Concurrent Log-Structured Memory for Many-Core Key-Value Stores

Alexander Merritt Ada Gavrilovska
Georgia Institute of Technology
mail@alexmerritt.us, ada@cc.gatech.edu

Yuan Chen Dejan Milojicic
Hewlett-Packard Labs
{yuan.chen, dejan.milojicic}@hpe.com

ABSTRACT

Key-value stores are an important tool in managing and accessing large in-memory data sets. As many applications benefit from having as much of their working state fit into main memory, an important design of the memory management of modern key-value stores is the use of log-structured approaches, enabling efficient use of the memory capacity, by compacting objects to avoid fragmented states.

However, with the emergence of thousand-core and peta-byte memory platforms (DRAM or future storage-class memories) log-structured designs struggle to scale, preventing parallel applications from exploiting the full capabilities of the hardware: careful coordination is required for background activities (compacting and organizing memory) to remain asynchronous with respect to the use of the interface, and for insertion operations to avoid contending for centralized resources such as the log head and memory pools.

In this work, we present the design of a log-structured key-value store called Nibble that incorporates a *multi-head log* for supporting concurrent writes, a novel *distributed epoch* mechanism for scalable memory reclamation, and an optimistic concurrency index. We implement Nibble in the Rust language in ca. 4000 lines of code, and evaluate it across a variety of data-serving workloads on a 240-core cache-coherent server. Our measurements show Nibble scales linearly in uniform YCSB workloads, matching competitive non-log-structured key-value stores for write- dominated traces at 50 million operations per second on 1 TiB-sized working sets. Our memory analysis shows Nibble is efficient, requiring less than 10% additional capacity, whereas memory use by non-log-structured key-value store designs may be as high as 2x.

PVLDB Reference Format:
Alexander Merritt, Ada Gavrilovska, Yuan Chen, Dejan Milojicic. Concurrent Log-Structured Memory for Many-Core Key-Value Stores. *PVLDB*, 11(4): 458 - 471, 2017.
DOI: 10.1145/3164135.3164142

1. INTRODUCTION

The combined growth in data, greater core counts, and new memory technologies have led to a push in developing new platforms for processing and accessing data with even lower performance tolerances. As industry continues to advance towards "memory-centric" computing [22] where tera- to peta-bytes of data pools are accessible by thousands of concurrent low-power processing cores, we are faced with growing demand in system software for enabling raw access to the performance of such hardware: with byte-addressable memory becoming persistent, memory itself is becoming the primary source for information storage [29].

Many hundred-core "fat memory" machines are available on the market today: SGI's UltraViolet 3000 [3], Dell's PowerEdge R920 [2], and HPE's Superdome X servers [32] with tens of terabytes of main memory (DRAM). Future systems are envisioned to include emerging rack-scale systems, such as Berkeley's Firebox [6], and HPE's The Machine [31], and claim to scale to the order of petabytes of globally accessible persistent memory pools, powered by highly multi-core low-power [1] or custom chips.

To fully benefit from the memory capacity and concurrency offered on such platforms, it is important to design efficient software stack for managing the resident data sets. Key-value stores have proven to be useful in managing large data sets for a variety of applications, for caching [23, 60], using DRAM for storage [59], and to balance persistence and performance of DRAM and disks [45, 26]. Key-value stores have especially benefited from log-structured designs, as the ability to re-balance objects presents opportunity to resist becoming fragmented over time, which is important for long-lived or shared data sets.

We argue that the introduction of these new, larger, more concurrent platforms places increasing burden onto the log-structured designs of in-memory key-value stores, specifically in their ability to sustain high levels of concurrent execution on shared-memory configurations. Existing systems today, such as HPE's Superdome X servers [32], contain many hundreds of cache-coherent cores across 16 sockets, and 48 TiB of DRAM. With hundreds of threads able to interact with a given data set, bottlenecks arise within the overall design of log-structured memory: it is not sufficient to replace individual components with concurrent implementations, such as the index. Designs must incorporate principles of infrequent synchronization, and an understanding of platform bottlenecks that restrict scalability, such as how easily individual sockets may become saturated.

To address these challenges, we introduce a scalable, concurrent log-structured key-value store for in-memory data called *Nibble*. Nibble presents a balance between both performance scalability and efficient memory usage (i.e., less fragmentation), benefiting scalable data-intensive applications on large scale machines with massive memory and hundreds of concurrently executing CPU cores. It introduces a multi-head log allocator design, combined with a concurrent index and linearly scalable memory reclamation scheme to achieve its goal.

Proceedings of the VLDB Endowment, Vol. 11, No. 4

DOI: 10.1145/3164135.3164142

We make the following contributions in the paper:

1. We describe the design of a *multi-head* log-structured allocator that supports scaling of write-intensive workloads, and a strict "global read, local write" policy to take maximum advantage of the underlying platform's performance characteristics;
2. We introduce a novel *distributed epoch-based synchronization mechanism* that enables scalable concurrency between applications and background threads that compact in-memory logs;
3. We provide an evaluation of the scalability and efficiency of our new key-value store *Nibble* using a mix of data-serving and file-system-based workloads on an HPE SuperDome X machine with 12 TiB DRAM and 240 cores.

Our measurements demonstrate Nibble exhibits linear SMP scalability under large write-intensive workloads using YSCB, supported by the use of multiple log heads. Together with distributed epochs, all operations – `GET`, `PUT`, and `DEL`– show linear scalability, with 12x greater throughput than competing systems for highly dynamic workloads at 240 threads.

2. BACKGROUND AND MOTIVATION

Computing platforms are moving towards a "memory-centric"-style of computing [22], whereby enormous pools of byte-addressable memory are shared by thousands of cores. In this work, we consider two important aspects of such platforms: their massive main-memory capacity, and their high concurrency. We begin this section discussing one challenge in managing large quantities of main memory – wasted capacity, a result of fragmentation of the heap space. State-of-the-art solutions that are used to overcome this, however, such as log-structured allocation, are limited in practice by their concurrency. We then illustrate how bottlenecks in modern proposals for log-structured designs limit their overall concurrency in scaling to hundreds of cores.

Table 1: Cost of DRAM as a percent of total, comparing a Dell R920 and an HPE Superdome X as of September 2017.

Platform	DIMM [$1k]	Total [$1k]	% of Total
Dell 4 CPU 3 TiB	$0.65 * 96 = $62.4	$80	78%
HPE 16 CPU 24 TiB	$2.0 * 384 = $768	$1200	64%

2.1 Fragmentation Eats Memory

Our proposal for using log-based management of memory is motivated by our observation that the *fragmentation of memory heaps* arising from frequent use of allocation and release of memory objects, even with overheads costing as little as 20% additional memory, can amount to enormous *absolute quantities* of wasted storage capacity. For example, 4.8 TiB of memory remains inaccessible on machines with 24 TiB of main memory, if allocator overhead due to fragmentation consumes 20% of available memory. Over-provisioning is not an acceptable solution [29]: (1) DRAM is expensive, the majority cost in a platform – see Table 1 for representative large systems; (2) how much additional memory is needed is not easily known in advance. The latter, for example, is seen in representations of graph data structures, and the amount of memory consumption due to heap allocators depends on the specific allocation patterns and the heuristics of the allocator.

The fragmentation problem is further amplified for applications using enormous data sets. It is desirable to allow such data sets

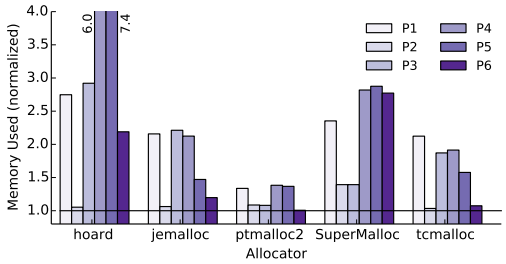


Figure 1: Fragmentation within general-purpose heap allocators. Allocation patterns P1-P6 are outlined in Figure 9b.

to remain longer in main memory, as regenerating or reloading them from alternative storage may take hours or days. Long-lived data sets, however, may fragment more easily over time, seen in servers [40], storage systems [63, 16], and distributed data processing [49], resulting in decreased performance (either due to allocator searching, or garbage collection activities). In-memory systems where allocation patterns are subject to client workloads characteristics, such as in object- or key- value stores (the focus of our work), face similar challenges.

We demonstrate the variability in memory consumption overhead by common heap allocators, shown in Figure 1, with a simple allocator benchmark we implemented. We observed ca. 2-3x greater use of memory, and noticed that allocator patterns did not exhibit the same effect across all implementations (e.g., see P2 and P3). While this is a simple measurement, it illustrates that a simple change in allocator heuristics or design can lead to wildly variable overheads. Much prior work on the performance of heap allocators show them to be very fast and scalable; our work is concerned with the memory consumption overheads incurred.

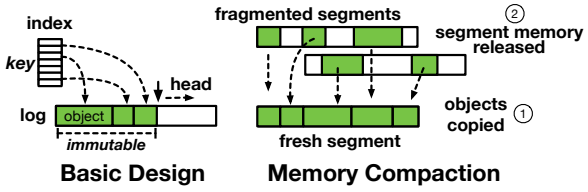


Figure 2: Overview of log-structured allocation.

2.1.1 Log-Structured Allocation

In-memory systems such as object stores have found benefit from log-structured allocation[1], with origins from earlier research on file systems [58]. Environments in which log-structured designs are used present software with higher performance when interfacing with persistent storage by converting random writes into sequential writes. Additionally, due to the longevity of persistent data, use of logs enables storage holes to be reclaimed, thereby also improving performance, e.g. due to faster allocation or scanning of blocks [16], and to reduce garbage collection overhead [49]. It is specifically the latter feature – reclamation of memory – that makes log-structured designs attractive for managing long-lived in-memory data sets.

Figure 2 illustrates a basic design; memory is arranged linearly (logically) with an offset referred to as the *head*; modifications are copied in whole to the head, and the head is incremented. Allocations are fast, as the location of the head already points to a free area, and bumping it is a few instructions. Memory "behind" the head is immutable; background workers typically rearrange this memory to

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

Table 2: Comparison of competing in-memory data management systems.

System	Defrag	[RD	Concurrent UP	Concurrent UP_{HO}	INS]	Obj Size
Redis	×	×	×	×	×	Any
Masstree	×	●	○	·	○	Any
Cuckoo	n/a	●	●	○	●	Fixed
MICA	×	●	●	●	○	Any
LevelDB	●	×	×	×	×	Any
bLSM	●	×	×	×	×	Any
cLSM	●	●	●	·	●	Any
RAMCloud	●	○	×	×	×	Any
OPlog	●	×	●	·	●	Fixed
Bw-Tree *	●	○	○	·	●	Any
Nibble	●	●	●	○	●	Any

(* Projected, source code not available.)

Legend

Defrag	The capability to rearrange objects.
RD, UP, INS	Reads, updates, insertions – all specify the ability to perform operations at high multi-core concurrency.
UP_{HO}	The same as **UP**, but under **h**igh memory **o**ccupancy.
●	Fully capable, given published results or as measured on available hardware, i.e., near-linear scalability.
○	Partially capable, i.e., performance peaks at low/moderate parallelism and does not improve or drops.
×	Incapable, i.e., maximum performance with no/low parallelism.
·	Unknown, either because a system may not understand how to work with limited memory, or no data or source code is available.

reclaim holes, moving available memory (logically) to be in front of the head. Object locations within the log are tracked by an index data structure, associating keys with logical offsets within the log. Relocation of objects cannot be done with heap allocators, as they give out the exact location to clients (i.e., virtual addresses).

2.2 Competing Systems

Table 2 highlights competing systems (a subset of which are evaluated against in this paper) describing their general capabilities to perform in concurrent settings, the main environment targeted for this work. The first category of data stores utilizes fit-based allocation via traditional heap allocators. The second group uses log-structured merge trees, a tree-based hierarchy of sorted buffers; top-level buffers are appended to and kept in memory, then merged in sorted order to lower layers of a hierarchy, where eventually buffers are kept primarily on persistent storage devices. The last set of systems use a form of log-based allocation as the primary principle for their operation, using the general design principle just described.

General key-value stores. General-purpose key-value stores aimed at fast, parallel operations in memory have adopted concurrent data structure designs for their indexes, as all operations must access or modify the index: Masstree [50] uses a trie-like structure formed of B+-trees, a cuckoo-based hash table by Li et al. [44] allows for insertions and lookups to occur optimistically, and MICA [46] leverages a linear-addressing optimistic hash table. While fixed-size objects can be placed in the index itself (such as with the work by Li), variable-sized keys and objects require use of an external allocator, such as used by Masstree and MICA. Doing so subjects the system to potential memory fragmentation, as seen in Figure 1, making them less suitable for managing large dynamically changing and long-lived data sets in main memory. Log-based methods thus allow a system to overcome fragmentation.

Log-structured merge trees. Key-value store interfaces match well with log-based methods: they present an interface to clients where objects have a key and value, copying them in their entirety, and the indirection between keys and their memory location is abstracted by the interface and index. Log-structured merge trees [55] (LSMs) such as LevelDB [26] arrange log buffers hierarchically, appending to buffers in the top layer, and merging downwards during overflows. Appending and merging of buffers allows these designs to avoid fragmentation. The complexity of appending, merging, and reclaiming buffers has limited their scalability: LevelDB and bLSM [62] assume a single-writer design, and thus allow for no concurrency. cLSM [28] uses an in-memory skip list for concurrent lookups, and applies atomics and finer-grained reader-writer locks for managing log buffers. While they demonstrate scalable operation, results are limited to their 8-core platform, and source code is unavailable to evaluate on larger systems. Reference counters on memory buffers in cLSM are used to avoid memory being reclaimed during a read; we suspect this may pose a bottleneck at high parallelism, as it introduces two atomic operations on common data structures in the critical path.

General log-structured allocation. Shown in Figure 2, log-based allocation differs from log-based merge trees in that data is not sorted. This obviates the need to perform sorted merging of buffers, removing this task from the critical path of client threads. Background tasks instead may, independently, scan log buffers and rearrange objects to release memory. Both RAMCloud [59] and our system, Nibble, take this approach, gaining parallelism in compaction and prioritizing client thread latencies. Designs like RAMCloud are intended for execution in a distributed system, with parallelism arising from many machines, and network being the primary bottleneck. Design choices such as coarse-grained locking in their hash table index and a single log head limit scalability severely when the network is no longer the bottleneck (such as execution on extremely large SMP machines). In contrast, Nibble adopts an optimistically concurrent design for the index, and provides many log heads, enabling concurrency in both reads and writes.

Designs intended for write-heavy data structures can improve performance further. OpLog [11], for example, utilizes per-core logs that append operations in temporal order, assigning a CPU clock time stamp to each update; threads may update a common data structure in parallel. Read operations become more complex, however, as all logs must be examined, and (previously deferred) updates applied; this requires locking each per-core log.

The Bw-Tree data structure [48] creates a log per *object* – called a 'delta chain' – allowing client threads to perform incremental updates to an object, appending each delta using non-blocking operations (atomics). Updates are thus very fast and support high parallelism. Ensuring that each delta chain does not grow too large is important for read performance, as the reading thread will need to replay updates in sequential order (to have the latest view of the object). This design is more amenable to larger objects where the meta data for update operations is smaller than the object itself (e.g., with data base tables).

Partitioning. One well-known scalability technique is to partition data structures, such as the logs in OpLog, or data base tables in FOEDUS [36]. Partitioned key-value stores have also been demonstrated on alternative multi-core platforms like the Tilera [9], or to better provision last-level caches [53], albeit for smaller working set sizes. MICA also takes advantage of this, for routing in-bound network requests to the appropriate core for processing. Partitioning is a useful technique, and enables threads to avoid contention on locks,

by accessing a unique resource partition. Nibble applies partitioning to both its logs, index, and background compaction activity.

Epochs and timestamps. Another common technique to improve critical path latency and support parallelism is to leverage time stamps to defer applying updates (e.g., OpLog, and Bw-Tree implicitly), or to remove memory reclamation operations and allow them to be performed in the background (e.g., RAMCloud, Nibble). In the latter, memory that is ready to be reused is stamped with a clock. This memory is released when no in-flight operations have begun prior to a memory chunk's reclamation time stamp, ensuring no client threads could hold a reference into it (e.g., via the index).

Work on the K42 [37] operating system maintains a list of in-flight operations or memberships each client would need to register themselves with. RAMCloud uses such a mechanism, but it limits scalability, becoming a contention point among all threads. Earlier work with the Bw-Tree had instead used a global epoch, incremented with atomics; this also proved detrimental to scalability. Recent work in Deuteronomy [43] adds to this global epoch a thread-local epoch; objects in a garbage list are released when their time stamp is smaller than the minimum of all thread-local epoch values. The dual-epoch design reduces contention on atomic increments, as they occur less frequently. As described in the next sections, this design is similar to Nibble, however we avoid all software management of epochs by utilizing the CPU clock itself, resulting in zero contention, as the processor core increments the clock autonomously.

Summary. Various techniques exist across data storage systems to achieve scalability on extreme-SMP machines. Our goals are to apply these techniques to a memory-fragmentation-resistant design using log-based allocations, supporting both high read and write throughputs on small and large objects. As described in the next section, Nibble combines the use of concurrent optimistic hash tables, parallel partitioned logs, and hardware-based epochs to achieve near linear scalability.

3. DESIGN

Nibble is a scalable log-structured key-value store intended for main-memory object allocation in highly concurrent and shared environments, maintaining low memory overhead. Data sets Nibble does well with are ones in which objects vary in size and lifetime (i.e., objects may be removed or inserted throughout execution). In this section, we review the design of the main components.

The basic design of Nibble consists of three key pieces (Figure 3):

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

Collectively, these three components enable Nibble to scale on extreme-SMP platforms and to support high capacity use for dynamic workloads. We next discuss each in detail.

3.1 Index – Optimistic Concurrency

As every operation requires interacting with the index, we must choose a design that allows enough concurrency between operations as possible. Given our target to run on extreme-SMP systems, we additionally must ensure there is sufficient bandwidth for scaling lookups. The former we address with *optimistic concurrency*, and the latter by *distributing the entire index* across sockets in the system, both techniques were used successfully in other systems (the latter we discuss in the subsequent section).

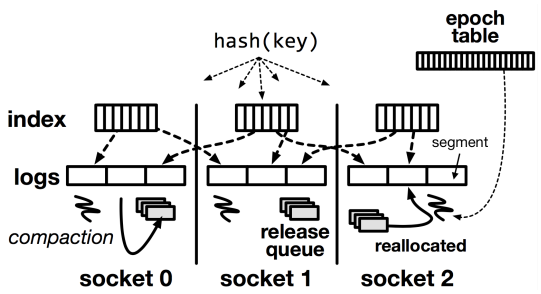


Figure 3: High-level design of Nibble. A global index is partitioned across sockets, pointing to objects anywhere within the system. Logs, segments, and compaction activities are isolated to the cores and memory within each socket.

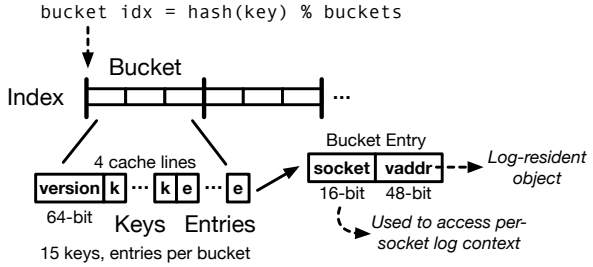


Figure 4: The hash table with optimistic concurrency. Buckets are inlined and entries searched for via linear probing. Nibble allocates 15 entries and one version counter per bucket. Update operations will atomically increment the version to an odd value, blocking new traversals into the bucket, and forcing in-progress reads to restart. Once complete, the version is bumped again, allowing read operations to scan the bucket concurrently.

We wish to avoid use of a sorted index as concurrent methods can add complexities to the overall design, such as for trees [38], skip lists [56], and especially balanced trees [41, 64, 33, 12]. Furthermore, the use of balanced or search data structures present longer latencies for item insertion and retrieval than other data structures, such as a hash table. Thus, our index implements an open-addressing hash table, with no per-bucket chaining, illustrated in Figure 4.

Optimistic concurrency in our hash table uses the following design. Each bucket has a version counter, initialized to zero. Mutators lock the bucket by atomically incrementing the version by one (to become odd), and again incrementing upon completion (to become even). Readers will enter only when the version is even, recording the version to their thread stack. Upon extracting an entry, the version is read again; if both are equal, it means no update modified the bucket contents, otherwise the read restarts. Writers attempt to increment the version by one when it is even; doing so ensures entry by one mutator, and forces in-progress reads to abort and retry. With an odd value, readers and writers will wait, reading the version until it becomes even. This method of optimistic concurrency has been demonstrated in other systems, such as in OLFIT [13], Masstree [50], MICA [46], and OPTIK [30], and is used in Nibble for its simplicity and performance.

The index is able to grow, which is achieved by incrementally

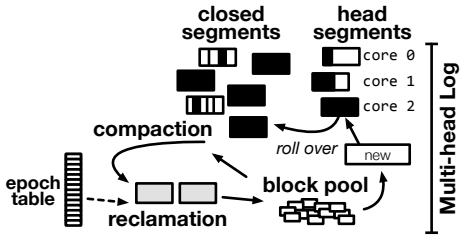


Figure 5: Decomposition of one multi-head log in Nibble.

locking each bucket, then iterating over each key within a bucket to compute its new location, moving it if required. A resize is triggered by an insertion that fails due to insufficient capacity in a bucket. By designing the index to contain 2^i buckets, resizing involves doubling the size of the index; we detail scalable techniques for allocating memory and avoiding relocation of the entire index in the following section. By partitioning the index into multiple (e.g., hundreds to thousands) pieces, we are able to allow growth within one area of the index while not impacting access to objects in the remainder of the index.

Within each bucket entry are two pieces of information: a partition number, enabling access to the log-specific meta data holding that object, and of course the virtual address of the object within the log.

3.2 Multi-Head Log Allocation

As shown earlier in Figure 2, traditional log-structured designs maintain a single head as a reference into unused memory for new updates, bumped incrementally. Data is immutable within the log, and subsequent updates to an object result in appending a new copy, updating the index. The previous object location is invalidated and must be reclaimed. Such a design is sufficient for systems which interact with much slower hardware than DRAM, such as storage or networks, but will not scale to hundreds of cores: (1) contention to mutually lock the head increases, and (2) without considering platform topology, many threads would quickly saturate the bandwidth available on one socket.

Nibble introduces a hierarchical log design to solve both problems: on each CPU socket we instantiate an instance of the log, shown in Figure 5, and within each, numerous head segments. When client threads wish to append data in Nibble, a log instance is selected, then a specific head. Nibble holds to a *"write local read global"* policy where append operations are always performed to local memory nodes, to avoid high memory write costs and contention on the shared interconnect. Local-memory log-based allocation allows Nibble to gracefully handle skewed workloads by restricting contention to exist only within the index; threads will not attempt to overwrite the same location in memory for the object (explored in Section 5.4). Removing contention on shared object memory in this way has been demonstrated in both Bw-Tree and OpLog during updates.

Asynchronous compaction. Designs for maintaining logs may borrow the executing client thread to perform cleanup activities (Bw-Tree), or use a single background thread (LevelDB, cLSM), or provide complete parallelism (RAMCloud). Nibble uses the latter design, as it removes maintenance code from the critical path of client threads, reducing jitter, and removes any dependence on availability of client threads to perform necessary work. The separation further allows assignment of sets of immutable segments to specific threads for monitoring, so that compaction may proceed when appropriate, instead of when a client thread requires more memory.

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

Once segments have rolled over from being a head, they become immutable and are candidates for compaction. Segment selection in Nibble follows prior work [59] where segments are sorted by a cost-benefit ratio of their age and amount of reclaimable bytes. Segments with the largest calculated *benefit* are selected first for compaction.

$$benefit_i = \frac{(1 - util_i) \cdot age_i}{1 + util_i} \text{ where } util_i = \frac{live_i}{length_i} \text{ for segment } i$$

Reclamation queue. Segments wait in a reclamation queue for *quiescent* periods, where no application thread may hold references into the segment. Identifying quiescent periods is required as we allow segments to be read concurrently by both compaction and client threads. Once this is determined (using *scalable epochs*, discussed below in Section 3.3), the segment is deconstructed, and its memory released. If the new segment was constructed with memory from the reserve pool, any released memory obtained is first used to fill the reserve pool, before going back to the general allocator.

SegmentInfo table. In order for compaction to determine the current set of live bytes within a segment, it would normally have to scan the segment, checking each key one-by-one against the index. This becomes increasingly prohibitive with greater numbers of segments, or smaller objects. Instead, we create a metadata table called the *SegmentInfo* table where each entry stores the current sum of live objects within the associated segment, shown in Figure 6. To distribute the work of maintaining the live bytes per segment, each application thread that performs a `PUT` or `DEL` will look up the associated segments and atomically increment or decrement the information directly.

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

3.3 Fast Memory Reclamation

To allow as much concurrency between `PUT`, `GET`, and `DEL` operations and background compaction activities, memory reclamation is made asynchronous, and not performed by client threads. Client threads may update, delete, or read objects concurrently on objects within a log that are also being compacted. The main challenge in knowing when to release old memory segments is in determining when a memory segment will not have any references into it held by threads; this requires maintaining an *active set* of in-flight operations, determining the *earliest* such operation, and *marking waiting segments* as they become ready for reclamation.

Determining whether in-flight operations may possibly hold references into a memory segment requires marking each operation

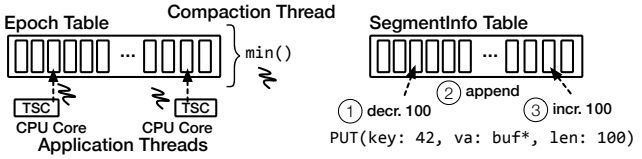


Figure 6: Illustration of synchronization between application threads recording their current epochs, and compaction threads determining minimum epoch for recycling segment memory (left), and updating the live bytes information for segments (right). Each entry in both tables is a separate cache line.

op_i with an epoch e_{op_i} indicating when it began. The set of epochs for all active operations is defined by $\varepsilon = \{e_{op_i}\}$, what we refer to as the *active participant set*. By also marking a segment *seg* that has completed compaction with the current epoch e_{seg} we may conclude that operations which have begun after that segment has finished compaction cannot contain references into it, due to atomic updates to entries in the index. In other words, a segment may be deconstructed when $e_{seg} < min(\varepsilon)$.

As each thread executes one operation at a time in Nibble, we need not record operations themselves, but rather record the state τ of the thread, represented by the current value of the global epoch: $\tau = \text{NIL}$ for executing outside of Nibble, in other words, it cannot have a reference into a log; or, $\tau = i : i \in \mathbb{N}^+$ when executing `PUT`, `GET`, or `DEL` inside Nibble and most likely stored a reference into the log on the stack or in a register. τ is recorded into a set data structure. This is the active participant set.

Our use of thread-local epochs is illustrated in Figure 6. The very first operation performed by client threads invoking Nibble operations is to record the global epoch into its thread-local storage. Upon completion, this storage is reset to NIL. No coordination between threads exists when recording the epoch. Nibble-internal threads which periodically examine segments in the reclamation queue will scan over the active participant set to identify the earliest epoch an operation began at. With this it pops segments from the waiting queue and destructs them.

4. IMPLEMENTATION

In this section, we elaborate on implementation details within Nibble that help explain its scalability claims, and assist in understanding its overall architecture.

4.1 Memory Organization and Terminology

In Nibble, we organize memory into indivisible contiguous chunks of memory called *blocks* (see Figure 7). Their length is a multiple of the size of a virtual memory page, each represented by a structure that encodes its starting virtual address, length, and segment information if it has been allocated to one. *Segments*, informally used in the prior section, are merely container objects, allowing the rest of the system to identify a specific collection of blocks, and apply operations on them in some linear order, such as writing new data. A set of segments forms a *log*, one per memory node or socket, and all segments and blocks of a log are managed by a single allocator. A *head segment* is used for appending new objects, the exact location determined by the *head offset* (and then bumping it). When full, head segments are *rolled over* – closed and replaced with a new segment from free blocks. If the block pool size crosses below some threshold, threads will *compact* existing fragmented segments. To reduce stampede effects on the underlying block allocator, new segments are allocated with a variable size.

Objects provided by application threads are placed next to each other within blocks, prefixed by a 32-bit header, encoding the length

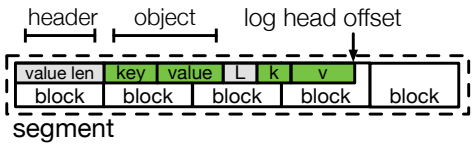


Figure 7: A segment – a container of blocks holding objects.

of the object, and a 64-bit key. To ensure optimal memory usage, we allow objects to be split across block boundaries within a segment; objects may not span segments. Storing the key within the segment allows us to determine whether this instance of the object is the live copy via a lookup in the index.

Appending objects. At high thread counts, the probability of eliminating contention with random selection of a head segment becomes small, as threads must attempt to choose unique segments with each operation. One way to overcome this is to assign each thread a head segment, however, we chose to avoid this option and instead assign a head segment to each CPU core. When an append operation is issued, we identify the specific core a thread is on, and use this information to lookup a given head segment. To ensure such an operation is as fast as possible, we directly leverage the hardware: on all modern x86 processors the `rdtscp` instruction (or `rdpid` on very recent implementations) returns both the socket and core ID within 30 cycles.[2]

Reading objects. When storing new objects in Nibble, we must be able to later identify the location of the object quickly. Appending to the log returns a virtual address of the object header, and this we store together with the ID of the specific log segment into our index (shown in Figure 4). We use the remaining 16 bits of our 64-bit entry to store this ID, as virtual addresses on x86 systems are a maximum of 48 bits. When executing read operations, if the head, key, or value lie within a block, we directly copy them out using the instructions `rep movsb` to minimize the implementation of the copy routine. As blocks within a segment may not be contiguous, for objects that span these boundaries, we must identify the subsequent block to copy from: we map the virtual address to a segment by (1) aligning the address to the next-lowest block boundary and using this to index into our vector of blocks, then (2) extract the segment information to locate the address of the next block to read from.

4.2 Concurrent Index

The index in Nibble is an open-addressing hash table that does not utilize chaining. Thus, on resize, we must expand the table and relocate objects, accomplished by asking the operating system for more physical memory. To avoid encountering allocation scalability bottlenecks within the operating system, we allocate 64 GiB of *virtual* address space per index partition, but only fault in the pages needed to hold the current range of buckets. Upon resizing, we do not invoke `mmap` (as it is serialized within the OS kernel [15]) and instead simply *write to the virtual memory areas immediately following the existing buckets* to cause physical page allocation. We bind the memory regions to sockets at the time the subtable is created, to avoid the "first-touch" policy, which would allocate pages potentially from other sockets. Thus, each time a table partition with 2^i buckets grows, we only allocate at most an additional 2^i buckets, instead of a second table of size 2^{i+1} into which all objects are moved. Modern heap allocators use this virtual memory method to also avoid concurrent updates to underlying page tables [39, 21].

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

Locking alternatives. Bucket versions are managed by atomic increment and decrement for mutators, and simple memory read operations by readers. We have considered alternative lock designs for the bucket such as MCS or ticket locks [52]. As our index is optimistically concurrent, there are a few considerations. First, exclusively locking a bucket is only done by mutators; readers must be free to read in parallel. Second, contention on a bucket may only occur in highly skewed write-heavy access distributions (as bucket selection is determined by a hash). Third, at any point readers must know when a mutator has acquired the bucket lock (regardless if it still holds it when the reader completes). If a bucket becomes locked after a reader begins, and unlocked before the reader completes, unless there was a counter to track this, the reader may conclude the bucket was not mutated from underneath it and return inconsistent data. Updating our bucket version after a mutator completes could be implemented without an atomic, reducing their use to one per update, as only one mutator will be doing so.

Given our goal to ensure index operations remain as fast as possible, we also strive to implement fast locking mechanisms. Scalable locks like MCS require more setup to be performed (connecting a new entry to the lock chain), increasing latency. If we use the version counter within a ticket lock, then two mutators may increment the version from even → odd → even, indicating to readers they may enter the bucket, when in fact multiple mutators are enqueued. Additional state in a bucket also reduces the capacity to hold object entries. Our optimistic method does not provide fairness for ordering. However, given operations are expected to be very fast, starvation is expected to be an unlikely occurrence.

Alternatively, having a lock for each record (object) instead of per bucket would increase storage capacity, and be especially pronounced with small objects. Should objects be 8 bytes, a lock of 8 bytes would double the amount of space needed. If we do not keep this in the hash table, then accessing a secondary structure would mean additional cache line miss delays on each lookup. Due to bucket placement, hot records may pose some unfairness on readers co-located in the same bucket. Keeping buckets small (handful of items) ensures this exposure is limited.

4.3 Supporting Memory Reclamation

Maintaining the participant set ε has been a source of past scalability challenges in other systems. RAMCloud, for example, based off the earlier work in K42 [8], maintains a single non-concurrent linked list of received messages from the network. Adding and removing entries from a single list becomes a bottleneck, especially with very low processing latencies. Early work in Deuteronomy [48, 42] maintained for each epoch a counter whose value indicated a count of active operations started during that epoch; atomic manipulations of the counter for each operation resulted in a bottleneck. Their later work [43] instead maintained a software-incremented global counter together with thread-local counters. At very large scales, however, software-maintained counters eventually become bottlenecks [19].

In Nibble, all threads register themselves in an *Epoch Table* (Figure 6). Each entry is a cache-line-sized unsigned value that records the current state τ for that thread (its epoch, if not NIL). The set of participants is then determined by scanning this table for non-NIL values without locking, resulting only in cache line invalidations. Joining or leaving simply involves modifying an entry in the table.

Instead of a software-maintained global epoch, we leverage the CPU's global time-stamp counter (TSC). Without the use of a software-maintained epoch within a cache line, synchronization points are eliminated and reading of the epoch becomes very fast, avoiding cache line invalidations, as the hardware autonomously increments its value. Reading the TSC is fast – 30 cycles on an Intel Haswell processor. The value zero we implement to represent NIL, indicating a thread is not in the active participant set.

Use of the TSC has been evaluated in other systems for global synchronization, such as OpLog [11] and work by Wang et al. [66], and is shown to not have measurable skew. Should skew exist, we may run a stress benchmark to ascertain which clock is farthest ahead and its time c_{max}, then calculate a distance $\Delta_{c_i} = c_{max} - c_i$ from that to every clock. When comparing epochs, we would first add the respective Δ_{c_i} to each. If clocks progress at different rates, these offsets would need to be periodically recalculated.

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

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

The actual process of compaction is straight-forward. The sum of live bytes is calculated from the set of candidate segments, using the SegmentInfo table, and a new segment is allocated. Should there not be sufficient memory, a reserve pool is used. Each candidate segment is scanned and keys found are verified with the index: lookup failures or mappings which do not point to the current location indicate the object is stale, and the item is thus skipped. Live objects are individually locked via the index while the object is copied – a minimal, required synchronization with client threads. Once completed, the new segment is added to the set of candidates, and the old is pushed into a reclamation queue.

5. EVALUATION

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

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

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

Table 3: Summary of compared systems.

System	Version	Description
Redis [60]	`3.2.8`	In-memory data structure store allocator: jemalloc [21]
RAMCloud [59]	`d802a969`	In-memory log-structured KVS
Masstree [50]	`15edde02`	Concurrent optimistic B^+-tree allocator: Streamflow [61]
MICA [46]	`b8e2b4ca`	Concurrent partitioned KVS allocator: custom segregated-fits [67]

Table 4: Summary of workloads.

Workload	Configuration(s)
Fragmentation benchmark	Benchmark derived from prior work [59]: Given a pair of sizes, allocate objects of first size, randomly delete 90%, allocate objects of second size until allocation fails (or until a specific capacity has been reached).
Dynamic *Postmark [35]*	2 million files, 500-4096 bytes, 32 million transactions, 10k subdirectories
As a trace: *per thread:*	18 million objects, 183 million operations 5.7 GiB working set size
Static data-serving *YCSB [17]*	1 billion 1 KiB objects, 100/95/50% `GET`, 100% `PUT`, uniform and zipfian

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

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

Workloads. Table 4 summarizes the workloads used in our study. We evaluate each system with workloads which stress scalability, variable access patterns, the level of stress on the underlying memory management, and susceptibility to memory fragmentation.

With computing becoming increasingly memory-centric, we aim to evaluate each system with a more dynamic workload at scale. Using Postmark [35] – a well-known file system benchmark – we expose each system to a working set with variable object sizes and a fairly high *churn* in objects – frequent insertion and deletion. In order to capture this behavior for key-value stores, we use TableFS [57] that converts I/O operations into equivalent key-value store operations (it implements a file system with LevelDB [26] within a FUSE [4] mount point).

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

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

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

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

Masstree: We extracted the primary radix tree data structure. We create a single instance thereof, on which client threads directly invoke `PUT`, `GET`, and `DEL` operations.

MICA: The 'EREW' mode is enabled to support shared-memory configuration, allowing application threads to read and update objects located on any core; the segregated fits allocator was configured with 8192 size classes, offset by 256 bytes to accommodate the range of object sizes in our workloads; additional code was added to object deletion to invoke region coalescing[3] and the index hash uses SipHash [7].

Nibble: We enable eight compaction threads per socket, segment and block sizes of 32 MiB and 64 KiB, respectively, and 1-2% of memory is reserved for compaction. While each CPU has only 15 cores, this many compaction threads does not unnecessarily steal the CPU; most of the time, these threads sleep. This is unlike in RAMCloud which devotes specific cores to perform compaction work. When compaction must be executed, perturbance of client threads is unavoidable and irrelevant, as they would otherwise wait for `PUT` to become unblocked.

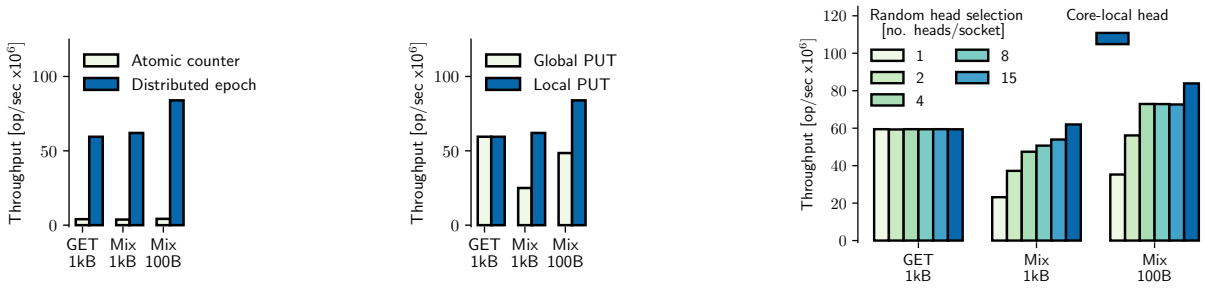
5.1 Breakdown of Components

To better understand the impact of each design feature within Nibble, we experiment with alternative designs, shown in Figure 8.

Epoch. We compare our distributed epoch design with that of a single atomic counter, shown in Figure 8a, and measure a large impact on performance — a 20x gain in throughput. While the epoch is required primarily by the compaction logic, to enable concurrent function with front-facing client operations `GET`, `PUT`, and `DEL`, it has the most impact on the latter. Releasing segments back to the free pools requires keeping track of in-flight operations, and the "time" at which the operation began. Use of an atomic counter requires threads to manually ensure forward progress. The hardware-based TSC increments autonomously, however, and can be read independently from any core in ca. 30 nanoseconds; the onus is on hardware designers to ensure all core TSCs are synchronized.

Furthermore, having threads remain as independent as possible by keeping track of their own operations and time stamps in a private cache line (their slot within the Epoch Table) allows for scalability. Only when compaction threads must interpret a global value will they scan the entire table and record the minimum; no locking

[3]Provided by the authors of MICA from personal correspondence.



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

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

(c) With PUT to local memory, more heads provide the greatest initial performance gains. Selecting the core-specific head, we avoid further contention, gaining an additional 15%.

Figure 8: Performance breakdown of various designs in Nibble: uniform YCSB 240 threads (Mix is a 50:50 ratio of PUT and GET). For (a) and (b) we allocate one log head per core (15 per CPU) and perform core-local selection.

is necessary for consistency, as we rely on the CPU coherence mechanism. The impact of our design is apparent regardless of the mix of operations.

Local vs remote memory. One design strategy to distribute work across all resources is to use the key hash. On small platforms with two sockets, this results in only half of all operations to traverse remote-socket distances. Our large test platform, with 16 sockets, exaggerates this ratio, resulting in 15/16 accesses (94%!) to remote memory. When compared with a strategy of always writing to local memory (Figure 8b), we measure an improvement in throughput of 1.75x and 2.48x, for large and small objects, respectively. Furthermore, use of a hash to determine the log head may result in increased contention between threads during append operations. We next look at how to reduce contention on selecting a log head when implementing PUT.

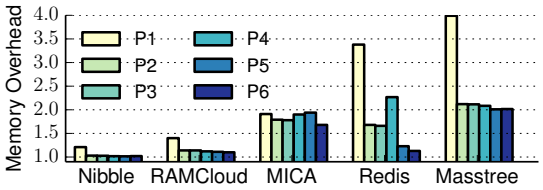
Log head design. In systems like RAMCloud with a single log head, supporting parallel write-heavy workloads becomes strenuous. We evaluate the effect of having multiple log heads, and furthermore the method of choosing a head when implementing PUT (Figure 8c). With threads writing only to local memory, we increase the number of heads available on each socket from 1 to 15, and select the head (i) randomly, or (ii) with 15 heads, based on the core the request is executing on. As discussed in Section 3, we accomplish the latter using the `rdtscp` instruction to determine socket and core IDs.

More heads results in greater, but diminishing, throughput: a maximum gain of ca. 2x over just one log head on each socket. For smaller objects, a performance plateau is reached with only four heads. There is room for further improvement, as random selection, and very high rates of operations, creates some contention on the local log heads. We designed Nibble to select a head based on the core; as long as threads do not migrate during a transaction, or cores are not over-subscribed, this results in zero contention for PUT, gaining an additional 15% in throughput.

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

5.2 Memory Overhead

Ensuring efficient use of available memory alleviates applications from the responsibility of handling failed allocations that result from fragmentation when a dataset is known to fit within the system's physical memory. Additionally, long-running services must remain



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

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

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

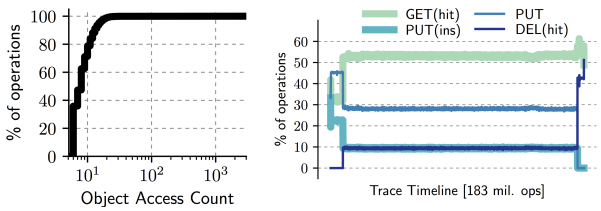
Figure 9: Measured memory overhead (bloat) for each system evaluated. Nibble has the lowest memory pressure – less than 10% additional memory (except for 'P1' with 21%). Only Nibble and RAMCloud utilize log-based allocators.

resistant to changes in both access and allocation patterns that may cause performance loss.

We visit the challenge of ensuring efficient use of available memory by subjecting each system to a series of object allocation and release cycles, modeled after prior work [59] shown in Figure 9. Both RAMCloud and Nibble are log-allocating systems, and are able to accommodate destructive allocation patterns by relocating memory to reclaim holes. Nibble consumes less than 10% additional memory for all but the smallest objects; all systems have greater pressure when managing small objects, due to the fixed-cost of meta-data needed to manage their greater numbers (as is expected).

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

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



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

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

Figure 10: Characteristics of our trace captured from Postmark. Overall, the workload presents a dynamic behavior: most objects are accessed infrequently, yet a consistently high *churn* in objects is present. 20% of all operations will add or remove objects, putting stress on each system's object allocator logic.

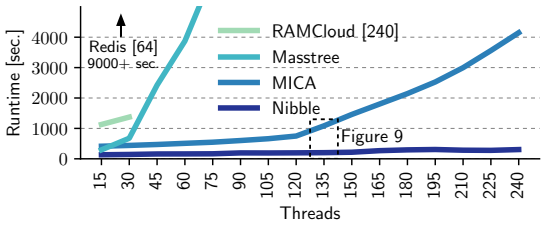


Figure 11: Weak scaling of our Postmark trace, measured in time-to-completion, at a consistent 17% memory utilization: as each thread executes the entire trace on a private working set, we increase available memory as appropriate (except for Masstree). Nibble completes the trace within a few minutes, whereas the other systems require upwards of an hour or more with more threads. RAMCloud was able to run with only 2% utilization until system memory was insufficient.

5.3 Dynamic Workloads – Postmark

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

Trace characteristics. The Postmark benchmark creates a set of files across a number of subdirectories, then performs transactions on them, such as append or deletion (refer to Table 4 for our configuration of Postmark). The trace we captured has 183 million operations and 18 million objects (see Table 4) with, on average, 5.7 GiB of active objects allocated in memory at any given moment. Object access frequency is show in Figure 10a. The majority of objects are accessed a small number of times – 18 individual operations or fewer per object for 80% of the entire trace. Such objects are created, accessed a minimal number of times, then removed. With little reuse of objects, most of the work in this trace is in the *insertion and removal of unique objects* and responding to GET requests.

Figure 10b shows a breakdown of the types of operations. There is a brief setup phase with a high portion of writes and object insertions – PUT and PUT(ins) respectively. Throughout the majority of the trace, a consistent 10% of operations insert new objects, and 10% remove existing objects – DEL(hit). At the end of the trace, nearly 50% of operations are deletions, as Postmark cleans up all objects.

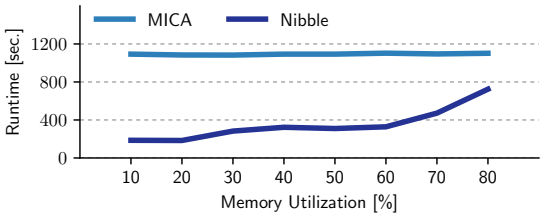


Figure 12: From Figure 11 we measure time-to-completion for Nibble and MICA at 135 threads, and progressively decrease available memory. At 65% utilization, Nibble's runtime doubles due to compaction costs. MICA seems unaffected, however any such effect is masked by contention on its per-core heap allocators.

Scalability. We measured the ability to sustain executing the trace, with a thread executing the trace independently on its own set of objects. As we add more threads, we are adding more work to the system (weak scaling), but as each thread is independent, no thread coordination occurs. Figure 11 shows our results. To ensure a consistent capacity utilization (ca. 17%), we increase the total amount of memory allocated to each system as more threads are added.

Runtime increases very little for Nibble, as each thread can independently (over-)write objects without synchronization with other threads, and having created all objects on the local socket, object accesses are fast. Within MICA, runtime is higher due to multiple factors: the majority of PUT and GET traverse the chip interconnect, as an object's location is determined by its key's hash, and, there is high contention on the per-core heap allocators, which arises when objects are newly allocated or destroyed. The latter is greatly increased with greater numbers of threads.

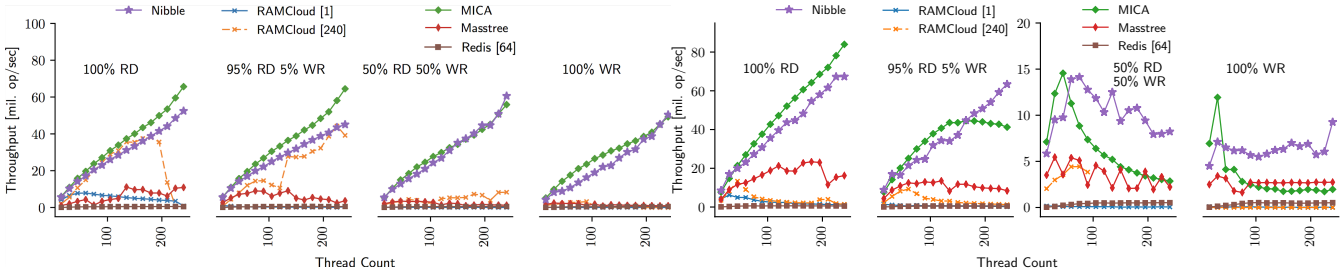
With high rates of insertions and deletions, Masstree unfortunately suffers from frequent rebalancing of its trees, resulting in very high completion times. We were able to execute RAMCloud only by reducing the utilization to 2% overall, with 240 instances. At 45 threads, this would require more memory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

Impact of capacity utilization. Using the two highest-performing systems from the prior experiment – MICA and Nibble – we measure the effect of memory pressure on overall performance. With 135 threads, we decrease available memory such that the utilization increases to 80%, shown in Figure 12. MICA's measured performance is as explained earlier: per-core heap allocators are guarded each by a mutex, creating contention for the 20% of operations which create and destroy objects throughout the trace. Any effects due to memory pressure may be masked by this bottleneck. As Nibble does not experience this bottleneck, it spends more time compacting segments, as expected, more than doubling the execution time at 80% utilization, but Nibble still outperforms MICA under 80% memory utilization.

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

5.4 Data Serving Workloads — YCSB

We implemented an internal YCSB workload driver, measuring the scalability of each system as a function of throughput and



(a) Uniform access

(b) Zipfian access.

Figure 13: YCSB throughput measurements on 1 TiB of data – ca. 2^{30} 1 KiB objects. "95% RD 5% WR" is a workload with `GET` composing 95% of all operations and 5% with `PUT`. No configuration uses `DEL`. Approximately 8 TiB of storage capacity was provided to ensure a low utilization rate of ca. 12%.

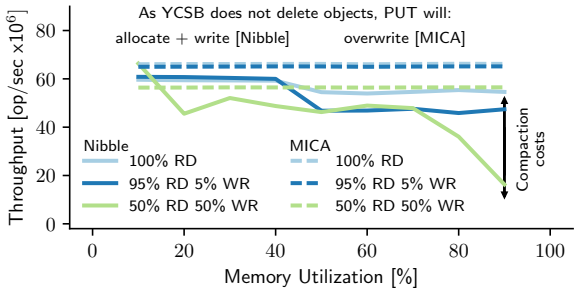


Figure 14: Throughput of MICA and Nibble driven with YCSB (uniform) using all 240 cores, steadily increasing the capacity used.

number of client threads. Figure 13a and Figure 13b show our YCSB measurements of each system. The notation [N] indicates N instances of the associated system used for that configuration. All YCSB experiments do not perform `DEL` and all objects are instantiated before throughput is measured; thus, `PUT`s never require insertion of new objects.

Uniform access. With uniform key distribution (Figure 13a), both Nibble and MICA are able to scale with increasing threads. For mostly-`GET` configurations, MICA demonstrates up to 25% and 43% greater throughput for 100% and 95% `GET`, respectively. With high percentages of `PUT`, both systems show roughly equivalent performance; Nibble executes `PUT` to local socket memory, whereas MICA distributes them based on the key hash. Scalability for both systems is achieved via optimistically concurrent indexes, and the ability to overwrite (and for Nibble, append) independently.

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

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

High percentages of `PUT` create additional bottlenecks for RAMCloud and Masstree. Single-instance RAMCloud is limited to a single log head, which becomes an immediate bottleneck, and multi-instance RAMCloud, nevertheless with 240 log heads, is limited by append operations that frequently chose the same head, as objects are assigned based on their hash.

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

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

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

In contrast, writes in Nibble *are not* directed to a shared subset of memory (the log-structured nature ensures this) and where an append occurs is not determined by the hash, but instead by the per-core log heads. Thus, threads will copy objects to private memory areas in local memory, but contention still exists within the index to reflect the new location created for the objects.

Impact of capacity utilization. Figure 14 examines the impact on overall throughput when the total system capacity is increased to near full. Given that MICA and Nibble are the highest-performing systems in our YCSB study, we limited this analysis to just these two. At high utilizations (≥ 70%), compaction in Nibble becomes the dominant overhead. For systems which do not utilize log-based allocation, such as MICA, no allocation logic is exercised in these experiments (as the YCSB workload does not delete or create new objects). At high capacity utilizations, log-based allocation becomes prohibitive under moderate to substantial write configurations.

▷ **Summary.** Nibble supports scalable execution, within 25%-43% of the best non-log-structured system (MICA) for read workloads, supported by the use of a scalable epoch implementation. Our use of multiple log heads supports write-dominated workloads and

out-performs all compared systems by 2x for skewed workloads at high thread counts, demonstrating our *write-local read-global* policy is advantageous. With static data sets and high memory pressure, use of a non-log-structured system proves better for write-dominated workloads, due to lack of compaction costs.

6. RELATED WORK

Section 2 provides detailed comparison of Nibble against other key-value stores and log-based systems. Here, we summarize additional related work with respect to concurrency and memory management.

Memory reclamation and concurrent data structures. Our system Nibble shares a technique for memory reclamation found in many prior systems using the time stamp counter [66, 11] as an epoch for enabling concurrent access with memory reclamation [59, 54, 24, 8]. Read-Copy-Update [51] is a form of memory reclamation applied to data structures to allow concurrent lockless reads, even in the presence of some writes, by copying portions of a data structure, atomically committing changes. RCU has been successfully applied to a variety of data structures, such as balanced trees in operating systems [14]. New hardware features such as HTM have also been explored in enabling concurrent memory reclamation [5].

Supporting dynamic resizing is challenging with RCU, thus alternative techniques for scaling data structures, such as hash tables [65] have been proposed. As Nibble prioritizes low-latency operations, use of relativistic hash tables would require chained buckets – a linked-list data structure that presents poor cache locality.

Comprehensive surveys and analyses of synchronization techniques on very large systems have given rise to general programming paradigms for designing a wide variety of concurrent data structures, such as ASCY [19, 20]. Scaling Optimistic Concurrency Control (OCC) data structures, such as in Optik [30], provide general design guidelines for applying OCC to a variety of data structures. How to resize OCC-based hash tables, however, is not reviewed in-depth.

Broom [27] is a recent study on "big data" processing systems, which found that high object churn causes language-based garbage collection to introduce large runtime costs. They propose a special region-based allocator (examined also in other work, Reaps [10]) to mitigate this overhead. From the study, it is unclear how one might use regions in multi-tenant systems using a common data store, where Nibble is intended to be used.

NUMA. Much prior work evaluates the impact of NUMA characteristics on applications [25], with proposals to use automated solutions to avoid memory bandwidth bottlenecks [18], or compiler and language support to infer memory access patterns [34]. Further challenges exist in overcoming the transparent nature of managing physical memory; proposals to use hardware acceleration for memory movement [47, 34] have demonstrated success in some situations, compared to CPU-based copying. These efforts may complement applications that use Nibble on large NUMA systems, as Nibble explicitly allocates memory across sockets, exposing interfaces for applications to explicitly place data in specific memories.

7. CONCLUSION

New hardware designs incorporating ultra-large volumes of main memory and parallelism into the hundreds of cores are being developed to address the requirements of scalable applications. The utility of these systems is highly dependent on the effectiveness of their memory management to efficiently utilize the available memory capacity and to provide scalable performance. In this paper, we illustrate that current solutions, include competing log-structured and non-copying-based key-value stores of various designs, fall short specifically when reconciling these two competing goals. In response, we propose Nibble, an in-memory object store that introduces a scalable multi-head log allocator design, combined with a concurrent index and linearly scalable memory reclamation scheme using a distributed epoch, to achieve balance between both performance scalability and efficient memory capacity usage.

Using various data-intensive workloads we show that, by leveraging a combined set of designs for concurrent indexing, memory reclamation, and log management that Nibble scales linearly and matches competitive non-log-structured key-value stores for write-dominated configurations on tera-byte-size data sets. Our memory analysis shows Nibble is efficient, requiring less than 10% additional capacity, whereas memory use by non-log-structured key-value store designs may be as high as 2x.

In the future we plan to extend this work to rack-scale systems with globally accessible fabric-attached memory and address persistence and failure recovery with emerging non-volatile memory.

8. ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their generous time and feedback in greatly improving the manuscript. We would also like to thank multiple colleagues at Hewlett Packard Labs, including both Haris Volos and Yupu Zhang for their suggestions in the use of file system traces, as well as Taesoo Kim and others at the Georgia Institute of Technology for reviewing our work and earlier drafts.

9. AVAILABILITY

We have made the source code for Nibble available at the following URL: `https://github.com/gtkernel/nibble-lsm`.

10. REFERENCES

[1] ARM and Cavium Extend Relationship with ARMv8 Architecture License. `https://www.arm.com/about/newsroom/arm-and-cavium-extend-relationship-with-armv8\-architecture-license.php`, Aug. 2012.

[2] Dell PowerEdge Specification. `http://media.zones.com/images/pdf/Dell_PowerEdge_R920.pdf`, 2014.

[3] SGI UltraViolet 3000. `https://www.sgi.com/products/servers/uv/uv_3000_30.html`, Oct. 2016.

[4] Filesystem in Userspace. `https://github.com/libfuse/libfuse`, Feb. 2017.

[5] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[6] K. Asanovic and D. Patterson. Firebox: A hardware building block for 2020 warehouse-scale computers. In *USENIX FAST*, volume 13, 2014.

[7] J. Aumasson and D. J. Bernstein. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, pages 489–508, 2012.

[8] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 279–291, 2005.

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

[10] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 1–12, New York, NY, USA, 2002. ACM.

[11] S. Boyd-Wickizer, F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, Massachusetts Institute of Technology, 2014.

[12] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

[13] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. *VLDB*, pages 181–190, Sept. 2001.

[14] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. 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, pages 199–210, New York, NY, USA, 2012. ACM.

[15] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[16] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, and M. Farach-Colton. File Systems Fated for Senescence? Nonsense, Says Science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, Santa Clara, CA, 2017. USENIX Association.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[18] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[19] T. David, R. Guerraoui, and V. Trigonakis. 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, pages 33–48, New York, NY, USA, 2013. ACM.

[20] T. David, R. Guerraoui, and V. Trigonakis. 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, pages 631–644, New York, NY, USA, 2015. ACM.

[21] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD, 2006.

[22] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond Processor-centric Operating Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 17–17, Berkeley, CA, USA, 2015. USENIX Association.

[23] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[24] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[25] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM*, 58(12):59–66, Nov. 2015.

[26] S. Ghemawat and J. Dean. LevelDB. `https://github.com/google/leveldb`, 2016.

[27] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[28] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[29] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. In-Memory Performance for Big Data. *PVLDB*, 8(1):37–48, Sept. 2014.

[30] R. Guerraoui and V. Trigonakis. Optimistic Concurrency with OPTIK. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 18:1–18:12, New York, NY, USA, 2016. ACM.

[31] HPE. Hewlett Packard The Machine. `http://www.labs.hpe.com/research/themachine/`, January 2016.

[32] HPE. HPE Integrity Superdome X. `https://www.hpe.com/us/en/servers/superdome.html`, January 2016.

[33] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency Control and Recovery for Balanced B-link Trees. *The VLDB Journal*, 14(2):257–277, Apr. 2005.

[34] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart Allocation and Replication of Memory For Parallel Programs. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 263–276, Santa Clara, CA, July 2015. USENIX Association.

[35] J. Katcher. PostMark: a new file system benchmark. Network Appliance Tech Report TR3022, Oct. 1997.

[36] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[37] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[38] H. T. Kung and P. L. Lehman. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.*,

5(3):354–382, Sept. 1980.

[39] B. C. Kuszmaul. SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 41–55, New York, NY, USA, 2015. ACM.

[40] P.-A. Larson and M. Krishnan. Memory Allocation for Long-running Server Applications. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 176–185, New York, NY, USA, 1998. ACM.

[41] P. L. Lehman and s. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[42] J. Levandoski, D. Lomet, and S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, Aug. 2013.

[43] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High Performance Transactions in Deuteronomy. Conference on Innovative Data Systems Research (CIDR 2015), January 2015.

[44] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM.

[45] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[47] F. X. Lin and X. Liu. 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, pages 369–383, New York, NY, USA, 2016. ACM.

[48] D. B. Lomet, S. Sengupta, and J. J. Levandoski. 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, pages 302–313, Washington, DC, USA, 2013. IEEE Computer Society.

[49] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-Based Memory Management for Distributed Data Processing Systems. *PVLDB*, 9(12):936–947, Aug. 2016.

[50] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[51] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[52] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[53] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHASH: A Cache-partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 319–320, New York, NY, USA, 2012. ACM.

[54] M. M. Michael. 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, pages 21–30, New York, NY, USA, 2002. ACM.

[55] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[56] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, June 1990.

[57] K. Ren and G. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, 2013. USENIX.

[58] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[59] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

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

[61] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable Locality-conscious Multithreaded Memory Allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 84–94, New York, NY, USA, 2006. ACM.

[62] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[63] K. A. Smith and M. Seltzer. A Comparison of FFS Disk Allocation Policies. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 2–2, Berkeley, CA, USA, 1996. USENIX Association.

[64] V. Srinivasan and M. J. Carey. Performance of B+ Tree Concurrency Control Algorithms. *The VLDB Journal*, 2(4):361–406, Oct. 1993.

[65] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[66] Q. Wang, T. Stamler, and G. Parmer. Parallel Sections: Scaling System-level Data-structures. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 33:1–33:15, New York, NY, USA, 2016. ACM.

[67] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.