SCALABLE MAIN-MEMORY OBJECT MANAGEMENT

A Thesis Proposal
Presented to
The Academic Faculty

by

Alexander Merritt

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

Georgia Institute of Technology
May 2016

SCALABLE MAIN-MEMORY OBJECT MANAGEMENT

Approved by:

Dr. 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 Kishore Ramachandran
School of Computer Science
Georgia Institute of Technology

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

Dr. Dejan Milojicic
Hewlett Packard Labs
Hewlett Packard Enterprise

Date Approved: TBD

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

MOTIVATION AND PROBLEM STATEMENT

Online information – collected, crawled, and extracted from numerous sources – drives today's applications and user-facing services. This has become the defining characteristic for the majority of data center applications and workload configurations today: very large data sets. Many companies consume thousands to millions of images, events, interactions, app downloads, video streams, and messages every minute [13]. Facebook reportedly holds 64PiB of information [100] collected from over 1 billion of their active users [30], and Google handles over 18 billion search queries per month [11]. Many user-centric services online support real-time image feature detection [49, 68], or exploration [6], and act as large datasets to support research in these areas [41]. Metadata such as networks found on common social media websites – LinkedIn, Google+, Facebook, etc. – contain millions of edges [79]. These form the basis for predicting trends [57], making recommendations, ad suggetions, or are even exposed directly to users to query [8]. Thus, accomodating such growth in information becomes a priority for systems researchers, no matter what level in the stack.

Data is a central focus of applications and services today, but such large data sets grew in size faster than systems could accommodate. Unable to fit in a single system's memory, researchers and industry have developed large scalable distributed systems to process these large volumes of information. These include batch processing systems like MapReduce [40], neural network training systems [39], stream processing [119, 67, 66], graph processing [30, 55, 56], many databases [115, 35, 97], general NoSQL data-service services [112, 48] and many others. Keeping data in memory is such a significant priority for performance [58], that various techniques have been studied, such as alternate representations of graphs [109], custom programming models that leverage compression [19], use of networks to access remote memories [103], avoiding local disk lookups, or heirarchical data storage [83].

Such code bases are, unfortunately, complex and buggy [127], and require tolerance of many levels of failures in the network [60, 86, 18]. As applications increasingly demand low-latency response times [61, 113], even for these large-scale analytics of massive datasets [126, 80], development has shifted to higher-performance network fabrics, such as InfiniBand [103, 42, 69, 98, 81]. Trying to minimize use of the network entirely is the focus of many distributed systems, but remains a challenge

where access patterns are random [89], making data partitioning extremely difficult.

New and emerging memory technologies are enabling platforms to contain compositions of massive hybrid memories – big enough to hold certain analytics working sets, avoiding use of networked system designs. Navigating such platforms is complex as they are characterized as having:

1. massive memory pools, in the order of hundreds of terabytes and more [15];
2. heterogeneous memories [15, 12, 77, 31, 105];
3. non-uniform topologies, with varying access latencies and (disaggregated) bandwidths [78],
4. one or more physical memory address spaces [52];
5. cache coherence domains (e.g., ARM [17]);

Systems developers are reconsidering the importance of single-machine designs [21, 94] due to the scalability and performance gains over distributed systems. This has led to the development of many systems optimized for using main memory to hold their computations [128, 72, 73, 114, 129]. *Building* shared-memory (software) systems is arguably easier, and with so much fast byte-addressable memory available, data-centric applications and services are easily colocated on the same platform. This thesis focuses on such systems and the challenges they present for efficient programming.

1.1 Challenges

Very large single-machine systems are themselves a challenge to program *efficiently*, as all layers of the software stack are affected:

(1) Resources on these systems are not uniform as applications and systems software were once able to assume [78, 70]. The most obvious resource challenge is the system's *disaggregated memory bandwidth*: tens of terabytes requires tens of sockets to accommodate [16]. The implication is that applications and runtimes must become aware of the placement of their data, not just due to the non-uniform access latencies, but due to the potentially limited bandwidth available for a given chunk of physical memory. For example, simple optimizations such the operating system "upgrading" virtual memory page sizes can cause memory bandwidth hotspots [51], as large page translations can limit dynamic data placement decisions to balance load. Heap allocators [124] primarily optimize for efficient use of *virtual* memory, not physical, meaning they are unaware of data placement requirements by applications, and can also prevent platform semantics from becoming visible in applications: applications have to do some of the work themselves [44]. A simple example

of where this lack of visibility and control becomes apparent is in the programming of very large data structures, where the backing memory for a hash table is allocated on one socket [38]. Performance bottlenecks when scaling up as the single socket's bandwidth becomes saturated.

(2) The ability to leverage full system bandwidth requires *extreme scale*: sockets can host 20-40 high-performance cores or more [17], for an aggregate of many hundreds of cores across the system. Scaling at this extreme exposes bottlenecks at all layers of the stack. Monolithic operating systems such as Linux are complex software systems and suffer from scalability issues [28]. For example, highly parallel applications are unable to manipulate their virtual memory mappings concurrently due to the use of blocking data structures in the memory subsystem [33, 32]. Others have argued we must completely redesign the operating system to look and act like a distributed system [23], with the same question posed for applications themselves on these systems, such as for graph processing [125] where message passing enables greater scalability than shared memory designs.

Data structures form the basis of data organization for applications and system software. On hundreds of cores, these structures – even with fine-granularity locking – suffer from a multitude of challenges, such as false sharing of cache lines on ccNUMA systems [106], an often hidden source of bottlenecks that are difficult to pinpoint [101]. There have been numerous efforts to scale data structures often avoiding the use of locks altogether, such as for concurrent access to hash tables [82] or in resizing them [120], search trees [75, 90, 65], or simple lists [62], and many more [38, 59, 37]. Nevertheless, these data structures must *still be aware* of the underlying platform performance and behavioral characteristics to truly scale [38].

(3) The *interfaces and abstractions* applications have to make efficient use of memory are outdated, make resources transparent, and are too coarse to be able to express, semantically, what the relationships are between an application and its data. These problems limit performance, which is unacceptable for today's highly latency-sensitive workflows and data-servicing applications.

Memory is abstracted to applications through translation in the hardware – *virtual memory*. Due to hardware support, the granularity of management is restricted to specific sizes of memory as pages, and allocating actual physical pages is performed lazily on page faults by the operating system. Virtual memory originated from a time when DRAM was scarce and systems were evolving to be multitasking machines; applications could execute while partially in memory, without knowing which pieces were on disk or in memory. Such scenarios are no longer representative of future systems, where all of byte-addressable memory may be a mix of volatile and non-volatile technologies.

Furthermore, data sharing is (and has been) predomenantly facilitated through file system interfaces, arguably an interface that is mismatched for memory [45] and the different kinds of memory

implementations now available [71], or simply must be implemented inside libraries, instead [121] (notably for backwards compatibility). Data must be expressed in terms of the file system interface, including the name, the access rights, etc. Additional properties that pertain to its in-memory state are controlled via "holes" poked through this file interface – `mmap`, `madvise`, and friends. There is no easy means to express how applications can *translate to* their data, or to support more complex operations such as the caching of page tables with overlapping translations (page tables are simply *rebuilt* on-demand): virtual memory abstractions themselves are meant to be transparent to applications. With little control over organization of in-memory data: (a) scenarios where memory mapping operations dominate latencies in the critical path can destroy performance [45, 29]; (b) it encourages stagnation of modern methods for data handling, where structures continue to be marshaled into and out of buffers – files, kernel-resident regions, TCP/IP or UNIX Domain sockets, etc. – instead of flexibly and directly shared among applications; (c) data *placement* [44] and *movement* operations are suboptimal [85] or ineffective [70] due to the lack of semantics in the interface primitives, as would be needed for load-balancing, more efficient use of bandwidth, or to isolate applications from each other.

Lastly, the importance of latency sensitivity of data-centric applications is greatly supported by new quality-of-service mechanisms in modern CPUs and memory systems (demonstrated at Google [87]), e.g., using Intel's "cache allocation technology" [9] which supports partitioning of last-level CPU caches, and in future implementations, to allocate the equivalent of InfiniBand "lanes" into the memory controller. Neither virtual memory nor the operating system interfaces to manage memory provide any means of expressing semantic requirements for such performance guarantees or isolation other than "first-touch", "never move this page", or "move this page now". Applications instead rely on associated command-line tools, third-party APIs, and specifications to make effective use of this hardware support.

Interfaces must evolve and keep pace with modern memory systems, as even the meaning of even what a physical address is has already begun to change [52]. On large-scale platforms, interfaces are increasingly limiting the performance of managing and interacting with physical memory.

(4) Despite large memory capacities becoming more available, the working set sizes of data-centric applications and the higher per-node costs place increasing importance on ensuring *all layers of the stack minimize capacity waste*; in other words, we do not want the overheads from metadata to encroach on the capacity needed by application data. Even a small overhead can result in large absolute losses of available memory, e.g., a 20% loss of capacity on a 20-terabyte system = four terabytes. Accepting such losses requires either purchasing larger overall system capacities

(expensive) or using multiple systems – returning to the same challenge a large single-node system was meant to address – avoiding the network. Data is also expected to continuously grow in the future, making the large memory capacities a "non-reason" to ignore effective memory management.

One source of bloat can be found in libraries, where heap data is managed by carefully implemented allocators that apply heuristics [124] to handle common application access patterns. As data is kept increasingly in main memory, heap allocators are tasked with managing more than just application contexts – structs, lists, etc. – but also actual memory objects, files, and other complex datasets. Such diverse data characteristics and access patterns stress allocators; they can easily consume 2-6x more memory than they are asked to manage (more in Chapter 2). Memcached [48], for example, uses its own form of slab allocation [27] for low-latency response times (an allocation strategy used in operating system kernels); allocations are often constant-time complexity. However, slab caches only allocate memory of fixed-size chunks. Memcached's interface accepts objects of any size, however, so to reduce fragmentation, the administrator must configure Memcached with an appropriate selection of slab cache sizes. Unfortunately, this cannot be changed without restarting the application, potentially leading to bloat if an application's allocation patterns are not optimal for the chosen slab cache configuration. Object sizes can come from anything using this data-serving tool: web pages, data base entries, simple keys, large log files, small real-time events, etc. With main memory used more and more to hold objects from a broad composition of sources, including behaving like a file system [43], heap allocators as they are today will struggle with fragmentation, when used as the underlying allocators for object storage.

Furthermore, heap allocator scalability techniques can lead to other kinds of fragmentation. For example, the use of per-thread page caches [76, 25, 47] do not require synchronization among threads for the common cases, but certain patterns where threads allocate large amounts of data and quickly release it result in large pools of memory "held" by some threads. Rebalancing requires some synchronization and is not optimized for the common case.

Within managed runtimes and languages, such as Java and Go, conveniences of heap memory management include the use of garbage collectors. Garbage collectors can significantly reduce memory fragmentation, but their main drawback is how much they block application threads during cleaning [36], and on systems with many cores and memory objects, how scalable they can be made [53, 54]. Even in modern languages like Go, garbage collection latencies are a challenge [116].

1.2 Thesis Statement

To achieve efficiency, scalability and performance on emerging massive shared-memory platforms, systems software requires new memory-centric interfaces and improved mechanisms for managing placement of and accesses to the partitioned, decentralized application state across the complex memory substrate.

1.3 Specific Contributions

This thesis makes the following contributions:

1. A characterization of the limitations and trade-offs of existing modern memory allocator designs on large-scale systems;
2. Design and implementation of a scalable log-structured object store using concurrent data structures, state partitioning, and efficient hardware primitives (e.g., DMA);
3. Design and implementation of a memory subsystem within an operating system to support multiple composable virtual address spaces per process;*
4. Empirical evaluation comparing (2) with modern memory allocators on scalability, fragmentation, and allocation latency using standard benchmarks;
5. Characterization of memory placement/management strategies and compute vs data movement policies on the scalability and latency of data access of (2) using accepted benchmarks and workloads, e.g., comparing to modern data-centric object stores like Redis;
6. Optimization on (2) using composable address spaces from (3) to reduce data movement costs and improve scalability using semantic knowledge about data access characteristics (compared to legacy OS interfaces and IPC);
7. Evaluation of end-to-end performance of a large multi-threaded image analytics workflow designed with both (2) and (3), and compared against use of alternative data stores, on a single system;

All evaluation will be performed on HP Superdome X servers.

*Regarding item (2): this contribution arose from an equal research effort of two other colleagues and myself, and was published in ASPLOS'16 [45]. I am one of three primary authors.

CHAPTER II

WHY SCALABILITY AND CAPACITY MANGEMENT ARE HARD

This chapter reviews prior work surrounding the challenges this thesis aims to address.

Why Not Heap Allocators? Highly multithreaded data-intensive applications rely on efficient memory object management from underlyling heap allocators. The default implementation of the general-purpose `malloc` and `free` is the GNU C library allocator. Originally based on dlmalloc [1] and more recently on ptmalloc [3], it once used a global allocation lock; subsequent iterations enabled multi-threading, but introduced more fragmentation. Alternative implementations have arisen specifically to scale allocations among threads, such as SuperMalloc [76], jemalloc [47], and the Hoard allocator [25]. These implementations use thread-local, and thus lockless, pools of memory to avoid costly synchronization in the critical path. Unfortunately, it is a necessary evil, as eventually these pools must be rebalanced, should some threads have greater, temporary, allocation demands than others. Not keeping this under control, under tight latency requirements, can lead to fragmentation. Challenges also arise from ownership changes – when one thread allocates an object, and hands it off to another thread to release. Scalable fragmentation control is typically never a high priority in practice; TCMalloc [4] developed at Google does not return memory to the OS, and the developers admit TCMalloc "...may be somewhat more memory hungry than other mallocs".

Object allocators, based on designs such as the slab allocator [27], can quickly allocate objects using object pools, which allocate object of a fixed-size. Their application has benefitted operating system kernels, where the vast majority of objects are statically known C `structs`.

Use of object allocators in user-level for management of general-purpose memory management is often ill-fated if done poorly, resulting in extreme fragmentation if the set of object pools is not well-chosen in advance to match the object sizes it will manage when deployed. Such is the case for memcache [48], which implements custom slab allocators to avoid scalability issues of the GNU C allocator. Use of slabs has been successful for process heap data in jemalloc, but this results in a very complex design to optimize for fragmentation. Earlier proposals have combined both low-fragmentation and scale, such as with Reaps [26] where run-ahead allocation is used, but upon freeing it dynamically constructs a freelist.

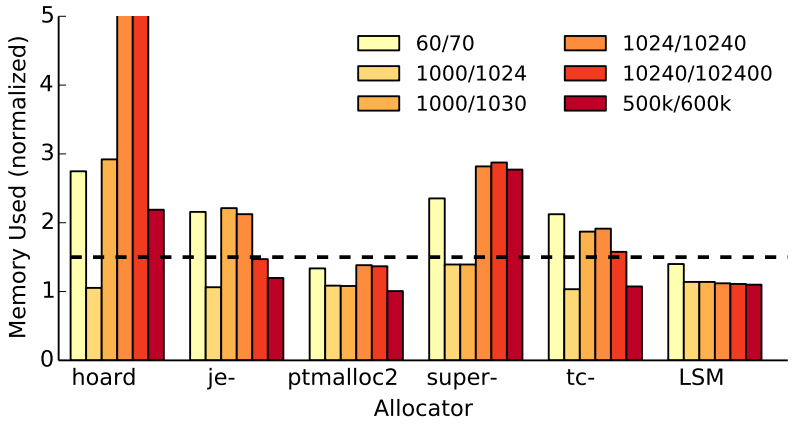


Figure 1: Fragmentation of current heap allocators.

Pattern	Δ
60 B → 70 B	0.16
1000 B → 1024 B	0.024
1000 B → 1030 B	0.03
1 KiB → 10 KiB	10.0
10 KiB → 100 KiB	10.0
500 KiB → 600 KiB	1.20

Table 1: Synthetic allocation workloads used to evaluate allocator bloat. Objects of one fixed size are allocated to "capacity" of the given memory size, 90% deallocated, and then objects of the second fixed size are allocated to capacity.

General-purpose allocator policies can nevertheless result in situations where extreme fragmentation results [111, 124]. This is due to the inability to relocate objects under management.

Figure 1 illustrates just this: we measured the fragmentation of many modern heap allocators on one socket using a single thread following the allocation behaviors from Table 1. We measured the ratio of the total size allocated from invocations of `malloc` to the `vmsize` metric for the process, as reported in `/proc`. The dotted line represents a "goodness" boundary at 50%: below this, fragmentation is considered tolerable (for this experiment).

It is worthwhile to mention a few points: small changes in initial and subsequent object sizes allocated can have a drastic effect on the fragmentation characteristics, as exemplified with Hoard on the 1000/1024 and 1000/1030 patterns – a difference in *6 bytes* results in a growth in the vm size of 3x (for jemalloc this is 2x). ptmalloc and SuperMalloc are unaffected. LSM is a log-structured implementation, and demonstrates high tolerance to all example microbenchmarks.

Furthermore, object size can affect the scalability and latency of object allocation. For example, the GNU C allocator [14] defines arbitrary allocation size thresholds, which upon allocation, (1) searches by best-fit, (2) uses an object cache, or (3) falls back to memory mapping in the OS

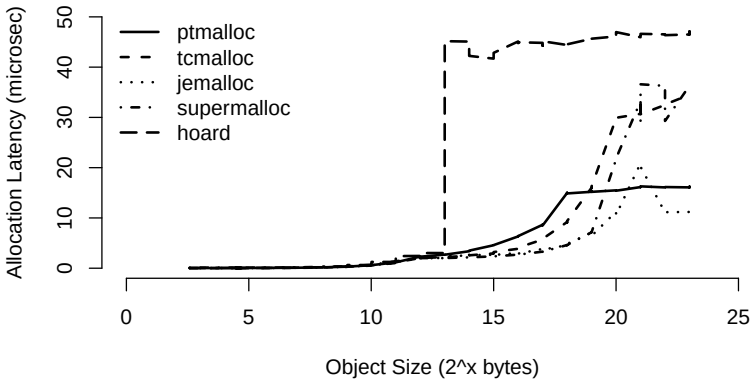


Figure 2: Effects of thresholds on allocation latency, across object sizes.

The Hoard [25] allocator exhibits similar behaviors. Memory mapping per-object is horrible for performance [45], making the performance and scalability of the allocator depend further on the operating system [33]. Figure 2 illustrates this effect; the Hoard allocator begins using `mmap` as a fall-back. Heap allocators in general are meant for small object sizes, as they are coded to assume "large" object allocation is rare.

Given this insight, we designed a producer-consumer benchmark to measure the scalability in aggregate throughput of `malloc` and `free` invocations, shown in Figures 3, 4, and 5. The benchmark allocates threads in pairs – one producer invoking `malloc` and one consumer invoking `free`. For fairness between allocators, the producer will touch the first word of the allocation (`ptmalloc` behaves in this way, incurring zero or one page faults). Each pair shares access to a non-blocking queue where messages (the pointer addresses) are asynchronously appended and popped off. This design is intended to model application workflows or pipelines that allocate data and pass them downstream for consumption, or a key-value store implemented with a thread-pool, where the allocating thread may not be the one to deallocate it. Statically-chosen allocation sizes were used throughout each execution. Threads were pinned in consecutive order on an HP Superdome X system, with 240 cores, 16 sockets with Intel Haswell CPUs. A few trends are apparent:

1. larger object sizes wreak havoc on the allocator performance, some due to the use of `mmap`. Surprisingly, `ptmalloc` scales further than all other allocators for only the smallest allocation sizes – 8 and 16 bytes – and even for 64 bytes it out-performs `hoard` and `tcmalloc`. Once sizes grow beyond that, `ptmalloc` does not scale;
2. the most recently touted scalable allocator – `jemalloc` – does not perform as we expected. It breaks down above 150 cores for small sizes, and then performance *drops* with objects larger than 128 bytes;

3. `tcmalloc` and `hoard` do not scale as far with small object sizes. When we zoom in, both scale up to ca. 30-50 cores, but thereafter performance degrades to become worse than a single thread! `hoard` has many lines flat due to the use of `mmap` above a certain object size threshold.

Clearly there are challenges with both old and modern heap allocators, in terms of scale, latency, and throughput on massive-memory systems.

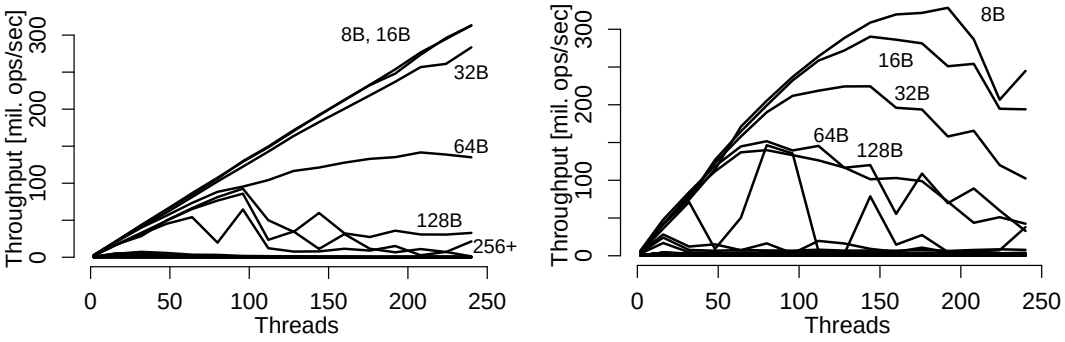
2.1 Scaling on Large Platforms

Data Structures. Scaling to hundreds of cores requires overcoming a multitude of challenges, for coordination and accessing shared data structures. Prior research has proposed to increase the *locking granularity*. In Linux, this has been the case due to the use of the Big Kernel Lock [2], where manual effort required determining the scope of locks more carefully [28].

Hash tables are often used as indexes into databases and key-value stores, and can easily benefit from lock striping [64], but do not support easy resizing, resorting to global locks to allocate a fresh table to migrate all keys into. This is still the case even in recent highly-concurrent hash table implementations like `libcuckoo` [82]. Even without growth of the table, as the utilization increases, the contention becomes a larger bottleneck – unrelated operations will contend for the same buckets. One must either grow the lock table (which is hard [64]) or grow the table itself. Figure 6 shows how a modern application of a lock-striped hash table fails to scale beyond 32 cores on a system with 244 cores; increasing the size of the hash table (given a fixed number of items) will reduce contention, at the expense of wasted memory.

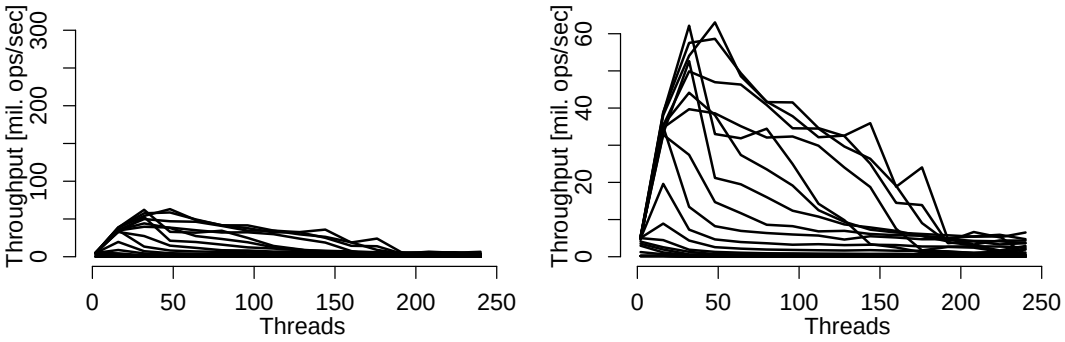
Resizing hash tables is important, as it also allows for balancing memory use with performance. Josh Triplett's relativistic hash table [120] support resizing, using *relatavistic buckets* – zipping buckets together or apart – allowing reads to proceed concurrently. Unfortunately, even these techniques suffer drawbacks: (1) the number of buckets are limited to integral multiples (i.e., resizing is not incremental), (2) inserts and removal operations are blocked until a bucket is unzipped (during growth), and (3) unzipping large buckets is sequential, delaying concurrency convergence after a growth.

Lists and trees have been made concurrent with techniques that are optimized for specific access patterns, such as read-copy-update (RCU) [93] where many readers can execute together with one writer. A recent adaptation called read-log-update (RLU) [92] enables multiple readers and writers. These techniques have been combined into larger data structures to demonstrate high concurrency and low-latency lookups, such as in Masstree [90] and in large-scale in-memory databases like FOEDUS [72] where partitioning of the data across sockets significantly contributes to its scalability.



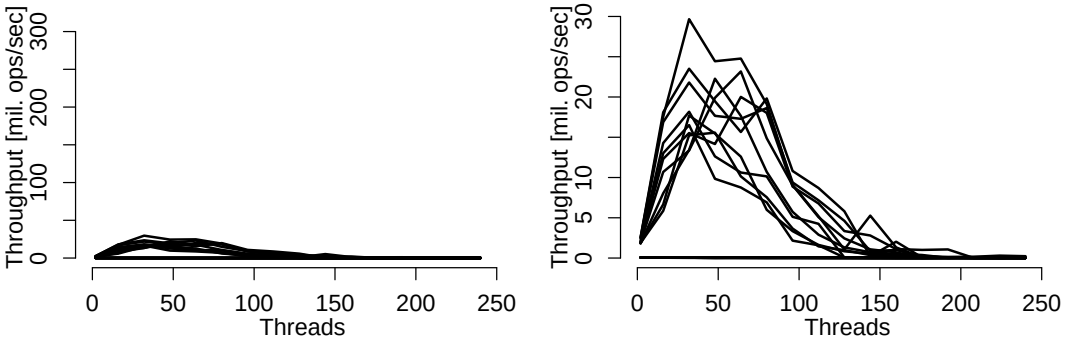
(a) `ptmalloc` scalability (b) `jemalloc` scalability

Figure 3: Producer-consumer benchmark. 8 bytes - 8 MiB chunk sizes evaluated. THP disabled.



(a) `tcmalloc` scalability (b) `tcmalloc` zoomed in

Figure 4: Producer-consumer benchmark. 8 bytes - 8 MiB chunk sizes evaluated. THP disabled.



(a) `Hoard` scalability (b) `Hoard` zoomed in

Figure 5: Producer-consumer benchmark. 8 bytes - 8 MiB chunk sizes evaluated. THP disabled.

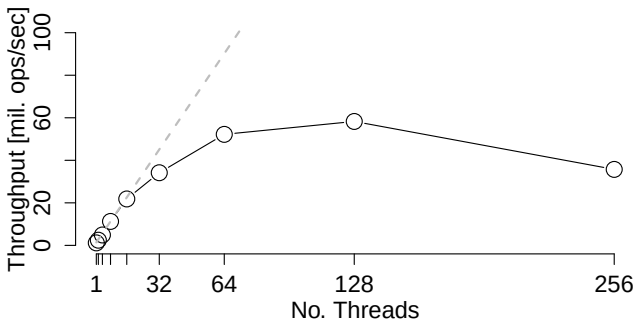


Figure 6: Scalability of a lock-striped hash table in RAMCloud. Workload is entirely reads, 4 mil. 10-Kbyte objects, with 10% of total memory (2.5GiB of 256GiB) allocated to the hash table.

The work of Tudor David et al. [38, 37] evaluates these and other data structures, including hash tables, trees, skip lists, and others. They promote the "asynchronized concurrency" design principles to develop search data structures, but more importantly that *platform-awareness* is vital to scaling. For example, the scalability of a concurrent hash table is limited if the memory pages are not distributed on platforms with disaggregated bandwidth. Their findings also showed that knowledge about the underlying platform can enable simpler designs, yet remain high performance. For example, Intel Xeon processors have a large last-level *inclusive* cache, where the use of simple spin locks can be used (to simplify code) and scale more than using complex queue- or ticket-based locks on a single socket.

Leveraging tens to hundreds of cores for coordination and shared data structures requires understanding concurrency, avoiding blocking operations that operate on disjoint data, and effectively being aware of the underlying hardware characteristics, such as cache coherence, and latencies. Much of the prior research is summarized in academic textbooks [64].

Distributed Principles. Many systems have promoted the use of distributed systems principles to further scale. The Barrelfish [23] operating system and Gram [125] – a shared-memory graph computation framework – advocate the use of message passing on large-scale systems. Most notably, this idea has been extensively used in operating system state as *clustered objects* in K42 [20, 74] (Barrelfish went a step further by distributing the kernel image itself).

Partitioning to Scale. Combining the aggregate performance of many resources can be achieved via partitioning the data to distribute the workloads. Chord [118] demonstrated this with a scalable

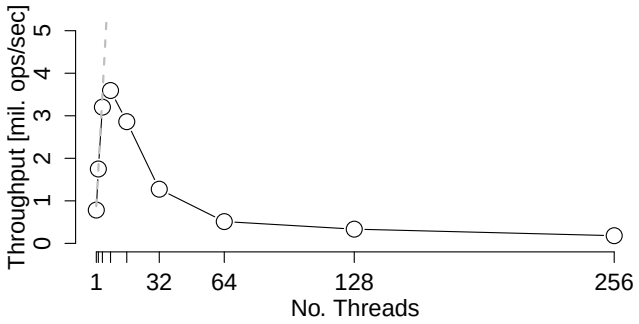


Figure 7: Scalability of a log-structured design using a single log-head in RAMCloud – protected by a single spin lock. 10% of total memory allocated to the hash table. Workload is 5% writes, 1 mil. 10-Kbyte objects.

distributed hash table. More recently, MICA [84], an in-memory key-value store, partitions its data across cores in a system to minimize cross-socket communication – a principle advocated time and time again [38]. The same principle is the core idea behind Barrelfish, and recent research based on it, Arrakis [104], that uses receive-side scaling to distribute inbound network packets across cores for scalable processing.

Memory Reclamation. Further concurrency can be extracted by deferring memory reclamation [95], such as the use of *epochs*, at the behest of avoiding (reader-writer) locks altogether. Techniques such as RCU and RLU make use of known quiescent periods to reclaim memory, allowing concurrent readers (and with RLU, concurrent writers). Use of epochs are a form of *inferred* quiescent period, as they are a measure of logical time. Epochs have been studied extensively [117, 50, 95, 75], a prime example found in the work of dynamic updates in the K42 operating system [24]. In many systems, an epoch is implemented as a globally incremented counter [110] – a potential scalability bottleneck. A recent study [122] has shown the use of epochs to be very effective in scaling on large platforms when implemented with each core's *timestamp counter* (TSC) register – essentially, a global atomic clock (global atomic clocks are also used by Google in their data centers [34]).

Log-Structured Designs. The use of log-structured allocation originated within Sprite [108] and is not new today, but the idea is often used in storage systems where random writes destroy performance, by converting them to sequential. Appending to a log provides – as a side-effect – the creation of object versions, for fast recovery or rollback. It is most recently used in the RAMCloud in-memory distributed key-value store, as the researchers' highest priority was attaining the lowest

latency possible. Bw-Tree [88], a log-based B-tree developed at Microsoft Research, uses the append feature of a log to scale across multi-core systems: each append operation touches unique memory (no other thread will append to the same location). Neither RAMCloud nor Bw-Tree were evaluated on massive-memory systems to substantiate the scalability of their design choices. Figure 7 illustrates an experiment with RAMCloud with a 5% write workload mix on a large system; with even such a small number of writes, having a single log head bottlenecks the system with more than four threads. As shown earlier, Figure 6 illustrates the limited scale of the index in RAMCloud – a balance between memory used and performance.

2.2 Managing Disaggregated Memory

Large NUMA systems are complex and require understanding the complex non-uniform topologies [51, 78]. Much like a distributed system with aggregate bandwidth found across many machines, the same is true for systems with tens of sockets or more. Important for these systems is understanding how other system management decisions, such as choice of virtual memory page sizes [51] play a role in bandwidth allocation, such as to react to hotspots. Efficient primitives must exist to implement decisions that must happen at runtime, such as the movement of physical pages using DMA engines [85, 70]. Databases like VoltDB [97] use replication across NUMA sockets to achieve robustness, at the expense of consuming more memory.

2.3 Efficient Shared-Memory Object Stores

There have been efforts to design high-performance shared-memory object stores, such as QuickStore [123] (from 1994). QuickStore allows applications to store pointer-based data structures in virtual memory, but must make use of pointer-swizzling – an expensive runtime cost – to accomodate data structures that are mapped into the application at later times but are unable to acquire the same virtual address region due to a variety of factors, such as ASLR. QuickStore was designed during a time when main memories were small, and thus the costs of `mmap` (used to compose the address space) were not as costly as is today, with even hundreds of gigabytes imposing overhead on the order of *seconds* [45].

Use of such object stores for large-memory systems will become increasingly important, to take advantage of the performance of byte-addressable latencies and bandwidth.

CHAPTER III

DESIGNING SCALABLE OBJECT MANAGEMENT

The first key contribution of this thesis addresses the challenges of *scaling* both memory object allocation and compaction – to reduce memory fragmentation – and the *effective use of disaggregated bandwidth* to better support applications on large-scale systems. We describe our implementation of a memory object allocation system called *Nibble* that uses a log-structured design to allocate and manage memory objects, is platform-aware by partitioning itself across sockets in the system, uses non-blocking (concurrent) data structures to ensure low latencies on the critical path, and leverages semantic information about application data to make more informed decisions as to appropriate data placement for increased performance, lower latency, etc.

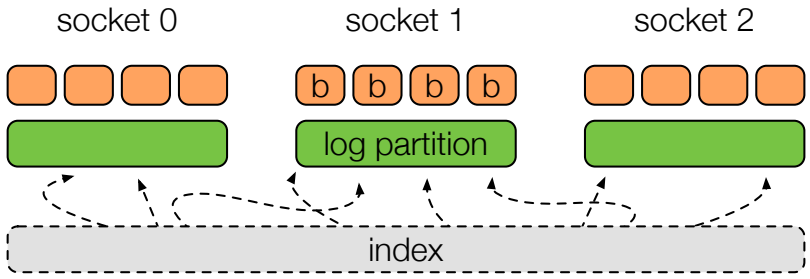


Figure 8: Design of Nibble. The log is logical, and partitioned across sockets. Each partition has separate block and segment allocators, as well as compaction threads.

3.1 High-level Overview

Nibble is a scalable shared-memory memory object allocator. It achieves low fragmentation by organizing objects in a log, and uses an index to enable objects to be relocated throughout the log. Scale is achieved by using concurrent data structures, for example for the index. It also scales by more effectively using the disaggregated memory bandwidth by partitioning its log and replicating log heads – decentralizing itself – to minimize cross-socket communication where possible (Figure 8). Compaction is the main disadvantage to using an index supporting object relocation, as it consumes CPU cycles and memory bandwidth. Nibble explores the use of memory-centric primitives in the hardware to make compaction practical, such as DMA engines, or by isolating compaction tasks from application threads using cache partitioning or scheduling work on separate sockets.

Memory management. Nibble allocates large contiguous ranges of memory and divides them in units called *blocks*, which are contiguous, fixed-sized regions of memory. Blocks are allocated and managed by a *Block Allocator*, are each assigned a unique *slot* identifier, and (currently) are 64KiB in size. This is the unit of memory management within Nibble.

Objects are organized inside of *segments*, which are containers for blocks. The set of blocks in a segment need not be contiguous themselves, as this relaxation allows for finer-granularity memory reclamation when objects are deleted.

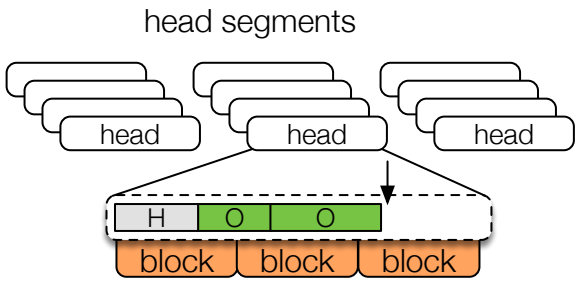


Figure 9: Log structured memory and object management in Nibble. Multiple log heads exist for each socket, and across sockets. Objects are placed contiguously in a segment, and may span across blocks. Small segment headers are placed in each to record pertinent segment-specific information, such as the number of objects (live or not).

Log design. Nibble manages all data in the form of a *log*, which means that there is a given *head reference* pointing to a specific offset within a segment (Figure 9). This head is incremented across successive segments as objects are appended to the log. Segments are either *mutable* or *immutable*. When used as a head, segments are mutable (since they must support appends), and when they become full, the segment becomes immutable – a process called *rolling*.

Index. To avoid traversing the entire log to locate an object, Nibble maintains an index, mapping an object's key to its location within the log. Each new `put` operation translates to an append and an update of the index; each `get` operation queries the index, then copies the data out of the log into a user buffer; delete operations do not touch the log, as they only remove an entry from the index.

As the log does not behave as an update-in-place allocator, each `put` increments the head, and at the same time, invalidates any prior versions of that object. These stale objects consume space and at some point must be reclaimed from the log to be recycled into new head segmenets. This is the role of the *compaction* module. Periodically, background threads will move live object versions out of existing immutable segments into new, empty segments, releasing the original segment's blocks for

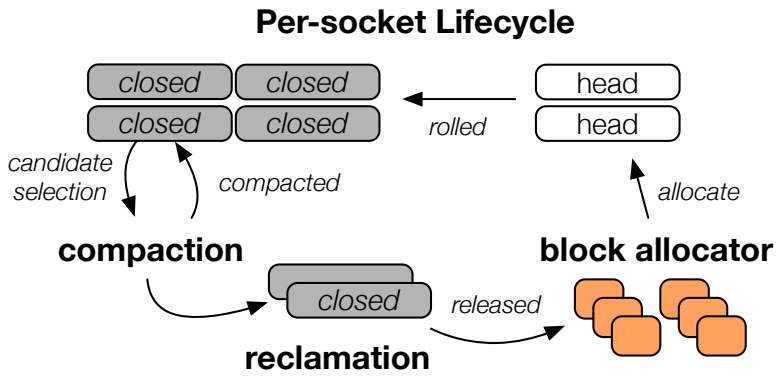


Figure 10: Lifecycle of segments in Nibble. Head segments are *rolled* when full and added to the compaction's candidate pool. Candidates are selected, compacted into new segments (which are returned to the candidate pool), and the old segments handed off for reclamation. Reclamation waits for quiescent phases before being broken down into their blocks for recycling.

subsequent allocations. *Liveness* of an object is determined by checking the index (object keys are stored within segments themselves, thus their keys can be queried in the index during compaction): if a lookup results in nil, the associated object is no longer valid and can be reclaimed; otherwise we verify whether the index points to the specific entry in the log, and if it does not, then the object is stale and can be released. Figure 10 illustrates the entire lifecycle from a high level. Note the reclamation stage – this is an optimization we discuss subsequently.

API and Implementation. Nibble is written in the Rust [99] systems programming language, a compiled language with compiler-enforced memory safety semantics and has no explicit memory management. Instead, there is a strict memory *borrow checker* and reference counting system built into the compiler itself, avoid the use of a garbage collector.

```
struct ObjectDescriptor { key, value }
put_object(obj:  ObjectDescriptor)
get_object(key:  String) → Buffer
del_object(key:  String)
```

Figure 11: Nibble API.

Figure 11 illustrates the public interface. Applications create memory objects in their own heap memory, then can store their data into Nibble via this interface. Adding objects is a blocking operation, returning only when data has been copied from the user buffer to the internal storage. Keys represent unique identifiers to objects.

Next, we discuss each component and techniques used for supporting scalability, efficient compaction, and use of memory bandwidth.

3.2 Quashing Scalability Bottlenecks

Execution on large-scale platforms requires overcoming many scalability bottlenecks. Here, we discuss each of the major components of Nibble's design and what role it plays in affecting its end-to-end performance and scalability.

The basic design of Nibble is to partition its data structures and background threads across the sockets: with N sockets there will be N block allocators, at least N compaction threads, etc. Appending to the log becomes an explicit data movement to a specific location in the system, a specific head in the log, etc.

Index. In our design of Nibble, the index is the only central data structure accessed by multiple components in the system – client operations `put`, `get`, and `del`, and compaction. It plays a central role in the performance of applications, as it sits on the critical path for each of the main functions. Many different designs are available, and as part of this proposal, we aim to evaluate various algorithms, and placement decisions of its backing memory, as well as techniques such as replication. To achieve lower latency, we will want to avoid acquring locks and reduce cache line false sharing.

Load experienced by the index also depends on the specific workload patterns. For example, with billions of small objects, `get` operations can be sustained at much higher throughputs as copying the data out of the log simply becomes a few cache line accesses to remote memory at worst. Large objects – megabytes or more – will shift the overhead to the underlying cross-chip network fabric, and the available memory bandwidth to move the objects out. Thus, depending on the specific workload patterns and composition of the objects, the load on the index may vary.

As part of this thesis, we aim to extract insights into understanding what design principles are important for supporting low-latency queries into Nibble. Currently, we leverage a reader-writer lock on a global hashtable, but aim to experiment additionally with more scalable hash tables, such as the use of Cuckoo chaining [82].

Compaction. The main highlight of using an index affords us the ability to relocate objects at-will. One outcome is greater control over fragmentation in the system. Without the ability to relocate objects, there will ultimately be an allocation pattern that causes extreme fragmentation [124](Sec. 1.2). This challenge plagues "non-copying" allocators used in modern software today – implementations of `malloc` and `free` [25, 47, 76]. With this wonderful property are important and potentially unavoidable costs: compaction (i) must synchronize with the index, (ii) may require

significant resources when relocating objects, such as cpu time (blowing out the CPU caches), or memory bandwidth which may interfere with applications (when the system is near capacity [111]), and (iii) can actively block client operations from executing when they access data in a segment that is being compacted. For each, we summarize potential areas to explore to make compaction practical on large-scale systems:

1. Synchronization with the index. One solution is to use lock striping [64](Sec. 13.2.2), where thousands of locks are used, each associated with a specific subset of the keys in the index. Locks would be identified by the hash of a key. Another option is to use non-blocking (concurrent) data structures [82, 38]. We aim to explore alternative designs.

2. Memory bandwidth interference. Given disaggregated bandwidth on the system, we may selectively enable compaction to those NUMA domains where application utilization is low, or create such scenarios by migrating threads or (for `put` operations) redirect to different segments. More radical approaches may attempt to reducing the work for compaction more upstream, by alternating between log and heap allocation semantics [91]. We aim to explore such optimizations near full utilization of the system.

3. CPU utilization. We have identified a few options: given the large number of CPU cores available, we can simply pin certain compaction threads to specific cores. An alternative is to offload movement requests to a DMA engine, located on each CPU core on modern processors [85, 5]. More below.

4. Blocking client requests. It is vital to ensure client operations do not experience delays; compaction designs can be made simple by *locking* segments (e.g. with read-write locks), but this forces trivial `get` operations to acquire a lock. As segments in Nibble are immutable after being rolled, compaction can *copy* live objects out to new segments, and *defer* reclamation of old segments when any in-flight operations that may have references to the segment have completed. We discuss how to accomplish this below using a novel epoch-based design.

Obviously, mileage of all of the aforementioned techniques will vary, depending on workload characteristics. We plan to explore these as part of the thesis.

There are additional opportunities to leverage compaction to perform "informed" object relocation to address higher-level system management goals, such as rebalanacing load among sockets, replicating objects identified by the application as always immutable (i.e. cannot be overwritten once created), etc.

Reclamation. Enabling concurrency between read operations and segment compaction is supported by deferred reclamation of the blocks backing the segment: compaction logic must synchronize with concurrently executing readers, so that blocks are not reused before an existing thread is finished referencing it. Memory reclamation is not a new topic [95] but is often used in systems today.

One strategy is to use *epochs* [50](Ch. 5.2.3), which record a value representing a logical progression of time. One means of implementing epochs is to use a global epoch counter, and to increment this each time a segment has been compacted, then place a reference to this segment and the current epoch in a reclamation list [110, 24]. Each incoming request would read the current epoch value, then add itself to an "in-progress" request queue, which the compaction logic would check to verify if it was safe to release a segment. On systems with hundreds of cores, supporting potentially tens to hundreds of millions of requests per second, the process of allocating a structure and appending to a list can quickly become a scalability bottleneck. Additionally, cleaning terabytes of memory over disaggregated bandwidth channels with multiple threads would cause the simple epoch (in a cache line) to bounce around the system – another scalability bottleneck.

Global counters for the epoch can be implemented also with combining trees [64](Ch. 12.3), however these suffer from higher per-increment latencies, and are not adaptive for applications or phases with lower parallelism.

We take two approaches to implement memory reclamation scalably in Nibble:

1. Epoch: processors have a 64-bit time-stamp counter which increments monotonically starting from system boot, and is present and synchronizable on all CPU cores [122];

2. Reclamation: instead of tracking each *operation* that is executed, which, on large-scale shared-memory systems may be more short-lived than on network-based systems, we track the latest epoch *per thread* using an epoch table. Each thread that invokes a method in Nibble's API will pin the current TSC to its private epoch, and upon exit it will mark a quiescent value. The reclamation thread can scan this table – without locks or use of atomic instructions – for the minimum value for determining which waiting segments may be released.

In this thesis, we aim to evaluate the different reclamation strategies, and their effects on the performance of client operations.

Appending. In write-intensive workloads, append operations and index updates dominate the performance. With scalable concurrent index structures (mentioned above), write latencies bottleneck at the log head. This is shown in Figure 7.

We envision using two strategies in Nibble:

1. The log will use *multiple log heads* to reduce the chance that threads will contend for the same segments for appending (already implemented);

2. In cases where threads contend for the same segment, we will reduce the critical section to a simple *atomic increment of the offset* so that multiple writers can then, in parallel, store their objects into the segment concurrently.

3. In some scenarios it may make sense to allocate a "private" segment for a writer thread that avoids all contention during appends. Once full, the segment would need to be merged with the log (or buffered with other segments and merged as a batch).

Allocation. Log head rolling at high rates will cause the log to frequently invoke allocation requests from the block allocator. Use of per-log-head block caches of varying sizes may help stagger allocation requests (however, the effectiveness also depends on the workload patterns). As part of this thesis, we aim to investigate appropriate strategies for managing the allocators.

3.3 Resource Management

Semantic operators. With byte-addressable heap spaces used for more than just application structures and metadata, e.g., files [43], objects managed by Nibble may be of variable sizes. `get` operations on such data may be pulling out very large chunks of memory; with large systems and random access patterns, the chance that the operation touches local data is low – ca. $\frac{1}{N}$ for N NUMA domains. Without an understanding of how such data is used, bandwidth may be used inefficiently. For example, a thread which merely wants to scan such an object for some value will experience higher latencies as the object is moved across the system to the thread's CPU caches.

```
get_with(key: String, fn: Closure(arg..)) → Buffer
```

Figure 12: Closure-enhanced operators to support resource management policies.

We envision the use of *closures* to associate with operations in Nibble, enabling applications to download part of their functionality into the Nibble service. With an understanding of where an object resides and its size, where a thread is executing, and how the object is used, Nibble may make

more informed decisions, such as executing a provided closure on a CPU closest to the object, and move only the result back to the application. If the result is smaller, cross-socket bandwidth may be spared, improving end-to-end application performance.

Inserting objects into the data store may be supported by alternate semantics when objects are "big". For example, instead of copying objects from client buffers into a position in the log, Nibble may instead allocate a position in the log for the application, and then provide direct access to this location for the client to write the object into. Such semantics support situations where the originating object does not yet exist, e.g., it is arriving from the network, or must be constructed from other objects or memory within the client.

Append redirection. With multiple log heads to choose from on each append operation, we may use different policies. For example, regardless of where the operating system scheduler places the thread, `put_object` may always only append to that part of the log which is local to its socket. This way cross-socket bandwidth is spared.

Such *local-always* policies for append may, in some cases, not achieve the desired results, if we ignore where the originating object is located. Instead of appending to the log local to the thread, we may choose to have the append execute on the socket local to the originating buffer, eliminating cross-socket traffic. Mechanisms to achieve this include dynamically spawning remote threads, using existing thread pools, or hijacking CPU interrupts for lower latency.

If an application wants to interleave its objects throughout the entire platform – be able to instead attain higher aggregate system throughput – append operations may, for example, rotate among all sockets in the platform.

Exploiting compaction. If we identify certain objects as *hot* (frequently accessed) we may inform the compaction logic to relocate such objects to different sockets entirely to achieve more balanced load in memory bandwidth, or to replicate the object(s) across sockets to match application demand.

As part of this thesis, we aim to explore various tradeoffs as described above, in conjunction with real workloads.

CHAPTER IV

DESIGNING EFFICIENT IN-MEMORY SYSTEMS

The second contribution of this thesis addresses the challenges of low-latency data access, and ineffective and performance-limiting data management on large platforms due to the *lack of effective semantics in the operating system interfaces* for managing both virtual and physical memory resources. Towards this end, (1) we propose an operating system design that exposes the capability to flexibly compose virtual address spaces – *SpaceJMP* – with more explicit abstractions for allocating, translating, and sharing complex data; (2) we apply the use of our scalable memory object store, Nibble, to leverage these abstractions in its ability to improve the performance of applications using and sharing large data sets, enabling applications to directly switch address spaces to access Nibble's internal state without marshaling data. The idea is that, with better understanding of how data is used, we can do better managing the memory backing the data to improve application performance.

We briefly review how the lack of, or non-specificity of, semantics for managing virtual and physical memory can lead to undesirable memory management decisions that negatively impact application performance (prior work has emphasized that OS interfaces are not as efficient as required for data center applications and services [107]). In the next sections we introduce the design of our operating system's memory subsystem to support composable virtual address spaces, and then how our data-serving memory object store Nibble can benefit from such semantics to improve application performance.

4.1 Challenges

Interface Spaghetti. On simple machines, memory allocation was simple: one CPU socket meant no need for NUMA awareness, memory locality, or page fault allocation strategies, etc. The introduction of complex systems required kernel developers to add "extensions" to the existing memory management API (basically, `mmap`) to handle specific management cases. In Linux, for example:

- `libnuma` was created to assist with memory management on a multi-socket system:
  - `numa_set_preferred()` configures a task-global policy for all threads of a process to allocate from a specific socket (similarly for interleaving pages);

  - `migrate_pages()` migrates all physical pages of a process and `numa_move_pages()` migrates a given set of physical pages, but these primitives can be ineffective [85];
  - platform topology is given by a seemingly arbitrary metric called "numa distance", reported as multiples of the scalar value 10;[1]

- out-of-band knowledge about kernel-level allocation policies must be known, such as "first-touch": pages are allocated wherever the thread happens to be at the time;
- copy-on-write is not exposed outside the kernel as an explicit management technique for user-level to leverage (it remains an optimization within the kernel);
- `madvise()` flags configure the behavior of the page-fault handler or exacts behavior on-demand over a specific region of virtual memory, e.g. `MADV_REMOVE` will free a memory range, however "only shmfs/tmpfs supports this" (the flag is bound to the underlying filesystem on which the region is mapped);
- `mmap()` has flags such as `MAP_POPULATE` which allocates pages based on the above configuration(s);
- the unit of grouping physical pages for sharing is a *file* (if not anonymous), but files originate from a universe where only disks contained our storage, and
- specifying sub-file granularity permissions for a group of processes requires mapping to a complex composition of multiple files and permissions of ACLs, users, groups, etc. [45];
- allocating large regions of "anonymous" memory can be performed quickly if user code spawns threads to force page faults (only if the kernel implementation supports it [33, 32]), because otherwise
- `MAP_POPULATE` executes with a single kernel thread and serializes page allocation for a memory region, and, if memory is based on a file, parallel page-faulting can, too, be bottlenecked by the file system implementation;
- operating systems have their own policies for which virtual page sizes are applied to a given memory mapping, called "transparent huge pages" [102, 7], and since there are many such "huge" pages (2MiB and 1GiB on x86_64) we still don't know which is chosen (furthermore, there are many conditions that can cause THP to fail to use a large page);

None of the above have any abstraction that, at its core, represents merely a set of physical pages; we must either use the file system and its security policies, or maintain meta data in our own state to remember the groupings of virtual memory pages and their policies we have configured for

[1] `man(3) numa` in the Linux man pages.

them. Debugging and optimizating performance require first understanding and navigating these complex semantics. A talk by Tim Harris of Oracle at NICTA (Australia) discussed at length the problems such complexity poses in empirical measurements reported in papers [63].

Disorganized Multi-tenancy. Systems with massive memory pools are expected to be multi-tenant, due to the better memory access characteristics compared to networked machines: applications, data storage, and other data-intensive analytics will be co-located on the same systems, some as tightly coupled workflows. Understanding the memory access characteristics – size, producers/consumers, mutability, layout, etc. – enables more effective memory placement, translation, and movement on large systems. Multi-tenant systems, however, can make it challenging to understand and control the allocation patterns experienced:

Using an example from above, some operating systems allocate backing memory for large data sets using specific policies, such as "first-touch". As the operating system scheduler may relocate threads, to satisfy load-balancing requirements or otherwise, data may be allocated non-deterministically across a multitude of sockets. If such data must be accessed with low latency, spreading it across sockets will hurt performance; this problem is exacerbated with more sockets and more complex scheduler heuristics, as well as the behaviors of other executing applications. Applications must resort to pinning threads, but these decisions may conflict if made independently. If such data must support high bandwidth access from a multi-threaded application, it is better to spread such data evenly across sockets to leverage the aggregate bandwidth. There is no means to express such scenarios to the kernel, aside from some composition of the above interfaces.

Exacting more control in these situations requires one to interact with the CPU scheduler, or to change the behavior of the page fault handling code. Instead, we should be using memory-based abstractions and the kernel should make this become a first-class citizen for user-level applications to reason with.

Sharing is Slow or Complex. Data-serving systems today, such as Redis, Memcached, HDFS, and others, all implement a socket interface (or RDMA for higher-performance networks). Directly adapting this for use in large shared memory systems presents unnecessary overheads of data marshaling, data copying, and implementations of additional layers of abstraction. Removing pieces of the network stack from sockets – UNIX domain sockets – does not solve the problem: applications want direct, unadulterated access to their information. Sockets furthermore require there to be active threads listening on both ends, which can complicate optimizations for low-latency, as the

CPU scheduler must be taken into account.

Directly sharing a set of physical pages between processes is, technically, the method with the least overhead, as it removes the kernel from all mediation. However, there are limitations to the amount of control applications are given over the layout of their address spaces, which make implementing such methods more difficult. For example, address space layout randomization (ASLR) is enabled on production systems and creates fragmentation within the virtual address space. Data structures are often composed of pointers, and are thus bound to specific locations in virtual memory; asking the kernel to make a specific range of addresses available is not always feasible: Linux will overwrite existing mappings without returning errors if user code requests a mapping be created at a specific address. Shared memory interfaces also require both participants to agree in advance to some layout of the shared memory area – an invasive agreement.

Sharing semantics of large regions among multiple participants must be still be expressed today using the file system semantics; if there are specific regions that have different mapping permissions, you must create those pages in separate files with appropriate permissions configured.

> *File system and network abstractions should not belong to the foundation for building scalable concurrent in-memory applications* [96].

One could continue to make the argument that operating system abstractions are broken altogether [46]. The argument in this thesis, however, is that operating system interfaces are mainly out-of-date and ill-matched, so much so that applications are no longer directly using them, but instead are relying on higher-level interfaces with more meaningful semantics [22].

Thus, operating systems should give explicit control of memory resources to applications and runtimes, enforcing – ideally – no more than protection. We next present the foundational operating system abstractions Nibble uses to enable applications unfettered access to the data stored within its logs.

4.2 Composable Virtual Address Spaces

SpaceJMP [45] is a set of operating system abstractions on top of virtual and physical memory management, enabling processes to directly create, compose, and share multiple virtual address spaces, with a key mechanism enabling a single process to explicitly *switch* its address space arbitrarily at runtime. This is achieved by elevating the virtual address space to become a first-class citizen in the operating system.

Memory can be allocated via *named segments*, which like in Nibble, are a set of physical memory

pages, and can be made accessible by linking them with a VAS created by a process. Processes attach to a VAS, creating a local copy of the page table, and map in their private memory regions (code, stacks, and libraries) before directly switching between them. Address spaces and segments can exist independent of any processes, much like a file in a file system. The difference, however, is that segments only refer to byte-addressable memory.

Segments can be allocated with various properties, such as the page size, distribution of pages in the system (near or far socket), have much richer sharing semantics, and are not associated with the file system. Segments remain allocated in the system, not associated with the lifetime of the process that created it.

Allowing applications to compose address spaces enables the sharing of address spaces and complex data to be independent of the layout of the memory in the address space. The benefit of this is simplified program design, while retaining the performance of using traditional shared memory interfaces.

4.3 Supporting Low-Latency Analytics

This thesis will explore the tradeoffs for scalable object management provided the ability to explicitly compose address spaces. Directions include:

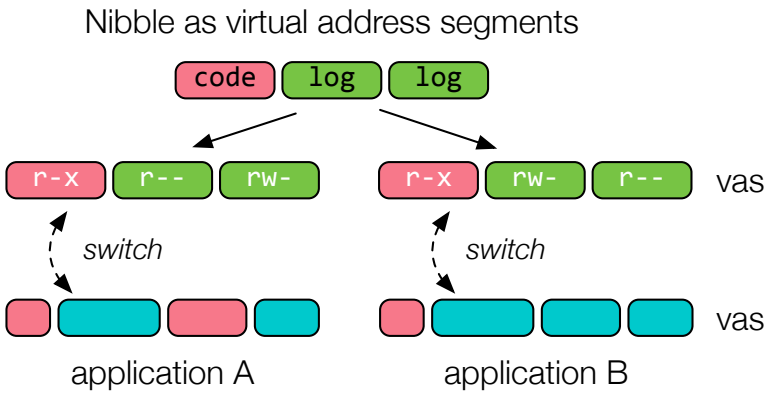


Figure 13: Using segments to share data in multi-tenant systems. The store can be divided into segments, with custom access control policies applied. We can avoid use of the file system (and associated scalability challenges [96]) and protect data at a finer granularity.

Sharing. On multi-tenant systems, applications will store their data together in a central store like Nibble. These applications may not trust each other. Thus specifying the granularity at which data is shared among clients will be made more natural using interfaces within SpaceJMP, as we avoid the filesystem semantics (users, groups, etc.) and the associated costs. Figure 13 illustrates

this idea conceptually.

Avoiding all object copying. The interface in Figure 11 has the semantics that data is always copied in and out of user buffers before applications can access them. With SpaceJMP, clients could instead get direct – safe – access to the internal log structure maintained by Nibble to avoid all copies. Without the complexities of setting up explicit shared memory regions with defined layouts, applications may instead *switch* into the address space maintained by Nibble to directly interact with objects. Safety can be achieved by configuring access through protection-constrained page tables, or via new hardware extensions such as Intel's *Memory Protection Keys* [10].

Workflow Support. We have a multi-threaded image analysis workflow whose performance metric is througput. All threads use a central data store to hand off images, analytics output, temporary data, etc. Traditional workflow configurations may use HDFS, or a generic KVS like Redis, but sockets are slow, and data movement costs may bottleneck the system. Using Nibble and SpaceJMP may allow use to use information about the types of data this application uses to make decisions at runtime: do we copy the data out of the log, or switch into the address space? where to we allocate data most effectively, given an understanding of the producers and consumers? etc.

CHAPTER V

STATUS AND TIMELINE

Completed work

1. The OS design with multiple virtual address spaces is designed and implemented on DragonFly BSD as part of prior work at Hewlett Packard Labs [45]
2. Nibble is implemented

Plan of Action

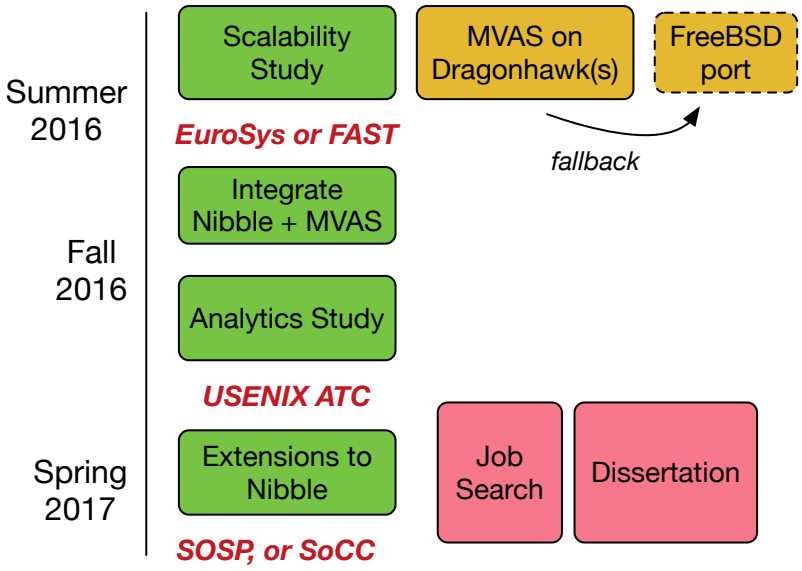
1. Scalability study of Nibble on Dragonhawk – investigate index, use of DMA engine, and Redis – 4-8 weeks
2. Port SpaceJMP to latest version of DF BSD (due to code rot) – 2-3 days
3. Get SpaceJMP/DF to boot on a Dragonhawk system – 2-7 days
4. (if DF port won't boot, port SpaceJMP to FreeBSD, as FreeBSD is known to boot on 3TiB+ systems) – 1-3 weeks + 1-3 weeks' time for testing/stability
5. port Nibble to use SpaceJMP/DF or SpaceJMP/FreeBSD – 2-6 weeks
6. integrate image analytics application to use SpaceJMP and Nibble – 1-2 weeks
7. evaluation study of analytics app on dragonhawk – 2-4 weeks

Paper Targets

1. *Stop Wasting Your Memory* (aka Nibble)
   - FAST (Sept. 27), or EuroSys (Oct. 21)
2. *System Support for Main Memory Analytics*:
   - USENIX ATC'17 (ca. Jan.)
3. follow-up papers include scalable heap allocators, data movement primitives, etc.
   - SoCC'17, SOSP'17

Timeline *(please turn to next page)*

Timeline


Summer 2016
Fall 2016
Spring 2017
Scalability Study
MVAS on Dragonhawk(s)
FreeBSD port
fallback
EuroSys or FAST
Integrate Nibble + MVAS
Analytics Study
USENIX ATC
Extensions to Nibble
Job Search
Dissertation
SOSP, or SoCC

REFERENCES

[1] "dlmalloc." http://g.oswego.edu/dl/html/malloc.html, 2000.

[2] "The big kernel lock lives on." http://lwn.net/Articles/86859/, May 2004.

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

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

[5] "Intel quickdata technology software guide for linux." http://www.intel.com/content/dam/doc/white-paper/quickdata-technology-software-guide-for-linux-paper.pdf, May 2008.

[6] "Microsoft photosynth." photosynth.net/, Aug. 2008.

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

[8] "Facebook graph search." www.facebook.com/about/graphsearch, Oct. 2014.

[9] "Cache allocation technology improves real-time performance." http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html, 2015.

[10] "Memory protection keys." http://lwn.net/Articles/643797/, May 2015.

[11] "U.s. desktop search engine rankings." http://www.comscore.com/Insights/Rankings/comScore-Releases-January-2015-US-Desktop-Search-Engine-Rankings, Jan. 2015.

[12] "3D XPoint Technology." https://www.micron.com/about/emerging-technologies/3d-xpoint-technology, May 2016.

[13] "Domo – data never sleeps 3.0." https://www.domo.com/learn/data-never-sleeps-3-0, May 2016.

[14] "Gnu malloc source code." http://www.gnu.org/software/libc/, May 2016.

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

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

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

[18] Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S., "Millwheel: Fault-tolerant stream processing at internet scale," in *Very Large Data Bases*, pp. 734–746, 2013.

[19] 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.

[20] Appavoo, J., *Clustered Objects*. PhD thesis, University of Toronto, 2005.

[21] Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A., "Scale-up vs scale-out for hadoop: Time to rethink?," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 20:1–20:13, ACM, 2013.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] Chou, C., Jaleel, A., and Qureshi, M. K., "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2014.

[32] 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.

[33] 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.

[34] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D., "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 251–264, USENIX Association, 2012.

[35] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D., "Spanner: Google's globally-distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, (Hollywood, CA), pp. 261–264, USENIX Association, Oct. 2012.

[36] Cutler, C. and Morris, R., "Reducing pause times with clustered collection," in *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, (New York, NY, USA), pp. 131–142, ACM, 2015.

[37] 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.

[38] 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.

[39] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y., "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25* (Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., eds.), pp. 1223–1231, Curran Associates, Inc., 2012.

[40] Dean, J. and Ghemawat, S., "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[41] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L., "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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

[49] Flickr, "Park or bird." `http://parkorbird.flickr.com/`, May 2016.

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

[51] 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.

[52] 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.

[53] Gidra, L., Thomas, G., Sopena, J., and Shapiro, M., "A study of the scalability of stop-the-world garbage collectors on multicores," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 229–240, ACM, 2013.

[54] Gidra, L., Thomas, G., Sopena, J., Shapiro, M., and Nguyen, N., "Numagic: A garbage collector for big data on big numa machines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 661–673, ACM, 2015.

[55] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C., "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012.

[56] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I., "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 599–613, USENIX Association, 2014.

[57] Google, "Google trends." `https://www.google.com/trends/`, May 2016.

[58] 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.

[59] 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.

[60] Gupta, A. and Shute, J., "High-availability at massive scale: Building googles data infrastructure for ads," in *Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2015.

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

[62] Harris, T. L., "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, (London, UK, UK), pp. 300–314, Springer-Verlag, 2001.

[63] Harris, T. L., "Do not believe everything you read in the papers." `https://timharris.uk/misc/2016-nicta.pdf`, Feb. 2016.

[64] Herlihy, M. and Shavit, N., *The Art of Multiprocessor Programming*. Morgan Kaufmann, 1 ed., Mar. 2008.

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

[66] Hu, L., Schwan, K., Amur, H., and Chen, X., "Elf: Efficient lightweight fast stream processing at scale," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 25–36, USENIX Association, 2014.

[67] IBM, "Ibm infosphere." `http://www.ibm.com/software/data/infosphere/`, May 2015.

[68] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T., "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, vol. abs/1408.5093, 2014.

[69] Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., ur Rahman, M. W., Islam, N. S., Ouyang, X., Wang, H., Sur, S., and Panda, D. K., "Memcached design on high performance rdma capable interconnects," in *2011 International Conference on Parallel Processing*, pp. 743–752, Sept 2011.

[70] 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.

[71] 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.

[72] 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.

[73] Klein, F., Beineke, K., and Schottner, M., "Memory management for billions of small objects in a distributed in-memory storage," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pp. 113–122, Sept 2014.

[74] 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.

[75] Kung, H. T. and Lehman, P. L., "Concurrent manipulation of binary search trees," *ACM Trans. Database Syst.*, vol. 5, pp. 354–382, Sept. 1980.

[76] 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.

[77] Lee, M., Kang, D. H., Kim, J., and Eom, Y. I., "M-clock: Migration-optimized page replacement algorithm for hybrid dram and pcm memory architecture," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, (New York, NY, USA), pp. 2001–2006, ACM, 2015.

[78] 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.

[79] Leskovec, J. and Krevl, A., "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, June 2014.

[80] Li, B., Diao, Y., and Shenoy, P., "Supporting scalable analytics with latency constraints," *Proc. VLDB Endow.*, vol. 8, pp. 1166–1177, July 2015.

[81] Li, H., Kadav, A., Kruus, E., and Ungureanu, C., "Malt: Distributed data-parallelism for existing ml applications," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 3:1–3:16, ACM, 2015.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] Liu, V., Halperin, D., Krishnamurthy, A., and Anderson, T., "F10: A fault-tolerant engineered network," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 399–412, USENIX, 2013.

[87] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C., "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 450–462, ACM, 2015.

[88] 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.

[89] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. W., "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] McSherry, F., Isard, M., and Murray, D. G., "Scalability! but at what cost?," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, (Kartause Ittingen, Switzerland), USENIX Association, May 2015.

[95] 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.

[96] 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.

[97] Minhas, U. F., Liu, R., Aboulnaga, A., Salem, K., Ng, J., and Robertson, S., "Elastic scale-out for partition-based database systems," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pp. 281–288, April 2012.

[98] 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.

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

[100] Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., and Kumar, S., "f4: Facebook's warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 383–398, USENIX Association, Oct. 2014.

[101] Nanavati, M., Spear, M., Taylor, N., Rajagopalan, S., Meyer, D. T., Aiello, W., and Warfield, A., "Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 141–154, ACM, 2013.

[102] 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 implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, (New York, NY, USA), pp. 89–104, ACM, 2002.

[103] 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.

[104] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T., "Arrakis: The operating system is the control plane," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2014.

[105] Qureshi, M. K., Srinivasan, V., and Rivers, J. A., "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 24–33, ACM, 2009.

[106] Ramos, S. and Hoefler, T., "Cache line aware optimizations for ccnuma systems," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, (New York, NY, USA), pp. 85–88, ACM, 2015.

[107] Rhoden, B., Klues, K., Zhu, D., and Brewer, E., "Improving per-node efficiency in the datacenter with new os abstractions," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 25:1–25:8, ACM, 2011.

[108] 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.

[109] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 472–488, ACM, 2013.

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

[111] 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.

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

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

[114] Shun, J. and Blelloch, G. E., "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 135–146, ACM, 2013.

[115] Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littleeld, K., Menestrina, D., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., and Apte, H., "F1: A distributed sql database that scales," in *VLDB*, 2013.

[116] Simone, S. D., "Go 1.6 will make its garbage collector faster." `https://www.infoq.com/news/2015/09/go-16-garbage-collection`, 2015.

[117] Solworth, J. A., "Epochs," *ACM Trans. Program. Lang. Syst.*, vol. 14, pp. 28–53, Jan. 1992.

[118] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H., "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 17–32, Feb. 2003.

[119] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D., "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] Yang, F., Tschetter, E., Léauté, X., Ray, N., Merlino, G., and Ganguli, D., "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 157–168, ACM, 2014.

[127] Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M., "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 249–265, USENIX Association, Oct. 2014.

[128] 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.

[129] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M., "In-memory big data management and processing: A survey," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 27, pp. 1920–1948, July 2015.

Scalable Main-Memory Object Management

Alexander Merritt

39 Pages

Directed by Professor Karsten Schwan

This is the abstract that must be turned in as hard copy to the thesis office to meet the UMI requirements. It should *not* be included when submitting your ETD. Comment out the abstract environment before submitting. It is recommended that you simply copy and paste the text you put in the summary environment into this environment. The title, your name, the page count, and your advisor's name will all be generated automatically.