MEMORY AND OBJECT MANAGEMENT IN RAMCLOUD

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stephen Mathew Rumble
March 2014

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

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 theory, copying garbage collectors can deal with workload changes by defragmenting memory, but in practice the general-purpose collectors used in programming languages perform poorly at high memory utilisations. The result is that DRAM-based storage systems must choose between making efficient use of expensive memory and attaining high write performance.

This dissertation presents an alternative, describing how a log-structured approach to memory management – called *log-structured memory* – allows in-memory storage systems to achieve 80-90% memory utilisation while still offering high performance. One of the key observations of this approach is that memory can be managed much more efficiently than traditional garbage collectors by exploiting the restricted use of pointers in storage systems. An additional benefit of this technique is that the same mechanisms can be used to manage backup copies of data on disk for durability, allowing the system to survive crashes and restarts.

This dissertation describes and evaluates log-structured memory in the context of RAMCloud, a distributed storage system we built to provide low-latency access (5μs) to very large data sets (petabytes or more) in a single datacentre. RAMCloud 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 novel two-level cleaning policy, which reduces disk and network bandwidth overheads by up to 76x and improves write throughput up to 5.5x at high memory utilisations. The cleaner runs concurrently with normal operations, employs multiple threads to hide most of the cost of cleaning, and unlike traditional garbage collectors is completely pauseless.

Preface

The version of RAMCloud used in this work corresponds to git repository commit f52d5bcc014c (December 7th, 2013). Please visit http://ramcloud.stanford.edu for information on how to clone the repository.

Acknowledgements

Thanks to John, my dissertation advisor, for spearheading a project of the scope and excitement of RAMCloud, for setting the pace and letting us students try to keep up with him, and for advising us so diligently along the way. He has constantly pushed us to be better researchers, better engineers, and better writers, significantly broadening the scope of our abilities and our contributions.

Thanks to my program advisor, David Mazières, for asserting from the very beginning that I should work on whatever excites me. Little did he know it would take me into another lab entirely. So ingrained is his conviction that students are their own bosses; I will never forget the look he gave me when I asked for permission to take a few weeks off (I may as well have been requesting permission to breathe).

Thanks to Mendel for co-instigating the RAMCloud project and for his work on LFS, which inspired much of RAMCloud's durability story, and for agreeing to be on my defense and reading committees. His knowledge and insights have and continue to move the project in a number of interesting new directions.

Thanks to Michael Brudno, my undergraduate advisor, who introduced me to research at the University of Toronto. Had it not been for his encouragement and enthusiasm for Stanford I would not have even considered leaving Canada to study here. I also owe a great deal of gratitude to my other friends and former colleagues in the biolab.

This work would not have been possible were it not for my fellow labmates and RAMClouders in 444, especially Ryan and Diego, the two other "Cup Half Empty" founding club members from 288. So many of their ideas have influenced this work, and much of their own work has provided the foundations and subsystems upon which this was built.

Finally, thanks to Aston for putting up with me throughout this long misadventure.

This work was supported in part by a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship. Additional support was provided 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. Further support was provided by Stanford Experimental Data Center Laboratory affiliates Facebook, Mellanox, NEC, Cisco, Emulex, NetApp, SAP, Inventec, Google, VMware, and Samsung.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last decade there has been a tremendous uptick in the use of solid-state storage in datacentres. The reason for this is simple: many large web applications – from search to social networking – demand much higher random access performance than hard drives can provide. In order to scale these applications, the primary location of data has gradually moved from spinning disks to flash memory and DRAM.

Solid-state storage technologies offer two distinct advantages over hard disks: higher I/O bandwidth and lower access latency. With no seek or rotational delays, the time to read a byte of data from DRAM or flash memory is at least 50-50,000x faster than from a hard disk (about 50μs for flash and 50ns for DRAM vs. 2ms or more for hard drives). By combining this latency improvement with internal I/O parallelism, solid-state devices offer random access bandwidth that is 2 to 6 orders of magnitude greater than hard disks.

Unfortunately, only the bandwidth advantage can be properly exploited in current datacentre storage systems. The reason is that network latency has been slow to improve. For example, even today round-trip latencies between servers in a single datacentre are often measured in the hundreds of microseconds, or even milliseconds. This has meant that while network storage systems can absorb thousands of times more requests per second with solid-state devices, the time needed to wait for each response has dropped only about 10x compared to hard disk latencies.

The result of this bandwidth-latency mismatch is that datacentre applications are artificially limited in the amount of interdependent storage accesses they can make. For example, if a web server needs to generate a page within 100ms to provide adequately fast response times to users, then it will only be able to issue a few hundred sequential accesses before exceeding its time budget. This severely restricts the amount of data such applications are able to interact with and forces them to issue many requests to the storage system in parallel in order to take advantage of the bandwidth available to them. This, in turn, adds to application complexity and requires workarounds such as denormalising data to reduce the number of dependent accesses and keep response times low. Even worse, it makes many sequential, data-intensive operations impossible, which means that any application requiring this level of interaction with the storage system is simply not implementable today.

The RAMCloud project is our answer to the problem of datacentre storage latency. RAMCloud started at a time when general-purpose solutions to high network latency were beginning to appear on the horizon. By 2009, commodity network device manufacturers were introducing low-latency 10Gb Ethernet switches and RDMA-capable NICs. We hypothesised that within 5 to 10 years the time to send a small request to a server and get a reply within a single datacentre would drop 10-100x with commodity hardware – down to 5μs per round-trip or less. Improvements in network switching, network interface card design, protocols, operating systems, and software stacks would finally allow the low latency advantage of solid-state storage to be exploited across the network.

The primary goal of RAMCloud has been to build a DRAM-based storage system fast and scalable enough to exploit future datacentre networking and provide extremely low-latency access to very large data sets (petabytes, or more). DRAM was chosen in order to push the boundaries of latency as aggressively as possible. Even flash, with about 10x higher latency than our performance goal of 5μs, would not be fast enough. The hypothesis was simple: if we were to build a storage system that offered 100-1,000x lower latency than existing systems, we could stimulate an exciting new breed of large-scale web applications that interact with data much more intensively than was possible before.

In designing RAMCloud we had to tackle a number of technical challenges, one of the most important of which was making efficient use of DRAM. We knew that to build a viable system we needed to use memory frugally – such a precious and expensive resource could not be wasted. At the same time, however, we needed to ensure that this did not conflict with our other goals of low latency, high performance, durability, and availability.

Each of these goals would have been relatively easy to achieve in isolation, but we wanted them all, and this made designing RAMCloud particularly challenging. For example, it is well known from work in memory and storage managers (allocators, copying garbage collectors, filesystems, etc.) that there is a fundamental trade-off between performance and storage efficiency. The larger the fraction of memory or disk one uses to store data, the higher the overheads are in managing it, and the lower the performance will be. As I show in Chapters 3 and 8, current memory allocation schemes favour performance well above efficiency, and are particularly wasteful under pathological access patterns. To meet RAMCloud's goals we needed something new.

This dissertation details the core storage mechanisms in RAMCloud, explaining how we manage volatile memory and durable disks to attain the aggressive goals we set out to achieve. The primary contribution is the single pervasive way in which memory is managed in the system: all data, whether in DRAM or on disk-based backups, are stored in append-only logs. This single technique has wide-ranging benefits, including low write latency and high write throughput, even at 90% server memory utilisation. This approach also provides excellent durability and availability (for more details on these aspects of the system, see [75]).

Although this text is set within the context of RAMCloud, the techniques developed for it can be more widely applied in the design and implementation of other DRAM-based storage systems and caches. As a proof of concept, Section 8.4 explores the advantages of using log-structured memory in the memcached [5] object caching server.

1.1 Contributions

The primary contributions of this dissertation are as follows:

1. **Log-structured Memory**
   RAMCloud uses a pervasive log structure: all data, whether active in DRAM or backed up on disk, is stored in append-only logs. This novel approach allows RAMCloud to achieve high write throughput, low write latency, high server memory utilisation, and fast recovery from server crashes. Although logging has been used for decades in disk-based storage systems, RAMCloud is the only system I know of that uses the same unified technique to manage both primary and secondary storage. Log-structured memory is a key enabler for RAMCloud: it mitigates the high cost of DRAM by allowing servers to run at up to 90% memory utilisation with high write throughput (over 400K durable 100-byte writes/second/server), and it only keeps a single copy of data in DRAM without sacrificing durability or availability.

2. **Two-level Cleaning**
   A natural consequence of log-structured storage is the need for a defragmentation mechanism (a *cleaner*) to consolidate free space into large contiguous regions that new data can be appended to. RAMCloud uses a *two-level* cleaning approach that significantly reduces the network and disk I/O costs of cleaning by exploiting the fact that data is present in both DRAM and on disk. In many cases cleaner I/O overheads are greatly reduced over a one-level technique: 12.5x on average, and up to 76x. This combined approach allows RAMCloud to exploit the strengths and mitigate the weaknesses of the two mediums: disk space is cheap, but bandwidth is poor, whereas DRAM space is precious and bandwidth is plentiful. Two-level cleaning allows the system to run at high DRAM utilisation while minimising its usage of disk and network bandwidth.

3. **Parallel Cleaning**
   To achieve our latency and bandwidth goals, RAMCloud's log cleaner runs in parallel with regular storage operations. New writes may be appended to the log while the cleaner is running. Moreover, the cleaner exploits processor parallelism by cleaning in multiple separate threads, as well as disk parallelism by writing to different backup disks. Our parallel cleaner allows the system to maintain high write rates while also having only a negligible impact on the latency of regular write requests while it is running. A 100ns increase in 90th percentile write latency due to simultaneous cleaning is typical (equivalent to about 0.6%), and unlike even the most advanced garbage collectors, the system never needs to pause other threads while cleaning.

4. **Methods for Maintaining Delete Consistency**
   One of the more surprisingly complicated data consistency issues RAMCloud has had to tackle is

ensuring that deleted objects do not come back to life after a crash recovery. RAMCloud uses special delete records, called *tombstones*, to denote deleted objects. However, these present a number of additional challenges. RAMCloud's ability to migrate data between servers further complicates delete consistency. This dissertation explains the complications tombstones add, how their presence in the log affects cleaning, and discusses several alternatives I considered.

5. **Cost-Benefit Policy Improvements**
   While developing RAMCloud I discovered some unexpected issues with the cost-benefit policy originally introduced in log-structured filesystems [69]. Although the general idea is sound, the implementation described has a few issues. First, the policy does not weigh the two factors it takes into account properly (stability of data and amount of free space in each segment). Second, it uses a weak approximation for data stability. This dissertation explores these problems, introduces some solutions, and shows that cleaning overheads can be reduced by as much as 50% with a very simple change to the original formula. I also briefly address the question of whether cost-benefit selection is an optimal strategy.

1.2 Dissertation Outline

Chapter 2 introduces the RAMCloud vision, motivates the idea, and overviews the architecture of the system. Chapter 3 explains why I chose not to use a standard memory allocator to manage DRAM in RAMCloud, opting for a pervasive log structure instead. Chapter 4 explains in detail the design and mechanisms behind log-structured memory in RAMCloud. This is the chapter most pertinent to readers wanting to understand how memory is managed in the system. It also introduces the two-level cleaning mechanism, which is key to using both memory and disk effectively. Chapter 5 explores scheduling policies for two-level cleaning (how to balance the running of each of the two cleaners, how many threads to use, etc.). Chapter 6 discusses the difficulties involved in maintaining consistency of deleted objects in the face of server failures and shows how RAMCloud uses tombstones to accomplish this. It also explores several alternatives in addition to the current solution. Chapter 7 further analyses and improves upon the cost-benefit cleaning policy originally introduced in [69]. Chapter 8 evaluates the performance of RAMCloud's log-structured memory. Chapter 9 discusses related work, and Chapter 10 concludes the dissertation.

Chapter 2

RAMCloud Background and Motivation

The modern commodity datacentre took form in the mid-to-late 1990s. As the world-wide web grew in scope and popularity, researchers and industry alike began to realise the cost and performance benefits of using commodity hardware to underpin large-scale web applications [40]. For example, within only half of a decade from 1994 to 1999, web search transitioned from using low-volume, high-end servers such as DEC Alphas at AltaVista to commonplace x86 servers – particularly, and most aggressively, at Google. Although the idea of harnessing commodity computer and network technologies to build distributed systems was not entirely new [19], the scope and scale of these emerging web services was unprecedented.

In the past fifteen years, the commodity hardware technologies that drive web datacentres have improved markedly with respect to nearly all metrics, with one important exception: network switches and interface cards have not significantly reduced communication latency. This means that the amount of time it takes to issue an RPC within a datacentre has stagnated, despite processing power, storage capacity, and network bandwidth having all improved dramatically. As a result, datacentre storage system latencies are much higher than they otherwise could be, especially for increasingly popular DRAM-based systems. Fortunately, however, a 100-fold reduction in commodity network latency is finally on the horizon.

This chapter explains how combining low-latency networking with solid-state memory will make it possible to build storage systems that respond to requests 100 to 1,000 times faster than what is commonly in use today. I then argue that these storage systems will be key enabling technologies, which will lead to an exciting new class of datacentre applications that could not be built without them. Finally, I will introduce the overall architecture of RAMCloud. RAMCloud is a DRAM-based storage system we built to provide the lowest possible storage latency for datacentre applications.

2.1 Enabling Low-latency Storage Systems

There are two important shifts in datacentre computing that make low-latency storage systems like RAMCloud possible. The first is the use of solid-state memory technologies like DRAM as the location of record

	1983	2013	Improvement
CPU Speed	1x10MHz	8x3GHz	2,000x
Main Memory Capacity	2MB	32GB	16,000x
Disk Capacity	30MB	3TB	100,000x
Network Bandwidth	3Mbps	10Gbps	3,000x
Network Latency (Round-trip)	2.5ms	50μs	50x

Table 2.1: Network latency has improved far more slowly over the last three decades than other performance metrics for commodity computers. Thirty years ago, the V Distributed System [27] achieved round-trip RPC times of 2.54ms. Today, a pair of modern Linux servers require about 50μs for small RPCs over TCP with 10Gb Ethernet, even when directly connected with no switches in-between (latencies across datacentre networks are commonly 5-10x higher).

for most or all data. This shift is already well under way. For example, at present all major search engines such as Google and Microsoft Bing answer their queries from in-memory storage in order to return high quality results quickly, regardless of how popular or esoteric the search terms may be [15]. Large social networking sites such as Facebook have also mirrored this intense need for fast datacentre storage, relying on hundreds of terabytes of DRAM caches to scale their applications [59].

Although DRAM-based storage systems provide both higher bandwidth and lower latency access than disk-based systems, the achievable latency benefit has been much lower than what is technically possible due to high latency datacentre networks. Companies like Facebook, for example, report access times of hundreds of microseconds – or even milliseconds – when servicing RPCs across their datacentre networks [59]. This means that while a random hard disk access takes about 2-8ms (due to seek and rotational delays), network round-trip times are only an order of magnitude less in typical datacentres today. Consequently, switching from disk- to DRAM-based storage provides at most a 10x latency improvement – only a small fraction of the 4-5 orders of magnitude difference between hard disk and DRAM access times.

Reducing network latency could unlock much more of the latency benefits of DRAM, but unfortunately improvements in commodity network latency have been hard to come by over the years. This is a particularly long-standing problem: latency reductions have simply not kept pace with other hardware improvements over the last three decades (Table 2.1). One reason for this may be that sources of high latency exist throughout the datacentre. For example, switches are slow, have large buffers, and have been long optimised for high bandwidth TCP connections. Network interface cards are designed for throughput as well: it is common for them to intentionally delay packets to coalesce interrupts at the cost of latency. Finally, operating system network stacks add tens of microseconds as data moves from network card, to kernel, to userspace, and back again.

The second enabling shift for RAMCloud will be fixing the latency bottlenecks in commodity networks. A further 100x reduction in storage system latency is possible if the underlying network latency is reduced to be competitive with more specialised networking technologies like Infiniband, which today are capable of sub-5μs round-trip times in a single datacentre. Fortunately, such solutions that exist in the supercomputing space have recently been migrating into commodity Ethernet hardware. For example, 10 Gb Ethernet switches with sub-microsecond latencies were introduced by Arista Networks in 2008. In addition, low-latency Ethernet network interface cards are now available from companies such as Mellanox and SolarFlare. These cards permit direct access to the network hardware by user applications, bypassing the kernel and significantly reducing software overheads. While early adopters of such low-latency Ethernet hardware have primarily used it in niche applications like high-frequency trading, we expect that it is only a matter of time before it is widely adopted in datacentres.

Once this happens, the hardware foundation will be set for storage systems such as RAMCloud that provide 100-1,000x lower latency than current disk- and DRAM-based systems. Applications will then be free to exploit this much faster storage in exciting new ways.

2.2 The Potential of Low-Latency Datacentre Storage

If systems like RAMCloud can improve datacentre storage latency by 2-3 orders of magnitude, we believe that there will be a revolutionary impact on applications. The following are a number of ways we speculate that such storage will change the application landscape.

The most immediate benefit will be for existing applications that suffer from high storage latency, such as Facebook's social network, or Google's statistical machine translation [22]. Low-latency storage will make these applications faster and more scalable, and more in-depth data exploration will be possible in real-time. In addition, they will become much simpler to develop, since it will no longer be necessary to employ complex workarounds for high latency. For example, high storage latency has limited Facebook's applications to 100-150 sequential data accesses within the datacentre for each web page returned to a browser (any more would result in unacceptable response times for users). As a result, Facebook has resorted to parallel accesses and de-normalised data, both of which add to application complexity [49].

Even more exciting is the prospect of enabling new applications that were simply not possible before. For example, an application that needs to harness thousands of machines in a datacentre and intensively access random bits of data distributed across the main memories of those machines (e.g., for large-scale graph algorithms), are just not practical today: they will bottleneck on the response time of the storage system. In the past, significant latency improvements were not possible because most network requests resulted in long disk I/Os. However, moving the primary location of data from disk to DRAM makes latency reductions of 100-1,000x is possible.

One particularly interesting class of new applications consists of those requiring real-time analysis of massive amounts of data (sometimes referred to as "Online Data- intensive Services" [58]). The web has made it possible to assemble enormous datasets, but high latency has severely restricted the kinds of on-demand operations that can be performed on them so far. We speculate that low latency will enable a new

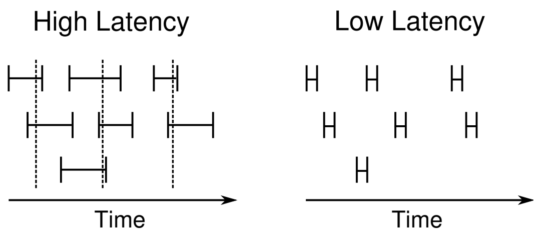


Figure 2.1: Reducing latency should lower the execution time of transactions and make conflicts less likely. In the above example, each horizontal bar represents the execution time of a single transaction. Vertical overlaps indicate simultaneous transactions that may conflict (dashed lines). Shorter transactions are less likely to overlap and thereby reduce the occurrence of conflicts.

breed of real-time, extremely data-intensive applications. For example, low-latency storage will allow more intensive and interactive manipulation of large datasets than has ever been possible in the past. It is hard to predict the nature of these applications, since they could not exist today, but possible examples include large-scale interactive applications, such as collaboration at the level of large crowds (e.g., in massive virtual worlds) and massively multiplayer games.

We also hypothesise that low-latency storage systems may avoid certain problems that have plagued previous storage systems. For example, many NoSQL systems have sacrificed consistency guarantees by limiting atomic updates to a single row or offering only eventual consistency [34, 25, 30]. Strong consistency is expensive to implement when there are many transactions executing concurrently, since this increases the likelihood of costly conflicts. In a very low latency storage system, however, transactions finish more quickly, reducing the overlap between transactions and minimising conflicts (Figure 2.1). We speculate that low-latency systems like RAMCloud may be able to provide stronger consistency guarantees at much larger system scale.

Low latency storage may also reduce the incast problems experienced by many applications. When accesses take a long time, applications are forced to issue concurrent data requests in order to meet user response deadlines. However, such requests can lead to simultaneous responses, which can cause congestion near the client, overflow small switch buffers, and result in packet loss. This scenario could be avoided if sequential accesses were sufficiently fast that concurrent requests are greatly reduced, or no longer needed at all.

2.3 RAMCloud Overview

The previous two sections provided the context and vision for building low-latency datacentre storage systems. The rest of this chapter describes the high-level details of the RAMCloud storage system that we built. It provides both an overall introduction to the system in general, as well as a basis for the rest of the dissertation.

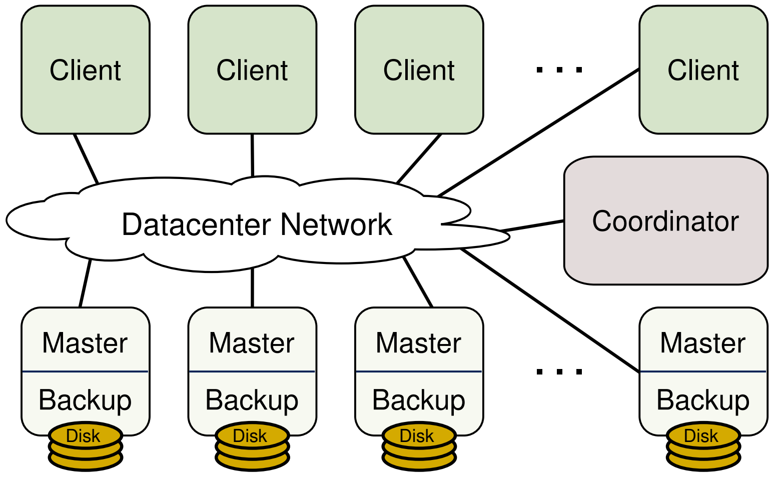


Figure 2.2: RAMCloud cluster architecture. Each storage server contains two modules: a master, which manages objects in its DRAM, and a backup, which stores segment replicas on disk or flash. A central coordinator manages the cluster of masters and backups. Client applications access RAMCloud with remote procedure calls.

2.3.1 Introduction

In RAMCloud, all data is present in DRAM at all times; secondary storage is only used to hold redundant copies for durability. We chose DRAM as our primary storage technology for one important reason: to get the lowest possible storage latency. Simply using DRAM does not guarantee a low-latency system by itself, however. We also needed to carefully architect the RAMCloud software to be efficient enough to exploit DRAM's latency advantage and not squander it with software overheads. This forced us to squeeze out latency wherever possible. Inefficiencies that would have been inconsequential had we chosen a slower storage technology are much more pronounced by using DRAM. We justified this extreme design by our desire to see how it would impact applications that use such a fast system. By pushing the envelope as far as we can, we hope to provide the best possible basis for fostering novel applications.

We could have chosen to use flash memory instead, but this would have required compromising on our low latency goal. Although flash is cheaper per bit, with access times measured in the tens of microseconds, it has much higher latency – significantly higher, in fact, than the network latencies we expect in the near future. DRAM is currently the lowest latency storage technology that is inexpensive enough to be financially viable, dense enough to store useful amounts of data, and lets us take maximum advantage of low-latency networking.

There is already strong precedent for using DRAM in datacentre storage. In particular, the popularity of memcached [5] and other "NoSQL" storage systems provides an initial indication of the need for – and feasibility of – a DRAM-based system like RAMCloud. The demand only seems to be growing. In late 2008, for example, Facebook had 800 memcached servers with 28TB of DRAM [1]. By mid-2009 these numbers had more than doubled [62], and as of 2013 the company runs multiple memcached clusters per datacentre, where each cluster consists of thousands of servers [59].

In order to store datasets much larger than the capacity of a single machine, RAMCloud is designed to aggregate the DRAM of hundreds or thousands of servers within a datacentre, as shown in Figure 2.2. The following subsections describe how we architected the system to achieve this.

RAMCloud is written in C++ and runs on commodity 64-bit x86 Linux servers. Although we do not rely on special features of Linux, we have taken advantage of Infiniband [74] to approximate the low-latency networks we expect to be commodity in the near future. This has allowed us to achieve 5μs reads and 16μs durable writes for small objects. For maximum performance, we take advantage of some x86-specific hardware features; in particular we assume little endianness, lax data alignment requirements, and use the *crc32* instructions for fast data checksumming and *rdtsc* for fine-grained performance measurement. However, we expect that porting RAMCloud to other architectures and operating systems will be straightforward.

2.3.2 Master, Backup, and Coordinator Services

There are three general types of building blocks in a RAMCloud system (Figure 2.2). This subsection explains what they are and the role that they play. These entities work together to provide a durable and consistent storage system for clients to use.

Each storage server consists of two distinct components. A *master* module manages the main memory of the server to store RAMCloud objects; it handles requests from clients such as reads and writes. A *backup* module uses local disks or flash memory to store backup copies of data owned by masters on other servers. The data written to backups is not used to service normal requests; it is only read during recovery of a failed master server or when restarting a cluster that was previously shut down. Subsection 2.3.4 describes the master server architecture in greater detail. For more details on backups, see [75].

The masters and backups are managed by a central *coordinator* that primarily handles configuration-related issues such as cluster membership (which masters and backups are online), and the distribution of data among the masters. Another important – though less frequently invoked – responsibility is determining which masters and backups have failed and then orchestrating their recovery. Importantly, the coordinator is not normally involved in common operations such as reads and writes. Clients of RAMCloud maintain local caches of the coordinator's data placement information in order to communicate directly with masters; they only contact the coordinator when their cache is missing the necessary information or if their cached information is stale. This reduces load on the coordinator and allows it to scale to large cluster sizes. It also reduces client latency, since in the common case clients can go straight to the master that owns a desired piece of data.

The coordinator appears to the rest of RAMCloud as a single service, and as far as other servers' interactions with it are concerned, it could as well be a single physical server. However, to avoid a single point of failure the coordinator service is actually built atop a fault-tolerant distributed consensus protocol involving

RPC Definition	Description
createTable(*tableName*) → *tableId*	Create a new table named *tableName* if it does not already exist and return the 64-bit handle used to manipulate objects in that table.
getTableId(*tableName*) → *tableId*	Obtain the 64-bit handle for the existing table named *tableName*.
dropTable(*tableName*) → *status*	Drop the existing table named by *tableName*. Any objects within the table are permanently deleted.
read(*tableId*, *key*) → *value*, *status*	Get the value of the object named *key* in table *tableId*.
write(*tableId*, *key*, *value*) → *status*	Set the value for the object named *key* in table *tableId* to *value*.
remove(*tableId*, *key*) → *status*	Remove the object named *key* in table *tableId*.

Table 2.2: Basic operations in the current version of RAMCloud. Clients issue these RPCs to RAMCloud to store, retrieve, and delete data. RAMCloud also supports other less frequently used operations, such as conditional and batched versions of read, write, and remove, as well as enumeration of all objects within a table.

approximately 3 to 7 distinct servers. At any point in time at most one of these servers is the coordinator and may process RPCs issued to the coordinator service. The others are hot standbys; if the current coordinator crashes, one of the standbys is selected to take its place. Determining the new coordinator, as well as keeping consistent replicas of the current coordinator's state (so that the new coordinator can recover from the previous one's failure), is provided by a separate distributed consensus service. RAMCloud's coordinator is designed to support various different consensus implementations, including ZooKeeper [47] and the LogCabin [4] implementation of the Raft [60] consensus protocol.

2.3.3 Data Model

RAMCloud provides a simple key-value data model consisting of uninterpreted data blobs called *objects* that are named by variable-length *keys*. Objects may be up to 1 MB in size and must be read or written in their entirety. Keys are uninterpreted byte strings up to 64 KB long. When designing RAMCloud we noted that existing DRAM-based stores such as *memcached* tend to be used for storing small objects (tens of bytes to a few kilobytes) and perform relatively simple operations on the data. Table 2.2 summarises the main operations that RAMCloud clients can perform to manage their objects.

Objects in RAMCloud are grouped into *tables*, which provide separate namespaces for sets of keys. Tables can be used to isolate and avoid naming conflicts between separate applications, or separate modules within applications. Clients invoke the coordinator to allocate tables on one or more masters. For efficiency, tables are internally named by unique 64-bit identifiers: clients access objects by specifying $\langle tableId, key \rangle$ tuples. For convenience, the coordinator maintains a mapping to each table identifier from a unique string name; clients first resolve the table name to the internal identifier and then use this concise handle to perform any object operations on the table. Tables may span one or more servers in the cluster; a subset of a table stored on a single server is called a *tablet*.

RAMCloud relies on hashing object keys to more or less evenly distribute data across tablets. Internally, each RAMCloud table is a 64-bit hash space, and tablets are simply contiguous, non-overlapping subsets of the 64-bit space that can be assigned to different servers. The hash value of an object's key determines which tablet it belongs to (and therefore which server it resides on). In the simplest case, a table consists of a single tablet that spans the entire range of hash values (from $[0, 2^{64})$) and is assigned to a single master. Spreading a table across multiple masters requires multiple tablets. For example, a table split evenly between two servers would consist of two evenly-sized tablets, namely ranges $[0, 2^{63})$ and $[2^{63}, 2^{64})$. RAMCloud relies on the uniformity of the hash function to evenly spread data across tables in proportion to the amount of hash space they cover.

Clients read and write objects by talking directly to masters in order to minimise latency. For this to work, clients must be able to figure out which master server is responsible for each object they want to access. This simply involves looking up the tablet to which an object's key belongs and contacting the master that owns the tablet. The lookup is done using the table identifier and the 64-bit hash value of key. Each client maintains a cache of tablet to master server mappings. The coordinator maintains the most up-to-date mappings of tables to tablets and tablets to master servers, so a client's initial object access requires fetching this information from the coordinator. Caching on the client speeds up subsequent accesses and avoids inundating the coordinator on each object operation. In some cases, such as after a master fails, a client's cache may become stale and operations can be sent to a dead master (or the wrong master). When this happens, either the RPC times out (because the server had failed), or the contacted master replies that it does not own the key in question. The client will then contact the coordinator, update its mappings, and try again.

Each object in RAMCloud has an associated 64-bit *version number* set by the server it was written to. An object's version number monotonically increases as it is modified and can be used by clients to perform conditional operations. These primitives can be used by clients to ensure consistency (for example: only overwrite the current value if the object has not changed since the last read). They can also be used to build higher-level abstractions in client-side libraries such as locks or transactions. For example, a lock can be represented by an object containing a single flag indicating if the lock is currently held or not. Acquiring the lock involves reading the object and, if the lock flag is not set, attempting a write conditional on the previous version of the object. If the write succeeds, the lock was successfully acquired.

The RAMCloud data model was intentionally kept simple. This allowed us to initially focus our attention on attaining the highest performance, lowest latency, and best system scalability. The popularity of key-value stores and caches such as memcached [5] indicated to us that there was still significant value in a simple data model. Future work by other students will explore extending the data model in various ways (for example, with secondary range-queryable indexes, transactions, and other higher-level features). It remains an interesting open question how rich the data model can be made while maintaining our latency, performance, and scalability goals.

2.3.4 Master Server Architecture

The rest of this dissertation focuses primarily on the inner workings of master servers, in particular how they manage their DRAM to store objects. This subsection briefly introduces the structure of masters and explains the important concepts needed to understand the subsequent chapters.

A master consists primarily of a collection of objects stored in DRAM and a hash table that is used to locate the objects in memory (see Figure 2.3). The hash table contains one entry for each live object stored on the master; it allows objects to be located quickly given a table identifier and key. Each live object has exactly one direct pointer to it, which is stored in its hash table entry. Objects are always referenced indirectly via the hash table.

Masters log data to backup disks to ensure durability in the face of server crashes and power failures. Each master maintains its own separate log, which is divided into 8 MB pieces called *segments*. Logs are always read or written in units of entire segments to maximise I/O bandwidth. This is especially important when rotating hard disks are used for secondary storage because 8 MB segments amortise seek and rotational latencies. Flash-based devices also benefit, however, since large reads and writes may be parallelised across many flash cells or chips.

The memory used to store objects in master DRAM is log-structured. This means that all objects in a master's memory are stored in an append-only log similar to the disk-based logs used for durability. In fact, the disk- and memory-based logs are very tightly coupled. The in-memory log also consists of 8 MB segments, and the segments stored on backups are just mirrors of the segments in DRAM. In other words, for every segment in the in-memory log there is a corresponding segment backed up on disk. Chapter 3 discusses why we chose this unusual technique for both memory and disk management, and the following chapters describe its implementation and the ramifications of this choice in detail.

Each log has a newest segment referred to as the *head* segment. Only the head may be appended to; all other segments are immutable. When a master receives a write request from a client, it adds the new object to its in-memory log, then forwards information about that object to the backups for its current head segment. Each backup appends the new object to a segment replica stored in a non-volatile buffer; it responds to the master as soon as the object has been copied into its buffer, without issuing an I/O to secondary storage. Once the master has received replies from all the backups, it adds a reference to the object in its hash table and 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: it allows writes to complete without waiting for I/O to secondary storage, and it uses secondary storage bandwidth efficiently by coalescing small objects into large disk writes.

Each head segment is replicated on several different backup servers (typically two or three). A master uses a different set of backups to replicate each segment so that its log ends up scattered across the entire cluster. This allows a single master to harness the I/O bandwidth of many backups to handle bursts of writes. It also makes the entire I/O bandwidth of the cluster available for reading a master's segment replicas during crash recovery, allowing a crashed master to be recovered in only a couple of seconds. This allows RAMCloud to

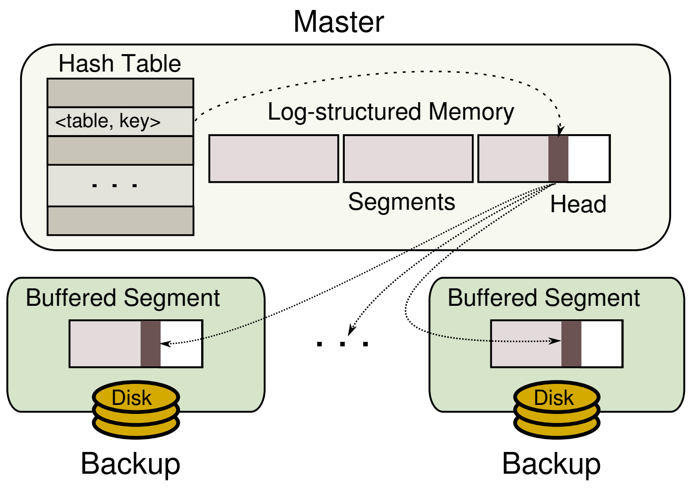


Figure 2.3: A master server consists primarily of a hash table and an in-memory log, which is replicated across several backups' disks for durability. New objects are appended to the head log segment and are synchronously replicated to non-volatile staging buffers on backups. Client writes are acknowledged once all backups have buffered the new addition to the head segment. Once the head segment is full, backups flush the corresponding staging buffers to disk.

provide high data availability while keeping only a single copy of each object in DRAM [75].

The segment replicas stored on backups are only read during crash recovery. Backup replicas are never read during normal operation; most are deleted before they have ever been read. 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 [61, 75]. Data is never read from secondary storage in small chunks; the only read operation is to read a master's entire log.

One consequence of structuring storage as an append-only log is that RAMCloud needs a mechanism to reclaim the free space that accumulates in segments when objects are deleted or overwritten. This fragmented free space needs to be collected and coalesced both in memory and on backup disks so that new segments can be created to store more data. RAMCloud accomplishes this with a *log cleaner* similar to that of log-structured filesystems [69]. Each master runs a cleaner for its own log as follows:

- The cleaner selects several segments to clean, using a cost-benefit approach similar to LFS (segments are chosen for cleaning based on the amount of free space and the stability of the data). Chapter 7 explains this in detail.
- 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 significantly reduces

the I/O overhead for cleaning, especially when the segments have little live data (otherwise an entire segment would have to be read in order to write out a small portion).

- 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 has important benefits for managing master memory. First, it copies data to eliminate fragmentation, which makes it more space-efficient than non-copying allocators, especially when allocation patterns change. Second, it operates incrementally, cleaning a few segments at a time. This means that space can be reclaimed without having to first scan all of memory. Finally, unlike traditional language-based garbage collectors, cleaning never requires pausing the master server process. Chapter 3 discusses these points in greater detail.

However, this approach also 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 Chapter 6. Second, log cleaning can be quite expensive at high memory utilisation [72, 73]. RAMCloud uses two techniques to reduce the impact of log cleaning: two-level cleaning and parallel cleaning with multiple threads. For details, see Chapter 4.

2.4 Summary

RAMCloud's goal is to provide a scalable datacentre storage system with very low latency: 100-1,000x lower than typical systems today. I have argued that this drastic reduction in response time will not only improve existing applications, but also enable interesting new breeds of applications that cannot be built today. Examples of these include traversal of massive data structures spread across many servers with little locality, such as social graphs. Reduced latency will greatly increase the amount of data that may be accessed in real time, which should allow for more sophisticated and scalable crowd applications like virtual worlds and multiplayer games.

Two computing trends are making systems like RAMCloud possible. First, DRAM density and cost has reached a point where large data sets can be cost-effectively stored entirely in main memory. Existing large-scale deployments of memcached and specialised in-memory stores like those that power web search engines demonstrate that DRAM-based storage is already acceptable and economically viable. Second, new commodity networking technology is bringing substantially reduced switching and network interface latencies to market, and we expect these technologies to become commonplace in datacentres of the near future. With efficient software systems, low-latency networking will allow storage systems to exploit DRAM's extremely low access times.

This chapter also presented the overall architecture of a RAMCloud storage cluster and introduced key concepts for the rest of the dissertation. Some important points include: 1) RAMCloud exposes a key-value data model to applications, 2) a master's hash tables contain the only direct pointers to its objects, 3) masters use logging to manage both memory and backup disk space, and 4) durable logs are spread across remote backups to use cluster I/O resources in aggregate. The rest of this dissertation builds on these foundations, explaining why RAMCloud uses log-structured memory, how it is implemented, and the ramifications of this design choice.

Chapter 3

The Case for Log-Structured Memory

RAMCloud could have used a traditional storage allocator for the objects stored in a master's memory (*malloc*, for example), but we chose instead to use the same log structure in DRAM that is used on disk. This chapter explains the rationale behind our approach to master memory management and why existing general-purpose allocators are ill-suited for DRAM-based storage systems such as RAMCloud.

3.1 Memory Management Requirements

Memory-based storage systems like RAMCloud need space-efficient memory management to keep costs under control. DRAM is an expensive resource, and we expect it to be a significant fraction of a RAMCloud cluster's total cost. In particular, we estimate that memory will account for about half of the price of each RAMCloud server, so waste can result in a large capital cost. Section 3.2 explains how existing general-purpose allocators fail to meet this efficiency requirement, especially when access patterns change over time. Optimally, RAMCloud's memory manager would allow a very high percentage of memory (80-90% or more) to be used to store object data, regardless of how applications use the system.

The memory manager must also have high allocation performance. Clearly the faster the allocator is, the better, but mean allocation throughput is not the only important metric. In particular, a suitable allocator should not only be capable of servicing a large number of allocations per second on average, but it should also have low variance in the latency of individual allocations. This is especially important for RAMCloud as low latency is a major system goal. With networking capable of sub-5μs RPCs, the allocator must be able to satisfy the vast majority of allocation requests in less time than it takes a client to issue a small write request to a master, and for the master to forward it to three replicas (a little over two round-trip times, or about 10μs).

RAMCloud's aggressive latency goals also require a memory manager that minimises latency penalties for threads that are not interacting with the allocator. For example, a common concern for language-based garbage collectors are *pauses*, periods of time in which all threads of the application are halted while the collector performs some or all of its work. Pauses are antithetical to low-latency systems like RAMCloud;

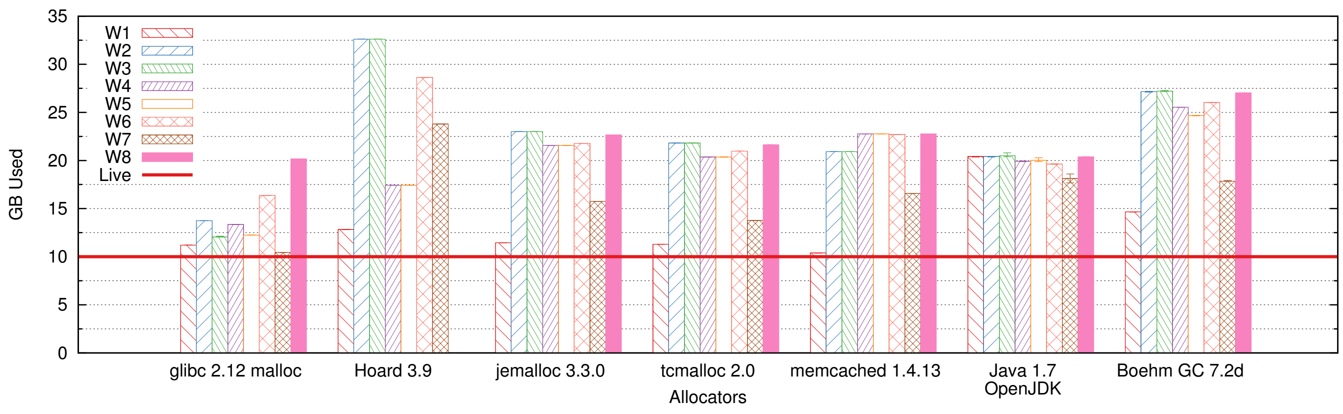


Figure 3.1: Total memory needed by allocators to store 10 GB of live data under the changing workloads described in Table 3.1 (average of 5 runs). "Live" indicates the amount of live data, and represents an optimal result. "glibc" is the allocator typically used by C and C++ applications on Linux. "Hoard" [18], "jemalloc" [37], and "tcmalloc" [3] are non-copying allocators designed for speed and multiprocessor scalability. "Memcached" is the slab-based allocator used in the memcached [5] 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).

a suitable memory manager should not cause collateral damage to the latency of requests that have nothing to do with memory allocation, such as reads (the most common operation), and ideally would not adversely affect writes, either.

The goals of high memory efficiency and high allocation throughput fundamentally conflict with one another. It is easy to build an allocator that is very fast, but inefficient. Likewise, if performance is not a goal it is much simpler to build a more efficient allocator. RAMCloud's requirements are particularly demanding: we want both properties. Moreover, we want both without any application pauses.

3.2 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, we will see that existing allocators are not able to use memory efficiently, particularly in the face of changing access patterns. For example, I 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 (free space is available, but in chunks that are too small to meet the needs of future allocations).

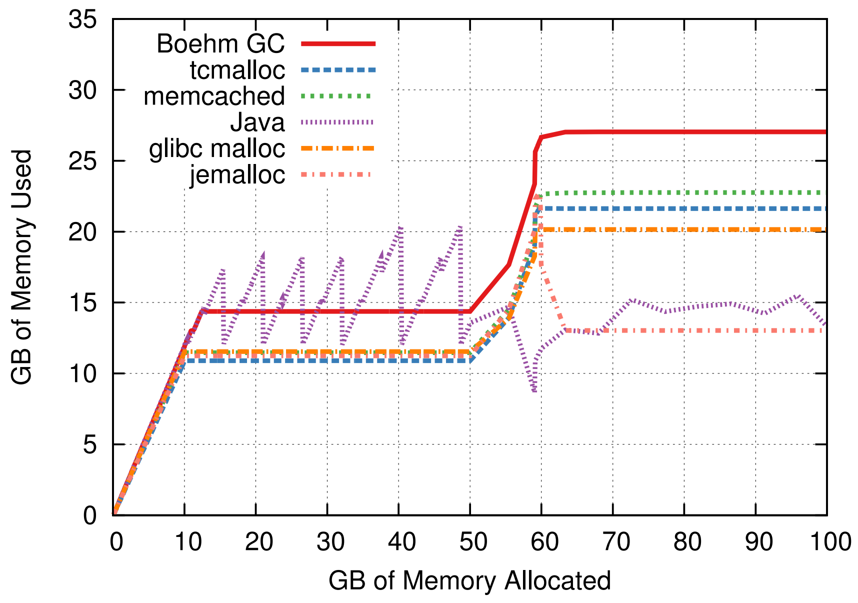


Figure 3.2: Total memory needed by allocators under one particular run of the W8 workload (after 50 GB of allocations the workload deletes 90% of the 50-150 byte allocated objects and transitions to 5,000-15,000 byte objects; see Table 3.1). At some point, every allocator uses at least twice as much space to store the same 10 GB of live data. Once transitioned, the Java copying collector is able to manage the larger objects more efficiently and reduces its overhead to about 50%. jemalloc releases some unused memory after fully transitioning, but not before peaking at almost 23GB. (jemalloc's actual memory use after release may be higher than reported because it uses the *madvise* syscall to return memory to the kernel. This operation unmaps the pages backing the freed virtual memory, but the addresses remain valid and empty pages are faulted in on demand if accessed again. Conservatively assuming that these pages were not accessed made accounting simpler and more efficient.)

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 3.1: Summary of workloads used in Figure 3.1. The workloads were not intended to be representative of actual application behaviour, but rather to illustrate pathological 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). See Figure 3.2 for an illustration of how these phases affect total memory use throughout a run of the W8 workload. 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.

Non-copying allocators work well for individual applications with a consistent distribution of object sizes, but Figure 3.1 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. As bad as these results were, the upper bound for non-copying allocators can be even worse [65], depending on the ratio of the largest to smallest allocation.

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 3.1 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 had to augment their cache eviction strategies with special-purpose code to cope with such eventualities [59]. 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. Such unpredictability forces overprovising; 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 [81, 46]. 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 [11, 26, 76] – 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 [61]). 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.

3.3 Benefits of Log-structured Memory

An ideal memory allocator for a DRAM-based storage system such as RAMCloud should have the following properties. First, it must be able to copy objects so that it can efficiently manage memory under any access pattern by eliminating 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, which improves its efficiency. Third, its operation should not pause application threads, especially those not that are not even allocating memory. It would not make sense to build a low-latency system like RAMCloud using a mechanism that adds significant variability to response times.

RAMCloud's log-structured memory addresses these requirements. For example, using a log cleaner (Section 2.3.4) to reclaim free memory is a form of defragmentation. This means that RAMCloud can adapt to changing access patterns and will not fall victim to the same space inefficiencies that plague non-copying allocators. As a result, memory is not wasted; it can be used cost-effectively, and does not need to be drastically overprovisioned to handle pathologies.

Log-structured memory also allows for incremental garbage collection, which enables RAMCloud to allocate memory efficiently even at high memory utilisation. In order for this 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 (in RAMCloud, the only direct pointer to an object is in its master's hash table). Chapter 8 evaluates RAMCloud's object allocation performance in detail. It is important to note that 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.

The restricted use of pointers in RAMCloud makes concurrent garbage collection relatively straightforward and avoids application pauses entirely. Part of the reason this is possible is that only one hash table reference must be changed when an object is moved by the cleaner. In contrast, language-based collectors must update any number of references anywhere in memory. Also, objects are immutable and written in a copy-on-write fashion, so a simple atomic swap of the hash table's object pointer is all that is needed to concurrently relocate or overwrite an object. This way, masters never have to worry about races between the log cleaner moving an object and a write request that updates it.

Using log-structured memory also has some nice implementation advantages for RAMCloud. For instance, this technique simplifies our code because a single unified mechanism is used for information both in memory and on disk. Most of the implementation size and complexity of RAMCloud's logging mechanism is due to persisting data to backups. Piggybacking on these subsystems to manage master memory as well as disk adds only a small amount of additional complexity.

Finally, log-structured memory reduces master metadata overhead in master servers: in order to perform log cleaning, 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). Furthermore, to track the amount of live data in each segment, the master must know the segment that an object belongs to so that statistics may be updated when it is deleted or overwritten. This would require adding another pointer from each object to the associated segment.

3.4 Summary

This chapter has argued that existing memory management techniques fail to provide all of the properties desired in DRAM-based storage systems like RAMCloud. Non-copying allocators, such as various malloc implementations, suffer from pathological access patterns that waste half of memory. Allocators that defragment memory, like copying garbage collectors, can adapt better to changing workloads, but have other downsides such as long pause times and inefficient allocation at high memory utilisation.

Log-structured memory is an alternative memory management technique for DRAM-based storage systems that addresses the weaknesses in other allocators. For example, log cleaning defragments memory to deal with changing access patterns, and the restricted use of pointers allows for efficient and pauseless reclamation of free space. The following chapters explain how RAMCloud implements log-structured memory and cleaning in detail before evaluating the performance and memory efficiency that RAMCloud is able to achieve with these techniques.

Chapter 4

Log-structured Storage in RAMCloud

Section 2.3.4 introduced the concept of log-structured memory and explained at a high level how RAMCloud uses this technique to store data in DRAM. The chapter also introduced the notion of using the same logging technique for managing durable copies of data on backup disks so that the system can recover from master failures.

The focus of this chapter is to expand on these topics in detail by describing how RAMCloud implements them. The goal is not only to explain the mechanisms chosen and the problems that they solve, but also to provide a primer for readers interested in better understanding the RAMCloud source code. For those interested in RAMCloud's internal implementation, this is the chapter to read. Some familiarity with the basic overall design of RAMCloud as described in Section 2.3 is assumed. In particular, I assume familiarity with the basic idea and operation of cleaning as described in Section 2.3.4.

This chapter also introduces and explains two important contributions of the dissertation: *two-level cleaning* and *parallel cleaning*. These methods reduce the I/O overheads for managing backup disks and hide much of the CPU expense of reclaiming free space in the log when cleaning. Chapter 8 evaluates the efficacy of these techniques in detail.

This chapter is structured as follows:

1. Sections 4.1 and 4.2 describe how logs are structured in memory and on disk. The sections detail the metadata that are preserved in or along with the log for crash recovery, as well as important soft-state that need not survive crashes. Together, these sections detail how objects are stored in the log, how it is iterated during cleaning and recovery, how recovery reconstitutes the log from segments scattered across different backups, how objects are quickly accessed using masters' hash tables, and how statistics are kept for making intelligent cleaning decisions.

2. Sections 4.3 and 4.4 build upon the previous sections by explaining what two-level and parallel cleaning are and how they are implemented. These mechanisms are crucial for RAMCloud's performance. Two-level cleaning reduces network and disk I/O overheads, which conserves bandwidth for useful log

writes. Parallel cleaning allows RAMCloud to reclaim free space and write to the log simultaneously, hiding much the overhead of cleaning. While parallel cleaning makes use of already-existing structures, two-level cleaning introduces an important new concept: discontiguous in-memory segments.

3. Section 4.5 explains why storing variable-size objects in fixed-length segments requires special care to avoid deadlocks during cleaning and how RAMCloud deals with this problem. This is a new and subtle problem for RAMCloud; log-structured filesystems with fixed block sizes did not have to worry about this issue.

4. Section 4.6 summarises this chapter.

4.1 Persistent Log Metadata

This section describes the persistent metadata that RAMCloud masters append to logs on backups in order to provide data durability. In particular, I describe how segments are structured, how they are linked together to form a log, how their contents are verified and iterated, and how the contents of objects and tombstones allow recovery to reconstruct the volatile hash tables that are lost when masters fail.

In contrast to log-structured filesystems, RAMCloud stores relatively little metadata in its log. While filesystems must maintain a considerable amount of indexing information in order to provide fast random access, each RAMCloud master only has a hash table in DRAM to provide fast access to information in its in-memory log. The hash table is volatile. Since it contains no information needed during recovery, it is never persisted to disk. There are two reasons why recovery does not need the hash table. First, the on-disk log is never randomly accessed (RAMCloud will only read an on-disk log during recovery and always reads the entire log, so random access need not be supported). Second, the hash table can be reconstructed from the metadata in the objects and tombstones of a master's on-disk log (assuming the log is replayed in its entirety, which is always the case). Section 4.2 discusses the hash table and other volatile metadata.

This section includes both metadata that are only useful during recovery (log digests, tombstones, and certificates) and well as metadata that is useful both for recovery and regular operation (log entry headers, and objects). In each case, however, these structures are both maintained in the memory of masters and are replicated to backup disks.

4.1.1 Identifying Data in Logs: Entry Headers

RAMCloud's logs need to be self-describing so that cleaning and recovery can interpret the raw contents of segments. Each log segment contains a collection of records of different types, such as objects and tombstones. It must be possible to identify all of the records in a segment and parse them using only information in that segment. To support this, RAMCloud adds a small piece of metadata to every *entry* appended to the log: an *entry header*. Each entry header consists of a type and length, and the actual content of each entry follows immediately after its entry header (see Figure 4.1). Headers are between 2 and 5 bytes long. The first

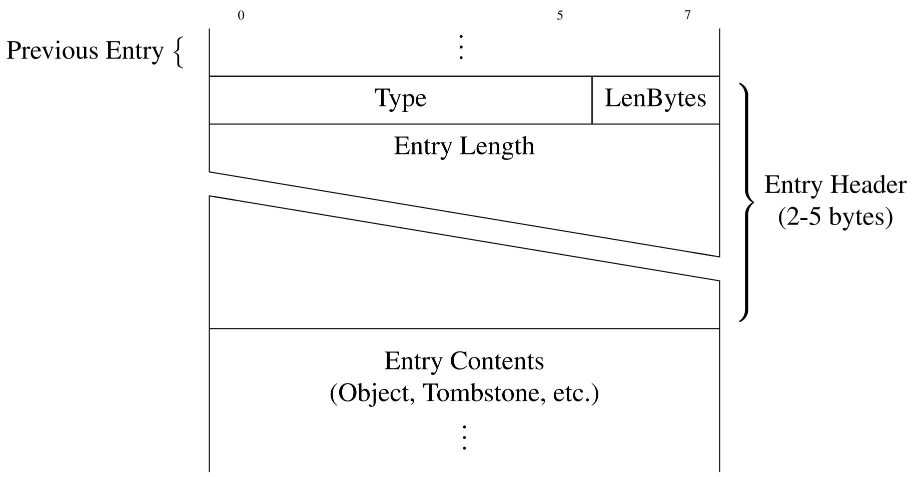


Figure 4.1: Bytes comprising the beginning of a log entry (the log is depicted growing down). Each entry in the log is preceded by an entry header with a variable-size length field. Every entry header begins with a common first byte that identifies the type of the entry (6 bits) and indicates the number of bytes needed to store the entry's length (2 bits; "LenBytes" above). Entries are often structured and contain their own type-specific headers as well (Figures 4.2 and 4.3 describe the format of object and tombstone entries). Entries are not aligned to any particular byte boundary; they are written starting at the next free byte in the segment.

byte of the header identifies the entry type (object, tombstone, etc.) and indicates how much space is needed to store the actual byte length of the entry. The subsequent 1 to 4 bytes contain the length. Since RAMCloud defaults to 8 MB segments and 1 MB objects, at most three bytes are currently used for the length (permitting a longer length field allows these limits to be raised in the future if needed).

A variable-length entry size field was chosen to minimise space overheads in the log, especially when entries are very small. This is particularly important given RAMCloud's focus on memory efficiency and the fact that we expect some important workloads to consist of small objects.

4.1.2 Reconstructing Indexes: Self-identifying Objects

Each object record in the log is self-identifying, containing its table identifier, key, and version number 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 volatile hash table that was lost when the object's old master server crashed. Figure 4.2 describes the format of objects as they are stored in RAMCloud logs. The overhead for storing each object is 26 bytes in the object's header, plus an additional 2-5 bytes in the entry header.

Since each object contains a 64-bit version number that monotonically increases as the object is over-written, RAMCloud does not need to replay the log in order to find the latest version of each object. In fact, the system exploits this property in a number of ways. For example, crash recovery will process whichever

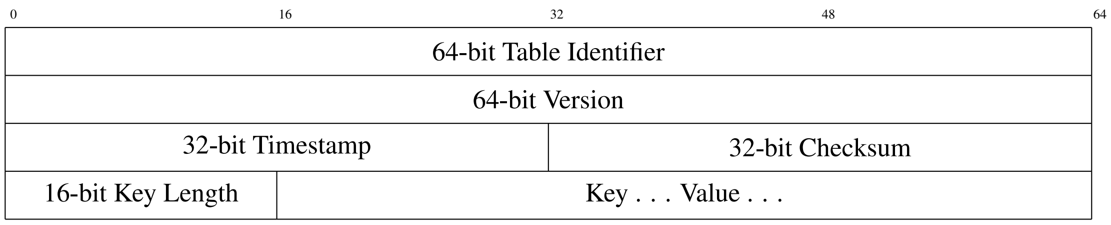


Figure 4.2: RAMCloud's object format. Each object record begins with a 26-byte header, which follows immediately after the generic entry header (Figure 4.1). An object's key and value follow directly after the object header. The "Key Length" field specifies how long the key is in bytes. The value length is not stored to save space (it can be computed using the total entry size in the entry header and the key length). The timestamp records when the object was written with one-second granularity. It is used by the cleaner to segregate older and newer objects to make future cleaning more efficient.

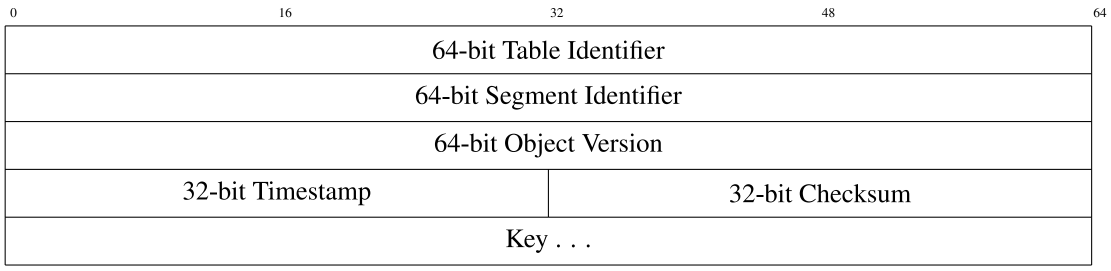


Figure 4.3: RAMCloud's tombstone format. Each tombstone record begins with a 32-byte header, followed immediately by the key of the deleted object. The key's length is not stored to save space; it is computed by subtracting the fixed tombstone header length from the length in the entry header (not depicted).

segments are fetched first to avoid any delays (the order in which segments are replayed has no effect on the correctness of the system).

4.1.3 Deleting Objects: Tombstones

Another important kind of log metadata identifies which objects have been deleted: *tombstones* (Figure 4.3). When an object is deleted or modified, RAMCloud does not modify the object's existing record in the log (the log is append-only). Instead, the object's master appends a tombstone record to the log. The tombstone contains the table identifier, key, and version number for the object that was deleted. Tombstones serve no purpose 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 are discussed later on. Given the issues with tombstones, I 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. I considered alternative approaches that use explicit metadata to keep track of live objects, but these approaches were complex and created new performance and consistency issues. I eventually decided that tombstones are the best of the available alternatives. Chapter 6 provides a more comprehensive discussion of tombstones and their alternatives.

4.1.4 Reconstructing Logs: Log Digests

RAMCloud uses another important piece of metadata to allow the system to find all of the segments belonging to a master's log: the *log digest*. In RAMCloud, each new log segment contains a *log digest* that describes the entire log. Every segment that a master creates has a unique 64-bit identifier, allocated in ascending order within a log. The log digest is nothing more than a list of identifiers for all the segments that currently belong to the log, including the current segment. Using the digest, RAMCloud can determine all of the segments that comprise a particular log on backup disks, despite the segments having been spread across many different backups as they were replicated.

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 most recently created segment, allows RAMCloud to find all of the live segments in a log. Identifying the most recently created head segment is fairly nuanced and beyond the scope of this work; see [61, 75] for details.

One potential problem with log digests is that their size scales with the capacity of the master server: the more segments a master's log consists of, the larger the digest will have to be. For example, if a server happens to have 8192 segments in memory (64 GB of data), then the corresponding log digest would be about 64 KB long (less than 1% of a segment). As memory capacities increase, however, the digest will grow larger, occupying a greater and greater percentage of each segment. Moreover, as we will see in Section 4.3, two-level cleaning exacerbates the problem because it allows segments in memory to be smaller than 8 MB (meaning there could be even more of them for any given amount of DRAM, and therefore even larger log digests).

Fortunately, this can be mitigated in several ways. First, and most importantly, the digest only needs to be present in the latest log segment (digests in older segments are obsolete). This means that the cleaner can discard old digests immediately and reclaim the space used to store them. Section 4.3 shows how two-level cleaning can remove old digests from memory without requiring any disk or network I/O.

A second way of mitigating large digests is to scale up the segment size proportionally so that digests do not occupy a larger percentage of each segment. For example, a 128 GB server could use 16 MB segments to ensure that its digests consume the same fraction of a segment that they would in a 64 GB server with 8 MB segments. This option may be unattractive, however, because it has wider-reaching consequences: backups will need larger non-volatile buffers for head segments, it will take longer to pull the first segment from disk and begin processing it during recovery, etc. Other options for dealing with large digests include compression or run-length encoding (most segment identifiers contain many 0 bits and are numerically close together).

The size of digests has not been a problem for RAMCloud thus far, but I expect that these (or similarly straightforward) solutions will suffice if it does become an issue in the future.

4.1.5 Atomically Updating and Safely Iterating Segments: Certificates

Every segment in RAMCloud has an associated *certificate*, which contains two important fields: a 32-bit integer that tracks how many bytes have been appended to the segment, and a 32-bit crc32c checksum [44]. The length defines how many bytes within the segment are valid and therefore what should be replayed during recovery. The checksum allows the segment's integrity to be tested.

RAMCloud uses certificates to solve two important problems. The first is allowing entries to be atomically appended to the log replicas on remote backups. This is crucial for object overwrites, for example, where a tombstone for the old object and the newer version of the object must either both be present in the log, or both be missing should a crash occur while the operation is in progress. Atomicity also ensures that partially-written entries are never encountered when replaying a segment. In other words, certificates are responsible for transitioning segments instantaneously from one set of complete entries to a larger set of complete entries, much like a database transaction. This simplifies recovery code because segments are either complete and therefore easy to parse, or they are invalid due to data corruption (random bit flips, for example) and thus are unusable. In the rare case of corruption, the system must locate other valid replicas.

Segment corruption is the second problem that certificates address. The checksum in each certificate makes it possible – with very high probability, at least – to verify that the corresponding segment has not been inadvertently modified due to hardware failures, cosmic rays, and so on. Ensuring the integrity of segments is particularly important in a large scale system such as RAMCloud, since even rare data corruption events are bound to happen eventually, if not fairly frequently [12, 71].

A certificate's checksum only covers the metadata fields within its segment (the entry headers described in Section 4.1.1), not the contents of the entries themselves (objects, tombstones, etc.). We chose this design for several reasons. Most importantly, entries are frequently processed or forwarded through RAMCloud outside of the segments they were written to. This requires that each entry have its own checksum that can be verified even when the record is not part of a log segment. For example, during crash recovery the objects and tombstones in many segments are shuffled to different recovery masters based on their tables and keys, rather than the segment in which they happen to exist. Having a checksum as part of the entry itself allows the recovery masters to perform an end-to-end integrity check. Furthermore, by leaving integrity up to the entries themselves, RAMCloud is free to choose checksums that are weaker or stronger, smaller or larger, depending on the size and importance of particular entries.

Certificates are maintained by masters and issued to backups. When a RAMCloud master has finished writing one or more entries to a segments' replicas and wishes to atomically apply them on the backups for that segment, it issues RPCs that instruct the backups to replace their previous certificate with a new one. Since there is only one certificate per segment, replacing the previous certificate with another serves as a commit point that instantaneously includes all data appended to the segment since the last certificate was issued. How the data was appended – in one RPC or many, in order or out of order, etc. – is unimportant so long as it is all present when the certificate is issued. For efficiency, masters will usually piggyback a new certificate along with a replication RPC if the RPC is small enough to include new entries in their entirety. In the case of very long appends that require multiple RPCs to backups, the certificate is included with the last RPC.

Backups durably store their certificates just as they do the segments themselves: they are first buffered in non-volatile memory, and then later flushed to disk. Importantly, backups keep only the latest certificate for each segment replica. In particular, there is one disk block adjacent to each segment on disk that is used to durably store the corresponding certificate. This ensures that the certificate itself can be updated atomically (hard disks ensure that blocks are either written in full or not at all in the event of a power or server failure) and that there is only one certificate per segment at any point in time. In practice, backups use this same disk block to record additional metadata, including a checksum of the entire block to detect corruption of the certificate and other metadata fields.

Certificates afford masters significant flexibility in how they append to segment replicas on backups and RAMCloud's replication module exploits this freedom in several ways. One important example is segment re-replication. When backup servers fail, masters must replace the segment replicas that were lost to restore the previous level of durability. However, it is important for these masters to ensure that each segment they are re-replicating cannot be used during recovery until all of the data for that segment has been written. Otherwise, data loss could occur if a partial replica were used during log replay because a master had failed part way through writing the replica's contents on the new backup. Certificates solve this problem by ensuring that new replicas are created atomically, even though transferring all of the data may have required many different RPCs. Masters need only delay issuing a segment's certificate until all of its data has been received by the backup to ensure this atomicity.

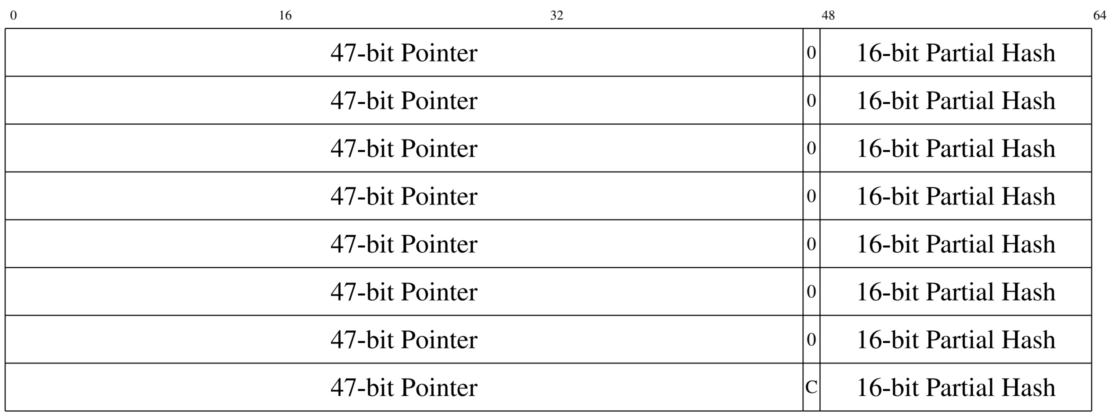


Figure 4.4: 64-byte hash table bucket. Each bucket is sized and aligned to a cache line. The hash table consists of an array of such buckets. Each $\langle tableId, key\rangle$ tuple is hashed into a particular bucket, which contains up to eight 47-bit direct pointers to objects within the master's in-memory log. Each pointer's upper 16 bits contains a partial hash of the object's $tableId$ and key, which allows RAMCloud to avoid reading from objects and performing key comparisons when there is no possibility of a match. The 48th bit is only used in the last 64-bit word of the bucket ("C"). If set to 1, it denotes that the corresponding 47-bit pointer points to another hash table bucket (of identical layout), rather than to an object in the log. This is used to chain buckets when they overflow to make room for more object pointers.

4.2 Volatile Metadata

In addition to persistent data, RAMCloud masters also maintain a number of structures and statistics that are not needed for crash recovery. This section describes the important volatile metadata that masters use to implement their log-structured storage. For example, the hash table need not be stored durably because it can be reconstructed using the information in objects and tombstones. Another example is segment statistics that the cleaner uses to make cleaning decisions. These, too, do not need to be saved, because they can be easily recomputed during recovery (in fact, they must be recomputed because recovered data is written out to entirely new segments during recovery).

4.2.1 Locating Objects: The Hash Table

The hash table is one of the most important data structures in a master server because it points to every live object in the server's memory. Objects are always accessed indirectly by table identifier and key, so every object operation interacts with the hash table at some point. A simple example is a read request from a client. To handle it, the master must use the hash table to locate the object in the log so that it can return the object's value. The log cleaner also makes extensive use of the hash table. Since an object in the log is alive if and only if the hash table points to it, the cleaner uses this data structure both to determine if objects are alive, and to update pointers when it relocates live objects.

The hash table also happens to be one of the simplest of the masters' data structures. The current implementation uses a straightforward closed addressing scheme in which a particular object can be pointed to by only one possible bucket in the table (Figure 4.4). If a bucket is full, chaining is used to store more object references in the same bucket (additional chained buckets are allocated separately on demand). The number of buckets is currently fixed; the user specifies this size when starting the master server. In the future the hash table will need to support dynamic resizing for better performance and memory efficiency.

Since the hash table is so widely and frequently used, it is important that it be fast. We could have chosen another data structure, such as a self-balancing tree, but a hash table allows RAMCloud to service most lookups with $O(1)$ steps and, more importantly, with two cache misses (one miss in the hash table bucket, and another to verify that the table identifier and key match the object). Reducing the number of cache misses is important since we expect the hash table to be much larger than the processor's cache. RAMCloud servers will have tens or hundreds of gigabytes of DRAM and objects are likely to be fairly small (100 to 1,000 bytes), so the hash table will refer to tens or hundreds of millions of individual objects. This means that the hash table working set is likely to be very large, and some cache misses will be unavoidable.

RAMCloud's hash table features a few simple optimisations to reduce cache misses as much as possible. First, each hash table bucket is actually one CPU cache line in size (64 bytes on current x86 processors, or eight pointers to objects). The reason for this is that accessing a bucket will often result in a cache miss, which loads a full cache line from memory. It is therefore worth considering all of the references that the loaded cache line contains when doing a lookup, since the latency of the cache miss has already been paid for.

The second optimisation is complementary. To avoid accessing each object referenced in the bucket in order to check for a match (that is, to compare the table id and key stored within the object, which would likely incur a cache miss), RAMCloud uses the upper 16 bits of each hash table pointer to store a partial hash of the table id and key of the object referred to. This way, if a bucket contains references to several objects, with high probability at most one will have a matching partial hash in the bucket, so only one object will need to be accessed to compare the table id and key. The remaining 48 bits are sufficient to address the object within the master's in-memory log and indicate whether or not the bucket is chained due to overflow (see Figure 4.4).

RAMCloud's hash table is commonly accessed by multiple threads simultaneously. For example, read, write, and delete requests may be processed at the same time on different cores, each of which needs to look up and sometimes modify the hash table. The log cleaner also runs concurrently with regular requests and must read the hash table for each object it encounters to determine if it is alive. If so, it relocates the object and updates the hash table to point to its new location. Fortunately, hash tables are amenable to fairly straightforward support for concurrent operation. Section 4.4.2 explains how RAMCloud currently addresses this by using fine-grained locking of buckets.

4.2.2 Keeping Track of Live and Dead Entries: Segment Statistics

RAMCloud masters maintain statistics that track the amount of space consumed by live entries and dead entries in each segment on a per-entry-type basis. For example, every segment S has two counters for objects in the segment: $totalObjectBytes$ and $deadObjectBytes$. The former keeps track of the total amount of space used by object entries, whether dead or alive, while the latter tracks the amount of space consumed by dead objects. The counters monotonically increase when objects are written to the segment, and when they are recorded as deleted (or overwritten), respectively. There are similar pairs of counters for every other entry type as well, including, of course, tombstones.

These statistics are not maintained in the segments themselves, but are soft-state kept only on the master in per-segment metadata structures. In the actual implementation, each log segment is represented by an instance of the $Segment$ class, which encapsulates the formatting of log segments. RAMCloud masters primarily interact with a subclass of $Segment$, called $LogSegment$, which includes the segment statistics, along with other important metadata used to track the segment's use by the master server.

The primary reason for these counters is to compute the utilisations of segments. That is, the fraction of a segment currently being used to store live data. Chapter 7 explains in depth how RAMCloud's log cleaner uses segment utilisation to choose the best segments to clean. Doing so simply requires aggregating the per-entry counters just described. Individual counters are most commonly used for external analysis of the system, and so that RAMCloud masters may identify segments with large numbers of tombstones.

Maintaining statistics for tombstones is more complicated than for other entries. The problem is that tombstones become obsolete when the objects they refer to are removed from the on-disk log during cleaning. In effect, they die implicitly (there is no explicit notification that they are no longer needed). Unfortunately, it is impossible to determine which tombstones have died as a result of cleaning a segment without storing more metadata to keep track of the relationship between each deleted object and its tombstone (Chapter 6 explains why this is in detail), so the counters that track deleted tombstones cannot be updated.

Early implementations of RAMCloud punted on this problem and simply did not keep track of dead tombstones. Unfortunately, this led to deadlocks: sometimes the system would generate segments full of tombstones that would never be cleaned because, according to the segments' statistics, they were full of live data. Even when not suffering from this particular pathology, the log cleaner's efficiency was hampered because it could not select segments optimally without precise utilisation statistics. As a result, RAMCloud now occasionally scans segments in memory to determine which live tombstones have become obsolete due to cleaning. Masters choose a segment to scan next by taking into account two pieces of information from each segment: the fraction of its space consumed by tombstones not yet known to be dead, and the amount of time since the segment was last scanned. The segment with the highest product of these values is the best candidate. This way, RAMCloud prefers segments that it estimates to have the most newly obsolete tombstones.

The current policy for deciding when to scan segments is very simplistic: for every N segments processed by the log cleaner, a master will scan the next most promising segment for dead tombstones. The current value of $N = 5$ was empirically chosen because it provided quick resolution of the deadlock problem with only marginal additional overhead. Importantly, by tying the rate of scanning to the rate of cleaning, this overhead is only incurred in proportion to the amount of cleaning required by the write workload of the master server. Furthermore, RAMCloud's cleaner helps to reduce future scanning overheads by relocating tombstones to the front of segments, so that much of the time only tombstones (and no other types of entries) need to be iterated during these scans. I suspect that better scanning policies could lower overheads even further, but have left this to future work.

While most entries remain alive until explicitly deleted or overwritten, some entries, such as log digests, are always obsolete by the time the cleaner gets to them. For example, by the time the cleaner encounters a log digest, it is guaranteed that a newer digest was already written to the log (the segment being cleaned cannot be the head segment, so a newer head segment must exist that contains a newer log digest). In such cases, the corresponding entry counters are never incremented when the entry is appended.

4.3 Two-level Cleaning

Almost all of the overhead for log-structured memory is due to cleaning. Allocating new storage is easy; the log just appends new objects to 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 2.3. Unfortunately, the cost of log cleaning rises rapidly as memory utilisation approaches 100% (see Figure 4.5). 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% utilisation, the cleaner must copy 9 bytes of live data for every byte freed. As memory utilisation rises, eventually the system will run out of bandwidth and write throughput will be limited by the speed of cleaner. Techniques like cost-benefit segment selection [69] (see Chapter 7) help by skewing the distribution of free space, so that segments chosen for cleaning have lower utilisation than the overall average, but they cannot eliminate the fundamental trade-off between memory utilisation and cleaning cost. Any copying storage allocator will suffer from intolerable overheads as memory utilisation 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 utilisation 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 utilisation the write bandwidth to disk created a severe limit on write throughput (see Chapter 8 for measurements).

On the other hand, we could have improved write bandwidth by adding more disk space to reduce its average utilisation. For example, at 50% disk utilisation 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 utilisation of disk space, we would also have reduced the

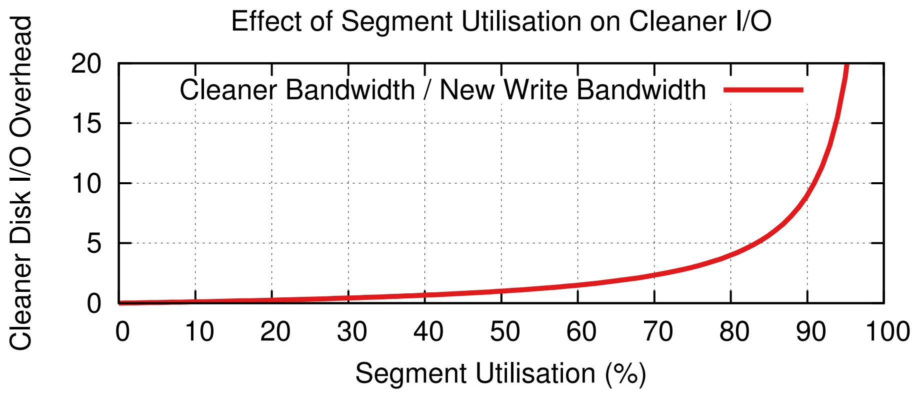


Figure 4.5: The amount of disk I/O used by the cleaner for every byte of reclaimed space increases sharply as segments are cleaned at higher utilisation.

utilisation 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 utilisation than disk. The cleaning cost for memory will be high, but DRAM can easily provide the bandwidth required to clean at 90% utilisation or higher. Disk cleaning happens less often. The disk log becomes larger than the in-memory log, so it has lower overall utilisation, 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 have been removed (see Figure 4.6).

The second level of cleaning is just the mechanism described in Section 2.3. 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 utilisation on disk 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 utilisation 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 utilisation. For disk, space is plentiful but bandwidth is precious, so we use extra space to save bandwidth.

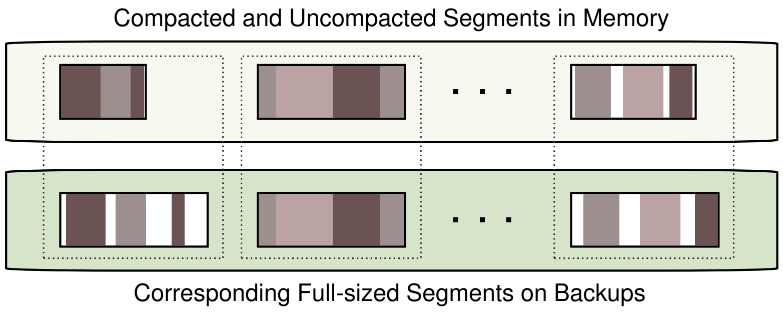

Compacted and Uncompacted Segments in Memory
. . .
. . .
Corresponding Full-sized Segments on Backups

Figure 4.6: 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 utilisation of memory is higher than that of disk, and disk can be cleaned more efficiently.

4.3.1 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. This meant that our original segment allocator, which was designed only to allocate fixed-size 8 MB blocks of memory from a single free list, was not up to the task. One possible solution was to use a standard heap allocator to allocate segments, but this would have resulted in the same sort of fragmentation problems described in Section 3.2. Instead, each RAMCloud master divides its 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 (depending on how much it has been compacted). In particular, the C++ $Segment$ class that encapsulates the formatting of RAMCloud's segments now also maintains an array of pointers to the constituent seglets used to store its data.

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 64 KB seglet size this fragmentation represents about 1% of memory space (in practice we expect compacted segments to average at least half the length of a full-size segment – $\geq$ 4 MB). Furthermore, even the worst case of wasting nearly a full seglet is a significant improvement. Previously, with fixed 8 MB segments, it was possible to waste nearly 1 MB of space at the end of each segment (for example, if the last attempted write to that segment was a maximum-size 1 MB object that narrowly missed fitting in the available space, causing a new segment to be allocated).

The second problem with seglets is discontiguity: 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). I solved this problem by introducing an additional layer of software that hides the discontiguous nature of segments. The abstraction is analogous to *mbufs* [56] and *skbuffs* [17] in the networking stacks of the BSD and Linux kernels, respectively. RAMCloud's requirements for low latency made the addition of a new layer cause for

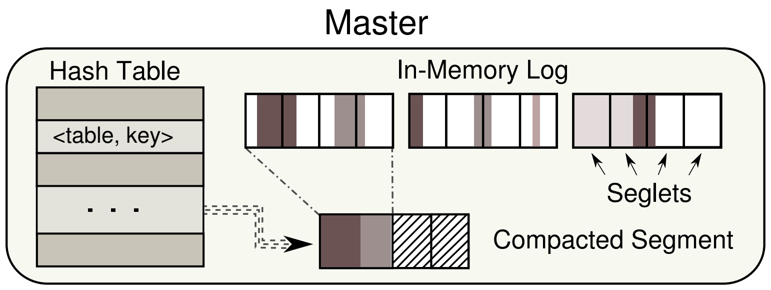


Figure 4.7: Segment compaction after live data has been relocated, but before the original segment has been freed. Segments are composed of discontiguous fixed-sized blocks called seglets. RAMCloud is able to reuse space freed during compaction by freeing the unused seglets at the tail of the newly compacted segment (hatched above) and allocating them to new head segments. For simplicity, each segment is depicted with 4 seglets; however, in the current version of RAMCloud, segments actually consist of 128 64 KB seglets.

concern (in particular, our goal is for masters to handle simple requests end-to-end in less than 1μs). However, I found that this layer is relatively efficient, especially in the common case where objects do not cross seglet boundaries (with 64 KB seglets, about 0.2% of 100-byte objects would be stored discontiguously).

I considered several other ways of handling the discontiguity of seglets. My first implementation of segments, for example, used *mmap* to map discontiguous seglets into a contiguous range of virtual addresses, thereby exploiting the processors' MMU to avoid the need for an extra software layer. Unfortunately, this approach introduced a number of problems. First, manipulating memory mappings from user space required expensive system calls[1], and constantly setting up and tearing down the mappings during cleaning added overhead for TLB misses and shootdowns. Second, the 64-bit x86 architecture only supports pages that are too small to use efficiently (4 KB), or too large to be useful without very large segments (2 MB or 1 GB). Third, use of smaller pages in the log would prevent RAMCloud from statically mapping log memory with 1 GB superpages, which make much more efficient use of the TLB and reduce access times (RAMCloud does not currently use 1 GB superpages, but I did not want to rule out the possibility). Finally, constantly changing address space mappings makes using high-performance NICs more complicated, since they maintain their own address translation tables in order to copy data directly to/from userspace. As a result of these issues, I eventually abandoned the *mmap* approach in favour of an extra software layer.

If an object is contiguous, the hash table only needs to store a single pointer into the in-memory log, but what happens if the object spans multiple seglets? Rather than storing pointers to each fragment of the object in the hash table, RAMCloud stores one pointer to the beginning of the object's log entry. Every time an object is looked up, the master must check if the object is stored contiguously. To do so, the master reads the object's length from the entry header in the log (Section 4.1.1) and determines if the object extends past the end of the seglet that the object began in. This is easy because all seglets are 64 KB and begin at 64 KB boundaries, so the starting offset of the object within the first seglet can be computed from the pointer. If the

[1]Userspace page mapping can be made faster with hardware virtualisation (see Dune [16]), but this does not solve the other problems.

object is contiguous, as is commonly the case, the pointer may be used directly.

If the object is not contiguous, then the master must figure out which additional seglets the object is contained in. This requires first finding the in-memory segment that the object was written to and then using its list of constituent seglets to locate each fragment of the object. To accommodate this, RAMCloud masters maintain a table that records the segment that each seglet currently belongs to. Once the master knows which segment the object resides in, it uses the segment's table of seglets to determine which other seglets the object spans.

4.3.2 Choosing Segments to Clean

Both segment compaction and combined cleaning require a policy to select which segment or segments to process. RAMCloud chooses segments to compact with a simple greedy method: it selects the segment with the best ratio of free space to live space. In other words, the cleaner compacts the segment that requires copying the least amount of live data for each byte that it can free. Combined cleaning, on the other hand, uses a more sophisticated selection method based on the cost-benefit policy introduced in LFS [69]. Chapter 7 describes the combined cleaner's policy in detail.

4.3.3 Reasons for Doing Combined Cleaning

The main function of compaction and combined cleaning are the same: to reclaim free space in the log. It might seem that combined cleaning is superfluous, and worse, less efficient because it consumes backup I/O. However, compaction alone is not sufficient. There are three important reasons why combined cleaning is needed as well.

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 utilisation 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 reorganisation. When the combined cleaner runs, it cleans several segments at once and reorganises 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 by segregating them. However, segment compaction cannot reorganise data, since it must preserve the one-to-one mapping between segments in memory and those on disk.

We considered allowing segment compaction to reorganise data into new segments, but that would have created complex dependencies between the segments in memory and those on disk. Essentially, this would allow the logs in memory and on backups to diverge, breaking the one-to-one correspondence that allows RAMCloud to clean on disk without reading the contents on backups. While it might be possible to maintain metadata in memory that would allow the master to locate the live entries of a particular on-disk segment after its contents had been scattered in main memory, we rejected this idea due to its complexity.

These reasons mean that RAMCloud must have a policy module that determines when to run compaction and when to run combined cleaning instead. We call this the *balancer*. Designing a balancer that achieves high performance under a wide range of workloads and system configurations is a challenge. Chapter 5 discusses RAMCloud's balancer policy in detail and describes why it was chosen.

4.4 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 simultaneous cleaner threads 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.

4.4.1 Concurrent Log Updates with Side Logs

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. Contention not only reduces bandwidth (cleaning and regular writes would go to the same set of backups for the head segment), but also increases client write latency. Optimally, RAMCloud should process both cleaner and client writes simultaneously.

RAMCloud's solution is for the cleaner to write survivor data to a different log – a *side log* – that is atomically fused to the masters' log after cleaning has completed. Each cleaner thread allocates a separate set of segments for its survivor data. Synchronisation is only required when allocating segments; once segments are allocated, each cleaner thread can copy data to its own survivor segments without additional synchronisation. Meanwhile, request-processing threads can write new data to the log head (see Figure 4.8). Once a cleaner thread has finished a cleaning pass (that is, it has written all of the live data it encountered to survivor segments), it arranges for its survivor segments to be included in the next log digest. This new log digest effectively inserts the side log segments into the main log, atomically fusing the two together. The cleaner also arranges for the cleaned segments to be dropped from this next log digest so that the disk space used by them on backups may be freed. This step requires additional synchronisation, which is discussed in Section 4.4.4.

Using a separate side log for survivor data has the additional benefit that the replicas for survivor segments will likely 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.

Another benefit of side logs is that their appends are batched before being sent to backups, rather than being issued synchronously, one at a time (as with regular log writes). Since a side log's data will not be recoverable until it is fused with the main log, it is only important that the side log's segments are fully replicated before fusing – replication of a side log's segments may proceed asynchronously until that point. This increases bandwidth to backups because the master issues fewer and larger RPCs. In fact, with side logs the cleaner both batches and pipelines survivor segment replication: the data in each survivor segment is not replicated until the segment is filled, at which point replication proceeds while the cleaner fills the next survivor segment.

In contrast, client writes to the master's log are always synchronous – each write operation requires RPCs to be issued to backups to ensure durability. If the cleaner were to write to the head segment instead, each client write would force all previously-committed survivor data to backups. This would not only reduce the bandwidth of the cleaner, but also increase the latency of any client writes that had to sync survivor data in addition to their own data.

Side logs have another important use case in RAMCloud: during crash recovery, masters use side logs to store replayed segment data. When a master server fails, portions of its log are replayed by many other master

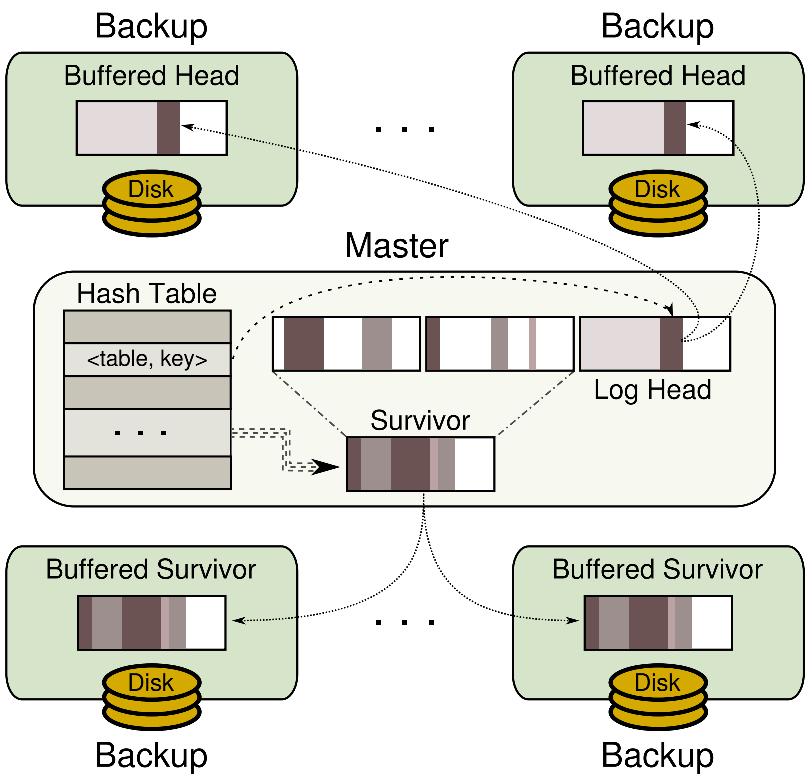


Figure 4.8: Master server and backups during parallel cleaning. A write request is being appended to the head of the log (top) while the cleaner relocates the live data from a few segments into a new survivor segment (bottom). As the cleaner copies objects, it updates the hash table to point to their 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.

servers in parallel to restore availability. These recovery masters must ensure the durability of their newly-inherited objects, so each recovered object must be written to its new master's log. Therefore, fast replication of the recovery masters' logs is crucial for restoring availability quickly (RAMCloud was designed to do so within 1 to 2 seconds). Side logs help to maximise replication bandwidth in two ways. First, unlike the regular log, segments in side logs may be replicated in any order (only when they have all been replicated is the side log fused into the main log). Second, just like with cleaning, side logs avoid contending with client writes. This makes it possible for masters to handle write requests and take part in recovery simultaneously.

4.4.2 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. RAMCloud uses locks to ensure hash table consistency. In order to reduce lock contention, RAMCloud currently uses many fine-grained locks that protect relatively small ranges of 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. One avenue for future improvement would be to use a lockless approach to eliminate this overhead.

4.4.3 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 are unable to access 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.

One possible 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. Shared locks are especially undesirable in such hot paths because even if the lock is available, they will still result in cache coherency overhead and cache lines being ping-ponged between cores.

Instead, RAMCloud uses a mechanism based on *epochs*, which avoids locking in service threads[2]. 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, only in-progress requests with older epochs can 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.

The current epoch implementation is simple, consisting of a single 64-bit integer to track the epoch number, and a doubly-linked list of incoming RPCs being (or waiting to be) processed by the master. A lock protects the linked list when RPCs are added or removed, and when the list is searched for the minimum epoch number; an atomic increment instruction is used to increment the epoch value. This simple concurrency control works well because the list is infrequently searched. The most common operation is adding and removing RPCs from the list, but this is performed by a single thread responsible for dispatching RPCs to service threads for processing. Moreover, the lock used to protect the epoch list is already held by the

[2]Epochs are similar to Tornado/K42's *generations* [9] and RCU's [55] *wait-for-readers* primitive.

dispatching thread while it runs, as the lock is used to protect a number of other important data structures.

4.4.4 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: after each segment is filled, it is handed off to the replication module for transmission to backups. 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.

4.5 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 utilisation. Furthermore, it delays cleaning as long as possible in order to minimise 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 maximising memory utilisation.

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 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 reorganises 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 4.9).

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

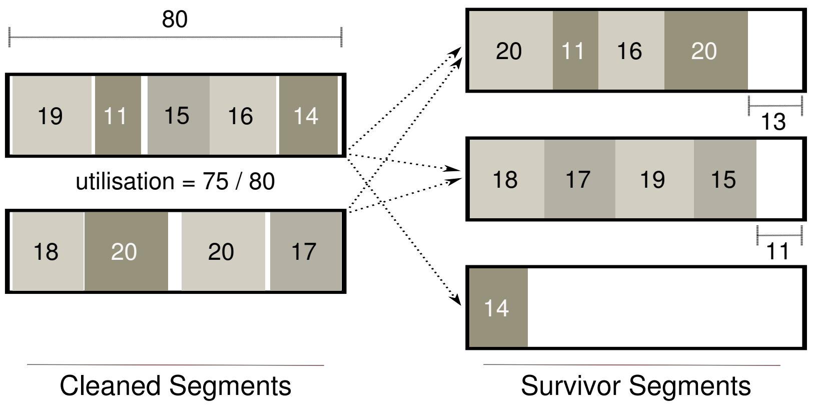


Figure 4.9: A simplified example in which cleaning uses more space than it frees. Two 80-byte segments at about 94% utilisation are cleaned: their objects are reordered by age (not depicted) and written to survivor segments. The label in each object indicates its size. Because of fragmentation, the last object (size 14) overflows into a third survivor segment.

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 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.

To ensure that F is always large enough, in RAMCloud we conservatively require a segment to have at least 2% free memory for it to be cleaned, which for a full segment would be a bit more than two seglets. The reasoning is that each cleaned segment can never result in more than two survivor segments, each of which could waste just under one seglet's worth of space at most due to fragmentation. RAMCloud's compactor is careful to ensure that at least 2% free space remains in each segment on the master, even after compaction, in order to avoid later deadlock when the combined cleaner runs. Free space need not be artificially preserved in new head segments, or survivor segments as they are created; the system simply must not perform combined cleaning on them while over 98% utilisation. This 2% 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 because they are never the head of the log). 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 memory for 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 memory for 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 utilisation as high as 98%. When utilisation reaches this limit, no new data (or tombstones) can be appended to the log until the cleaner has freed space. However, RAMCloud sets a slightly lower utilisation 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.

4.6 Summary

This chapter explained how RAMCloud implements log-structured memory and introduced the mechanisms it uses to efficiently manage it (two-level cleaning and parallel cleaning). RAMCloud has a concise set of persistent metadata it needs to ensure that logs can be recovered when masters crash (objects, log digests, etc.). However, it also has important soft-state that can be easily recomputed when logs are replayed (the hash table and segment statistics). RAMCloud uses log digests, side logs, and its concurrent hash table to perform cleaning in parallel with regular operation, thereby minimising the performance impact of cleaning. Furthermore, by slightly decoupling the in-memory and on-disk layouts, RAMCloud's segment compaction is able to reduce cleaning's network and disk I/O overheads. As Chapter 8 will show, this allows RAMCloud masters to make use of 80-90% of their DRAM while offering excellent write performance.

The next four chapters build upon topics presented in this chapter. Chapter 5 explains in depth why striking a good balance between memory compaction and combined cleaning is important and how RAMCloud implements its policy. Chapter 6 explains tombstones in greater depth, including how they are used during recovery, some of their shortcomings and complexities, and alternatives I considered. Chapter 7 explains the cost-benefit formula that RAMCloud's combined cleaner uses when selecting segments and how it differs from what was used in Sprite LFS. Finally, Chapter 8 evaluates the performance of log-structured memory in RAMCloud with various synthetic micro- and macrobenchmarks.

Chapter 5

Balancing Memory and Disk Cleaning

With two distinct cleaners, RAMCloud must decide when and how much of each to run. This chapter explores the problem of scheduling – or *balancing* – memory compaction and combined cleaning in order to get the best write throughput out of the system. As we will see, striking a good balance is critical for high-performance; a poor balance can lead to throughput losses of up to 80%. An added complication is that the optimum balance is not static; it changes depending on both the workload (whether objects are large or small and how long they live, for example) and system configuration (how much backup bandwidth is available and how much memory is free in each server, for instance).

This chapter explores the balancing problem by analysing the behaviour of two different cleaner scheduling policies (*balancers*) across thousands of combinations of workloads and configurations that were run against RAMCloud. The experiments show that although neither balancer is optimal in all cases, it is possible to configure a balancer that achieves at least 80% of maximum throughput in nearly 95% of cases (especially those we expect to be most typical) and that has only moderate pathologies in rare cases (the worst case performance penalty is about 35%).

The chapter is structured as follows:

1. Section 5.1 introduces, describes, and motivates the two balancing strategies that I implemented in RAMCloud.

2. Section 5.2 explains the experiments that I conducted to evaluate the two balancers. The rest of the chapter analyses the data generated by these experiments.

3. Section 5.3 explores the overall performance of the two balancers across thousands of different workloads and system configurations.

4. Section 5.4 takes the most promising parameters for the two balancers and explores how they perform under the most likely access patterns and system configurations.

5. Section 5.5 takes the best two balancers from Section 5.4 and explores the pathological cases in which they under-perform.

6. Section 5.6 summarises the chapter.

5.1 Two Candidate Balancers

This section introduces the two balancers that are implemented in RAMCloud. First, I discuss the logic that both balancers share in common. For instance, in some cases the balancer has no flexibility in making decisions; there is only one choice regardless of the policy. Using too much disk space is one example, since only the combined cleaner can reclaim space on disk. Then I describe how each balancer decides between memory compaction and combined cleaning when there is a choice between the two.

5.1.1 Common Policies for all Balancers

Both balancers share three common policy components. The first determines when free space is running so low that cleaning must occur. 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. Deciding when to clean is independent of which cleaner to use. The second common component determines when the on-disk log has grown too large. If it has, the system has no choice but to do combined cleaning in order to avoid running out of disk space on backups and to keep crash recovery fast (the size of the on-disk log directly affects recovery speed since the entire log must be replayed). The third common component determines if compaction is not yielding any free space (because too many tombstones have accumulated). In particular, if the last memory compaction failed to free any seglets, then the combined cleaner must be run. This ensures that the system will not waste resources doing useless compactions.

RAMCloud determines that memory is running low as follows. Let L be the fraction of all memory (including unallocated seglets) 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 (rather than unallocated seglets). This formula represents a trade-off: on the one hand, it delays cleaning to make it more efficient; on the other hand, it starts cleaning a little early so that the cleaner has a chance to collect free memory before the system completely runs out of unallocated seglets.

Given that the cleaner must run, the balancer needs to choose which cleaner to employ. In two cases mentioned above, however, there is no choice to be made. First, if the on-disk log grows too large, then the combined cleaner must be run to reduce its size. RAMCloud implements a configurable *disk expansion factor* that sets the maximum on-disk log size as a multiple of the in-memory log size. With either balancer, the combined cleaner runs when the on-disk log size exceeds 90% of this limit. The second case is simply when the memory compactor fails to free any seglets.

When free memory is low and the on-disk log is not too large, the system is free to choose either memory compaction or combined cleaning. The next two subsections describe two different balancing policies that I evaluated.

5.1.2 Fixed Duty Cycle Balancer

The first balancer I implemented is very simple: it sets a *fixed duty cycle* for the combined cleaner. In other words, the balance is static: the policy restricts the combined cleaner thread to spending at most *combinedDutyCycle%* of its elapsed run time doing combined cleaning; the remaining *100-combinedDutyCycle%* of the time is spent running the memory compactor. For example, by setting *combinedDutyCycle* to 70%, 70% of the combined cleaner thread's run time would be spent doing combined cleaning, and 30% would be spent doing memory compaction. The desired *combinedDutyCycle* value is statically set when a RAMCloud master starts up.

The idea behind this algorithm was to have a simple mechanism to generate a baseline against which I could compare other approaches. As we will see later on, the optimal value for *combinedDutyCycle* depends on many variables, including both the client workload and how the system is configured. By trying different values of *combinedDutyCycle* with a fixed workload and system configuration, we can experimentally determine what balance works best in each particular case. I explore this in detail beginning with Section 5.2. One unanticipated result was that some moderate *combinedDutyCycle* settings do very well in large numbers of cases.

In the current implementation of RAMCloud, at most one thread is permitted to perform combined cleaning (the reasoning being that one combined cleaner can generate a substantial amount of I/O traffic, so it only makes sense to use additional threads to try to reduce this traffic via memory compaction). Therefore, when the duty cycle of the combined cleaner is discussed, this refers to the duty cycle of the single thread permitted to run the combined cleaner, not taking into account the amount of time spent compacting in the other cleaner threads.

5.1.3 Adaptive Tombstone Balancer

The second balancer I implemented tries to vary the combined cleaner's duty cycle in order to get the best balance of memory compaction and combined cleaning regardless of the workload or system configuration. This *adaptive tombstone balancer* runs compaction unless too many tombstones have accumulated in memory (as defined below), in which case it runs the combined cleaner.

The reason that this balancer uses tombstones to decide when to clean is that they directly affect the efficiency of compaction. The problem with tombstones is that memory compaction alone cannot remove them: the combined cleaner must first remove dead objects from disk before their corresponding tombstones can be erased. If only the memory compactor is run, tombstones continue to pile up as objects are deleted or overwritten; as a result, segment utilisations increase. This causes compaction to become more expensive and less efficient over time. If left unchecked, tombstones would eventually eat up all free memory. Running the combined cleaner ensures that tombstones do not exhaust memory and makes future compactions more efficient again.

This balancer determines when too many tombstones have accumulated as follows. Let T be the fraction of memory occupied by tombstones that are believed to be alive (as explained in Section 4.2.2, RAMCloud does not always perfectly account for live and dead tombstones), and L be the fraction of memory consumed by live objects (as in 5.1.1). Too many tombstones have accumulated once $T/(1 - L) \geq maxTombstoneRatio\%$. In other words, there are too many tombstones when they account for *maxTombstoneRatio%* of the freeable space in a master ($1 - L$; i.e., all tombstones, dead objects, and unallocated seglets).

My hypothesis was that defining a fixed threshold for when there are too many tombstones (*maxTombstoneRatio%*) would work better across a large number of access patterns and system configurations than a fixed duty cycle. For example, when objects are small, tombstones correspond to a relatively large proportion of the freeable space (they are nearly the size of the objects they delete) and have a more adverse impact on the efficiency of memory compaction. This suggests that the combined cleaner should run more frequently to remove them and make memory compaction more efficient. For workloads with larger objects, however, tombstones will constitute a smaller fraction of reclaimable space, and so the combined cleaner should need to run less frequently (because memory compaction will not become bogged down by tombstones as quickly). A fixed duty cycle is unlikely to work optimally in both cases, but the hunch was that a tombstone threshold might result in a more effective balance.

The rest of this chapter evaluates the effect of different *combinedDutyCycle* and *maxTombstoneRatio* parameters for the fixed duty cycle and adaptive tombstone balancers, respectively, across many different workloads and configurations.

5.2 Experimental Setup

A good balancer should provide near-optimal throughput across a wide range of access patterns and system configurations. In order to evaluate the balancers introduced in Section 5.1, I ran over 3000 experiments on sets of three servers running RAMCloud (see Table 8.1 in Chapter 8 for details on the hardware used). I varied a number of parameters in order to exercise the balancers in many different cases:

Memory Utilisation

Experiments were run with 30%, 50%, 70%, and 90% of the master servers' in-DRAM log being used to store live objects. The higher the memory utilisation, the more work the cleaners must do to reclaim free space. In particular, as memory utilisation increases, conserving disk and network bandwidth with memory compaction becomes more important (because otherwise the combined cleaner could generate a large volume of write traffic to survivor segments, reducing the bandwidth available for new writes).

Object Size

Each experiment used a fixed object size: either 100 bytes, 1,000 bytes, or 10,000 bytes per object. Smaller objects are more difficult to clean because of per-object overheads (for example, looking up in the hash table to determine if an object is alive, and changing the hash table's pointer when objects are deleted or overwritten). Tombstones are also more problematic for smaller objects because the tombstone's size is the same regardless of the size of the object deleted. This means that for each overwrite of a 100-byte object, a new tombstone that is about one third of the obsolete object's size is added to the log. Effectively, only about two thirds of deleted object space can be reclaimed without running the combined cleaner to make their tombstones obsolete as well. However, for each deleted 10,000-byte object, the corresponding tombstone is nearly 300x smaller – insignificant, by comparison. Therefore, with large objects nearly all of an object's space can be reclaimed by compaction alone.

Object Lifetimes (Access Locality)

Experiments were run with three different access patterns to simulate different object popularity distributions. The "uniform" access pattern overwrote objects uniform-randomly (that is, each object had an equal chance of being overwritten) and represented a pathological case of zero locality. The "zipfian" pattern issued writes according to a Zipfian probability distribution in which 90% of writes were made to 15% of the objects. Finally, the "hot-and-cold" distribution split the objects into two sets: the first contained 90% of objects and received 10% of the writes, while the second contained 10% of the objects and received 90% of the writes (objects were selected uniformly within each set).

Backup Bandwidth

Experiments were run with 100 MB/s, 250 MB/s, 500 MB/s and 1 GB/s of aggregate backup bandwidth available to each master. The amount of bandwidth available to write to backups has an important effect on the balancing policy: the less bandwidth there is, the more important using memory compaction becomes in order to preserve as much as possible for new object writes to head segments.

Balancer Parameters

The fixed proportion balancer was run with 11 different duty cycles for the combined cleaner thread: 0% to 100% in increments of 10. The tombstone ratio balancer was also run with 11 different tombstone thresholds: 0% through 100%, in 10% increments. With a 0% duty cycle, the combined cleaner only ran when forced to, either because the log had grown too large on disk, or because memory compaction failed to free any seglets. A 100% tombstone ratio had the same effect, since a 100% ratio of tombstones cannot be attained in practice.

The experiments each required three servers: one to run the coordinator and a client benchmark program, one to run the master server, and another to run three backup servers. Master servers were configured to use up to two CPU cores for cleaning (at most one running the combined cleaner, and up to two running compaction). In order to ensure that each master was maximally stressed, the client was configured to issue 75 writes per RPC and pipelined 10 multi-write RPCs at a time (that is, up to 10 RPCs were in-flight to the master). To ensure a fixed amount of backup bandwidth, each backup was configured to write its segments to */dev/null*, and throttled its writes as needed. For example, if configured to provide 100 MB/s of aggregate bandwidth, each of the three backups would write at most 33 MB/s to */dev/null*. Avoiding actual disks both removed a potential source of performance inconsistency and allowed me to run all three backup servers on a single node while providing up to 1 GB/s of simulated backup bandwidth (much more than the machines' physical disks could provide).

Each experiment began by filling the single master up to the desired memory utilisation with objects named by unique 8-byte numerical keys. Once the desired threshold was reached, the experiment continuously overwrote these objects according to the chosen access distribution. The experiments terminated when the cost of cleaning stabilised (the ratio of live data copied by the combined cleaner to space freed was stable to two decimal places for 30 seconds of combined cleaner run time).

5.3 Performance Across All Workloads

This section analyses the experimental data by looking at all of the workloads at once. In doing so, we will see how the various balancers perform across the full spectrum of workloads and get an idea of how frequently they perform well, and how poorly they do in the worst cases. After first seeing all of the data together, I slice into it by looking at different system configurations – memory utilisations and available backup bandwidth – to get an idea of how these operational parameters affect balancing and write throughput. Each balancer was run with 11 different parameters (values of *combinedDutyCycle* or *maxTombstoneRatio*) against 144 unique workloads and system configurations, for a total of 3,168 experiments.

5.3.1 Is One of the Balancers Optimal?

A reasonable first question to ask is whether one of the balancers is already optimal, that is: does one of them provide the best throughput in every situation? Figure 5.1 shows that this is not the case. The graph depicts the performance of the fixed duty cycle and adaptive tombstone balancers across all of the experiments. Performance is defined as follows. For each combination of memory utilisation, object size, access locality, and amount of backup bandwidth, I looked for the fixed duty cycle that gave the highest throughput and called this optimal performance for that combination. I then looked at all of the balancer configurations (type of balancer and the duty cycle or tombstone threshold parameter) for this same combination, and computed the ratio of each ones' throughput to the optimal throughput. The ratio represents how close (or how far) each balancer came to getting the best performance in that scenario. The figure aggregates these ratios for each balancer into a cumulative distribution.

There are several important lessons to take away from Figure 5.1. First, none of the adaptive tombstone balancers represented is optimal: each one suffers from pathological cases in which it provides throughput

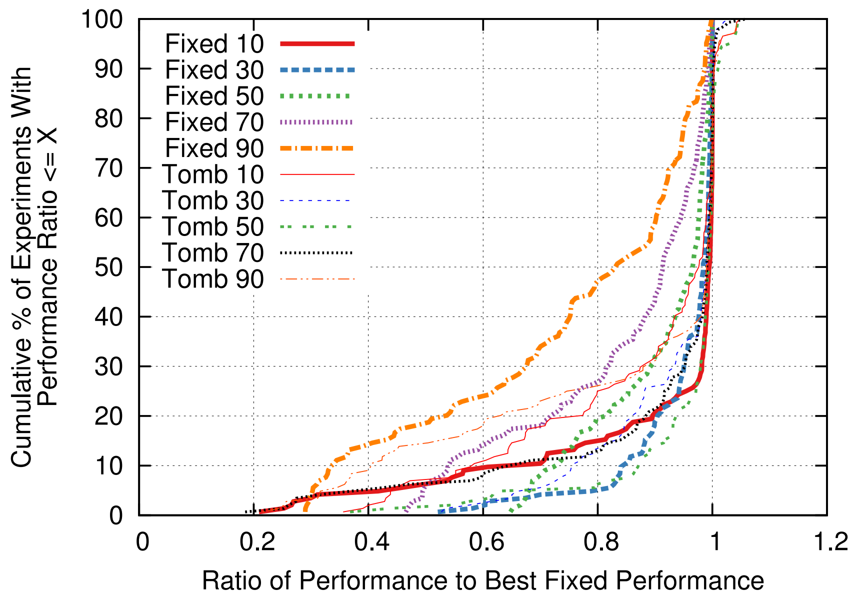


Figure 5.1: Comparison of 11 different balancer configurations each run on 144 combinations of memory utilisation, object size, access locality, and backup bandwidth as detailed in Section 5.2. Each curve shows how close each balancer is to achieving optimal throughput through all of the experiments. For example, the "Fixed 10" curve shows how well the fixed duty cycle balancer performed when set to run the combined cleaner 10% of the time. For about 80% of the experiments this cleaner is within 10% of optimal (an optimal cleaner would be depicted by a straight vertical line at $X = 1$). However, the curve also shows that for about 15% of cases this balancer was worse than the best fixed duty cycle by 20% of more (in the most pathological case it provides 5x worse throughput than the best fixed balancer). Note that in a small number of cases, the adaptive tombstone balancer exceeds "optimal" due to the fact that not all fixed duty cycles were tested. In other words, these are cases in which a fixed multiple-of-10% duty cycle (0%, 10%, 20%, etc.) was not the best choice, and the adaptive tombstone balancer managed to strike a better balance in-between (e.g., 35% or 73%, perhaps).

that is lower than optimal by 45% or more. Second, although there is no optimal fixed duty cycle, using a moderate duty cycle of around 30-50% is a surprisingly safe choice. For example, with a 50% duty cycle, the fixed balancer never does more than about 35% worse than optimal, and 80% of the time it is within 20% of the best performance. My expectation from the outset was that any fixed duty cycle would suffer significant pathologies in some cases, but this does not appear to be true.

What Figure 5.1 does not show, however, is how important the individual cases are (they are all weighted the same and are unordered). This can be misleading. One might be inclined to choose a 30-50% fixed duty cycle because they have the best worst case performance (and are therefore a safe and conservative choice). However, if the worst cases happen to coincide with the most common workloads and configurations of the system, then it might be better to choose a balancer that has poorer worst-case pathologies in unlikely situations, but performs better under expected conditions. Section 5.4 digs more deeply into this data and explores performance under typical workloads.

5.3.2 The Importance of Memory Utilisation

As Chapter 4.3 explained, memory utilisation – the fraction of DRAM log space occupied by live objects – factors significantly into the cost of cleaning. As the fraction of live objects increases, the cleaning overheads rise exponentially. This is the fact that motivated RAMCloud's two-level cleaning approach.

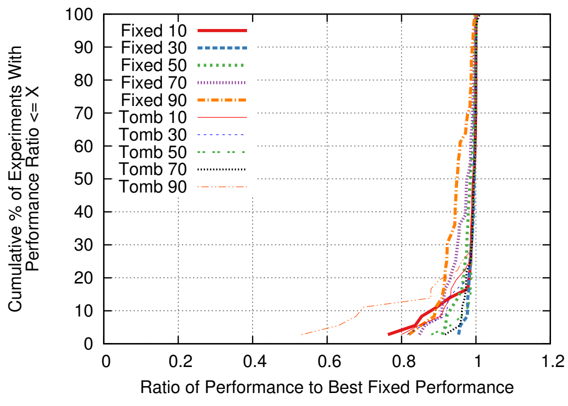
In light of the importance of memory utilisation, another interesting question to ask is, how do the balancers that I tested perform across varying memory utilisations? Figure 5.2 addresses this question by looking at subsets of the data in Figure 5.1. It separately depicts results from experiments that ran the master servers at 30%, 50%, 70%, and 90% memory utilisation.

Most notably, Figure 5.2 shows that there are balancers well-suited to all of the workloads and configurations tested up through 70% memory utilisation. For instance, a fixed 30% duty cycle achieves at least 90% optimal throughput in all cases with 70% or less memory occupied by live objects. This means that if RAMCloud servers are run at about three-fourths capacity, then a single balancer can be chosen that provides high-performance without any known pathologies.

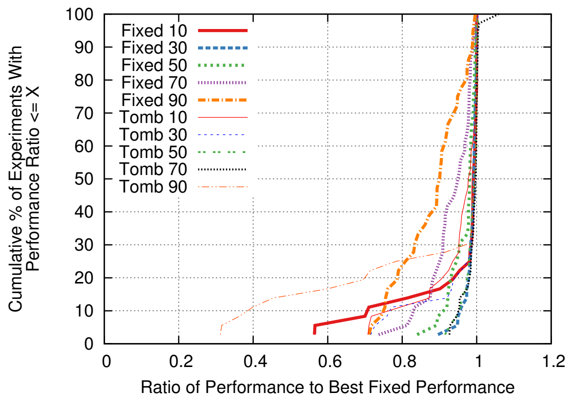
However, as memory utilisation increases to 90% (which more than doubles the overhead of cleaning), the pathologies become markedly worse. By comparing Figure 5.1 and Figure 5.2(d), it is apparent that the balancers' worst-case performance occurs at very high memory utilisation. In particular, no balancer is able to avoid a 35% or greater performance penalty in some instances, and a majority of the balancers suffer penalties of 50% or more (up to nearly 80% at worst).

5.3.3 Effect of Available Backup Bandwidth

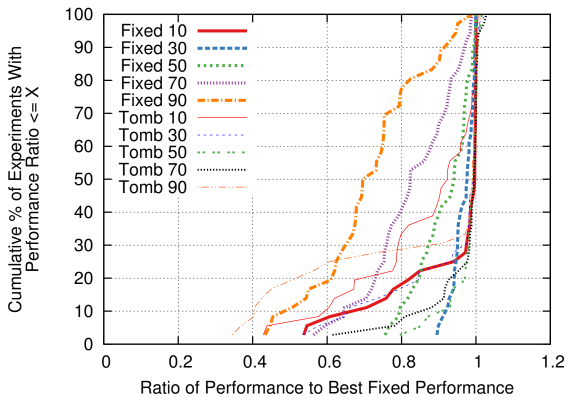
The previous subsection explored the effect of changing memory utilisation; this subsection explores how changing the amount of backup bandwidth available to a master affects the balancers.



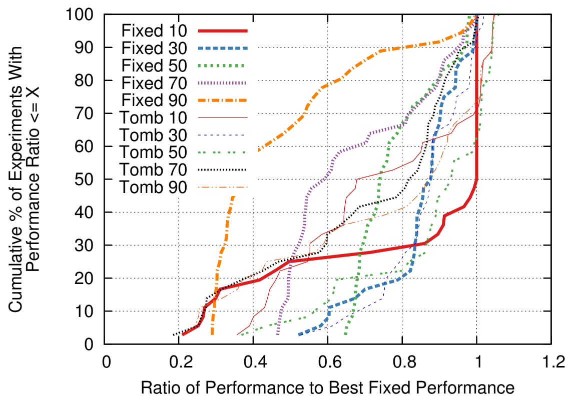
(a) 30% Utilisation



(b) 50% Utilisation



(c) 70% Utilisation



(d) 90% Utilisation

Figure 5.2: Comparison of the same balancer configurations in Figure 5.1, but at specific memory utilisations. For example, (a) depicts the performance of each balancer across all experiments in which 30% of the masters' log was occupied by live data. At low memory utilisations, the overhead of cleaning is small so the choice of balancer is less important. At higher memory utilisations, however, choosing the right balancer is crucial. For instance, (d) clearly shows that either too much or too little combined cleaning will adversely affect a large number of workloads.

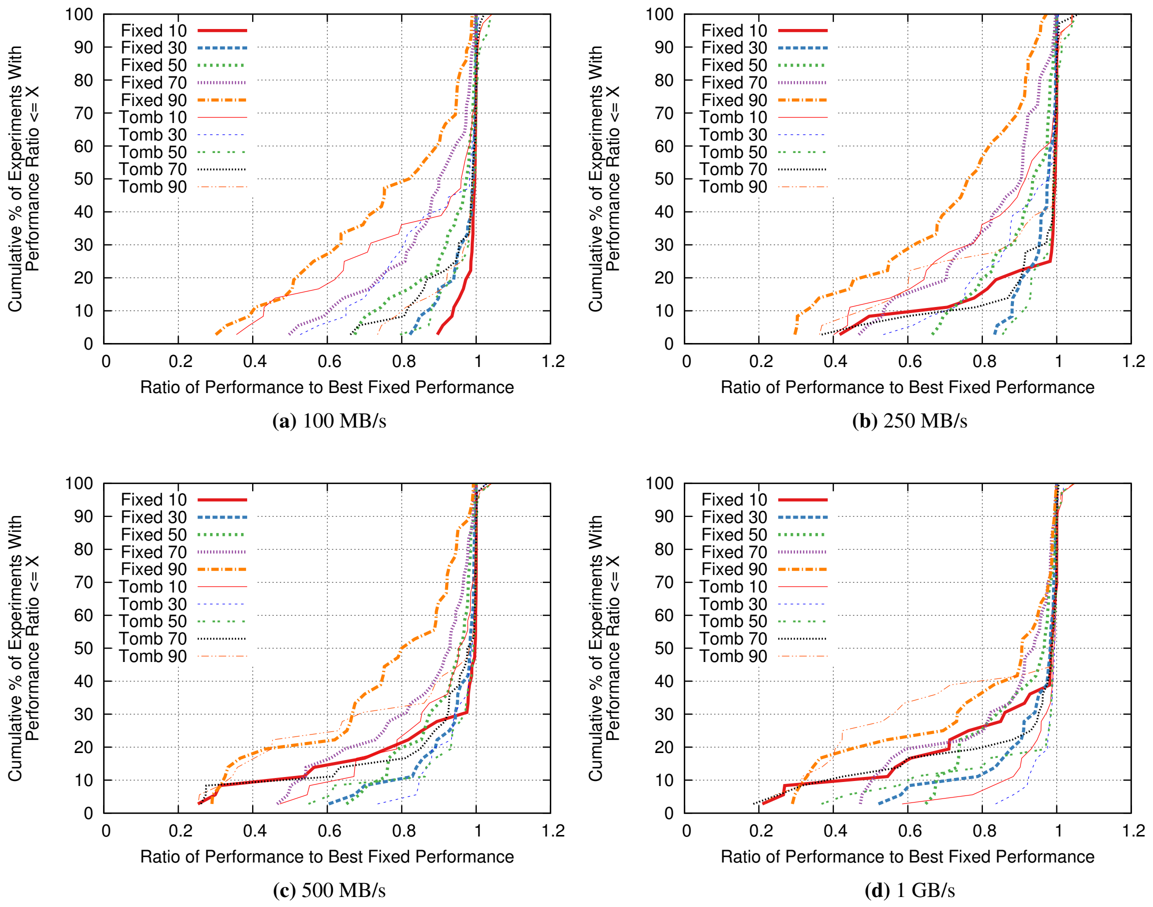
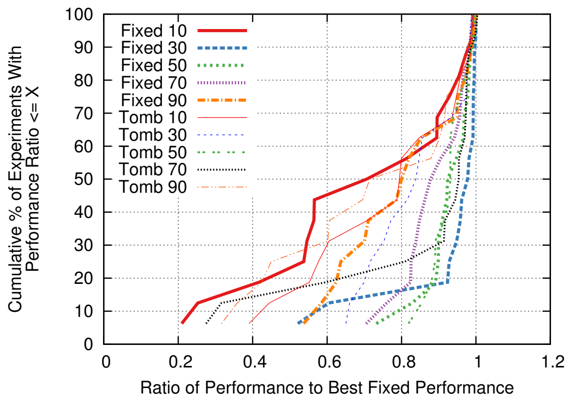
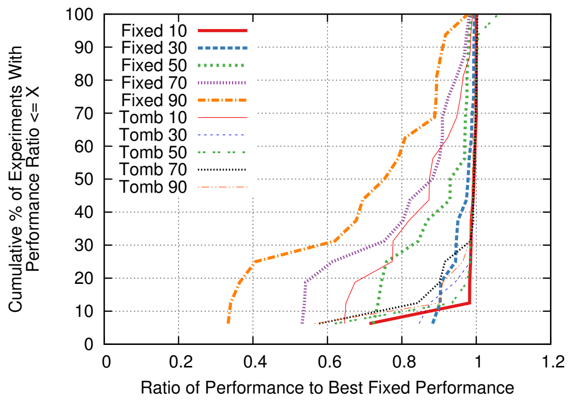


Figure 5.3: Comparison of the same balancer configurations in Figure 5.1, but with varying amounts of available backup bandwidth.

Analysing the effect of backup bandwidth on the balancers is important because we would like RAMCloud to perform well independent of how much bandwidth it has at its disposal. This is particularly important for two reasons. First, we do not know how many disks each server in a RAMCloud cluster will have, nor how much bandwidth each disk will provide (if solid state disks are used, for example, they may provide many times the throughput of a single hard disk). Second, since RAMCloud masters will have many shared backups at their disposal, the amount of free bandwidth available to a master may vary greatly depending on how much other masters in the system are writing to backups. Optimally, RAMCloud's balancer would perform well across a wide range of available backup bandwidths, allowing the system to adapt to different configurations (more or fewer backup disks, for example) and varying amounts of contention for backup bandwidth between masters.



(a) 100-byte Objects



(b) 1000-byte Objects

Figure 5.4: Comparison of balancers with experiments using the Zipfian access pattern and either 100-byte or 1000-byte objects. This data includes experiments at all tested memory utilisations.

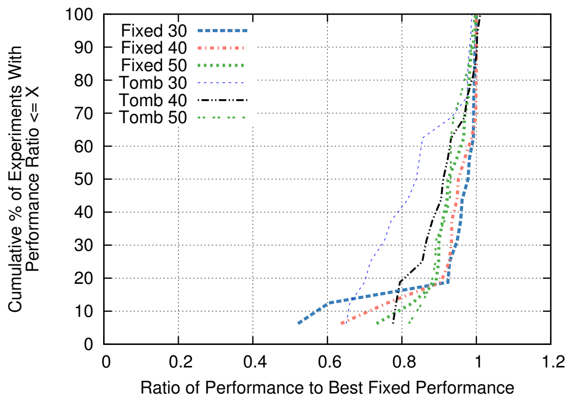
Figure 5.3 takes another view into the results depicted in Figure 5.1, but this time divides the experiments into those given 100 MB/s, 250 MB/s, 500 MB/s, or 1 GB/s of aggregate backup bandwidth. The main lesson is that although there is no single optimal balancer across all cases, a few perform consistently well most of the time. For example, in each case a tombstone ratio of 50% provides throughput that is at least 80% of optimal over 80% of the time.

5.4 Performance Under Expected Workloads

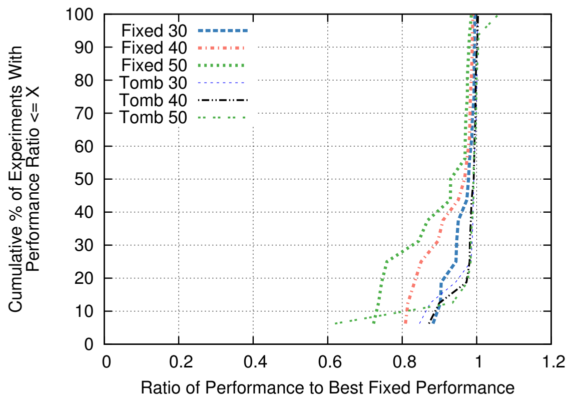
So far we have looked at how the balancers perform under all workloads; however, we expect that many of those workloads are atypical. For example, given what has been reported in previous studies of datacentre and local storage systems [10, 70, 67], we can expect a typical RAMCloud workload to have relatively small objects (1 KB or less) and for objects to have greatly varying lifetimes. This suggests that we should optimise more for smaller objects and workloads with reasonable locality; providing the same level of performance with large objects and low or no access locality is less likely to be important in practice. Consequently, this section analyses balancer performance with 100 and 1,000-byte objects accessed with the Zipfian distribution.

Figure 5.4 shows that there is a small subset of viable balancers and parameters. The graphs depict experiments using the Zipfian access pattern and 100 or 1000-byte objects. For smaller objects, a 50% fixed duty cycle or tombstone ratio provides consistently good throughput (about 90% of optimal in most cases) with minimal pathologies. With 1000-byte objects, however, a 30% duty cycle or tombstone ratio provides the best balance of average and worst case throughput.

Fortunately, by choosing a parameter between 30 and 50% we can combine much of the benefit of either



(a) 100-byte Objects

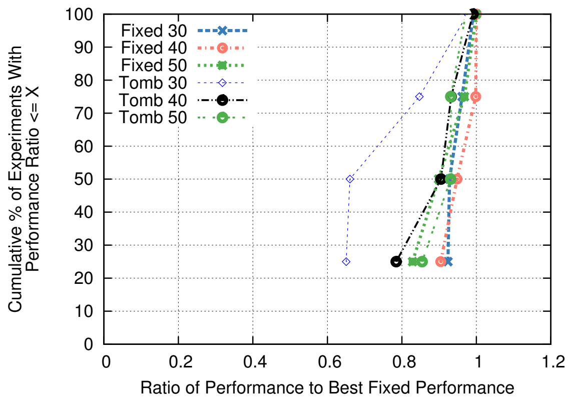


(b) 1000-byte Objects

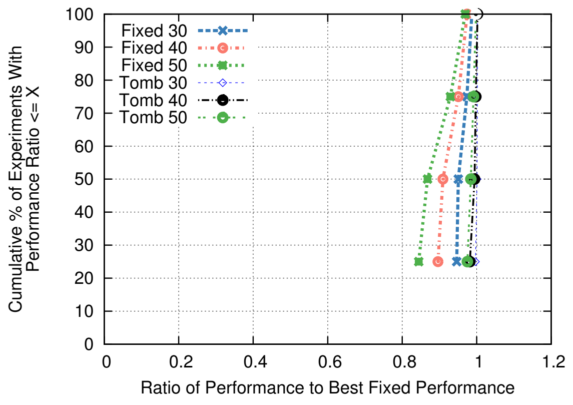
Figure 5.5: Subset of Figure 5.4 that also includes 40% duty cycle and tombstone ratio experiments. For each type of balancer, the 30% and 50% parameters work well with one object size but not the other (30% is better for larger objects, but 50% is better for smaller ones). Fortunately, taking a value in between the two avoids most of the pathologies and provides very good throughput (within 10% of the better of the 30% and 50% parameters in every case).

Balancer	Avg	Min	Max	Stddev
Fixed 40%	93.4%	63.6%	100.0%	8.5%
Tomb 40%	95.1%	63.4%	109.3%	8.3%

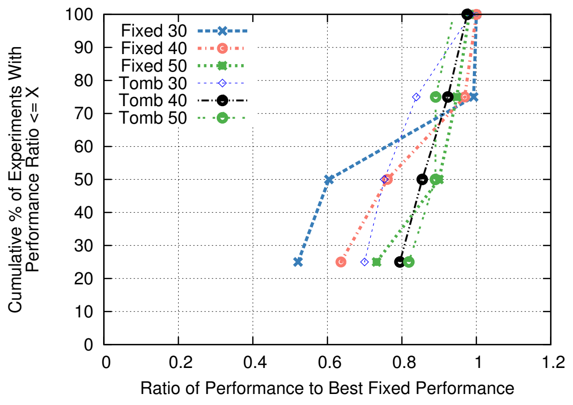
Table 5.1: Throughput optimality of the most promising two parameters for each balancer across all experiments conducted. A 40% parameter provides a good balance of high average throughput and minimal pathologies for both the fixed duty cycle and adaptive tombstone balancers.



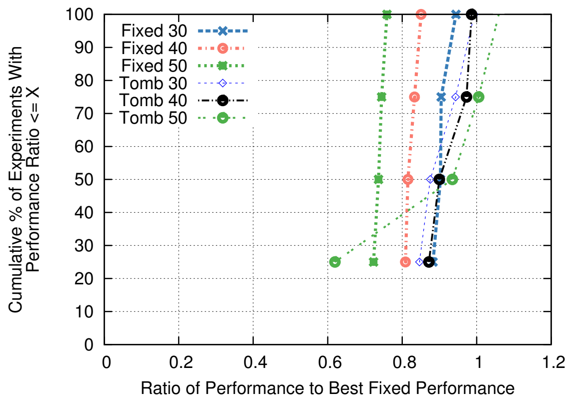
(a) 100-byte Objects, 70% Utilisation



(b) 1000-byte Objects, 70% Utilisation



(c) 100-byte Objects, 90% Utilisation



(d) 1000-byte Objects, 90% Utilisation

Figure 5.6: Subset of the data depicted in Figure 5.4, focusing on higher memory utilisations (70% and 90%). At high utilisations (particularly so at 90%), the 40% tombstone ratio balancer is a very good choice. These graphs also show how even a small parameter change has a substantial effect on the worst case performance of both the fixed duty cycle and tombstone ratio balancers. For example, the four big outliers in (a) and (c) are due to insufficient compaction (the combined cleaner slows writing down by consuming too much bandwidth). Conversely, the sole outlier on the "Tomb 50" curve in (d) is due to compacting too much (there is enough bandwidth left over that the combined cleaner should be run more frequently). Changing the duty cycle or ratio parameter by 10% avoids all of these pathologies.

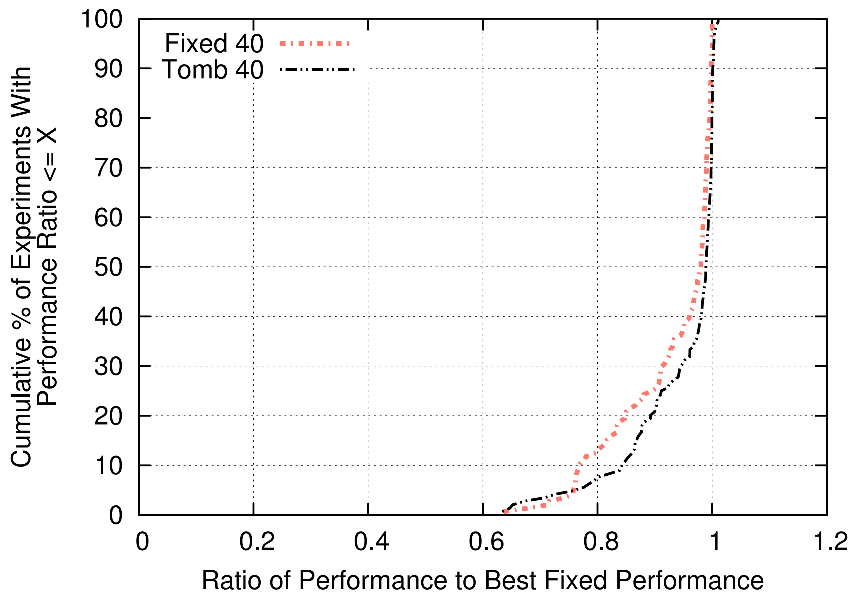


Figure 5.7: Optimality of the the fixed 40% duty cycle and 40% tombstone ratio balancers across all experiments.

and avoid significant pathologies. Figure 5.5 plots the same 30% and 50% curves, but adds experiments with a 40% duty cycle or tombstone ratio. Figure 5.6 depicts the same experiments at 70% and 90% utilisation (one of the primary goals of log-structured memory is to allow RAMCloud masters to be run at high memory utilisation, so good throughput under memory pressure is an especially desirable quality in a balancer). The tombstone balancer does particularly well. In the cases depicted, it averages 94.7% of optimal throughput and at worst suffers a 21.5% performance penalty (for 1000-byte objects the worst is is only 12.8% less than optimal). Figure 5.7 and Table 5.1 show that a 40% tombstone ratio also works well across all of the experiments, averaging 95% of optimal throughput overall, with a minimum of 63.6% in the worst case.

5.5 Pathologies of the Best Performing Balancers

This final section looks into the pathological cases where the 40% fixed duty cycle and 40% tombstone ratio balancers perform poorly. As we saw in Figure 5.7 and Table 5.1, both balancers perform very well overall, but throughput can degrade by up to about 36% in certain cases. By understanding each balancer's weaknesses we can estimate how likely we are to see them in practice (are they typical or atypical workloads and configurations?) as well as cast light on areas that might be worth considering for the design and implementation of future balancers.

Table 5.2 summarises the worst pathologies for the two balancers (cases in which throughput was $\leq$ 80%

Experiments with Throughput $\leq$ 80% of Optimal	Fixed 40	Tomb 40	Combined
Overall	18 (12.5%)	9 (6.3%)	27 (9.4%)
With 90% Memory Utilisation	18 (12.5%)	7 (4.9%)	25 (8.7%)
With 70% Memory Utilisation	0 (0%)	2 (1.4%)	2 (0.7%)
With 100 MB/s Backup Bandwidth	10 (6.9%)	5 (3.5%)	15 (5.2%)
With 250 MB/s Backup Bandwidth	3 (2.1%)	3 (2.1%)	6 (2.1%)
With 500 MB/s Backup Bandwidth	5 (3.5%)	1 (0.7%)	6 (2.1%)
With 1 GB/s Backup Bandwidth	7 (4.9%)	2 (1.4%)	9 (3.2%)
With 100-byte Objects	6 (4.2%)	6 (4.2%)	12 (4.2%)
With 1000-byte Objects	5 (3.5%)	3 (2.1%)	8 (2.8%)
With 10000-byte Objects	7 (4.9%)	0 (0%)	7 (2.4%)
With Zipfian Accesses	3 (2.1%)	2 (1.4%)	5 (1.7%)
With Hot-and-Cold Accesses	7 (4.9%)	2 (1.4%)	9 (3.2%)
With Uniform Accesses	8 (5.6%)	5 (3.5%)	12 (4.2%)

Table 5.2: Breakdown of the cases in which the 40% duty cycle balancer and 40% tombstone ratio balancer achieve less than or equal to 80% of optimal throughput.

of optimal). There are several interesting things to note from the table. First, the tombstone ratio balancer has half as many sub-80% pathologies as the fixed duty cycle balancer (the opposite is true below 75%, however). Second, all pathologies occur at memory utilisations greater than or equal to 70%, and predominantly at 90% (this is consistent with Figure 5.2). Third, the tombstone balancer's pathologies are dominated by small object (100-byte) cases, whereas the fixed balancer has trouble across all object sizes. Fourth, the majority of pathologies occur when locality is reduced or non-existent (only about 20% of the pathologies coincided with Zipfian workloads).

Importantly, of the tombstone balancer's very worst pathologies, none of them coincide with expected workloads. Table 5.3 lists the ten worst pathologies from the data in Table 5.2. Of them, a Zipfian workload occurred only once (and with the fixed balancer). The majority of pathologies were due to a uniform access pattern and/or small objects, both cases in which we expect the cleaning to be especially burdensome.

Overall, the pathologies of both balancers are moderate and occur under low locality workloads, which we expect to be atypical. The tombstone balancer is especially good in this respect, and has half as many cases in which it suffers a 20% or greater performance penalty than the fixed duty cycle balancer.

Balancer	Distribution	Object Size	Memory Utilisation	Backup Bandwidth	Ratio
Tomb 40	Uniform	1000	90	1 GB/s	0.634
Fixed 40	Zipfian	100	90	1 GB/s	0.636
Tomb 40	Hot-and-Cold	1000	90	1 GB/s	0.648
Tomb 40	Hot-and-Cold	1000	90	500 MB/s	0.652
Tomb 40	Uniform	100	90	100 MB/s	0.673
Fixed 40	Uniform	100	90	1 GB/s	0.678
Tomb 40	Uniform	100	70	100 MB/s	0.707
Fixed 40	Hot-and-Cold	100	90	1 GB/s	0.716
Tomb 40	Uniform	100	70	100 MB/s	0.726
Fixed 40	Uniform	10000	90	500 MB/s	0.741

Table 5.3: Ten worst experiments with the 40% fixed duty cycle and 40% tombstone ratio balancers. Most of the experiments used small objects, low access locality, high memory utilisation, and very high backup bandwidth.

5.6 Summary

This chapter introduced the problem of balancing memory compaction and combined cleaning to optimise system throughput. The crux of the problem is that while memory compaction can save I/O bandwidth for useful writes, the amount of compaction needed to obtain the highest throughput depends on a number of factors related to the client workload as well as the system configuration. Achieving a good balance is paramount; the difference in throughput between a good strategy and a bad one can be up to 5x.

I implemented two balancing policies in RAMCloud to explore how they behave across the large workload and configuration space. The first uses a fixed CPU duty cycle for the combined cleaner. The second tries to select an appropriate duty cycle based on the accumulation of tombstones in the log. This balancer uses the intuition that tombstones adversely affect compaction performance because they are essentially additional live data (until the combined cleaner is run). I then ran about 3,000 experiments to evaluate both balancers under many different workloads, system configurations, and parameters (duty cycle and tombstone threshold percentages).

The results showed that neither balancer could avoid moderate (35% or more) throughput pathologies in rare situations, but that under realistic workloads these pathologies are much less significant (about 20% in the worst case). Overall, the adaptive tombstone balancer with a 40% tombstone ratio provided an average throughput within 5% of optimal across all experiments. Due to its high consistency and minimal pathologies, especially under workloads with realistic levels of locality, it is currently the default balancer in RAMCloud.

Although there appears to be only modest room for improvement over the current best balancers, the fact that performance is so sensitive to a magic constant (the duty cycle or tombstone ratio) is unfortunate. The empirical data in this chapter suggest that a good value can be found, but whether it will change considerably in the future as RAMCloud evolves and hardware changes is unclear.

Chapter 6

Tombstones and Object Delete Consistency

Ensuring that deleted objects remain deleted is one of the more intricate and subtle problems in RAMCloud's log-structured memory. Although removing object references from a master server's hash table ensures that the particular server will no longer return them to clients, this does not change the state of durable replicas of the log on backups. In particular, simply removing from the hash table would not prevent the system from resurrecting dead objects when the log is replayed during crash recovery. RAMCloud needs additional mechanisms to ensure that deleted objects are never brought back to life when crash recovery is invoked. This chapter explores several ways of ensuring delete consistency of objects in the face of master server recoveries, as well as additional problems that surfaced while introducing a mechanism to migrate tablets between servers.

The approach we chose to handle object deletions in RAMCloud uses special records – called *tombstones* – that masters add to their logs every time an object is deleted or overwritten. As we will see, tombstones have a number of downsides. One example is that the system must use space to free space (to store tombstones for what has been deleted). Another downside is that tombstones complicate cleaning because it is difficult to track which tombstones are dead and eligible for cleaning, and they make compaction more expensive, because they cannot be removed without first running the combined cleaner. In the end, we decided to stay with tombstones because the alternatives were either complex in their own right, or imposed additional design restrictions and performance trade-offs.

6.1 Tombstones

RAMCloud avoids resurrecting dead objects by appending tombstones to the log whenever objects are deleted or overwritten. Each tombstone records the death of a particular version of an object so that it will not be restored if the log is replayed during recovery. In order to identify the deleted object, a tombstone records the table the object existed in, as well as the object's key and version number (see Figure 4.3 for the precise tombstone format).

6.1.1 Crash Recovery

When crash recovery restores a failed server's data, it must be sure to discard any objects that have an associated tombstone in the log (not doing so could cause formerly deleted objects to come back to life). The recovery system ensures that each tombstone is replayed on the same master that would encounter the corresponding object, assuming it had not already been cleaned out of the log. However, because recovery masters replay log entries in arbitrary, first-come first-served order (for maximum recovery performance), this means that a tombstone may be replayed before the corresponding object. If a deleted object is replayed before the tombstone, then the object can simply be expunged from the hash table when the tombstone is finally processed. However, if the tombstone arrives first, the master must keep enough state to ensure that it will drop the corresponding object.

To accommodate this out-of-order replay, RAMCloud masters use their hash tables to track tombstones during recovery, in addition to their objects (in normal operation, the hash table only tracks objects). More precisely, during replay recovery masters use their hash tables to track the latest record for each object, which may be either an object or a tombstone. As the server replays each object and tombstone, it checks to see if the hash table already refers to a newer version of the object or a tombstone for the same or newer version. If so, the current entry being replayed is obsolete and can be discarded without storing it in the log; if not, it is appended to the log and the hash table is updated to point to it. This way objects and tombstones may be replayed in any order, allowing masters to process whatever data backups have loaded from disk first.

After recovery completes, the recovery masters no longer need to look tombstones up in their hash tables, so the table is scanned and purged of all tombstone references. Eventually, the masters' log cleaners purge the actual tombstone contents from their logs (but only when it is safe to do so, meaning that the corresponding deleted objects are also not present in the log).

6.1.2 Cleaning

A tombstone can only be cleaned once the associated object's segment has been cleaned, meaning that the object is no longer in the durable log on backups and will never be replayed during recovery. In order to determine if the object's segment has been cleaned, each tombstone in RAMCloud includes the identifier of the segment that contained the deleted object. This makes testing the liveness of a tombstone straightforward: if the segment specified by a tombstone still exists in the log, then so does the object, therefore the tombstone must be kept; otherwise, if the object's segment is no longer in the log, then the object will never be replayed and the tombstone may be discarded. Checking this on master servers translates into a simple lookup from segment identifier to active segment: if found, the tombstone is alive; otherwise it is dead. Each master maintains a small hash table (distinct from the hash table used to look up objects) to make these lookups fast.

Ensuring that there is a simple one-to-one mapping between deleted objects and tombstones simplifies deciding whether a tombstone is dead or not. We initially thought that tombstones could be avoided when objects are overwritten, since the existence of the new object version would override any previous ones. This would save log space since a tombstone would only need to be written when the last object were deleted. The problem with this scheme is that a single tombstone would mark all previous versions of the object in the log as deleted, so the cleaner could only reclaim the tombstone after all of the objects it marked as deleted were cleaned from the log.

For example, suppose a master creates a new object O. Since it is a brand new object, its version number starts at 1 (let us refer to this specific version of the object as O_1). Now assume that the master overwrites the object with O_2, and later deletes it. In the one-to-one scheme where each deleted or overwritten object has a tombstone associated with it, the log would contain (perhaps spread across many segments) O_1, T_1, O_2, and T_2, where T_i represents the tombstone for version i of object O (O_i). In this approach, T_i can be cleaned once O_i is no longer on disk (its segment has been cleaned).

However, if RAMCloud were to only write a tombstone when objects are deleted (not overwritten), cleaning tombstones would be much more complicated. In this case, the log would contain O_1, O_2, and $T_{1,2}$, where $T_{1,2}$ represents a tombstone that deletes both O_1 and O_2 and cannot be cleaned until both versions of O are cleaned from the on-disk log. Unfortunately, figuring out when all previous versions of O have been cleaned is difficult. This would require being able to query all of the previous versions of an object still in the log, which would require maintaining additional metadata, which consumes space and processor cycles, and adds complexity. Because of these problems we decided to adopt the simpler approach in which every obsolete object is given its own corresponding tombstone.

6.1.3 Advantages

The key advantage to tombstones is their simplicity. Reasoning about their correctness is easy, which is an important property for properly tackling the delete consistency problem: due to the one-to-one mapping between object and tombstone, an object is alive in the log if and only if a tombstone for it does not exist.

6.1.4 Disadvantages

One downside to tombstones is that they occupy space in the log (it takes space to free space). For example, in order to delete object O_1, a master must first write a tombstone for it before the object's space can be reclaimed (by either cleaning or compacting its segment). This has secondary effects on the system design. For example, it means that RAMCloud must be careful to reserve space in each master so that objects may later be deleted. After all, if there is no space left in the master, it cannot store tombstones, and therefore cannot remove objects to free up space. Therefore, masters must reject new writes before memory has completely filled, otherwise they risk deadlock.

Tombstones also complicate cleaning and make it more expensive. Although we have shown that it is easy to determine a tombstone's liveness, it is difficult to keep track of the amount of space occupied by dead tombstones in each segment. Maintaining the statistics is difficult because, unlike objects, tombstones are implicitly deleted: they die as a result of cleaning the segments containing the objects they refer to. The key issue is that there is no way to find an object's tombstone to update statistics without maintaining additional metadata (a lookup table from object to tombstone, for example), or violating the immutability of the log to keep track using the space occupied by dead objects and their tombstones. The result is that the cleaner must select segments with imperfect information about their utilisations (some segments may have much lower utilisation than they appear to, but the cleaner does not know which tombstones are dead).

6.1.5 Difficulty of Tracking Tombstone Liveness

At one point we considered an alternative design that would track the tombstones associated with dead objects so that statistics could be updated when the objects are cleaned. For example, RAMCloud could have overwritten deleted objects in the master's memory to point to their corresponding tombstones. This way, the tombstone could be marked as deleted and segment statistics could be updated when the object is cleaned. Moreover, this would only require changing the in-memory log; segment replicas on backup disks would not need to be modified because they are only used during recovery (in which case the volatile liveness statistics would have to be recomputed on new recovery masters anyway). Unfortunately, this has a number of downsides.

A major problem with updating objects in place is that segments would no longer be immutable. This makes concurrent access to segments much more complicated and expensive. For example, when the Infiniband transport layer is used, RAMCloud will not copy objects out of the log to service read requests. Instead, the network interface is instructed to DMA the data directly out of the log in order to reduce latency and conserve memory bandwidth. Making segments immutable ensures that the data does not change out from under the network card. Allowing segments to be modified would require addressing this concurrency problem with mutual exclusion, which would add undesirable latency to both read and delete operations.

Another problem is that tombstones may move around due to cleaning, which makes tracking them more complicated. For example, if the segment a tombstone is in were to be cleaned before the segment containing the object it refers to, then the tombstone would still be alive and the cleaner would relocate it to a new segment. As a result, the reference in the deleted object would need updating to point to the new location of the tombstone. To do this efficiently would require having a direct pointer from the tombstone back to the object, but this would mean increasing the size of the tombstone in order to store the byte offset of the deleted object in addition to its segment identifier.

Finally, segment compaction could actually remove the deleted object from memory before it is cleaned on disk. This would also remove the reference to the tombstone that had been stored in it, so statistics for the tombstone's segment could not be updated when the compacted segment is eventually cleaned on disk. One option is to have the compactor reclaim most of the space from the object and keep just the tombstone reference in the segment, but this has problems of its own. First, it would make compaction less efficient at reclaiming space, especially when objects are small. Second, it would require updating the tombstone's pointer to the deflated "object" (which is now just a reference back to the tombstone), since compaction would have changed its position within the segment and its pointer would need to be updated if the tombstone's segment were cleaned first.

Due to the problems associated with accounting for dead tombstones, I decided to forgo exactly accounting for them in RAMCloud. Section 4.2.2 discusses how RAMCloud's cleaner instead periodically selects segments to scan for dead tombstones.

6.1.6 Potential Optimisation: Disk-only Tombstones

Another alternative design would have been to keep tombstones only on disk, and not in memory. After all, since tombstones are only used during replay of on-disk logs, and considering the complexity and inefficiencies that tombstones add to cleaning, it would be tempting to store them only in backup logs and not in the master's in-memory log.

The current RAMCloud implementation could be adapted to this scheme with relatively few modifications. For example, one straightforward way to avoid storing tombstones in memory would be to allow the memory compactor to remove all tombstones from a segment as soon as the segment is closed. In other words, with the exception of those in the head segment, all tombstones in memory would be considered dead, just like log digests, and could be removed immediately via compaction. This would only require a very simple policy change to the compactor (consider all tombstones to be dead), and it would preserve the same one-to-one mapping between disk and memory segments that the current system has.

There are two problems that would have to addressed before this scheme would work, however. Both currently require that RAMCloud masters keep tombstones in DRAM in order to ensure that the tombstones persist in the replicated log (and therefore keep deleted objects from being resurrected during recovery). The first problem is combined cleaning. Since the cleaner never reads from disk, it must have tombstones in memory to ensure that live ones are rewritten to survivor segments when the combined cleaner runs. If in-memory segments contained no tombstones, then cleaning the on-disk segment would result in losing all of the live tombstones in that segment. The second problem involves re-replicating segments when backups fail and replicas are lost. Currently, RAMCloud masters restore replicas by re-replicating from their DRAM-based segments. Just as in the cleaning case, if these segments did not contain tombstones, then any live tombstones in the lost replica would not be re-replicated.

Fortunately, there is a straightforward solution to this problem: read segments from disk when cleaning and re-replicating, rather than using in-memory copies. This way, any tombstones written to the log could be immediately reclaimed in master memory via segment compaction, yet they would still be perpetuated in the disk-based replicated log because cleaning and re-replication would use the disk copies of segments, not the memory copies lacking tombstones.

Unfortunately, this seemingly simple approach has a number of fundamental trade-offs. First, cleaning would be more I/O intensive, since segments would have to be read from disk first. Technically, only segments that had been compacted in the master (and therefore purged of their tombstones) would have to be read from disk. Uncompacted segments would still have all of their tombstones and the in-memory copy could be used. However, we expect most segments to have been compacted before the combined cleaner gets to them, so disk I/O would usually be required.

The upside is that whereas segments are written to multiple backups for redundancy, only one of those replicas would need to be read for cleaning. This means that the I/O overhead for each new byte stored would increase by at most $\frac{1}{R}$, where R is the number of times each segment is replicated. The reason is that in RAMCloud the cost to write each byte of data in steady state is determined by the cost of freeing that space with the cleaner and replicating data R ways:

$$writeCost = \frac{total\ I/O}{new\ data} = \frac{R \times newBytesWritten + R \times survivorBytesWritten}{newBytesWritten} \tag{6.1}$$

At steady state, when cleaning segments at utilisation u (the fraction of live data in the segment) each segment cleaned will allow $1-u$ more segments' worth of data to be appended to the log. The write cost above in terms of u is then:

$$writeCost = \frac{R(1-u) + Ru}{1-u} = \frac{R}{1-u} \tag{6.2}$$

If instead we were to keep tombstones on disk only and read from a single backup disk when cleaning, then our write cost would be:

$$\frac{total\ I/O}{new\ data} = \frac{R \times newBytesWritten + R \times survivorBytesWritten + diskBytesRead}{newBytesWritten} \tag{6.3}$$

Which in terms of u equals:

$$\frac{R(1-u) + Ru + 1}{1-u} = \frac{R+1}{1-u} \tag{6.4}$$

The ratio of the costs with reading versus without is then:

$$\frac{\frac{R+1}{1-u}}{\frac{R}{1-u}} = \frac{R+1}{R} = 1 + \frac{1}{R} \tag{6.5}$$

Therefore, if $R = 3$ (as is the default in RAMCloud), total I/O overhead will increase by about 33%.

A 33% increase may seem significant, but it can be mitigated in several ways. First, only segments that contain tombstones need to be read from disk. It might be possible for RAMCloud to segregate tombstones and objects into different segments, thereby allowing the combined cleaner to read only on-disk replicas that contain tombstones (the rest could be read from memory). Second, it is important to consider the effect of keeping tombstones in memory on cleaner bandwidth. That is, preserving tombstones in memory increases server memory utilisation, which makes compaction less efficient and therefore increases the frequency and I/O overhead of disk cleaning. This is particularly true when objects are small (100 bytes, for example) and therefore only marginally larger than tombstones. It may be that the increase in memory compaction efficiency gained by removing tombstones from memory would counter-balance the I/O overhead of reading from disk. Unfortunately, I have left determining how these two schemes would compare to future work because there is another trade-off in reading from disk that is more difficult to resolve.

The second trade-off is more subtle: handling backup failures by reading from backup disks could significantly slow down re-replication under certain replication schemes. For example, RAMCloud supports a replication mode based on *copysets* [28], a technique that reduces the probability of data loss after large, coordinated failures such as power loss. The copyset scheme implemented in RAMCloud divides all backup servers into non-overlapping groups ("copysets") containing as many backups as there are replicas for each segment (3, typically). So, for example, a 9-node cluster would have three copysets, and each node would belong to exactly one copyset. When choosing replicas for a segment, masters randomly pick a copyset and place one replica on each node in that copyset.

The problem this scheme introduces is that when a backup crashes, the only other backups containing the same segments are members of its copyset. In other words, with 3-way replication, there will only be two nodes' worth of disks to read the lost segments when re-replicating them, regardless of whether there are 3 or 3,000 nodes in the cluster. This makes recovering from backup failures (restoring adequate redundancy) take much longer, which in turn threatens durability because there is a significantly larger window during which the system has too few replicas of some segments and is therefore more vulnerable to data loss. RAMCloud avoids this problem now because it re-replicates from the in-memory segments. When a backup is lost, many masters lose a small number of segments and independently re-replicate them to new copysets. Unfortunately, this scheme will not work if tombstones are not kept in memory. There are relaxed copyset schemes that help to mitigate this problem, but none are implemented on RAMCloud and they come at the cost of increased data loss probability and implementation complexity.

Disk-only tombstones are an interesting alternative to RAMCloud's current scheme of keeping live tombstones in both DRAM and on backups. It is currently unclear whether the benefits will outweigh the drawbacks, but it would be interesting future work to prototype this scheme and understand its performance implications. If it does significantly improve cleaner performance, it may argue for tackling the re-replication problem and applying this technique in a future version of the system.

6.2 Alternative: Persistent Indexing Structures

In many other storage systems the liveness of data is determined by indexing structures. For example, in typical Unix filesystems the tree formed by the superblock, inodes, directory blocks, indirect blocks, etc. can be walked to discover all live data on disk. Anything not referenced is dead. Although secondary data structures such as bitmaps or trees are often used to efficiently allocate free space, they can always be reconstructed by traversing the tree. The key point is that these systems have no need for tombstones.

This raises a natural question: could RAMCloud have simply persisted its object index – the hash table

– to record which objects are alive, rather than using tombstones or segment masks to record what has been deleted? Perhaps this would be a better approach in terms of performance or implementation complexity.

Unfortunately, this approach would not work well because persisting the hash table to the log would consume too much additional memory. The reason is that the hash table used for lookups needs to be contiguous, so it cannot reside in the log. Instead, a second logical copy would need to be maintained in the log for durability (for example, each time an object is added or deleted and a bucket in the table is modified, that change could be logged). This would mean at least doubling the amount of memory used by the hash table, which would be prohibitively expensive, especially when objects are relatively small. In particular, the hash table space overhead is at least 8 bytes per object, so when storing small objects – 100-bytes, for example – the hash table can easily consume 10% or more of a master's memory. Increasing this to 20% or more is untenable.

More fundamentally, tracking live data is less efficient than tracking dead data (what tombstones do) because we expect the majority of objects in masters to be alive. Moreover, tombstones are only needed for the time interval between when an object is deleted and when it is cleaned. In this case it makes sense to add space overhead for the minority of dead objects, rather than the majority of live objects.

6.3 Tablet Migration

RAMCloud supports moving tablets between servers. This can be used for load balancing, restoring data locality after a master server crash, allowing graceful shutdowns of master servers, and so on. Unfortunately, tablet migration further complicates delete consistency. The problem is that if an object is migrated from one machine M_1 to another machine M_2 and then deleted on M_2, the object may still be present in M_1's log. If the tablet containing the object were later migrated back to M_1, a crash of M_1 could cause the object to be replayed and it could come back to life.

A RAMCloud master cannot solve the problem by simply writing tombstones for each object it migrates away. First of all, this would preclude migrating data back again since an object that was unchanged would now have a tombstone in the log marking it as deleted. Second of all, this would require additional space to free space. We would like migration to be useful for freeing space on servers that are running low on free memory and might not have space to efficiently write out a large number of tombstones without a lot of expensive cleaning.

RAMCloud's current solution to this problem is to mark every tablet in the system with the log position ($\langle$ segment id, byte offset $\rangle$) of the owning master server at the time the tablet was assigned or migrated to it. The coordinator keeps this creation position in its persistent map of tablets and backup servers use it during recovery to filter out objects from previous instances of the tablet. One can think of the log position as a logical timestamp: objects older than the tablet's creation time cannot belong to it.

Unfortunately, cleaning complicates this approach in two ways. First, parallel cleaning may allocate survivor segments with segment identifiers larger than the current head segment. This means that if the log

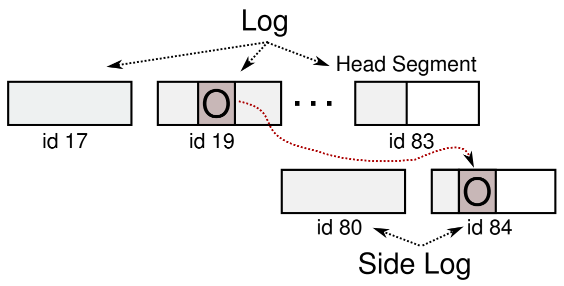


Figure 6.1: Cleaning can relocate live objects to segments newer than the current log head. In the case above, object O is alive at the time of cleaning and is relocated to segment 84, greater than head segment 83. Suppose that the tablet containing O is migrated away to another master, and back again, and that in the meanwhile the log head remains at segment 83 (no, or little new data has been written). If RAMCloud uses the current log position in segment 83 to mark the starting point of the re-migrated tablet, it will erroneously consider object O in segment 84 as being part of that instance of the tablet, rather than the previous instance. The danger is that O may have been deleted on the second master before being migrated back, in which case a failure of the original master could cause it to be resurrected during recovery.

is simply queried for the current log head position to determine a tablet's creation position, the cleaner could have previously moved objects from a prior tablet instance ahead of that position (Figure 6.1). To avoid this problem, when the coordinator queries a master for its current head position while assigning tablets, the current log head is closed and a new one is opened. This guarantees that the log head contains the highest segment identifier, and therefore the position returned is the highest in the log. In addition, since the newly-allocated segment has no object data written to it, the creation position need only take into account the segment identifier, not the byte offset (in other words, creation positions are always $\langle$ segment id, 0 $\rangle$ tuples).

Rolling the log head over also solves another problem introduced by segment compaction. Recall that when segment replicas are lost due to backup server crashes, masters handle the degraded replication by writing new replicas to other backups using in-memory copies of the lost segments (Section 4.1.5). Any segments that had been compacted on the master will result in compacted copies being written to backups. The problem is that compaction may cause valid objects to appear earlier in the log (rather than invalid objects appearing later, as in the combined cleaning problem described in the previous paragraph). This could result in valid objects appearing to exist before a tablet's creation log position for their tablet, which would cause them to be erroneously discarded during crash recovery. However, rolling the head over and using the very beginning of the new segment (offset 0) as a tablet's creation position also avoids this issue, since compaction cannot shift live objects beyond the beginning of the segment.

Rolling over the log head is not completely sufficient, however. The cleaner must also ensure that once the head has been rolled over and the creation position has been queried that it does not relocate any objects from prior tablet instances (otherwise they may appear to be part of the new instance of the tablet). RAMCloud achieves this by ensuring that all objects from a previously-migrated tablet are flushed from the hash table before the tablet is migrated back to the server again [1]. Since the objects are not referenced in the hash table, the cleaner will consider them dead and not relocate them.

Given the intricate and tricky interaction between cleaning and determining object liveness based on log position, we considered alternate solutions. One possibility is to give each instance of a tablet a unique *tablet identifier* that is allocated by and stored in the coordinator; whenever the tablet is migrated, it is considered a new instance and is given a new identifier. In addition, objects would be marked with the tablet identifier instead of the table identifier. This way, if the tablet is migrated away, the new instance will be given a new tablet identifier and any migrated objects will be updated to mark them as part of the newly migrated instance of the tablet. The same occurs if the tablet is migrated back to the original master. Although a master's log may contain objects from a prior instance of the tablet, they will not be replayed during recovery because their tablet identifiers will not match any tablet currently part of the table.

This method avoids the reordering issues introduced by cleaning since the liveness of an object no longer depends on the location of it within the log. The simple equality check when filtering segments during recovery also makes it easier to reason about correctness (the object is replayed if and only if the tablet identifier matches a tablet currently part of the table). Moreover, each tablet identifier would be inherently master-specific, meaning that a cluster-unique value would not be necessary and less space would be needed to store the tablet identifier in each object and tombstone. However, this approach also has a few downsides. First, it makes debugging more difficult because it is no longer clear when looking at an object which table it actually belongs to (the table must be looked up based on the tablet). We could store both a table and tablet identifier in the object, but that would consume additional space. Second, objects must be modified during migration and crash recovery to include the new tablet identifier, otherwise they will not be replayed after the next crash. This would introduce extra overhead in the tightly-optimised recovery system, since objects would need to be verified, updated, and checksummed again.

Another alternative is to not permit tablets to be migrated back to a machine on which they used to live until that server has cleaned every segment in its log (or every segment created after the tablet was assigned to it). This would guarantee that no objects from the old instance of the tablet still remain, and therefore none could be erroneously replayed during recovery. If migrations are relatively rare, and we expect that they will be, then migrations back to the original server again are likely to be even more infrequent. This should provide plenty of time for masters to first purge old instances of a tablet's data before accepting it back again. The downside to this approach is that it introduces an artificial limitation: even if a server has plenty of space, it may not be able to accept a tablet.

As we have seen, making tablet migration work safely in the face of failures is surprisingly delicate. It is unfortunate that most of this subtlety stems from an unlikely scenario: a tablet being migrated back to the master it used to be on, before the server could clean the old tablet instance's objects from its log. RAMCloud uses the log position approach outlined above to make this safe, but it comes at the cost of being

[1] If you are wondering why references may still exist in the hash table after migrating the data away, the answer is that RAMCloud flushes the hash table lazily. Masters avoid using stale objects when processing client requests by filtering out any objects they find in the hash table that correspond to tablets for which the master does not have ownership.

difficult to reason about. Time will tell if this complexity is manageable and has been implemented correctly. Fortunately, several alternatives can be adopted if this approach must be abandoned.

6.4 Summary

RAMCloud uses tombstones to ensure that deleted or overwritten objects are not replayed during crash recovery. Writing a tombstone to the log for every obsolete object is a simple way to avoid restoring these objects after a crash, but the approach has a number of downsides. For example, tombstones require free space to be available in order to delete data. Furthermore, it is difficult to accurately track which tombstones are dead, and as they accumulate memory compaction becomes less efficient.

Tablet migration adds additional complexity because writing a tombstone for every object being migrated would require too much time and space. This is normally not a problem (if a master no longer owns a tablet, objects still in its log will not be replayed during recovery). However, it does require that additional precautions be taken if tablets are migrated away from a master, and then back again (because the log may contain data from different instances of the tablet).

Tombstones leave something to be desired, but we have learned to live with them in RAMCloud. There are some interesting avenues for further research and optimisation, including storing tombstones only on disk (where they are needed), and not in memory. This has some important ramifications on backup server recovery, and may or may not be a net efficiency win, but the benefits and trade-offs should be explored more fully.

Chapter 7

Cost-Benefit Segment Selection Revisited

Log-structured filesystems [69] typically use a garbage collection technique called *cleaning* to incrementally reclaim free space in the log so that new data may be appended. Rather than reclaiming space from the entire log at once, cleaning works by selecting smaller portions of the log, called *segments*, and reclaims their free space individually.

How much work the system must do when reclaiming space is directly related to which segments are chosen, so cleaners rely on policies that attempt to minimise the overhead of reclamation by deciding which segments are best to clean. Compared to a simple greedy policy that chooses segments with the most free space, the LFS [69] work showed that the overhead of reclaiming free space can be significantly reduced – up to about 50% – if a more clever policy is used to choose segments to clean. This *cost-benefit* policy uses a heuristic that chooses segments based on the both amount of free space they have and the stability of their data.

However, I found that the LFS formula as it was originally described has some shortcomings. First, it does not appear to balance the free space and stability components properly: often times stability far outweighs free space and leads to poor segment selection. Second, stability is inferred using the ages of data in each segment, which can sometimes be a poor approximation, especially under uniform random access patterns. These problems can cause cleaning to be half as efficient as compared to better policies.

This chapter introduces new formulas that perform better than the cost-benefit policy described in [68] by addressing these shortcomings. By striking a better balance between free space and stability, these alternatives are up to twice as efficient when utilisation is high (most of the disk or memory is full of live data). Moreover, by directly computing a segment's stability rather than approximating it by the ages of data contained within, they avoid pathologies in the original cost-benefit formula that could otherwise double cleaning overheads when access patterns are uniform random. I will show that a straightforward and effective alternative is to simply use the age of the segment, rather than the age of its data when computing its cost-benefit score.

Finally, I briefly show that even the improved cost-benefit heuristics are not optimal. I augmented a cleaning simulator with an oracle that provides the cleaner with perfect knowledge of future accesses. The result is that the oracular cleaner can be 2-3x more efficient in some workloads, indicating that there may still be room for significant improvement to the cost-benefit policy.

7.1 Introduction to Cleaning in Log-structured Filesystems

The log-structured filesystem (LFS [69]) was introduced 20 years ago as a significant departure from the update-in-place designs of previous filesystems. LFS replaced complex block allocation strategies with a simple "bump-a-pointer" scheme in which all writes were made to the end of a contiguous log. Even modifications to existing blocks were done in a copy-on-write manner: rather than overwriting the old block, the new contents were also appended to the end of the log. This allowed the filesystem to issue large sequential I/Os to underlying disks, amortising much of their seek and rotational latencies. The result was that LFS could attain upwards of 70% of the devices' bandwidth, rather than the 5-10% of other filesystems at the time.

In order for this simple allocation policy to work, however, LFS needed to ensure a steady supply of contiguous free space to append to. This meant that eventually the log would need to be defragmented (or "cleaned") to reclaim any free space that had accumulated throughout the log (due to blocks either having been deleted or superseded by newer versions later in the log). Any extra disk bandwidth spent cleaning was pure overhead – bandwidth that could have otherwise been used to write new data to the log – so it was important to reduce the cost of cleaning to mitigate its effect on filesystem performance. A simple solution, for example, would have been to sequentially scan the log from beginning to end and write out all of the live data in one contiguous chunk. Unfortunately, the cost of this approach would be prohibitively high: if the disk were run at 90% utilisation (90% of blocks contain live data), LFS would have to read and write nearly the entire disk to reclaim the 10% of disk space that was free.

Instead of cleaning the entire disk at once, LFS allowed finer-grained cleaning by dividing the disk into fixed-sized *segments*, each of which could be cleaned independently. This scheme allowed the filesystem to clean more efficiently since it could choose to clean the segments that would return the most space for the least cost (it could also ignore segments containing little or no free space at all and avoid the high price of reading and rewriting their data). At a high level, the cleaning procedure was very simple: LFS chose a set of segments to clean, wrote their live data out contiguously to new segments, and then reused the cleaned segments for new writes. These individual segments, though potentially discontiguous on disk, were threaded together to form a logically contiguous log. The goal of high write bandwidth was maintained by making the segments large enough to still effectively amortise seek and rotational latencies (512 KB in Sprite LFS).

In steady state, writing new data in LFS meant writing to space that was previously reclaimed through cleaning. Therefore the true cost of writing data to disk needed to include not only the size of the new data, but also the bandwidth used by the cleaner to reclaim the space being written into. The more bandwidth the cleaner used, the less would be left over for writing new data.

To reason about the total bandwidth needed for writing new data, LFS introduced the notion of *write* *cost*. Write cost is simply a multiplier that reflects the actual bandwidth required to store each byte on disk, including cleaning. For example, a write cost of 1 is optimal and means that the cleaner never read from or wrote to disk. A write cost of 2 means that the cleaner doubled the amount of traffic to disk, leaving less bandwidth for new writes (half of the disk bandwidth is used to write new data, the other half is used by the cleaner). Write cost is equivalent to the more contemporary term *write amplification*, which is used in the context of solid-state flash disks ("SSDs").

Unless cleaning were performed on a completely empty segment, the entire segment would have to be read in before the live data could be written out. In these cases the write cost would be > 2: to write a segment of new data, the cleaner would have needed to first read in at least one segment to clean. This alone yields a write cost of at least 2, and the live data written out during cleaning correspondingly increases the write cost. The result is that write costs in LFS were typically greater than 2.

7.2 The Cost-Benefit Cleaning Policy

LFS showed that the most important policy decision when cleaning was deciding which segments to select. The cost of cleaning depends on the amount of live data in the segments chosen for cleaning (more live data means that not only is more work needed to relocate it, but also that less free space is reclaimed). More interestingly, however, LFS showed that an obvious policy – to greedily choose segments with the lowest utilisation – would not necessarily yield the lowest cleaning cost in the long run. It turns out that sometimes it is worth cleaning segments at higher utilisation if they are slowly changing (that is, the amount of live data in them is decreasing slowly). In fact, LFS simulations showed that the total amount of disk bandwidth used by the filesystem could sometimes be halved just by more judiciously selecting which segments to clean, rather than choosing greedily [68].

To understand the intuition behind this, consider two segments A and B, which have an equal amount of free space. Based on utilisation alone, the cost of cleaning them is the same (the same amount of work – copying live data – will reclaim the same amount of free space). However, if A is changing more slowly than B, then A is a better choice in the long run. The reason is that the segments that A's live data will be moved to during cleaning will be more stable than they would have been had B been chosen instead. This means that those segments will accumulate fragmented free space more slowly than had they contained B's data. Consequently, if fewer files are deleted in these new segments, then more files must be deleted in other segments. As a result, the amount of live data in those other segments will drop more quickly because deletions are focused on a smaller set of segments. This makes them cheaper to clean in the future.

It follows from this reasoning that if cleaning the more stable segment is preferable when utilisations are equal, then it will sometimes also be better to prefer a segment with higher utilisation if it has higher stability as well. This intuition formed the basis of LFS' *cost-benefit* selection policy, which heuristically selects segments based on both the amount of free space they have and how stable their remaining data is expected to be. At cleaning time, segments were chosen with the highest benefit-to-cost ratio, as determined by the following formula:

$$\frac{benefit}{cost} = \frac{bytesFree \times age}{bytesRead + bytesWritten} = \frac{(1-u) \times age}{1+u} \tag{7.1}$$

That is, if a full segment's size is normalised to 1 and if $u \in [0, 1]$ is the fraction of data still alive (the utilisation), then cleaning the segment will reclaim $1 - u$ free space. In the denominator, the cost of cleaning includes reading the entire segment from disk (1) and writing out the live data (u).

The age factor is simply a heuristic to approximate how much the segment is changing. The larger the value, the more stable the segment is. Sprite LFS computed a segment's age as the youngest file (or portion of a file) contained in the segment. It did so by keeping track of each segment's most recently modified file in an in-memory data structure (the "segment table"). When files were written to a segment either during cleaning or normal operation, this modification time would be increased if any of the files' modification times were greater. The cleaner would then use the current system time and these timestamps to compute age for each segment on disk. The older the files in a segment were, the longer they could be expected to stay alive, and therefore the more preferable cleaning the segment would be.

7.3 Problems with LFS Cost-Benefit

I first encountered problems with the LFS cost-benefit formula while sanity-checking the cleaner I was developing for RAMCloud. In order to better reason about RAMCloud's behaviour I reimplemented the LFS simulator described in [68] to have a comparison point. Unfortunately, the original LFS simulator source code is no longer available, so I attempted to reproduce its behaviour as faithfully as its description allowed. I was not able to reproduce the results from [68], however, which led to discovering problems in LFS's cost-benefit approach.

This section reviews the structure of the simulator, introduces the surprisingly poor simulation results it initially produced, and then explores problems with the original cost-benefit formula as well as some candidate solutions.

7.3.1 LFS Simulator

The simulator models a simple version of LFS with 2 MB segments (4x larger than the segments in the actual Sprite LFS implementation), fixed 4 KB files and no metadata. In other words, there are exactly 512 files per segment. It takes as input the percentage of live data in the simulated disk (u – the utilisation) and an access distribution. The amount of live data is fixed at 100 segments' worth of files (51200) and the total number of segments is scaled according to u. For example, if u is 50%, then there will be 200 total segments on the simulated disk. When the simulator finishes running it emits the average overall write cost.

When the simulation starts, segments are filled one-by-one with fixed-sized files until the percentage of the disk occupied by the files reaches u. At this point, about u% of the segments are filled, and $100 - u$% are empty. Files are then overwritten according to the chosen access distribution. The act of overwriting causes the old version of the file to be deleted, and a new version to be written to the segment at the end of the log.

Which files are overwritten is determined by the specified access distribution. I implemented the same three distributions that the original LFS simulator used:

1. **Uniform Random** Each file has an equal chance of being overwritten. In other words, if there are N live files on disk then each file has a $\frac{1}{N}$ chance of being overwritten by each write.

2. **Hot-and-Cold** Files are statically divided into two groups: a *hot* group and a *cold* group. The hot group contains 10% of the files, but receives 90% of the writes. The cold group contains 90% of the files, but receives 10% of the writes. Before each write a group is randomly chosen with 10% and 90% probability, and then a file is uniform randomly selected within the group (that is, the files within each group have an equal probability of being overwritten). This scheme models a simple form of locality.

3. **Exponential** The probabilities for selecting each file to overwrite follow an exponential distribution with $\mu = 0.02172$ ($\lambda = \frac{1}{\mu} = 46.04052$). That is, probabilities are assigned to each file so that when randomly selected their frequencies form this distribution. This models much higher locality: approximately 90% of writes are made to 2% of the files, and 99.9% of writes are to about 6.5%.

The simulator cleans when it runs out of free space (there are no more empty segments to append files to). Each time the cleaner runs it ranks segments by their cost-benefit score and chooses the highest ranking ones for cleaning. The number of segments cleaned is limited by their total number of live files: the cleaner will write out up to ten full segments' worth of live files each time it runs. For example, if it cleans segments that contain 25% live data on average, it will clean about 40 segments.

Like the original LFS simulator, during cleaning my simulator also takes the live files in selected segments and sorts them by age before writing them back out. File ages are determined by simulated time: a logical clock is incremented for each write and files are stamped with this value when overwritten. Sorting improves cleaning efficiency by segregating old and new data into different segments. This helps to focus the accumulation of free space into fewer segments since old data are likely to persist longer (assuming some degree of write locality) and new data are more likely to be overwritten soon. This way, those new data segments will have more free space when cleaned and will therefore be cheaper to process.

The simulation ends when cleaning reaches steady state. Steady state is determined when the write cost converges to a stable value (it has not changed up to four decimal places in the last N writes, where N is equal to twice the number of files that could fit on the simulated disk). The original simulator's description did not detail exactly how convergence was determined, but did provide a range of total writes needed for the simulations to complete. This heuristic is conservative: my simulations appear to require about 1-3 times as many writes to complete as the original simulator.

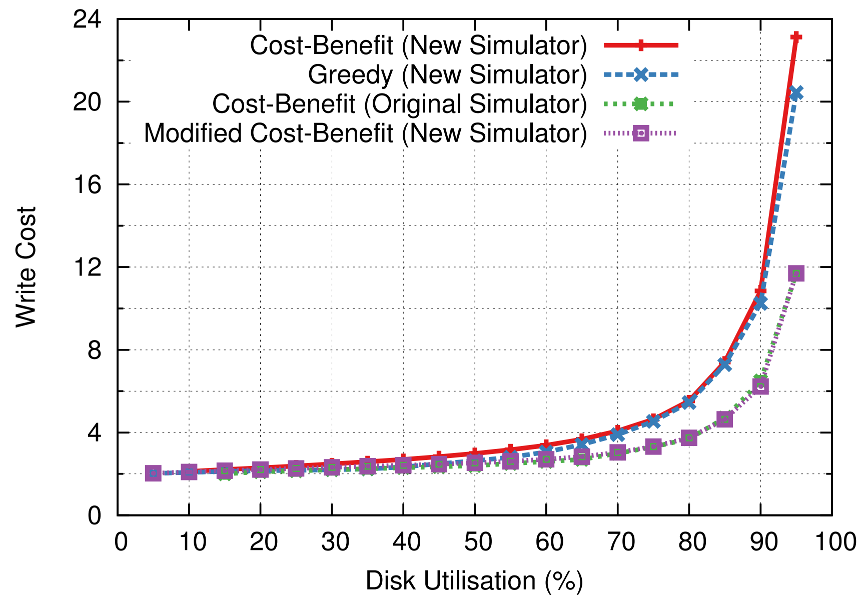


Figure 7.1: Comparison of cleaning performance between the original LFS simulator (as reported in [68]) and my re-implementation under the hot-and-cold access pattern. Performance is characterised by the overall write cost during the simulation when the simulated disk was run at various utilisations from 5% to 95%. As the percentage of live data on the disk increases, so does the cleaning overhead. Surprisingly, not only is performance worse than the original simulator when using the cost-benefit policy, but my new simulator is actually more efficient when using the greedy policy than cost-benefit. In comparison, a very minor modification to the formula (”Modified Cost-Benefit”) almost exactly reproduces the original simulator’s results. This revised policy is described in Section 7.3.5 (Equation 7.3). It differs from the LFS cost-benefit formula (Equation 7.1) only in that it uses the age of the segment itself, rather than the age of the youngest file within the segment to estimate stability.

7.3.2 Initial Simulation Results

The first simulation results were surprising: they showed that my reimplemented simulator would clean much less efficiently – with up to twice the write cost of the original LFS simulator – under the hot-and-cold access pattern (Figure 7.1). In fact, the new simulator's cost-benefit algorithm performed slightly worse than the greedy approach. Something was clearly wrong. According to the LFS literature, cost-benefit should have been significantly better than the greedy policy given the presence of locality.

While I could not look for bugs in or other differences in the original simulator source code that might explain the discrepancy, I was able to replicate the experiments with the RAMCloud system under the same parameters (disk cleaner only/no compaction, with the same cost-benefit formula, access pattern, utilisation, and number of objects per segment, and with no tombstones). Despite the completely different code bases, write costs in RAMCloud differed from my simulations by at most 5%. This suggested that I was looking at legitimate behaviour, rather than the result of a bug in my simulator.

I then tried to reproduce the original simulator's results by making minor modifications to the cost-benefit policy in my simulator – small changes that could have been simple bugs in the original implementation. Eventually, I succeeded: Figure 7.1 also depicts the performance of a slight modification to the cost-benefit formula that performs almost identically to the original simulator. The details of this formula are presented in Section 7.3.5. For now I will use it as a comparison point to show how the original LFS cost-benefit formula – as it was described – behaved differently than expected and as a result did not produce the expected write costs. Although there is no way to be sure, one reasonable explanation for the discrepancy is that the original simulator inadvertently implemented a formula (like my modified one, perhaps) that was subtly different than what the text described.

The rest of this section describes the problems with the original LFS cost-benefit formula that I discovered while trying to understand the performance discrepancy, as well as possible solutions. The following section then discusses the modified formula used in Figure 7.1 as an alternate solution.

7.3.3 Balancing Age and Utilisation

While comparing simulations to RAMCloud to sanity check the unexpected hot-and-cold results, I also noticed that cleaning was about 10-20% more efficient when using the original RAMCloud cost-benefit formula instead of the LFS formula. RAMCloud's formula differed slightly because it is not necessary to read segments from disk when cleaning them, so the $cost$ portion in the denominator was u, rather than $1 + u$ (the 1 accounts for the cleaner reading in the segment from disk and u for writing out the live data).

This subtle difference had an unexpected consequence. In LFS, the utilisation portion of the formula is strictly bounded and has a small magnitude ($0 \leq \frac{(1-u)}{(1+u)} \leq 1$)[1]. This portion of the RAMCloud cost-benefit formula, however, may grow much larger: as u approaches 0, $\frac{(1-u)}{u}$ approaches ∞. This difference led me

[1]The special case of $u = 0$ can be ignored here because it rarely occurs at interesting disk utilisations (a completely empty segment requires no disk I/O to clean and can be considered to have a cost-benefit value of ∞).

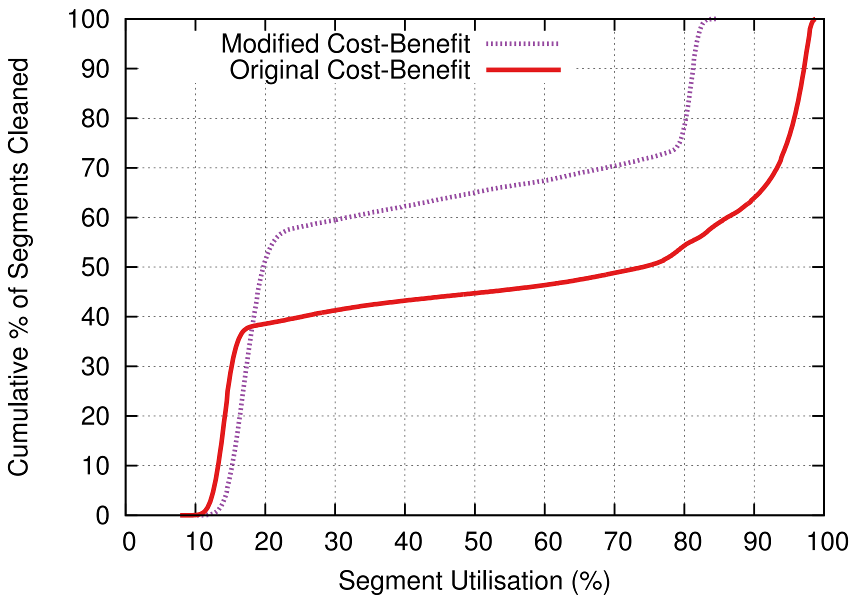


Figure 7.2: Cumulative distribution of segments cleaned by my simulator (hot-and-cold access pattern, 75% disk utilisation). The original cost-benefit policy cleans 40% of segments with under 25% utilisation. Surprisingly, 35% of segments are cleaned at 90% or higher utilisation, which is almost an order of magnitude more expensive than cleaning at $u = 0.25$. In comparison, the modified policy that reproduces the publicised LFS write costs cleans no segments above about 85% utilisation, and nearly 60% of all segments cleaned are under 25% utilisation. Figure 7.3 shows how this contributes to much lower cleaning overheads.

to question the proper balance between utilisation and age, since the RAMCloud formula was performing slightly better with a different weighting of these two factors.

One immediate concern with using file age in the cost-benefit formula is that ages are unbounded (they will increase so long as files remain alive). Yet while the age component of the cost-benefit formula can grow arbitrarily large, the $\frac{(1-u)}{(1+u)}$ portion is always ≤ 1. At least in principle, the age factor in very stable segments could easily overwhelm the utilisation factor.

Figure 7.2 shows that this appears to be the case. Surprisingly, when the disk is run at 75% utilisation, the LFS formula cleans nearly 40% of all segments when their utilisations are above 90%. Furthermore, nearly 25% of all cleaned segments had utilisations above 95%, which are twice as expensive to clean as 90% segments. Figure 7.3 indicates why this is problematic: too much work is expended cleaning these very highly utilised segments. What was happening was that the ages of the stable segments were out-weighing their high utilisation and causing them to be cleaned too frequently (very high utilisation segments were often given the highest cost-benefit scores).

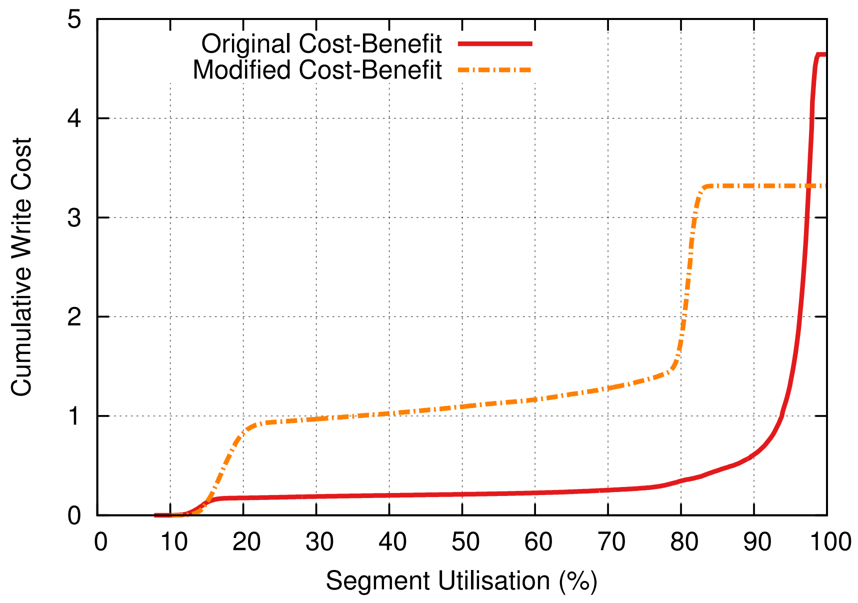


Figure 7.3: Cumulative distribution of work done during cleaning in the same experiment as in Figure 7.2. The maximum height of each curve represents the overall write cost for that experiment. The curve itself shows how much of that cost is attributed to segments cleaned at various utilisations (the more live data a segment has, the more expensive it is to clean). Even though Figure 7.2 showed that only about 25% of segments were cleaned at 95% or higher utilisation with the original policy, nearly 75% of the cleaner's work was spent processing these segments. This highlights the imbalance produced by large age values: almost 95% of work is spent cleaning segments greater than the disk utilisation (75%). In comparison, the modified policy better balances work between segments slightly higher than the disk utilisation, and ones substantially lower. This is the expected cost-benefit behaviour.

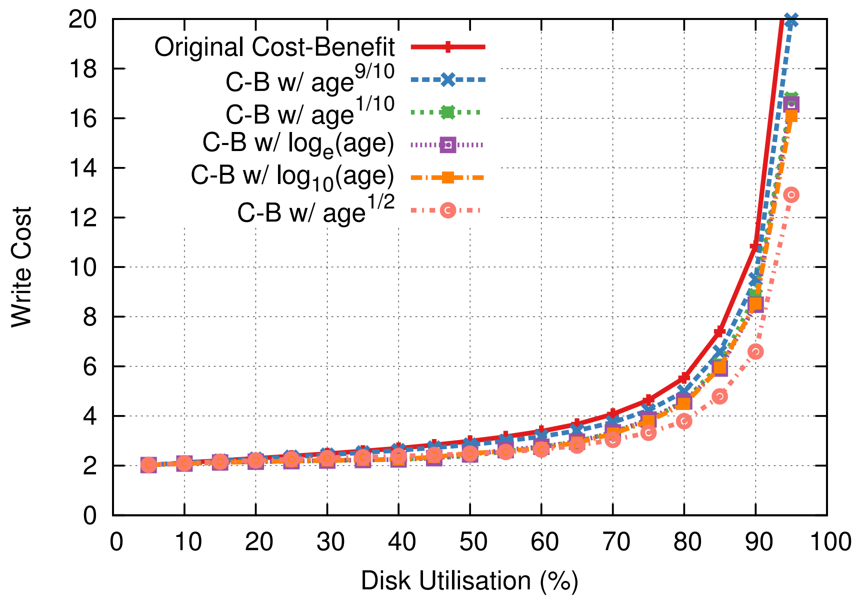


Figure 7.4: Effect on cleaning costs by reducing the weight of age in the cost-benefit formula. All of the moderate roots and logarithms improve cleaning efficiency. In particular, taking the square root of age offers nearly a 2x improvement at high memory utilisations. The only data point not depicted is the "Original Cost-Benefit" value at 95% utilisation (its y-value is 23.1). All curves were generated using the hot-and-cold distribution.

Moderating Age

Unfortunately, it is not obvious how the space ($\frac{(1-u)}{(1+u)}$) and time (age) components of the cost-benefit formula should be balanced. If age is given too much precedence, then it can cause very stable segments of very high utilisation to be cleaned more often than they should. On the other hand, if utilisation is given too much precedence, then the policy will behave more greedily and could miss the opportunity to save work in the long run by expending more effort now. Either case results in suboptimal performance: the cleaner uses more I/O resources than it should because it cleans at a higher segment utilisation in the long run.

Since age dominated and adversely impacted the hot-and-cold case, I initially experimented with different ways of curbing age's magnitude. Simply dividing age by a constant would have no effect, since that would only apply a multiplicative constant to all of the segments' cost-benefit scores and would not change their relative ranking. Instead, I explored taking various roots and logarithms of age (for example, $\frac{benefit}{cost} = \frac{(1-u)\times\sqrt{age}}{1+u}$). Figure 7.4 compares simulations of the original cost-benefit formula to several variants in which age is reduced in weight. In particular, taking the square root of age reduces cleaning costs by at least 40% at utilisations above 90%.

Unfortunately, even if we could find the best balance between utilisation and age, we would still be left

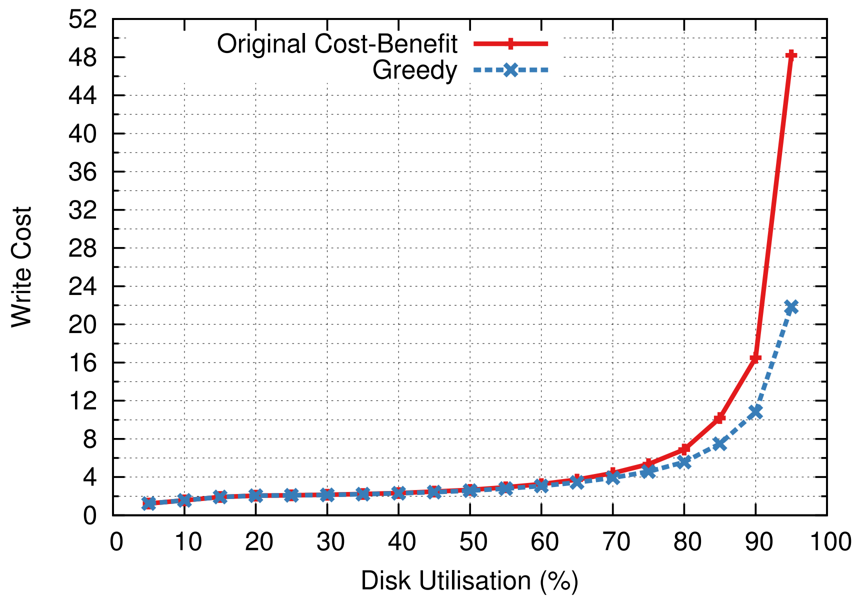


Figure 7.5: Write cost of the cost-benefit and greedy selection policies as a function of disk utilisation under the uniform random access pattern. File ages are not valid predictors of segment stability, so the cost-benefit policy performs up to twice as much work while cleaning. The problem is that it erroneously assumes that colder-appearing segments will contain longer-lived data.

with another significant problem to solve. The next section describes how the original cost-benefit formula behaves poorly under uniform access patterns and proposes a simple solution.

7.3.4 File Age is a Weak Approximation of Stability

Another problem with using file age is that it leads to selecting the wrong segments to clean when there is no access locality. For example, suppose that files are overwritten uniform randomly (that is, the probability of each file being overwritten is always $\frac{1}{N}$, where N is the total number of live files in the system). In this case past behaviour is not a good predictor of future behaviour because how long a file has managed to survive is no indication of how much longer it is likely to persist.

Unfortunately, in this scenario the cost-benefit algorithm will give preference to segments that appear to be colder, even though they are not (in fact, all segments are decaying at more or less the same rate, regardless of how long their data has been around). As Figure 7.5 shows, the result is that cost-benefit selection can be less than half as efficient as the simple greedy policy as disk utilisation increases.

Figure 7.6 illustrates why this is happening: the cost-benefit policy tries to be clever and selects some segments of higher- than-average utilisation because they appear to be colder (but in fact are not). Even worse, the cleaner exacerbates the problem when it sorts live files by age and segregates them into different

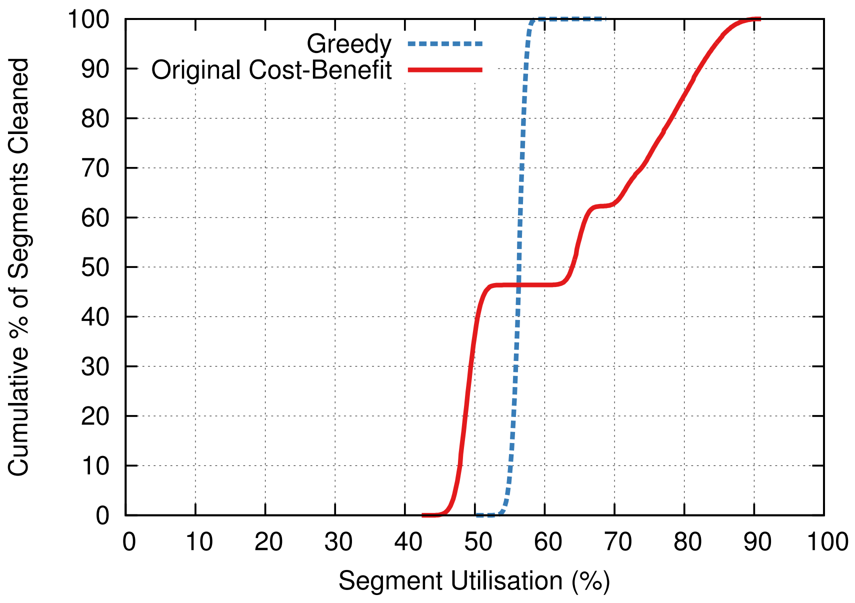


Figure 7.6: Cumulative distribution of cleaned segments' utilisations under the uniform random access pattern, comparing the LFS cost-benefit and greedy selection policies at 75% disk utilisation. The greedy policy cleans the overwhelming majority of segments at around 64% utilisation. Cost-benefit cleans about 40% of segments at up to about 10% lower utilisation than greedy, but the rest of the segments are cleaned at significantly higher utilisation – up to 35% higher. The uniform pattern fools the cost-benefit policy into preferring highly-utilised segments, even though their data are no colder than any other segments'.

segments. This ensures that there are segments that appear distinctly hotter or colder based on file age, but in fact are no different. The cost-benefit policy will favour the seemingly colder segments, even though they can be expected to decay at approximately the same rate as any other segment.

More Accurately Predicting Segment Stability

The idea behind using file age in the cost-benefit calculation is to predict the future stability of a segment, but this is a rather indirect way of doing so: file age estimates prior stability, which estimates future stability. A more direct alternative is to simply measure the actual stability the segment over time, rather than attempting to infer what it will be based on its contents.

The simplest way of measuring stability is to compute the decay rate of a segment's data over its lifetime ($\frac{1-u}{segmentAge}$). Cost-benefit already requires the system to keep track of each segment's utilisation (u), so the only additional information needed to track stability is the segment's age. Simply recording the creation time of each segment suffices. More elaborate schemes are also possible, such as computing decay as an exponentially weighted moving average, which may allow the system to better adapt to changes in decay rates as access patterns change. I have left these extensions for future work.

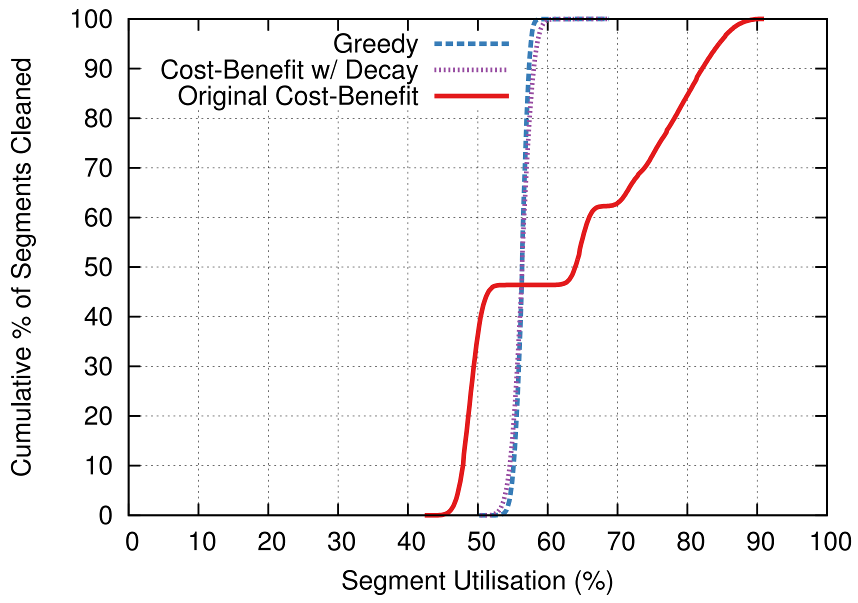


Figure 7.7: Cumulative percentage of segments cleaned at various utilisations under the uniform random access pattern at 75% disk utilisation. The greedy and decay-based cost-benefit policies perform almost identically. We can expect random variations to cause a small number of segments to have significantly higher decay and lower decay than average, which explains why the decay-based policy cleans some segments at both higher and lower utilisation than the greedy policy. Overall, however, the two policies behave nearly the same.

To test using decay in cost-benefit selection, I modified the simulator to take into account each segment's decay rate, rather than file age, as follows:

$$\frac{benefit}{cost} = \frac{1-u}{(1+u) \times \sqrt{decay}} = \frac{1-u}{(1+u) \times \sqrt{\frac{1-u}{segmentAge}}} \tag{7.2}$$

The square root of segment decay was taken for two reasons. First, it avoids cancelling the $1 - u$ term in the numerator (the $benefit$ portion of the formula). That would have caused all segments to have the same $benefit$ value of 1, effectively taking only cost into account and devolving into a greedy algorithm that selects on utilisation alone. Second, it moderates the weight of decay in the same heuristic way that worked well with age (Section 7.3.3).

One immediate benefit of directly measuring segment stability is that it solves the uniform access pattern problem: when file lifetimes are uniformly random, most segments will have similar stability, so cost-benefit selection will depend primarily upon utilisation when choosing segments. Figure 7.7 shows that this is largely the case. As a result, the decay-based selection strategy naturally devolves into the more appropriate greedy policy when there is no locality to make use of.

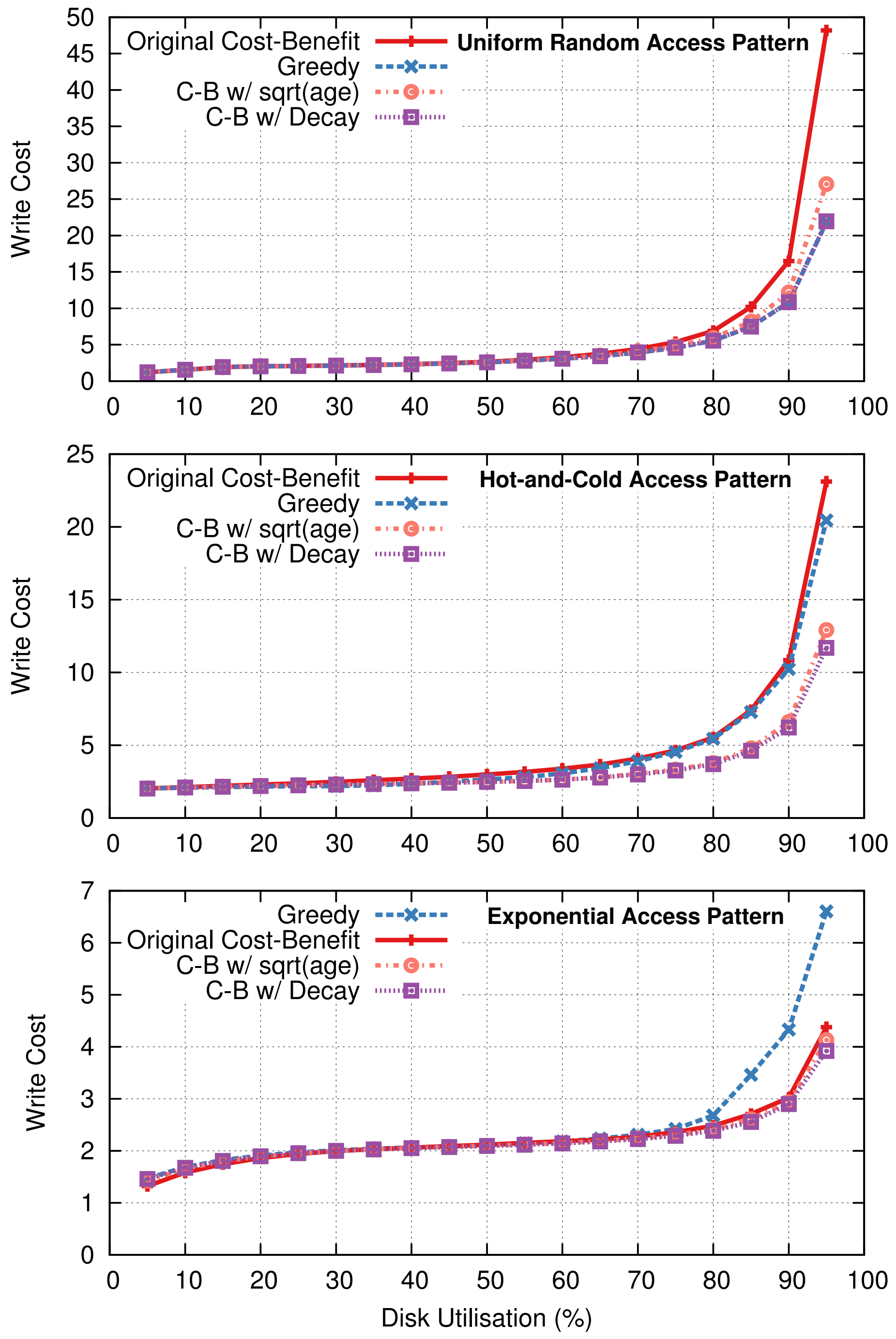


Figure 7.8: Cleaning costs with the original LFS cost-benefit formula and the two alternatives described in Sections 7.3.3 and 7.3.4. Both alternative formulas nearly halve the write cost at high disk utilisation under the hot-and-cold and uniform access patterns. In the uniform random case, using measured segment decay rather than file age provides an additional 20% improvement. The exponential access pattern is the same one used in [68] (approximately 90% of writes are made to 2% of the files). The original cost-benefit formula does considerably better in this case because many of the stable segments stay at 100% utilisation for a long time and are rarely cleaned.

Figure 7.8 compares the efficiency of the original LFS cost-benefit formula with this decay-based formula across all three access distributions. The results are excellent: at very high utilisations, using decay can reduce the cost of cleaning by nearly 50% and consistently out-performs the previous best formula based on file age (the square-root-moderated formula from Section 7.3.3). Furthermore, with the uniform random distribution it behaves almost identically to the greedy policy. This is exactly what we want when there is no locality to exploit.

7.3.5 Replacing Minimum File Age with Segment Age

Rather than taking the square root of $decay$ in Equation 7.2, I also experimented with squaring the numerator. Although somewhat arbitrary, the hope here was that squaring the $1-u$ term would also help moderate the impact of decay on the formula by increasing the contribution of utilisation. Moreover, it would also have a nice aesthetic result since a $1-u$ term would cancel with the $1-u$ term in the denominator and yield a simple and familiar equation:

$$\frac{benefit}{cost} = \frac{(1-u)^2}{(1+u)\times decay} = \frac{(1-u)^2}{(1+u)\times\frac{1-u}{segmentAge}} = \frac{(1-u)\times segmentAge}{1+u} \tag{7.3}$$

The result is nearly equivalent to the original cost-benefit formula, except that instead of using the minimum file age in a segment, it uses the age of the segment itself. The simplicity of this equation belies the fact that the $1+u$ term is essentially being implicitly squared while taking decay into account.

Figure 7.9 shows that this policy performs very under well under the hot-and-cold access pattern – nearly 30% better than the original cost-benefit formula at 75% disk utilisation. The figure compares the amount of work performed by the cleaner with the three other alternative cost-benefit policies introduced in this chapter and shows that they all perform similarly. It also illustrates that giving greater weight to utilisation improves overall cleaning performance by avoiding segments of very high utilisation (see Figure 7.10 for the cumulative distribution of segments cleaned at different utilisations). For example, it makes no sense for the original cost-benefit and RAMCloud formulas to expend 2x or more effort to clean at $\geq$ 90% instead of 80% utilisation when that investment only reduces the u of the lowly-utilised cleaned segments from about 25% to 15% (only a 1.13x difference in write cost).

Equation 7.3 also performs almost identically to the decay-based policy (Equation 7.2) under the other access patterns. In particular, for the same set of experiments as depicted in Figure 7.8 (uniform, hot-and-cold, and exponential access patterns across different disk utilisations) write costs seen when using segment age were always within 5% of the write costs obtained with the decay formula. Figures 7.11 and 7.12 show the work and segment utilisation distributions under the uniform random access pattern.

The simplicity of Equation 7.3 and its consistently good performance makes it an excellent replacement for the original cost-benefit formula. For these reasons I based the segment selection policy used by RAMCloud is based on this formula.

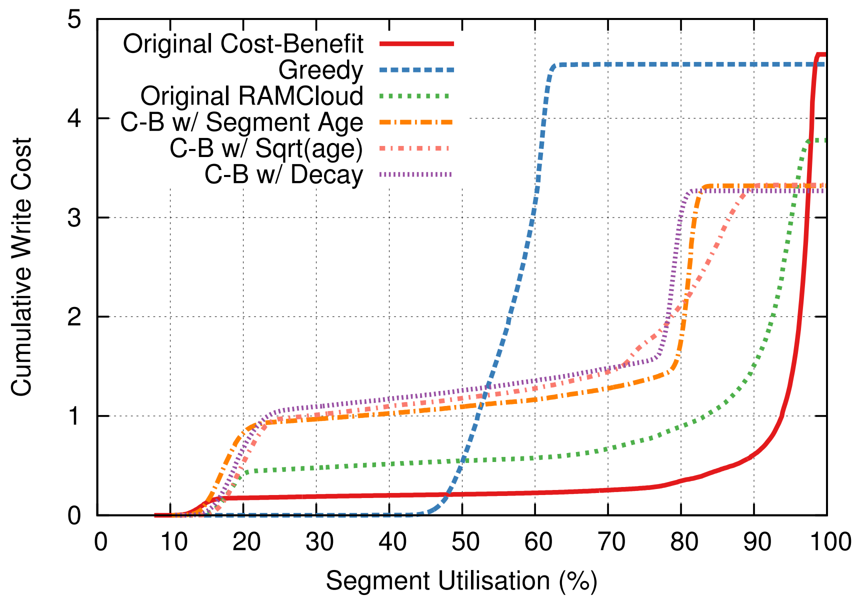


Figure 7.9: Cumulative distribution of work done during cleaning under the hot-and-cold access pattern with various cost-benefit formulas at 75% disk utilisation. The “RAMCloud” and original cost-benefit formulas strike a poor balance between cleaning segments at low and high utilisation: they clean far too many segments at the high end and derive little additional benefit at the low end. The bottom three curves in the legend (”C-B w/ ...”) correspond to alternative policies that increase the weight of utilisation when choosing segments. This causes them to clean much more efficiently overall. Although the minimum segment utilisation at which they clean is slightly higher than the original cost-benefit formula, the difference in write cost is negligible below a u of 25%. This small amount of extra overhead at the low end avoids having to clean very expensive segments at the high end.

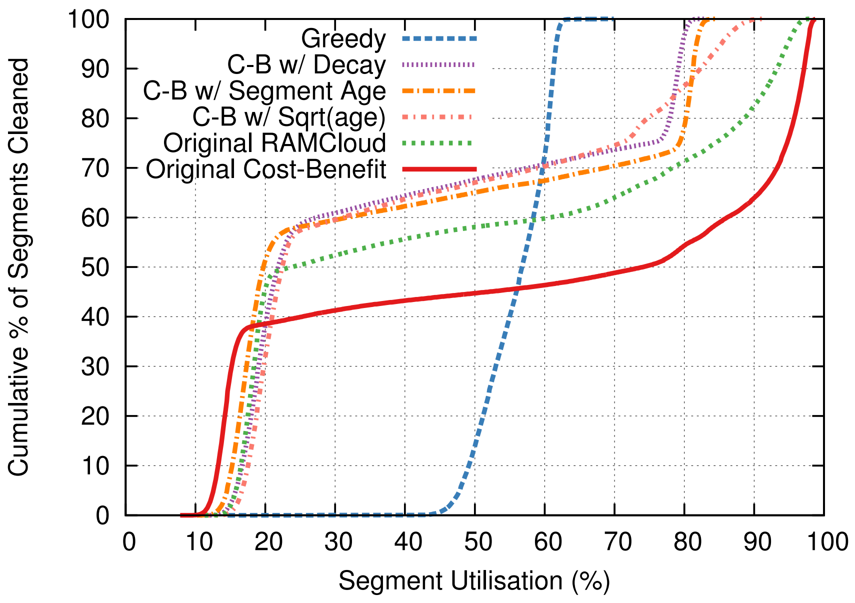


Figure 7.10: Cumulative percentage of segments cleaned in the hot-and-cold experiments depicted in Figure 7.9. The segment-age-based policy avoids cleaning the extremely expensive segments at 85% and higher utilisation that contribute to nearly all of the overhead of the original cost-benefit formula (Figure 7.9).

7.4 Is Cost-Benefit Selection Optimal?

The Sprite LFS work showed that cost-benefit selection often out-performs a simple greedy approach, but provided no indication of how close it is to optimal. Given how expensive cleaning becomes as utilisation increases, even a modest improvement in the average utilisation of segments cleaned could dramatically reduce cleaning overheads. So the natural question is: how good is cost-benefit selection?

In principle, if one were given a finite sequence of file writes, it would be possible to brute force the optimal cleaning cost by selecting all possible combinations of segments each time cleaning is invoked. One could then compare the best resultant write cost with the performance of the cost-benefit approach. Unfortunately, this appears to be an intractable problem. Even in a small-scale simulation where 5 segments are chosen out of 100 at each cleaning, the branching factor would be in the tens of millions: $\binom{100}{5} \approx 7.5 \times 10^7$. Unfortunately, I am aware of neither an NP-hardness proof nor a polynomial time solution.

In order to get a rough idea of how well cost-benefit performs and to set an upper bound on what the optimal write cost is, the simulator was modified to replay a sequence of writes and take into account the actual lifetimes of files while cleaning (I call this the “oracle” cleaner). This meant that segments could be selected with perfect knowledge of how long the live data would persist, and that surviving files could be sorted in perfect order of longevity.

I found that perfect sorting provided a considerable benefit when used with the cost-benefit policy based

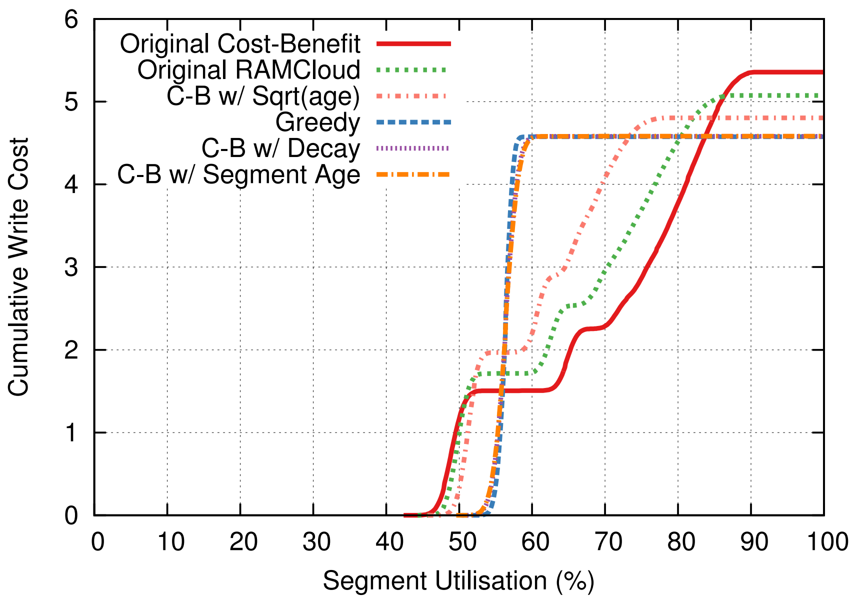


Figure 7.11: Cumulative distribution of work done during cleaning under the uniform random access pattern with various cost-benefit formulas at 75% disk utilisation. The decay-based and segment age-based policies behave nearly the same as the greedy policy. In contrast, the other policies all use file age to predict segment stability and suffer up to a 15% efficiency penalty.

on segment age (Equation 7.3). However, I was unable to significantly improve upon this by also taking the oracle into account when choosing segments to clean. Figure 7.13 depicts the improvement of the oracle cleaner across three access patterns. For example, although write costs with and without the oracle are similar below 50% utilisation, they diverge quickly thereafter in both the uniform and hot-and-cold distributions. In particular, cleaning without an oracle is 1.2 times as expensive at 70% utilisation and 2.5 times as expensive at 95% with hot-and-cold. Uniform has a moderately larger gap (up to 3.3x at 95%), which is to be expected given that the cleaner has no access locality to exploit.

Figures 7.14 and 7.15 depict the cumulative distribution of work and segments cleaned when the simulated disk is run at 75% utilisation. There are two interesting observations at each end of the locality spectrum. First, it is unlikely that the uniform random case could be improved without actually knowing future accesses; greedy is the best realistic policy if the future cannot be predicted. Second, the exponential distribution shows little benefit with perfect sorting. Figure 7.13 showed that this holds for all disk utilisations as well: the oracular cleaner experiences at most a 4% reduction in write cost under this distribution. This does not necessarily prove that cost-benefit is close to optimal in this case, but it suggests that it performs quite well in what might be an important access pattern for RAMCloud (multiple filesystem studies across several decades have found that file lifetimes follow hyperexponential distributions [67, 70]).

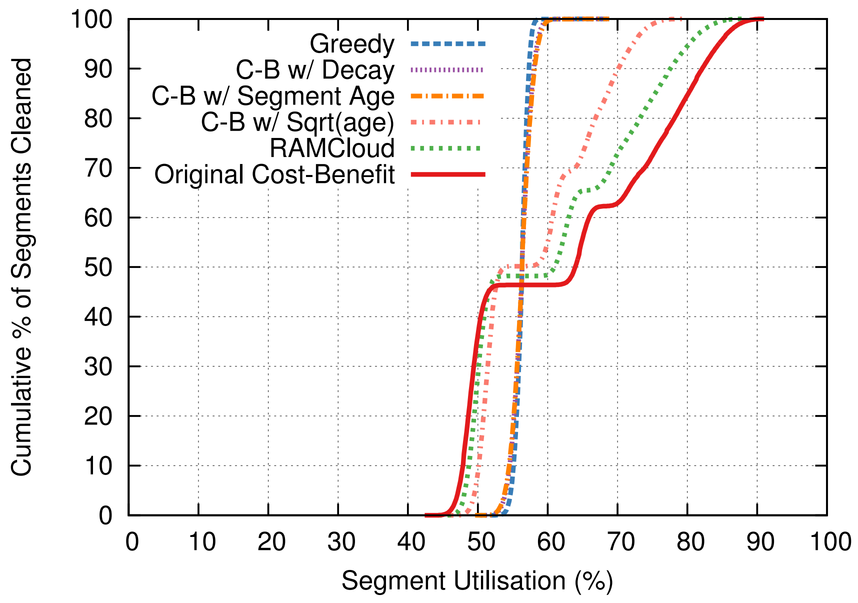


Figure 7.12: Cumulative percentage of segments cleaned in the uniform random experiments depicted in Figure 7.11. The bottom three curves represent policies that estimate segment stability using file age. Although they do clean nearly half of segments at a lower utilisation than the more greedy policies, this comes at the cost of cleaning the other half at higher utilisation.

7.5 Summary

As originally published, the LFS cost-benefit policy fails to make the right trade-off between segment utilisation and stability. This chapter showed how the failure manifests in two problems. First, it does not properly balance the age and utilisation components of the formula, resulting in cleaning that is up to half as efficient as previously reported. Second, using file age to predict future segment stability causes the policy to make poor and expensive selections – up to 2x as expensive as the greedy policy.

I have shown that simply substituting segment age for minimum file age in the cost-benefit formula appears to resolve both issues. Segment age balances better with segment utilisation and avoids cleaning segments at excessively high utilisation. It also naturally devolves into a greedy-like policy when there is no access locality, which is far more efficient (up to 2x).

Finally, some additional simulations that take future accesses into account when cleaning provide some limited answers to the question of optimality of the cost-benefit formula. Although the complexity of computing an optimal cleaning schedule is still an open question, my simulations provided an upper bound on optimal write cost and suggest that the cost-benefit approach performs very well when there is significant locality. However, when locality is reduced an oracular cleaner can perform significantly better (2-3x), so there may be room for improvement.

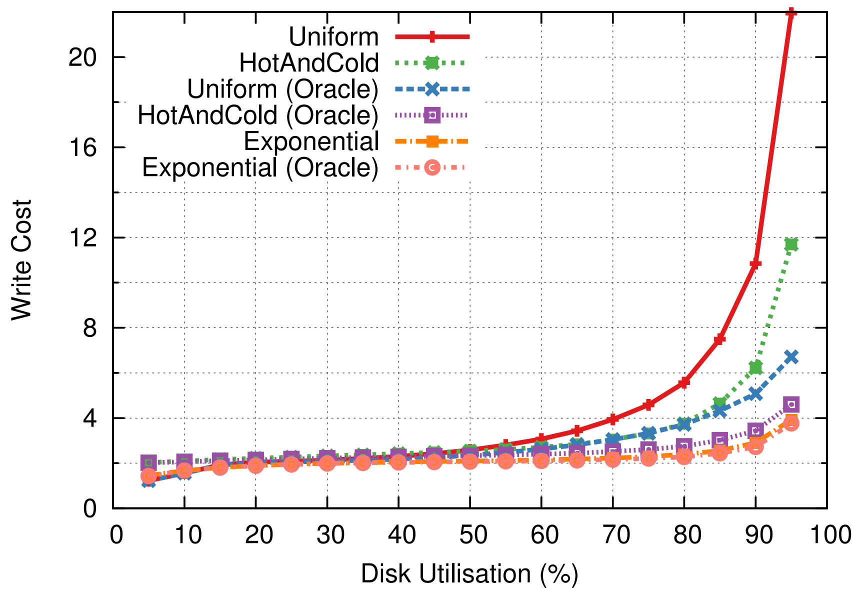


Figure 7.13: Comparison of write costs with and without perfect sorting of live files by remaining lifetime. In all cases the cost-benefit policy using segment age (Equation 7.3) was used. The "oracle" lines denote experiments where the cleaner was given the entire sequence of future writes and was able to sort live files perfectly by their remaining lifetime. The oracle results demonstrate that the current cost-benefit policy performs at least a factor of 2 worse than an optimal cleaner at high disk utilisation when locality is limited (uniform and hot-and-cold distributions). With high locality – the exponential distribution – the oracular cleaner is only marginally better.

Which access patterns will be most important to RAMCloud is still an open question. The only way to answer it will be to analyse real application traces when the system is deployed in production.

It is possible that the original LFS simulator implemented the segment age policy that this chapter advocates, rather than what was described in the literature. This hypothesis would not only explain the results in Figure 7.1, but also why one derivative of the simulator was described as using "the age of the segment" in the cost-benefit formula [54]. Attempts to locate either the original source code for the LFS simulator or its derivatives used in [33] and [54] were unsuccessful, and the authors involved did not have recollection of this apparent discrepancy. Whether this difference was known at the time, or overlooked is not clear. However, the original Sprite LFS implementation does not use this modified policy; it behaves just as the publications on it describe.

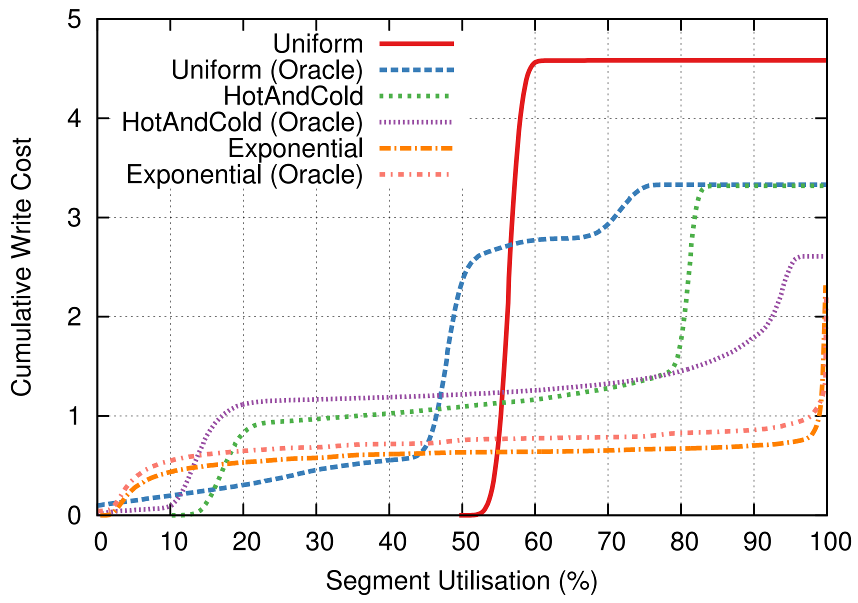


Figure 7.14: Cumulative distribution of work done during cleaning under the oracle experiments depicted in Figure 7.13 at 75% disk utilisation. The benefit of adding an oracle decreases as locality increases: perfect sorting of live files greatly benefits the uniform random distribution, but is of almost no benefit for the exponential distribution.

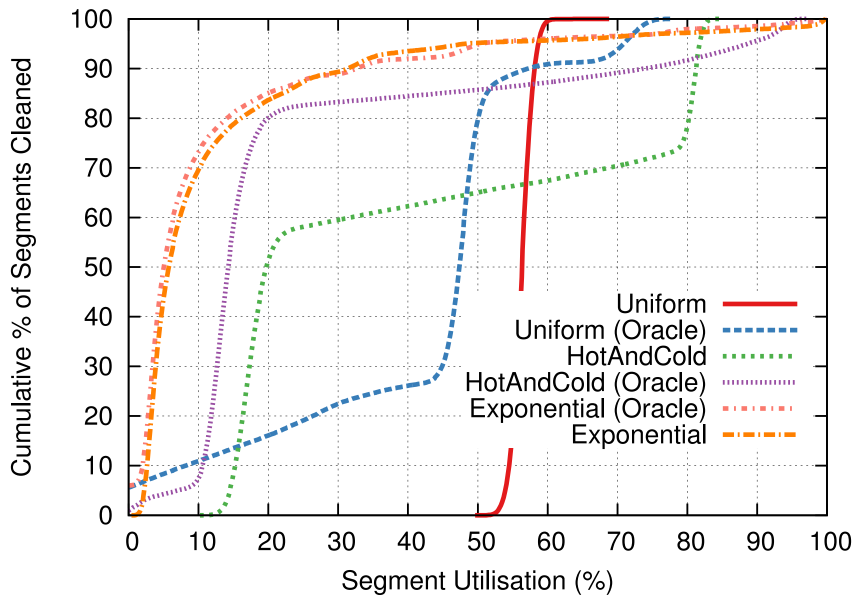


Figure 7.15: Cumulative percentage of segments cleaned in the oracle experiments depicted in Figure 7.13 at 75% disk utilisation.

Chapter 8

Evaluation

All of the features described in the previous chapters are implemented in RAMCloud version 1.0, which was tagged in January, 2014. The experiments in this chapter were conducted using a slightly older version of RAMCloud: git commit f52d5bcc014c (December 7th, 2013). See the Preface for information on how to access the repository.

In this chapter I explore a number of questions regarding log-structured memory's performance in RAMCloud, how RAMCloud compares to other storage systems, and whether the log-structured approach can be fruitfully applied to other storage systems as well. The chapter is structured as follows:

1. Section 8.1 answers a number of questions about RAMCloud's performance using a common set of simple synthetic workloads that evaluate various fixed combinations of server memory utilisation, object size, write locality, and write load. The questions, and a summary of their answers, include:
   - Section 8.1.2: How does increasing memory utilisation affect write throughput? We will see that, with a realistic degree of locality, throughput drops by about 30% for small and medium-sized objects when going from 30% to 90% memory utilisation. For larger objects, the penalty is considerably less (about 10%).
   - Section 8.1.3: How much does adding memory compaction (two-level cleaning) improve throughput and reduce backup I/O compared to a system that only uses combined cleaning? Results show throughput improvements of up to 5.5x, while backup I/O overheads are reduced by up to 76x.
   - Section 8.1.5: How well does parallel cleaning hide cleaning costs from clients? Experiments at 90% utilisation under a moderate write load show that cleaning adds only about 0.6% to the 90th percentile client write latency.
   - Section 8.1.6: How much faster can RAMCloud's cleaner free space when not restricted by backup I/O? Depending on the workload and memory utilisation, the cleaner can scale to handle write throughputs between 30% and 525% higher when not restricted by backup bandwidth.
   - Section 8.1.7: How well does RAMCloud's cleaner scale when using more processor cores to

perform memory compaction in parallel? This section shows that the cleaner scales well up to at least 3 or 4 cores. Further scaling could not be tested because the system eventually bottlenecks on appending to the log (the cleaner is able to generate free space faster than it can be consumed).

2. Section 8.2 explores how RAMCloud performs under the same changing workloads used to demonstrate the space inefficiency of other allocators in Section 3.2. RAMCloud is able to run all of the workloads at memory utilisations of up to 90%, trading 5-40% throughput (depending on write load) for 90% storage efficiency.

3. Section 8.3 addresses the question of how RAMCloud's performance compares to other systems. The section presents results from the popular YCSB [31] benchmark suite against RAMCloud, as well as Redis [6] and HyperDex [36]. RAMCloud out-performs HyperDex in all cases, and outperforms Redis in all read-dominated workloads. When using its native low-latency Infiniband support, RAMCloud can match Redis' write performance while providing better data durability and consistency guarantees.

4. Section 8.4 shows that log-structured memory has important uses beyond RAMCloud. It presents a straightforward adaptation of RAMCloud's log and cleaner to memcached [5], and shows that the logging approach can be up to 30% more space efficient than memcached's current memory management mechanism.

5. Section 8.5 summarises and concludes the chapter.

8.1 Performance Under Static Workloads

This section describes a series of experiments I ran to evaluate the throughput, latency and thread scalability of RAMCloud's log-structured memory under a number of static workloads. Many of these workloads are intentionally unrealistic and designed to stress the cleaner as much as possible in order to understand the performance limits of log-structured memory in RAMCloud.

8.1.1 Experimental Setup

The experiments were performed by dividing an 80-node cluster of identical commodity servers (Table 8.1) into groups of five servers. Different groups were used to measure different data points in parallel (running experiments sequentially would have taken hundreds of hours). Each data point was measured with multiple experiments on different groups on machines. 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 (there could be more or fewer available backups with bandwidth to spare, more or fewer disks installed in each backup, and so on). I experimented with

CPU	Xeon X3470 (Four 2.93 GHz cores, 3.6 GHz Max Turbo Frequency)
RAM	24 GB DDR3 at 800 MHz with 2 Memory Channels
Flash Disks	2x 128 GB Crucial M4 SSDs (CT128M4SSD2)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switch	Mellanox SX6036 (4X FDR)

Table 8.1: The server hardware configuration used for benchmarking. All nodes were connected to an Infiniband fabric with full bisection bandwidth (approximately 25 Gbps per node, due to PCI Express bus limitations). The SSDs provided about 125-150 MB/s of write bandwidth each (250-300 MB/s/server). Each server had approximately 10-11 GB/s of main memory bandwidth. All machines ran Linux 2.6.32.

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.

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

I varied the workload in four ways in order to measure system behaviour under different operating conditions:

1. **Object Size** In all experiments, the objects for each test had the same fixed data size. I ran different tests with sizes of 100, 1000, 10000, and 100,000 bytes (I have omitted the 100 KB measurements, since they were nearly identical to 10 KB). In addition to the data, each object had a fixed-length 8-byte key.

2. **Memory Utilisation** Each master server had a 16 GB in-memory log and was permitted up to a 48 GB on-disk log (a 3x *disk expansion factor*; see Section 4.3.3). I varied the percentage of live data (not including tombstones) from 30% to 90% (about 4.8 to 14.4 GB). Lower utilisations are cheaper to clean because segments tend to be less full. In some experiments, actual memory utilisation 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 Zipfian distribution in which 90% of accesses were made to 15% of the objects. The uniform random case represents a workload with no locality; the Zipfian case represents a more realistic degree of locality based on real-world measurements of memcached [5] deployments at Facebook [10].

4. **Stress Level** For most of the tests I 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. I also performed tests where the client issued one synchronous request at a time, with a single write

Object Size	CPU %	Disk Bandwidth	Disk Bandwidth %	Cleaner Bandwidth %
100 B	98.81%	607.47 MB/s	73.87%	62.63%
1,000 B	99.08%	736.75 MB/s	89.59%	20.72%
10,000 B	50.11%	830.79 MB/s	101.02%	20.29%

Table 8.2: CPU and disk bottlenecks under the Zipfian 90% utilisation experiments. "CPU %" represents the percentage of both cleaner cores in use; that is, 50% means that the cleaner used the equivalent of one core all of the time. "Disk Bandwidth" is the average rate of traffic to backup disks, including both new log heads and survivor segments. The value takes into account triple replication. "Disk Bandwidth %" represents how much of the estimated maximum available disk bandwidth was used. The 10,000-byte's bandwidth slightly exceeds 100% because the average bandwidth across the entire cluster was used as the divisor (the average, 822.39 MB/s, is slightly lower due to write bandwidth variation between disks). "Cleaner Bandwidth" shows what fraction of the disk traffic was due to writing survivor segments (the remainder was used to write new head segments).

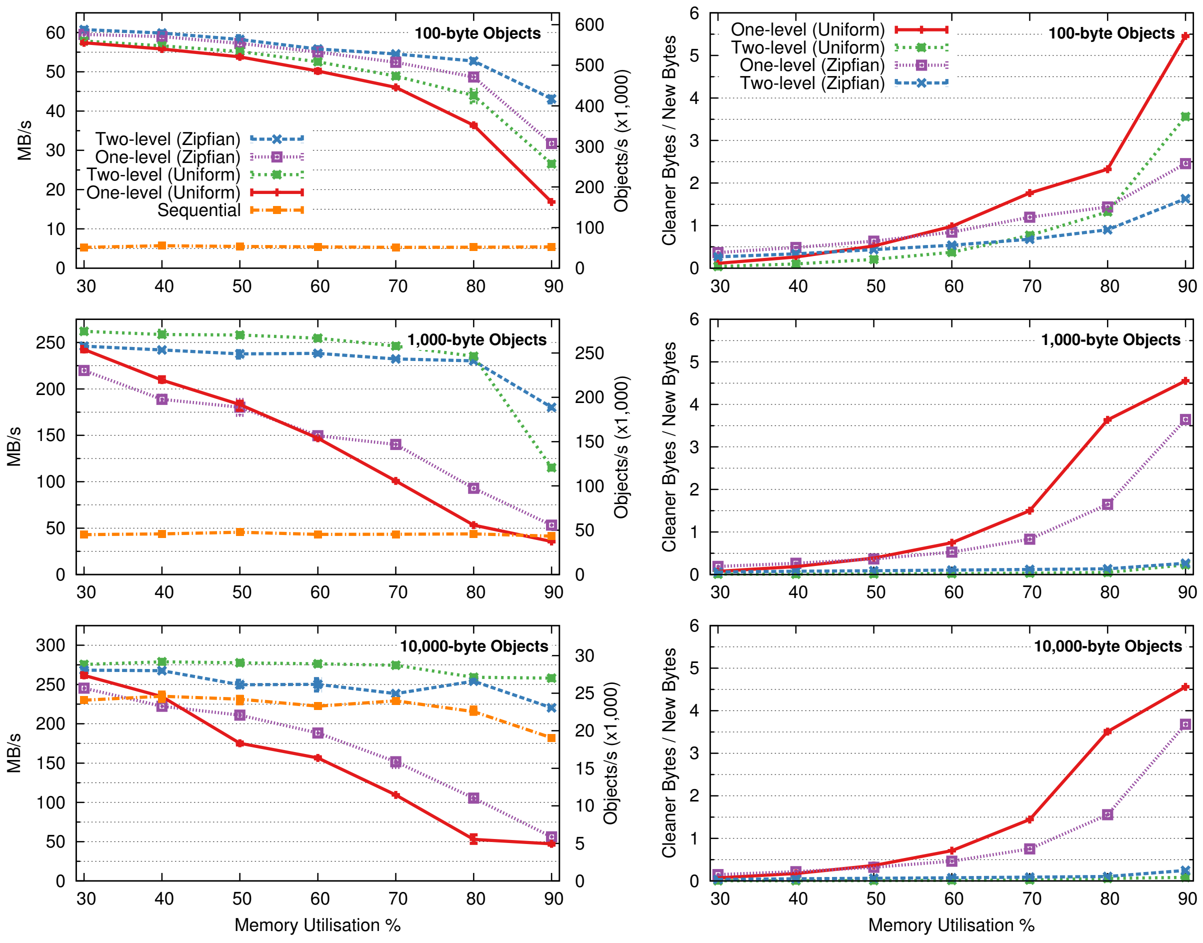
operation in each request; these tests are labelled "Sequential" in the graphs. I believe the Sequential tests are more indicative of actual RAMCloud workloads, but for most of our tests they did not stress the cleaner (RPC request processing dominated memory management overheads). We do not yet know the workloads that will run on real RAMCloud systems, so I would like RAMCloud to support a wide range of access patterns.

8.1.2 Performance vs. Memory Utilisation

The most important metric for log-structured memory is how it performs at high utilisation. All storage allocators perform worse as memory utilisation 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 utilisations in the range of 80-90%.

Figure 8.1(a) graphs the overall throughput of a RAMCloud master with different memory utilisations and workloads. In each experiment the master had a 16 GB log. The client initially filled the log with live data to the desired utilisation by writing objects in pseudo-random key order; then it overwrote the objects (maintaining a constant amount of live data) until the overhead for cleaning converged to a stable value. I chose a 16 GB log size because it resulted in a relatively large number of segments while also keeping experiments small enough to run on our 24 GB nodes and in a manageable amount of time (in production, RAMCloud servers are likely to be fitted with at least 64 GB of DRAM).

With two-level cleaning enabled, Figure 8.1(a) shows that client throughput drops only 10-20% as memory utilisation increases from 30% to 80%, even with an artificially high workload. Throughput drops more significantly at 90% utilisation: in the worst case (medium-sized objects with no locality), throughput at 90% utilisation is about half of what it is at 30%.



(a) Client Throughput

(b) Backup I/O Overhead

Figure 8.1: **(a)** End-to-end client write throughput as a function of memory utilisation. For some experiments ("One-level") 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. **(b)** Cleaner bandwidth overhead (ratio of cleaner bandwidth to regular log write bandwidth) for the workloads in Figure 8.1(a). 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 (and eventually to disk) while freeing segment space. The optimal ratio is 0. In both (a) and (b), each point represents the average of three runs on different groups of servers, with error bars for min and max.

At high utilisation the cleaner is limited by one of two bottlenecks, depending on the workload. For example, Table 8.2 shows the bandwidth and processor utilisation of the three Zipfian experiments at 90% utilisation, making the bottleneck in each case clear. With small and medium-sized objects (100 and 1,000 bytes), RAMCloud runs out of CPU resources. In these cases, the cleaner uses both of the two cores allocated to it nearly all of the time, while about a quarter of the total disk bandwidth is still available. Since the cleaner cannot keep up with write traffic, writes quickly exhaust all available segments. With large (10 KB) objects, the system needs only one core to keep up with the cleaning demand and bottlenecks on disk bandwidth instead.

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

If actual RAMCloud workloads are similar to our "Sequential" case, then it should be reasonable to run RAMCloud clusters at 90% memory utilisation. Even with 100B objects the cleaner is able to keep up with the throughput equivalent of between 4 and 8 single clients (depending on the access pattern) that issue continuous back-to-back writes. With 1000B objects it can keep up with the demand of about 3 to 4 such clients. If workloads include many bulk writes, like most of the measurements in Figure 8.1(a), then it would probably make more sense to run at 80% utilisation: 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 3.2, log-structured memory permits significantly higher memory utilisation: up to 90%, instead of 50% or less.

8.1.3 Benefits of Two-Level Cleaning

Figure 8.1(a) 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 experiments, the system used only the combined cleaner (never the memory compactor). The two-level cleaning approach provides a considerable performance improvement: across all experiments at 90% utilisation, client throughput is 3x higher on average (min 0.95x, max 5.5x) with two-level cleaning than one-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 8.1(b) shows that two-level cleaning reduces disk and network bandwidth overheads by up to 76x at high memory utilisations for the workloads in Figure 8.1(a). The greatest benefits occur for workloads with larger object sizes and/or no locality. For the 100B workloads, the combined cleaner must be run more frequently because tombstones consume a relatively large amount of space (they are about one third the size of each 100B object).

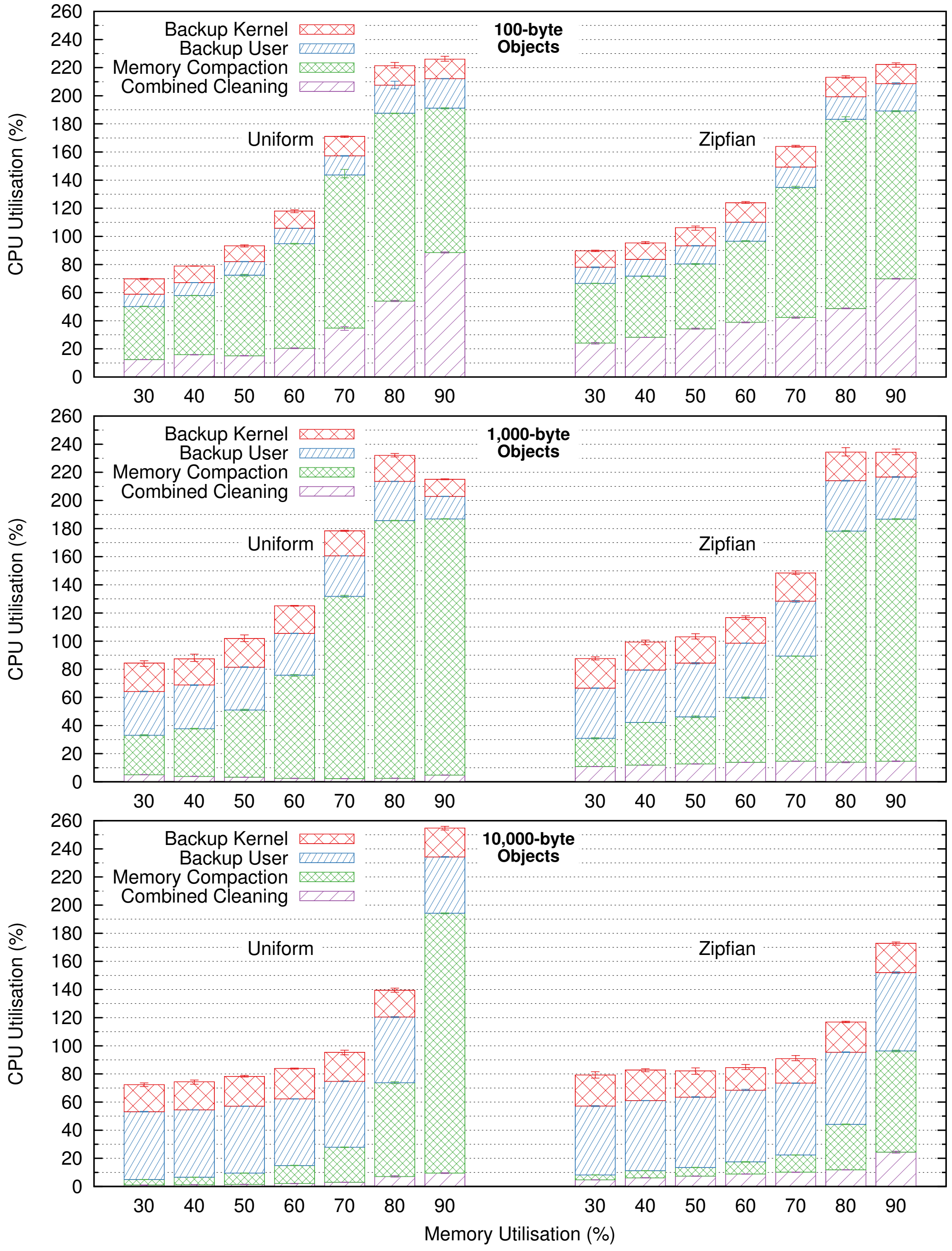


Figure 8.2: Average aggregate processor utilisations for the workloads in Figure 8.1 using two-level cleaning. Each bar represents the amount of CPU resources consumed by cleaning/compacting on the master, and all I/O handling on the three backups combined (including both cleaner and log write traffic). 100% means that in total an average of one core was actively handling cleaning and backup duties for the entire experiment across all machines. "Backup Kernel" is the amount of kernel time spent issuing I/Os to disks. "Backup User" is the time spent servicing segment write RPCs in the servers' backup modules. "Memory Compaction" and "Combined Cleaning" are the amount of time spent compacting memory and performing combined cleaning on the master. Each bar represents the average over 3 runs, with error bars for min and max.

Figure 8.2 details the amount of processor resources used in the experiments from Figure 8.1. In addition to overhead on the master from compaction and segment cleaning, writing to the log and performing combined cleaning uses CPU resources on the backups, which process RPCs to write new segments to disk and delete freed ones. At low memory utilisation a master under heavy load uses about 30-60% of one core for cleaning and backups account for the equivalent of at most 65% of one core across all three of them. As we also saw earlier in Table 8.2, 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 to the log head (Figure 8.1). 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 runs the combined cleaner more frequently 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.

8.1.4 Randomness vs. Locality

One surprising result in Figure 8.1(a) is that in several cases the Zipfian workload performed worse than a workload with no locality (see, for example, 10,000B objects with utilisations 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 accumulates rapidly in these segments. As a result, the cleaner can usually find segments to clean that have significantly 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 utilisation become more likely. In Figure 8.1(a), the uniform random behaviour of the "Uniform" workload was often more successful at generating very-low-utilisation segments than the "Zipfian" workload with much higher locality. As the scale of the system increases, the benefits of randomness should increase.

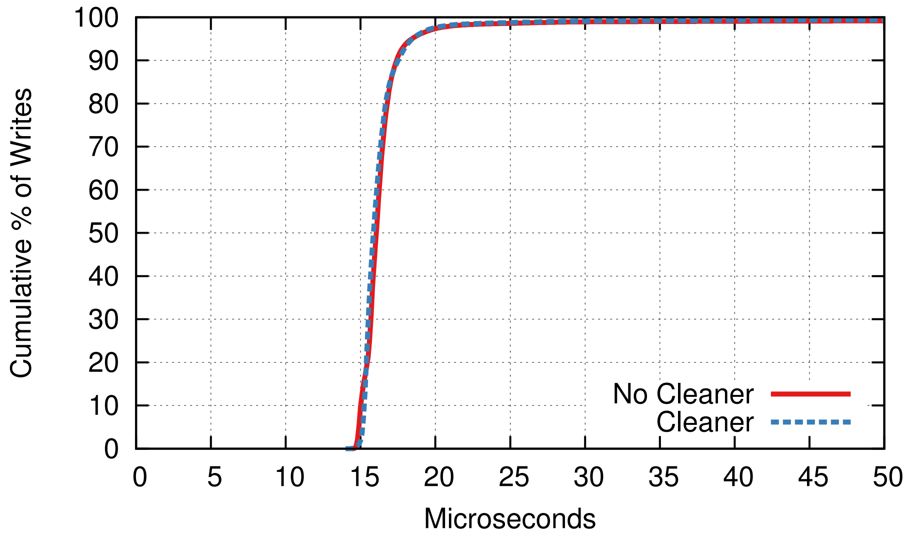


Figure 8.3: Cumulative distribution of client write latencies when a single client issues back-to-back write requests for 100-byte objects using the uniform distribution. For example, about 90% of all write requests completed in 17.5μs or less. The "No cleaner" curve was measured with cleaning disabled. The "Cleaner" curve shows write latencies at 90% memory utilisation with cleaning enabled. The 90th percentile latency with cleaning enabled was about 100 ns higher than without a cleaner. Each curve contains hundreds of millions of samples from multiple runs across different sets of identical nodes. Figure 8.4 more closely examines the tail of this distribution.

8.1.5 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. Figures 8.3, 8.4, and 8.5 graph the latency of client write requests in normal operation with a cleaner running, and also in a special setup where the cleaner was disabled. Even at 90% utilisation the distributions are nearly identical up to about the 99.9th percentile, and cleaning added only about 100ns to the 90th percentile latency, or about 0.6%. About 0.1% of write requests suffer an additional 1ms or greater delay when cleaning. Further 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. Figure 8.5 suggests that the latency overheads are not strongly correlated with memory utilisation, since the 99.9th and 99.999th percentile latencies are very similar from 70% through 90% utilisation.

8.1.6 Performance Without Replication

In order to evaluate the suitability of log-structured memory in a DRAM-only storage system (i.e. no backups), we first re-ran some of the experiments from Figure 8.1 with replication disabled. We also disabled

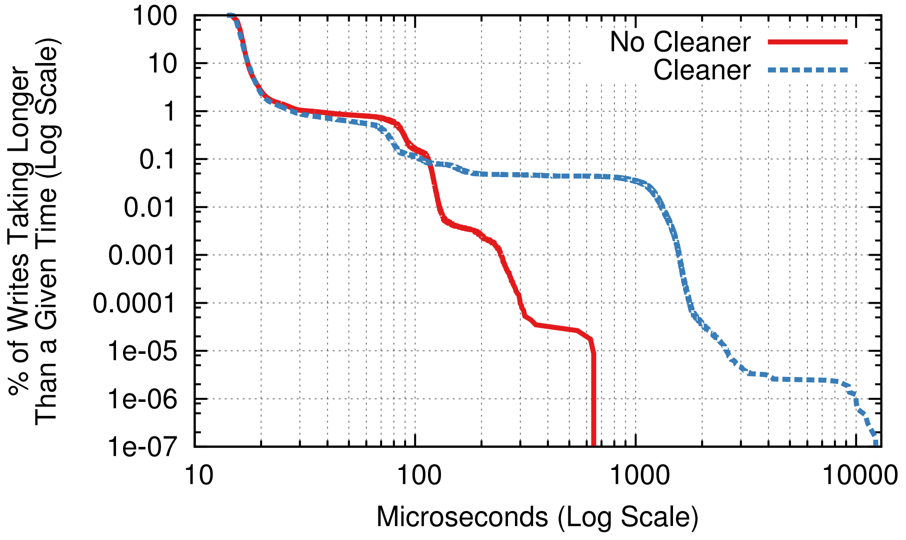
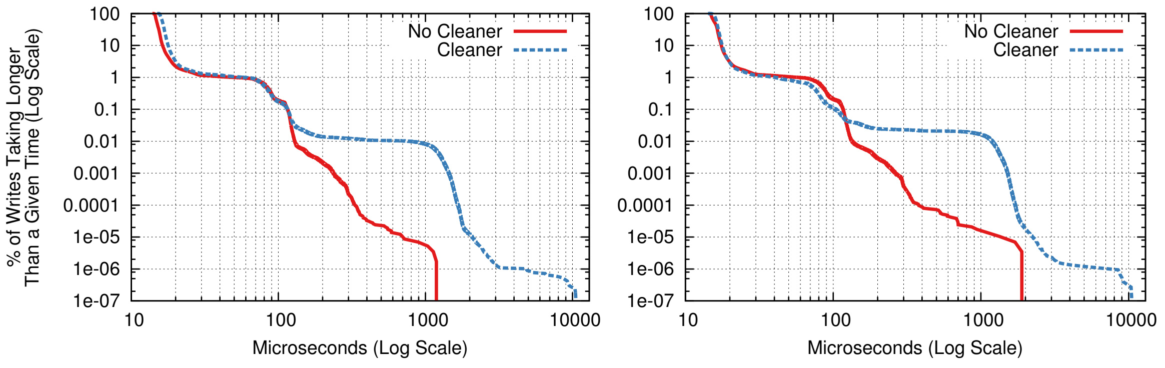


Figure 8.4: Reverse cumulative distribution of client write latencies when a single client issues back-to-back write requests for 100-byte objects using the uniform distribution. This is the same data from Figure 8.3, but plotted with a log-log cumulative distribution of client write latencies greater than x microseconds. For example, all accesses took longer than 14.0μs, and 2% of them took longer than 20.6μs. The 99.9th, 99.99th, and 99.999th percentiles are 114, 129, 245μs and 106, 1318, 1573μs for the "No cleaner" and "Cleaner" cases, respectively.



(a) 70% Memory Utilisation **(b)** 80% Memory Utilisation

Figure 8.5: The same experiments as in Figure 8.4, but at lower memory utilisation. (a)'s 99.9th, 99.99th, and 99.999th percentiles are 116, 129, 237μs and 115, 774, 1443μs for the "No cleaner" and "Cleaner" cases, respectively. (b)'s 99.9th, 99.99th, and 99.999th percentiles are 117, 135, 264μs and 104, 1190, 1517μs for the "No cleaner" and "Cleaner" cases, respectively.

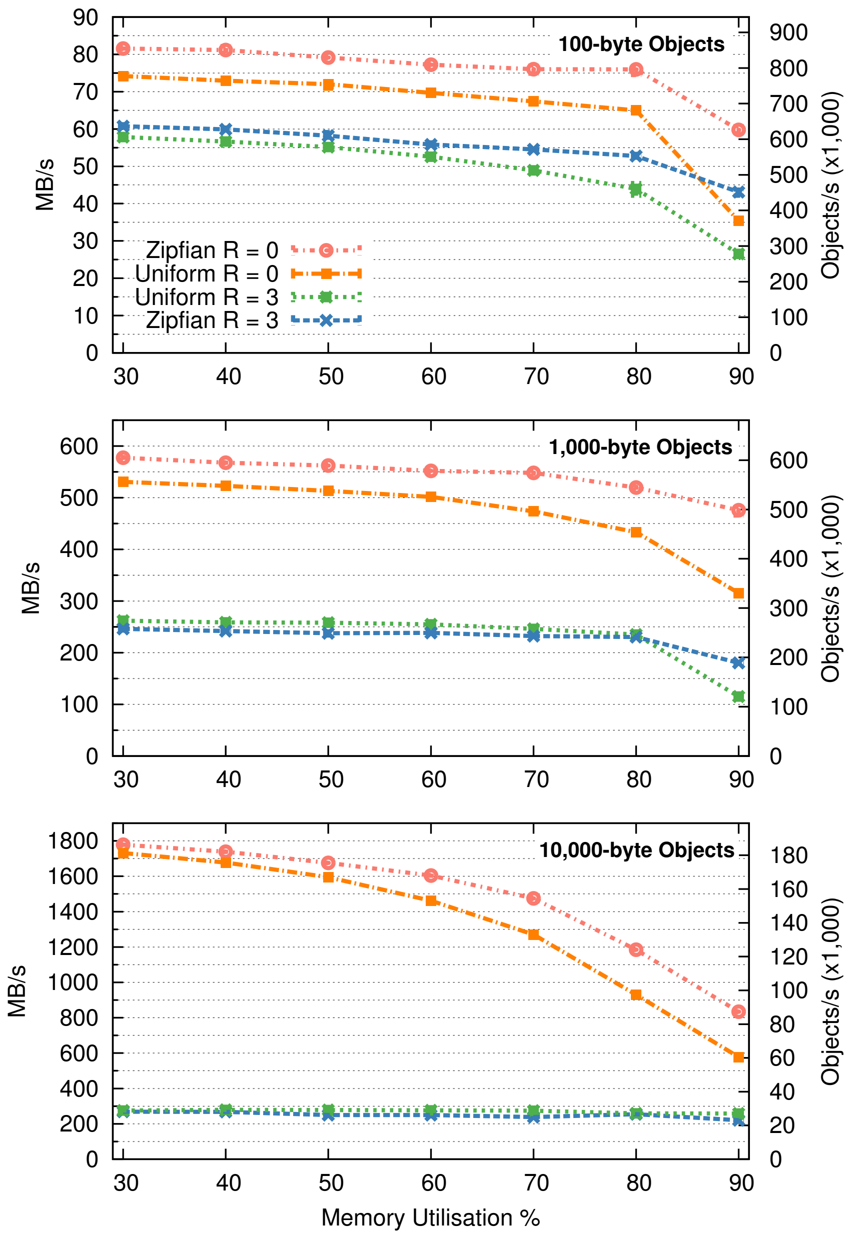


Figure 8.6: Client performance of two-level cleaning with ("R = 0") and without replication ("R = 3"). Values with replication are the same as in Figure 8.1(a) and experiments were run on the same hardware. When replication is disabled RAMCloud disables the memory compactor, since combined cleaning is more efficient (it reorganises objects by age and frees tombstones). The "Uniform R = 0" line dips drastically at 90% utilisation in the 100-byte case because up until 80% the cleaner was able to just keep up with demand. By 90% cleaning became approximately twice as expensive, leading to the significant drop. Each point is the average over 3 runs, with error bars for min and max.

compaction (since there is no backup I/O to conserve) and had the server run the combined cleaner on in-memory segments only. Figure 8.6 shows that without replication, log-structured memory supports significantly higher throughput: RAMCloud's single writer thread scales to nearly 850K 1,000-byte operations per second. Under very high memory pressure throughput drops by about 20-50% depending on access locality and object size. With 1,000 to 10,000-byte objects, 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.

When objects are small (100 bytes), running without replication provides about 30-40% more throughput than the replicated experiments for two reasons. First, when utilisations are low, the replicated cases are limited not by backup bandwidth, but by backup latency (at 30% utilisation, about 25% of the object write latency is due to backups, which accounts for most of the differential). Second, at higher utilisation (90%) with replication the cleaner bottlenecks on CPU throughput; more cores are needed to clean segments. With larger objects the differential is greater because cleaning is more efficient and replicated performance is bottlenecked primarily on backup bandwidth.

Without replication there are two bottlenecks. In the 100-byte and 1,000-byte cases the cleaner eventually runs out of threads to do the work needed. However, in the 10,000-byte cases, the system is limited instead by memory bandwidth. The X3470 systems used have a theoretical maximum 12.8 GB/s of memory bandwidth, and I measured maximum read throughput at about 11 GB/s and write bandwidth at 5 GB/s with lmbench [57]. I monitored performance counters on the servers' memory controllers during the 10,000-byte 90% utilisation experiments and found that about 9.1 GB/s were being read and written (about 53% reads). This is over 80% of the empirically-measured bandwidth and about 86% of the maximum write bandwidth. About 5 GB/s of bandwidth can be attributed to the cleaner copying live log entries, and another 2-3 GB/s, depending on whether the access pattern was Zipfian or Uniform, can be attributed to the network card DMAing write requests into memory, reading those objects, and appending them to the log.

The promising performance of log-structured memory in an unreplicated setting prompted us to adapt RAMCloud's log and cleaner to manage memory in the memcached [5] object caching server. Section 8.4 explains and evaluates this in detail.

8.1.7 Multiprocessor Scalability

Experiments without replication, as in Section 8.1.6, are also useful in evaluating the thread scalability of RAMCloud's cleaner implementation because they remove all backup bottlenecks that may limit write throughput. Figure 8.7 shows how client throughput increases as the number of cleaner threads is scaled from 1 up to 5 under a uniform workload with 100-byte objects at 90% memory utilisation. Up to about 4 cores the system is bottlenecked on the cleaner, and client throughput scales almost linearly. At about 410% cleaner CPU utilisation (an average of 4.1 cores running in the cleaner simultaneously) the system becomes bottlenecked on the sequential write path – RAMCloud cannot append objects to the log as fast as the cleaner can make more space, so performance plateaus.

This result also suggests that performance of the cleaner's memory compactor may scale well even with

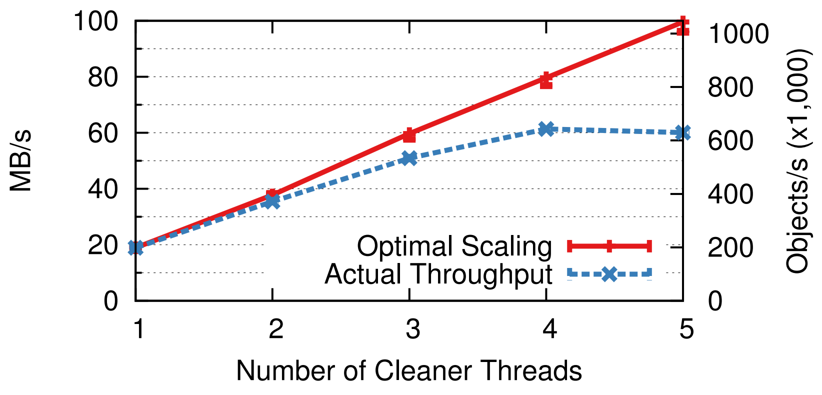


Figure 8.7: Cleaner thread scalability under a uniform 100-byte workload with no replication on an 8-core E5-2670 Xeon system. This workload is the same as in Figure 8.6 ("Uniform R = 0"), but results are not directly comparable as these experiments were run on somewhat different hardware. The cleaner scales well as it is given more cores (92% of optimal at three threads). By four threads the system is bottlenecked not on cleaning, but on appending entries to the log (RAMCloud does not currently process writes in parallel). Adding a fifth thread has no effect; the cleaner has no use for it and it sleeps. Each data point is the average of 3 runs with error bars for min and max.

replication enabled, however this will depend on the average object size, and therefore on the accumulation of tombstones. So long as objects are relatively large, tombstones will remain a small fraction of memory, and compaction will be efficient. However, if objects are small (100 bytes, for example), then tombstones will be relatively large in comparison and constitute a significant fraction of freeable space. This drives up the effective memory utilisation of compacted segments, making memory compaction increasingly expensive until combined disk cleaning is performed. Therefore, as more threads and cores are added, we might expect each core to have to work harder to reclaim the same amount of space, counteracting some of the benefit of having more compactors. The average object size that RAMCloud will be used to store in practice remains to be seen, so how many threads can be usefully applied to memory compaction in practice is not yet clear.

8.2 Performance Under Changing Workloads

Section 3.2 showed that changing workloads can result in poor memory utilisation with traditional storage allocators. For comparison, we ran those same workloads on RAMCloud. However, whereas Figure 3.1 measured the total amount of memory used by each allocator under a given workload, this experiment instead measures the performance of RAMCloud under each workload (at several different memory utilisations). The reason is that RAMCloud's memory efficiency is workload-independent; the system will store as much live information as it can in the memory it is given, trading performance for space-efficiency as utilisation increases and cleaning becomes more expensive. Therefore, the appropriate question here is not "how much

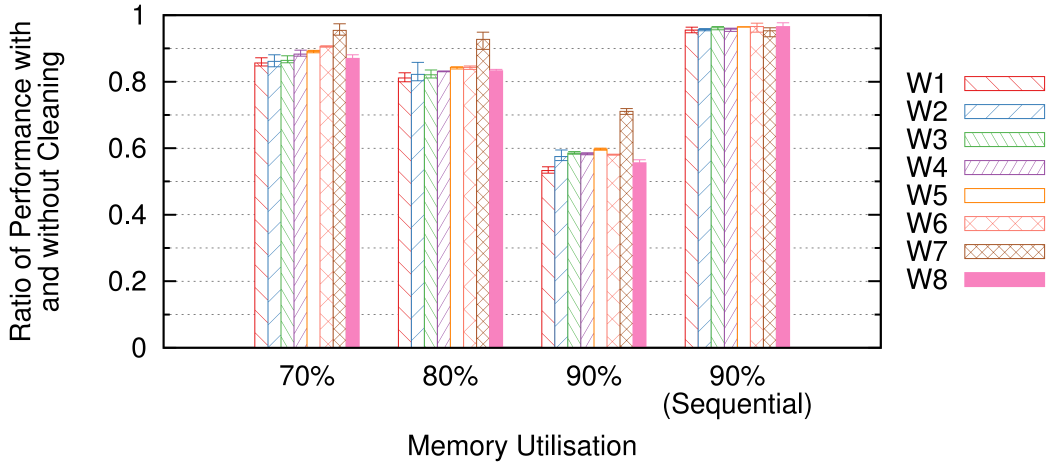


Figure 8.8: Client performance in RAMCloud under the same workloads as in Figure 3.1 from Section 3.2. Each bar measures the performance of a workload (with cleaning enabled) relative to the performance of the same workload with cleaning disabled. Higher is better and 1.0 is optimal; it means that the cleaner has no impact on the processing of normal requests. As in Figure 3.1, 100GB of allocations were made and at most 10GB of data was alive at once. The 70%, 80%, and 90% utilisation 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. Each bar is the average of 3 runs, with error bars for min and max.

space does RAMCloud need?", but rather "how much does throughput decrease as RAMCloud uses its space more efficiently?". Although we used the same general setup as for the earlier experiments in this chapter, in order to handle the large datasets we had to run the master on a different server, a Xeon E5-2670 system with 384 GB of DRAM (the same machine we used in Section 3.2).

We expected these workloads to exhibit performance similar to the workloads in Figure 8.1(a) (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 8.8 confirms this hypothesis: with the high-stress request pattern, performance degradation due to cleaning was about 5-15% at 70% utilisation and 30-45% at 90% utilisation. With the more representative "Sequential" request pattern, performance degradation was 5% or less, even at 90% utilisation.

8.3 Comparison to Other Systems Using YCSB

I used the Yahoo! Cloud Storage Benchmark (YCSB) [31] to evaluate the performance of RAMCloud under more realistic workloads, as well as to compare it against other storage systems. Two other modern storage systems were also benchmarked: Redis [6], a popular open-source in-memory store and HyperDex [36], a disk-based system that focuses on high performance and a rich data model with strong durability and consistency guarantees.

Workload	% Reads	% Updates	% Inserts	% Read-modify-writes	Description
A	50%	50%	0%	0%	Update-heavy
B	95%	5%	0%	0%	Read-mostly
C	100%	0%	0%	0%	Read-only
D	95%	0%	5%	0%	Read-latest
F	50%	0%	0%	50%	Read-modify-write

Table 8.3: YCSB workload characteristics. Each workload uses fixed 1,000-byte objects consisting of ten internal 100-byte fields. Records are accessed according to a Zipfian distribution in every workload except D, which always accesses the most recently written record. Before running the workloads, the initial working set of 10 million records was loaded into the storage system. D is the only workload that grows the working set with additional inserted records; all others maintain a constant 10 million records. Workloads were only modified to change the total number of objects and operations; all other parameters were left at their original values.

YCSB is a framework that generates a variety of access patterns designed to mimic typical real world applications in use at Yahoo! and other web companies. The benchmarks I ran used the pre-defined workloads A, B, C, D, and F (workload E was omitted because it relies on range queries, which RAMCloud does not support). Table 8.3 briefly summarises the individual workloads' characteristics.

8.3.1 Experimental Setup

An apples-to-apples comparison of RAMCloud to other storage systems is difficult given the spectrum of varying goals and design decisions. For example, while Redis is an in-memory store, it offers a richer data model than RAMCloud, but has weak consistency and durability guarantees. HyperDex also has a richer data model, and while it was designed to provide strong consistency and durability, it is not an in-memory store. For a more detailed discussion on how these systems relate to RAMCloud in design and functionality, see Chapter 9.

To make the comparison as close as possible, I used the same number of servers and clients for each system, and configured each storage system to use triple replication. When given a choice between configuration options, I favoured ones that would benefit Redis and HyperDex the most. This conservative approach was taken to avoid under-reporting their potential and provide stiff competition for RAMCloud. For example, since HyperDex is a disk-based store, it was run against a memory filesystem to ensure that, like Redis and RAMCloud, no reads would need to go to disk. This choice also provides HyperDex with a strong advantage in its write path, since Redis and RAMCloud were configured to write data to physical disks. Furthermore, each Redis and HyperDex server was configured to write one copy to local disk and replicate to two other servers. This meant that RAMCloud required 50% more network traffic for each write, since it would store one copy in DRAM and replicate to three other servers' disks.

All experiments were run on the same set of machines described in Table 8.1. A total of 37 distinct nodes were used: 12 servers were configured to run as storage nodes and 24 were used to run YCSB clients in parallel. For HyperDex and RAMCloud an additional server was used to run a cluster coordinator service. Redis has no central coordination; YCSB's Redis interface was modified to use static client-side sharding based on key hashes (the ShardedJedis class provided by the "Jedis" Java interface to Redis). The Redis YCSB interface was also modified to elide indexing operations needed to support the omitted range-scan workload, E. HyperDex's YCSB interface already implemented the same optimisation.

Each experiment was repeated five times as follows. Individual runs began by starting up fresh processes on each server and then loading the complete data set. Workloads were then run one after another in the recommended order A, B, C, F, D (workload D is the only one that inserts additional, new objects). In all cases an initial database of 10 million objects was used. Each YCSB client performed 10 million operations against the storage cluster using 8 threads in parallel. Across all 24 clients, each workload consisted of 240 million operations. The 10 million object data set size was chosen so that HyperDex could fit all "persistent" data in the 16 GB memory filesystem on each server.

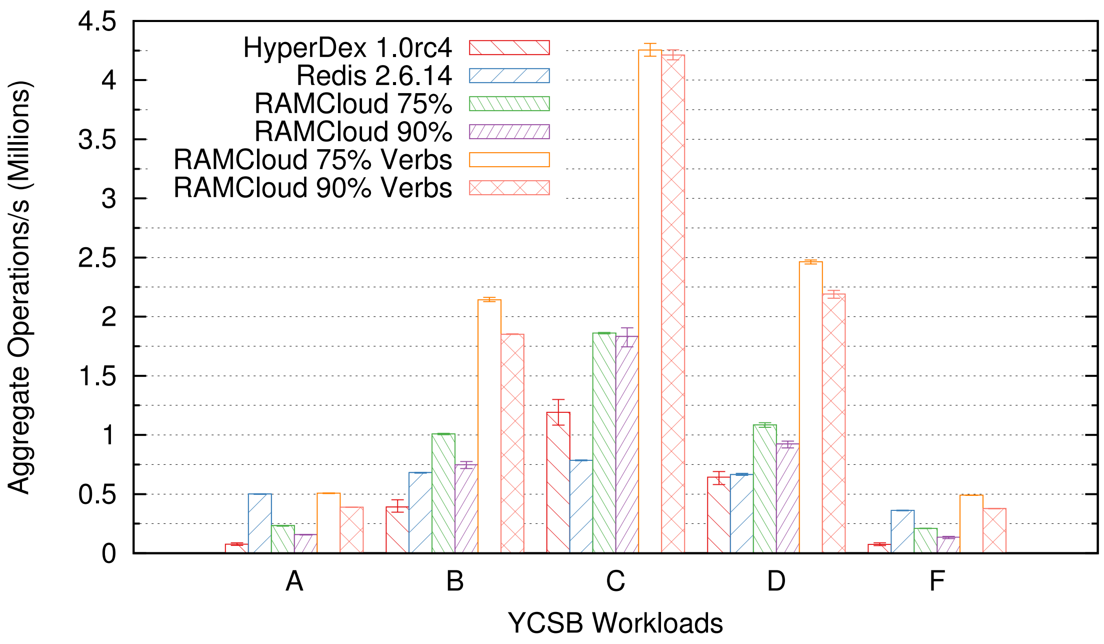
RAMCloud was configured with each server acting as both a master and a backup (backups used only one of the two disks in the server), and the the log replication factor was set to 3. Redis was configured to write updates asynchronously to an "append only log file" on a Linux ext4 filesystem with a one-second fsync interval, making its use of disks roughly comparable to RAMCloud's buffered logging technique. Redis only supports asynchronous master-slave replication, so to achieve triple disk redundancy each server machine ran a Redis master as well as two Redis slaves that replicated for masters on two other machines. The slaves were not used to service read requests. As mentioned above, HyperDex was configured to write to a RAM-based "tmpfs" filesystem.

Since Redis and HyperDex do not have Infiniband support, separate RAMCloud experiments were run both with kernel sockets over Infiniband and with RAMCloud's native Infiniband transport layer that takes advantage of direct access to the network card for zero-copy and low-latency I/O. Redis and HyperDex also used the Infiniband fabric, but only via the kernel socket layer since neither natively supports Infiniband. Experiments were also run with RAMCloud configured such that each master's memory would run at 75% and 90% utilisation after loading the 10 million objects. This was done to see the effect of memory pressure on performance.

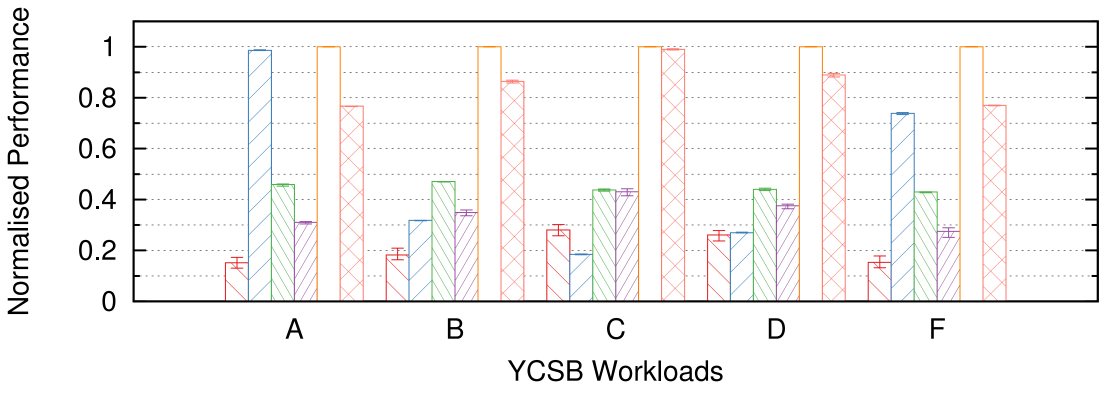
8.3.2 Results

Figure 8.9 depicts the performance of HyperDex, Redis, and RAMCloud across the five YCSB workloads listed in Table 8.3.

RAMCloud performs consistently better than HyperDex: between 1.4 to 3x as fast over kernel sockets. In particular, the largest performance differences are in write-heavy workloads A and F, despite RAMCloud writing to disk and HyperDex to a RAM-based filesystem. In read-heavy workloads, RAMCloud provides 1.4 to 2.6x as much throughput as HyperDex via the same kernel sockets.



(a) YCSB Performance in Millions of Operations Per Second



(b) YCSB Performance Normalised to the “RAMCloud 75% Verbs” Configuration

Figure 8.9: **(a)** Performance of HyperDex, RAMCloud, and Redis under the YCSB workloads described in Table 8.3. Values on the y-axis are the aggregate average throughputs of all 24 YCSB clients. HyperDex and Redis were run using kernel-level sockets over Infiniband. The “RAMCloud 75%” and “RAMCloud 90%” bars represent RAMCloud running 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 depict RAMCloud running with its “kernel bypass” user-level Infiniband transport layer, which uses reliably-connected queue pairs via the Infiniband “Verbs” API. **(b)** The same data represented in **(a)**, but normalised to the RAMCloud results running with the Infiniband Verbs API and at 75% memory utilisation (“RAMCloud 75% Verbs”). For example, in workload C, HyperDex achieves about 30% of the throughput of this particular configuration of RAMCloud under the same workload. Each graph plots the average of three runs with error bars for min and max.

RAMCloud also out-performs Redis in read-dominated workloads, where B, C, and D are 1.1 to 2.4x as fast as Redis (using kernel sockets). However, RAMCloud pays about a 2-3x performance penalty compared to Redis in write-heavy workloads A and F because Redis' asynchronous replication results in much more efficient batched writes (at the cost of durability). RAMCloud is also at a significant disadvantage in the read-modify-write workload (F), since the server does not support multiple named fields per object. Instead, the RAMCloud interface to YCSB emulates this on the client side by packing all of the ten fields into one object. This means that every update requires a read-modify-write, and that each read-modify-write that YCSB performs at a higher layer results in an additional read in the RAMCloud interface layer. Both Redis and HyperDex servers support multiple fields per object and require only one round-trip per update.

RAMCloud offers good performance even at high memory utilisation. In particular, at 90% utilisation RAMCloud incurs at most a 36% performance penalty over running at 75% utilisation. For workloads B, C, and D, which have a higher read-to-write ratio, the performance degradation is less than 26%.

Finally, when using the user-level Infiniband stack, RAMCloud's performance increases 2.1 to 2.8x over kernel-level sockets. Write throughput degrades less at higher utilisation, with a 23% drop going from 75% to 90% utilisation, as opposed to a 36% reduction with kernel sockets. Write-heavy workloads benefit particularly well from the reduced communication latency since RAMCloud will only reply to a write RPC once the modification has been sent to and acknowledged by three backups. In fact, with this transport RAMCloud's write performance is nearly on par with Redis. Although Redis would likely show some improvement if ported to the Verbs API, the fact that it batches during replication means that its performance is much less reliant on latency than is RAMCloud's. Therefore a latency reduction is unlikely to provide Redis a similarly large percentage increase in overall write performance.

8.4 Using Log-structured Memory in Other Systems

An important question is whether the log-structured approach can be fruitfully applied to other memory-based storage systems. To address this question I modified the memcached [5] 1.4.15 object caching server to use RAMCloud's log allocator and cleaner to manage memory and compared this prototype to the original memcached, which uses a slab-based [21] allocator. This section details how memcached was modified to use log-structured memory and the effects on performance and efficiency.

8.4.1 Log-based memcached

Memcached's architecture is similar to a RAMCloud master server with no backup mechanism: objects are stored in memory and indexed by a hash table based on a string key. This made porting RAMCloud's log-based allocation relatively straightforward: less than 350 lines of changes were made to memcached's source, including instrumentation for performance measurements, and only very minor modifications to the adapted RAMCloud log and cleaner sources were needed. None of the replication code needed to be adapted since memcached provides no persistence.

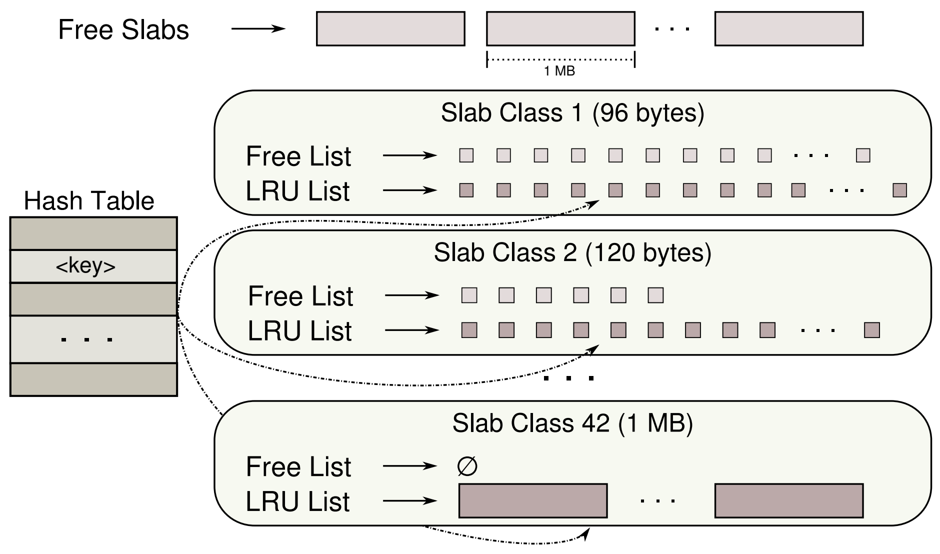


Figure 8.10: Memory management in memcached. 1 MB contiguous *slabs* of memory are allocated to individual *slab classes*, each of which subdivides its slabs into fixed-sized pieces that are used to allocate space for individual objects. Slab classes consist mainly of a free list of deleted objects and an LRU list of live objects, which is used to evict cold objects if the free list is empty. Slabs are never freed once allocated to a slab class. However, they may occasionally move between classes if the *slab rebalancer* is enabled (moving a slab necessitates evicting all objects stored in it first). A hash table indexes all live objects by user-specified string keys.

To understand the problems with memcached's allocator, we must first review how it works (Figure 8.10). memcached divides memory into 1 MB *slabs* and allocates them to buckets called *slab classes*. Each slab class chops up the slabs it owns into fixed-sized chunks that are used to service allocations. Classes consist primarily of a free list for available chunks and an LRU list for allocated chunks that can be evicted if the system runs out of memory. Different slab classes have different chunk sizes; by default, memcached creates 42 exponentially larger classes from 96 bytes to 1 MB where the chunk size for each class is 25% larger than the chunk size for the previous slab class. When memory is requested from the allocator, only the smallest slab class that could fit the request is used. If the appropriate class has no free chunks, it attempts to allocate another slab to chop up and repopulate its free list. If there are no more slabs available, then it tries to evict an object from its LRU list. Memory allocation only fails if there are no objects to evict and no free slabs to allocate (the allocator does not fall back to allocating from larger slab classes).

There are two problems with this design that log-structured memory overcomes. First, the per-object overhead is high: the *item* structure, which precedes each object stored in memory is 48 bytes. Some of this space is important object metadata, but much of it consists of fields used by the allocator even when not on a free list (for example, LRU list pointers and the identity of the slab class the object was allocated from). As observed by Fan et al [38], shrinking this structure can greatly increase the number of small objects each server can store, and may correspondingly improve cache hit rates.

The second problem is fragmentation: workload changes can cause the slab allocator to fail to satisfy allocations because there are no objects of the desired size to evict [59]. For example, if a memcached server is out of memory and a workload change results in new objects that use an empty slab class, then the server will fail to allocate space for them because there are no free slabs and nothing available to LRU. To address this, both the open source and Facebook internal versions of memcached have implemented "slab rebalancers", which detect when allocations fail due to workload changes and reallocate memory from other slab classes. These mechanisms select a slab in another class that contains infrequently-accessed objects and evicts every object in the slab so that the slab can be reallocated to another class. See [59] for details on Facebook's rebalancing policy.

In contrast to the slab allocator, the modified log-based memcached manages memory very differently. Rather than maintaining LRU and free lists, it sequentially allocates all objects from the head segment just like RAMCloud. Also like RAMCloud, when free space runs low it invokes the cleaner. However, unlike RAMCloud this cleaner reclaims space not only from deleted objects, but also evicts cold objects to ensure that space is available for new cache insertions. The cleaner was slightly modified to support eviction and implements the following policies. Segments are selected for cleaning based on how many objects were accessed in each segment since the last cleaning pass: the colder the segment (the fewer times its objects have been read), the less impact evicting objects from it will have on cache hit performance. Once segments are selected, objects are sorted by descending access time and the cleaner writes out at most 75% of the hottest objects (by space) to survivor segments. This ensures that each cleaning pass reclaims 25% of the segment space processed while avoiding the eviction of popular objects. Seglets and memory compaction were disabled since there is no cleaner disk or network I/O to reduce.

This alternative design resolves both the overhead and fragmentation problems as follows. First, the *item* header size is halved to 24 bytes by removing four unnecessary fields: two LRU list pointers, a reference counter, and the slab class identifier. The LRU pointers were not needed because the cleaner evicts objects during cleaning as described above. The slab class field was superfluous (the log-based allocator has no classes) and the reference counter was also unnecessary, since, like in RAMCloud, the cleaner needs to know only when an entire segment is no longer being accessed, rather than each individual object. The cleaner also actively defragments memory, so it naturally resolves the slab allocator's fragmentation problem, too.

8.4.2 Evaluation

To evaluate the modified memcached server I conducted several experiments using YCSB-generated workloads. In each test I ran a memcached server with 2 GB of cache storage and replayed a workload via a single memcached client that used libmemcached and 16 threads to read and write objects. The client behaved like a typical memcached application: if a read request were to fail (a cache miss), it would insert the object into the cache. Both client and server were run on the same 16-core Linux server as described in Section 8.2. Memcached was configured to use up to 8 threads and the slab rebalancer was enabled.

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

Table 8.4: Average number of objects stored per megabyte of cache. The log allocator can store nearly 31% more small objects due to reductions in per-object metadata, even when there is very little internal fragmentation in the slab allocator (the 25-byte objects fit perfectly in the 96-byte slab class). Under the Zipfian-distributed object sizes, nearly 15% more objects can be stored by eliminating internal fragmentation within slab classes. Results were averaged over 5 runs.

Memory Efficiency

The first experiment measured the memory efficiency improvement of reducing the *item* header size. A 100 million object write workload was used, where each object consisted of a 23-byte key and a data blob. Experiments were run with both fixed 25-byte data blobs and Zipfian-distributed blobs from 0 bytes to 8 KB (median 67, average 863). The fixed experiments simulate a very simple workload whereas the Zipfian experiments simulate more complex applications that store a range of variable-sized data, most of which is relatively small.

Table 8.4 shows the improvement in memory efficiency that the log-based approach provides over the default slab allocator. Nearly 31% more small fixed-sized objects and 15% more Zipfian-sized objects can be stored using the log approach. In the fixed object case, the space savings is due entirely to reducing the item header's size, rather than any other losses such as wasted space when rounding to the allocation size of the appropriate slab class. Each object requires 96 bytes of memory using the slab allocator, including header, key, and data blob. This means that each object fits perfectly in the first slab class, so there is no space lost to internal fragmentation. On the other hand, the log-based server requires only 72 bytes per object stored, so the slab allocator uses 33% more space per object. The observed 31% improvement in objects stored is slightly less than the expected 33% due to entry header overheads in the log (see Section 4.1.1). The efficiency improvement in the variably-sized object case is due primarily to the fact that the logging approach avoids internal fragmentation, rather than its shrinking of the item header. For example, an object that requires 800 bytes of memory fits imperfectly into the smallest fitting 944-byte slab class, wasting nearly 15% space in internal fragmentation. Memcached can be tuned to increase the number of slab classes and therefore reduce this fragmentation loss, but this comes at a trade-off: the more slab classes there are, the more sensitive the system will be to changing size distributions, which results in more rebalancing.

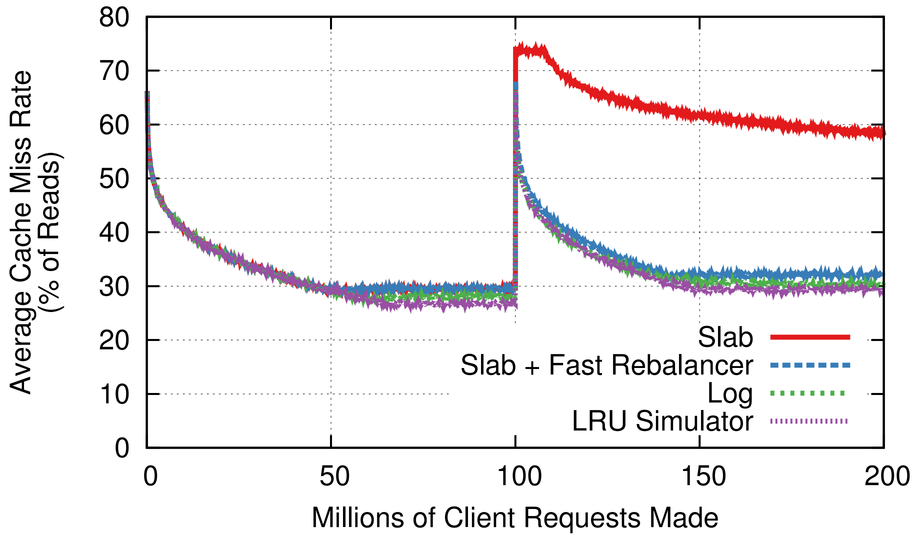


Figure 8.11: Comparison of cache miss rates under a changing access pattern. The "Slab" and "Log" lines represent the regular memcached and log-based memcached miss rates, respectively. The "Slab + Fast Rebalancer" line represents the slab allocator with artificial restrictions removed (the original memcached code will move at most one slab every ten seconds). The "LRU Simulator" line represents an optimal miss rate calculated by a simulator that uses a global LRU eviction policy and assumes a 48-byte item header for each object and no memory lost to fragmentation. Each line is the average of 5 runs and consists of 2000 samples.

Susceptibility to Changing Access Patterns

Next I compared the miss rates of the original memcached and log-based servers across a changing access pattern. A workload of 200 million operations across a potential set of 100 million objects was generated. Keys were accessed according to a Zipfian distribution, and 95% of accesses were reads while the other 5% were writes. Like the previous experiment, each object initially consisted of a 23-byte key and 25 bytes of data. However, after the first 100 million operations, the object data size was increased to 50 bytes. At this point, any read returning a 25-byte object was considered a cache miss and the client would attempt to replace it with the new 50-byte version.

Figure 8.11 shows that the log-based approach adapts to the workload change better than the slab rebalancer. The experiment begins with an empty cache, so miss rates are initially high. The depicted miss rate is always less than 100% because each data point is averaged over 100,000 requests (approximately 300 ms). All servers converge to between 26 and 32% misses before the workload changes, causing the rate to spike. The original memcached slab allocator fails to adapt quickly because the rebalancer is hard-coded to move at most one slab every ten seconds. Its miss rate remains high because the rebalancer limits the number of larger objects that can be stored. Removing this artificial limitation greatly improves the rebalancer's ability

Allocator	Average Throughput (writes/sec x1000)	% CPU in Cleaning/Rebalancing
Slab	259.9 ± 0.6	0%
Log	268.0 ± 0.6	5.37 ± 0.3 %

Table 8.5: Average throughput and percentage of CPU used for cleaning or rebalancing. Log-structured memory imposed no performance penalty and the cleaner contributed to only a small fraction of the process' CPU use. Results were averaged over 5 runs.

to handle the workload change, though it does lead to some instability (the server occasionally crashes). It is unclear why this limit was put in place and whether or not such an accelerated rebalancer would work well in practice. Perhaps the limit is intended to avoid excessive rebalancing, or to paper over a concurrency bug that caused instability. Regardless, the log-based server still maintains about a 2% lower cache miss rate, primarily because it wastes less memory and can store more objects.

CPU Overhead of Cleaning

The final experiment addressed the question of whether cleaning adds significant overhead that could degrade write throughput under heavy load. For this test the same Zipfian workload was used as in the memory efficiency experiment: 100 million writes, Zipfian-distributed sizes from 0 to 8 KB and Zipfian-distributed key popularity. This unrealistic workload was designed to maximally stress the servers' write path. Table 8.5 compares the total client write throughput of the original and log-based servers, as well as the percentage of CPU used by the cleaner. The results show that under an all-write workload there was no loss in throughput when using log-structured memory (in fact, throughput improved slightly by about 3%). Furthermore, only 5% of the process' CPU time was spent in the cleaner. Measurements of Facebook workloads [10] have shown that the majority of access patterns feature a much lower percentage of write requests, so the expected overhead should be negligible under typical conditions.

Memcached Should Have Used Log-structured Memory

The results from this memcached experiment indicate that the system would have been better off using a log-structured approach over the slab allocator. Log-structured memory increases storage efficiency considerably, with minimal CPU and no throughput overheads. It deals well with changes in access patterns and avoids internal fragmentation when objects fit imperfectly into their slab class. Moreover, memcached's slab rebalancer is essentially a cleaner tacked on the side to deal with exceptional cases, so they have already paid the implementation complexity cost of cleaning without taking advantage of its other benefits.

8.5 Summary

This chapter asked a number of questions concerning RAMCloud and its log-structured memory implementation, and it answered them with experimental data. For instance, we found that under a realistic level of access locality, RAMCloud's throughput drops by only between 10 and 30% when increasing memory utilisation from 30 to 90%. Two-level cleaning is critical for maintaining good throughput at high memory utilisations, as it reduces I/O overheads by up to 76x, preserving more bandwidth for new writes to the log head. Furthermore, parallel cleaning is able to hide much of the cost of cleaning from clients. In one experiment, for example, the 90th percentile client write latency increased marginally by about 100 ns (less than 1%) with cleaning enabled.

This chapter also compared RAMCloud's performance to two other storage systems: Redis [6] and HyperDex [36]. RAMCloud out-performed HyperDex in all cases, Redis in all read-dominated workloads, and matched Redis' performance in writes while using RAMCloud's native Infiniband support. Low-latency RPCs are critical for allowing RAMCloud to provide both high throughput and strong durability guarantees. Redis sacrifices the latter to improve the former, but RAMCloud can equal the performance without compromising.

Finally, this chapter showed that the log-structured memory technique is useful in other storage systems, even if they are not durable. In particular, experiments run against a prototype of memcached [5] that uses a log and cleaner to manage memory show that logging increases memory efficiency up to 31% with no loss in throughput and minimal CPU overhead.

Chapter 9

Related Work

Both RAMCloud in general, and log-structured memory in particular, have been influenced greatly by other systems and techniques. This chapter discusses how a number of them relate and provides greater context for RAMCloud as a system, as well as the log-structured memory techniques presented in this dissertation.

The chapter is split into several subsections:

1. Section 9.1 discusses other storage systems that use DRAM as the primary storage medium, such as in-memory databases, caches, and web search.
2. Section 9.2 compares RAMCloud to so-called "NoSQL" storage systems. Like RAMCloud, these systems generally provide simplified data models and APIs (as opposed to full-blown SQL query engines).
3. Section 9.3 explains how RAMCloud's push for low latency can be traced back to a number of earlier systems that foresaw the importance of network latency in tightly-coupled distributed systems.
4. Section 9.4 describes related systems that use log-structured approaches to managing data on disk.
5. Section 9.5 discusses how garbage collecting allocators compare to logging and cleaning in RAMCloud.
6. Section 9.6 summarises this chapter.

9.1 Use of DRAM in Databases and Storage Systems

DRAM has long been used to improve performance in main-memory database systems [35, 42]. Even thirty years ago, researchers noted that main memories had grown capacious enough to store useful datasets mostly, or even entirely, in memory. This spurred significant research into optimising database implementations for fast random access memory, rather than slow, high-latency rotating disks. Changing the underlying storage medium meant revisiting design decisions all the way down to fundamental data structures. For example, in some cases simple balanced binary trees suddenly became preferable to the popular shallow B+-trees originally chosen to minimise expensive disk I/Os.

More recently, large-scale web applications have rekindled interest in DRAM-based storage. By combining decades of hardware improvements with clusters of commodity computers, far more enormous datasets could be maintained in memory (in some cases, such as billion-user social networks, at scales of the entire planet). Some of the earliest systems to exploit DRAM on the web were search engines [15], which needed to serve requests entirely out of main memory in order to keep up with ever-increasing user demand for finding relevant information among an exponentially increasing corpus of data. As the Internet transitioned from limited research and industrial use to widespread commercial and now increasingly social use, DRAM-based storage has been crucial in providing the data throughput necessary to scale with both growth in users and the increasing sophistication of web applications.

In addition to special-purpose systems like search, modern general-purpose storage systems have also embraced main-memory storage. For example, along with RAMCloud, both H-Store [51] and Google's Bigtable [25] are designed to support many different applications and keep part or all of their data in memory to maximise performance. Furthermore, main-memory caching systems such as memcached [5] have become critical pieces of infrastructure, especially for companies whose applications aggregate large numbers of small bits of information, such as social networks like Facebook [59]. RAMCloud's key differentiator from these systems is its focus on low latency, microsecond-level access to data over the network.

9.2 NoSQL Storage Systems

RAMCloud's simplified data model is similar to many so-called "NoSQL" storage systems. These systems are typified by the abandonment of complex and sophisticated data models and query languages in favour of other goals such as very high performance and sometimes very strong durability guarantees (geo-replication to survive entire datacentre failures, for instance). Because NoSQL systems often focus on performance, it is not surprising that some of them also use DRAM as the location of record for all data. Some even sacrifice durability guarantees [6] for higher performance. However, unlike RAMCloud, many of these systems were designed for disk-based storage, which means that they are incapable of the same aggressive, single-digit microsecond access times.

One in-memory NoSQL store is Redis [6]. Redis has a richer data model than RAMCloud, allowing users to store abstract data structures like hash tables, lists, and other simple primitives. However, Redis has a number of downsides compared to RAMCloud. For example, durability and consistency guarantees are weaker. Redis supports a "persistence log" for durability, similar to RAMCloud, but logging is asynchronous by default, meaning that "completed" operations may not survive a server crash. Redis also does not stripe the log across a cluster as RAMCloud does with segments; instead, it logs to the servers' local filesystems. To survive a disk crash, either a replicated network filesystem must be used to store the log, or each Redis server must be mirrored (asynchronously, and therefore inconsistently) to another server. This second option means that data is replicated in DRAM, which increases the total system cost or reduces capacity. Also, unlike RAMCloud, Redis does not perform cleaning to reclaim free space, making its log management much more expensive (instead, Redis rewrites the entire log at once in the background, copying all live data each time it reclaims space).

Memcached [5] is another in-memory system with a simple key-value data model similar to RAMCloud's. However, memcached is only a cache, and provides no durability guarantees whatsoever. Instead, it is primarily used to reduce query load on slower SQL databases (which are responsible for storing data durably). Unlike with RAMCloud, cache coherence can be a major concern in systems like memcached, since the cache needs to be kept consistent with other tiers of storage. For example, since data usually lives in a separate permanent storage system (like a SQL database), care must be taken to ensure that the caching tier stays consistent with the underlying storage tier by issuing cache shoot-downs when data in the storage tier changes. Another problem with multi-tiered caching systems is that they can be fragile. If cache hit rates drop (due to failures, bugs, misconfigurations, etc), then load is redirected to the much slower back-end databases, which may be unable to keep up with rising demand. Such a problem was responsible for at least one major Facebook outage [2]. RAMCloud avoids the consistency and stability problems because it is a single tier. By offering both durability and high performance, there is no underlying storage system to maintain consistency with and to shield from load.

There are many other NoSQL storage systems, some of which differ even more considerably from RAMCloud. Google's Bigtable [25], for instance, is a distributed key-value storage system for massive datasets. Bigtable has a number of interesting properties. For example, each value consists of an arbitrary number of columns, like a database, but queries can only be made on keys, not on the values in each column. Also, updates to Bigtable columns are timestamped, and previous values, if any, can be accessed in addition to the current value (depending on the garbage collection policy for old values). Bigtable also allows users to pin tables in memory for increased lookup performance.

Other NoSQL examples include Dynamo [34] and PNUTS [30], which also have simpler data models than traditional databases, but do not service all reads from memory. Both Dynamo and PNUTS were designed for geo-replicated durability, but both are disk-based, and not strictly consistent. Instead, applications sometimes have to resolve data inconsistencies due to things like machine, network, or datacentre failures, or high latency. Geo-replication also means that these systems are designed to run at timescales of milliseconds, rather than RAMCloud's microseconds. However, at present RAMCloud has no support for tolerating entire datacentre failures.

NoSQL systems are continuing to evolve. One recent trend has been to add back some of the functionality that was lost when complex data models and query languages were abandoned. For example, HyperDex [36] is a disk-based NoSQL system that offers similar durability and consistency to RAMCloud, but supports a richer data model, including range scans and efficient searches across multiple table columns. Another example is Google's Spanner [32], a geo-replicated datacentre storage system that supports transactions and SQL-like queries. Such systems strive to provide the performance and scalability of NoSQL systems, without jettisoning so many of the useful database primitives that application programmers have long relied on.

To my knowledge, RAMCloud is the only NoSQL system to provide strong durability and consistency coupled with high throughput and low latency. In particular, the focus on latency is a major differentiator; no other system has the same aggressive microsecond round-trip time goals as RAMCloud (including other DRAM-based stores like Redis). Furthermore, no other system uses a log-structured approach for managing memory, let alone a two-level approach that combines memory and disk management. Redis is distributed with and uses jemalloc [37] by default because it "has proven to have fewer fragmentation problems than libc malloc", but as demonstrated earlier (Section 3.2), any non-copying allocator will suffer significant memory efficiency pathologies that RAMCloud avoids.

9.3 Low Latency in Distributed Systems

RAMCloud is neither the first nor only system to acknowledge the importance of low latency for building distributed systems [53]. A number of projects throughout the 1980s and 1990s sought to provide low latency interconnection of commodity computers in order to build distributed systems that were both higher performance and more tightly coupled. Many of these ideas have since been adopted in the supercomputing space with interconnect technologies like Infiniband [74]. More recently, as explained in Section 2.1, some of the same ideas have started entering the commodity networking space, promising much wider adoption in the future.

On the hardware side, work in low latency routers [7] and network interfaces [23, 39] has reduced the time taken to switch packets within the network as well as minimised the overhead in handing packets between software layers and the network interface hardware within servers. Common practices include reducing or eliminating buffering within switches ("cut- through" and "wormhole" switching) and minimising the number of bus transactions needed to move data between server software and network cards.

On the software side, a number of projects have tried to give applications low-latency access to network resources by providing more efficient library interfaces (often ones that explicitly avoid expensive kernel crossings [23, 78]). For example, U-Net [78] noted in the mid-90s that software was not keeping up with advances in networking and was therefore introducing an artificial bottleneck. By providing direct user-level access to the network interface, they were able to achieve round-trip message latencies as low as 65μs – more than an order of magnitude faster than using the operating system kernel's networking API.

A pervasive theme in this area has been the co-development of hardware and software. After all, a low-latency OS network stack is useless without an efficient hardware interface, and vice-versa. As a result, much of the prior work has resorted to designing custom software interfaces, custom hardware, and sometimes custom network protocols as well [23, 78, 39, 79].

Unfortunately, incompatibility with existing software/hardware and the need to adapt applications to new networking APIs makes it particularly difficult to move these techniques out of research labs and supercomputers and into commodity systems. Attempts have been made to provide compatibility shim layers for legacy applications (such as “fast sockets” [66] and Infiniband’s “sockets direct protocol”), but these necessarily sacrifice some performance.

Perhaps a better explanation for the historically slow transition of low latency interconnects into the commodity space is the gulf between the supercomputing and web computing communities. Infiniband has been popular for years in the high-performance computing space, supporting scientific applications such as large-scale physics simulations, but has not gained traction in the very different commodity web application space, where Ethernet has long dominated. With very different tools, applications, requirements, and motives, it might be difficult to foresee the benefits of applying high-performance computer technologies to problems facing web applications. As a specific example, it is interesting to note that much of the early work on low latency networking for cluster computing came out of Active Messages [79], originally a supercomputing technology, but has been slow to gain commercial traction outside of the supercomputing domain.

RAMCloud has so far been a user of low-latency networking technologies, rather than a contributor towards advancements in this area. However, I expect that our early adoption of low-latency networking in the web space will spur additional innovations, particularly if we are successful and other storage systems adopt similarly stringent latency goals, and therefore much greater demands of the underlying network.

9.4 Log-structured Storage on Disk

The log-structured memory design in RAMCloud was heavily influenced by ideas introduced in log-structured filesystems [69], 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 for backup I/O when doing regular log appends.

Efficiency has been a controversial topic in log-structured filesystems [72, 73], and additional techniques were later introduced to reduce or hide the cost of cleaning [20, 54]. 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 utilisation, 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.

Ideas from log-structured filesystems have found a particularly fruitful niche recently in flash-based storage systems. For example, logging techniques are often used inside solid state disks (SSDs) because flash cells cannot be updated in place; they must be erased before they can be written to, making an append-only log an excellent fit. To support legacy filesystems, SSDs feature “flash translation layers” that emulate a 512-byte or 4 KB block device on top of flash memory [41]. These layers must perform actions that move live data around, similar to cleaning, because the unit of erasure in flash is much larger than the write block size (the devices must relocate the live blocks in an erasure cell so that unused blocks can be reclaimed by erasing it entirely). Like with cleaning, the expense of moving live data in order to free space has a significant impact on device write throughput. Moreover, since flash cells have a limited number of write-erase cycles, efficient cleaning is imperative for longevity, as well as performance.

RAMCloud is not the first distributed storage system to use logging. Zebra [45] was a network filesystem that treated distinct servers like RAID does individual disks in an array: Zebra striped parity information across multiple remote nodes to get high throughput. RAMCloud uses replication, rather than coding, to protect its data, but it effectively stripes its log across many different backups by placing segments on different sets of servers. Log-structured techniques have also been used more recently in distributed flash-based storage systems such as Corfu [13]. Corfu provides a durable, high throughput, distributed log designed to be used to implement higher-level shared data structures [14].

RAMCloud is also similar to Bigtable [25] in that both use a log-structured approach to durably store data. For example, writes to Bigtable are first logged to GFS [43] and then stored in a DRAM buffer on the Bigtable server. In some ways, however, the similarity is largely superficial. Bigtable has several different mechanisms referred to as “compactions”, which flush the DRAM buffer to a different GFS file when it grows too large (an *sstable*). Compactions also reduce the number of sstables on disk and reclaim space used by “delete entries” (analogous to tombstones). Bigtable’s compactions serve a different purpose than RAMCloud’s. Unlike RAMCloud, compactions do not reduce backup I/O, nor is it clear that they improve memory efficiency. Most of Bigtable’s compactions serve to manage disk space only, making them more akin to traditional log cleaning in disk-based filesystems.

Microsoft’s Windows Azure Storage [24] is a datacentre storage system similar to RAMCloud in several ways (it was also designed and built contemporaneously). In particular, Azure uses a similar log structure at its core, which is broken up into 1 GB “extents” (equivalent to segments). Azure differs most fundamentally in its goals: it is a disk-based system that provides several different durable data models and provides cross-datacentre replication; it was not designed for low latency. This leads to a number of internal differences. For example, unlike RAMCloud it uses a central service to determine and track where extents are stored. Azure also uses a form of cleaning, but it is not clear how it is implemented.

9.5 Garbage Collectors

Log-structured memory is similar to copying garbage collectors found in many common programming languages such as Java and LISP [50, 80]. For example, garbage collected languages frequently allocate memory from large contiguous regions using “bump-a-pointer” allocation, a very fast technique that is similar to appending to a head segment in RAMCloud. In order to ensure that there are contiguous spans of memory to allocate from, these collectors reclaim space in ways similar to log cleaning that involve defragmenting memory and relocating live allocations. Moreover, many garbage collectors use *generational* techniques to reduce the cost of defragmenting memory [77]. That is, they segregate allocations into different generations based on how long they have been alive. This way, generations containing young objects are likely to become sparsely populated as their objects die off quickly and are therefore cheaper to garbage collect. This is much like the segregation that RAMCloud performs on objects during combined cleaning when it sorts live objects by age before writing them to survivors. As noted in the LFS work [68], while garbage collectors often have only a few generations, each segment in RAMCloud can be considered a generation unto itself.

Although log-structured memory is similar to many copying garbage collectors in design, the motivations for the two differ significantly. The point of automating memory management with garbage collectors is to make programming safer and easier. By letting the runtime worry about managing unused memory, common forms of memory leaks can be avoided entirely. Furthermore, since application code need not explicitly free memory, software interfaces become simpler and programming less onerous (for example, there is no need to worry about keeping track of when an allocation is no longer needed, especially as it is passed or shared between different modules). Log-structured memory, on the other hand, was designed to provide high performance and high memory efficiency in a storage system. The point was not to simplify application writing (objects in RAMCloud must be explicitly deleted), but to ensure that DRAM is used as efficiently as possible.

In terms of design, log-structured memory differs from garbage collectors primarily in that it was designed for storage systems that make restricted use of pointers. All general-purpose garbage collectors must at some point scan all of memory to determine liveness of allocations. In log-structured memory, however, only one pointer must be checked to determine if data is alive, and one pointer must be changed to relocate it. This greatly increases collection efficiency. In comparison, garbage collectors must necessarily go to great lengths to determine liveness (they need information from the entire reference graph to be sure that an allocation is no longer in use) and must update any number of references if an allocation is moved[1]. Log-structured memory, on the other hand, is fundamentally more incremental due to its restricted use of pointers; it never needs to scan all of memory to free space, and therefore has a key performance advantage over general purpose garbage collectors.

In garbage collectors, determining liveness and updating references becomes especially complicated when applications are permitted to run either concurrently with collection or may be interleaved with it. In these cases, the garbage collector must be able to deal with the fact that the application (*mutator*), can attempt to modify allocations as they are being moved, can change the reference graph as it is being walked, and so on. A great deal of prior work has focused on these problems [52, 63, 64, 76, 29, 48]. In contrast, log-structured memory skirts these issues for two reasons. First, object pointers are located only in the hash table and objects may not refer to one another directly. As a result, the reference graph is trivial and managing concurrency is greatly simplified. Second, objects are never updated in place, so there is no need to worry about concurrently writing and relocating an object (updating an object implicitly relocates it).

[1]In some cases these updates may be done lazily, by unmapping the virtual memory that backs an allocation and handling page faults, or by encoding extra information in pointers that can be used to check if a referred-to allocation has since been moved before accessing it[76].

9.6 Summary

Both RAMCloud in general and log-structured memory in particular share much in common with prior systems and techniques. The use of DRAM as the primary location of record for fast access to datasets can be traced back at least three decades to work in in-memory databases. Moreover, RAMCloud's simplified data model is reminiscent of many so-called NoSQL storage systems, which have chosen to sacrifice complex data structures and query models for other benefits including higher performance and geo-replicated durability. RAMCloud's focus on low latency is also not new. Supercomputing has relied on fast inter-machine communication for decades, and in the early 1990s researchers noted the importance of fast communication for building high performance, tightly-coupled distributed systems. Finally, our log-structured memory technique has a clear genesis in log-structured filesystems and copying garbage collectors.

RAMCloud stands out among other storage systems primarily due to its focus on extremely low latency data access. RAMCloud assumes that latency-reducing technologies from the supercomputing space will continue to permeate commodity networking, enabling much lower latency communication than other systems have been built to expect. A main hypothesis behind RAMCloud is that this extreme change in web datacentre storage performance will spur the creation of novel, highly-scalable applications that are not possible today.

Although log-structured memory builds upon work in log-structured filesystems and garbage collectors, it extends these techniques to manage both memory and disk space in a durable, DRAM-based, key-value storage system. Doing so while maintaining high performance, durability, and consistency required new techniques such as two-level cleaning and buffered logging, and led to an important advancement in segment selection during cleaning (Chapter 7). Finally, the restricted use of pointers in storage systems fundamentally allows log-structured memory to operate more efficiently than general purpose garbage collectors can.

Chapter 10

Conclusion

This dissertation introduced log-structured memory – a technique for managing memory in DRAM-based storage systems – and described and evaluated the log-structured memory implementation in RAMCloud. This approach was key in attaining a number of important and orthogonal goals in RAMCloud, including high write performance, excellent memory efficiency, and durability.

This work also introduced a number of important mechanisms that make log-structured memory particularly effective. For example, two-level cleaning substantially reduces the disk I/O overheads of reclaiming free space. This approach saves more bandwidth for client write requests and makes it feasible to use a unified logging approach to manage both memory and disk resources. Parallel cleaning is another important technique that increases cleaner throughput while isolating regular requests from most of the overheads involved in collecting free space.

When we began designing RAMCloud, it seemed only natural to use a logging approach on disk to back up the data stored in main memory. After all, logging has been used for decades to ensure durability and consistency in many different storage systems. However, it was surprising to discover that logging also makes sense as a technique for managing the data in DRAM.

In retrospect, a log-structured approach may seem obvious. Ever since log-structured filesystems were first introduced, the parallel has been drawn to copying garbage collectors used to manage memory in many common programming languages. After all, writing to the head of a log is basically bump-a-pointer memory allocation, and cleaning is a form of generational garbage collection.

One could reasonably claim that log-structured filesystems were as much an application of memory management techniques to filesystem design as it was of database techniques. In that sense, log-structured memory has brought the idea full circle, back to managing memory. Only this time, unlike garbage collectors, it does so in a specialised setting and is able to exploit this fact to improve memory efficiency and performance.

What was not so obvious from the outset was that log-structured memory would perform very well even at high memory utilisation, a scenario in which general-purpose garbage collectors struggle significantly. Log-structured memory is able to scale past the point where traditional garbage collectors fail because it takes advantage of the restricted use of pointers in storage systems. Since there is only one pointer to each object and pointers are in a fixed location (the hash table), log-structured memory eliminates the global memory scans that fundamentally plague existing garbage collectors. The result is an efficient and highly incremental form of copying garbage collector that allows memory to be used very efficiently (80-90% utilisation).

A pleasant side effect of log-structured memory was that, thanks to two-level cleaning, we were able to use a single technique for managing both disk and main memory, with small policy differences that optimise the usage of each medium. This simplicity was one of the original motivations behind a log-structured approach. The other motivation was that we could not see anything obviously wrong with using an in-memory log and had a hunch that it might have significant benefits beyond easing RAMCloud's implementation. In time we began to appreciate how crucial a copying allocator is for memory efficiency, and how well a copying memory manager can perform when pointer use is restricted.

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 many DRAM-based storage systems. For instance, I showed in Chapter 8 that memcached could benefit significantly by adopting our techniques. I hope that the ideas presented here will power future systems, and moreover, that they will stimulate improvements and altogether superior techniques.

10.1 Future Directions

This dissertation leaves a number of interesting and worthwhile questions unanswered in the context of RAMCloud's log-structured memory. This section discusses several of them, including ones particular to RAMCloud, as well as to log-structured memory in general.

10.1.1 Cleaning and Log Structure in RAMCloud

One of the biggest open questions in my mind is whether it would have been better to read segments from disk during cleaning. As mentioned in Section 6.1.6, this would have allowed RAMCloud to avoid storing tombstones in memory, which would have made memory compaction more efficient. Moreover, without tombstones the system might only need a very simple balancer that runs the combined cleaner only when disk space runs low (since there would be no need to take into account the adverse effect of accumulating tombstones on compaction performance; see Chapter 5 for details). Although reading segments from disk would increase the I/O overhead of combined cleaning, this would be counter-balanced by more efficient memory compaction.

The main argument against reading segments from disk is that it would slow recovery from backup failures. The reason is that masters would not be able to safely re-replicate segments from their in-memory copies (since the necessary tombstones would not be present). As a result, segments lost due to backup failure would need to be re-read from other backups' disks containing their replicas. This is not a problem if the lost segments are scattered widely and the work can be distributed across many machines. However, in addition to random scattering of segments, RAMCloud also supports a copyset [28] replication scheme. This scheme greatly reduces the occurrence of data loss events, but a consequence is that when one backup fails, only $R - 1$ other nodes in the cluster contain replicas for the lost segments. With substantially fewer machines to do the work of reading the replicas, recovery would take much longer. The scheme can be made more complicated, by trading off additional scattering of replicas across more machines with more frequent data loss events. However, it is unclear how much complexity this would add, and whether it would be worth it.

Another important question is how to scale RAMCloud's write throughout with multiple threads. The problem is that right now each RAMCloud master currently has a single log, which can be appended to by only one thread at a time. Multiple service threads may process write RPCs simultaneously, but they are serialised by the log module when storing object data. This artificially limits server throughput, especially when clients perform small single object writes. Moreover, it means that small writes may be delayed a long time while they wait for large writes to complete (a classic problem of "elephants" and "mice"). I have not addressed how to remove this bottleneck in this dissertation. One solution could be to have multiple logs per master so that threads could write to their own local head segments. Another possibility might be to allow multiple heads in the same log (after all, parallel cleaning effectively does this already). How this would affect crash recovery, data consistency, cleaning, and the architecture of master servers are also open subproblems.

The implementation of RAMCloud's cleaner could be better optimised, particularly the memory compactor. For example, the cleaner consists of a small, very hot code path that decides whether or not a log entry is alive and correspondingly updates some metadata and either drops the entry or writes it to a survivor segment. I suspect that this code could be further improved, both with careful code tuning and other optimisations. For instance, on large memory machines we will almost always take a TLB miss on each cache miss in the hash table or log, which could be mitigated by using 1 GB superpages to map that memory. Judicious use of batching and memory prefetching may also significantly mitigate these expensive random memory accesses. Finally, it should be possible to use hardware (such as an FPGA) to implement or accelerate this code. This would not only alleviate load from the expensive and power-hungry main processor, but also allow finer-grained parallelism. For example, multiple segments could be compacted in parallel, with hardware switching between different log entries as requisite cache lines are retrieved from memory, rather than blocking as the CPU would.

RAMCloud's balancer also leaves some room for improvement. Chapter 5 introduced some policies that work well most of the time and do not have severe pathologies. However, there are two key outstanding problems. First, I did not discover an optimal policy that works well across all access patterns and system configurations. Even if a single optimal policy cannot be found, it might be possible to dynamically choose among several different policies, each of which works best in different situations. Machine learning techniques might be a good fit here (other work has used machine learning to choose the best garbage collector for a given Java application, for example [8]). Second, the best balancers I found relied on carefully configured magic constants. Although they worked well across a wide range of experiments, having either a balancer that does not depend on voodoo constants, or having an analytical justification for why a particular constant works and will continue to work well would be preferable.

10.1.2 RAMCloud in General

One of my biggest concerns moving forward is how well RAMCloud's architecture will adapt to future code and feature changes. In particular, RAMCloud's current data model is very simplistic. Making it more sophisticated will undoubtedly add new types of entries to the log, which will need their own handling during cleaning. To work in concert with the log cleaner, these types will need to be efficiently moveable, otherwise they will slow cleaning down. How onerous a requirement this is is not yet clear. Objects and tombstones are easy and cheap to check for liveness and to update pointers because of the hash table. If RAMCloud had to traverse more complex data structures to perform these operations, however, it might greatly slow cleaning.

Finally, RAMCloud has been built for applications that do not exist yet, and as such we do not know what their workloads might look like. It is possible that RAMCloud will only ever be used for read-dominated access patterns, in which case cleaning efficiency may not be very relevant. In other words, it could be that the work in this dissertation has over-optimised a part of the system. Only time will tell. Regardless, I think it will be fascinating to see the sorts of workloads that RAMCloud is eventually used with and how the mechanisms described and analysed here hold up.

Bibliography

[1] Scaling memcached at facebook, December 2008. [10]

[2] More Details on Today's Outage — Facebook, September 2010. `http://www.facebook.com/note.php?note_id=431441338919`. [121]

[3] Google performance tools, March 2013. `http://goog-perftools.sourceforge.net/`. [xiv, 18]

[4] logcabin/logcabin - github, October 2013. `https://github.com/logcabin/logcabin`. [11]

[5] memcached: a distributed memory object caching system, March 2013. `http://www.memcached.org/`. [xiv, 2, 9, 12, 18, 96, 97, 106, 112, 118, 120, 121]

[6] Redis, March 2013. `http://www.redis.io/`. [96, 108, 118, 120]

[7] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, November 1993. [122]

[8] Eva Andreasson, Frank Hoffmann, and Olof Lindholm. To collect or not to collect? machine learning for memory management. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 27–39, Berkeley, CA, USA, 2002. USENIX Association. [130]

[9] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J.*, 42(1):60–76, January 2003. [41]

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM. [55, 97, 117]

[11] David F. Bacon, Perry Cheng, and V. T. Rajan. 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*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM. [21]

[12] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 15:1–15:16, Berkeley, CA, USA, 2008. USENIX Association. [28]

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association. [124]

[14] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340, New York, NY, USA, 2013. ACM. [124]

[15] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003. [6, 120]

[16] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association. [36]

[17] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2005. [35]

[18] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM. [xiv, 18]

[19] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: an exercise in distributed computing. *Commun. ACM*, 25(4):260–274, April 1982. [5]

[20] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 277–288, Berkeley, CA, USA, 1995. USENIX Association. [123]

[21] Jeff Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 87–98, Berkeley, CA, USA, 1994. USENIX Association. [112]

[22] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '07, pages 858–867, 2007. [7]

[23] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the hamlyn sender-managed interface architecture. *SIGOPS Oper. Syst. Rev.*, 30(SI):245–259, October 1996. [122]

[24] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM. [124]

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association. [8, 120, 121, 124]

[26] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM. [21]

[27] David R. Cheriton and Willy Zwaenepoel. The distributed v kernel and its performance for diskless workstations. In *Proceedings of the ninth ACM symposium on Operating systems principles*, SOSP '83, pages 129–140, New York, NY, USA, 1983. ACM. [xii, 6]

[28] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, John Ousterhout Sachin Katti, and Mendel Rosenblum. Copyset replication: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 Usenix Annual Technical Conference*, San Jose, June 2013. [68, 129]

[29] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM. [125]

[30] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, August 2008. [8, 121]

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. [96, 108]

[32] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013. [121]

[33] Michael Donald Dahlin. *Serverless network file systems*. PhD thesis, Berkeley, CA, USA, 1995. AAI9621108. [92]

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. [8, 121]

[35] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM. [119]

[36] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM. [96, 108, 118, 121]

[37] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proceedings of the BSDCan Conference*, April 2006. [xiv, 18, 122]

[38] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association. [113]

[39] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX ATC'13, pages 333–346, Berkeley, CA, USA, 2013. USENIX Association. [122]

[40] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM. [5]

[41] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005. [123]

[42] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4:509–516, December 1992. [119]

[43] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. [124]

[44] Shay Gueron. Speeding up crc32c computations with intel crc32 instruction. *Information Processing Letters*, 112(5):179–185, February 2012. [28]

[45] John H. Hartman and John K. Ousterhout. The zebra striped network file system. *ACM Trans. Comput. Syst.*, 13(3):274–310, August 1995. [124]

[46] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM. [21]

[47] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, pages 145–158, Berkeley, CA, USA, 2010. USENIX Association. [11]

[48] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. The collie: A wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, New York, NY, USA, 2012. ACM. [125]

[49] Robert Johnson and J Rothschild. Personal Communications, March 24 and August 20, 2009. [7]

[50] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. [124]

[51] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1:1496–1499, August 2008. [120]

[52] Haim Kermany and Erez Petrank. The compressor: Concurrent, incremental, and parallel compaction. *SIGPLAN Not.*, 41(6):354–363, June 2006. [125]

[53] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *SIGARCH Comput. Archit. News*, 25(2):85–97, May 1997. [122]

[54] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev.*, 31(5):238–251, October 1997. [92, 123]

[55] Paul E. Mckenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998. [41]

[56] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.4BSD operating system*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. [35]

[57] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, ATEC '96, pages 279–294, Berkeley, CA, USA, 1996. USENIX Association. [106]

[58] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 319–330, New York, NY, USA, 2011. ACM. [7]

[59] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association. [6, 10, 20, 114, 120]

[60] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford, CA, USA, 2014. [11]

[61] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM. [14, 21, 27]

[62] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M.

Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Commun. ACM*, 54:121–130, July 2011. [10]

[63] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 159–172, New York, NY, USA, 2007. ACM. [125]

[64] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM. [125]

[65] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *J. ACM*, 21(3):491–499, July 1974. [20]

[66] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the 1997 USENIX Annual Technical Conference*, ATEC '97, pages 257–274, Berkeley, CA, USA, 1997. USENIX Association. [123]

[67] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, ATEC '00, pages 41–54, Berkeley, CA, USA, 2000. USENIX Association. [55, 90]

[68] Mendel Rosenblum. *The design and implementation of a log-structured file system*. PhD thesis, Berkeley, CA, USA, 1992. UMI Order No. GAX93-30713. [xviii, xx, 73, 75, 76, 78, 86, 125]

[69] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10:26–52, February 1992. [4, 14, 33, 37, 73, 74, 123]

[70] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 96–108, New York, NY, USA, 1981. ACM. [55, 90]

[71] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1):193–204, June 2009. [28]

[72] Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. An implementation of a log-structured file system for unix. In *Proceedings of the 1993 Winter USENIX Technical Conference*, USENIX'93, pages 307–326, Berkeley, CA, USA, 1993. USENIX Association. [15, 123]

[73] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 249–264, Berkeley, CA, USA, 1995. USENIX Association. [15, 123]

[74] T. Shanley, J. Winkles, and Inc MindShare. *InfiniBand Network Architecture*. PC system architecture series. Addison-Wesley, 2003. [10, 122]

[75] Ryan Scott Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford, CA, USA, 2013. [2, 10, 14, 27]

[76] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM. [21, 125]

[77] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM. [124]

[78] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. *SIGOPS Oper. Syst. Rev.*, 29(5):40–53, December 1995. [122]

[79] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20(2):256–266, April 1992. [122, 123]

[80] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag. [124]

[81] Benjamin Zorn. The measured cost of conservative garbage collection. *Softw. Pract. Exper.*, 23(7):733–756, July 1993. [21]