volatile memory.

(3) In addition to the ASICs which connect many processing sockets together in one cache-coherent network, new memory interconnects are providing access to "distant" memory directly onto the memory bus, reachable from processor load/store ops. One such recent memory fabric is Gen-Z, mentioned above. The insight here is that even when memory does not fit per-se into an isolated machine, it is made accessible to look local to the processor by hooking into the off-chip network, without a software network or storage stack to provide abstractions on top. Such a capability enables load/store domains to grow with the overall size of the system itself; an individual processor can directly address a single arbitrary byte across an entire array of machines. Operating systems and applications can directly "map in" such memory and leverage the processor-local memory-management unit (MMU) to virtualize it. Not all memory is made cache-coherent, however, as that would present an undesired scalability bottleneck, thus software must be explicitly written to handle shared-memory situations to maintain correctness. According to draft specifications, the Gen-Z protocol can encode memory addresses at a full 64 bits per component (264 bytes or 8,192 pebibytes); within a subdomain, theoretically up to 276 bytes are addressable.

With memory capacities increasing, software is beginning a shift to treat memory as the primary location for both short and long-lived data, instead of persistent storage devices like Flash and

magnetic devices that are typically hidden behind many software layers (e.g., device drivers, file systems, etc.). The most important capability in facilitating this shift is the ability to address all such memory with as little undue burden as possible.

Can we access all this memory? While new memory fabrics such as Gen-Z are designed to support very large address ranges, processors and software must still be programmed to take advantage of it. Virtual memory enables software to access physical memory, and the translation thereof is under control of the processor. As an example, Intel's 64-bit x86 CPUs support 48 bits of virtual memory, and 46 bits of physical — 256 TiB and 64 TiB, respectively. Translation is configured via hardware-defined radix-tree page tables (on x86-based processors) initialized by system software which is walked by the CPU on page access translation misses, and cached in translation look-aside buffers (TLBs).

While it is beyond the scope of this dissertation to review the addressing capabilities of all modern processors, past and present, we will discuss what steps some processor manufacturers are taking. Processors have had a long history in continuously expanding their virtual memory support to address larger and larger physical memory capacities, for example, earlier 32-bit CPUs addressed memories larger than 4 GiB using PAE that provided a few additional addressing bits, then the shift to 64-bit computing provided up to 48 bits in virtual memory which we have today.

More recently, Intel released a white paper [12] on 5-level paging (thus indicative of future processor specifications) to suggest virtual memory page tables will include a fifth level of the translation tree; support for this is already being coded into the Linux kernel[2]. See above figure from a white paper published by Intel for the translation structure; according to the white paper, future processors may include support for 57 bits of virtual memory, and up to 52 bits for physical - 256 petabytes and 4 petabytes, respectively. Intel has, in the past, introduced other extensions to their products that expand the addressing capabilities of hardware in times where physical memory was quickly becoming abundant, such as Physical Address Extensions (PAE) for the 32-bit Pentium processors to access more than 4 GiB of DRAM.

ARM-based and other embedded or low-power processors are also finding their way into the data center space [28, 117, 178], with ARM having recently introduced support for cache-coherent SMP

[2] `https://lwn.net/Articles/717293/`

capabilities and 64-bit processing, among other capabilities. The ARMv8-A architecture is recent, and supports 48-bit virtual addresses (max of 256 TiB). Lower-power processors are becoming increasingly attractive in data center servers, with HPE advertising their use to support their current "The Machine" server architecture. It is this environment, however, where large physical memory spaces will become accessible to such processors.

It is not apparent whether AMD will consider supporting larger physical address spaces in future installments of their processors; current specification manuals state there is still 4-level translation hierarchy, which is limited to 48 bits of virtual address. 64-bit virtual addresses are, however, translated to 52-bit physical addresses, on the AMD64 architecture, though implementations may support less.

Thus, we can see that support for large physical memories is scattered among modern hardware. Why do processor manufacturers not simply fully enable 64-bit virtual and physical addressing from the start? Virtual memory is expensive, thus making it efficient is necessary else nearly all applications will spend their time stalling in the CPU, waiting for page walker mechanisms to translate load and store operations. Even on modern processors, certain applications with random access patterns can cause 50-80% of time spent in the program to be wasted on page table walks [32]. With tree structures representing virtual memory (used in x86-based Intel and AMD processors), more address bits means more levels to the tree, thus additional memory fetches by the page walkers to traverse the tree. Hardware caches storing parts of the page table in buffers, such as the TLB, would need to grow [40]. Being fully-associative buffers, lookup latencies bound their capacity and take up more die space; they must be fast, as physical addresses are used to index L2 caches, occurring in parallel with L1 cache misses. Processor caches that hold address tags would also grow; more address bits means more bits in the tag store. Some architectures provide alternative designs to radix-tree address translation, such as the IBM POWER9 architecture, which allows the use of hash tables for address translation (called Hashed Page Table Translation, or HPT), currently supporting virtual addresses with 78 bits [18].

The general-purpose processor is the bottleneck in enabling address translation to physical memory; fabrics such as Gen-Z or the other underlying ASICs connecting such processors discussed

earlier support physical memory capacities beyond the capabilities of the processors they connect. To reiterate an earlier example, Gen-Z can support up to 264 bytes of addressable memory; SGI systems cannot provide more memory until their main processor manufacturer — Intel — increases their physical address sizes[3].

7.2 Limited control over in-memory data organization

We next discuss multiple aspects of managing virtual address spaces that motivate this work, illustrated in Figure 36.

Maintaining data in its in-memory representation is problematic. Data structures provide a natural means with which applications interact with information. Many of these are facilitated through the use of pointers, including hash tables, trees, and lists. The values of the set of pointers within a data structure defines its mapping range: the region of memory the data using these pointers must be allowed to be mapped into, in order to be dereferenced safely. In other words, it is a contiguous range of virtual addresses specified at some base address and a given length. If a data structure is not mapped into virtual memory within an area containing its mapping range, some or all of the pointers become dangling, leading to undefined behavior at runtime, such as process termination, or worse, corrupt data.

This obvious and subtle requirement, we argue[4], motivates programmers to use radically different secondary formats to represent their data. In scenarios of data persistence, in-memory data structures are marshaled into linear sequences suitable for writing to files, or pushing out over the network. Examples of this include indexes used to refer to data maintained within logs [151], messaging and container formats like Protobuf [16], and intricate software tools optimized for querying large complex datasets, such as with SAMtools in the field of bioinformatics.

Why not persist or share data using the natural in-memory representation? One reason is an understanding that the underlying system software does not provide guarantees to higher-level software about the availability of ranges of virtual memory for mapping such data. Certain system policies put in place may impact the location returned by the system call used for creating new

[3]`https://www.enterprisetech.com/2014/03/12/sgi-reveals-numalink-7-plans-data-intensive-computing/`
[4]Discounting other reasons, such as byte endianness, if used across machines.

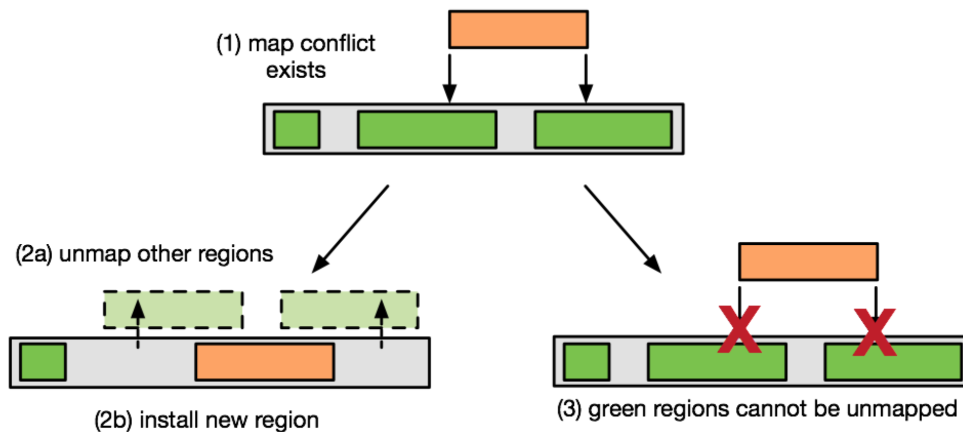


Figure 36: Figure illustrating how to access an alternate memory region in (1) that either (a) does not fit, or (b) conflicts with one or more existing regions (e.g., if the new region requires a specific base virtual address). (2a)-(2b) illustrate unmapping the conflicting regions; (3) if they cannot be unmapped (e.g., the application may have stored references into such regions, which we cannot track), then the new region cannot be mapped at the required location (e.g., otherwise pointers must be swizzled).

mappings — mmap(2) — one notable being address space layout randomization (ASLR), which attempts to avoid any determinism in selection of new memory areas. One cannot request from the operating system that certain regions of memory be "reserved" for specific applications or data sets; libraries and runtimes simply attempt multiple times to map the region of interest until it is acquired. New programming models that are targeted at working with persistent pointer-rich data structures, such as Atlas, are forced to do just this. Certain regions of memory are predetermined by the linker after a program is compiled to binary object files — code, global data, and constants — which may conflict with address ranges needed. Not being able to acquire the memory ranges needed can become worse over time: after a data structure is created, activities such as recompilation of source code, linking to additional libraries, and ordering of operations elsewhere in the program as it gains features over time may conflict with existing data structures.

A further argument why address ranges may be unavailable for mapping such data sets is that, as a process ages, its virtual memory map may become fragmented. Various entities that support process execution require support from the operating system to create virtual memory mappings, and do so without any sort of coordination with the OS or other software about where such mappings must necessarily occur. The dynamic linker, which typically executes immediately after a program

is loaded but before the main routine is invoked, will create initial mappings; some programs may continue to dynamically load libraries throughout execution, depending on functionality required. Certain heap allocator libraries may request from the operating system additional large regions from which to allocate, when an application's working set increases; these are also allocated with mmap(2). Other influences include certain libraries that may pre-allocate large regions of memory in advance of use, such as for device memory-mapped I/O (e.g., GPGPUs), or to pin physical memory (e.g., memory used by network device drivers like InfiniBand) — some of which are mapped to specific offsets, others not.

Legacy data-sharing semantics are complex, ill-matched, and expensive – files, files, files. How do processes share physical memory pages? Implementing persistent-memory interfaces requires acute understanding of the intricacies of the underlying APIs and system calls [46, 165, 63, 64].

Allocating (swappable) memory within a process is relatively straight-forward: use malloc(3), or mmap(2) for "anonymous" memory, then touch the memory to force the operating system to allocate physical memory backing the new address range. If one wishes to share these physical pages with another process, memory derived from use of these interfaces is not easily possible; sharing pointers makes no sense, as the receiving process would need appropriate translations installed in its page table to the same physical memory referenced by the originating process. The pseudo-file-system /proc enables a process to identify the physical page frames that have been allocated to a virtual memory range, but another process does not have any method to "identify" these pages to the operating system for mapping into its address space (doing so arbitrarily would be dangerous).

Instead, one must resort to the file system. A process creates a file somewhere, maps the file in with mmap(2) and touches the returned address range to have physical pages allocated. To avoid any undesirable interactions with actual disks, one must use an memory-only file system, such as tempfs; doing so requires correct choice of path, too, which varies from system to system (/tmp is not always backed by tempfs). The path itself is communicated to another process, which in turn opens, and then maps in the data. As mentioned before, mapping in data results in an address that we do not have control over, as invoking mmap(2) returns a randomized address (in Linux, or any system with ASLR enabled).

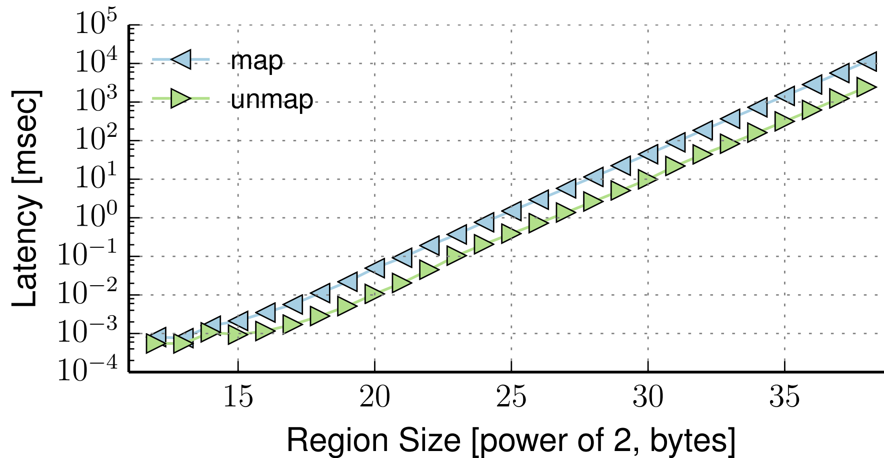


Figure 37: Cost of creating a mapping in a page table with 4-KiB pages, as a function of the size of the region. Measured on Linux.

mmap(2) does allow the caller to specify an address the kernel should attempt to use for the new mapping. Specifying an address to map in is seen as a hint to the kernel; if an existing mapping is found, it proceeds to find another available address range, and completes the request. Thus a caller would need to compare the desired address with the returned to determine if the hint was obeyed. Unfortunately, this interface is broken, as behaviors vary from system to system; in Linux, there is no way to require mmap to either map in at the requested address, or fail. Specifying the parameter MAP_FIXED indicates a specific address must be used. The manpage clearly states, however, what happens if you do –

> `If the memory region specified by addr and len overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded.`

Thus, the kernel will silently remove any existing mapping without notifying the caller that it had done so, resulting in possible process termination or corrupt data. In FreeBSD, use of MAP_FIXED together with MAP_EXCL forces mmap(2) to fail should there be a conflict with an existing mapping. In Linux, if one wanted to verify if any existing mappings existed, it would have to write code to parse entries in /proc; a race condition nevertheless will exist, between it scanning mapping entries, and requesting a new mapping with MAP_FIXED — unsafe!

Mapping costs are non-trivial. Creating mappings in a process page table to existing physical pages is not free. In addition to the fixed cost of invoking a system call, there are costs linear in time in the size of the memory to map in; each page requires a page table entry to initialize or update. We measured this cost using the above method for sharing memory between processes — created a file in a memory-based file system, allocated a certain number of pages — and then mapped in the file, measuring the latency. This methodology eliminates page allocation costs and directly measures page table initialization overhead. (The figure above is reproduced from earlier in the dissertation.) A region of 238 bytes, or 256 GiB, requires on the order of ten seconds to map in — to existing memory — using 4 KiB page sizes. Use of larger page sizes shifts these measurements to the right, but remains linear.

There are a few challenges here. Large memory regions will be expensive to implement translations to, even with larger page sizes, impacting startup costs for a program (e.g., important for checkpoint/restarting of processes, or recovery costs from failure, etc.). Should a process be forced to alternate between conflicting mappings, these costs will become compounded throughout execution. Multi-threaded programs that aim to invoke mmap(2) in parallel on disjoint memory regions will be serialized by the operating system, as doing so mutates the single page table instance as well as the virtual memory area (vma) data structure representing it in kernel memory. Prior work proposed a design to accomplish parallel mmap(2) in multi-threaded programs, albeit on a minimal kernel and not a commodity enterprise operating system [**?**]. mmap(2) is not an efficient tool for dynamic modifications to the address space.

Sharing granularities differ. Processes that share data may not wish to do so with full permissions to the entire range of memory for all participants. Let us briefly discuss a scenario where this already occurs, but for a single process using private memory — loading ELF binaries, such as executables or shared libraries. When the operating system loads a binary (or dynamic linker loads a shared library) ELF headers are parsed to understand the individual sections within the file, e.g., global data, executable code, and zero-initialized data. Each of these would be mapped into the process address space with different permissions: globals would be mapped in with read/write permissions in the page table, and executable code mapped in with read/execute permissions. The dynamic linker can

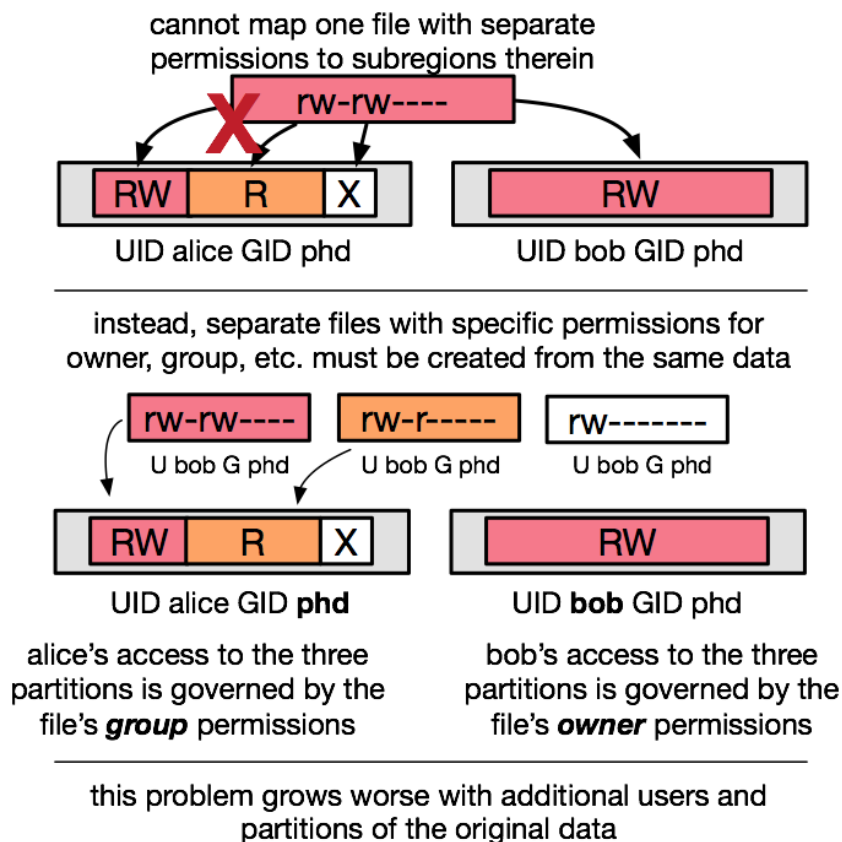


Figure 38: Figure illustrating how sharing in-memory data among multiple processes with varying access rights to disjoint subsets of the data. An individual file is insufficient to express relationships such as these, and programmers must use system administration access policies on multiple files to accurately express and enforce them.

also be invoked after a process has already begun execution by the program calling the dlopen(3) routine. The code in ld.so (i.e., the dynamic linker, which is itself a shared library executing in user-space) invokes system calls to open shared library files, searching appropriate paths in the file system; the program itself could do this just the same, thus the linker is meant in some form as a set of convenience routines. There is nothing that would prevent a process from breaking convention and mapping in sections within the ELF binary in unexpected ways, such as loading code as read/write, mutating it, and updating page access permissions to read/execute. While access to libraries are shared among processes that depend on them, physical memory representing them is typically private (except for read-only areas, which are shared).

Why do we care? If we change the scenario *such that the physical pages backing some shared*

data must be the same (i.e., no mappings created to private copies of the memory), then we see the challenge in using the file system to accomplish our goal. In the Figure, two users — Alice and Bob — need access to a common in-memory data set, which is contained in a single file (e.g., kept within tempfs). If the file's group is 'phd' then both Alice and Bob would be able to establish a shared mapping of the file with read/write permissions, but the desired configuration is for Alice to have read/write permissions to only the top portion of data, no access to the bottom, and read-only for the remainder of the data. Since one section for Alice requires greater privileges than the rest (with read/write), we cannot use a single permission specification in the file to restrict subregions to a single user like this. Instead, the original data must be broken up into separate files, the boundaries of each defined by the granularity of access for each user; this allows us to specify different permissions for Alice, using the same group. What if there are more users with disjoint subregions of varying permissions? Or, if two users must be able to access the same offsets in the data but with conflicting permissions? Basic file permissions only have "user", "group", and "global" to tweak. More complex permission attributes, such as access control lists (ACLs) would be required. What if, after two users have started using such data, another comes along whose required mappings are not possible to accomplish with the given configuration? The system administrator would also need to become involved, to create new users, groups, etc. Use of files and system administration to enforce fine-granular access control to in-memory data does not make sense. On top of these exist all the aforementioned challenges that stem from lack of control over the address space layout.

More appropriately, we should express memory and accesses using interfaces that directly associate with memory, instead of translating intents between multiple security models. Some operating systems have models to do this, such as Barrelfish or seL4, where the concept of *capabilities* are used to grant access to system resources including physical memory pages, not *users*.

7.3 Summary

User-level software face many challenges when attempting to efficiently program large-memory platforms, especially in accomodating very large individual data sets into their address spaces. Hardware may not have sufficient translation bits given access to large quantities of memory, both volatile and non-volatile. The operating system limits control over what is essentially still an invisible

resource managed entirely by the kernel – virtual memory – and what control is visible are equivalent to ‘poking holes’ into the interfaces. Persistent and sharing of pointer-based data structures requires fine control, acquiring the base address needed by the data, which may not be available. Even so, setting up access to such large data sets is expensive and not scalable, as it requires creating new translations.

In the following chapter, we introduce our solution for addressing these challenges: explicitly exposing virtual memory abstractions by commodity industry operating systems to better express, manipulate, and share in-memory data.

CHAPTER VIII

ADDRESS SPACES - NEW ABSTRACTIONS FOR MEMORY

We introduce a new set of interfaces for commodity operating systems and a design for the memory subsystem to make available for user-level software to use. The key contribution is the introduction of the use of multiple virtual address spaces accessible by individual threads of a process to allocate, create, and switch between at runtime. To facilitate this, we specify a new design for the memory subsystem that associates with a process a set of address spaces[1] – each with its own page table – and new system calls that enable processes to explicitly create new address spaces with varying memory regions in them. To make alternate memory mappings available to a thread, instead of mutating the current page table in use, a thread may request from the kernel it switch the address space (and thus the current page table), enabling memory regions to become very quickly activated and deactivated, or *swapped*.

How does this ability solve the aforementioned challenges? We briefly enumerate each here and discuss how the new design alleviates symptoms otherwise introduced.

(1) If a process is unable to create a new mapping to some area of physical memory, either due to severe fragmentation of its virtual address space, or because the amount of memory to be mapped is simply too large to place in the available virtual memory, a process can instantiate a second address space with some or none of the mappings of the current address space and map all or a portion of the new memory in. The process (or specific thread therein) switches its context to the secondary address space and may thus access the new memory region directly. There is no theoretical limit to the number of address spaces a process may create; limits are subject to available memory required to back page tables, and any implementation particulars of the operating system.

(2) Pointer-based data structures and the sharing thereof is no longer subject to the whim of ASLR or heap allocators creating conflicting regions: a process that requires a specific range of virtual memory to be available may create a new address space to accommodate the region, and maps

[1]The term “address space” is used synonymously with “virtual address space” in this dissertation.

in auxiliary regions to support interpretation of the new memory, and instructs the kernel to switch into the address space. As we show later, switching an address space has a much lower cost than modifying the mappings for an individual address space (ca. 450 nanoseconds vs milliseconds to seconds or greater). Multi-threaded programs which may require frequent access to new regions of memory unknown in advance — and thus make frequent use of mmap(2) — may each be placed into separate address spaces. As mmap(2) mutates the current address space in use by a thread, the resulting kernel data structures describing the memory may be accessed without serialization across operations performed by threads.

(3) Sharing of data is more explicitly implemented and with lower runtime overheads. Programs can share data by switching into address spaces that share the same mappings to a common data set in memory (separate address spaces with process-specific mappings kept private in each for that process to use). Doing so is facilitated by naming an address space, and requesting from the operating system to attach to it before switching (more details in the following section). Implementing fine-granularity access permissions can be done with access control lists associated with specific regions of memory, also presented as part of the interface we are introducing; no need to introduce the system administrator.

In the next section, we dive into the details of the design and a reference implementation of this idea in a common operating system — Dragonfly BSD.

8.1 SpaceJMP - an Operating System Model

Two main abstractions are introduced and defined: a memory segment and a virtual address space. We then provide a new definition for the process model in an operating system that implements our model, and detail a reference implementation in DragonFly BSD. The API we expose to user-level software is summarized in Figure 40.

Clarification note — In this dissertation, when we state 'a process switches into an address space', we mean more specifically that any thread of said process may switch, and do not imply that a process is single-threaded, nor that all threads in said process must or do switch in unison to some common address space. Processes may be single- or multi-threaded, and threads may exist in disjoint address spaces at any time during execution. Some definitions in this document go beyond what is

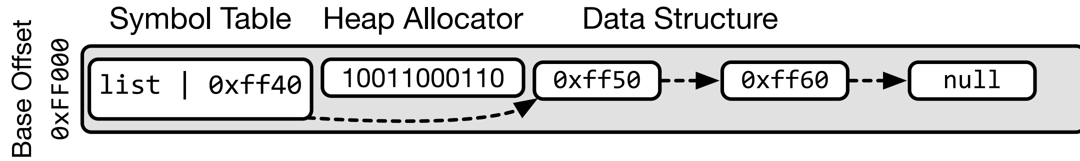


Figure 39: Example segment. Segments are a contiguous region of virtual memory, composed of physical memory pages, with a base address and a length. The segment shown contains a symbol table and heap allocator context that a process may use to create and manipulate memory-resident data structures, e.g. a linked list.

VAS API	**Segment API**
`vas_find(name) → vid`	`seg_find(name) → sid`
`vas_detach(vh)`	`seg_attach(vh, sid)`
`vas_create(name, perms) → vid`	`seg_detach(vh, sid)`
`vas_clone(vid) → vid`	`seg_ctl(sid, cmd[, args])`
`vas_attach(vid) → vh`	`seg_attach(vid, sid)`
`vas_switch(vh)`	`seg_detach(vid, sid)`
`vas_ctl(cmd, vid[, args])`	`seg_clone(sid) → sid`
	`seg_alloc(name,base,size,perms) → sid`

```
vid:  global VAS identifier
sid:  global segment identifier
vh:  process VAS handle (attached)
```

Figure 40: API in SpaceJMP for managing address spaces and memory segments.

described in the published paper [65], to enhance clarification that was otherwise omitted due to page limit brevity.

Memory Segments. A segment, shown in Figure 39, is a collection of physical memory pages, a base virtual address, and a length in bytes. The physical pages need not be contiguous, but the virtual memory mapping it must be. Within it, software may store what is typically used by application code — data, stack, heap, code, or a combination thereof. Information describing how such memory should be accessed, such as the set of subregions and access rights, owner, and a name that is assigned to it, allowing other processes a means of identifying it (similar to how the path is used to identify a file). Application software is giving a new API (described later) to be able to explicitly allocate a new segment with a given size, name, and base address. Other processes may query the kernel for the existence of such segments; access to the memory of a segment is guarded by who may include a segment in their address space, done only so via a memory map. Note the use of the term ‘segment’ is dissimilar to its use earlier.

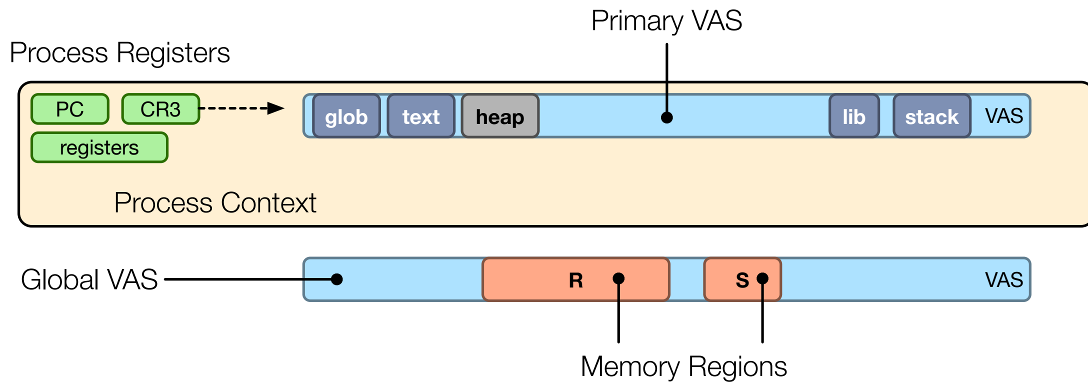
Segments may be instantiated as *lockable*. When done so, the address space that includes it will obey switching semantics that follow the locking requirements, using a reader-writer lock or similar, enforced within the kernel during an active switch. If a segment is in an address space as read-only and a process switches into it, the kernel will acquire a reader instance on the lock; if the segment is mapped as writeable in any way, the kernel will acquire an exclusive writer instance on the lock. Upon a switch by the application, should a lock instance not be available (e.g., acquiring a read when a writer is active), the kernel may block the invoking application thread inside the switch system call, or return a status code indicating the switch was not performed.

Segments are given names by the creating process and are global resources, visible to other processes. We introduce an API for processes to create segments, and for them to search or query existing segments that other processes may have previously created. Similar to the relationship between files and processes, segments have a name, access rights, a length, and do not disappear when the creating process terminates. In our current design, physical memory backing a segment is allocated at segment creation and is not swappable. Once created, segments can be attached to existing named address spaces (discussed below), making their data they map to available for other processes to access.

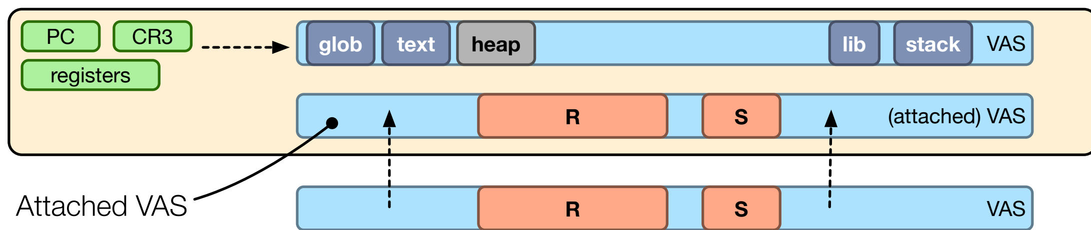
Process Model and Virtual Address Spaces. A 'virtual address space' as used in the SpaceJMP model is not unlike the conventional definitions from modern operating systems. It contains a mapping from a virtual memory space to a physical memory space, with each page containing specific access rights of read, write, or execute, and is active for a thread when it is used as the current translation during execution. The differentiation here is that virtual address spaces are treated like first-class citizens, much like files, or other abstractions created for processes to take advantage of. We give user-level software the ability to explicitly create new address spaces (not bound to any given process, unlike forking), to associate them with segments, and for a single process to be able to change its address space arbitrarily — coined as address space switching.

In SpaceJMP, there are three kinds of address spaces:

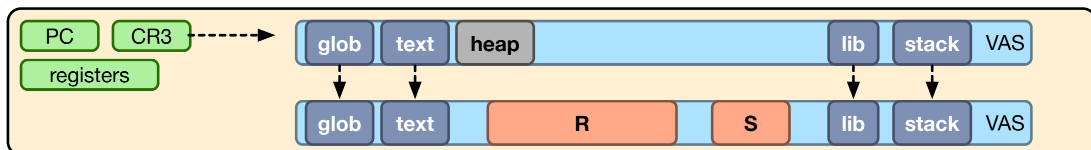
- *Primary, or private,* address space. When a process is created, this address space is given to it to be able to being execution. It is the same as in the traditional UNIX model; it contains the process binary information, heaps, stacks, kernel translations, and anything else that might be



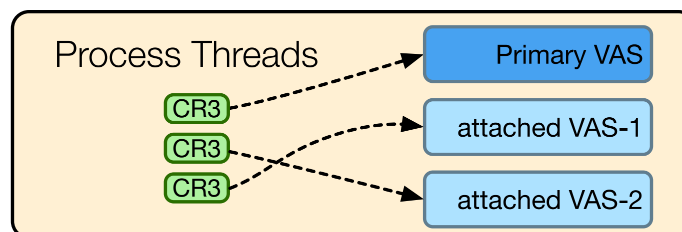
(a) A process with its primary virtual address space, holding global variables, code, and heap regions, mapped shared libraries, and a thread stack. Below, a global address space, accessible by name, containing two global segments – R and S.



(b) *Attaching* to a global address space is necessary before a process may access the memory within it. Doing so, a private version of the address space is created, with a copy of the page table.



(c) The last step of attaching requires also copying in process-specific mappings, such as code and stack regions that would allow a process to continue execution within the attached address space.



(d) Simplified view of a multi-threaded process and multiple attached address spaces. Each thread has its own register state. Switching address spaces involves modifying CR3 to point to the page table of the associated address space. Threads may switch arbitrarily and independently.

Figure 41: Examples of virtual address space types and their uses by a process.

mapped in. The primary address space and all resources are released upon process termination. No process can see, query, attach, or switch into another primary address space.

- *Named, or global,* address space. New address spaces created with our new API are automatically made into named address spaces. They are called ‘named’ as they are given a textual, human-readable identifier that processes may use to query the kernel to obtain information about it, and to request attachment to. Processes never execute within a named address space, as they are meant to contain segments and mappings that are process-agnostic — to represent a set of memory resources available for a set of processes, or processes over time, to access and manipulate.

- *Shadow, or attached,* address space. Processes that wish to access memory mapped by a named address space must first attach to it. The kernel will create a copy of the named address space, private to the invoking process, to exist as a shadow of the named address space. Memory regions that are specific to the process context will be added in, such as its code, stack, and data. This last step makes it possible for a thread to resume execution after switching, as its code (and kernel routines) would otherwise not be accessible. It is a shadow address space because it represents the state of the named address space, merged with a subset of the context from the process that allow it to execute within it.

The process model in SpaceJMP is thus expanded to consist of a primary address space and a set of shadow address spaces, between all of which, including the primary, threads of a process may switch. Illustration of address spaces, segments, and their use by a process are shown in Figure 41.

Why would we require a separate procedure of “attaching” to an address space before using it? Because an address space contains no translations when first created, or if it only contained mapped-in segments pointing to application data, switching into it would cause a runtime abort: depending on how the switch procedure is implemented, a subsequent instruction fetch may cause a page access violation, potentially in the kernel if implemented as a system call. Beyond this minute aspect, attaching allows us to install translations to data and code that make it possible for threads in a process to continue to access process-specific state, such as any dynamic libraries, heap allocations, shared synchronization variables, and of course thread stacks themselves.

Programming Interface. A pseudo-interface is illustrated in Figure 40 for programming address spaces and memory segments in a SpaceJMP OS. We next outline an example use-case to demonstrate how one would use this API.

How can two processes share some common data, when the virtual address range conflicts with another region in a process? Process "Alice" creates a new segment named "data", maps it into its active address space (the primary) and initializes data structures within. It then detaches this segment from its active address space, and adds it to a newly created named address space, called "shared". Process "Bob" then locates and attaches to the named address space, and switches into it, activating the mapping to the common physical pages holding the required data. We illustrate this process in Figures 42 and 43.

This method differs from both processes simply mapping in a shared memory region by:

- avoiding existing region conflicts
- avoiding use of the file system API
- minimizing costs otherwise incurred in activating a region for access (e.g., via mmap(2))

Note the use of separate handles in the code to differentiate between a global instance, and a shadow instance of an address space. "vid" in the pseudo-code indicates a global identifier for the named address space, and "vh" a process-private handle that associates with process-specific address space contexts; here, an attached address space. An astute reader will notice as well that as the segment "data" overlaps with some other existing region in process "Bob", some variables that may be instantiated to an address in the primary address space *should not* be dereferenced once switched into address space "shared", as they either are no longer active (access permissions may have changed, as the region is not available in "shared") or actually point to data within the segment, and vice versa! This situation is what we call a *confused dangling pointer* because while the pointer may still be active, it may point to completely different physical memory and dereferencing it may result in corruption of data. Our original paper reviews some methods in the compiler that attempt to mitigate these situations.

Ideally, not all application software would meddle with the entire interface. Libraries, runtimes, and "expert programmers" may wish to explicitly manage segments, but we expect general application

```
const long BASE   = 0xFFFF000;
const size_t SIZE = 1 << 30;
sid_t sid = seg_alloc(‘‘data’’, base, size READ|WRITE);

seg_attach(CURRENT, sid);
/* initialize data at 0xFFFF000 */
seg_detach(CURRENT, sid);

vid_t vid = vas_create(‘‘shared’’, READ|WRITE);
seg_attach(vid, sid);
/* give ‘‘shared’’ string to others */
```

Figure 42: Process Alice creating a memory segment with shared data for other processes to attach to.

```
/* Process cannot attach segment to current active
 * address space (primary), as base address conflicts with
 * another mapped region. So, we attach to the address space,
 * and provide a mapping for the segment in a new area.
 */
vid_t vid = vas_find(‘‘shared’’);
vh_t vh   = vas_attach(vid);
vas_switch(vh);

/* now segment is accessible, avoiding conflict of other region */
const long BASE = 0xFFFF000;
struct data *d = (struct data*)BASE;
/* use ‘d’ */

vas_switch(PRIMARY);
/* ‘d’ no longer points to data */
```

Figure 43: Process Bob attaching to the “shared” named address space and accessing the common data within.

code to either (a) only invoke address space methods, or (b) never invoke any of the provided API, leaving all the work to underlying libraries that perform attachment and switching automatically.

Note that "permissions" as illustrated in the pseudo-interface are loosely defined. Given this is concept work, we have not explored exactly how to implement or represent a permissions model for segments, but expect they may look similar to what is used for files, such as access control lists.

Heap allocation. Assuming a single heap exists universally is the de facto approach to dynamic memory allocation, with applications, all libraries, and threads sharing the same heap and using the same interface – malloc(3). In a model where the virtual address may change arbitrarily, dynamic memory allocators would need to be cognizant of the region(s) in which they allocate memory from. For example, if an application and a library maintain their own disjoint memory segments for heap allocation (inaccessible to either), then implementations for methods such as malloc(3) must be given some context as to the pool of memory they must operate on, while in the same address space. If switching between address spaces when crossing the library interface, we must understand that a call to malloc(3) may or will allocate memory resident within a single address space, and potentially cannot be accessed unless within its address space of origination.

8.2 Summary

SpaceJMP is an example model for the memory subsystem of an operating system that exposes virtual memory abstractions directly to applications. Processes may allocate and compose address spaces and memory segments to exist perpetually in the system (much like files). Accessing an address space is performed via attaching, allowing any threads within a process to then arbitrarily switch between translations, avoiding the cost of page table modification. With these abstractions, applications can program memories larger than processors can translate, and share pointer-rich data structures without concerns over the specific layout or state of their address space at the time; making a region available would be implemented by simply creating a new address space, mapping the data into it, and switching.

In the next chapter, we review the implementation within a commodity operating system, DragonFly BSD.

CHAPTER IX

SPACEJMP - REFERENCE IMPLEMENTATION IN DRAGONFLY BSD

The SpaceJMP design was implemented in DragonFly BSD [14] v4.0.6, a scalable, minimalistic kernel — supporting only the x86_64 architecture — based originally on FreeBSD4. Modifications included the memory subsystem and memory allocation routines (as described above). In the remainder of this dissertation, we use the acronym "BSD" in place of DragonFly BSD in cases where the subsequent discussion is generally applicable to multiple BSD-based systems. Where discussion specifically pertains to DragonFly BSD, we use the more specific identifier itself.

BSD-based operating system kernels use a "VM object" model[1], derived originally from the Mach virtual memory system, as the design for their virtual memory management, including DragonFly BSD, FreeBSD, and macOS (originally Mac OS X). Within this model, a **memory object** (or **vm object**) is an abstraction used to represent some data object — e.g., a file, zeroed memory, or a device — which can be sourced for data and used to fill pages of memory. In the kernel, a memory object contains a set of physical memory pages that actively hold data pulled from the underlying data object, arranged in a tree data structure, represented by the vm_object struct. Memory objects are not specific to a process, and may be referenced by multiple processes (reference-counter-protected to ensure safe removal). For example a dynamic library (a file on disk) will be represented as a vm object in the code, and its physical pages that are meant as read-only directly shared by many applications, whereas library globals would be mapped into applications as read-only with copy-on-write (COW) to provide private writeable pages on page faults. COW modifications result in new vm objects created for that process, holding physical pages private to that process, and references to read-only or as-yet unmodified COW pages held by the derived-from vm object. This process results in a chain of vm objects, typical of the VM object model.

A SpaceJMP memory segment is simply a wrapper around a BSD vm object, with only physical pages as the backing memory (no file, or otherwise). On creation of a segment, physical memory is

[1] `https://www.freebsd.org/doc/en/articles/vm-design/vm-objects.html`

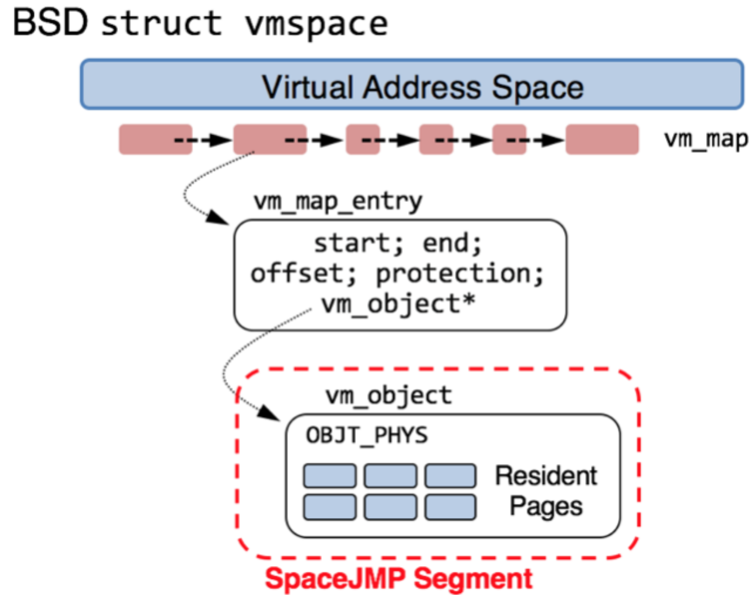


Figure 44: Illustration of a virtual address space in DragonFly BSD, VM objects. The implementation of a segment and address space in SpaceJMP are simply wrappers around each.

immediately reserved and made unswappable. In BSD-based systems, virtual memory is represented additionally by a set of region descriptors that make managing the state for individual memory mappings easier. Each region has pages provided and managed by a VM object, and it is the latter that is sourced for data upon soft page faults. Thus, implementing a SpaceJMP segment as a VM object results in minimal modifications to the virtual memory system within the kernel, as all code to create and manipulate VM objects can be leveraged directly.

Address spaces in BSD are represented by a vmspace struct and hold a set of memory region descriptors (the vm_map struct), describing each active memory region (a vm_map_entry) — with some form of permissible access rights, copy-on-write regions, memory-mapped I/O, etc. in Figure 44. Each entry encodes a start address, length, protection rights, and a reference to the associated VM object supplying it with data, as well as the offset into said object (the latter is unused for SpaceJMP segments). Similar to our implementation of a SpaceJMP segment, an address space in SpaceJMP is also a simple wrapper around a vmspace struct. Overall programming effort in DragonFly BSD was greatly supported by existing features in the kernel called 'VKERNEL' that allows executing kernels in userspace, to assist with kernel development. The feature provides interfaces for creating and manipulating address spaces directly.

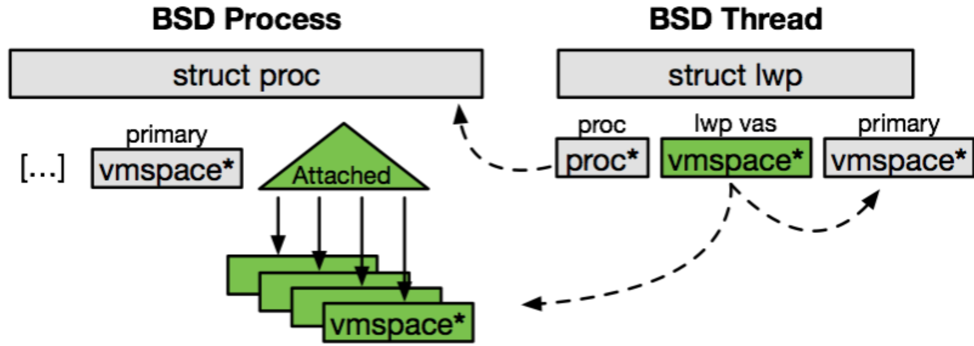


Figure 45: Figure illustrating modifications made to the BSD process and thread structures. Green shapes are additions. A process maintains a set of references to attached address spaces, and each thread an additional reference to its active address space, if not currently in the primary.

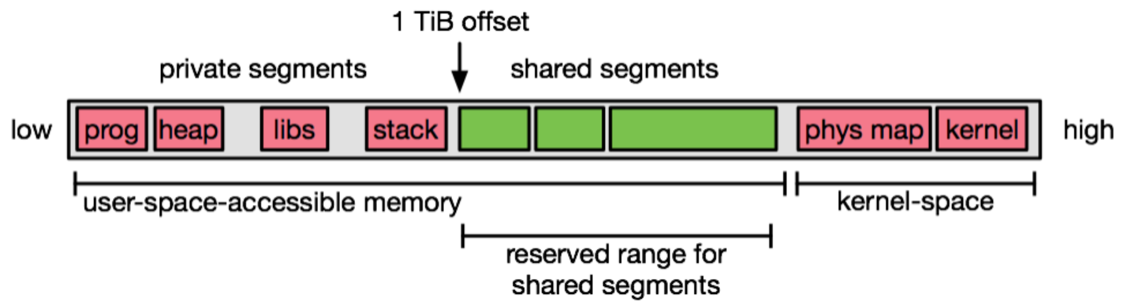


Figure 46: Diagram to show the private, shared, reserved segments.

The set of named address spaces and segments is maintained in a kernel-resident global data structure accessible from within system call implementations, similar to the global task list. We also expand the process structure to include a set of references to address spaces representing those the process has attached to, indexed by the process-local VAS handle ('vh' in the above code), similar to how an open file handle is used. When a process attaches to a named address space, the vas_attach system call will search the global set of named address spaces for the handle, clone its content (except for the physical pages), create a new local VAS handle identifier, and add it to the attached address spaces set for the process. We illustrate these changes in Figure 45.

Avoiding address collisions in SpaceJMP. Collisions may still arise from stray memory mappings or regions of a binary that cannot be relocated (non-PIC). To insure against such situations, we reserve an upper region of virtual memory in the user-space range and do not allow memory mappings or stack to be allocated within this range (green in Figure 46, not to scale). SpaceJMP

segments are strictly given addresses within this range, and non-segments are allocated to addresses outside this range. While such a static policy may not be ideal, it ensures no previously created segment can conflict with a region necessary to be mapped in during an address space attachment, which would otherwise defeat the entire purpose of SpaceJMP in the first place. Placing a static offset as the boundary dividing private and shareable segment regions does leave open the risk that the offset does not leave enough space for future sizes of private memory regions. Heap and global memory regions may at some point exceed this offset, and when that happens, we may revisit the chosen offset.

Creating a new named address space. The pseudo-code above shows how we create a new named address space in SpaceJMP. A global system setting controls whether the kernel has VAS features and functionality enabled. If it is enabled, we create a new empty address space, then clone the kernel segment mappings from the current process (they are the same for all processes). Once complete, we add it to the kernel-resident global tree for all processes to see (depending on their viewing/access rights).

Attaching to a named address space. The kernel does not understand the concept of attaching to an address space. User-level code, typically a library or runtime, will implement this functionality, first allocating an address space that is not named (shadow or process-private). User-space will then request additional process-private segments be included in the new address space, such as thread stacks, certain heaps, code, and global data using seg_attach. The new address space is thus a union of a subset of the primary address space, and the segments contained in the named address space. Threads in the process may then proceed to switch into it.

How does an address space switch happen? The user-level code for a switch is illustrated earlier. Within the kernel, a switch is handled by a system call that has a very straight-forward implementation, shown above. User-level code specifies the process VAS handle of the address space to switch into; a special value (implemented as zero) indicates the primary address space, shown in the code fragment as PRIMARY. Whenever a thread switches into a non-primary VAS, we increment the reference count on the state representing the attached VAS, and similarly decrement it when a thread switches out of one.

Isn't context-switching very expensive? *Context-switching* in the typical sense involves the

kernel swapping the entire context of what represents the thread from one process to a thread of another process — TLB translations, CPU registers, and other CPU context such as the page-table and stack. Much of this is quite costly, especially the need to flush the TLB: translations cached from the page table are stored here, and must be removed for another process (which has a different page table) to avoid corrupting otherwise safe execution, or giving a new process additional translations to physical memory it would otherwise not have access rights to touch. On a busy CPU, context switching would normally occur every processor tick — every tens of milliseconds, depending on whether the kernel is preemptible, a server or desktop configuration, etc. Each switch means the TLB is cleared, and translations must be reloaded — incurring expensive page walks of the page table. When a context switch is performed between threads of the same process, however, much of that cost is avoided: page tables are the same, thus the TLB retains its translations. Only the CPU registers, stack, program counter are modified (whatever context specifically is needed for representing just the thread itself).

In SpaceJMP, two aspects are different. First, when a thread switches its address space, a full context switch in the general sense is not performed. Instead, a VAS switch is more akin to an inter-task thread switch, where stack, CPU registers, and program counter are saved and restored. In a SpaceJMP kernel, a thread VAS switch does involve changing the page table pointed to, similar to a process context switch — CR3 is overwritten (control register 3, the register storing the address of the root of the page table) which implicitly flushes the TLB on x86-based CPUs. Second, VAS switching may occur at a higher rate than traditional process or thread context-switching, depending on the workload (how frequently data items must be accessed) and on the configuration of the segments and address spaces they are mapped into (i.e., the partitioning of the data). In our measurements of a high-performance benchmark — GUPS — we were able to measure thousands to millions of switching taking place every second. Flushing the TLB as part of changing the page table used results in an immense amount of wasted and redundant work done by the CPU in page walking and refilling the TLB.

With TLB sizes growing every processor generation, the cost of TLB flushing becomes increasingly burdensome. Thus, many architectures have support for TLB tagging, or extra state in the TLB that identifies the translation with the specific page table it originated from. The kernel will put a

tag value (or *address space identifier*) into a special register in the CPU when modifying the CR3, which the CPU associates with a translation in the TLB after performing a page walk. On performing lookups in the TLB, the current tag value set is compared to cached translations to determine which are value for the given thread context. In SpaceJMP, by default, all address spaces use tag value zero to indicate that the TLB should be flushed when CR3 is reloaded; non-zero values indicate otherwise. Use of tags does not mitigate this problem entirely, however, as it does not prevent storing duplicate translations used by a single thread across address spaces it is able to switch between. Duplicate translations reduces the effective TLB capacity, an effect that may artificially increase translations that are evicted, thus causing more page walks to occur — the problem we originally are trying to overcome by using tags in the first place. Other hardware architectures have richer controls over the behavior of and context within the TLB, such as the *domain bits* in ARMv7-a.

We examine these costs in more detail in the next chapter.

Source code, and implementation in other operating systems. At the time of this writing, the source code for the original implementation of SpaceJMP in DragonFly BSD has not yet been made open source by the host company at the time this project was undertaken — Hewlett Packard Labs (part of Hewlett Packard Enterprise).

A reimplementation of the idea has been started in the Linux kernel by Till Smejkal while an intern with Hewlett Packard Labs in 2016. He is a student at the Technische Universität Dresden, in Germany, and can be reached at `till.smejkal@tu-dresden.de` for follow-up. Patches of this effort and subsequent discussion can be found on the Linux Kernel Mailing List (LKML), list linux-mm, dated 2017-03-13 here: `https://marc.info/?l=linux-mm&m=148944326229700&w=2`

An additional implementation of the SpaceJMP model has been written for the Barrelfish [33] research operating system. It has a radically different design for its memory subsystem, using software-based capabilities from seL4. We allude to our published paper to read more details about the specific implementation. Barrelfish can be found at the website for ETH Zürich, Switzerland here: `http://www.barrelfish.org`

CHAPTER X

EVALUATION – PROGRAMMING WITH LARGE ADDRESS SPACES

In this chapter, we review the performance benefits from our implementation of SpaceJMP in DragonFly BSD.[1] We begin with an understanding of the latencies for a thread to switch address spaces (to compare with the earlier mmap(2) results). Next we evaluate a micro-benchmark to highlight the performance of frequent switching throughput using HPCC GUPS [145], a code that is targeted at large-scale in-memory concurrent updates. With GUPS we wish to demonstrate the advantages of being able to address very large memories, and the ability to switch address spaces very quickly to access such memory.

Table 5: Large-memory platforms used in this study.

Name	Memory	Processors	Freq.
M1	92 GiB	2x12c Xeon X5650	2.66 GHz
M2	256 GiB	2x10c Xeon E5-2670v2	2.50 GHz
M3	512 GiB	2x18c Xeon E5-2699v3	2.30 GHz

Table 6: Breakdown of *context switching*. Measurements on M2 in cycles. Numbers in **bold** are with the use of CPU cache TLB tagging.

Operation	DragonFly BSD		Barrelfish	
`CR3` load	130	**224**	130	**224**
system call	357	–	130	–
`vas_switch`	1127	**807**	664	**462**

10.1 Micro-benchmarks

Given the ability to switch into other address spaces at arbitrary instances, a process in SpaceJMP will cause context switching to occur at more frequent intervals than is typical, e.g., due to task rescheduling, potentially thousands of times per second. Table 6 breaks down the immediate costs imposed due to an address space switch in our DragonFly BSD implementation; a system call imposes

[1]As the work was a joint effort, I include in this dissertation the work I contributed to. Later discussion is provided covering additional results obtained from this work.

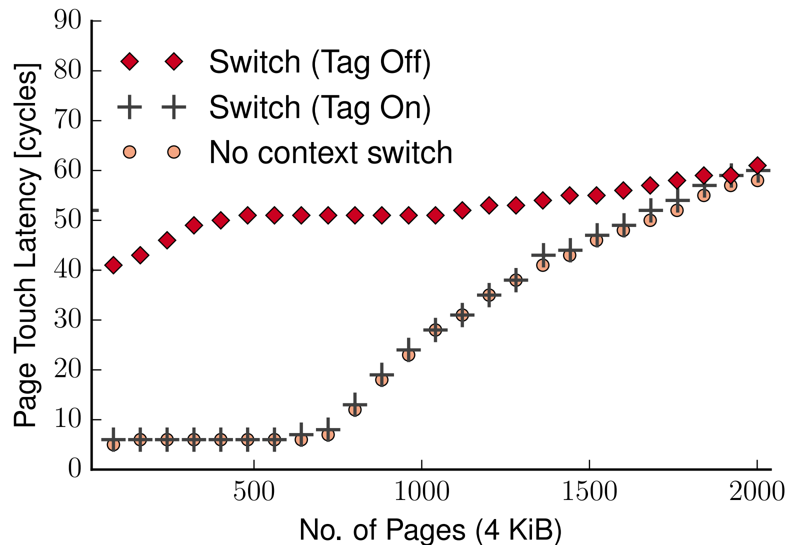


Figure 47: Latency to access one cache line within a random page between context switches – simply modifying the base address of the active translation page table in the processor, held in CR3. When CR3 is modified, the CPU will automatically (for x86-based processors) flush the TLB. With a small number of pages, the translation state will fit within the TLB, illustrating the direct impact of flushing the TLB on memory load latencies. As the working set size increases, we see the effect of the TLB capacity to translate for a random-access workload, where tagging exhibits diminished benefits.

the largest cost. On machine M2 where these measurements were taken, the overall switching latency with one attached address space is 450 nanoseconds. If a process would have a large number of attached address spaces, this cost would be reflective of the data structure and algorithm used to identify the address space based on its VAS ID (whether as an index into a table, or a lookup into a binary search tree).

While immediate costs may only be improved within the micro-architecture, subsequent costs can be improved with TLB tagging. Tagging allows for the TLB to avoid being flushed when the page table base address in CR3 is modified (*control register* – used by system software to alter the behavior of the processor, where CR3 holds the base address of a page table) by associating a few extra bits in the TLB with the translation state for specific entries, to denote which page table cached translations were derived from. Tags (also called address space identifiers) are simple integer values stored in the hardware. On a page fault, the translation hardware will examine the current ASID and use that when observing cached entries. Using tags allows us to avoid costly TLB flushes, at the expense of increasing pressure in the TLB (e.g., due to duplicate translations to a common segment

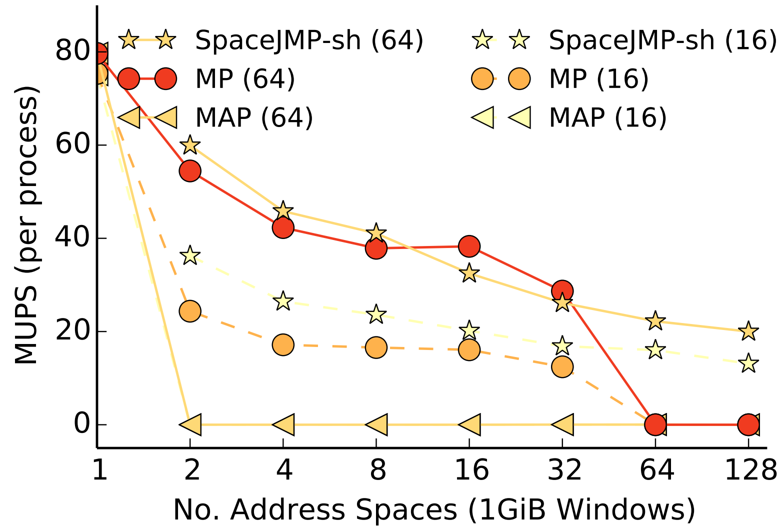


Figure 48: Comparison of three designs to program large memories with GUPs M3. Update set sizes 16 and 64.

used across multiple address spaces).

In Table 47 we directly measured the impact of tagging in the TLB using a random page-walking benchmark we wrote. For a given set of pages, it will load one cache line from a randomly chosen page. A write to CR3 is then introduced between each iteration, and the cost in cycles to access the cache line are measured. Lastly, we enable tags. The benefits, however, tail off: with tags, the cost of accessing a cache line reduces to the cost incurred without writes to CR3 as the working set grows. As expected, benefits gained using tags are limited by a combination of TLB capacity (for a given page size), and sophistication of TLB prefetchers. In our experiment, access latencies with larger working sets match that of latencies where the TLB is flushed.

As an aside, notice that in Table 6 the time to change the value in CR3 increases with tagging enabled: doing so invokes additional hardware circuitry that must consider extra TLB-resident state upon a write. Naturally, these are hardware-dependent costs. However, the overall performance of switching is improved, due to reduced TLB misses from shared OS entries.

10.2 GUPS: Addressing Large Memory

Recall earlier from Figure 6 that page table modification does not scale, even in optimized, mature OS implementations. The reason is because entire page table subtrees must be edited – costs which

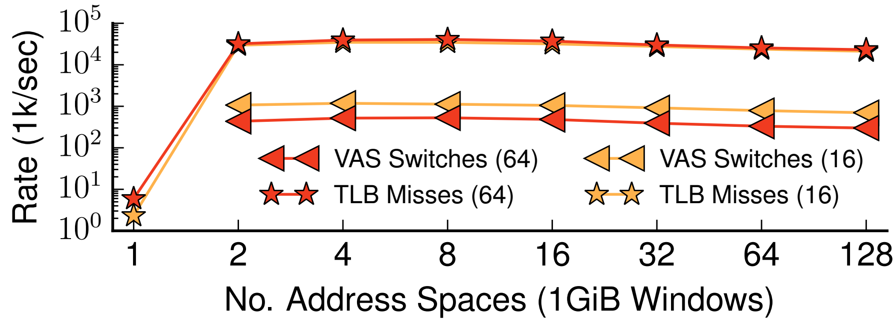


Figure 49: Rate of VAS switching and TLB misses for GUPS executed with SpaceJMP, averaged across 16 iterations.

are directly proportional to the region size and inversely proportional to page size. When restricted to a single per-process address space, the changing of translations for a range of virtual addresses using mmap and munmap creates these costs. With SpaceJMP, these costs are removed from the critical path by switching the translations themselves instead of modifying them.

To deal with the limits of addressability for virtual memory, applications adopt various solutions for accessing larger physical memories. In this section, we use the GUPS bench- mark to compare two approaches in use today with a design using SpaceJMP. We ask two key questions: (i)how can applications address large physical memory regions, and (ii) what are the limitations of these approaches?

GUPS is appropriate for this: it measures the ability of a system to scale when applying random updates to a large in-memory array. This array is one large logical table of integers, partitioned into some number of windows. GUPS executes a tight loop that, for some number of updates per iteration, computes a random index within a given window for each update and then mutates the value at that index. After each set of updates is applied, a new window is randomly chosen. Figure 48 illustrates the performance comparison between three different approaches, where performance is reported as the rate of million updates applied per second.

The first approach (MAP) leverages address space remapping: a traditional process may logically extend the reachability of its VAS by dynamically remapping portions of it to different regions of physical memory. Our implementation of this design uses the BSD mmap and munmap system calls (configured to attach to existing pages in the kernel's page cache) opening new windows for writing.

The second traditional approach (MP) uses multiple processes: each process is assigned a distinct portion of the physical memory (a window). In our experiment, one process acts as master and the rest as slaves, whereby the master process sends RPC messages using OpenMPI to the slave process holding the appropriate portion of physical memory. It then blocks, waiting for the slave to apply the batch of updates be- fore continuing, simulating a single thread applying updates to a large global table. Each process is pinned to a core.

We compare these techniques with VAS switching, modifying GUPS to take advantage of SpaceJMP. Unlike the first technique, we do not modify mappings when changing windows, but instead represent a window as a segment in its own VAS. Pending updates are maintained in a shared heap segment mapped into all attached address spaces. Figure 49 illustrates the rate of VAS switching and TLB misses.

10.3 Discussion

For a single window – no remapping, RPC, or switching – all design configurations perform equally well. Changing windows immediately becomes prohibitively expensive for the MAP design, as it requires modification of the address space on the critical path. For the MP and SpaceJMP designs, the amount of CPU cache used grows with each added window, due to cache line fills from updates, as well as the growth in required page translation structures pulled in by the page-walker. The SpaceJMP implementation performs at least as well as the multi-process implementation, despite frequent context switches, with all data and multiple translation tables competing for the same set of CPU caches (for MP, only one translation table resides on each core). At greater than 36 cores on M3, the performance of MP drops, due to the busy-wait characteristics the OpenMPI implementation. The same trends are visible across a range of update set sizes (16 and 64 in the figure). Finally, a design leveraging SpaceJMP is more flexible, as a single process can independently address multiple address spaces without message passing.

This experiment shows that SpaceJMP occupies a useful point in the design space between multiple-process and page-remapping techniques – there is tangible benefit from switching between multiple address spaces rapidly on a single thread.

10.4 Other Applications

While not specifically a contribution in this dissertation, other application scenarios that were evaluated with SpaceJMP are mentioned here briefly for completeness (more details found in published work [65]).

We demonstrate the advantages of address space switching in two additional application scenarios. The first examines the use of a key-value store similar to the work described earlier in this dissertation. Redis [152] was modified to take advantage of address space abstractions, with a single server address space that client threads could attach to (no server threads were alive or active). By entering the server address space, we can avoid costly thread-to-thread communication and synchronization, see immediate benefits of parallelization, and provide a very fast mechanism for clients to either insert or extract information from another process' context. Our measurements on M2 demonstrate a 4× increase in throughput when compared to a single Redis server instance, and 36% increase using 6 Redis server instances. Further gains would be possible if the implementation within Redis utilized finer-grained locking mechanisms to perform object lookups and memory allocation.

A final application scenario we evaluated was with SAMTools [109], a tool suite for manipulating DNA sequence alignment data sets. Each tool reads data from files and performs some operation on the data in memory, after converting it into an in-memory representation amenable for searching or manipulating. The idea is that at each stage, data must be converted from the on-disk format to an in-memory format, costing time and the need for additional code. The use of SpaceJMP allows the tools to keep data in its in-memory representation, entirely avoiding conversion costs. Each data set is stored within an address space, and a process working on the data would attach to the required address space and switch as-needed to access the information. Our measurements on multi-gigabyte data sets show the avoidance of data transformation allows for significant speedups, being the main cost in the benchmark.

10.5 Summary

Our implementation in SpaceJMP is demonstrated across a variety of application use cases. For each, we show that switching address spaces is much more efficient than use of memory mapping in the critical path for accessing data, allows tools to avoid data transformations between storage an main

memory, and provides shared environments with the ability to avoid alternative IPC mechanisms for managing large data sets.

CHAPTER XI

RELATED WORK

The work in this dissertation draws on inspiration across a wide distribution of areas in the research community. In this chapter, we identify the main themes of overlap, such as in memory allocation, scalability, data management, and operating system research.

11.1 Memory Management and Allocation

There has been much prior work studying the characteristics of heap allocators. Custom techniques that match application behavior are difficult to implement, thus it has been demonstrated that general-purpose allocation can provide a good balance between performance and portability [94, 38]. Many allocator designs have proposed greater levels of concurrency of their implementations, avoiding locks or global data structures, or leveraging thread-local pools from which to allocate, or use of concurrent data structures [153, 24, 2, 68, 37]. To achieve low fragmentation, designs have incorporated mechanisms of allocation from other systems software, such as slab allocation in operating system kernels [42] (in jemalloc these are called 'runs'). Without the ability to relocate objects, however, there will inevitably be scenarios that unavoidably cause fragmentation [170], an increasingly important characteristic to study for long-running servers [106] such as key-value stores like Nibble. We note that the heap allocators evaluated in our work do scale for general applications, as heap objects based on structs are not expected to frequently shift in size, and not all applications exhibit such dense coupling. In our work, applications handle externally created objects, whose sizes are not based on application code.

NUMA. Much prior work evaluates the impact of NUMA characteristics on applications [76], with proposals to use automated solutions to avoid memory bandwidth bottlenecks [58], or compiler and language support to infer memory access patterns [95]. Further challenges exist in overcoming the transparent nature of managing physical memory; proposals to use hardware acceleration for memory movement [118, 95] have demonstrated success in some situations, compared to CPU-based copying.

These efforts may complement applications that use Nibble on large NUMA systems, as Nibble explicitly allocates memory across sockets, exposing interfaces for applications to explicitly place data in specific memories.

11.2 Scalable Key-Value Stores

A variety of techniques have been explored for object stores and in-memory caches to overcome memory load imbalances. Systems like Redis [152] and memcache [**?**] are not fully distributed systems (unlike, for example, RAMCloud [143]), thus it is up to the client software to take part in managing the distributed service. Techniques such as specializing partitions based on access patterns have been used to improve object cache hit ratios [174], and the use of concurrent designs to improve load balancing [51]. In Nibble, imbalances in capacity use may occur when client threads load new objects into a subset of the memory sockets of a machine; compaction in the current design will not relocate items across sockets, thus PUT operations may fail if directed to a socket with low memory available. Balancing is important even for such large machines, as the performance of compaction and frequent use of PUT and DEL may experience delays unless directed to other less-utilized sockets.

On large SMP machines, numerous designs have been explored that allow request processing to scale across sockets. MICA [116], discussed in this dissertation, partitions its heap allocator and index across each core. When in a networked environment, advanced network hardware mechanisms allow for packet routing to the appropriate core (based on the key's hash) to efficiently implement scalable processing. CPHash [130] similarly partitions its state across cores in the system to better make use of processor caches, with message-passing buffers used to transfer requests to appropriate cores. Joint work between Facebook and Tilera [36] used similar approaches in memcache on alternative hardware (Tilera processors). Nibble leverages some of these common techniques to scale, such as partitioning, as they have shown to be a proven technique in scaling on large SMP machines.

11.3 Log-Structured Systems

The origination of this and related areas of work stems from the log-structured file system efforts by Rosenblum and Ousterhout from 1991 [149] in the Sprite operating system. They designed and developed a file system whereby all mutations to file data resulted in appends to data on

disk. The design was motivated by the observation that with file data cached in memory, most operations to disk would be writes; given disk fragmentation over time, writes would incur many disk seeks (for cylindrical-based storage). Converting writes to append a log sequentially would allow greater utilization of disk bandwidth. As it applies to key-value stores, where interactions with storage are concerned, log-structured writes enable fast persistence and recovery, thus its use is still valued [151, 115]. For non-persistent stores like Nibble, converting random writes to sequential writes seems to provide less value, however the ability to rearrange memory items and exploit low-latency allocation times remain effective.

RAMCloud [151] is a distributed in-memory storage system that emphasizes fast recovery of durable data. It also implements a log with concurrent compaction within each server, must balance memory cleaning with disk cleaning, and prioritizes different efforts based on the guarantees it provides to applications. For example, RAMCloud is intended for extreme scale-out data centers and very fast recovery of failed server nodes, whereas Nibble targets extreme scale-up systems, thus RAMCloud's innovations are complementary to those in Nibble. Recent work based on RAMCloud [53] examines the use of log cleaner threads to enforce memory credits assigned to clients when a key-value store is used in a multi-tenant environment. Such methods may be useful in Nibble, as well, to allow for re-balancing memory among sockets.

Prior work exists on evaluating the scalability of log-structured designs for structured data storage. Bwtree [119] is an SSD-optimized database that uses logs to allow for fast writes of data to disk, by keeping track of only differences in updates, benefiting large objects such as tables. The design goals in this work are optimized for cooperating with storage systems, but have evaluated the scalability of log designs on multi-core platforms. While allowing for concurrent updates to objects in pieces using concurrent queues (by appending the modifications from partial updates), extra work is necessary to both read the object back (replaying deltas to reconstruct the object) and to ensure the delta list does not grow large enough to begin wasting memory. Optimizing for concurrent writes was also the approach seen in OpLog [44] which combined the use of log allocation with techniques discussed above, assigning a mini-log to each core in the system, where an object's state is written to together with a hardware time stamp (thus allowing concurrent updates by avoiding contention on shared memory locations). This design allows for supporting scalable write-heavy workloads, but also at the

expense of reading back data: locks across all queues must be acquired and per-core logs traversed to determine the most recent state of an item. Nibble does not interact with storage systems, thus its priorities lie with multi-core and multi-socket scalability, and like Bwtree and OpLog, is able to scale write-heavy workloads, but due to its use of whole objects, does not require costly synchronization to reconstruct one upon a read.

Log-structured merge trees [142] are an additional area of work that has seen a lot of attention recently, especially in designing for scalability on storage [155, 78]. A recent work, cLSM [80], specifically evaluates the scalability of log-structured designs for single-node systems, leveraging similar designs such as non-blocking operations. Scalable designs for log-structured merge trees, however, have not been evaluated on machines of similar scale as those evaluated in this dissertation's work, thus their absolute scalability on large SMP machines remains an open question. As Nibble does not provide persistence of data, its use of logs is strictly to benefit fully in-memory object management systems, and due to its separation of compaction as a separate background activity, is able to independently parallelize compaction independent of client write operations. To the best of our knowledge, there are no prior studies examining the scalability of non-persistent log-structured allocators on extreme-scale memory systems.

11.4 Memory Reclamation and Concurrent Data Structures

Our system Nibble shares a technique for memory reclamation found in many prior systems using the time stamp counter [166, 44] as an epoch for enabling concurrent access with memory reclamation [151, 131, 73, 34]. Read-Copy-Update [128] is a form of memory reclamation applied to data structures to allow concurrent lockless reads, even in the presence of some writes, by copying portions of a data structure, atomically committing changes. RCU has been successfully applied to a variety of data structures, such as balanced trees in operating systems [54]. New hardware features such as HTM have also been explored in enabling concurrent memory reclamation [25].

Supporting dynamic resizing is challenging with RCU, thus alternative techniques for scaling data structures, such as hash tables [162] and trees [91] have been proposed. As Nibble prioritizes low-latency operations, use of relativistic hash tables, for example, would require chained buckets – a linked-list data structure that presents poor cache locality, and limited concurrency during resizing

operations.

Comprehensive surveys and analyses of synchronization techniques and atomics on very large systems [84, 59] have given rise to general programming paradigms for designing a wide variety of concurrent data structures, such as ASCY [60]. Scaling optimistic concurrency control (OCC) data structures, such as in Optik [85], provide general design guidelines for applying OCC to a variety of data structures. How to resize OCC-based hash tables, however, is not reviewed in-depth. In Nibble, by partitioning the index across sockets and cores, we are able to attain some concurrency for resizing by allowing each partition to be resized independent of the rest.

11.5 Garbage collection systems

Garbage collection as a memory management technique aims to track dynamic memory objects, periodically scanning active objects to determine reachability. If no references exist to a given object, it is considered dead, and its resources reclaimed. The process of scanning objects consumes extra CPU time, and can lead to indeterminism in when garbage collection tasks run, and how long they run for, requiring up to 5× as much memory to perform equivalent to explicit memory allocators [90]. Knowing both when and how long GC kicks in are important, especially in real-time or transaction systems that are latency sensitive, as well as in multi-threaded applications where 'stop-the-world' collection strategies are used. Recent efforts in the Go programming language [81] have introduced a semi-concurrent garbage collector [4] (where previously it had used a stop-the-world collector), allowing applications to grow the heap at the same time the collector runs, but is not entirely without pauses (a millisecond-range bound is described). Pauseless garbage collector implementations exist [160].

In "big data" processing systems, there have been a number of projects examining the automatic reclamation of dead objects, such as in managed languages like Java. Broom [79] is a recent study on such systems, which found that high object churn causes language-based garbage collection to introduce large runtime costs. They propose a special region-based allocator (examined also in other work, Reaps [38]) to mitigate this overhead. A similar work, Yak [139], proposes to separate the heap managed by the JVM into 'control' and 'data' spaces, where objects in the former are subject to standard garbage collection mechanisms, and objects placed in the latter are grouped also using

region-based allocation, and released via epochs that track their creation and use. It is natural to study memory reclamation strategies for applications written in managed runtimes, as these languages (e.g., Java, Scala, etc.) have become increasingly popular in building modern "big data" systems, such as Spark [176], Hive [161], Hadoop [13], Yarn [164], Dynamo [61], and many others.

Similar to Nibble's log-structured allocation, garbage collectors may use bump allocation and copying methods. Immix [41, 158] implements similar strategies for memory reclamation as in Nibble, combining bump allocation with 'sweep-to-region' collection that allocates and reclaims memory in contiguous blocks, much like Nibble's segments.

Nibble is written in Rust [135], a compiled memory-safe language that uses the compiler to perform memory management – via resource acquisition is initialization (RAII) – and thus does not use automated garbage collection features, such as in the Java or Go languages. While memory leaks are still possible, pauses due to reclamation will not exist; Nibble's compaction, however, will kick in at certain times depending on the amount of available memory found. If an application could be characterized to exhibit certain phases of execution, some latency- and others bandwidth-bound, one might modify the compaction thread logic to engage during phases that are not latency- or time-critical. An advantage of Nibble's memory reclamation design over typical garbage collection strategies is that no global pauses exist: compaction occurs one segment at a time, and locking only happens per-object, when it is relocated. Thus this strategy is much more concurrent. An obvious difference is that Nibble is not used to implement *languages* where garbage collection is typically used, but higher-level data management.

11.6 Operating Systems and Address Spaces

Overlap with operating system research in the past is large. Thus in this section, we make an attempt to cover the main topics.

Greater control over address spaces. The topic of 'regaining' control over virtual memory from the operating system has been a balancing act between operating systems providing high-level abstractions (e.g. Linux, *BSD, Microsoft Windows, etc.) and designs which provide merely protection (such as the Exokernel work [67]).

Much work from the nineties focused on making operating systems more extensible for applications to configure for their own use. The SPIN [39] operating system allowed user-space to dynamically link in code to execute within the kernel address space. As SpaceJMP is not defined by its implementation, but by the design itself, facilities in SPIN, for example, would support implementing address space abstractions, segments, and the ability to dynamically switch the address space by supplementing in modifications to the page fault handler, address space management facilities, and additional system calls for enabling interaction between applications and virtual memory controls.

The Nemesis [87] operating system specifically focused on extensibility of virtual memory. Their goal was to target quality-of-service in the memory subsystem of the kernel, benefiting multimedia applications which are performance-sensitive workloads, by requiring such applications to be individually responsible for servicing their own memory faults.

Microkernels provide a form of recursive construction of address spaces [112] (a good overview of their history and evolution [66, 88]), pushes memory management out of the core kernel and into services. Granting and mapping of pages are controlled from the core kernel, but the ability to page memory is implemented within user-space. One might envision the ability to switch the translations (the page tables) as a form of 'fast mapping' provided by the kernel itself, with pages already granted and accessible to a program.

Single-Address-Space Designs. The idea of sharing translations has a long history in operating systems, especially designs with a single address space.

IVY [110] proposed the idea to share translations as a contrasting mechanism to message-passing for data sharing on multiprocessor machines. Similarly to SpaceJMP, the address space is divided into partitions, one for shared data, and a private partition for process-specific (unshared) data. 20 years later, in the high-performance computing community, the ability to efficiently share in-memory data structures remains important, as runtimes used for implementing large-scale applications use multi-process models, such as MPI. Instead of copying data via IPC channels, work such as SMARTMAP [45] allows processes – without kernel involvement – to directly use the virtual memory mappings of other processes (copying the entries in the root page translation table directly).

With architecture support for very wide virtual addressing, the idea that virtual memory was

practically infinite led to work to design entire operating systems around the use of a single operating system for all processes, such as the work from Opal [49, 102, 47, 48]. To enforce protection in such a model, the idea of protection domains was created that allowed processes to – as in SpaceJMP – *attach* and *switch between* to alter the access control to memory, enabled by the use of multiple page tables and address space identifier registers in hardware. SpaceJMP does not enforce a model where processes all by default share a single address space, but rather advocates a data-centric model where address spaces are specifically used for very fast translation swapping between segments of memory.

Similar work followed in Mungi [89], advocating for efficient sharing of pointer-based data structures, especially of 'persistent data' that is placed into files. Loading shared memory objects results in varied virtual memory locations, rendering the addresses kept within a data structure unusable. A further motivation is the observation that on commodity processors, the virtual addressing range would also be practically infinite, even for sharing memory across distributed systems.

With very dense persistent memory technologies and memory routing networks such as Gen-Z [5], physical memory is quickly growing in size; commodity processors also do not have full 64-bit addressing (e.g., 48 bits in x86), making, again, virtual memory less available than previously thought. The single-address-space model, we posit, would be limited in the amount of physical memory it could address. SpaceJMP assumes this from the beginning, but acknowledges the importance of data sharing, placing itself in a 'middle road' amongst such prior work: instead of sharing one large address space, we make the ability to change translations efficient, while not being encumbered by the limitations of virtual memory addressing.

Outside of operating systems, the use of shared memory mappings was explored at length in QuickStore [168], a mapped object store for persistent C++. Objects are stored in their in-memory representation on disk within pages, mapped in or out depending on the access patterns of the application. Different from prior work, they assume address mapping may result in a different location, forcing swizzling techniques to take place. Conflicts may arise due to the fragmentation of the current address space layout. Building a system like QuickStore on top of an implementation of SpaceJMP would remove the need to swizzle, using address space allocation to acquire new memory regions, and switching the address space to implement a fast mapping to such data.

Fast Context Switching. The ability to quickly perform a context switch was studied in μ-kernels, such as the work by Liedtke [114, 113], where control transfer between services is implemented via IPC. This work advocated for a cooperation between the hardware features and the operating system to ensure efficient implementation of underlying mechanisms. Later work by Wiggins and Heiser [169] examined how to leverage hardware features in ARM processors to accelerate context switching in the L4 μ-kernel, where tagging of the TLB is unavailable.

Scalable Address Spaces. In parallel applications, heavy use of address space modification may create hard bottlenecks for performance. Heap allocators open new areas of memory when expanding the heap, or when allocating 'large' objects; modern heap allocators [104] will allocate large swaths of virtual memory, but only fault in the physical memory upon actual allocation; multi-threaded applications that process large file data sets may leverage mmap(2) to access each file independently. Clements et al. examined designs within the operating system to allow for scalable modification of the address space, via mmap(2) [55] as well as enabling parallel page-faulting to allocate physical memory [54].

Work from the K42 [103] operating system introduced the notion of 'clustered objects', allowing system software to choose a specific design implementation of kernel objects to support scalability across multi-core platforms.

The approach in SpaceJMP allows for concurrency in address space modifications by virtue of using multiple address spaces at the same time. Methods above may complement our work by enabling the OS design overall to scale.

11.7 Hardware Address Space Support

Given the size of memories expected to be used by applications, translation costs may grow, and put pressure onto the processor-managed translation caches [40]. Large pages are an example solution, through use of page-table-based virtual memory translation, but some research has shown the full management thereof by the operating system kernel (such as Transparent Huge Pages in Linux [3]) to be insufficient or detrimental for applications [138, 105, 75].

Recent research in the architecture community has examined alternative designs for virtual memory support. The use of a Direct Segment [32, 74] removes the use of page tables for large

contiguous regions of virtual memory – appropriate for SpaceJMP segments (depending on use), as well as the large per-socket logs in Nibble. A Direct Segment is intended to eliminate translation for dense memory areas that share the same memory protection characteristics, saving on runtime costs (found to be up to 80% of time in some workloads).

Sharing data and data structures at very fine granularities is useful for a variety of workloads [171, 172], such as deduplication, virtual machine monitors, and compression. Recent work in architectural designs alter the virtual memory subsystem in processors to support efficient sub-page-level tracking [156, 97]. With improved architectural support, SpaceJMP address spaces may better implement copy-on-write techniques to implement memory versioning or checkpointing, or improve the granularity of sharing data structures between applications (no longer bound to page sizes). Improvements in hardware to support virtual memory may be leveraged by the work in this dissertation.

Multi-address-space support is present in other hardware architectures, such as the HP PA-RISC [122] as well as Intel's Itanium [177]. Registers on these architectures provide access for applications to manage multiple address spaces, designs that would complement how SpaceJMP maintains attached address spaces for switching.

CHAPTER XII

CONCLUSION

Summary. With memory becoming the central resource of future machines, it is vital to tailor systems to provide applications, and thus system software, the most efficient means of managing it. This dissertation introduced two aspects of efficient programming demonstrated on very large-SMP memory machines – Nibble and SpaceJMP.

We first introduced Nibble, a concurrent log-structured key-value store. Its design leverages well-known techniques that together enable it to support both read-heavy and write-heavy workloads. By partitioning its log across sockets in a machine, allocating a log head to each core, it can support scalable writes as well as high volumes of object churn (allocations and deletions). Isolating compaction activities to sockets limits cross-socket noise. A concurrent open-addressing hash table as an index allows all operations – clients and compaction – to work independently and concurrently with each other. A distributed hardware-based epoch provides a low-latency mechanism for tracking memory segments to reclaim. Comparing with competitive data management designs, we demonstrate both scalability in throughput, as well as efficient use of memory capacity. We also provide an analysis of the influence on performance of compaction.

In support of the diverse set of software executing at-scale on large machines, we promote the use of virtual address spaces to become first-class citizens, and for processes to be able to maintain multiple address spaces at the same time, switching between them dynamically to access in-memory data sets. Control over the layout of virtual memory is regained by being able to freshly allocate new address spaces, without having to modify existing systems, such as ASLR, or being subject to fragmented virtual memory state.

Thoughts and Lessons. Throughout the work presented in this dissertation, we demonstrated across a variety of memory management designs that log-based allocation supports efficient use of memory capacity: each general or custom-designed heap allocator studied exhibited moderate to

severe fragmentation, consuming $2 - 3\times$ the amount of memory needed. Parallelizing log-based allocation methods via a concurrent index and partitioned state across NUMA domains allowed for our design in Nibble to competitively scale in both write- and read-heavy workloads. Without a concurrent index, read-heavy workloads would experience bottlenecks; without the use of multiple log heads, write-heavy patterns would contend on each update to a log. Applications working with static data sets at-scale may do just as well with a general-purpose heap allocator, as no online maintenance of objects would be required; our combined methods are most useful for parallelizing access to dynamically changing data sets, seen in many modern data management tools, such as key-value stores.

As it applies to building general applications and tools on large-memory SMP machines, current operating system interfaces were discussed and empirically measured to demonstrate either their inability to scale, or their growing costs with respect to data set sizes (e.g., mmap). Explicit control given to user-level software over memory mappings provides an outlet to both continue use of current memory management methods deployed by the operating system (e.g., ASLR) and support for applications to address increasingly large memories, to seamlessly share in-memory data sets, and to persist data by dissociating memory areas from process lifetimes. While much prior research in operating systems have examined allowing greater control over virtual memory for applications (e.g., exokernel, SPIN, Barrelfish, etc.) we have not explicitly seen support for fast address space manipulation abstractions.

Next we briefly discuss opportunities for extensions as well as future work directions to pursue.

12.1 Extensions and Opportunities

A variety of next steps exist, both incremental and larger efforts.

12.1.1 Load-Balancing in Nibble

The write-heavy scalability in Nibble is primarily due to its use of multiple log heads, and the ability for threads to pick which socket memory to write to. As compaction threads are pinned and bound to specific sockets, no memory is every relocated except by an application performing a PUT to another socket. In order to fully utilize all available memory in a multi-socket machine, applications must be

aware of this limitation, and direct PUT requests to appropriate sockets; in essence, applications are responsible for load-balancing across sockets. Such a scenario may occur in loading data sets are larger than the memory on a single socket, especially if done by a small number of threads (as they may be placed onto just 1-2 sockets and consume all the memory there).

Ensuring better load-balancing may be achieved by allowing compaction threads to relocate objects across sockets, perhaps only then when an imbalance is detected. An easy implementation might be achieved by having compaction threads behave as clients, issuing PUT operations with the original log-resident buffer to remote sockets.

12.1.2 Index Optimizations in Nibble

One drawback in the design of Nibble's index is that it currently does not automatically shrink in size if it grows much larger than the number of keys it holds. Fragmentation of heap memory vs log allocation was the main focus in this work, but should applications create large numbers of objects (e.g., a spike in short-lived data), the index would grow to accommodate it. Resizing a hash table while allowing concurrent operations is non-trivial [162]. Some support exists in Nibble to *grow* the index concurrently, by partitioning it into multiple pieces, growing each partition independently. As each table grows, resizing becomes increasingly costly, thus partitioning does not solve the problem. Even hash table designs that can more fully utilize the set of buckets, such as the use of Cuckoo Hashing [111], cannot help when the number of keys exceeds the actual capacity of the index itself.

Our index, while concurrent, places multiple keys into a bucket at a time, locking it in the presence of mutations. Should a mutator be *descheduled* by the operating system task scheduler, all concurrent reading (and other mutator) threads will be forced into extended retry loops, waiting for the version to increment to an even value. While this does not affect correctness, it may impact performance. Past research has looked at similar challenges with respect to locking for general-purpose software [93], an example being priority inversion [157]. We expect scenarios where this occurs to be in over-subscribed systems (more threads than cores), and where the work performed while holding a bucket lock is non-trivial (thus requiring a long or indeterministic amount of time). The latter has happened in Nibble, specifically when compaction threads must relocate an object (acquiring the bucket lock) but the new segment cannot fit it; expanding the segment thus requires allocating a few more blocks.

Such sequences of operations cause non-trivial deadlocks and performance problems to solve.

12.1.3 Hybrid DRAM + NVM Machines

The emergence of new memory types will create machines presenting a mix of memory for applications to take advantage of, e.g., the new supercomputer Summit [21] will carry a mix of non-volatile and volatile memory on each machine. Non-volatile memory programming is currently a hot area of research, because it presents systems software with large potential for both performance and alternative programming models.

One step towards enabling a key-value store such as Nibble to become persistent (with byte-addressable persistent memory) is to examine the persistence capabilities of the index. RAMCloud, for example, is persistent, but does not save the index across failures. Instead, they record *deletions* of objects in the log as separate metadata called 'tombstone' objects, a design that adds complication to log cleaning and recovery [150]. None of the recent work on persistent data structures allows for concurrency – a hard limit on performance on large SMP machines, thus addressing this angle would prove valuable.

Thus, an open area to pursue would be in examining data structures combining their ability to leverage persistent memory with concurrent programming methods. Very recent work has begun to adapt balanced trees to directly use non-volatile memory [50], relaxing the sorted property of items within a node to avoid expensive write amplification. Subsequent work on radix trees [107] takes advantage of the natural ordering of keys to reduce write costs. One recent area of work mirrors the index across memories: a sorted tree kept in DRAM, and a persistent hash table in non-volatile memory [175].

Developing a fast, persistent key-value store (if with log-structured approaches) would require an architectural analysis of the interplay between non-volatile memory and compaction activities. Relocation of objects is a write-heavy activity. In some settings it might be valuable to revisit the ideas from Chapter 6 as it applies to non-volatile memory, looking for ways to reduce object relocation. Changing translations is faster than relocating objects (when large) as is done with the Linux implementation of mremap(2); taking advantage of fine-grained translation [**?**] might assist here.

Operating system interfaces that export a more true understanding of memory, such as the address spaces and segments in SpaceJMP, may also expose the characteristics of such memories to user-space. For example, one could allocate persistent segments or volatile-memory segments. Each may have alternate implementations of allocators that would be linked in when accessing those segments, such as a persistent malloc.

A large area of study would be to re-examine the entire design of file systems, and whether the use of stronger address space constructs available to user-space could assist in developing file systems with leaner interfaces and stronger protection granularities than is current with the mixture of files and virtual memory abstractions. Interesting questions to explore would be, can files be address spaces themselves? If we think of files as just another form of binary data, would a file system just be adding meaningful semantics on top, such as a 'name' and 'location' that we find useful from a traditional file system?

12.1.4 Hardware Extensions

Modern commodity processors are introducing a wealth of new features that would be exiting areas to explore. Encrypted memory spaces (e.g. 'SGX' on Intel processors) could provide protected memory spaces that may supplement or remove the need for strict access control (if you cannot decrypt it, who cares if you can read it?). Use of this feature may be interesting to explore for new in-memory file systems.

Encrypted memory spaces may also be useful in the design of a multi-tenant shared-memory key-value store. Applications that wish to place their data under the control of another data management tool may not trust other clients of the same tool. Giving a separate, encrypted log to each tenant might be a curious area to explore. Systems such as Barrelfish, SpaceJMP, and others provide explicit control for user-space management of virtual memory, and are likely candidates for use as a foundation to do this.

Another feature worth exploring is the use of memory protection keys (Intel's MPK feature) that allows for modifying the access permissions to virtual memory without altering the page table (and thus avoiding TLB flushes). If multiple applications share a common address space, differing only by the access permissions to subsets of the data, it may be beneficial to leverage such protection

keys and use a common page-table itself. Doing so may allow better use of the TLB capacity as translations may be reused (due to the same page-table) but access restrictions enforced instead by the keys.

12.1.5 Miscellaneous

Opportunities to leverage copy-on-write (COW) techniques in address space semantics for performing memory check-pointing or versioning of processes or data seem rich to explore. COW techniques are typically used by the operating system to implement components of sharing memory objects (e.g., in *BSD systems), as well as for forking processes. Exposed as an explicit construct for user-space to take advantage of might allow further areas of study to develop memory-centric control of data sharing.

One challenge we had not yet explored in SpaceJMP were use-cases where processes attached to the same address space would change the protection or the mappings of segments. If an attached address space maintains a copy of the page-table, then shrinking or constraining the access to a shared region would require complex coordination to ensure correct enforcement of changes: on large machines, TLB shoot-downs become costly to implement (due to broadcasting of IPIs to all cores), and may benefit from implicit tracking of TLB state [26]; modifying numerous page-tables that mirror the original also becomes expensive at-scale – should we instead share page-tables instead of mirroring them via copies?

12.2 Closing Remarks

We continue to see applications take full advantage of massive-memory SMP machines in various ways – caching data, generating temporary objects, close-coupled sharing via pipelines, etc. – and predict this demand will further interest in research that explores areas of data management, not just for volatile memory, but also for machines with mixed memory technologies, alternate addressing schemes, relaxed consistency models, as well as on hardware seeing a closer merging of features between distributed networks and direct addressing methods. Already there is a flurry of related research evaluating OS support for non-volatile memory, but now with enormous scales of access, hybridization, and high-performance, further opportunities for re-evaluating current system software stacks to support such machines are opening up.

“And now for something completely different.”

-Monty Python’s Flying Circus

REFERENCES

[1] "ptmalloc." http://www.malloc.de/en/, 2006.

[2] "TCMalloc." http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2006.

[3] "Transparent huge pages in 2.6.38." http://lwn.net/Articles/423584/, 2011.

[4] "Go 1.5 concurrent garbage collector pacing." https://golang.org/s/go15gcpacing, 2015.

[5] "Gen-Z Consortium." http://genzconsortium.org/, Oct. 2016.

[6] "Hewlett packard the machine." http://www.labs.hpe.com/research/themachine/, 2016.

[7] "Hp proliant dl580." http://www8.hp.com/us/en/products/proliant-servers/product-detail.html?oid=8090149, 2016.

[8] "Hpe integrity superdome x." https://www.hpe.com/us/en/servers/superdome.html, 2016.

[9] "HPE Integrity Superdome X system architecture and RAS." https://www.hpe.com/h20195/v2/getpdf.aspx/4AA5-6824ENW.pdf, 2016.

[10] "Siphash: a fast short-input prf." https://131002.net/siphash/, Aug. 2016.

[11] "Thunderx arm processors." http://www.cavium.com/ThunderX_ARM_Processors.html, May 2016.

[12] "5-level paging and 5-level ept." https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, May 2017.

[13] "Apache hadoop." http://hadoop.apache.org/, 2017.

[14] "DragonFly BSD." https://www.dragonflybsd.org, May 2017.

[15] "Filesystem in userspace." https://github.com/libfuse/libfuse, Feb. 2017.

[16] "Google Protocol Buffers." https://developers.google.com/protocol-buffers/, July 2017.

[17] "K computer." https://www.top500.org/system/177232, July 2017.

[18] "Power isa version 3.0b." https://www.ibm.com/systems/power/openpower, 2017.

[19] "Sequoia - BlueGene/Q." https://www.top500.org/system/177556, July 2017.

[20] "SGI UltraViolet." https://www.sgi.com/products/servers/uv/, 2017.

[21] "Summit – oak ridge national laboratory's next high performance supercomputer." https://www.olcf.ornl.gov/summit/, 2017.

[22] "Sunway TaihuLight." https://www.top500.org/system/178764, July 2017.

[23] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R., "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical Conference*, ATC'08, (Berkeley, CA, USA), pp. 57–70, USENIX Association, 2008.

[24] Aigner, M., Kirsch, C. M., Lippautz, M., and Sokolova, A., "Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), pp. 451–469, ACM, 2015.

[25] Alistarh, D., Eugster, P., Herlihy, M., Matveev, A., and Shavit, N., "Stacktrack: An automated transactional approach to concurrent memory reclamation," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 25:1–25:14, ACM, 2014.

[26] Amit, N., "Optimizing the TLB shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 27–39, USENIX Association, 2017.

[27] Amur, H., Richter, W., Andersen, D. G., Kaminsky, M., Schwan, K., Balachandran, A., and Zawadzki, E., "Memory-efficient groupby-aggregate using compressed buffer trees," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 18:1–18:16, ACM, 2013.

[28] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V., "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 1–14, ACM, 2009.

[29] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M., "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, (New York, NY, USA), pp. 53–64, ACM, 2012.

[30] Atlidakis, V., Andrus, J., Geambasu, R., Mitropoulos, D., and Nieh, J., "Posix abstractions in modern operating systems: The old, the new, and the missing," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 19:1–19:17, ACM, 2016.

[31] Aumasson, J. and Bernstein, D. J., "Siphash: A fast short-input PRF," in *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, pp. 489–508, 2012.

[32] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M., "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 237–248, ACM, 2013.

[33] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A., "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 29–44, ACM, 2009.

[34] Baumann, A., Heiser, G., Appavoo, J., Silva, D. D., Krieger, O., Wisniewski, R. W., and Kerr, J., "Providing dynamic update in an operating system," in *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pp. 279–291, 2005.

[35] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P., "Finding a needle in haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[36] Berezecki, M., Frachtenberg, E., Paleczny, M., and Steele, K., "Many-core key-value store," in *2011 International Green Computing Conference and Workshops*, pp. 1–8, July 2011.

[37] Berger, E., McKinley, K., Blumofe, R., and Wilson, P., "Hoard: A scalable memory allocator for multithreaded applications," tech. rep., Austin, TX, USA, 2000.

[38] Berger, E. D., Zorn, B. G., and McKinley, K. S., "Reconsidering custom memory allocation," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, (New York, NY, USA), pp. 1–12, ACM, 2002.

[39] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S., "Extensibility safety and performance in the spin operating system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 267–283, ACM, 1995.

[40] Bhattacharjee, A., "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 383–394, ACM, 2013.

[41] Blackburn, S. M. and McKinley, K. S., "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 22–32, ACM, 2008.

[42] Bonwick, J. and Microsystems, S., "The slab allocator: An object-caching kernel memory allocator," in *In USENIX Summer*, pp. 87–98, 1994.

[43] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N., "An analysis of linux scalability to many cores," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[44] Boyd-Wickizer, S., Kaashoek, F., Morris, R., and Zeldovich, N., "OpLog: a library for scaling update-heavy data structures," Tech. Rep. MIT-CSAIL-TR-2014-019, Computer Science and Artificial Intelligence Laboratory, MIT, September 2014.

[45] Brightwell, R., Pedretti, K., and Hudson, T., "SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-core Processor," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Austin, Texas), pp. 25:1–25:12, 2008.

[46] Chakrabarti, D. R., Boehm, H.-J., and Bhandari, K., "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, (New York, NY, USA), pp. 433–452, ACM, 2014.

[47] Chase, J. S., Levy, H. M., Baker-harvey, M., and Lazowska, E. D., "How to Use a 64-Bit Virtual Address Space," tech. rep., Department of Computer Science and Engineering, University of Washington, 1992.

[48] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D., "Sharing and protection in a single-address-space operating system," *ACM Trans. Comput. Syst.*, vol. 12, pp. 271–307, Nov. 1994.

[49] Chase, J. S., Levy, H. M., Lazowska, E. D., and Baker-Harvey, M., "Lightweight Shared Objects in a 64-bit Operating System," in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '92, (Vancouver, British Columbia, Canada), pp. 397–413, 1992.

[50] Chen, S. and Jin, Q., "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, pp. 786–797, Feb. 2015.

[51] Cheng, Y., Gupta, A., and Butt, A. R., "An in-memory object caching framework with adaptive load balancing," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 4:1–4:16, ACM, 2015.

[52] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S., "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, pp. 1804–1815, Aug. 2015.

[53] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R., "Memshare: a dynamic multi-tenant key-value cache," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 321–334, USENIX Association, 2017.

[54] Clements, A. T., Kaashoek, M. F., and Zeldovich, N., "Scalable address spaces using rcu balanced trees," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 199–210, ACM, 2012.

[55] Clements, A. T., Kaashoek, M. F., and Zeldovich, N., "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 211–224, ACM, 2013.

[56] Conway, A., Bakshi, A., Jiao, Y., Jannen, W., Zhan, Y., Yuan, J., Bender, M. A., Johnson, R., Kuszmaul, B. C., Porter, D. E., and Farach-Colton, M., "File systems fated for senescence? nonsense, says science!," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, (Santa Clara, CA), pp. 45–58, USENIX Association, 2017.

[57] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R., "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

[58] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M., "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, (New York, NY, USA), pp. 381–394, ACM, 2013.

[59] David, T., Guerraoui, R., and Trigonakis, V., "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 33–48, ACM, 2013.

[60] David, T., Guerraoui, R., and Trigonakis, V., "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 631–644, ACM, 2015.

[61] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[62] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M., "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 401–414, USENIX Association, 2014.

[63] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J., "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 15:1–15:15, ACM, 2014.

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

[65] El Hajj, I., Merritt, A., Zellweger, G., Milojicic, D., Achermann, R., Faraboschi, P., Hwu, W.-m., Roscoe, T., and Schwan, K., "SpaceJMP: Programming with Multiple Virtual Address Spaces," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 353–368, ACM, 2016.

[66] Elphinstone, K. and Heiser, G., "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 133–150, ACM, 2013.

[67] Engler, D. R. and Kaashoek, M. F., "Exterminate all operating system abstractions," in *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, HOTOS '95, (Washington, DC, USA), pp. 78–, IEEE Computer Society, 1995.

[68] Evans, J., "A scalable concurrent malloc(3) implementation for freebsd," 2006.

[69] Faraboschi, P., Keeton, K., Marsland, T., and Milojicic, D., "Beyond processor-centric operating systems," in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2015.

[70] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W., "Sap hana database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, pp. 45–51, Jan. 2012.

[71] Filippova, D. and Kingsford, C., "Rapid, Separable Compression Enables Fast Analyses of Sequence Alignments," in *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '15, (New York, NY, USA), pp. 194–201, ACM, 2015.

[72] Fitzpatrick, B., "Distributed caching with memcached," *Linux J.*, vol. 2004, pp. 5–, Aug. 2004.

[73] Fraser, K., "Practical lock-freedom," Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[74] Gandhi, J., Basu, A., Hill, M. D., and Swift, M. M., "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 178–189, IEEE Computer Society, 2014.

[75] Gaud, F., Lepers, B., Decouchant, J., Funston, J., Fedorova, A., and Quema, V., "Large pages may be harmful on numa systems," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 231–242, USENIX Association, June 2014.

[76] Gaud, F., Lepers, B., Funston, J., Dashti, M., Fedorova, A., Quéma, V., Lachaize, R., and Roth, M., "Challenges of memory management on modern numa systems," *Commun. ACM*, vol. 58, pp. 59–66, Nov. 2015.

[77] Gerber, S., Zellweger, G., Achermann, R., Kourtis, K., Roscoe, T., and Milojicic, D., "Not your parents' physical address space," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, (Kartause Ittingen, Switzerland), USENIX Association, May 2015.

[78] Ghemawat, S. and Dean, J., "Leveldb." `https://github.com/google/leveldb`, 2016.

[79] Gog, I., Giceva, J., Schwarzkopf, M., Vaswani, K., Vytiniotis, D., Ramalingam, G., Costa, M., Murray, D. G., Hand, S., and Isard, M., "Broom: Sweeping out garbage collection from big data systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, (Kartause Ittingen, Switzerland), USENIX Association, 2015.

[80] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I., "Scaling concurrent log-structured data stores," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 32:1–32:14, ACM, 2015.

[81] Google, "The go programming language." `https://www.golang.org/`, 2017.

[82] Gostin, G., Collard, J.-F., and Collins, K., "The architecture of the hp superdome shared-memory multiprocessor," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 239–245, ACM, 2005.

[83] Graefe, G., Volos, H., Kimura, H., Kuno, H., Tucek, J., Lillibridge, M., and Veitch, A., "In-memory performance for big data," *Proceedings of the VLDB Endowment*, vol. 8, pp. 37–48, 2014.

[84] Gramoli, V., "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 1–10, ACM, 2015.

[85] Guerraoui, R. and Trigonakis, V., "Optimistic concurrency with optik," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), pp. 18:1–18:12, ACM, 2016.

[86] Hamilton, J., "The cost of latency." `http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/`, 2009.

[87] Hand, S. M., "Self-paging in the Nemesis Operating System," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (New Orleans, Louisiana, USA), pp. 73–86, 1999.

[88] Heiser, G. and Elphinstone, K., "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Trans. Comput. Syst.*, vol. 34, pp. 1:1–1:29, Apr. 2016.

[89] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J., "The mungi single-address-space operating system," *Software: Practice and Experience*, vol. 28, no. 9, pp. 901–928, 1998.

[90] Hertz, M. and Berger, E. D., "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 313–326, ACM, 2005.

[91] Howard, P. W. and Walpole, J., "Relativistic red-black trees," *Concurr. Comput. : Pract. Exper.*, vol. 26, pp. 2684–2712, Nov. 2014.

[92] Huang, J., Schwan, K., and Qureshi, M. K., "Nvram-aware logging in transaction systems," *Proc. VLDB Endow.*, vol. 8, pp. 389–400, Dec. 2014.

[93] Johnson, F. R., Stoica, R., Ailamaki, A., and Mowry, T. C., "Decoupling contention management from scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 117–128, ACM, 2010.

[94] Johnstone, M. S. and Wilson, P. R., "The memory fragmentation problem: Solved?," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, (New York, NY, USA), pp. 26–36, ACM, 1998.

[95] Kaestle, S., Achermann, R., Roscoe, T., and Harris, T., "Shoal: Smart allocation and replication of memory for parallel programs," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 263–276, USENIX Association, July 2015.

[96] Kannan, S., Gavrilovska, A., and Schwan, K., "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 13:1–13:16, ACM, 2016.

[97] Karakostas, V., Gandhi, J., Ayar, F., Cristal, A., Hill, M. D., McKinley, K. S., Nemirovsky, M., Swift, M. M., and Ünsal, O., "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (Portland, Oregon), pp. 66–78, 2015.

[98] Katcher, J., "Postmark: a new file system benchmark.." Network Appliance Tech Report TR3022, Oct. 1997.

[99] Keeton, K., "Memory-driven computing," (Santa Clara, CA), USENIX Association, 2017.

[100] Kejriwal, A., Gopalan, A., Gupta, A., Jia, Z., Yang, S., and Ousterhout, J., "Slik: Scalable low-latency indexes for a key-value store," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, (Denver, CO), USENIX Association, June 2016.

[101] Kimura, H., "Foedus: Oltp engine for a thousand cores and nvram," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 691–706, ACM, 2015.

[102] Koldinger, E. J., Chase, J. S., and Eggers, S. J., "Architecture Support for Single Address Space Operating Systems," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, (Boston, Massachusetts, USA), pp. 175–186, 1992.

[103] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V., "K42: Building a complete operating system," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, (New York, NY, USA), pp. 133–145, ACM, 2006.

[104] Kuszmaul, B. C., "Supermalloc: A super fast multithreaded malloc for 64-bit machines," in *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, (New York, NY, USA), pp. 41–55, ACM, 2015.

[105] Kwon, Y., Yu, H., Peter, S., Rossbach, C. J., and Witchel, E., "Coordinated and efficient huge page management with ingens," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (GA), pp. 705–721, USENIX Association, 2016.

[106] Larson, P.-A. and Krishnan, M., "Memory allocation for long-running server applications," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, (New York, NY, USA), pp. 176–185, ACM, 1998.

[107] Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H., "WORT: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, (Santa Clara, CA), pp. 257–270, USENIX Association, 2017.

[108] Lepers, B., Quema, V., and Fedorova, A., "Thread and memory placement on numa systems: Asymmetry matters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 277–289, USENIX Association, July 2015.

[109] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., and Subgroup, . G. P. D. P., "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[110] Li, K., "IVY: A Shared Virtual Memory System for Parallel Computing," *ICPP (2)*, vol. 88, p. 94, 1988.

[111] Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J., "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 27:1–27:14, ACM, 2014.

[112] Liedtke, J., "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 237–250, ACM, 1995.

[113] Liedtke, J., "Improving ipc by kernel design," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, (New York, NY, USA), pp. 175–188, ACM, 1993.

[114] Liedtke, J., "Improved address-space switching on Pentium processors by transparently multiplexing user address spaces," tech. rep., Center for Information Technology, Sankt Augustin, 1995.

[115] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M., "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 1–13, ACM, 2011.

[116] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M., "Mica: A holistic approach to fast in-memory key-value storage," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (Seattle, WA), pp. 429–444, USENIX Association, Apr. 2014.

[117] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch, T. F., "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 36–47, ACM, 2013.

[118] Lin, F. X. and Liu, X., "Memif: Towards programming heterogeneous memory asynchronously," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 369–383, ACM, 2016.

[119] Lomet, D. B., Sengupta, S., and Levandoski, J. J., "The bw-tree: A b-tree for new hardware platforms," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, (Washington, DC, USA), pp. 302–313, IEEE Computer Society, 2013.

[120] Maass, S., Min, C., Kashyap, S., Kang, W., Kumar, M., and Kim, T., "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), pp. 527–543, ACM, 2017.

[121] Maass, S., Min, C., Kashyap, S., Kang, W., Kumar, M., and Kim, T., "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), pp. 527–543, ACM, 2017.

[122] Mahon, M. J., Lee, R. B.-L., Miller, T. C., Huck, J. C., and Bryg, W. R., "The Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, vol. 37, pp. 16–22, August 1986.

[123] Mao, Y., Kohler, E., and Morris, R. T., "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, (New York, NY, USA), pp. 183–196, ACM, 2012.

[124] Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., and Anderson, T. E., "Improving the performance of log-structured file systems with adaptive methods," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, (New York, NY, USA), pp. 238–251, ACM, 1997.

[125] Matveev, A., Shavit, N., Felber, P., and Marlier, P., "Read-log-update: A lightweight synchronization mechanism for concurrent programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 168–183, ACM, 2015.

[126] McKenney, P., *Exploiting Deferred Destruction: An Analysis of Read- Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, July 2004.

[127] McKenney, P. E., Boyd-Wickizer, S., and Walpole, J., "RCU Usage In the Linux Kernel: One Decade Later," tech. rep., September 2012.

[128] McKenney, P. E. and Slingwine, J. D., "Read-copy update: Using execution history to solve concurrency roblems," in *Parallel and Distributed Computing and Systems*, (Las Vegas, NV), pp. 509–518, October 1998.

[129] Mellor-Crummey, J. M. and Scott, M. L., "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, Feb. 1991.

[130] Metreveli, Z., Zeldovich, N., and Kaashoek, M. F., "Cphash: A cache-partitioned hash table," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 319–320, ACM, 2012.

[131] Michael, M. M., "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes," in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, (New York, NY, USA), pp. 21–30, ACM, 2002.

[132] Michael, M. M. and Scott, M. L., "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, (New York, NY, USA), pp. 267–275, ACM, 1996.

[133] Min, C., Kashyap, S., Maass, S., and Kim, T., "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, (Denver, CO), USENIX Association, June 2016.

[134] Mitchell, C., Geng, Y., and Li, J., "Using one-sided rdma reads to build a fast, cpu-efficient key-value store," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 103–114, USENIX Association, 2013.

[135] Mozilla, "The rust programming language." `https://www.rust-lang.org/`, May 2016.

[136] Mozilla, "Crossbeam – Support for concurrent and parallel programming.." `https://docs.rs/crossbeam`, 2017.

[137] Nanavati, M., Schwarzkopf, M., Wires, J., and Warfield, A., "Non-volatile storage," *Queue*, vol. 13, pp. 20:33–20:56, Nov. 2015.

[138] Navarro, J., Iyer, S., Druschel, P., and Cox, A., "Practical, transparent operating system support for superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, (New York, NY, USA), pp. 89–104, ACM, 2002.

[139] Nguyen, K., Fang, L., Xu, G., Demsky, B., Lu, S., Alamian, S., and Mutlu, O., "Yak: A high-performance big-data-friendly garbage collector," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (GA), pp. 349–365, USENIX Association, 2016.

[140] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V., "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 385–398, USENIX, 2013.

[141] Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., and Grot, B., "Scale-out numa," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 3–18, ACM, 2014.

[142] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E., "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[143] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R., "The case for ramcloud," *Commun. ACM*, vol. 54, pp. 121–130, July 2011.

[144] Petrović, D., Shahmirzadi, O., Ropars, T., and Schiper, A., "High-performance rma-based broadcast on the intel scc," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, (New York, NY, USA), pp. 121–130, ACM, 2012.

[145] Plimpton, S., Brightwell, R., Vaughan, C., Underwood, K., and Davis, M., "A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark," in *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, pp. 1–7, Sept 2006.

[146] Psaroudakis, I., Scheuer, T., May, N., Sellami, A., and Ailamaki, A., "Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement," *PVLDB*, vol. 8, no. 12, pp. 1442–1453, 2015.

[147] Reble, P., Lankes, S., Fischer, F., and Müller, M. S., "Effective communication for a system of cluster-on-a-chip processors," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, (New York, NY, USA), pp. 122–131, ACM, 2015.

[148] Ren, K. and Gibson, G., "Tablefs: Enhancing metadata efficiency in the local file system," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 145–156, USENIX, 2013.

[149] Rosenblum, M. and Ousterhout, J. K., "The design and implementation of a log-structured file system," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, (New York, NY, USA), pp. 1–15, ACM, 1991.

[150] Rumble, S., *Memory and Object Management in RAMCloud*. PhD thesis, Stanford University, Mar. 2014.

[151] Rumble, S. M., Kejriwal, A., and Ousterhout, J., "Log-structured memory for dram-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2014.

[152] Sanfilippo, S., "Redis." `http://redis.io`, May 2015.

[153] Schneider, S., Antonopoulos, C. D., and Nikolopoulos, D. S., "Scalable locality-conscious multithreaded memory allocation," in *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, (New York, NY, USA), pp. 84–94, ACM, 2006.

[154] Schurman, E. and Brutlag, J., "The user and business of server delays." `http://conferences.oreilly.com/velocity/velocity2009/public/schedule/detail/8523`, 2009.

[155] Sears, R. and Ramakrishnan, R., "blsm: A general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 217–228, ACM, 2012.

[156] Seshadri, V., Pekhimenko, G., Ruwase, O., Mutlu, O., Gibbons, P. B., Kozuch, M. A., Mowry, T. C., and Chilimbi, T., "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (Portland, Oregon), pp. 79–91, 2015.

[157] Sha, L., Rajkumar, R., and Lehoczky, J. P., "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, pp. 1175–1185, Sep 1990.

[158] Shahriyar, R., Blackburn, S. M., and McKinley, K. S., "Fast conservative garbage collection," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, (New York, NY, USA), pp. 121–139, ACM, 2014.

[159] Smith, K. A. and Seltzer, M., "A comparison of ffs disk allocation policies," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 1996.

[160] Tene, G., Iyengar, B., and Wolf, M., "C4: The continuously concurrent compacting collector," in *Proceedings of the International Symposium on Memory Management*, ISMM '11, (New York, NY, USA), pp. 79–88, ACM, 2011.

[161] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R., "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, pp. 1626–1629, Aug. 2009.

[162] Triplett, J., McKenney, P. E., and Walpole, J., "Resizable, scalable, concurrent hash tables via relativistic programming," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2011.

[163] Tsai, C.-C., Zhan, Y., Reddy, J., Jiao, Y., Zhang, T., and Porter, D. E., "How to get more value from your file system directory cache," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 441–456, ACM, 2015.

[164] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E., "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[165] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M., "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 14:1–14:14, ACM, 2014.

[166] Wang, Q., Stamler, T., and Parmer, G., "Parallel sections: Scaling system-level data-structures," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 33:1–33:15, ACM, 2016.

[167] Wang, T. and Johnson, R., "Scalable logging through emerging non-volatile memory," *PVLDB*, vol. 7, no. 10, pp. 865–876, 2014.

[168] White, S. J. and DeWitt, D. J., "Quickstore: A high performance mapped object store," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, (New York, NY, USA), pp. 395–406, ACM, 1994.

[169] Wiggins, A. and Heiser, G., "Fast address-space switching on the strongarm sa-1100 processor," in *Proceedings 5th Australasian Computer Architecture Conference. ACAC 2000 (Cat. No.PR00512)*, pp. 97–104, 2000.

[170] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D., "Dynamic storage allocation: A survey and critical review," pp. 1–116, Springer-Verlag, 1995.

[171] Witchel, E., Cates, J., and Asanović, K., "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (San Jose, California), pp. 304–316, 2002.

[172] Witchel, E., Rhee, J., and Asanović, K., "Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, (Brighton, United Kingdom), pp. 31–44, 2005.

[173] Wu, M., Yang, F., Xue, J., Xiao, W., Miao, Y., Wei, L., Lin, H., Dai, Y., and Zhou, L., "Gram: Scaling graph computation to the trillions," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, (New York, NY, USA), pp. 408–421, ACM, 2015.

[174] Wu, X., Zhang, L., Wang, Y., Ren, Y., Hack, M., and Jiang, S., "zexpander: A key-value cache with both high performance and fewer misses," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 14:1–14:15, ACM, 2016.

[175] Xia, F., Jiang, D., Xiong, J., and Sun, N., "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 349–362, USENIX Association, 2017.

[176] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[177] Zahir, R., Ross, J., Morris, D., and Hess, D., "OS and Compiler Considerations in the Design of the IA-64 Architecture," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, (Cambridge, Massachusetts, USA), pp. 212–221, 2000.

[178] Zhao, Y., Li, S., Hu, S., Wang, H., Yao, S., Shao, H., and Abdelzaher, T., "An experimental evaluation of datacenter workloads on low-power embedded micro servers," *Proc. VLDB Endow.*, vol. 9, pp. 696–707, May 2016.

[179] Ziwisky, M., Persohn, K., and Brylow, D., "A down-to-earth educational operating system for up-in-the-cloud many-core architectures," *Trans. Comput. Educ.*, vol. 13, pp. 4:1–4:12, Feb. 2013.