Log-structured Memory for DRAM-based Storage

Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout
{rumble, ankitak, ouster}@cs.stanford.edu
Stanford University

Abstract

Traditional memory allocation mechanisms are not suitable for new DRAM-based storage systems because they use memory inefficiently, particularly under changing access patterns. In contrast, a log-structured approach to memory management allows 80-90% memory utilization while offering high performance. The RAMCloud storage system implements a unified log-structured mechanism both for active information in memory and backup data on disk. The RAMCloud implementation of log-structured memory uses a two-level cleaning policy, which conserves disk bandwidth and improves performance up to 6x at high memory utilization. The cleaner runs concurrently with normal operations and employs multiple threads to hide most of the cost of cleaning.

1 Introduction

In recent years a new class of storage systems has arisen in which all data is stored in DRAM. Examples include memcached [2], Redis [3], RAMCloud [30], and Spark [38]. Because of the relatively high cost of DRAM, it is important for these systems to use their memory efficiently. Unfortunately, efficient memory usage is not possible with existing general-purpose storage allocators: they can easily waste half or more of memory, particularly in the face of changing access patterns.

In this paper we show how a log-structured approach to memory management (treating memory as a sequentially-written log) supports memory utilizations of 80-90% while providing high performance. In comparison to non-copying allocators such as malloc, the log-structured approach allows data to be copied to eliminate fragmentation. Copying allows the system to make a fundamental space-time trade-off: for the price of additional CPU cycles and memory bandwidth, copying allows for more efficient use of storage space in DRAM. In comparison to copying garbage collectors, which eventually require a global scan of all data, the log-structured approach provides garbage collection that is more incremental. This results in more efficient collection, which enables higher memory utilization.

We have implemented log-structured memory in the RAMCloud storage system, using a unified approach that handles both information in memory and backup replicas stored on disk or flash memory. The overall architecture is similar to that of a log-structured file system [32], but with several novel aspects:

- In contrast to log-structured file systems, log-structured memory is simpler because it stores very little metadata in the log. The only metadata consists of *log digests* to enable log reassembly after crashes, and *tombstones* to prevent the resurrection of deleted objects.
- RAMCloud uses a *two-level* approach to cleaning, with different policies for cleaning data in memory versus secondary storage. This maximizes DRAM utilization while minimizing disk and network bandwidth usage.
- Since log data is immutable once appended, the log cleaner can run concurrently with normal read and write operations. Furthermore, multiple cleaners can run in separate threads. As a result, *parallel cleaning* hides most of the cost of garbage collection.

Performance measurements of log-structured memory in RAMCloud show that it enables high client throughput at 80-90% memory utilization, even with artificially stressful workloads. In the most stressful workload, a single RAMCloud server can support 270,000-410,000 durable 100-byte writes per second at 90% memory utilization. The two-level approach to cleaning improves performance by up to 6x over a single-level approach at high memory utilization, and reduces disk bandwidth overhead by 7-87x for medium-sized objects (1 to 10 KB). Parallel cleaning effectively hides the cost of cleaning: an active cleaner adds only about 2% to the latency of typical client write requests.

2 Why Not Use Malloc?

An off-the-shelf memory allocator such as the C library's malloc function might seem like a natural choice for an in-memory storage system. However, existing allocators are not able to use memory efficiently, particularly in the face of changing access patterns. We measured a variety of allocators under synthetic workloads and found that all of them waste at least 50% of memory under conditions that seem plausible for a storage system.

Memory allocators fall into two general classes: non-copying allocators and copying allocators. *Non-copying* allocators such as malloc cannot move an object once it has been allocated, so they are vulnerable to fragmentation. Non-copying allocators work well for individual applications with a consistent distribution of object sizes, but Figure 1 shows that they can easily waste half of memory when allocation patterns change. For example, every allocator we measured performed poorly when 10 GB of small objects were mostly deleted, then replaced with 10 GB of much larger objects.

Changes in size distributions may be rare in individual

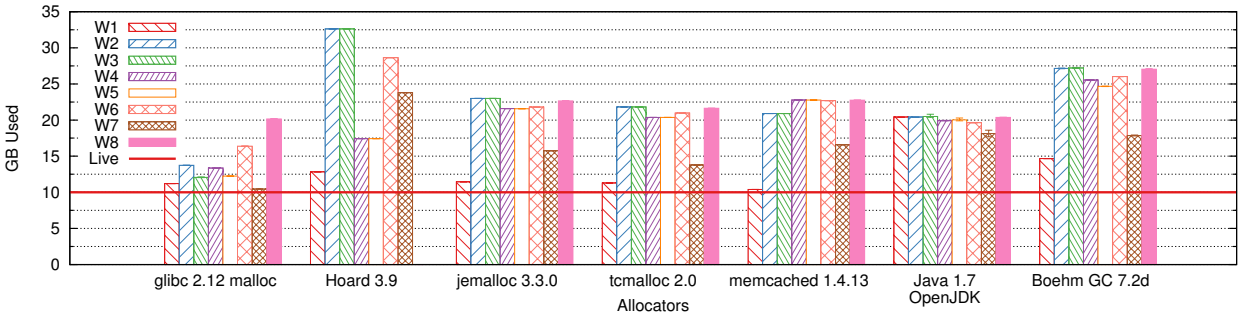

Figure 1: Total memory needed by allocators to support 10 GB of live data under the changing workloads described in Table 1 (average of 5 runs). "Live" indicates the amount of live data, and represents an optimal result. "glibc" is the allocator typically used by C and C++ applications on Linux. "Hoard" [10], "jemalloc" [19], and "tcmalloc" [1] are non-copying allocators designed for speed and multiprocessor scalability. "Memcached" is the slab-based allocator used in the memcached [2] object caching system. "Java" is the JVM's default parallel scavenging collector with no maximum heap size restriction (it ran out of memory if given less than 16 GB of total space). "Boehm GC" is a non-copying garbage collector for C and C++. Hoard could not complete the W8 workload (it overburdened the kernel by *mmap*ing each large allocation separately).

Workload	Before	Delete	After
W1	Fixed 100 Bytes	N/A	N/A
W2	Fixed 100 Bytes	0%	Fixed 130 Bytes
W3	Fixed 100 Bytes	90%	Fixed 130 Bytes
W4	Uniform 100 - 150 Bytes	0%	Uniform 200 - 250 Bytes
W5	Uniform 100 - 150 Bytes	90%	Uniform 200 - 250 Bytes
W6	Uniform 100 - 200 Bytes	50%	Uniform 1,000 - 2,000 Bytes
W7	Uniform 1,000 - 2,000 Bytes	90%	Uniform 1,500 - 2,500 Bytes
W8	Uniform 50 - 150 Bytes	90%	Uniform 5,000 - 15,000 Bytes

Table 1: Summary of workloads used in Figure 1. The workloads were not intended to be representative of actual application behavior, but rather to illustrate plausible workload changes that might occur in a shared storage system. Each workload consists of three phases. First, the workload allocates 50 GB of memory using objects from a particular size distribution; it deletes existing objects at random in order to keep the amount of live data from exceeding 10 GB. In the second phase the workload deletes a fraction of the existing objects at random. The third phase is identical to the first except that it uses a different size distribution (objects from the new distribution gradually displace those from the old distribution). Two size distributions were used: "Fixed" means all objects had the same size, and "Uniform" means objects were chosen uniform randomly over a range (non-uniform distributions yielded similar results). All workloads were single-threaded and ran on a Xeon E5-2670 system with Linux 2.6.32.

applications, but they are more likely in storage systems that serve many applications over a long period of time. Such shifts can be caused by changes in the set of applications using the system (adding new ones and/or removing old ones), by changes in application phases (switching from map to reduce), or by application upgrades that increase the size of common records (to include additional fields for new features). For example, workload W2 in Figure 1 models the case where the records of a table are expanded from 100 bytes to 130 bytes. Facebook encountered distribution changes like this in its memcached storage systems and was forced to introduce special-purpose cache eviction code for specific situations [28]. Non-copying allocators will work well in many cases, but they are unstable: a small application change could dramatically change the efficiency of the storage system. Unless excess memory is retained to handle the worst-case change, an application could suddenly find itself unable to make progress.

The second class of memory allocators consists of those that can move objects after they have been created, such as copying garbage collectors. In principle, garbage collectors can solve the fragmentation problem by moving live data to coalesce free heap space. However, this comes with a trade-off: at some point all of these collectors (even those that label themselves as "incremental") must walk all live data, relocate it, and update references. This is an expensive operation that scales poorly, so garbage collectors delay global collections until a large amount of garbage has accumulated. As a result, they typically require 1.5-5x as much space as is actually used in order to maintain high performance [39, 23]. This erases any space savings gained by defragmenting memory.

Pause times are another concern with copying garbage collectors. At some point all collectors must halt the processes' threads to update references when objects are moved. Although there has been considerable work on real-time garbage collectors, even state-of-art solutions have maximum pause times of hundreds of microseconds, or even milliseconds [8, 13, 36] – this is 100 to 1,000 times longer than the round-trip time for a RAMCloud RPC. All of the standard Java collectors we measured exhibited pauses of 3 to 4 seconds by default (2-4 times longer than it takes RAMCloud to detect a failed server and reconstitute 64 GB of lost data [29]). We experimented with features of the JVM collectors that re-

duce pause times, but memory consumption increased by an additional 30% and we still experienced occasional pauses of one second or more.

An ideal memory allocator for a DRAM-based storage system such as RAMCloud should have two properties. First, it must be able to copy objects in order to eliminate fragmentation. Second, it must not require a global scan of memory: instead, it must be able to perform the copying *incrementally*, garbage collecting small regions of memory independently with cost proportional to the size of a region. Among other advantages, the incremental approach allows the garbage collector to focus on regions with the most free space. In the rest of this paper we will show how a log-structured approach to memory management achieves these properties.

In order for incremental garbage collection to work, it must be possible to find the pointers to an object without scanning all of memory. Fortunately, storage systems typically have this property: pointers are confined to index structures where they can be located easily. Traditional storage allocators work in a harsher environment where the allocator has no control over pointers; the log-structured approach could not work in such environments.

3 RAMCloud Overview

Our need for a memory allocator arose in the context of RAMCloud. This section summarizes the features of RAMCloud that relate to its mechanisms for storage management, and motivates why we used log-structured memory instead of a traditional allocator.

RAMCloud is a storage system that stores data in the DRAM of hundreds or thousands of servers within a datacenter, as shown in Figure 2. It takes advantage of low-latency networks to offer remote read times of 5μs and write times of 16μs (for small objects). Each storage server contains two components. A *master* module manages the main memory of the server to store RAMCloud objects; it handles read and write requests from clients. A *backup* module uses local disk or flash memory to store backup copies of data owned by masters on other servers. The masters and backups are managed by a central *coordinator* that handles configuration-related issues such as cluster membership and the distribution of data among the servers. The coordinator is not normally involved in common operations such as reads and writes. All RAMCloud data is present in DRAM at all times; secondary storage is used only to hold duplicate copies for crash recovery.

RAMCloud provides a simple key-value data model consisting of uninterpreted data blobs called *objects* that are named by variable-length *keys*. Objects are grouped into *tables* that may span one or more servers in the cluster. Objects must be read or written in their entirety. RAMCloud is optimized for small objects – a few hundred bytes or less – but supports objects up to 1 MB.

Each master's memory contains a collection of objects stored in DRAM and a hash table (see Figure 3). The

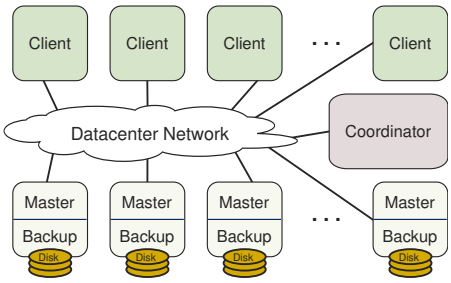

Figure 2: RAMCloud cluster architecture.

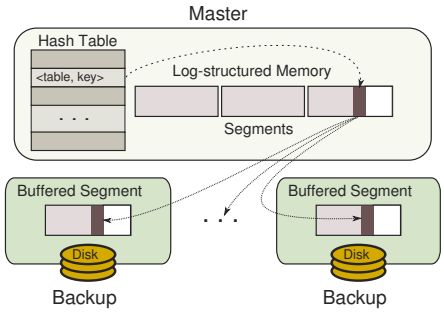

Figure 3: Master servers consist primarily of a hash table and an in-memory log, which is replicated across several backups for durability.

hash table contains one entry for each object stored on that master; it allows any object to be located quickly, given its table and key. Each live object has exactly one pointer, which is stored in its hash table entry.

In order to ensure data durability in the face of server crashes and power failures, each master must keep backup copies of its objects on the secondary storage of other servers. The backup data is organized as a log for maximum efficiency. Each master has its own log, which is divided into 8 MB pieces called *segments*. Each segment is replicated on several backups (typically two or three). A master uses a different set of backups to replicate each segment, so that its segment replicas end up scattered across the entire cluster.

When a master receives a write request from a client, it adds the new object to its memory, then forwards information about that object to the backups for its current head segment. The backups append the new object to segment replicas stored in nonvolatile buffers; they respond to the master as soon as the object has been copied into their buffer, without issuing an I/O to secondary storage (backups must ensure that data in buffers can survive power failures). Once the master has received replies from all the backups, it responds to the client. Each backup accumulates data in its buffer until the segment is complete. At that point it writes the segment to secondary storage and reallocates the buffer for another segment. This approach has two performance advantages: writes complete without waiting for I/O to secondary storage, and backups use secondary storage bandwidth efficiently by performing I/O in large blocks, even if objects are small.

RAMCloud could have used a traditional storage allocator for the objects stored in a master's memory, but we chose instead to use the same log structure in DRAM that is used on disk. Thus a master's object storage consists of 8 MB segments that are identical to those on secondary storage. This approach has three advantages. First, it avoids the allocation inefficiencies described in Section 2. Second, it simplifies RAMCloud by using a single unified mechanism for information both in memory and on disk. Third, it saves memory: in order to perform log cleaning (described below), the master must enumerate all of the objects in a segment; if objects were stored in separately allocated areas, they would need to be linked together by segment, which would add an extra 8-byte pointer per object (an 8% memory overhead for 100-byte objects).

The segment replicas stored on backups are never read during normal operation; most are deleted before they have ever been read. Backup replicas are only read during crash recovery (for details, see [29]). Data is never read from secondary storage in small chunks; the only read operation is to read a master's entire log.

RAMCloud uses a *log cleaner* to reclaim free space that accumulates in the logs when objects are deleted or overwritten. Each master runs a separate cleaner, using a basic mechanism similar to that of LFS [32]:

- The cleaner selects several segments to clean, using the same cost-benefit approach as LFS (segments are chosen for cleaning based on the amount of free space and the age of the data).
- For each of these segments, the cleaner scans the segment stored in memory and copies any live objects to new *survivor segments*. Liveness is determined by checking for a reference to the object in the hash table. The live objects are sorted by age to improve the efficiency of cleaning in the future. Unlike LFS, RAMCloud need not read objects from secondary storage during cleaning.
- The cleaner makes the old segments' memory available for new segments, and it notifies the backups for those segments that they can reclaim the replicas' storage.

The logging approach meets the goals from Section 2: it copies data to eliminate fragmentation, and it operates incrementally, cleaning a few segments at a time. However, it introduces two additional issues. First, the log must contain metadata in addition to objects, in order to ensure safe crash recovery; this issue is addressed in Section 4. Second, log cleaning can be quite expensive at high memory utilization [34, 35]. RAMCloud uses two techniques to reduce the impact of log cleaning: two-level cleaning (Section 5) and parallel cleaning with multiple threads (Section 6).

4 Log Metadata

In log-structured file systems, the log contains a lot of indexing information in order to provide fast random access to data in the log. In contrast, RAMCloud has a separate hash table that provides fast access to information in memory. The on-disk log is never read during normal use; it is used only during recovery, at which point it is read in its entirety. As a result, RAMCloud requires only three kinds of metadata in its log, which are described below.

First, each object in the log must be self-identifying: it contains the table identifier, key, and version number for the object in addition to its value. When the log is scanned during crash recovery, this information allows RAMCloud to identify the most recent version of an object and reconstruct the hash table.

Second, each new log segment contains a *log digest* that describes the entire log. Every segment has a unique identifier, and the log digest is a list of identifiers for all the segments that currently belong to the log. Log digests avoid the need for a central repository of log information (which would create a scalability bottleneck and introduce other crash recovery problems). To replay a crashed master's log, RAMCloud locates the latest digest and loads each segment enumerated in it (see [29] for details).

The third kind of log metadata is *tombstones* that identify deleted objects. When an object is deleted or modified, RAMCloud does not modify the object's existing record in the log. Instead, it appends a *tombstone* record to the log. The tombstone contains the table identifier, key, and version number for the object that was deleted. Tombstones are ignored during normal operation, but they distinguish live objects from dead ones during crash recovery. Without tombstones, deleted objects would come back to life when logs are replayed during crash recovery.

Tombstones have proven to be a mixed blessing in RAMCloud: they provide a simple mechanism to prevent object resurrection, but they introduce additional problems of their own. One problem is tombstone garbage collection. Tombstones must eventually be removed from the log, but this is only safe if the corresponding objects have been cleaned (so they will never be seen during crash recovery). To enable tombstone deletion, each tombstone includes the identifier of the segment containing the obsolete object. When the cleaner encounters a tombstone in the log, it checks the segment referenced in the tombstone. If that segment is no longer part of the log, then it must have been cleaned, so the old object no longer exists and the tombstone can be deleted. If the segment still exists in the log, then the tombstone must be preserved.

5 Two-level Cleaning

Almost all of the overhead for log-structured memory is due to cleaning. Allocating new storage is trivial; new objects are simply appended at the end of the head segment. However, reclaiming free space is much more expensive. It requires running the log cleaner, which will have to copy live data out of the segments it chooses for cleaning as described in Section 3. Unfortunately, the cost of log cleaning rises rapidly as memory utilization in-

creases. For example, if segments are cleaned when 80% of their data are still live, the cleaner must copy 8 bytes of live data for every 2 bytes it frees. At 90% utilization, the cleaner must copy 9 bytes of live data for every 1 byte freed. Eventually the system will run out of bandwidth and write throughput will be limited by the speed of the cleaner. Techniques like cost-benefit segment selection [32] help by skewing the distribution of free space, so that segments chosen for cleaning have lower utilization than the overall average, but they cannot eliminate the fundamental tradeoff between utilization and cleaning cost. Any copying storage allocator will suffer from intolerable overheads as utilization approaches 100%.

Originally, disk and memory cleaning were tied together in RAMCloud: cleaning was first performed on segments in memory, then the results were reflected to the backup copies on disk. This made it impossible to achieve both high memory utilization and high write throughput. For example, if we used memory at high utilization (80-90%) write throughput would be severely limited by the cleaner's usage of disk bandwidth (see Section 8). On the other hand, we could have improved write bandwidth by increasing the size of the disk log to reduce its average utilization. For example, at 50% disk utilization we could achieve high write throughput. Furthermore, disks are cheap enough that the cost of the extra space would not be significant. However, disk and memory were fundamentally tied together: if we reduced the utilization of disk space, we would also have reduced the utilization of DRAM, which was unacceptable.

The solution is to clean the disk and memory logs independently – we call this *two-level cleaning*. With two-level cleaning, memory can be cleaned without reflecting the updates on backups. As a result, memory can have higher utilization than disk. The cleaning cost for memory will be high, but DRAM can easily provide the bandwidth required to clean at 90% utilization or higher. Disk cleaning happens less often. The disk log becomes larger than the in-memory log, so it has lower overall utilization, and this reduces the bandwidth required for cleaning.

The first level of cleaning, called *segment compaction*, operates only on the in-memory segments on masters and consumes no network or disk I/O. It compacts a single segment at a time, copying its live data into a smaller region of memory and freeing the original storage for new segments. Segment compaction maintains the same logical log in memory and on disk: each segment in memory still has a corresponding segment on disk. However, the segment in memory takes less space because deleted objects and obsolete tombstones were removed (Figure 4).

The second level of cleaning is just the mechanism described in Section 3. We call this *combined cleaning* because it cleans both disk and memory together. Segment compaction makes combined cleaning more efficient by postponing it. The effect of cleaning a segment later is that more objects have been deleted, so the segment's utilization will be lower. The result is that when combined cleaning does happen, less bandwidth is required to reclaim the same amount of free space. For example, if the disk log is allowed to grow until it consumes twice as much space as the log in memory, the utilization of segments cleaned on disk will never be greater than 50%, which makes cleaning relatively efficient.

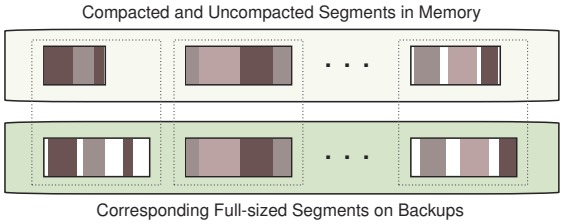

Figure 4: Compacted segments in memory have variable length because unneeded objects and tombstones have been removed, but the corresponding segments on disk remain full-size. As a result, the utilization of memory is higher than that of disk, and disk can be cleaned more efficiently.

Two-level cleaning leverages the strengths of memory and disk to compensate for their weaknesses. For memory, space is precious but bandwidth for cleaning is plentiful, so we use extra bandwidth to enable higher utilization. For disk, space is plentiful but bandwidth is precious, so we use extra space to save bandwidth.

5.1 Seglets

In the absence of segment compaction, all segments are the same size, which makes memory management simple. With compaction, however, segments in memory can have different sizes. One possible solution is to use a standard heap allocator to allocate segments, but this would result in the fragmentation problems described in Section 2. Instead, each RAMCloud master divides its log memory into fixed-size 64 KB *seglets*. A segment consists of a collection of seglets, and the number of seglets varies with the size of the segment. Because seglets are fixed-size, they introduce a small amount of internal fragmentation (one-half seglet for each segment, on average). In practice, fragmentation should be less than 1% of memory space, since we expect compacted segments to average at least half the length of a full-size segment. In addition, seglets require extra mechanism to handle log entries that span discontiguous seglets (before seglets, log entries were always contiguous).

5.2 When to Clean on Disk?

Two-level cleaning introduces a new policy question: when should the system choose memory compaction over combined cleaning, and vice-versa? This choice has an important impact on system performance because combined cleaning consumes precious disk and network I/O resources. However, as we explain below, memory compaction is not always more efficient. This section explains how these considerations resulted in RAMCloud's current

policy module; we refer to it as the *balancer*. For a more complete discussion of the balancer, see [33].

There is no point in running either cleaner until the system is running low on memory or disk space. The reason is that cleaning early is never cheaper than cleaning later on. The longer the system delays cleaning, the more time it has to accumulate dead objects, which lowers the fraction of live data in segments and makes them less expensive to clean.

The balancer determines that memory is running low as follows. Let L be the fraction of all memory occupied by live objects and F be the fraction of memory in unallocated seglets. One of the cleaners will run whenever $F \leq min(0.1, (1 - L)/2)$ In other words, cleaning occurs if the unallocated seglet pool has dropped to less than 10% of memory and at least half of the free memory is in active segments (vs. unallocated seglets). This formula represents a tradeoff: on the one hand, it delays cleaning to make it more efficient; on the other hand, it starts cleaning soon enough for the cleaner to collect free memory before the system runs out of unallocated seglets.

Given that the cleaner must run, the balancer must choose which cleaner to use. In general, compaction is preferred because it is more efficient, but there are two cases in which the balancer must choose combined cleaning. The first is when too many tombstones have accumulated. The problem with tombstones is that memory compaction alone cannot remove them: the combined cleaner must first remove dead objects from disk before their tombstones can be erased. As live tombstones pile up, segment utilizations increase and compaction becomes more and more expensive. Eventually, tombstones would eat up all free memory. Combined cleaning ensures that tombstones do not exhaust memory and makes future compactions more efficient.

The balancer detects tombstone accumulation as follows. Let T be the fraction of memory occupied by live tombstones, and L be the fraction of live objects (as above). Too many tombstones have accumulated once $T/(1 - L) \geq 40\%$. In other words, there are too many tombstones when they account for 40% of the freeable space in a master ($1 - L$; i.e., all tombstones and dead objects). The 40% value was chosen empirically based on measurements of different workloads, object sizes, and amounts of available disk bandwidth. This policy tends to run the combined cleaner more frequently under workloads that make heavy use of small objects (tombstone space accumulates more quickly as a fraction of freeable space, because tombstones are nearly as large as the objects they delete).

The second reason the combined cleaner must run is to bound the growth of the on-disk log. The size must be limited both to avoid running out of disk space and to keep crash recovery fast (since the entire log must be replayed, its size directly affects recovery speed). RAMCloud implements a configurable *disk expansion factor* that sets the maximum on-disk log size as a multiple of the in-memory log size. The combined cleaner runs when the on-disk log size exceeds 90% of this limit.

Finally, the balancer chooses memory compaction when unallocated memory is low and combined cleaning is not needed (disk space is not low and tombstones have not accumulated yet).

6 Parallel Cleaning

Two-level cleaning reduces the cost of combined cleaning, but it adds a significant new cost in the form of segment compaction. Fortunately, the cost of cleaning can be hidden by performing both combined cleaning and segment compaction concurrently with normal read and write requests. RAMCloud employs multiple cleaner threads simultaneously to take advantage of multi-core CPUs.

Parallel cleaning in RAMCloud is greatly simplified by the use of a log structure and simple metadata. For example, since segments are immutable after they are created, the cleaner need not worry about objects being modified while the cleaner is copying them. Furthermore, the hash table provides a simple way of redirecting references to objects that are relocated by the cleaner (all objects are accessed indirectly through it). This means that the basic cleaning mechanism is very straightforward: the cleaner copies live data to new segments, atomically updates references in the hash table, and frees the cleaned segments.

There are three points of contention between cleaner threads and service threads handling read and write requests. First, both cleaner and service threads need to add data at the head of the log. Second, the threads may conflict in updates to the hash table. Third, the cleaner must not free segments that are still in use by service threads. These issues and their solutions are discussed in the subsections below.

6.1 Concurrent Log Updates

The most obvious way to perform cleaning is to copy the live data to the head of the log. Unfortunately, this would create contention for the log head between cleaner threads and service threads that are writing new data.

RAMCloud's solution is for the cleaner to write survivor data to different segments than the log head. Each cleaner thread allocates a separate set of segments for its survivor data. Synchronization is required when allocating segments, but once segments are allocated, each cleaner thread can copy data to its own survivor segments without additional synchronization. Meanwhile, request-processing threads can write new data to the log head. Once a cleaner thread finishes a cleaning pass, it arranges for its survivor segments to be included in the next log digest, which inserts them into the log; it also arranges for the cleaned segments to be dropped from the next digest.

Using separate segments for survivor data has the additional benefit that the replicas for survivor segments will be stored on a different set of backups than the replicas

of the head segment. This allows the survivor segment replicas to be written in parallel with the log head replicas without contending for the same backup disks, which increases the total throughput for a single master.

6.2 Hash Table Contention

The main source of thread contention during cleaning is the hash table. This data structure is used both by service threads and cleaner threads, as it indicates which objects are alive and points to their current locations in the in-memory log. The cleaner uses the hash table to check whether an object is alive (by seeing if the hash table currently points to that exact object). If the object is alive, the cleaner copies it and updates the hash table to refer to the new location in a survivor segment. Meanwhile, service threads may be using the hash table to find objects during read requests and they may update the hash table during write or delete requests. To ensure consistency while reducing contention, RAMCloud currently uses fine-grained locks on individual hash table buckets. In the future we plan to explore lockless approaches to eliminate this overhead.

6.3 Freeing Segments in Memory

Once a cleaner thread has cleaned a segment, the segment's storage in memory can be freed for reuse. At this point, future service threads will not use data in the cleaned segment, because there are no hash table entries pointing into it. However, it could be that a service thread began using the data in the segment before the cleaner updated the hash table; if so, the cleaner must not free the segment until the service thread has finished using it.

To solve this problem, RAMCloud uses a simple mechanism similar to RCU's [27] *wait-for-readers* primitive and Tornado/K42's *generations* [6]: after a segment has been cleaned, the system will not free it until all RPCs currently being processed complete. At this point it is safe to reuse the segment's memory, since new RPCs cannot reference the segment. This approach has the advantage of not requiring additional locks for normal reads and writes.

6.4 Freeing Segments on Disk

Once a segment has been cleaned, its replicas on backups must also be freed. However, this must not be done until the corresponding survivor segments have been safely incorporated into the on-disk log. This takes two steps. First, the survivor segments must be fully replicated on backups. Survivor segments are transmitted to backups asynchronously during cleaning, so at the end of each cleaning pass the cleaner must wait for all of its survivor segments to be received by backups. Second, a new log digest must be written, which includes the survivor segments and excludes the cleaned segments. Once the digest has been durably written to backups, RPCs are issued to free the replicas for the cleaned segments.

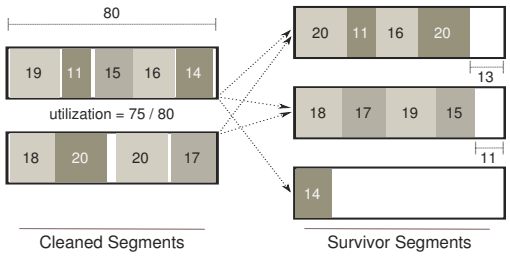

Figure 5: A simplified situation in which cleaning uses more space than it frees. Two 80-byte segments at about 94% utilization are cleaned: their objects are reordered by age (not depicted) and written to survivor segments. The label in each object indicates its size. Because of fragmentation, the last object (size 14) overflows into a third survivor segment.

7 Avoiding Cleaner Deadlock

Since log cleaning copies data before freeing it, the cleaner must have free memory space to work with before it can generate more. If there is no free memory, the cleaner cannot proceed and the system will deadlock. RAMCloud increases the risk of memory exhaustion by using memory at high utilization. Furthermore, it delays cleaning as long as possible in order to allow more objects to be deleted. Finally, two-level cleaning allows tombstones to accumulate, which consumes even more free space. This section describes how RAMCloud prevents cleaner deadlock while maximizing memory utilization.

The first step is to ensure that there are always free seglets for the cleaner to use. This is accomplished by reserving a special pool of seglets for the cleaner. When seglets are freed, they are used to replenish the cleaner pool before making space available for other uses.

The cleaner pool can only be maintained if each cleaning pass frees as much space as it uses; otherwise the cleaner could gradually consume its own reserve and then deadlock. However, RAMCloud does not allow objects to cross segment boundaries, which results in some wasted space at the end of each segment. When the cleaner reorganizes objects, it is possible for the survivor segments to have greater fragmentation than the original segments, and this could result in the survivors taking more total space than the original segments (see Figure 5).

To ensure that the cleaner always makes forward progress, it must produce at least enough free space to compensate for space lost to fragmentation. Suppose that N segments are cleaned in a particular pass and the fraction of free space in these segments is F; furthermore, let S be the size of a full segment and O the maximum object size. The cleaner will produce $NS(1-F)$ bytes of live data in this pass. Each survivor segment could contain as little as $S-O+1$ bytes of live data (if an object of size O couldn't quite fit at the end of the segment), so the maximum number of survivor segments will be $\lceil \frac{NS(1-F)}{S-O+1} \rceil$. The last seglet of each survivor segment could be empty except for a single byte, resulting in almost a full seglet of

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2x Crucial M4 SSDs CT128M4SSD2 (128 GB)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switch	Mellanox SX6036 (4X FDR)

Table 2: The server hardware configuration used for benchmarking. All nodes ran Linux 2.6.32 and were connected to an Infiniband fabric.

fragmentation for each survivor segment. Thus, F must be large enough to produce a bit more than one seglet's worth of free data for each survivor segment generated. For RAMCloud, we conservatively require 2% of free space per cleaned segment, which is a bit more than two seglets. This number could be reduced by making seglets smaller.

There is one additional problem that could result in memory deadlock. Before freeing segments after cleaning, RAMCloud must write a new log digest to add the survivors to the log and remove the old segments. Writing a new log digest means writing a new log head segment (survivor segments do not contain digests). Unfortunately, this consumes yet another segment, which could contribute to memory exhaustion. Our initial solution was to require each cleaner pass to produce enough free space for the new log head segment, in addition to replacing the segments used for survivor data. However, it is hard to guarantee "better than break-even" cleaner performance when there is very little free space.

The current solution takes a different approach: it reserves two special *emergency head segments* that contain only log digests; no other data is permitted. If there is no free memory after cleaning, one of these segments is allocated for the head segment that will hold the new digest. Since the segment contains no objects or tombstones, it does not need to be cleaned; it is immediately freed when the next head segment is written (the emergency head is not included in the log digest for the next head segment). By keeping two emergency head segments in reserve, RAMCloud can alternate between them until a full segment's worth of space is freed and a proper log head can be allocated. As a result, each cleaner pass only needs to produce as much free space as it uses.

By combining these techniques, RAMCloud can guarantee deadlock-free cleaning with total memory utilization as high as 98%. When utilization reaches this limit, no new data (or tombstones) can be appended to the log until the cleaner has freed space. However, RAMCloud sets a lower utilization limit for writes, in order to reserve space for tombstones. Otherwise all available log space could be consumed with live data and there would be no way to add tombstones to delete objects.

8 Evaluation

All of the features described in the previous sections are implemented in RAMCloud version 1.0, which was released in January, 2014. This section describes a series of experiments we ran to evaluate log-structured memory and its implementation in RAMCloud. The key results are:

- RAMCloud supports memory utilizations of 80-90% without significant loss in performance.
- At high memory utilizations, two-level cleaning improves client throughput up to 6x over a single-level approach.
- Log-structured memory also makes sense for other DRAM-based storage systems, such as memcached.
- RAMCloud provides a better combination of durability and performance than other storage systems such as HyperDex and Redis.

Note: all plots in this section show the average of 3 or more runs, with error bars for minimum and maximum values.

8.1 Performance vs. Utilization

The most important metric for log-structured memory is how it performs at high memory utilization. In Section 2 we found that other allocators could not achieve high memory utilization in the face of changing workloads. With log-structured memory, we can choose any utilization up to the deadlock limit of about 98% described in Section 7. However, system performance will degrade as memory utilization increases; thus, the key question is how efficiently memory can be used before performance drops significantly. Our hope at the beginning of the project was that log-structured memory could support memory utilizations in the range of 80-90%.

The measurements in this section used an 80-node cluster of identical commodity servers (see Table 2). Our primary concern was the throughput of a single master, so we divided the cluster into groups of five servers and used different groups to measure different data points in parallel. Within each group, one node ran a master server, three nodes ran backups, and the last node ran the coordinator and client benchmark. This configuration provided each master with about 700 MB/s of back-end bandwidth. In an actual RAMCloud system the back-end bandwidth available to one master could be either more or less than this; we experimented with different back-end bandwidths and found that it did not change any of our conclusions. Each byte stored on a master was replicated to three different backups for durability.

All of our experiments used a maximum of two threads for cleaning. Our cluster machines have only four cores, and the main RAMCloud server requires two of them, so there were only two cores available for cleaning (we have not yet evaluated the effect of hyperthreading on RAMCloud's throughput or latency).

In each experiment, the master was given 16 GB of log space and the client created objects with sequential keys until it reached a target memory utilization; then it over-

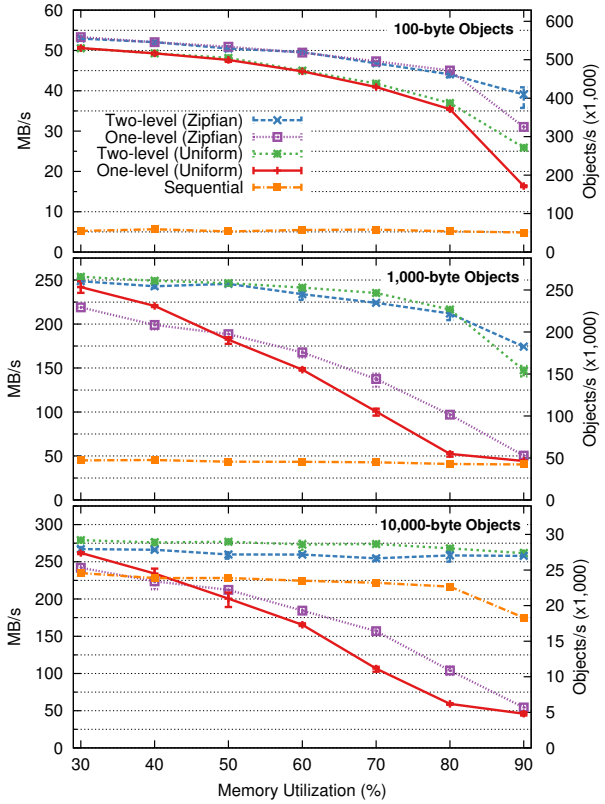

Figure 6: End-to-end client write performance as a function of memory utilization. For some experiments two-level cleaning was disabled, so only the combined cleaner was used. The "Sequential" curve used two-level cleaning and uniform access patterns with a single outstanding write request at a time. All other curves used the high-stress workload with concurrent multi-writes. Each point is averaged over 3 runs on different groups of servers.

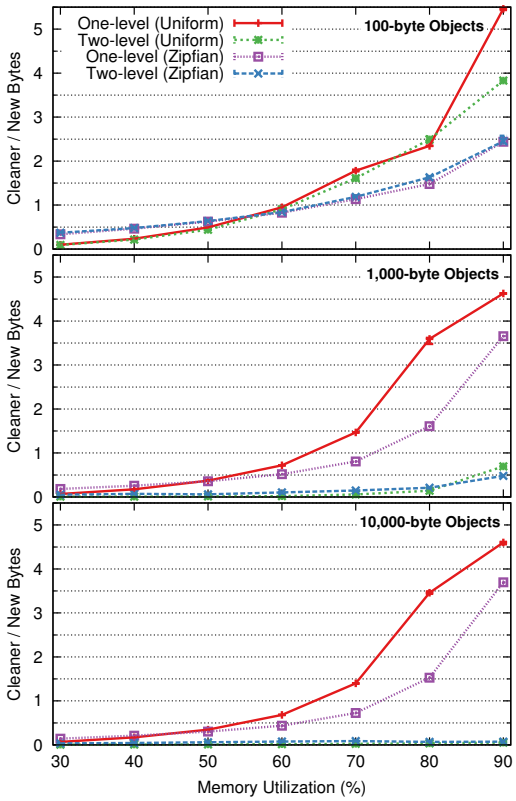

Figure 7: Cleaner bandwidth overhead (ratio of cleaner bandwidth to regular log write bandwidth) for the workloads in Figure 6. 1 means that for every byte of new data written to backups, the cleaner writes 1 byte of live data to backups while freeing segment space. The optimal ratio is 0.

wrote objects (maintaining a fixed amount of live data continuously) until the overhead for cleaning converged to a stable value.

We varied the workload in four ways to measure system behavior under different operating conditions:

1. Object Size: RAMCloud's performance depends on average object size (e.g., per-object overheads versus memory copying overheads), but not on the exact size distribution (see Section 8.5 for supporting evidence). Thus, unless otherwise noted, the objects for each test had the same fixed size. We ran different tests with sizes of 100, 1000, 10000, and 100,000 bytes (we omit the 100 KB measurements, since they were nearly identical to 10 KB).

2. Memory Utilization: The percentage of DRAM used for holding live data (not including tombstones) was fixed in each test. For example, at 50% and 90% utilization there were 8 GB and 14.4 GB of live data, respectively. In some experiments, total memory utilization was significantly higher than the listed number due to an accumulation of tombstones.

3. Locality: We ran experiments with both uniform random overwrites of objects and a Zipfian distribution in which 90% of writes were made to 15% of the objects. The uniform random case represents a workload with no locality; Zipfian represents locality similar to what has been observed in memcached deployments [7].

4. Stress Level: For most of the tests we created an artificially high workload in order to stress the master to its limit. To do this, the client issued write requests asynchronously, with 10 requests outstanding at any given time. Furthermore, each request was a multi-write containing 75 individual writes. We also ran tests where the client issued one synchronous request at a time, with a single write operation in each request; these tests are labeled "Sequential" in the graphs.

Figure 6 graphs the overall throughput of a RAMCloud master with different memory utilizations and workloads. With two-level cleaning enabled, client throughput drops only 10-20% as memory utilization increases from 30% to 80%, even with an artificially high workload. Throughput drops more significantly at 90% utilization: in the worst case (small objects with no locality), throughput at 90% utilization is about half that at 30%. At high utilization the cleaner is limited by disk bandwidth and cannot keep up with write traffic; new writes quickly exhaust all available segments and must wait for the cleaner.

These results exceed our original performance goals for RAMCloud. At the start of the project, we hoped that each RAMCloud server could support 100K small writes per second, out of a total of one million small operations per second. Even at 90% utilization, RAMCloud can support almost 410K small writes per second with some locality and nearly 270K with no locality.

If actual RAMCloud workloads are similar to our "Sequential" case, then it should be reasonable to run RAMCloud clusters at 90% memory utilization (for 100 and 1,000B objects there is almost no performance degradation). If workloads include many bulk writes, like most of the measurements in Figure 6, then it makes more sense to run at 80% utilization: the higher throughput will more than offset the 12.5% additional cost for memory.

Compared to the traditional storage allocators measured in Section 2, log-structured memory permits significantly higher memory utilization.

8.2 Two-Level Cleaning

Figure 6 also demonstrates the benefits of two-level cleaning. The figure contains additional measurements in which segment compaction was disabled ("One-level"); in these experiments, the system used RAMCloud's original one-level approach where only the combined cleaner ran. The two-level cleaning approach provides a considerable performance improvement: at 90% utilization, client throughput is up to 6x higher with two-level cleaning than single-level cleaning.

One of the motivations for two-level cleaning was to reduce the disk bandwidth used by cleaning, in order to make more bandwidth available for normal writes. Figure 7 shows that two-level cleaning reduces disk and network bandwidth overheads at high memory utilizations. The greatest benefits occur with larger object sizes, where two-level cleaning reduces overheads by 7-87x. Compaction is much more efficient in these cases because there are fewer objects to process.

8.3 CPU Overhead of Cleaning

Figure 8 shows the CPU time required for cleaning in two of the workloads from Figure 6. Each bar represents the average number of fully active cores used for combined cleaning and compaction in the master, as well as for backup RPC and disk I/O processing in the backups.

At low memory utilization a master under heavy load uses about 30-50% of one core for cleaning; backups account for the equivalent of at most 60% of one core across all six of them. Smaller objects require more CPU time for cleaning on the master due to per-object overheads, while larger objects stress backups more because the master can write up to 5 times as many megabytes per second (Figure 6). As free space becomes more scarce, the two cleaner threads are eventually active nearly all of the time. In the 100B case, RAMCloud's balancer prefers to run combined cleaning due to the accumulation of tombstones. With larger objects compaction tends to be more efficient, so combined cleaning accounts for only a small fraction of the CPU time.

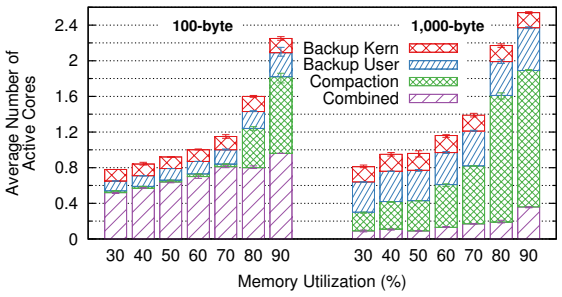

Figure 8: CPU overheads for two-level cleaning under the 100 and 1,000-byte Zipfian workloads in Figure 6, measured in average number of active cores. "Backup Kern" represents kernel time spent issuing I/Os to disks, and "Backup User" represents time spent servicing segment write RPCs on backup servers. Both of these bars are aggregated across all backups, and include traffic for normal writes as well as cleaning. "Compaction" and "Combined" represent time spent on the master in memory compaction and combined cleaning. Additional core usage unrelated to cleaning is not depicted. Each bar is averaged over 3 runs.

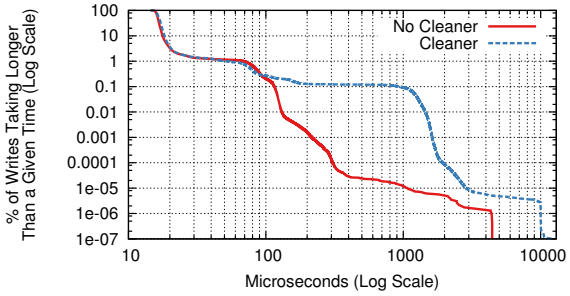

Figure 9: Reverse cumulative distribution of client write latencies when a single client issues back-to-back write requests for 100-byte objects using the uniform distribution. The "No cleaner" curve was measured with cleaning disabled. The "Cleaner" curve shows write latencies at 90% memory utilization with cleaning enabled. For example, about 10% of all write requests took longer than 18μs in both cases; with cleaning enabled, about 0.1% of all write requests took 1ms or more. The median latency was 16.70μs with cleaning enabled and 16.35μs with the cleaner disabled.

8.4 Can Cleaning Costs be Hidden?

One of the goals for RAMCloud's implementation of log-structured memory was to hide the cleaning costs so they don't affect client requests. Figure 9 graphs the latency of client write requests in normal operation with a cleaner running, and also in a special setup where the cleaner was disabled. The distributions are nearly identical up to about the 99.9th percentile, and cleaning only increased the median latency by 2% (from 16.35 to 16.70μs). About 0.1% of write requests suffer an additional 1ms or greater delay when cleaning. Preliminary

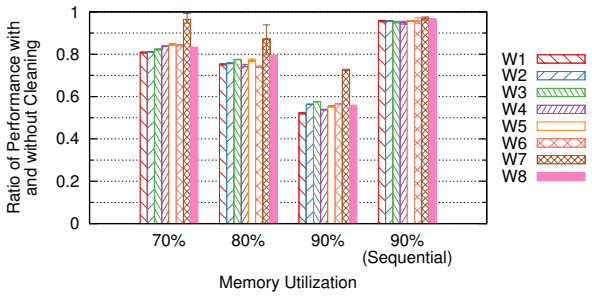

Figure 10: Client performance in RAMCloud under the same workloads as in Figure 1 from Section 2. Each bar measures the performance of a workload (with cleaning enabled) relative to the performance of the same workload with cleaning disabled. Higher is better and 1.0 is optimal; it means that the cleaner has no impact on the processing of normal requests. As in Figure 1, 100 GB of allocations were made and at most 10 GB of data was alive at once. The 70%, 80%, and 90% utilization bars were measured with the high-stress request pattern using concurrent multi-writes. The "Sequential" bars used a single outstanding write request at a time; the data size was scaled down by a factor of 10x for these experiments to make running times manageable. The master in these experiments ran on the same Xeon E5-2670 system as in Table 1.

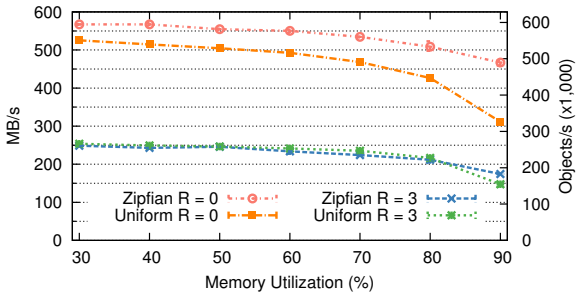

Figure 11: Two-level cleaning with (R = 3) and without replication (R = 0) for 1000-byte objects. The two lower curves are the same as in Figure 6.

Allocator	Fixed 25-byte	Zipfian 0 - 8 KB
Slab	8737	982
Log	11411	1125
Improvement	30.6%	14.6%

Table 3: Average number of objects stored per megabyte of cache in memcached, with its normal slab allocator and with a log-structured allocator. The "Fixed" column shows savings from reduced metadata (there is no fragmentation, since the 25-byte objects fit perfectly in one of the slab allocator's buckets). The "Zipfian" column shows savings from eliminating internal fragmentation in buckets. All experiments ran on a 16-core E5-2670 system with both client and server on the same machine to minimize network overhead. Memcached was given 2 GB of slab or log space for storing objects, and the slab rebalancer was enabled. YCSB [15] was used to generate the access patterns. Each run wrote 100 million objects with Zipfian-distributed key popularity and either fixed 25-byte or Zipfian-distributed sizes between 0 and 8 KB. Results were averaged over 5 runs.

experiments both with larger pools of backups and with replication disabled (not depicted) suggest that these delays are primarily due to contention for the NIC and RPC queueing delays in the single-threaded backup servers.

8.5 Performance Under Changing Workloads

Section 2 showed that changing workloads caused poor memory utilization in traditional storage allocators. For comparison, we ran those same workloads on RAMCloud, using the same general setup as for earlier experiments. The results are shown in Figure 10 (this figure is formatted differently than Figure 1 in order to show RAMCloud's performance as a function of memory utilization). We expected these workloads to exhibit performance similar to the workloads in Figure 6 (i.e. we expected the performance to be determined by the average object sizes and access patterns; workload changes per se should have no impact). Figure 10 confirms this hypothesis: with the high-stress request pattern, performance degradation due to cleaning was 10-20% at 70% utilization and 40-50% at 90% utilization. With the "Sequential" request pattern, performance degradation was 5% or less, even at 90% utilization.

8.6 Other Uses for Log-Structured Memory

Our implementation of log-structured memory is tied to RAMCloud's distributed replication mechanism, but we believe that log-structured memory also makes sense in other environments. To demonstrate this, we performed two additional experiments.

First, we re-ran some of the experiments from Figure 6 with replication disabled in order to simulate a DRAM-only storage system. We also disabled compaction (since there is no backup I/O to conserve) and had the server run the combined cleaner on in-memory segments only. Figure 11 shows that without replication, log-structured memory supports significantly higher throughput: RAMCloud's single writer thread scales to nearly 600K 1,000-byte operations per second. Under very high memory pressure throughput drops by 20-50% depending on access locality. At this object size, one writer thread and two cleaner threads suffice to handle between one quarter and one half of a 10 gigabit Ethernet link's worth of write requests.

Second, we modified the popular memcached [2] 1.4.15 object caching server to use RAMCloud's log and cleaner instead of its slab allocator. To make room for new cache entries, we modified the log cleaner to evict cold objects as it cleaned, rather than using memcached's slab-based LRU lists. Our policy was simple: segments

Allocator	Throughput (Writes/s x1000)	% CPU Cleaning
Slab	259.9 ± 0.6	0%
Log	268.0 ± 0.6	5.37 ± 0.3 %

Table 4: Average throughput and percentage of CPU used for cleaning under the same Zipfian write-only workload as in Table 3. Results were averaged over 5 runs.

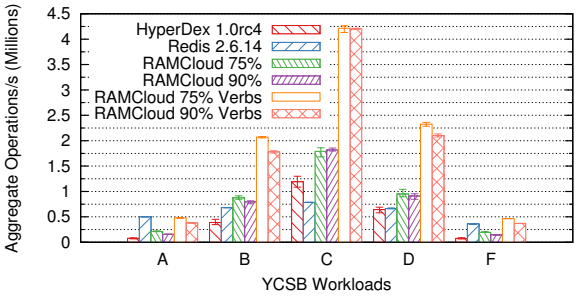

Figure 12: Performance of HyperDex, RAMCloud, and Redis under the default YCSB [15] workloads B, C, and D are read-heavy workloads, while A and F are write-heavy; workload E was omitted because RAMCloud does not support scans. Y-values represent aggregate average throughput of 24 YCSB clients running on 24 separate nodes (see Table 2). Each client performed 100 million operations on a data set of 100 million keys. Objects were 1 KB each (the workload default). An additional 12 nodes ran the storage servers. HyperDex and Redis used kernel-level sockets over Infiniband. The "RAMCloud 75%" and "RAMCloud 90%" bars were measured with kernel-level sockets over Infiniband at 75% and 90% memory utilisation, respectively (each server's share of the 10 million total records corresponded to 75% or 90% of log memory). The "RAMCloud 75% Verbs" and "RAMCloud 90% Verbs" bars were measured with RAMCloud's "kernel bypass" user-level Infiniband transport layer, which uses reliably-connected queue pairs via the Infiniband "Verbs" API. Each data point is averaged over 3 runs.

were selected for cleaning based on how many recent reads were made to objects in them (fewer requests indicate colder segments). After selecting segments, 75% of their most recently accessed objects were written to survivor segments (in order of access time); the rest were discarded. Porting the log to memcached was straightforward, requiring only minor changes to the RAMCloud sources and about 350 lines of changes to memcached.

Table 3 illustrates the main benefit of log-structured memory in memcached: increased memory efficiency. By using a log we were able to reduce per-object metadata overheads by 50% (primarily by eliminating LRU list pointers, like MemC3 [20]). This meant that small objects could be stored much more efficiently. Furthermore, using a log reduced internal fragmentation: the slab allocator must pick one of several fixed-size buckets for each object, whereas the log can pack objects of different sizes into a single segment. Table 4 shows that these benefits also came with no loss in throughput and only minimal cleaning overhead.

8.7 How does RAMCloud compare to other systems?

Figure 12 compares the performance of RAMCloud to HyperDex [18] and Redis [3] using the YCSB [15] benchmark suite. All systems were configured with triple replication. Since HyperDex is a disk-based store, we configured it to use a RAM-based file system to ensure that no operations were limited by disk I/O latencies, which the other systems specifically avoid. Both RAMCloud and Redis wrote to SSDs (Redis' append-only logging mechanism was used with a 1s fsync interval). It is worth noting that Redis is distributed with jemalloc [19], whose fragmentation issues we explored in Section 2.

RAMCloud outperforms HyperDex in every case, even when running at very high memory utilization and despite configuring HyperDex so that it does not write to disks. RAMCloud also outperforms Redis, except in write-dominated workloads A and F when kernel sockets are used. In these cases RAMCloud is limited by RPC latency, rather than allocation speed. In particular, RAMCloud must wait until data is replicated to all backups before replying to a client's write request. Redis, on the other hand, offers no durability guarantee; it responds immediately and batches updates to replicas. This unsafe mode of operation means that Redis is much less reliant on RPC latency for throughput.

Unlike the other two systems, RAMCloud was optimized for high-performance networking. For fairness, the "RAMCloud 75%" and "RAMCloud 90%" bars depict performance using the same kernel-level sockets as Redis and HyperDex. To show RAMCloud's full potential, however, we also included measurements using the Infiniband "Verbs" API, which permits low-latency access to the network card without going through the kernel. This is the normal transport used in RAMCloud; it more than doubles read throughput, and matches Redis' write throughput at 75% memory utilisation (RAMCloud is 25% slower than Redis for workload A at 90% utilization). Since Redis is less reliant on latency for performance, we do not expect it to benefit substantially if ported to use the Verbs API.

9 LFS Cost-Benefit Revisited

Like LFS [32], RAMCloud's combined cleaner uses a cost-benefit policy to choose which segments to clean. However, while evaluating cleaning techniques for RAMCloud we discovered a significant flaw in the original LFS policy for segment selection. A small change to the formula for segment selection fixes this flaw and improves cleaner performance by 50% or more at high utilization under a wide range of access localities (e.g., the Zipfian and uniform access patterns in Section 8.1). This improvement applies to any implementation of log-structured storage.

LFS selected segments to clean by evaluating the following formula for each segment and choosing the segments with the highest ratios of benefit to cost:

$$\frac{benefit}{cost} = \frac{(1-u) \times objectAge}{1+u}$$

In this formula, u is the segment's utilization (fraction of data still live), and $objectAge$ is the age of the youngest data in the segment. The cost of cleaning a segment is

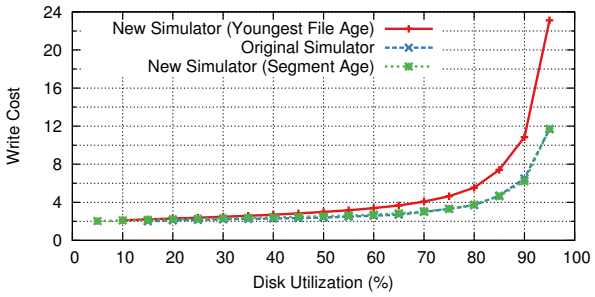

Figure 13: An original LFS simulation from [31]'s Figure 5-6 compared to results from our reimplemented simulator. The graph depicts how the I/O overhead of cleaning under a particular synthetic workload (see [31] for details) increases with disk utilization. Only by using segment age were we able to reproduce the original results (note that the bottom two lines coincide).

determined by the number of bytes that must be read or written from disk (the entire segment must be read, then the live bytes must be rewritten). The benefit of cleaning includes two factors: the amount of free space that will be reclaimed $(1 - u)$, and an additional factor intended to represent the stability of the data. If data in a segment is being overwritten rapidly then it is better to delay cleaning so that u will drop; if data in a segment is stable, it makes more sense to reclaim the free space now. $objectAge$ was used as an approximation for stability. LFS showed that cleaning can be made much more efficient by taking all these factors into account.

RAMCloud uses a slightly different formula for segment selection:

$$\frac{benefit}{cost} = \frac{(1-u) \times segmentAge}{u}$$

This differs from LFS in two ways. First, the cost has changed from $1 + u$ to u. This reflects the fact that RAMCloud keeps live segment contents in memory at all times, so the only cleaning cost is for rewriting live data.

The second change to RAMCloud's segment selection formula is in the way that data stability is estimated; this has a significant impact on cleaner performance. Using object age produces pathological cleaning behavior when there are very old objects. Eventually, some segments' objects become old enough to force the policy into cleaning the segments at extremely high utilization, which is very inefficient. Moreover, since live data is written to survivor segments in age-order (to segregate hot and cold data and make future cleaning more efficient), a vicious cycle ensues because the cleaner generates new segments with similarly high ages. These segments are then cleaned at high utilization, producing new survivors with high ages, and so on. In general, object age is not a reliable estimator of stability. For example, if objects are deleted uniform-randomly, then an objects's age provides no indication of how long it may persist.

To fix this problem, RAMCloud uses the age of the *segment*, not the age of its objects, in the formula for segment selection. This provides a better approximation to the stability of the segment's data: if a segment is very old, then its overall rate of decay must be low, otherwise its u-value would have dropped to the point of it being selected for cleaning. Furthermore, this age metric resets when a segment is cleaned, which prevents very old ages from accumulating. Figure 13 shows that this change improves overall write performance by 70% at 90% disk utilization. This improvement applies not just to RAMCloud, but to any log-structured system.

Intriguingly, although Sprite LFS used youngest object age in its cost-benefit formula, we believe that the LFS simulator, which was originally used to develop the cost-benefit policy, inadvertently used segment age instead. We reached this conclusion when we attempted to reproduce the original LFS simulation results and failed. Our initial simulation results were much worse than those reported for LFS (see Figure 13); when we switched from $objectAge$ to $segmentAge$, our simulations matched those for LFS exactly. Further evidence can be found in [26], which was based on a descendant of the original LFS simulator and describes the LFS cost-benefit policy as using the segment's age. Unfortunately, source code is no longer available for either of these simulators.

10 Future Work

There are additional opportunities to improve the performance of log-structured memory that we have not yet explored. One approach that has been used in many other storage systems is to compress the data being stored. This would allow memory to be used even more efficiently, but it would create additional CPU overheads both for reading and writing objects. Another possibility is to take advantage of periods of low load (in the middle of the night, for example) to clean aggressively in order to generate as much free space as possible; this could potentially reduce the cleaning overheads during periods of higher load.

Many of our experiments focused on worst-case synthetic scenarios (for example, heavy write loads at very high memory utilization, simple object size distributions and access patterns, etc.). In doing so we wanted to stress the system as much as possible to understand its limits. However, realistic workloads may be much less demanding. When RAMCloud begins to be deployed and used we hope to learn much more about its performance under real-world access patterns.

11 Related Work

DRAM has long been used to improve performance in main-memory database systems [17, 21], and large-scale Web applications have rekindled interest in DRAM-based storage in recent years. In addition to special-purpose systems like Web search engines [9], general-purpose storage systems like H-Store [25] and Bigtable [12] also keep part or all of their data in memory to maximize performance.

RAMCloud's storage management is superficially sim-

ilar to Bigtable [12] and its related LevelDB [4] library. For example, writes to Bigtable are first logged to GFS [22] and then stored in a DRAM buffer. Bigtable has several different mechanisms referred to as "compactions", which flush the DRAM buffer to a GFS file when it grows too large, reduce the number of files on disk, and reclaim space used by "delete entries" (analogous to tombstones in RAMCloud and called "deletion markers" in LevelDB). Unlike RAMCloud, the purpose of these compactions is not to reduce backup I/O, nor is it clear that these design choices improve memory efficiency. Bigtable does not incrementally remove delete entries from tables; instead it must rewrite them entirely. LevelDB's generational garbage collection mechanism [5], however, is more similar to RAMCloud's segmented log and cleaning.

Cleaning in log-structured memory serves a function similar to copying garbage collectors in many common programming languages such as Java and LISP [24, 37]. Section 2 has already discussed these systems.

Log-structured memory in RAMCloud was influenced by ideas introduced in log-structured file systems [32]. Much of the nomenclature and general techniques are shared (log segmentation, cleaning, and cost-benefit selection, for example). However, RAMCloud differs in its design and application. The key-value data model, for instance, allows RAMCloud to use simpler metadata structures than LFS. Furthermore, as a cluster system, RAMCloud has many disks at its disposal, which reduces contention between cleaning and regular log appends.

Efficiency has been a controversial topic in log-structured file systems [34, 35]. Additional techniques were introduced to reduce or hide the cost of cleaning [11, 26]. However, as an in-memory store, RAMCloud's use of a log is more efficient than LFS. First, RAMCloud need not read segments from disk during cleaning, which reduces cleaner I/O. Second, RAMCloud may run its disks at low utilization, making disk cleaning much cheaper with two-level cleaning. Third, since reads are always serviced from DRAM they are always fast, regardless of locality of access or placement in the log.

RAMCloud's data model and use of DRAM as the location of record for all data are similar to various "NoSQL" storage systems. Redis [3] is an in-memory store that supports a "persistence log" for durability, but does not do cleaning to reclaim free space, and offers weak durability guarantees. Memcached [2] stores all data in DRAM, but it is a volatile cache with no durability. Other NoSQL systems like Dynamo [16] and PNUTS [14] also have simplified data models, but do not service all reads from memory. HyperDex [18] offers similar durability and consistency to RAMCloud, but is a disk-based system and supports a richer data model, including range scans and efficient searches across multiple columns.

12 Conclusion

Logging has been used for decades to ensure durability and consistency in storage systems. When we began designing RAMCloud, it was a natural choice to use a logging approach on disk to back up the data stored in main memory. However, it was surprising to discover that logging also makes sense as a technique for managing the data in DRAM. Log-structured memory takes advantage of the restricted use of pointers in storage systems to eliminate the global memory scans that fundamentally limit existing garbage collectors. The result is an efficient and highly incremental form of copying garbage collector that allows memory to be used efficiently even at utilizations of 80-90%. A pleasant side effect of this discovery was that we were able to use a single technique for managing both disk and main memory, with small policy differences that optimize the usage of each medium.

Although we developed log-structured memory for RAMCloud, we believe that the ideas are generally applicable and that log-structured memory is a good candidate for managing memory in DRAM-based storage systems.

13 Acknowledgements

We would like to thank Asaf Cidon, Satoshi Matsushita, Diego Ongaro, Henry Qin, Mendel Rosenblum, Ryan Stutsman, Stephen Yang, the anonymous reviewers from FAST 2013, SOSP 2013, and FAST 2014, and our shepherd, Randy Katz, for their helpful comments. This work was supported in part by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA, and by the National Science Foundation under Grant No. 0963859. Additional support was provided by Stanford Experimental Data Center Laboratory affiliates Facebook, Mellanox, NEC, Cisco, Emulex, NetApp, SAP, Inventec, Google, VMware, and Samsung. Steve Rumble was supported by a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship.

References

[1] Google performance tools, Mar. 2013. http://goog-perftools.sourceforge.net/.

[2] memcached: a distributed memory object caching system, Mar. 2013. http://www.memcached.org/.

[3] Redis, Mar. 2013. http://www.redis.io/.

[4] leveldb - a fast and lightweight key/value database library by google, Jan. 2014. http://code.google.com/p/leveldb/.

[5] Leveldb file layouts and compactions, Jan. 2014. http://leveldb.googlecode.com/svn/trunk/doc/impl.html.

[6] APPAVOO, J., HUI, K., SOULES, C. A. N., WISNIEWSKI, R. W., DA SILVA, D. M., KRIEGER, O., AUSLANDER, M. A., EDEL-

SOHN, D. J., GAMSA, B., GANGER, G. R., MCKENNEY, P., OSTROWSKI, M., ROSENBURG, B., STUMM, M., AND XENIDIS, J. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J. 42*, 1 (Jan. 2003), 60–76.

[7] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[8] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), POPL '03, ACM, pp. 285–298.

[9] BARROSO, L. A., DEAN, J., AND HÖLZLE, U. Web search for a planet: The google cluster architecture. *IEEE Micro 23*, 2 (Mar. 2003), 22–28.

[10] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 117–128.

[11] BLACKWELL, T., HARRIS, J., AND SELTZER, M. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference* (Berkeley, CA, USA, 1995), TCON'95, USENIX Association, pp. 277–288.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 205–218.

[13] CHENG, P., AND BLELLOCH, G. E. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 125–136.

[14] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1* (August 2008), 1277–1288.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[17] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data* (New York, NY, USA, 1984), SIGMOD '84, ACM, pp. 1–8.

[18] ESCRIVA, R., WONG, B., AND SIRER, E. G. Hyperdex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 25–36.

[19] EVANS, J. A scalable concurrent malloc (3) implementation for freebsd. In *Proceedings of the BSDCan Conference* (Apr. 2006).

[20] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Memc3: compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 371–384.

[21] GARCIA-MOLINA, H., AND SALEM, K. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng. 4* (December 1992), 509–516.

[22] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[23] HERTZ, M., AND BERGER, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 313–326.

[24] JONES, R., HOSKING, A., AND MOSS, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Chapman & Hall/CRC, 2011.

[25] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow. 1* (August 2008), 1496–1499.

[26] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev. 31*, 5 (Oct. 1997), 238–251.

[27] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, Oct. 1998), pp. 509–518.

[28] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 385–398.

[29] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[30] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Commun. ACM 54* (July 2011), 121–130.

[31] ROSENBLUM, M. *The design and implementation of a log-structured file system*. PhD thesis, Berkeley, CA, USA, 1992. UMI Order No. GAX93-30713.

[32] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10* (February 1992), 26–52.

[33] RUMBLE, S. M. *Memory and Object Management in RAMCloud*. PhD thesis, Stanford, CA, USA, 2014.

[34] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for unix. In *Proceedings of the 1993 Winter USENIX Technical Conference* (Berkeley, CA, USA, 1993), USENIX'93, USENIX Association, pp. 307–326.

[35] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference* (Berkeley, CA, USA, 1995), TCON'95, USENIX Association, pp. 249–264.

[36] TENE, G., IYENGAR, B., AND WOLF, M. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management* (New York, NY, USA, 2011), ISMM '11, ACM, pp. 79–88.

[37] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (London, UK, UK, 1992), IWMM '92, Springer-Verlag, pp. 1–42.

[38] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association.

[39] ZORN, B. The measured cost of conservative garbage collection. *Softw. Pract. Exper. 23*, 7 (July 1993), 733–756.