Log-structured Memory for DRAM-based Storage

Paper #147 (14 Pages)

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 2-8x 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 [23], and Spark [29]. 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. In comparison to traditional 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 [24], but with several novel aspects:

- In contrast to a log-structured file system, 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 the utilization of DRAM while minimizing bandwidth requirements for network and disk.
- 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 230,000-400,000 durable 100-byte writes per second at 90% memory utilization. The two-level approach to cleaning improves performance by 2-8x over a single-level approach at high memory utilization, and reduces disk bandwidth overhead by 2-20x. Parallel cleaning effectively hides the cost of cleaning: an active cleaner adds only about 3% 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 applications, but they are more common 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

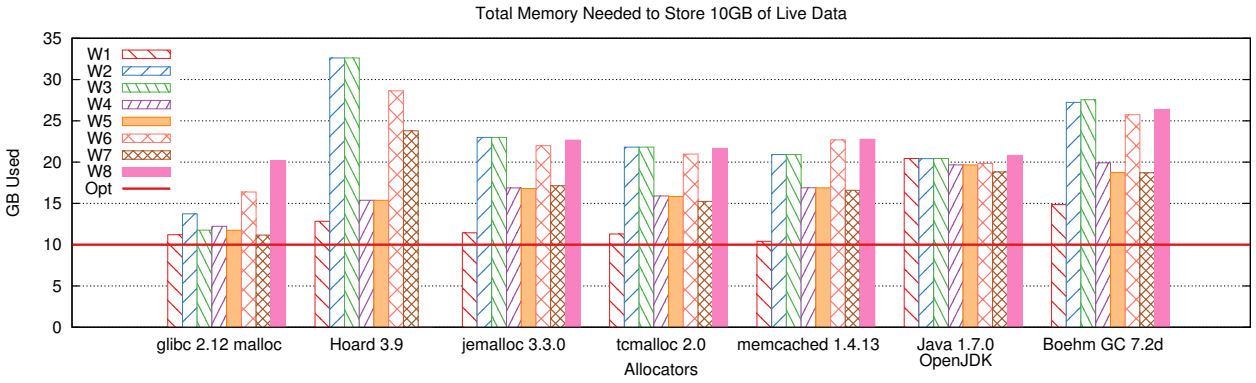

Figure 1: Total memory needed by allocators to support 10 GB of live data under the changing workloads described in Table 1. "Opt" indicates the amount of live data, and represents an optimal result. "glibc" is the allocator typically used by C and C++ applications on Linux. "Hoard" [7], "jemalloc" [14], 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 16GB 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 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.

records of a table are expanded from 100 bytes to 130 bytes. Facebook encountered distribution changes like this in its memcached storage systems and had to augment their cache eviction strategies with special-purpose code to cope with such eventualities [21]. Non-copying allocators may work well in many situations, 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 [30, 16]. 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 [5, 10, 27] – 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 [22]). We experimented with features of the JVM collectors that reduce 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 to use a log-structured approach to memory management to achieve 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 must 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. The 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

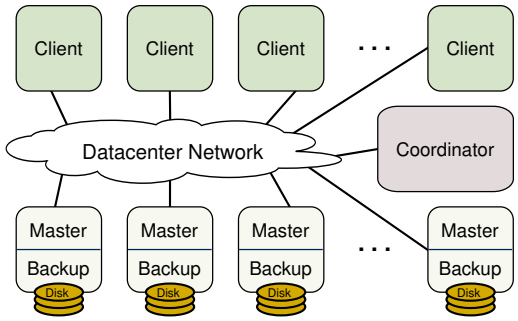

Figure 2: RAMCloud cluster architecture. Each storage server contains a master, which manages objects in its DRAM, and a backup, which stores segment replicas on disk or flash. A central coordinator manages the cluster configuration. Client applications access RAMCloud with remote procedure calls.

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; a subset of a table stored on a single server is called a *tablet*. 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 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. This means that a single master can harness the I/O bandwidth of many backups to handle bursts of writes, and it also makes the entire I/O bandwidth of the cluster available for reading a master's segment replicas during crash recovery.

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. 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 re-

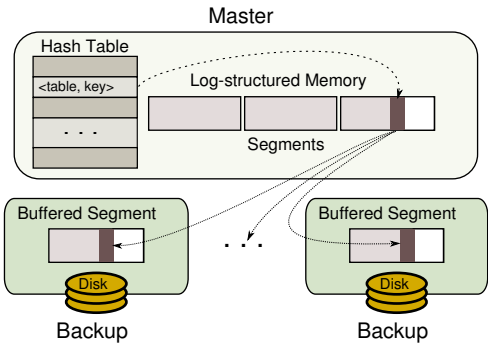

Figure 3: Master servers consist primarily of a hash table and an in-memory log, which is replicated across several backups for durability. New objects are appended to the head log segment and are synchronously replicated to non-volatile staging buffers on backups. Client writes are acknowledged once all backups have buffered the new addition to the head segment.

allocates the buffer for another segment. This approach has two performance advantages: it allows writes to complete without waiting for I/O to secondary storage, and it uses secondary storage bandwidth efficiently by performing I/O in large blocks, even if individual 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. When a master crashes, one replica for each of its segments is read from secondary storage; the log is then replayed on other servers to reconstruct the data that had been stored on the crashed master. For details on RAMCloud crash recovery, see [22]. 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 [24]:

- 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; this reduces the I/O overhead for cleaning by more than a factor of 2.
- The cleaner makes the old segments' memory available for new segments, and it notifies the backups for those segments that they can reclaim the storage for the replicas.

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 [25, 26]. 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 a log-structured file system, the log must contain a considerable amount 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, and the on-disk log is never read during normal use. The on-disk log 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, allocated in ascending order within a log. The log digest is a list of identifiers for all the segments that currently belong to the log, including the current segment. Log digests avoid the need for a central repository of log information (such a repository would create a scalability bottleneck and introduce crash recovery problems of its own). When recovering a crashed master, the coordinator communicates with all backups to find out which segments they hold for the dead master. This information, combined with the digest from the latest segment, allows RAMCloud to find all of the live segments in a log

(see [22] 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. The first 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 by the cleaner.

Tombstones have other drawbacks that will be discussed in later sections. Given the issues with tombstones, we have occasionally wondered whether some other approach would provide a better solution to the liveness problem. LFS used the metadata in its log to determine liveness: for example, an inode was still live if the inode map had an entry referring to it. We considered alternative approaches that use explicit metadata to keep track of live objects, but these approaches were complex and created performance issues. We eventually decided that tombstones are the best of the available alternatives.

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 approaches 100% (see Figure 4). For example, if segments are cleaned when 80% of their data are still live, the cleaner must copy four bytes of live data for every byte it frees. If segments are cleaned at 90% utilization, the cleaner must copy 9 bytes of live data for every byte freed. As memory utilization rises, eventually the system will run out of bandwidth and write throughput will

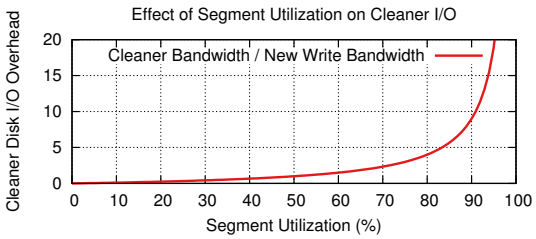

Figure 4: The amount of disk I/O used by the cleaner for every byte of reclaimed space increases sharply as segments are cleaned at higher utilization.

be limited by the speed of cleaner. Techniques like cost-benefit segment selection [24] 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 memory utilization and cleaning cost. Any copying storage allocator will suffer from intolerable overheads as memory utilization approaches 100%.

In the original implementation of RAMCloud, disk and memory cleaning were tied together: cleaning was first performed on segments in memory, then the results were reflected to the backup copies on disk. Unfortunately, this made it impossible to achieve both high memory utilization and high write throughput. Given that DRAM represents about half the cost of a RAMCloud system, we wanted to use 80-90% of memory for live data. But, at this utilization the write bandwidth to disk created a severe limit on write throughput (see Section 8 for measurements).

On the other hand, we could have improved write bandwidth by adding more disk space 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 in-memory and on-disk logs independently – we call this approach *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 re-

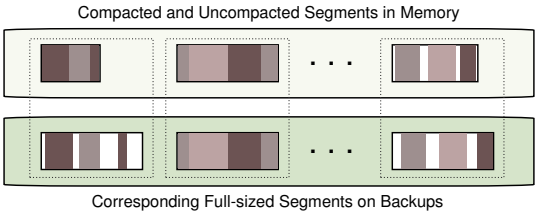

Figure 5: Compacted segments in memory have variable length because unneeded objects and tombstones have been removed, but the corresponding segments on disk remain full-size. As a result, the utilization of memory is higher than that of disk, and disk can be cleaned more efficiently.

gion 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 have been removed (see Figure 5).

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.

Two-level cleaning combines 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 memory into fixed-size 64KB *seglets*. A segment consists of a collection of seglets, and the number of seglets varies with the size of the segment.

Seglets create two potential problems. The first problem is fragmentation: since seglets are fixed-size but segments are variable-size, on average there will be one-half seglet of wasted space at the end of each compacted segment. Fortunately, with a 64KB seglet size this fragmentation represents less than 1% of memory space (in practice we expect compacted segments to average at least half the length of a full-size segment).

The second problem with seglets is that individual objects or tombstones in the log can now span a seglet boundary, and this complicates access to them (objects and tombstones are not allowed to cross segment boundaries, so prior to the introduction of seglets they were always contiguous in memory). We solved this problem by introducing an additional layer of software that hides the discontiguous nature of segments. RAMCloud's requirements for low latency make the addition of a new layer cause for concern (e.g. our goal is for servers to handle simple requests end-to-end in less than 1μs). Fortunately, we were able to find an API for this layer that is relatively efficient in the common case where objects do not cross seglet boundaries.

Our first implementation of segments used *mmap* to map discontiguous seglets into a contiguous range of virtual addresses, thereby avoiding the need for an extra software layer. Unfortunately, this approach introduced several problems. For example, manipulating memory mappings from user space required expensive system calls, and constantly setting up and tearing down the mappings during cleaning added overhead for TLB misses and shootdowns. As a result, we eventually abandoned the *mmap* approach in favor of an extra software layer.

5.2 When to Clean on Disk?

Two-level cleaning requires a policy to determine when to run each cleaner. We call this policy the *balancer*. In order to understand the balancer design, it is useful to review three reasons why combined cleaning is still needed; these factors impacted the design of the balancer.

The first reason for combined cleaning is to limit the growth of the on-disk log. Although disk space is relatively inexpensive, the length of the on-disk log impacts crash recovery time, since the entire log must be read during recovery. Furthermore, once the disk log becomes 2-3x the length of the in-memory log, additional increases in disk log length provide diminishing benefit. Thus, RAMCloud implements a configurable *disk expansion factor* that determines the maximum size of the on-disk log as a multiple of the in-memory log size. We expect this value to be in the range of 2-3. The combined cleaner will begin cleaning as the on-disk log length approaches the limit, and if the length reaches the limit then no new segments may be allocated until the cleaner frees some existing segments.

The second reason for combined cleaning is to reclaim space occupied by tombstones. A tombstone cannot be dropped until the dead object it refers to has been removed from the on-disk log (i.e. the dead object can never be seen during crash recovery). Segment compaction removes dead objects from memory, but it does

not affect the on-disk copies of the objects. A tombstone must therefore be retained in memory until the on-disk segment has been cleaned. If the combined cleaner does not run, tombstones will accumulate in memory, increasing the effective memory utilization and making segment compaction increasingly expensive. Without combined cleaning, tombstones could eventually consume all of the free space in memory.

The third reason for combined cleaning is to enable segment reorganization. When the combined cleaner runs, it cleans several segments at once and reorganizes the data by age, so that younger objects are stored in different survivor segments than the older ones. If there is locality in the write workload, then the segments with older objects are less likely to be modified than the segments with younger objects, and cleaning costs can be reduced significantly. However, segment compaction cannot reorganize data, since it must preserve the one-to-one mapping between segments in memory and those on disk. We considered allowing segment compaction to reorganize data into new segments, but that would have created complex dependencies between the segments in memory and those on disk (e.g. which in-memory segments must be cleaned in order to free a particular set of on-disk segments?).

Given these issues, our current balancer operates as follows. The segment compactor runs when free seglets account for less than 10% of total memory space. The combined cleaner runs when free seglets account for less than 2% of total memory space, or when the on-disk log length is within 10% of the limit determined by the disk expansion factor. The reasoning behind this approach is that we want to delay combined cleaning and let the segment compactor run first in order to reduce the work of the combined cleaner. If the segment compactor is having trouble keeping up (i.e. free space is dwindling) or we are running out of disk space, then the combined cleaner should run.

6 Parallel Cleaning

Two-level cleaning reduces the cost of combined cleaning, but it adds a significant 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 architectures and scale the throughput of cleaning.

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 never needs to worry about objects being modified while the cleaner is copying them. Furthermore, since all objects are accessed indirectly through the hash table, it provides a simple way of redirecting references to the new log location of live objects that are relocated by the cleaner. 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 (see Figure 6). Once a cleaner thread has finished 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 log digest. This step requires additional synchronization, which is discussed in Section 6.4.

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. Cleaning will still compete with new writes for network bandwidth to backups, but we assume the availability of high-speed networks (e.g. 2-3 GB/s in our cluster), so network bandwidth will be much more plentiful than disk bandwidth.

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 ob-

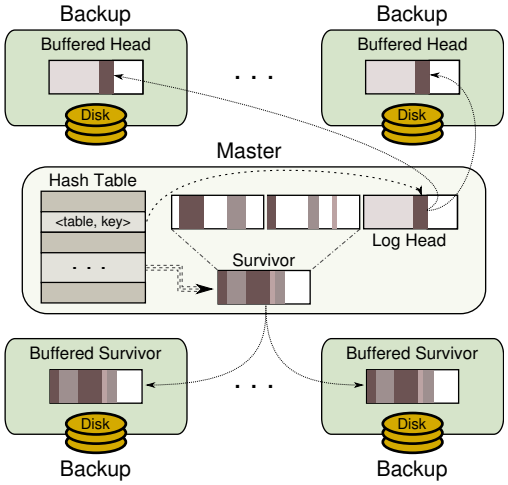

Figure 6: Master server during parallel cleaning. A write request is being appended to the head of the log (top) while the cleaner relocates the live data from a few segments into a new survivor segment (bottom). As the cleaner copies objects, it updates the hash table to point to their new locations in memory. The survivor segment is replicated to a different set of backups than the log head, avoiding disk I/O contention. It will become part of the durable log on disk when cleaning completes and a log digest is written to a new head segment.

jects 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. Although contention for these locks is low (only 0.1% under heavy write and cleaner load) it still means that the cleaner must acquire and release a lock for every object it scans, and read and write requests must also acquire an extra lock. 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 cannot use data in the cleaned segment, because there are no hash table entries pointing to that segment. However, it is possible that a service thread had begun 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.

A simple solution is to use a reader-writer lock for each segment, with service threads holding reader locks while using a segment and cleaner threads holding writer locks before freeing a segment. However, this would increase the latency of all read requests by adding another lock in the critical path.

Instead, RAMCloud uses a mechanism based on *epochs*, which avoids locking in service threads. The only additional overhead for service threads is to read a global epoch variable and store it with the RPC. When a cleaner thread finishes a cleaning pass, it increments the epoch and then tags each cleaned segment with the new epoch (after this point, no new request will use the cleaned segment). The cleaner occasionally scans the epochs of active RPCs and frees the segment when all RPCs with epochs less than the segment's epoch have completed. This approach creates additional overhead for segment freeing, but these operations are infrequent and run in a separate thread where they don't impact read and write times.

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.

The second step is for the survivor segments to be included in a new log digest and for the cleaned segments to be removed from the digest. The cleaner adds information about the survivor and cleaned segments to a queue. Whenever a new log head segment is created, the information in this queue is used to generate the digest in that segment. Once the digest has been durably written to backups, the cleaned segments will never be used during recovery, so RPCs are issued to free their backup replicas. The cleaner does not need to wait for the new digest to be written; it can start a new cleaning pass as soon as it has left information in the queue.

7 Avoiding Cleaner Deadlock

Since log cleaning copies data before freeing it, the cleaner must have some 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 minimize the amount of live data in segments (the longer it waits, the more objects may have been deleted). Finally, the two-level cleaning model 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 segments for the cleaner to use. This is accomplished by reserving a special pool of segments for the cleaner. When segments 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

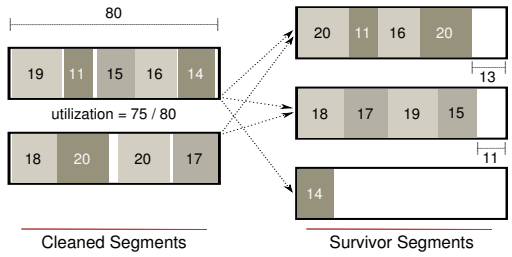

Figure 7: A simplified situation in which cleaning uses more space than it frees. Two 80-byte segments at about 94% utilization are cleaned: their objects are reordered by age (not depicted) and written to survivor segments. The label in each object indicates its size. Because of fragmentation, the last object (size 14) overflows into a third survivor segment.

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 7).

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 the empty except for a single byte, resulting in almost a full seglet of 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 segment cleaned. 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 (though at the expense of storing more objects discontiguously).

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.

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

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

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 segments' 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 all 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 space could be consumed with live data and there would be no way to add a tombstone to the log to delete objects.

8 Evaluation

All of the features described in the previous sections are implemented in the current version of RAMCloud. This section describes a series of experiments we ran to evaluate the performance of log-structured memory under sustained write loads. The experiments were performed by dividing an 80-node cluster of identical commodity servers (Table 2) into groups of five servers. Different groups were used 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 the conclusions of this section. In all experiments, every byte stored on a master was replicated to three different backups for durability.

All of our experiments used 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 tried experiments that used hyperthreading to expand the number of cleaner threads, but this produced worse overall performance.

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

1. **Object Size** In all experiments except those in Section 8.5, 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 have omitted 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. Different tests used values ranging from 30% to 90%. Lower utilizations are cheaper to clean because segments tend to be less full. In some experiments, actual memory utilization was significantly higher than these numbers due to an accumulation of tombstones.

3. **Locality** Experiments were run with both uniform random overwrites of objects and a simple "hot-and-cold" scheme in which 90% of writes were spread random-uniformly across 10% of the objects, with the remaining 10% of the writes spread across the other 90% of the objects. The uniform random case represents a workload with no locality; hot-and-cold represents a modest degree of locality.

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 performed tests where the client issued one synchronous request at a time, with a single write operation in each request; these tests are labeled "Single Client" in the graphs. We believe the Single Client tests are more indicative of actual RAMCloud workloads, but for most of our tests they did not stress the master (RPC request processing dominated memory management overheads). We do not yet know the workloads that will run on real RAMCloud systems, so we would like RAMCloud to support a wide range of access patterns.

8.1 Performance vs. Utilization

The most important metric for log-structured memory is how it performs at high utilization. All storage allocators perform worse as memory utilization approaches 100%, and this is particularly true for copying allocators such as log-structured memory. Our hope at the beginning of the project was that log-structured memory could support memory utilizations in the range of 80-90%.

Figure 8 graphs the overall throughput of a RAMCloud master with different memory utilizations and workloads. In each experiment, the client created 4 GB of data by writing objects with sequential keys; then it overwrote objects (maintaining 4 GB of live data continuously) until the overhead for cleaning converged to a stable value. We chose a 4 GB dataset size because it resulted in a relatively large number of segments while also keeping experiment times manageable (in production, we expect RAMCloud servers to be fitted with at least 64 GB of DRAM). The total memory usage varied with the utilization: e.g., for 50% utilization, the total memory pool for each master was 8 GB.

With two-level cleaning enabled, Figure 8 shows that 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 less than 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, so they must wait for the cleaner to free new segments.

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 400K small writes per second with some locality and nearly 250K with no locality. Both are well above our original target. In the "Single Client" curves, which are more indicative of actual RAMCloud workloads, 100B and 1000B objects show almost no performance degradation even at 90% memory utilization.

If actual RAMCloud workloads are similar to our "Single Client" case, then it should be reasonable to run RAMCloud clusters at 90% memory utilization. If workloads include many bulk writes, like most of the measurements in Figure 8, then it would probably make more sense to run at 80% utilization: the 12.5% additional cost for memory would be more than offset by increases in throughput.

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

8.2 Two-Level Cleaning

Figure 8 also demonstrates the benefits of two-level cleaning. In addition to measurements with two-level cleaning, the figure also contains measurements in which segment compaction was disabled ("one-level"); in these

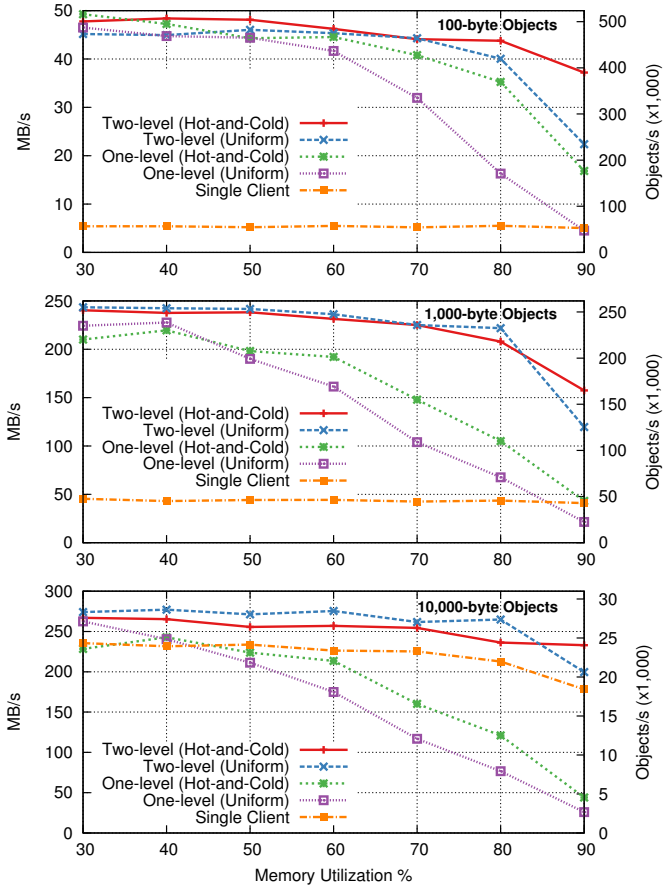

Figure 8: End-to-end client write performance as a function of memory utilization. For some experiments two-level cleaning was disabled, so only the combined cleaner was used. The "Single Client" curve used two-level cleaning and uniform access patterns with a single outstanding write request at a time. All other curves used the high-stress workload with concurrent multi-writes. Each point represents the average of three runs on different groups of servers.

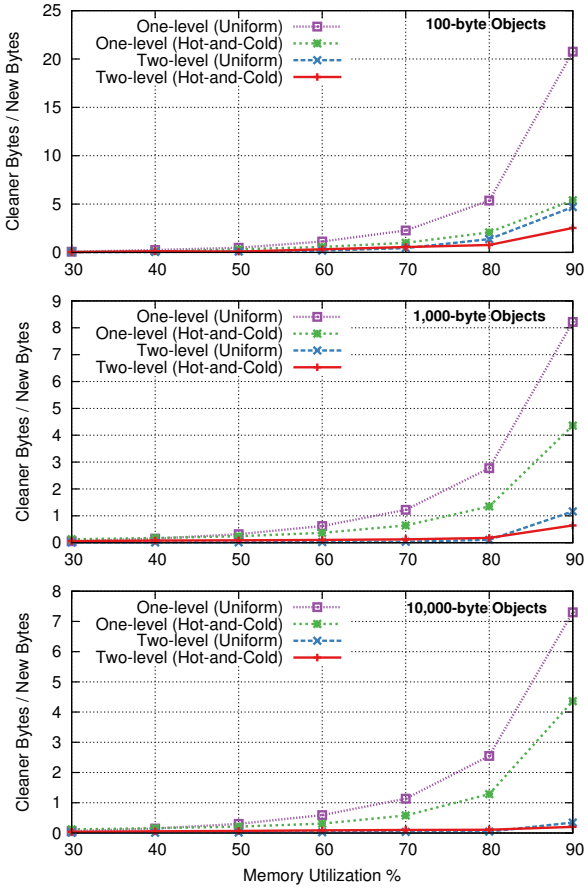

Figure 9: Cleaner bandwidth overhead (ratio of cleaner bandwidth to regular log write bandwidth) for the workloads in Figure 8. A ratio of 1 means that for every byte of new data written to backups, the cleaner writes 1 byte of live data to backups while freeing segment space. The optimal ratio is 0.

experiments, the system used a 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 2-8x 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 9 shows that two-level cleaning reduces disk and network bandwidth overheads by 2-20x at high memory utilizations for the workloads in Figure 8. The greatest benefits occur for workloads with larger object sizes and/or no locality.

8.3 Randomness vs. Locality

One surprising result in Figure 8 is that in several cases the hot-and-cold workload performed worse than a workload with no locality (see, for example, 10,000B objects with utilizations of 30-80%). We initially assumed this must be a mistake: surely the cost-benefit cleaning policy could take advantage of locality to clean more efficiently? After considerable additional analysis we concluded that the figure is correct, as described below. The interesting overall conclusion is that randomness can sometimes be better than locality at creating efficiencies, especially in large-scale systems.

Cleaning works best when free space is distributed unevenly across segments. In the ideal case, some segments will be totally empty while others are totally full; in this case, the cleaner can ignore the full segments and clean the empty ones at very low cost.

Locality represents one opportunity for creating an uneven distribution of free space. The cost-benefit approach to cleaning tends to collect the slowly changing data in separate segments where free space builds up slowly. Newly written segments contain mostly hot information that will be overwritten quickly, so free space grows rapidly in these segments. As a result, the cleaner can usually find segments to clean that have significantly

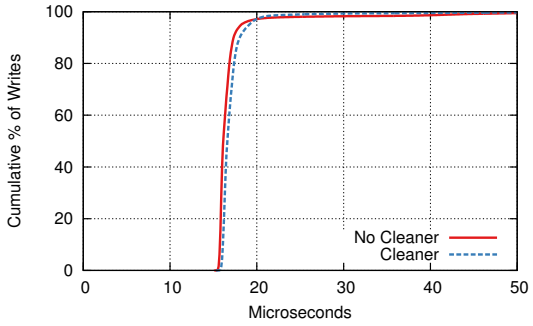

Figure 10: Cumulative distribution of client write latencies when a single client issues back-to-back write requests for 100-byte objects using the uniform distribution. For example, about 80% of all write requests completed in 17.3μs or less. The "No cleaner" curve was measured with cleaning disabled. The "Cleaner" curve shows write latencies at 90% memory utilization with cleaning enabled. The median latency with cleaning enabled was about 500ns higher than without a cleaner. Each curve contains tens of millions of samples.

more free space than the overall average.

However, randomness also creates an uneven distribution of free space. In a workload where overwrites are randomly distributed, some segments will accumulate more free space than others. As the number of segments increases, outlier segments with very low utilization become more likely. In Figure 8, the random behavior of the "uniform" workload was often more successful at generating very-low-utilization segments than the workload with locality. As the scale of the system increases, the benefits of randomness should increase.

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 10 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 cumulative distributions are nearly identical: cleaning added about 500ns to the median latency, or about 3%. We hypothesize that the increased latency is due to cache and memory bandwidth interference from the cleaner: at 90% utilization the segment compactor is running continuously, modifying hash table entries and accessing shared locks while processing nearly 4 million objects and tombstones per second.

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. However, in order to handle the large datasets we had to use a different server for the master; in these experiments the master was a Xeon E5-2670 system with 384 GB of DRAM running Linux 2.6.32.

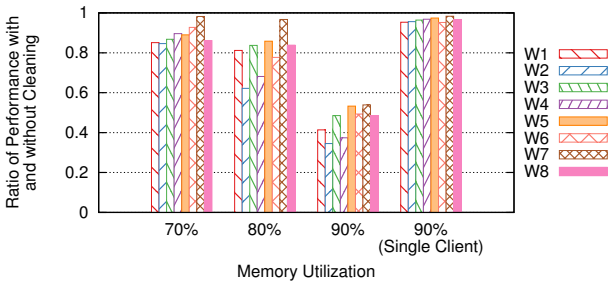

Figure 11: Client performance in RAMCloud under the same workloads as in Figure 1 from Section 2. Each bar measures the performance of a workload (with cleaning enabled) relative to the performance of the same workload with cleaning disabled. Higher is better and 1.0 is optimal; it means that the cleaner has no impact on the processing of normal requests. As in Figure 1, 100GB of allocations were made and at most 10GB 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 "Single Client" 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.

We expected these workloads to exhibit performance similar to the workloads in Figure 8 (i.e. we expected the performance to be determined by the object sizes and access patterns; workload changes per se should have no impact). Figure 11 confirms this hypothesis: with the high-stress request pattern, performance degradation due to cleaning was 10-20% at 70% utilization and 45-65% at 90% utilization. With the more representative "Single Client" request pattern, performance degradation was less than 10% even at 90% utilization.

9 Related Work

DRAM has long been used to improve performance in main-memory database systems [13, 15], and large-scale Web applications have greatly rekindled interest in DRAM-based storage in recent years. In addition to special-purpose systems like Web search engines [6], general-purpose storage systems like H-Store [18] and Bigtable [9] also keep part or all of their data in memory to maximize performance. RAMCloud is also similar to Bigtable in that both use a log-structured approach to durably storing data.

Cleaning in log-structured memory serves a function similar to copying garbage collectors in many common programming languages such as Java and LISP [17, 28]. However, log-structured memory was designed for storage systems that make restricted use of pointers. This greatly increases efficiency since only one pointer must be checked to determine liveness, and one pointer must be changed to relocate data. Reducing the scope of the problem to DRAM-based storage systems avoids many of the complexities and overheads inherent in language-based collectors.

The log-structured memory design in RAMCloud was

heavily influenced by ideas introduced in log-structured filesystems [24], borrowing nomenclature, structure, and techniques such as the use of segments, cleaning, and cost-benefit selection. However, RAMCloud differs considerably in its structure and application. The key-value data model, for example, allows RAMCloud to use much simpler metadata structures than LFS. Furthermore, as a cluster system, RAMCloud has many disks at its disposal, allowing parallel cleaning to reduce contention with regular log appends for backup I/O.

Efficiency has been a controversial topic in log-structured filesystems [25, 26], with additional techniques having been introduced to reduce or hide the cost of cleaning [8, 19]. 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 does not need to run its disks at high 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 data 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] supports a "persistence log" for durability, but does not do cleaning to reclaim free space. Instead, it writes all live data into a second log in the background and garbage collects the first log when done. Memcached [2] stores all data in DRAM, but is only a cache; it does not provide any durability guarantees. Other NoSQL systems like Dynamo [12] and PNUTS [11] also have simplified data models, but do not service all reads from memory.

Finally, RAMCloud's epoch-based mechanism for safely reclaiming cleaned segment memory is similar to Tornado/K42's *generations* [4] and RCU's [20] *wait-for-readers* primitive.

10 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 backup 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 any DRAM-based storage system.

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

[5] 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.

[6] 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.

[7] 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.

[8] BLACKWELL, T., HARRIS, J., AND SELTZER, M. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (Berkeley, CA, USA, 1995), TCON'95, USENIX Association, pp. 23–23.

[9] 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 USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Berkeley, CA, USA, 2006), USENIX Association, pp. 15–15.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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 *To appear in the Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013).

[22] 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.

[23] 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.

[24] 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.

[25] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for unix. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (Berkeley, CA, USA, 1993), USENIX'93, USENIX Association, pp. 3–3.

[26] 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 Proceedings* (Berkeley, CA, USA, 1995), TCON'95, USENIX Association, pp. 21–21.

[27] 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.

[28] 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.

[29] 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* (2011).

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