Matthew Huxtable

Unbuckle

Faster in the kernel

A high-performance key-value store

Computer Science Tripos, Part II

St John's College

15th May 2014

The name ***Unbuckle*** refers to the perilous journey of programming in an operating system kernel. All traditional safety mechanisms are missing and the hardware of the computer is directly accessible. Crashing the machine and corrupting its data is par for the course, but the payoffs are worthwhile.

My project builds a key-value store in this as-yet unexplored context. Despite the inherent difficulties, I show that significant performance gains are possible. The ultimate goal is to answer some outstanding questions in systems research through building a platform-independent, general-purpose, in-kernel key-value store to compete with the incumbent systems in this field.

Proforma

Name:	**Matthew Huxtable**
College:	**St John's College**
Project Title:	**Unbuckle: An in-kernel, high-performance key-value store**
Examination:	**Computer Science Tripos, Part II (June 2014)**
Word Count:	**11934**
Project Originators:	**Malte Schwarzkopf & Dr Steven Hand**
Supervisors:	**Malte Schwarzkopf & Dr Steven Hand**

Original aims of the project

The implementation and performance analysis of a platform-independent key-value store within an operating system kernel. This seeks to investigate the overhead imposed by the privilege transition on applications running in modern data centres. Specifically, I am interested in how much faster a software key-value store can be made by removing layers of abstraction and multi-process safeguards. The deliverable is a kernel-based, memcached-compliant key-value store with an evaluation of its performance in three contexts: the kernel, user-space, and in comparison with existing commercial and research systems.

Work completed

The project has been highly successful. All success criteria were met and many optional extensions implemented. I built the *Unbuckle* key-value store, which can be compiled as a Linux kernel module or a user-space application. On a 10 Gbps network, the in-kernel version achieves a 25% reduction in latency at the same time as a 40% throughput increase compared to the user-space version. *Unbuckle* scales to 22.5 Gbps throughput using eight cores of a modern server. It outperforms memcached, an optimised key-value store used by many web companies, in both throughput (3× improvement) and latency (50% reduction).

Special difficulties

Familiarisation and working within a large, pre-existing codebase (the Linux kernel) with challenging failure modes.

Declaration

I, Matthew Huxtable of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: 15th May 2014

Contents

List of Figures

Acknowledgements

Throughout this work, I have greatly appreciated the contributions and suggestions put forward by many people. However, I owe particular thanks to:

- **Malte Schwarzkopf**, for supervising my project, all his regular advice and the encouragement to push the boundaries of what I thought was possible.
- **Matthew Grosvenor**, whose in-depth technical knowledge of the Linux kernel was especially useful and undoubtedly saved me many hours.
- My **family and friends**, for their support, understanding of my workload and tolerating many more hours of technical discussion than were perhaps preferred.

Chapter 1

Introduction

This dissertation describes the implementation and evaluation of a key-value datastore in software, as a platform-independent Linux kernel module and an equivalent userspace application. The implementation is shown to achieve a 300% throughput improvement while simultaneously reducing latency by 50% as a result of the store running within the most privileged layer of the operating system.

1.1 Motivation

In recent years, web-based applications have become ubiquitous and now demand high-performance methods for storing and serving an unprecedented volume of data. As these systems become more pervasive, it is becoming apparent that our conventional methods for managing these data are unable to scale to support the demands of today's interactive, real-time workloads. Key-value stores represent a low-cost, lightweight alternative solution to traditional databases. They have already seen widespread deployment in the infrastructure behind the web's most popular websites.

The data centres at the core of the web are indispensable assets to modern internet companies. These facilities consist of thousands of commodity, off-the-shelf server machines collaborating to provide the abstraction of a single "warehouse-scale" computer. Yet, despite this apparent departure from the traditional approach, rapid growth means legacy code remains widespread throughout the software stack. The software interfaces available to today's applications have been found to be poorly aligned to their requirements [47].

One key area of concern is the traditional placement of data centre applications under the jurisdiction of an operating system, where a penalty is incurred for performing privileged operations involving hardware and other system state. This *system call interface* exported by a system kernel is important for multiplexing access to system resources and enforcing system integrity through process isolation. However, data centres operate sealed machines without interactive users – or indeed, separate containers or virtual machines for each application task – so many of these security benefits are of little to no consequence. Interrupting the processor to perform frequent privilege switches can have negative consequences in terms of performance, bringing into question the continued applicability of this abstraction layer in a data centre context.

In this dissertation, I explore the hypothesis that it is possible to achieve further performance gains for a key-value store by bypassing the system call interface and deploying the application in the kernel, where access to the hardware and memory is directly available. Unlike other work in this area (see §1.3), the proposed system is designed specifically with platform-independence in mind, permitting the key-value store to be used in a variety of contexts with only minimal modification to existing systems.

1.2 Challenges

A project of this nature will not be trivial. Modern system kernels were never designed to allow the implementation of arbitrary software platforms in the core of the code. Most programmers use the kernel as a black box; few possess sufficient knowledge of its internals and its complex interactions between hardware and software to successfully implement kernel code. Kernels hide details of the underlying hardware precisely because this interaction is difficult, especially when code is required to be platform agnostic.

The ideal solution would implement a minimal, highly customised kernel from scratch, with its sole task being to run a key-value store for an IP-based network. However, kernels are large software projects which take many years to perfect. Successfully implementing the core components necessary to boot a machine and interact with basic hardware would be challenging in the time available, especially for such interaction to be cross-platform. Familiarisation with an existing kernel upon which the key-value store can be based will be imperative for a successful outcome.

The implementation work in the kernel is high risk. However, a successful outcome has the potential to deliver large rewards of relevance to the challenges facing today's web companies.

1.3 Related work

High performance key-value stores are an active area of research. Much of this work attempts to ensure performance can scale to the next generation of data-intensive applications. I highlight the most important publications in the field to date, especially those which influenced this project.

The case for distributed systems techniques in the web is not new. It was recognised in the early 2000s that research in this area would be needed for long-term scalability [22,51], but workloads have only recently outgrown the traditional approaches.

Ousterhout *et al.* made the case for key-value techniques as a method of resolving the disparity between access times to DRAM and conventional magnetic storage in the modern web [42]. Significant practical work has already been undertaken by large internet companies to transform these research findings into technology suitable for practical use. Such systems are embedded in most popular websites, including, for example: Amazon Dynamo [14], Google BigTable [10], LinkedIn's Project Voldemort [52], memcached at Facebook [32,41] and Twitter [2], and Redis at Instagram [31].

1.3.1 Hardware-based work

A key source of inspiration for this project was a suggestion in recent publications that a scalability limit to software-based key-value solutions has been reached. Several studies have suggested that achieving 10 Gbps performance requires the use of custom hardware-based solutions as a replacement for present software platforms. These include a deployment using a specialist hardware design on FPGAs [25] and remote direct memory access to server memory over a high-performance Infiniband network [15,40].

However, these proposals are unlikely to succeed at scale in practical data centres. Such installations favour x86-based hardware platforms and traditional Ethernet networks; upgrading them with custom technology and specialist interconnect networks restricted to key-value store use would be neither cheap nor practical.

The claim that software-based solutions have reached their limit is a strong assertion which merits further investigation, especially to determine whether the poor scalability of general-purpose x86 processors or outdated operating system interfaces contribute to the problem.

1.3.2 Key-value stores and system kernels

There is already some evidence that the latter reason may be to blame. While there is no existing research in the performance of a key-value store running as part of an OS kernel, research in the early 2000s provided evidence that in-kernel web servers were able to achieve performance gains over their user-space counterparts [28]. In more recent work by Madhavapeddy *et al.*, the concept of *Unikernels* is introduced – single machines with specialised kernels and minimal overhead which are dedicated to performing a single task well [37]. These works add support to the thesis that bypassing superfluous protection mechanisms and unnecessary abstraction layers can lead to performance gains.

Exokernels were proposed in the mid 1990s as a means of securely permitting applications to control and manipulate hardware directly, shaking up the current fixed and archaic system call interface in the process [18]. This presents an exciting opportunity to deliver the best of both worlds: a new kernel design applicable to data centre workloads, without sacrificing the fault tolerance and sanity checking properties of today's kernels. Sadly, exokernels have received only minimal commercial interest, perhaps due to inertia behind present operating system design and a lack of enthusiasm in re-engineering entire software stacks from the ground up.

Other recent developments in kernel and network stack design are also of direct relevance to this project. Kernel-bypass networking has been proposed as a method of performing packet processing directly in user-space code, with promising results [16,17,27].

A very recent proposal, MICA [35], reports significant advantages to key-value systems by taking advantage of such a user-space network stack. However, this system misses many desirable features. Its compatibility and cost-effectiveness is restricted by dependence on specialist network stacks and compatible hardware. Its execution in user-space also imposes other restrictions: there is limited scope to avoid virtual memory overheads or the impact of process scheduling, for instance.

By contrast, *Unbuckle*, the system I describe, achieves performance gains without demanding specialist networking hardware, by utilising the interfaces available to kernel code to bias the system in favour of its key-value store workload.

1.3.3 Novel data structures

Finally, recent work on the impact of data structure design gives further credence to operating systems being a contributing factor to the scalability problem.

Masstree is a proposed data structure which combines tries and B^+-trees to suit the idiosyncrasies of processor caching behaviour [39]. This is achieved in spite of the implementation being in user-space and using the kernel network stack, where it continues to incur the associated performance overheads.

Algorithmic improvements have been obtained by re-engineering hash tables to support *cuckoo-hashing* [19,33]. By avoiding chaining in hash buckets, lookups in the data structure are guaranteed to proceed in constant time. It can also be shown, under certain assumptions, that methods exist for insertions to be performed in constant time [43].

Chapter 2

Preparation

In order for a project of this scale to be successful, it is crucial that sufficient preparatory work is undertaken prior to commencing with the implementation. As part of this, I had to analyse the project goals, define the expected outcomes and derive a development plan to adhere to. In addition, it was essential that I acquainted myself with the theory behind the systems I would be basing the project upon.

This chapter commences with a brief review of key-value stores and operating system principles. Building upon the background knowledge gained from this review, the outcomes of the requirements analysis phase for *Unbuckle* are presented. Finally, the decisions made with regard to managing the project implementation workflow are discussed with reference to the requirements specification.

2.1 Introduction to key-value stores

Key-value store technology has grown in importance in recent years as a building block of data-intensive applications in the modern web. I will first give a brief background to the historical methods of electronic data management, followed by an analysis of memcached, a widely used key-value store implementation whose protocol is borrowed for this project.

2.1.1 Historical background

The advent of the web and availability of affordable consumer technology has dramatically changed the nature of data production and consumption. For several decades, data storage has been in terms of formally specified mathematical relations in the *relational database model* [11], which enforces a set of guarantees[1] to ensure data accurately model real world interactions.

[1]The ACID properties – Atomicity, Consistency, Isolation and Durability – provide data integrity during operations on a relational database [48, pp. 625–ff.].

Unfortunately, this approach is a limiting factor for today's most popular applications. In a distributed system, enforcing the ACID properties requires all participating machines to be synchronised and reach consensus before any modifications to data can be made. This is not compatible with the high-intensity data demands of these applications. The success of large web search engines and social media services, for instance, is wholly dependent upon finding alternative data management methods which avoid the co-ordination bottleneck and make individual nodes in the distributed system more autonomous.

The semantics of many of today's web applications are also very different to the original systems targeted by the relational model. Many applications do not require updates to be visible everywhere immediately and many can tolerate limited data loss without negative consequences. Weaker forms of data integrity, such as an *eventual consistency* model [54], are more suitable for these applications, but the relational model is ineffective at relaxing its guarantees.

2.1.2 memcached

The first key-value store, memcached,[2] was introduced in the early 2000s in an attempt to solve the database scalability problem by adding a caching layer.

memcached is built for short-term caching of frequently referenced data objects in the main memory of one or more servers [20]. This offers greater scalability and lower latency data access than is possible by reconstructing this data from disk for each request. The key-value store is an extremely simple, unstructured database system in which mappings between arbitrary `<key, value>` pairs are stored and retrieved.

The client software interacting with memcached permits the key space to be distributed across multiple machines. As the logic for finding hosts in a memcached cluster is pushed into the client application, memcached servers are unaware of their contemporaries on the network, allowing the capacity of the cluster to increase without overheads. Viewed in this way, the store is effectively a large, network-connected hash table of hash tables – the client software uses *consistent hashing* to map a key to a particular server, which implements a local hash table to service requests. This interaction is summarised in figure 2.1.

The data stored have no associated formal schema in the key-value store representation, allowing a wide range of data to be cached. Data stored will typically be programming language primitives, such as alphanumerical values or serialised objects in object-oriented programming environments. This complements modern system architecture, where results are dynamic and typically retrieved and rendered on-the-fly from a variety of data sources. Similarly, modern applications are increasingly aligned with object-oriented principles rather than normalised data storage models, as exemplified by the popular *model-view-controller* design pattern [9].

[2] `http://www.memcached.org`

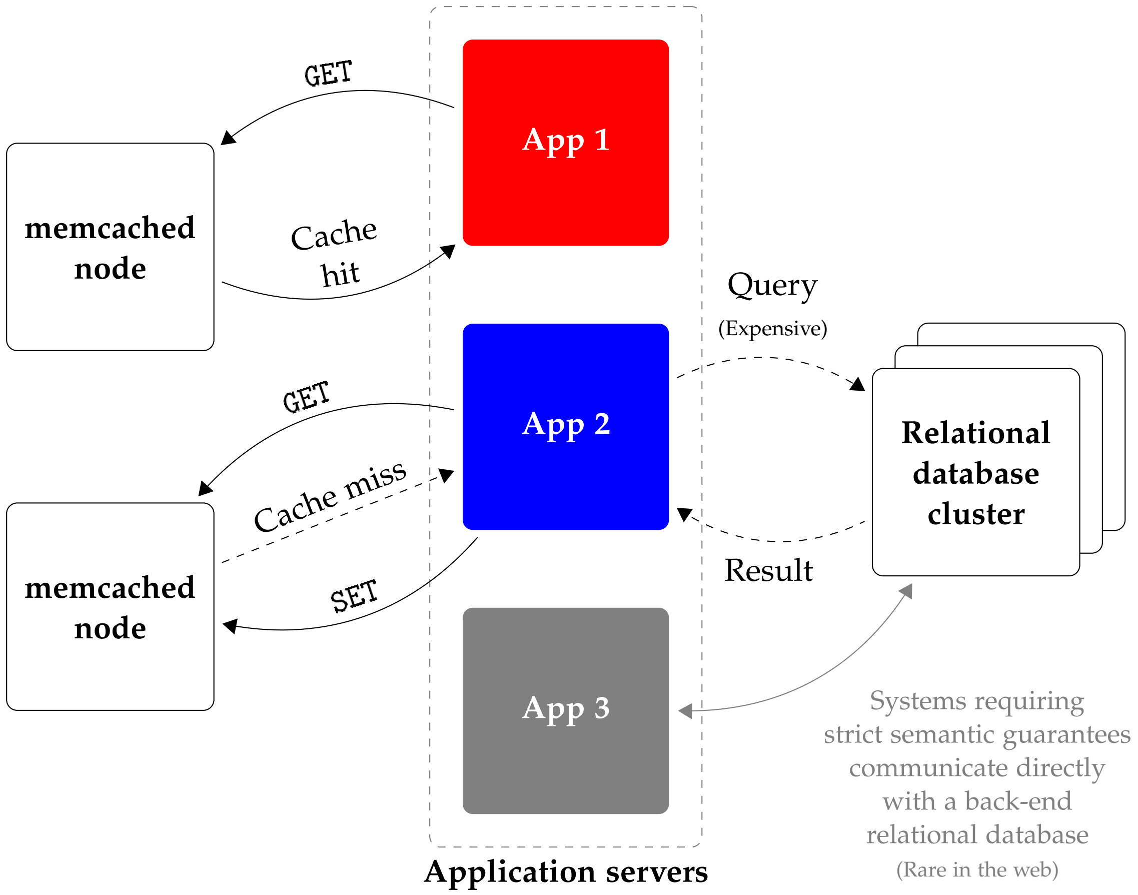


Figure 2.1: The interaction of memcached with application servers. In the event of a cache miss, an application is responsible for satisfying its information need via some other source, such as a back-end database server, and pushing the result into the cache.

memcached protocol

memcached servers export a simple protocol for applications to store and manipulate data in the cache.[3] Two operations in the protocol, `GET` and `SET`, are crucial for any interaction with the system. The former is issued to obtain the value associated with a particular key, while the latter inserts or updates data in the cache. For drop-in compatibility with existing implementations, I decided to adopt the memcached protocol for the purposes of this project.

Very few semantic guarantees are provided by the memcached protocol. There is no guarantee that an application will even be able to read its own writes, for the system is simply a cache, not a persistent data store. Data may be aged out and replaced due to cache churn. In most implementations, a least-recently-used cache eviction policy is implemented, which optimises for temporal locality of reference [1].

The protocol provides several mechanisms for communicating with the store, including the reliable, in-order TCP protocol and the message-oriented, connectionless UDP protocol. For the purposes of the project, I elected to pursue the UDP

[3]Full protocol details: `https://memcachedb.googlecode.com/sv/trunk/doc/protocol.txt`.

approach, which does not suffer TCP overheads linear in the number of connections to the store. In particular, the TCP three-way handshake imposes several round-trip times on the latency of a request to access the store.

There is evidence the UDP protocol is already widely deployed [41], as the relatively low frequency of lost or corrupt packets in modern Ethernet networks outweighs the benefit of delivery guarantees for a caching system like memcached. Occasional lookups to a database due to a memcached request timing out are considered an acceptable compromise. Alternative implementations might provide reliability using persistent TCP connections without succumbing to the performance overhead [26].

2.2 Introduction to the operating system kernel

The term *operating system* normally refers to the *kernel*, the body of code running in privileged mode and first given control when an operating system is booted. Kernels are typically responsible for managing system hardware, enforcing security primitives for process isolation and multiplexing system resources between active user sessions. The kernel is the most privileged component of an operating system.

On today's multitasking machines, hardware primitives are built into the microarchitecture of the processor to enforce the separation of privilege between regular, user-invoked processes, which execute in the low-privilege *user-space* environment, and the kernel. User-space applications are prevented from accessing physical memory or manipulating the system hardware, in an effort to isolate running applications (and even separate users) from each other. This also ensures system stability is maintained if an application exhibits a destabilising bug, as the kernel acts as a sanity check for privileged operations and can intervene when necessary.

2.2.1 System calls

In practice, applications frequently need to perform privileged tasks. The system call interface is the mechanism by which user-space applications invoke the kernel to request privileged work be carried out on their behalf by one of the kernel subsystems, as shown in figure 2.2. Various standards exist for this interface, including the well-known POSIX interface supported by almost all modern Unix derivatives.

Applications typically execute a system call by preparing registers with arguments and then raising a software interrupt. This halts the execution of the processor and causes it to jump to interrupt handler code configured as the entry point to the kernel's system call interface. In the process, an internal flag is modified to indicate the processor is now executing in privileged kernel mode. This removes processor-enforced hardware restrictions, in particular those imposed on the memory

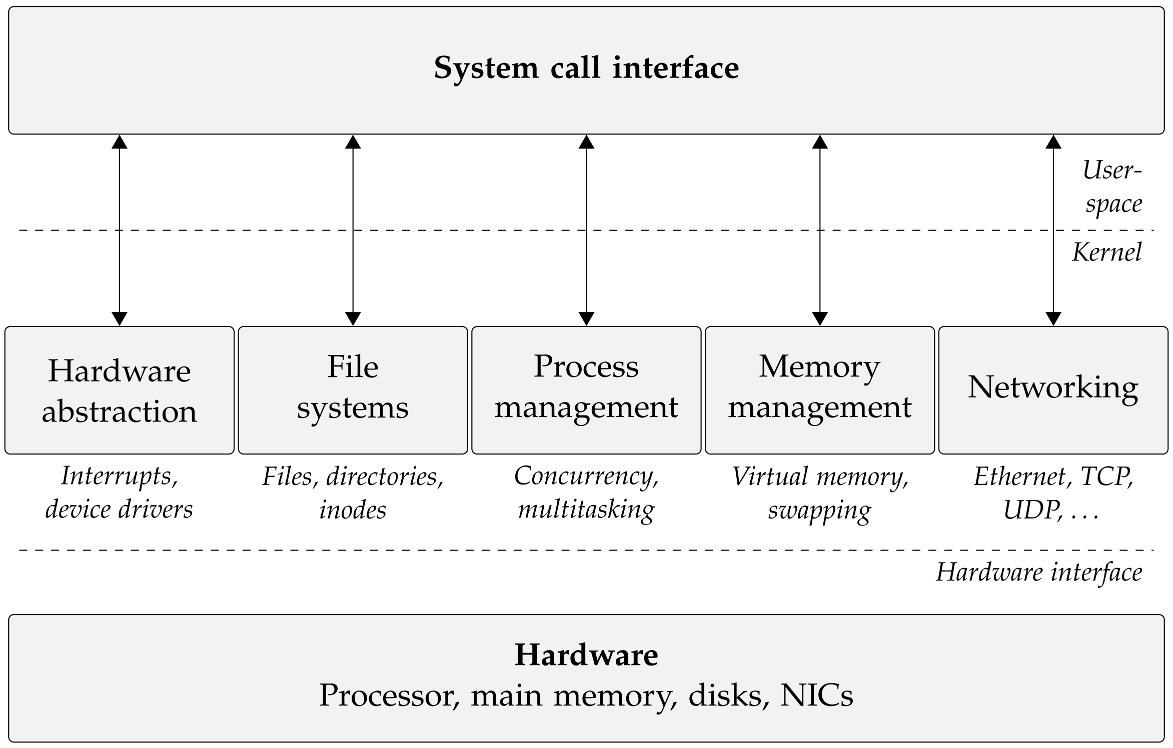


Figure 2.2: An abstract system call interface.

management unit. This permits the kernel to modify system descriptor tables, enable and disable interrupts and gain direct access to physical memory.

It is widely known that interruption of the processor in order to execute a system call carries considerable overhead. Recent work shows that an average latency of $5\mu s$ is incurred when servicing a system call on a modern Intel Core i7-based machine [46]. While this can be amortised within an interactive session, it seems wasteful on machines dedicated to a high-performance key-value store role in a data centre environment. These systems predominantly perform input/output on the network interface card (NIC) and with main memory, and hence perform a considerable number of system calls. However, each system call has relatively little work to do, thus further exacerbating the overheads.

Further discussion on the realisation of this hardware-based protection mechanism on x86-based microprocessors is provided in appendix A.1.

2.2.2 Hypervisor technology

Modern data centres are increasingly using virtualisation solutions, such as Xen [5], to run multiple instances of an operating system in a parallel but isolated manner on a single machine. Virtualisation technology further diminishes the applicability of the system call interface in these environments: virtualised worker machines are typically dedicated to a single task and have no interactive users, so concerns over process isolation and system fairness are irrelevant, as they are handled at the hypervisor level. Moreover, the execution of code in the kernel context poses little additional risk, as any faults or malicious activities will be contained by the security mechanisms provided by the underlying hypervisor. As a result, specialised OS kernels have seen considerable interest lately [38]. *Unbuckle* is such a specialised kernel.

2.3 Requirements analysis

With the requisite background knowledge understood, the requirements for the project were carefully reviewed. This was an important early step for understanding the most applicable system architecture and planning the work to minimise risk where possible. The main success criteria for the project are summarised in table 2.1. These are largely similar to the goals outlined in the original project proposal (see appendix A.4).

Goal Requirement	**Priority**	**Risk**	**Difficulty**
Implement the memcached protocol	High	Low	Medium
Build the UDP server	High	High	High
Construct suitable data structures for storage	High	Medium	Medium
Platform agnostic hardware interaction	High	High	High
Port the key-value store to user-space for evaluation	Medium	Low	Medium
Optimisation: multi-threaded key-value store	Medium	Medium	High
Optimisation: back-end data structure improvements	Medium	Medium	Medium

Table 2.1: High-level goals and deliverables for the Unbuckle *project.*

Further to the requirements analysis, I conducted a detailed survey of the modular dependencies within the project (illustrated in figure 2.3).

It quickly became clear that there were several modules upon which there was heavy dependence, most notably the implementation of the back-end data structures and the network server. Early commitment to the modular design and their associated interfaces made it possible for these modules to be developed over several iterations of the project. A prototype implementation could be supplied in advance in order to provide basic functionality without necessarily offering any performance advantages. A later iteration of the module, after a working end-to-end system had been demonstrated, would refine such code to a version more suitable for a rigorous performance evaluation and long-term use.

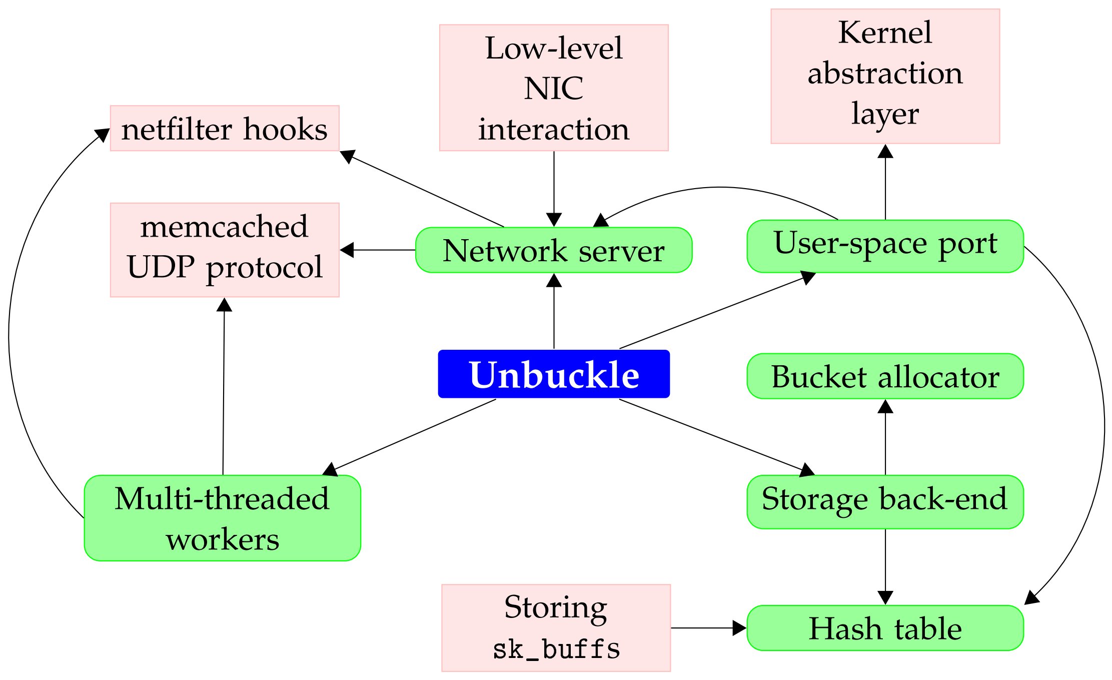


Figure 2.3: Dependencies in the Unbuckle *project.* $x \rightarrow y$ *indicates a dependency of* x *on* y*.*

2.4 Choice of tools

2.4.1 Libraries and kernels

The nature of the project meant there would necessarily be dependence on third-party libraries to implement the functionality which was infeasible or simply not instructive to re-implement in the time available.

Choice of kernel

The identification of a suitable operating system kernel whose code could be inspected and freely modified was a crucial early decision underpinning the entire project. Two major open-source kernel projects were evaluated for suitability: Linux[4] and FreeBSD.[5]

Ultimately, after some early research into programming styles, support structures and market factors, I selected the Linux kernel as the basis for the project.

A key decision behind the use of Linux was its widespread deployment, which ensured the final implementation would have immediate real-world utility. Due to economies of scale, today's modern data centres typically comprise many commodity, off-the-shelf servers running a Linux-based operating system and interacting via some distributed middleware layer. In this context, BSD-derived works are more

[4] http://www.kernel.org
[5] http://www.freebsd.org

commonly constrained to embedded systems, such as networking hardware, rather than the machines responsible for storing and processing data.

Linux development progresses rapidly, so to keep abreast of releases and bugs which might hinder progress, I subscribed to the Linux Kernel Mailing List,[6] the main forum in which discussion of the kernel internals, the reporting of bugs and the setting of development priorities takes place. This subscription was useful on several occasions, despite the high number of daily messages, which often approached 1,000.

For conciseness, I will hereinafter adopt the convention that references to "Linux" and "the kernel" refer specifically to the Linux kernel, except as may be clarified separately in the text. This is a standard typographical convention in most kernel literature.

Test harnesses

As the evaluation of the project was necessarily going to be quantitative, it was important for a test harness to be committed to early on. This test harness should allow me to reliably test the project and produce suitable results to determine whether the kernel implementation has an impact on performance.

Developing my own test harness was out of the question. Generating sufficient requests to keep a large network pipe full while simultaneously dealing with packet corruption, timeout and sequence number repetition are tasks which are time consuming to implement correctly, without contributing anything original to the project.

Hence, several memcached testing frameworks were evaluated for suitability. A harness called *memaslap*[7] was eventually adopted, primarily due to it being unique in supporting the memcached UDP protocol. Full details of the harnesses evaluated are given in appendix A.2.

Other libraries

Programming in the kernel makes it difficult to directly utilise any third-party libraries intended for user-space use, for a number of compatibility reasons I will outline in §2.5. Very few libraries were suitable, with the exception of:

- **SpookyHash** for hashing keys of arbitrary length to an internal 64-bit integer representation. It was originally implemented by Bob Jenkins in C++,[8] and ported to C by Matthew Grosvenor.[9] I ported the latter C variant to be compatible with the kernel.

[6] `https://www.lkml.org`

[7] Component of libmemcached, available at `http://www.libmemcached.org`

[8] `http://burtleburtle.net/bob/hash/spooky.html`

[9] The C port is a component of *libchaste*, available at `https://github.com/mgrosvenor/libchaste`

- **UTHash** is a user-space hash table implementation due to Troy Hanson.[10] It features in the user-space implementation and was ported to the kernel.

2.4.2 Programming languages

C

The main programming language used for the implementation of the project was C.

For historic and performance reasons, this is the *de facto* programming language for all code in the main kernel source tree. Thus, C was the only reasonable choice which would guarantee compatibility with the pre-existing kernel data structures and interfaces.

Nevertheless, despite this choice being predetermined, the reasons underpinning C's continued relevance in the development of the kernel also make it a highly suitable candidate for this project:

- **Performance** – proximity to the hardware and the availability of carefully researched optimising compilers offer a distinct advantage in producing efficient software programs;
- **Cross-platform** – a compiler for C can normally be found for most hardware platforms, allowing code to be ported to alternative systems and satisfying the platform independence requirement for the project;
- **Maintainability** – despite its considerable efficiency gains, C continues to offer some of the programming abstractions common to higher-level programming languages.

Python & shell

For support tasks not directly involving the kernel, it was most efficient to write Python programs or shell scripts. This was particularly the case for managing the test framework and for analysing and visualising performance data.

2.4.3 Development and test environment

Development work took place largely on my own personal machines, running the Ubuntu 13.10 *Saucy Salamander* operating system. Additional resources on the University MCS[11] were used for backups. I am also grateful to the Systems Research

[10] `http://troydhanson.github.io/uthash/`

Group at the Computer Laboratory, who provided access to their high-performance network environment for testing purposes.

It was impractical to use the MCS workstations for any development or testing. Modifications to a running kernel require super-user privileges over the machine concerned, owing to the risk of compromise to other system users. For obvious security reasons, these privileges are not routinely available on the shared MCS machines.

Revision control

An essential prerequisite for a project of this scale is the use of a revision control system to organise the development process and capture changes to the source code as it evolves. I chose to use the popular open-source *git*[12] package for this purpose.

git is a distributed revision control environment which was originally built to manage Linux kernel development. It provides the ability to develop new features on development branches, which are isolated forks of the repository. For new features with no circular dependencies, branches were used for development to avoid introducing instability to existing code. Thus I could delay integration of a logical revision history for a completed feature until I was satisfied it adhered to specification. This complemented the spiral model of development I adopted for most of the work, which I will discuss in §2.7.

The same arrangement was also used to store each iteration of the LaTeX source code of this dissertation.

Backup strategy

A backup system was carefully arranged so as to mitigate the risk of unforeseen hardware or software failure or indeed, user error. The system so designed is illustrated in figure 2.4.

The use of git for revision control provided additional benefits for data security. As a distributed revision control system, all development machines automatically maintain a full working copy of the repository and its history, independent of any central server.

In order to keep changes on either of the development machines in-sync, git auto-commit hooks were deployed to automatically synchronise commits to my personal Ubuntu-based file server. Geographic isolation was obtained by virtue of the file server being situated outside Cambridge.

[11] Managed Cluster Service, formerly Public Workstation Facility (PWF)

[12] `http://www.git-scm.com`

[13] The GitHub logo is a registered trademark of GitHub, Inc. It is used here with kind permission.

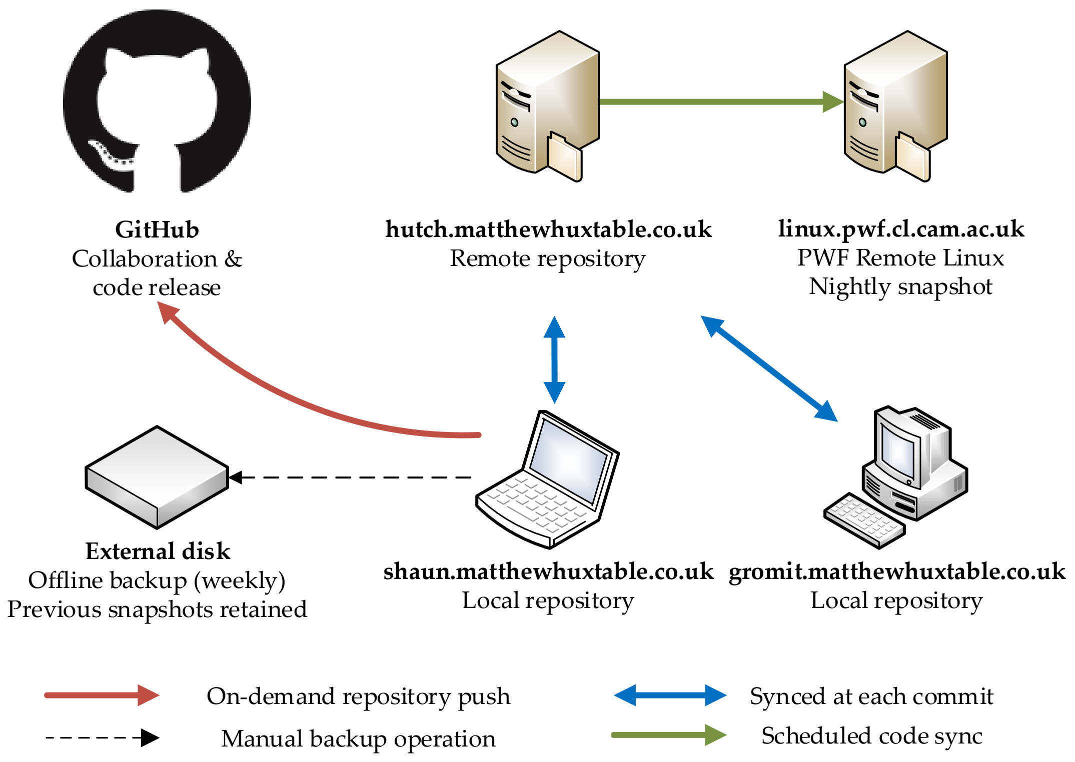


Figure 2.4: The backup strategy adopted for the Unbuckle *project.*[13]

Although the simultaneous failure of the file server and both development machines was unlikely, it is possible user-error could have caused data loss across the synchronised repository. To manage this risk, the file server pushed nightly snapshots to the MCS filestore and a weekly snapshot was made on Sunday evenings to an external hard disk, which was normally kept powered down.

A free online git hosting service, GitHub, was also employed. Although this system was primarily used for collaboration with my supervisors, the regular upload of code at key project milestones also served as a backup.

Due to some disk driver issues when prototyping some kernel code, there were several occasions when recovery from a backup was necessary to fix corrupted files in the repository. I am pleased to report that the backup plan worked, because in spite of these failures, no actual data loss occurred.

Testing strategy

The lack of safety and security protection for code executing in kernel context made it imprudent to use the development machines for project testing. Software bugs, however trivial, can have unpredictable consequences in the kernel, in certain circumstances requiring a forced reboot of a machine or even corrupting data on disk and damaging the operating system.

An abstraction layer was constructed to bridge the differences between user-space and kernel software interfaces, permitting the same source-code to be compiled as either a user-space or kernel application. Unstable code was first tested using the user-space variant to identify bugs before they were permitted to enter the kernel and cause system instability.

In addition, I used a sandbox environment to test unstable code against a running kernel. This consisted of a lightweight Debian-based virtual machine running within the VirtualBox[14] virtualisation environment. Virtual machine snapshots were used to facilitate efficient rollback of the virtual machine's hard disk and other state in the event a bug caused irrecoverable damage.

2.5 Initial experience

At the outset of the project, I had a reasonable amount of experience with C and was already familiar with many of the formalisms and concepts underpinning the language. This was due, in part, to the *Programming in C and C++* course in Part IB of the Tripos.

However, I had no prior kernel programming experience. This transition proved challenging due to some fundamental differences of this style of programming, *viz.*:

- **No C standard library** – the kernel's compilation suite does not provide any of the libraries commonly used by user-space code for memory management, input-output traffic on a network, threading and similar tasks. Instead, it is expected that low-level kernel interfaces will be used to call upon hardware and other kernel subsystems directly. Unlike the system call interface (per §2.2) and the C standard library, the kernel interfaces are not standardised and are subject to change between kernel releases. They do not necessarily follow a uniform convention across the many kernel subsystems and there are often incompatibilities in the formats of the data they produce and consume.
- **Documentation** – there is very little up-to-date documentation for the kernel, as most kernel subsystem developers assume the source code also serves as the documentation. This can be particularly troublesome – tracing code in an attempt to understand the kernel internals is hard work, especially when there is heavy use of indirection (such as the network stack, which I will discuss later in §3.2). I made some reference to literature [36], but unfortunately this has partially been superseded by recent kernel architectural modifications.
- **Development difficulties** – user-space programming benefits from sanity checks being made by the kernel to gracefully resolve any software faults before they cause the system to become unstable. However, the kernel cannot sanity check its own operations nor can it trap itself in the event of a serious

[14] https://www.virtualbox.org/

error. Unpredictable crash behaviour and volatility of log files often necessitates inventive solutions to obtain debug information to trace a bug. In the absence of more advanced techniques, this often requires carefully constructing log messages at strategic points in the code to print debug output to a serial console.

Several Part II courses were also useful at various stages of implementation. The kernel makes extensive use of the optimiser supplied as part of the GNU Compiler Collection,[15] for which *Optimising Compilers* was very useful to understand the theory of the transformations made. Interaction with the network built upon knowledge acquired in the *Principles of Communications* course.

2.6 Implementation approach

Before development could commence, it was crucial that ideas were consolidated and a suitable plan developed. There were many possible approaches I could have taken with implementation, so an initial consolidation was necessary to segment the work into discrete work packages with clear internal milestones to measure success.

I chose to segment the project into three phases:

1. **Core Implementation** – a naïve implementation of the key-value store was produced as a kernel module adhering to the memcached protocol, but with no effort made to optimise its performance. The deliverable was a working kernel module capable of communicating over a network. This represented the core infrastructure upon which the following stages were based.

2. **Performance evaluation** – at this transition point in the project, integration with the test harness was performed and some unforeseen bugs in memaslap were resolved. Performance figures were collected to provide a baseline for the forthcoming optimisations.

3. **Enhancement & Optimisation** – finally, I implemented the many enhancements and optimisations available by virtue of operating in the kernel, returning to phase 2 on each occasion to evaluate their impact, if any, on system performance.

The core modules identified by the modular dependency analysis as being on the critical path were the network server and back-end storage engine. As these were also identified as being high risk items in the requirements specification, I elected to restrict my early implementation of these modules to the simplest possible design. This enabled early completion of phase 1 and promoted the demonstration of a working end-to-end system early in the project timeline.

The integration of a network server was considered a high-risk task and deserved some careful attention. The ultimate goal was to interact with a machine's NIC at the lowest possible level in the network stack while retaining platform independence.

[15] `http://gcc.gnu.org`

I relegated this task to an optimisation, rather than a requirement for the core implementation. For phase 1, the kernel's socket interface was used instead. This interface does not claim to improve performance over user-space, but it was very similar to the one I was familiar with from user-space socket programming. Later work once again optimised this into a custom network server in parallel with other optimisations, rather than attempting to interact with and tame recalcitrant system hardware as part of the project's critical path.

2.7 Software engineering techniques

The methodology I adopted to manage the development and testing had the potential to make or break the project, so it was important that a methodology for development appropriate for the foregoing requirements, modular structure and implementation approach was incorporated as early as possible.

2.7.1 Development model

I decided to adopt the *spiral model* [7], which fits particularly well with the evolutionary nature of the project described previously. This model permits a multifaceted approach in which project phases can be assigned independent development methodologies. The core implementation (phase 1) which formed the critical path in the module dependency diagram proceeded via the sequential *waterfall model* [45] with strictly defined timeframes and progress criteria. The riskier experimental work was prototyped and integrated via a more rapid *prototype-measure-refine* feedback loop between phases 2 and 3.

The operation of the spiral model in the manner described is illustrated in figure 2.5.

2.7.2 Testing

In addition to the development framework, testing was performed in accordance with the methods outlined previously in §2.4.3. The difficulty of testing and debugging code in the kernel meant it was only feasible to test the external interfaces exported by the kernel module, for which a custom unit and regression test suite was implemented to verify memcached protocol compliance.

2.8 Summary

In this chapter, I have summarised the work undertaken prior to implementing the project. Background reading on key-value stores, in particular memcached, and

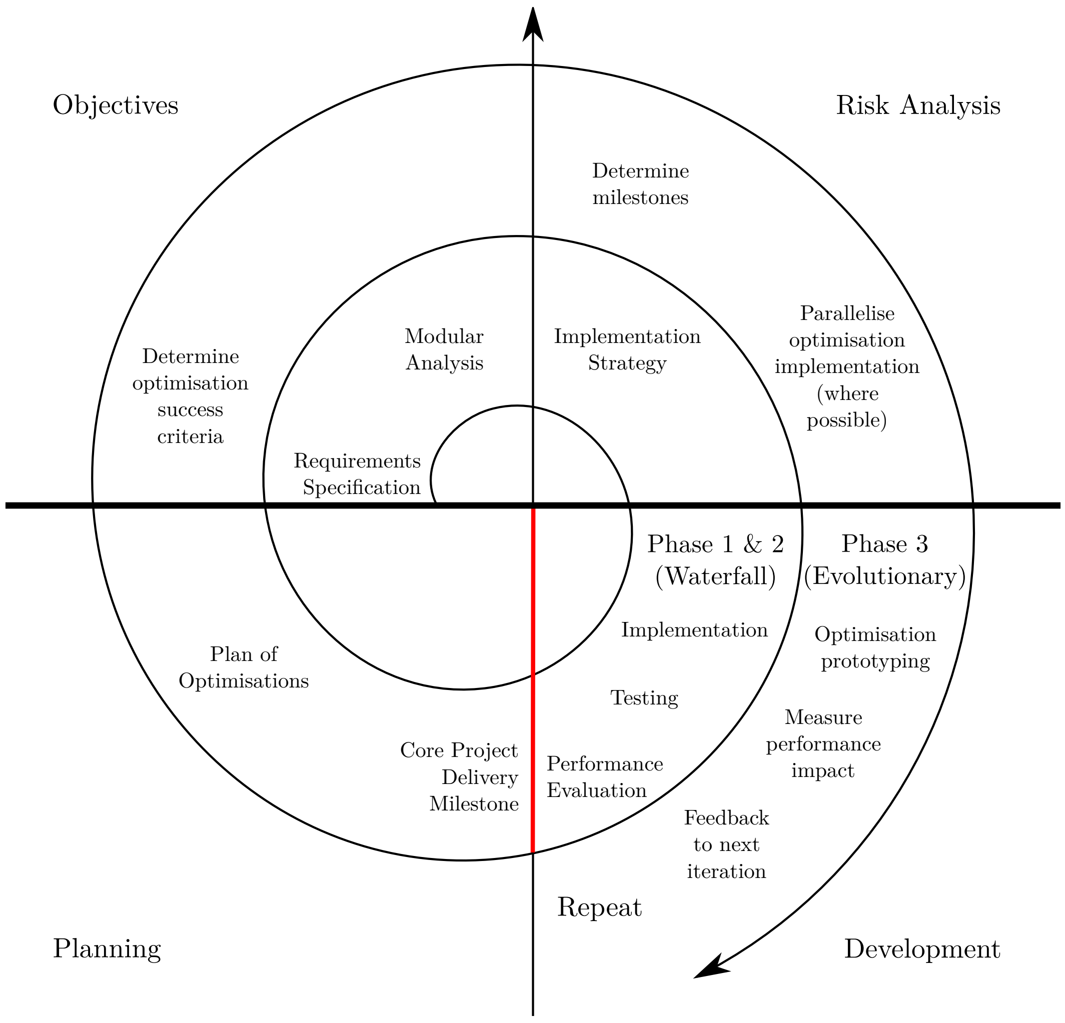


Figure 2.5: The spiral model *as it applies to the* Unbuckle *project. The red line denotes the point at which the core project is implemented, and thus the transition from sequential to iterative development.*

operating system internals was presented. The initial analysis of requirements and division of the project into manageable work packages was discussed to describe the planning work which took place to ensure the project delivered on its goals.

The following chapter provides a thorough breakdown of the implementation behind *Unbuckle* and the interactions which take place between its various modules.

Chapter 3

Implementation

This chapter describes the implementation of the *Unbuckle* key-value store introduced previously, as both a kernel-resident module and a user-space application for evaluation purposes.

As expected, the implementation process for this project proved to be a considerable undertaking: the final end-to-end system consists of more than 6,000 lines of source code. The most demanding components to build were those interacting directly with the kernel, where the failure modes were challenging and effective debugging was hard.

Given the complexity of the project, I will largely discuss the high-level modular structure and architectural design of *Unbuckle*. I will do so with reference to the specific data structures and algorithms used in its implementation and the various techniques adopted to optimise the performance of the kernel-based version. For brevity, I will not dwell on low-level technical details unless absolutely necessary.

The key system components described in this chapter are:

1. **Request pipeline** (§3.1.1) – outlines the lifecycle of a request to *Unbuckle* in terms of the operations necessary to receive requests, process them and transmit responses.
2. **Data structures** (§3.1.2) – introduces the mechanisms by which key-value mappings are stored in the back-end of the system for efficient retrieval, primarily by use of hashing.
3. **Network stack** (§3.2) – gives an overview of the core mechanism for data exchange between clients and the server which took two very different forms over the course of the project. The optimal network stack, purpose-built for *Unbuckle*'s use, minimises request latency by bypassing the kernel network stack and integrating the key-value store directly with the network hardware.
4. **Bucket allocator** (§3.3) – describes a memory pre-allocation scheme which reduces system latency by pre-allocating memory in buckets, allowing demands

to be met more quickly.

5. **Concurrency / multi-threading** (§3.4) – presents the parallelised version of the store, including details of how data structures were adapted to be thread-safe.
6. **Scheduling** (§3.5) – briefly discusses the adaptations made to influence the kernel's scheduler in the key-value store's favour.
7. **Port to user-space** (§3.6) – the mechanisms by which the store was dual-implemented for both kernel and user-space operation are discussed.

3.1 System architecture

3.1.1 Request pipeline

One of the first components to be implemented was the high-level request pipeline. This code represents the core of the key-value store by interfacing the various system modules together, co-ordinating the processing of a request from receipt to transmission of a response.

As in many distributed systems applications, I decided to adopt a state machine model to co-ordinate request processing. The organisation of states and their transitions is depicted in figure 3.1.

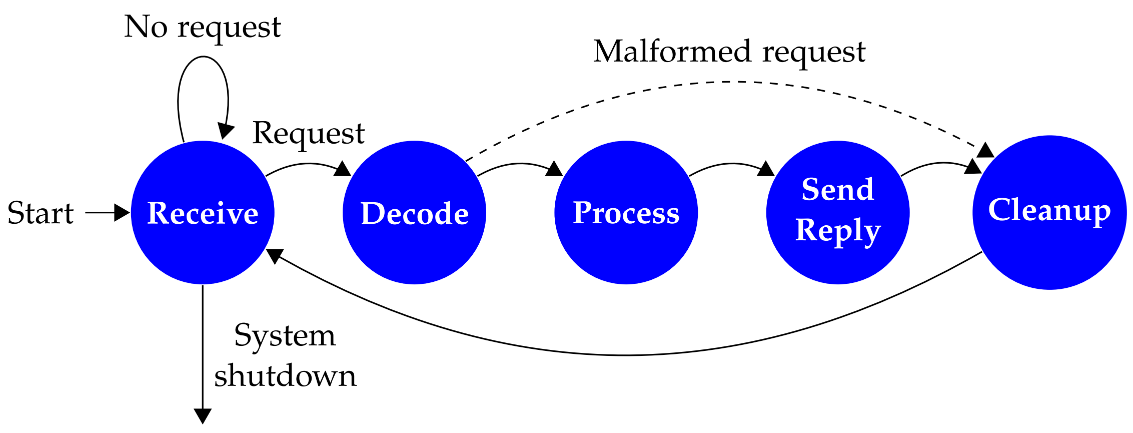


Figure 3.1: The state machine at the Unbuckle *core*

The pipeline is implemented by means of an `enum` field associated with each request. State-dependent branching is used to select the correct code to execute as requests make progress through the system. The five core states of the system are:

1. **Receive** – raw data are received from the network-handling code;
2. **Decode** – the packet is decoded, verifying its headers are correct and determining the action the client application wishes the key-value store to make;

3. **Process** – interaction takes place with the back-end data structures to retrieve the value associated with a given key in the case of `GET` requests, or to store a new key-value mapping in the case of a `SET` request;
4. **Send Reply** – a suitable response complying with the memcached protocol is built and queued on an output queue for transmission back to the client;
5. **Cleanup** – request-dependent state is cleaned up in preparation for a new request to be serviced.

The initial work was single-threaded for simplicity. In this version, the request pipeline was spawned as a new thread of execution within the kernel and responsible for all aspects of network interaction and request processing.

Later versions refined this approach. When multi-threading was added (§3.4), several worker threads were spun up to provide parallel instances of this pipeline. However, dedicated network worker threads were also introduced to co-ordinate the network communication. It was necessary to adapt the "receive" and "send reply" stages to exchange packets with the network handling threads by means of inter-process communication (IPC) techniques, rather than performing the network interaction as part of the core pipeline.

3.1.2 Data structures

Efficient storage of the key-value mappings in the back-end was an important undertaking. Several data structures were tested for suitability, including a linked list in the core implementation, and multiple hash tables as subsequent optimisations. Each of the data structures adheres to a standardised interface, a simplified variant of which is shown in listing 3.1. This enforces the treatment of the data structure as a black box by the other modules in the system, facilitating rapid prototyping and drop-in replacement of alternative back-ends at compile time.

```c
int  ub_store_init(void);
void ub_store_exit(void);

struct ub_entry *ub_store_find(char *key, size_t len_key);
int  ub_store_add(struct ub_entry *entry);
void ub_store_del(struct ub_entry *entry);
```

Listing 3.1: Interface for back-end data structures.

As a linked list is not particularly interesting and, as expected, showed poor performance under high load, I will only discuss the hash table here.

Hash table

A hash table proved to be a suitable data structure for the types of random access workloads observed in a key-value store. It delivers an average cost of $O(1)$ for the three provided operations: find key, add entry and delete entry. The kernel provides a standard hash table, implemented using C preprocessor macros for performance reasons, which was used as the basis for the hash table in the project.

To deliver constant time performance, a hash function must be selected to avoid excessive collisions in hash table buckets, as figure 3.2 demonstrates in terms of the *load factor*, α. This was ensured in two ways. A large number of buckets, m, were selected, to make the value of α negligible, and a hashing approach was carefully designed so as to minimise collisions as much as possible. There is no easy method for doing this, especially in this case, as the input keys are arbitrary and not drawn from any well-specified distribution.

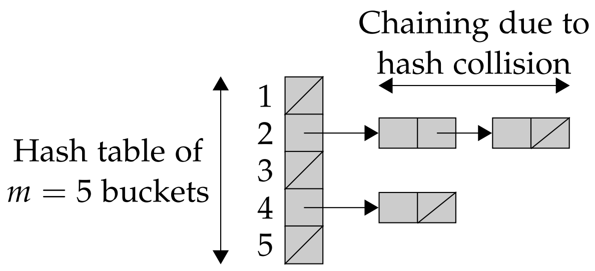


Figure 3.2: A simple hash table example demonstrating the effects of chaining due to the hash function suffering a collision. Due to chaining in bucket 2, any operation on the hash table could require traversing the linked list of chained nodes, increasing the overall cost of operations to $O(1 + \alpha)$, where α is the ratio between total items stored, n, and the number of buckets, m. Here, $\alpha = \frac{n}{m} = \frac{3}{5}$. A constant value of α, which does not vary as a function of n, produces $O(1 + \alpha) = O(1)$ constant time.

For the hash function, the SpookyHash function (introduced in §2.4.1) was used to collapse the entropy of an arbitrary length memcached key onto a 64-bit integer. This is subsequently hashed again by the kernel upon insertion into the hash table.

The choice of SpookyHash was motivated by several key factors:

- **Designed for performance** – among hashes, SpookyHash is particularly fast, hashing, in the best case, one byte per clock cycle.
- **Avalanche property** – each bit of the computed hash depends on a large number of input key bits, ensuring the input entropy is distributed as much as possible throughout the output.
- **Consistency** – SpookyHash is also used internally by memcached. While this was not my primary motivation for selecting this hash, its use reduces bias during the performance evaluation.

Internal data structures

In addition to the back-end, several additional data structures are necessary for internal processing of state, especially to track the progression of a request along the request pipeline. The high overhead in performing memory allocation means special care needs to be taken to ensure memory is reused where possible, rather than slowing the request pipeline by re-allocating memory during request processing.

3.2 Network stack

Key-value stores are typically network services, intended to act as a building block for data storage in large distributed systems. *Unbuckle* is no exception to this rule, for it is intended to replicate the function of memcached as a cache for a large number of application servers requiring rapid access to data.

The construction of a scalable network stack which takes advantage of the benefits of operating within the kernel to minimise per-request processing latency is thus a central component of the project.

As discussed previously in §2.3, the implementation of a full network stack interacting with networking hardware at the lowest level represented a high-risk task. Hence, two network stacks were implemented at different stages of the project: one using the socket interface, and another which specialises the IP and UDP processing code specifically to this project's requirements.

3.2.1 Socket interface

The socket interface represents the most common and well understood networking API on UNIX-based platforms [50]. It abstracts the complexity of network interaction, protocol implementation and hardware details away from user-space code into *sockets* and a set of related kernel system calls, as shown in figure 3.3. The interface is a fundamental component of many client-server applications due to its ease-of-use and support for modern network and transport-layer protocols.

Familiarity with the socket interface from user-space programming made it the obvious choice for the first key-value kernel module, thereby minimising the inherent risks in writing a full custom network stack as part of the core project.

My code integrates at the entry point to the kernel network stack from user-space, just below the system call layer. Consequently, the code functions almost identically to user-space code, with only minor modifications necessary to use kernel data structures rather than numeric file descriptors.

Interacting with the socket API within the kernel also permitted early performance evaluation. The only difference between this interaction and the comparable user-

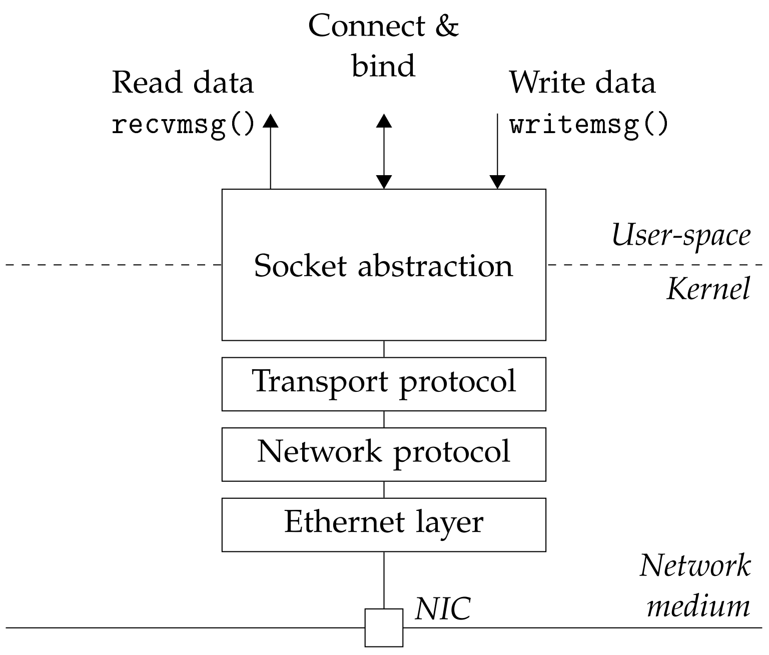


Figure 3.3: The socket interface abstracts many low-level details into the kernel.

space operation is the initial mode of the processor when the operation is invoked. *Unbuckle* executes in kernel context, so accesses the API without performing any system calls. The hypothesis that the system call interface imposes significant overheads on user-space packet processing code may thus be tested by running the store in both user and kernel-space and comparing results (see §4.5.1).

3.2.2 *Unbuckle* low-level network stack

A later iteration of the network module made use of the exposed kernel interfaces to integrate directly with the network hardware. To do this, I implemented a custom IP and UDP protocol stack for packets destined for the key-value store. This was specially optimised for the memcached protocol.

Unlike related work in this area, the final solution retains cross-platform compatibility by using only the standard kernel interfaces, delivering on a design requirement discussed in §1.1 and §2.3. In particular, all network hardware ever supported by the Linux kernel is compatible; this stack is not restricted to a subset of expensive, high-performance network interface cards (NICs), unlike other similar research systems (see §1.3.2).

Linux network stack

In order to determine a suitable location to intercept packets as they arrive from hardware, it was first necessary for me to gain a thorough understanding of the lifecycle of a packet in the standard Linux network stack, from arrival at a NIC up

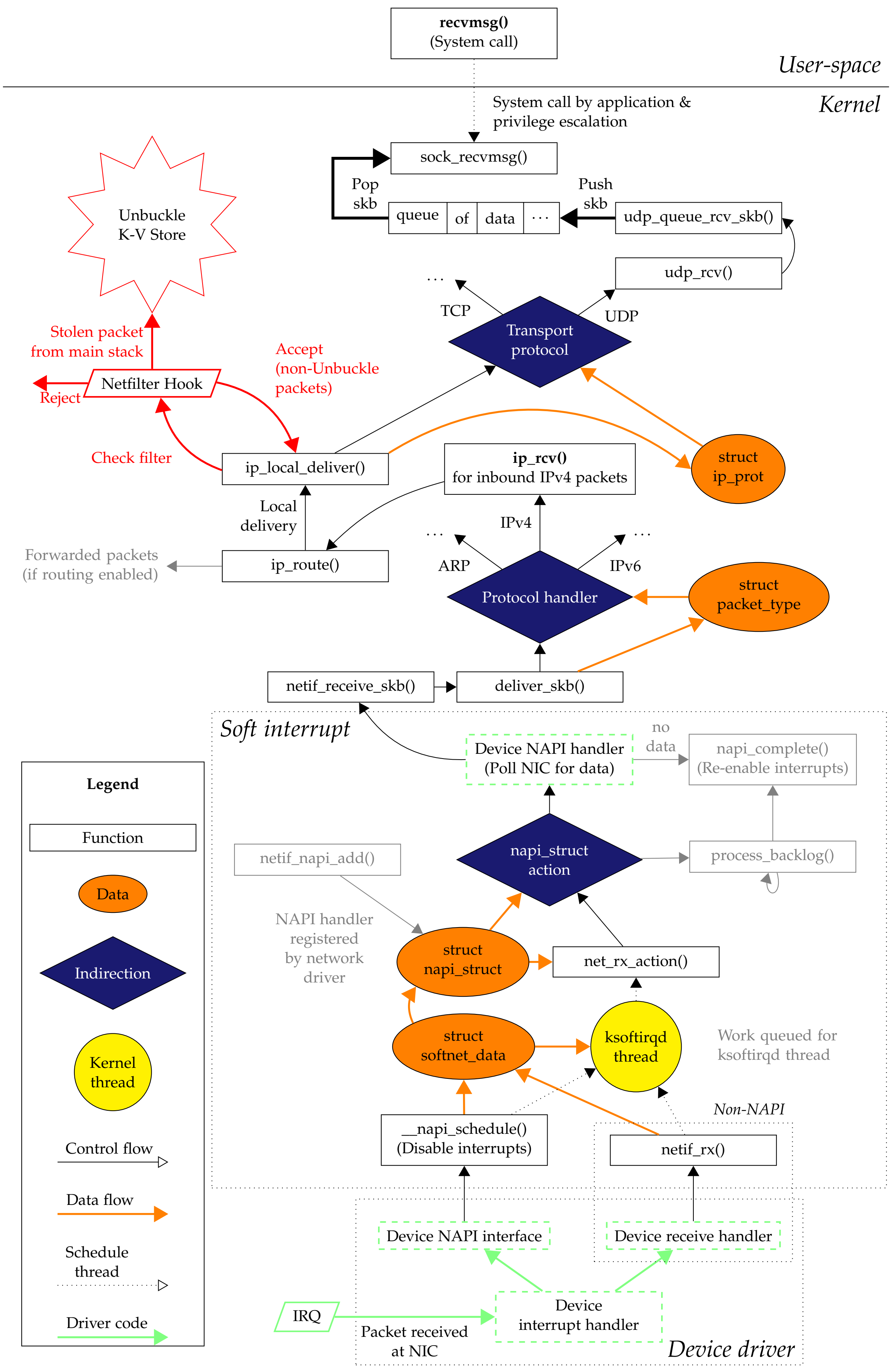


Figure 3.4: The internals of the receive path of the Linux network stack.

to delivery to an application. The complexity of the network stack made this a time-consuming task. The culmination of my analysis is presented in figure 3.4.

The socket abstraction within the kernel is a misnomer, as transmit and receive operations follow independent paths in the network stack and must be addressed independently. However, it does provide a useful queueing capability for received packets waiting to be consumed by an application. Throughout the stack, packets are represented by a socket buffer data structure, `struct sk_buff`, commonly known in the kernel community as an `skb`.

The kernel's compatibility with a range of networking hardware dictates a need to cleanly interface the various supported protocols and platforms together into a single cohesive stack at every layer. Indirect function calls are widespread in this code to enable late binding and runtime registration of new protocols with the kernel. This makes tracing the path taken by a packet more challenging than simply following function calls along the stack, as the protocol descriptor identifying the function to be invoked is elsewhere in the kernel code.

Despite the network stack's complexity, Linux has been subject to extensive work to improve the performance of its network stack under high throughput workloads. Several features introduced for this purpose are demonstrated in figure 3.4, including the following two:

- **The New API (NAPI) interface** mitigates the number of interrupts raised by a NIC by polling for new data when the rate of packet arrivals is high. Under the assumption that a busy NIC will remain in this state for some time, this reduces interrupts and significantly increases packet throughput. Network drivers must be adapted to provide NAPI support, so the historic path involving an interrupt per packet is retained for compatibility.
- **Bottom half processing** – as the name suggests, interrupts immediately halt any active work on the processor, which can create performance difficulties and prevent the system from progressing with useful work. To maintain system responsiveness, modern device drivers defer work to other kernel contexts where it has no priority advantage. One method for achieving this is the *soft interrupt* mechanism shown in the network stack, implemented by a set of `ksoftirqd` threads.

Performance bottlenecks arise at the top of the stack, where the interface with applications requires system calls and privilege-level changes to take place. Furthermore, the socket interface's packet queue introduces latency, as there is necessarily some delay between writing to this queue and an application processing the signal that data is waiting.

There is much more that could be said about the Linux network stack, but for space constraints, I leave further discussion to relevant literature [6] [13, Ch. 17]. In the following, I discuss how *Unbuckle*'s low-level network stack integrates with the Linux kernel stack.

Integration location

Ideally, the key-value store would be integrated with the network stack in such a way that latency due to queueing is minimised, while simultaneously avoiding unnecessary code duplication.

	Location	Benefit	Drawback
✗	**NAPI action handler:** Register with *napi_struct action* indirection node	Low latency – low in the stack and polls NIC directly.	Driver-specific: premature hardware specialisation.
✓	***netfilter* interface:** Register callback with IP-layer netfilter hooks	IP routing and packet fragmentation available.	A malfunctioning hook may drop packets for other applications.
✗	**New protocol handler:** Register with `struct packet_type` indirection node	Standard network stack behaviour for new protocols.	Not drop-in replacement as client code must be changed. Risk of layering violations.

Table 3.1: Possible locations for the key-value store to integrate in the network stack.

By careful analysis of the network stack, a set of possible integration locations was drawn up. These are listed in table 3.1 alongside an indication of their suitability. Of particular importance here are the requirements for hardware independence and co-operation with other networked applications on the machine.

Ultimately, I decided to extract packets from the stack using the *netfilter* interface, provided as part of the IP protocol implementation. A netfilter hook comprises a callback function dynamically registered with the kernel, which is queried for every inbound IP packet. The function can inspect and mangle the packet as it wishes, returning one of three decisions to the network stack:

- `ACCEPT` – the packet continues up the stack towards the socket interface;
- `REJECT` – the packet is dropped and its memory freed; or
- `STOLEN` – the netfilter hook becomes responsible for onward processing of the packet.

The possibility of accepting and rejecting packets makes netfilter a common tool in the implementation of firewalls. The lesser known option to steal a packet is very useful here, allowing the *Unbuckle* kernel module to inspect inbound packets and filter off only those destined for its UDP port number. As this occurs within the IP protocol handler, it remains possible to benefit from the kernel provided IP fragmentation. This permits data up to the maximum size of an IP packet, 65 KiB,[1] to be transferred, if support for this was added to the memcached protocol.

[1] 1 KiB = 1024 bytes, where KiB means *kibibyte*.

This requires transcending layer boundaries to inspect UDP port numbers, but is no different from the behaviour implemented by a firewall. In particular, the integrity of packets not destined for the key-value store is retained as they are not modified.

```
unsigned int
ub_udpserver_nethook_callback(..., struct sk_buff *skb, ...)
{
  struct iphdr  *iph  = iphdr(skb);

  /* Verify the packet uses the UDP protocol.
     Non-UDP packets cannot be for Unbuckle. */
  if (iph->protocol != IPPROTO_UDP)
    return NF_ACCEPT;

  struct udphdr *udph =
    (struct udphdr*) ((char*) iph + iph->ihl * 4);

  /* Check whether the UDP packet is destined for Unbuckle,
     or some other UDP port on this system */
  if (ntohs(udph->dest) != UB_UDP_PORT)
    return NF_ACCEPT;

  /* Control flow reaching this point indicates the packet
     is for Unbuckle. Perform onward processing and halt
     further network stack progression. */
  process_request(skb);
  return NF_STOLEN;
}
```

Listing 3.2: An example netfilter *hook for filtering off UDP packets for* Unbuckle.

An example of the code required to inspect a network packet and steal it when appropriate is given in listing 3.2.

This approach permits requests to be processed directly in the `ksoftirqd` handler, reducing the latency between the packet arriving at the machine and it moving into service. The loss of the queue in the socket interface to buffer packets under high system load is not an issue, as the queues and memory buffers allocated within most modern NICs are typically sufficient. In any case, under high load, the NIC becomes a bottleneck far more quickly than the queue at the top of the stack.

Transmission

With a method to receive network packets directly from the hardware, our attention must now turn to a method of transmitting responses. The socket interface packages

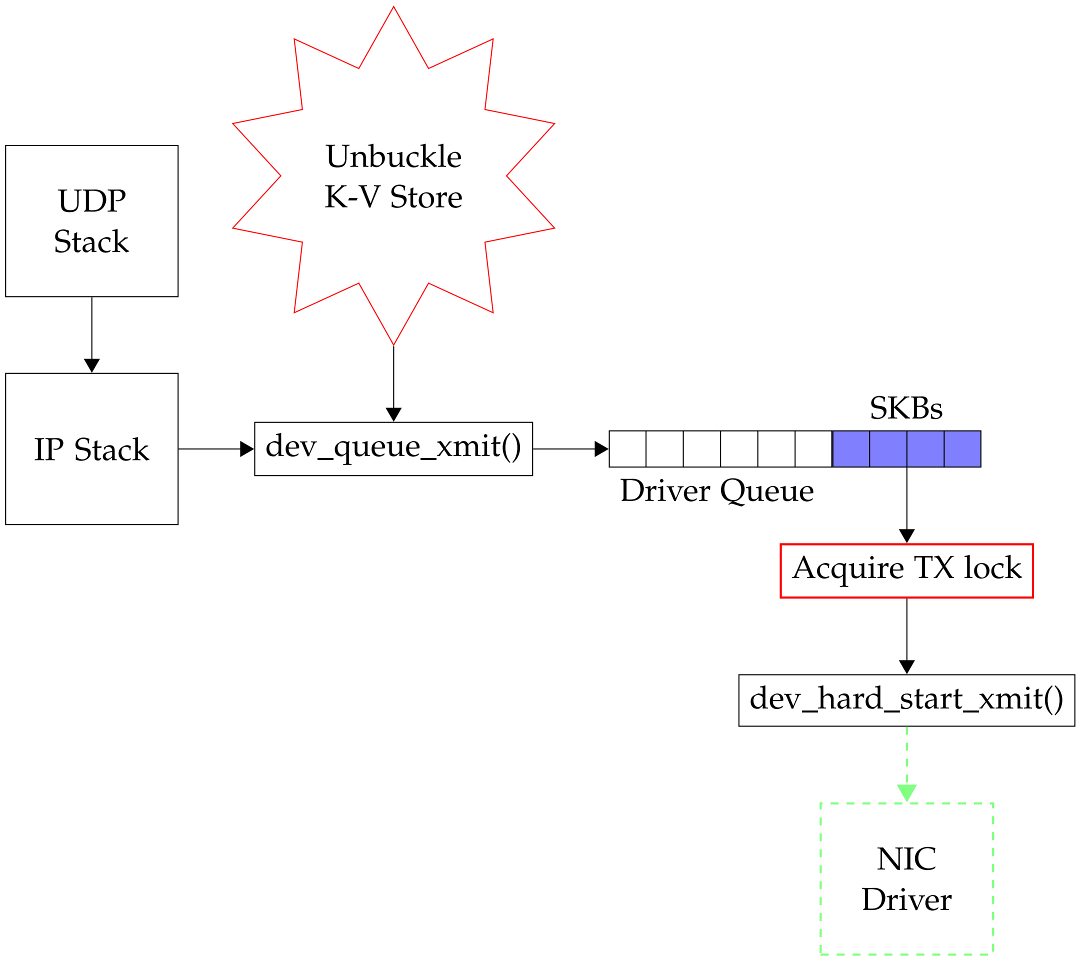


Figure 3.5: The UDP transmission path in the Linux network stack.

both receive and transmit operations into a single abstraction, but the two paths are distinct in the network stack and must be implemented independently. As expected, the socket interface is overly general and exhibited subpar performance, motivating an alternative low-level solution.

The Linux network stack for transmission is mostly a reverse of the aforementioned receive path. A brief overview of the transmit path is shown in figure 3.5. A key requirement of the transmit handlers is their generation of suitable protocol headers based on data contained in the socket descriptor. If the socket interface is not used, the responsibility for generating such headers falls to the key-value store.

Queueing also plays an important role in the transmit path to ensure packets are appropriately prioritised based on user-selected queueing disciplines, for example to allow latency-sensitive packets to jump ahead of others.

Once again, there were several choices as to possible integration locations, as detailed in table 3.2. Based on these options, the choice was made for packets to be injected in the prioritisation and queueing stage. This avoided the need to acquire locks to transmit on a NIC and ensures the selected queueing discipline is enforced on the output packet flow, providing administrative control over the priority of packets in accessing the NIC.

	Location	Benefit	Drawback
✗	**Socket interface** Abstracts over low-level network stack details.	Packet headers generated automatically.	Subpar performance as too general.
✓	***dev_queue_xmit()*** Passes complete socket buffer to queueing functions.	Respects a user's traffic prioritisation requirements.	Packet header generation must now be implemented correctly.
✗	***dev_hard_start_xmit()*** Last generic kernel method before NIC transmit path is invoked.	Low latency – packet ejected directly to NIC TX buffer by DMA.	Contention in acquiring TX lock. Does not respect a user-configured output queueing discipline.

Table 3.2: Possible locations for the key-value store to integrate with the network stack for transmission.

Data structure optimisations

The work invested in constructing the network server permitted further performance optimisations to the data storage in the back-end hash table. This ultimately produced a request pipeline with significantly fewer memory copy operations than comparable code would require using the socket interface.

The availability of the raw `struct sk_buff` packet data structures permits this format to be used for storing key-value mappings in the hash table. Rather than incur the overhead of constructing a new socket buffer to send a response to every request, the previously constructed socket buffer for the request key-value mapping is simply returned from the hash table, destination UDP, IP and Ethernet headers are prepended, and the result is written to the output queue, reducing request latency.

3.3 Memory pre-allocation

As an in-memory key-value store, *Unbuckle* places high demands on the availability of system memory to store data added to the cache by a memcached `SET` request. The kernel memory allocator, `kmalloc()`, provides the kernel module with access to physical, pinned system memory, without the burden of a virtual memory abstraction. However, profiling of an early iteration of the store demonstrated that inefficiencies abound in on-demand memory allocation, namely:

- **Internal bookkeeping overheads** induce latency as high as 1,000 cycles in the memory allocator. This is further hindered by the kernel treating itself as sacrosanct, always attempting to satisfy its own allocation requests even if this requires moving memory or swapping user-space memory pages out to disk.

- **Internal fragmentation** due to the unit of memory allocation being a memory page, typically 4 KiB on an x86 system. Key-value store data sizes are rarely larger than several hundred bytes [41], so calling the allocator and only using a fraction of the allocated page leads to wastage.

A custom *bucket allocator* was implemented to amortise these memory allocation costs and more efficiently utilise the allocated memory. The single change of removing the memory allocation stage from the processing path of a `SET` request was highly effective, reducing latency and its variability for `SET` requests by as much as 60%.

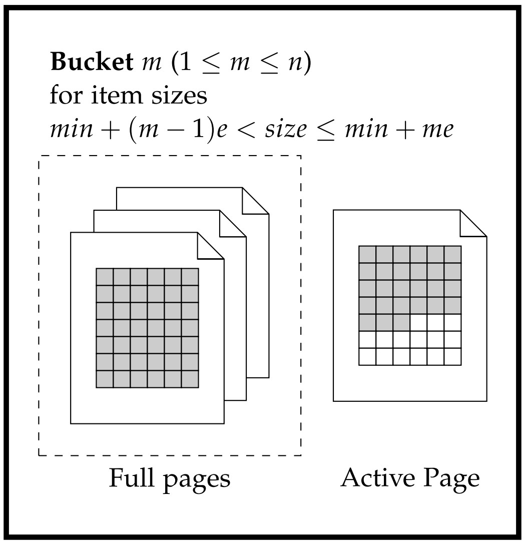

Bucket m $(1 \leq m \leq n)$
for item sizes
$min + (m-1)e < size \leq min + me$
Full pages
Active Page

Figure 3.6: An example bucket used in the bucket allocator *for amortising memory allocation overheads and bounding internal fragmentation. There are n such buckets, each of which is assigned one or more memory pages to store items in a fixed size interval. The size of the intervals is user-definable by a minimum bucket size, min, and a growth factor, e, which defines the width of the interval.*

This system follows the principles of the slab allocators often used for memory management [8], with some further specialisations for this particular implementation. The range of possible sizes for a key-value pair is divided into non-overlapping segments, or *buckets*, ranging from 32 bytes to 1 MiB, the largest item supported by the memcached protocol.

Each bucket is responsible for a small range of this space according to a logarithmic scale. This makes sense as smaller keys and values are more common and thus warrant finer granularity to reduce fragmentation. The factor defining the growth rate of this scale is user-configurable to permit system tuning.

Buckets are allocated one or more pages of physical memory, which are subdivided into chunks for storing the individual key-value pair mappings, as shown in figure 3.6. In this way, memory only needs to be allocated when all of a bucket's pages become full, and the bounded data stored in each bucket provides an upper bound on the maximum degree of memory fragmentation.

3.4 Concurrent request processing

3.4.1 Overview

The early development of *Unbuckle* focussed on a single threaded implementation of the store, which delivered some performance gains over user-space and comparable commercial systems. However, the law of diminishing returns led to an impenetrable performance wall, with throughput eventually stabilising at approximately 3 Gbps, well below the 10 Gbps target.

To continue improving, it became necessary to explore threading techniques to explicitly parallelise the workload. Key-value stores are excellent candidates for such parallelisation, as requests to the store are independent of one another and require minimal co-ordination owing to their relaxed semantic guarantees.

Many synchronisation primitives are already available to kernel module developers through work done to benefit from multiprocessor systems. Nevertheless, these tools do nothing to ease the complexity of writing safe code without overzealous use of locks.

3.4.2 Multi-threaded architecture

The concurrent version of *Unbuckle* re-used earlier work as much as possible in its modified structure. Most of the core logic of the system required minimal modification, with the exception of two modules: the back-end data storage and the network server. These were areas in which correctness would be compromised through naïve re-use of code, thus demanding more careful attention.

The revised system employs three distinct bodies of code:

- The **request router** provides the low-level interaction on the network stack receive path. It intercepts key-value store requests and routes them to worker threads for processing.
- Multiple **worker threads** process requests according to the standard request pipeline described in §3.1.1.
- The **transmit manager** co-ordinates transmission of response data from each of the worker threads onto the NIC, with particular care taken to avoid lock contention in the network stack's transmit path, which could clobber performance gains achieved elsewhere.

The high-level design of the system is shown in figure 3.7.

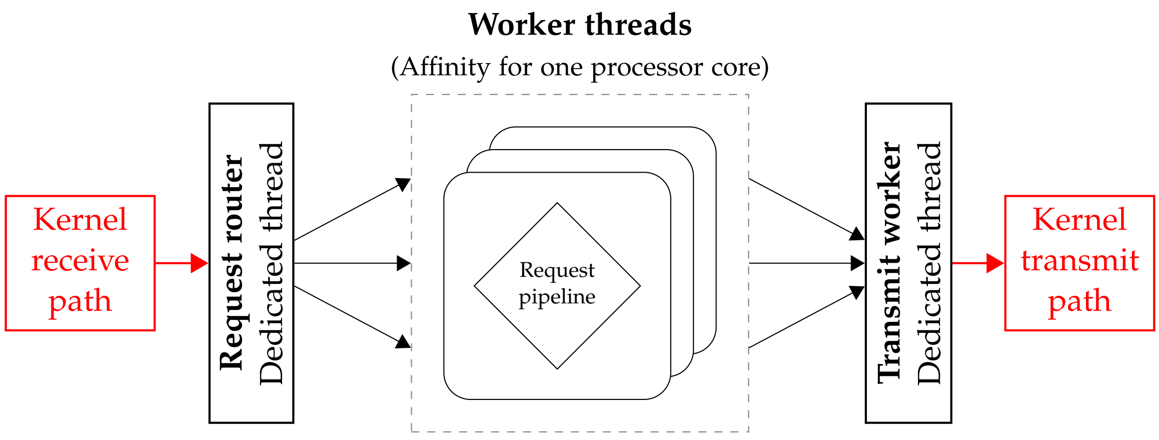


Figure 3.7: The multi-threaded workflow in Unbuckle.

3.4.3 Work distribution

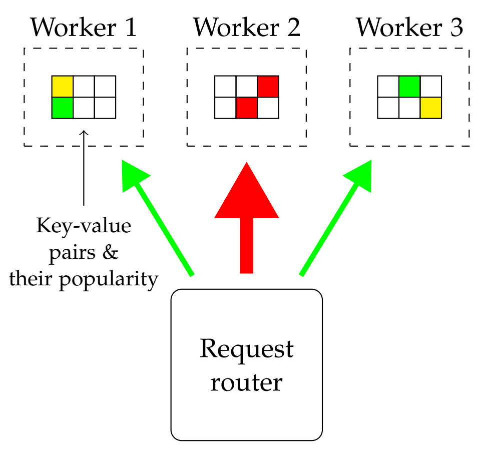
A number of possibilities were available with respect to routing requests to each of the worker threads, including deterministic key routing and round robin. These approaches are illustrated in figure 3.8 and evaluated in the following sections.

Deterministic key routing

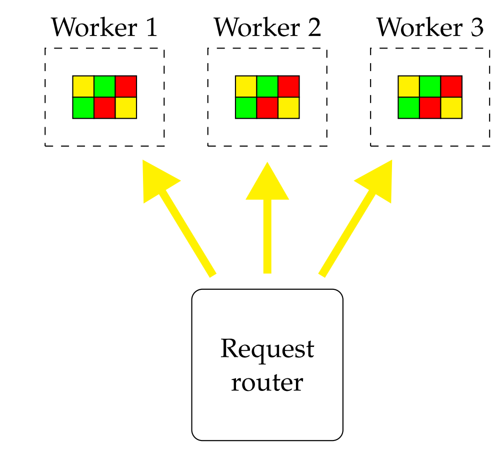
This scheme subdivides responsibility for the key space between each of the workers. Inbound requests are inspected and passed to the appropriate worker for their requested key, as shown by the pseudo-code algorithm in listing 3.3. The motivation is for each thread to benefit from local CPU caching to keep its working set of keys closer than DRAM, reducing latency.

```
static int number_of_worker_threads;

thread_id route_request_to_thread(char *key)
{
  /* Hash key to fixed width integer using Spooky */
  uint64_t key_hash = spooky_getHash(key);

  /* Route to correct worker thread */
  int thread = key_hash % number_of_worker_threads;

  return thread;
}
```

Listing 3.3: Request routing in the deterministic key routing scheme.



(a) ***Deterministic key routing*** *– worker 2 is overwhelmed due to the relative popularity of the keys under its jurisdiction, despite other workers standing mostly idle.*



(b) ***Round robin routing*** *– each worker services requests for the entire key space, ensuring hotspots in the key distribution do not saturate individual workers.*

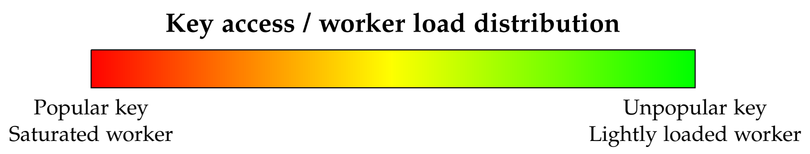


Figure 3.8: Comparison of two approaches to distributing work between worker threads.

Although this approach to routing would be optimal if all key-value pairs were uniformly popular, this is not the case in real-world deployments, which observe a power law distribution in the frequency of lookups of particular keys [4].

If this approach was adopted for *Unbuckle*, the distribution of work across threads could be highly non-linear, as shown in figure 3.8a. This would eventually lead to cascading system instability across the distributed system as requests for popular keys to overloaded worker threads time out. Back-end database servers would instead be queried to render this data, increasing load and risking congestive collapse.

Round robin routing

The round robin scheme which was actually taken forward for implementation takes the view that any worker thread can process any inbound request. Under modest load, requests will be shared equally between the workers irrespective of the key specified, as shown in figure 3.8b. Although this approach is more complicated to implement safely, it assists with scaling up to the long-tailed distribution of key-value pair accesses. The algorithm implementing this is given in listing 3.4.

Despite this methodology reducing temporal and spatial locality for caching, it is expected that the shallow hash table described in §3.1.2, in combination with a shared L3 cache, continues to permit popular keys to be served from cache rather than DRAM. Moreover, exceptionally popular keys are likely to be retained in the private caches of individual processor cores.

```c
static int number_of_worker_threads;

thread_id route_request_to_thread(char *key)
{
  /* Last thread a request was routed to */
  static int last_thread;

  /* Route to the next thread in the round robin,
     with wrap around, if necessary */
  thread_id route =
    last_thread++ % number_of_worker_threads;

  return route;
}
```

Listing 3.4: Request routing in the round robin routing scheme.

Hybrid approach

I will briefly note here that a hybrid of the two mentioned schemes may be appropriate in some circumstances, most notably on machines employing non-uniform memory access (NUMA) memory hierarchies.

As many-core chip multiprocessors grow in popularity, the memory gap between processor and DRAM performance becomes a burden to scalability [56]. NUMA is a workaround to this problem which makes sets of processor cores responsible for subsets of main memory. A group of one or more processors sharing a subset of main memory is termed a *NUMA node*. NUMA ensures memory bandwidth continues to scale with processor core count for parallel applications. Threads maintain affinity to particular NUMA nodes, staying local to the subset of memory containing their working set. Accessing remote system memory requires crossing an off-chip bus to another node, which introduces latency.

Careful scheduling of key-value lookup requests on such systems may benefit from a layered routing scheme: deterministic key routing to a nominated NUMA node, followed by round robin routing to individual workers. This ensures worker threads spend the majority of their time accessing local memory to service requests, without the risk of saturation in the presence of popular keys. However, the performance tradeoffs are likely to vary across NUMA architectures, so machine-specific configuration would be necessary to obtain optimal performance. NUMA-aware scheduling support is still in its infancy in Linux and further discussion of

NUMA is beyond the scope of this dissertation, although alternative sources describe it in detail [24, Ch. 5].

Inter-process communication (IPC)

Once a routing approach had been chosen, it was necessary to construct the infrastructure for communicating work units to the worker threads. The kernel offers a variety of methods for IPC, including soft interrupts, tasklets and work queues [55], but none of these techniques are suitable. They all involve the kernel scheduler, which induces extra high latency due to kernel delay in scheduling work – as much as a $5\times$ increase to $1{,}000 \pm 800\ \mu$s. A dedicated communication scheme involving queues of network packets, represented by `sk_buff` data structures, was instead constructed for this purpose. The work done to adjust the behaviour of the scheduler is described in more detail in §3.5.

3.4.4 Synchronisation primitives

Of the few guarantees provided by the memcached protocol, perhaps the most important is the assurance that client operations on the store will be atomic – either they occur in their entirety or not at all, and readers observe either the old or new state, not a corrupted mixture of the two.

To ensure data are not modified by multiple threads simultaneously, the correct implementation of locking to key data structures was necessary in the multi-threaded version.

Hash table

Protection of the hash table was implemented by way of a reader-writer locking mechanism, which permits multiple concurrent reader threads to proceed in parallel. Updates to the data structure take place under an exclusive lock which prohibits any other readers or writers during the time the lock is held, thereby protecting the integrity of the data structure.

Special care was taken to reduce contention for the lock by ensuring only the essential lines of code were protected in the critical section. Moreover, to minimise the risk of concurrency bugs, locks were confined to the hash table and did not cross the external interface to other modules. The operation of this locking system is demonstrated in figure 3.9.

This design was motivated by an analysis of a large memcached deployment at Facebook, which indicated as many as 97% of requests to their memcached clusters are `GET` requests [41]. It may be possible to implement a more complex locking strategy which permits multiple writers, but since writing does not represent the common case, any real-world performance gain would most likely be negligible.

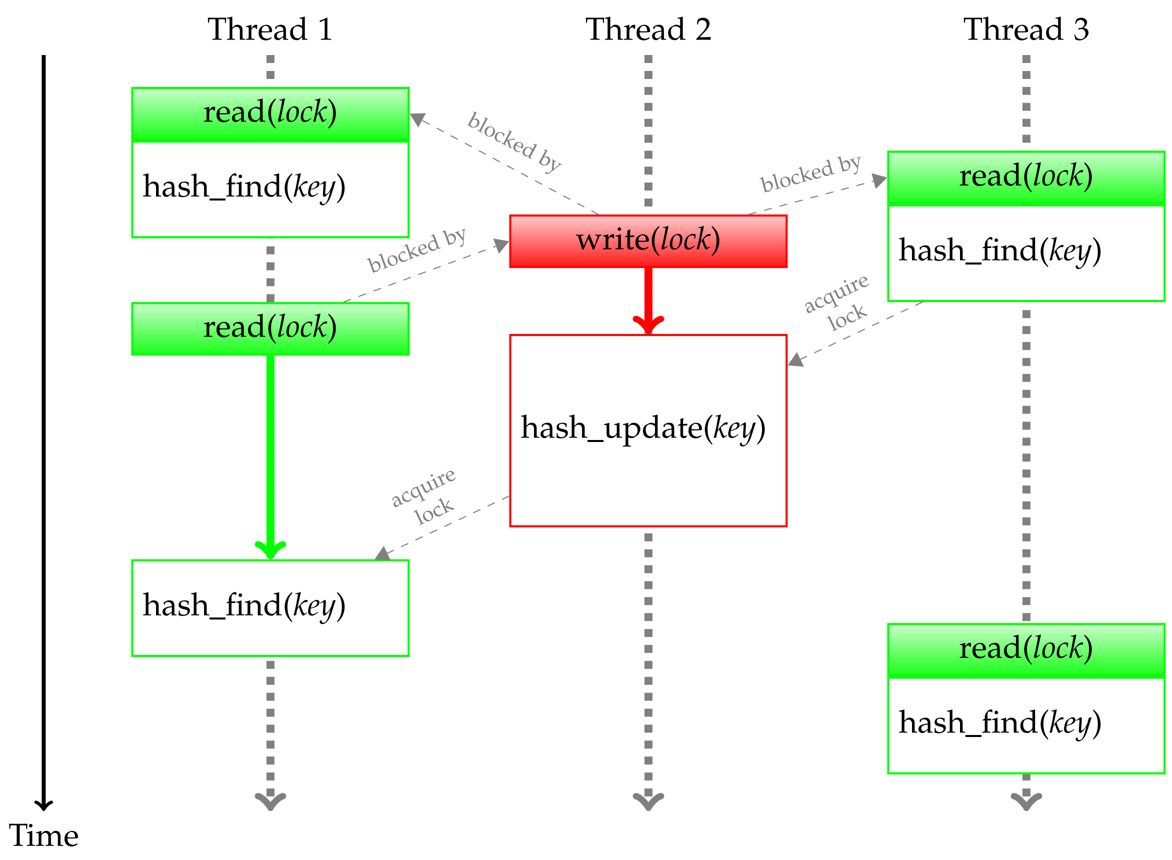


Figure 3.9: A time sequence diagram demonstrating the co-ordination of three workers using a multiple readers, single writer synchronisation scheme.

Thread queues

Two additional locations require the use of locking to co-ordinate the behaviour of threads:

1. The interface over which the request router queues work up for individual worker threads; and
2. the equivalent interface at the end of the pipeline, when a worker queues an outbound packet for the transmit worker to send.

To minimise latency, this co-ordination is implemented by means of a simple spinlock. Contention is reduced in each case by providing each worker thread with a local receive and transmit queue, which the request router and transmit worker access for each thread in order to push and pop packets.

3.5 Scheduling

One of the many benefits to moving into the kernel is the ability to interact directly with the system scheduler.

The scheduler is the kernel subsystem which solves the *scheduling problem* to allocate finite processor resource to processes which are waiting to run. Processes are typically allocated a bounded period of time, a *quantum*, during which they can use the processor before the kernel recomputes priorities and considers pre-empting active threads. The decisions made by the scheduler can be influenced to some extent by user-space code, but the behaviour ultimately remains at the mercy of the kernel, who can choose to ignore the provided hints.

Working inside the kernel, this subterfuge can be overruled. *Unbuckle* takes advantage of the available control layers to minimise scheduling behaviours which are a detriment to performance. It makes a number of configuration tweaks, especially in the multi-threaded version:

- **Hard thread pinning** is configured by setting the processor affinity for each thread used by the system. Without this configuration, the kernel often migrates threads between cores, causing performance to suffer during the context switch and after due to the loss of cache affinity.
- **Threads are prioritised** to always run and are **not pre-emptible**. This permits *Unbuckle* to hold a processor for as long as necessary to complete the active requests to the store, without risking a process context switch.
- On NUMA systems, **NUMA-aware thread pinning** is used to ensure thread processing (and memory allocation) does not cross NUMA boundaries, which can have poor performance due to crossing an off-chip bus, as described in §3.4.3.

The result of these changes is demonstrated in figure 3.10, which shows the multi-threaded version of the store in operation. The priority given to the *Unbuckle* threads is clearly visible.

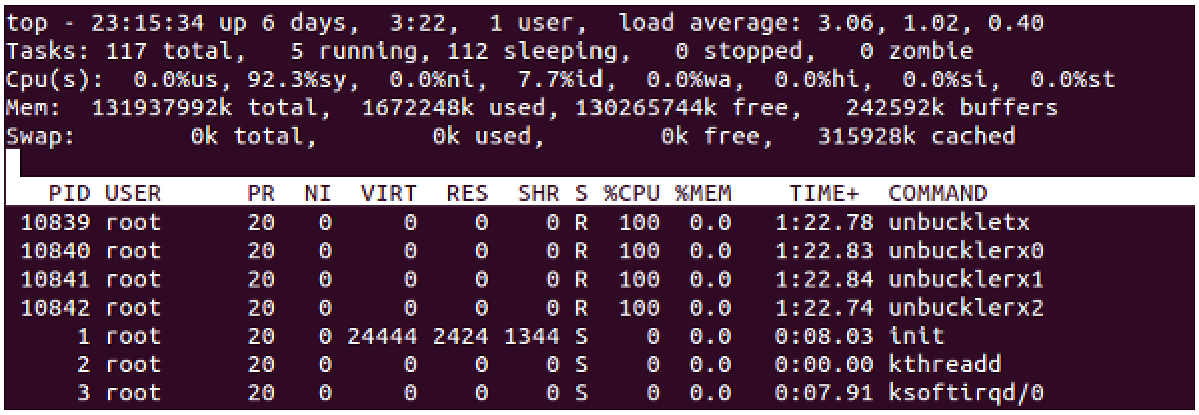

```
top - 23:15:34 up 6 days,  3:22,  1 user,  load average: 3.06, 1.02, 0.40
Tasks: 117 total,   5 running, 112 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us, 92.3%sy,  0.0%ni,  7.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  131937992k total,  1672248k used, 130265744k free,   242592k buffers
Swap:        0k total,        0k used,        0k free,   315928k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
10839 root      20   0     0    0    0 R  100  0.0   1:22.78 unbuckletx
10840 root      20   0     0    0    0 R  100  0.0   1:22.83 unbucklerx0
10841 root      20   0     0    0    0 R  100  0.0   1:22.84 unbucklerx1
10842 root      20   0     0    0    0 R  100  0.0   1:22.74 unbucklerx2
    1 root      20   0 24444 2424 1344 S    0  0.0   0:08.03 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:07.91 ksoftirqd/0
```

Figure 3.10: Output from the top *command on a machine running the multi-threaded kernel version of* Unbuckle *described in §3.4. This demonstrates the worker threads (`unbucklerx`) and transmit worker (`unbuckletx`) occupying their respective processor cores with 100% utilisation. The request router runs as part of the* netfilter *hook in soft interrupt context, under one of the `ksoftirqd/N` threads, where* N *is the processor core SMP identifier. The soft interrupt handler for processor core zero is shown at the bottom of the screenshot.*

3.6 User-space *vs.* kernel-space

A unique problem for *Unbuckle* was the requirement that the store should function in two orthogonal scenarios: as a user-space application and as a kernel module. Ideally, minimal modifications would be made to code compiled for either mode to ensure validity of performance comparisons.

This task was non-trivial: as previously described in §2.2.1 and §2.5, programming paradigms differ significantly between kernel and user-space. Code written for the former is incompatible with the system call interface provided in the latter, and vice versa.

A survey of modules and their suitability for porting is given in table 3.3.

Module	**Portable?**	**Difficulty**	**Notes / Mitigation in user-space**
Request pipeline	✓	Low	Pipeline & packet decode unchanged.
Kernel hash table	✗	–	Use UTHash (§2.4.1) instead.
Socket interface	✓	High	Socket function call semantics different.
Low-level network stack	✗	–	No direct access to hardware. Must use socket interface.
Bucket allocator	✓	Low	Allocate memory by system calls – `malloc(1)` rather than `kmalloc(2)`.
Concurrency library	✗	–	Depends on non-portable low-level network stack. Kernel thread implementation incompatible with `pthreads`.
Scheduling	✓	Medium	Requests may not be honoured.

Table 3.3: Suitability of Unbuckle *modules for user-space.*

To facilitate on-demand transitions between the two versions, an abstraction layer was constructed to hide the external interfaces. The compilation process was re-configured to swap in an implementation of the abstraction layer compatible with the target compilation mode, a process greatly simplified by the preparatory work to plan the interfaces between modules.

For the majority of function calls, method signatures in the kernel and user-space are similar. This permitted C preprocessor macros to be used for the abstraction, which minimises detrimental performance impacts by specialising to a particular mode of operation at compile time, rather than at runtime.

However, there were cases where differing semantics made the port more challenging. An example is given in listing 3.5, where the method provided for reading data from an open socket is compared between the user-space and kernel interfaces.

```
/* Kernel */
int kernel_recvmsg(struct socket *sock,
                   struct msghdr *msg,
                   struct kvec    *vec,
                   size_t num,
                   size_t size,
                   int    flags);

/* User-space */
ssize_t recvmsg(int sockfd,
                struct msghdr *msg,
                int flags);
```

Listing 3.5: Method signatures for receiving data from a socket in the kernel and user-space.

The kernel's `kernel_recvmsg` method expects the internal kernel data structure describing the socket to be passed as its first argument. However, in user-space, these details are hidden behind a numeric socket identifier passed to the system call `recvmsg`. Another problem is obscured in the method signature, but arises when the semantics of each method are considered. The memory buffer provided[2] is expected to be a different type of memory address in each of the two methods – user-space expects a virtual address, while a call within the kernel requires a physical address. Attempting to mix address types risks a memory fault and kernel crash.

It is evident the solution to these problems lies beyond the remit of the abstraction layer – dependencies on the nuances of how data are stored and manipulated to suit these method calls exist throughout the store's modules. As a reasonable compromise, a separate implementation was dynamically linked in at compile time for the code segments which were sufficiently different as to make an abstraction layer unworkable.

3.7 Summary

This chapter detailed the implementation work I have undertaken for the *Unbuckle* project. The overall system architecture was discussed with reference to the core state machine and data structures. The development of two distinct network stack modules was described, with particular emphasis on the work carried out to understand the Linux network stack for communicating directly with the machine's NIC for

[2]The pointer is passed by the `*vec` argument in the kernel and as a member of the `*msg` structure in user-space.

performance. The cross-section of layers from textbook networking and the kernel implementation which had to be integrated to form a cohesive system made this module one of the most technically challenging aspects of the implementation. Then, I moved on to discuss the implementation of a slab allocator-like scheme for optimised memory allocation performance.

This work produced a functional end-to-end key-value store, but further work took place to make the code suitable for multi-threading within a kernel context. This included the resolution of complex issues of work distribution and thread synchronisation within the hostile kernel programming environment. The benefits of moving to the kernel were demonstrated by several modifications made to the scheduler. Finally, a user-space port of the project was discussed.

As expected, the implementation of a key-value store in the kernel proved a challenging endeavour, during which I gained substantial knowledge of the kernel's inner workings. The success of the final solution and its performance improvement over the user-space port and pre-existing commercial systems will be reviewed in the following chapter.

Chapter 4

Evaluation

The objective of this chapter is to review *Unbuckle*'s success at meeting the original project success criteria and at delivering benefits over state-of-the-art key-value stores. Hence, I begin by discussing the original project success criteria and how *Unbuckle* meets (and exceeds) them. Then, I briefly discuss the testing strategies applied. An extensive quantitative evaluation to assess the performance impact of various optimisations is carried out. Finally, I compare the user-space and kernel versions of *Unbuckle*, and show that it outperforms contemporary optimised commercial key-value stores. Comparisons against research systems are also made, where I find *Unbuckle* is competitive.

4.1 Overall results

The success criteria of the project, as described in the original proposal (see appendix A.4), have all been met or surpassed. They are briefly summarised below, with pointers to the relevant discussion in this document:

Criterion 1: *The implementation of a [platform-independent] key-value store as a Linux kernel module which is compliant with the memcached protocol, including [...] network protocols [...] and selection of suitable data structures to maximise efficiency.*

This constituted the first phase of the project, which after teaching myself how to program in the kernel, was completed very early in the project to provide a framework for later success criteria to be accomplished. The overall organisation and approach to the project was described in chapter 2, the implementation of suitable interfaces and back-end data structures in §3.1.2 and integration with the kernel's socket interface in §3.2.1.

By using a standard Linux kernel, standard system interfaces and commodity server hardware to build and evaluate the store, the system requires no specialist hardware or software stacks, making it as platform independent as memcached and similar user-space stores.

Criterion 2: *Performing suitable testing [and load simulation] to gather performance data against an in-kernel and userspace instance of the key-value store.*

Testing and load simulation is the primary subject of the forthcoming chapter. A suitable test harness, *memaslap*, was selected in §2.4.1 from a variety of choices given in appendix A.2. The port of the kernel module for user-space operation was introduced in §3.6 along with the abstraction layer specially constructed for this purpose.

Criterion 3: *The introduction of optimisations in a second iteration of the key-value store module [...] to exploit the additional information and control interfaces in the kernel in a bid to further optimise the system's performance.*

A wide array of extensions made to the project with the goal of improving performance have already been discussed in §3. These include optimisations mentioned in the original proposal, but the majority came to light as the project progressed and the available opportunities became apparent:

- **Low-level network interaction** – one of the most complex optimisations was the construction of a network server which integrates with the lowest level of the network stack just after packets are emitted from the network driver. The steps taken to implement this were described in §3.2.2, including a summary of the work undertaken to trace the intricate Linux network stack.
- **Memory pre-allocation** – the costs of allocating memory were amortised over many `SET` requests by introduction of the *bucket allocator*, described in §3.3.
- **Multi-threading** – the implementation of the custom network stack unlocked the ability to spread the internal work across multiple CPU cores to benefit from parallelism. This was discussed in §3.4.
- **Scheduler** – the modifications made with respect to improving the scheduling of key-value store work in the kernel were discussed in §3.5.

4.2 Testing

As described earlier, the challenges of testing kernel code are substantial. Neither formal methods nor user-space test frameworks are directly applicable due to the extremely large space of system and configuration permutations.[1] Developers typically rely on a combination of community feedback and external interface testing; full coverage testing cannot be guaranteed. The complexity of this task is perhaps demonstrated by Yang *et al.*'s work to formally verify file systems [57]. While a substantial effort, this touched but a small subsystem in the kernel codebase.

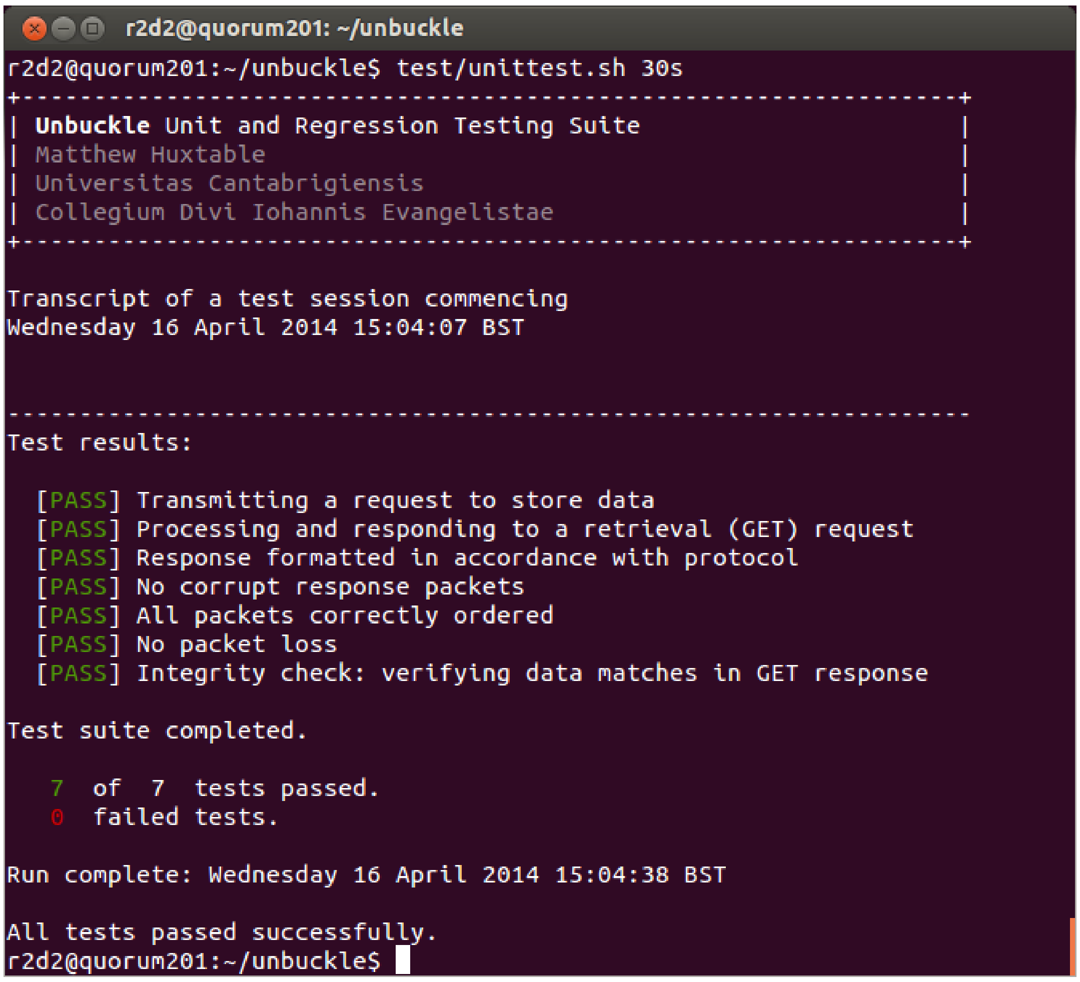
Nevertheless, it was necessary to verify the correct operation of *Unbuckle* with respect to the memcached protocol to ensure data are correctly stored. This was especially

[1]For more information, see *Greg Kroah-Hartman on the Linux Kernel*, a Google Tech Talk with a segment on kernel testing, available at `http://youtu.be/L2SED6sewRw?t=14m29s`.

important as high-risk modular designs were introduced. The complexity of the multi-threading and network server modules, for instance, necessitates independent assurance that they were functioning as specified.

A shell script is used to co-ordinate the testing process, running a sequence of unit tests to verify compliance with the initial specification. Some regression tests are included for major bugs which I had corrected. To avoid the kernel testing difficulties, the *black-box testing* approach is used, whereby tests are written to verify correct output, in this case over the network, in response to well-defined input without knowledge of the internals of the implementation [44, pp. 55–ff.]. The test routines provide a user-definable parameter to control test duration. Shorter tests provide confirmation of correct semantic operation, while longer runs are able to test failure scenarios, such as running out of memory.

An example run of the test suite is demonstrated in figure 4.1.



```
r2d2@quorum201:~/unbuckle$ test/unittest.sh 30s
+-----------------------------------------------------------------+
| Unbuckle Unit and Regression Testing Suite                      |
| Matthew Huxtable                                                |
| Universitas Cantabrigiensis                                     |
| Collegium Divi Iohannis Evangelistae                            |
+-----------------------------------------------------------------+

Transcript of a test session commencing
Wednesday 16 April 2014 15:04:07 BST


-------------------------------------------------------------------
Test results:

  [PASS] Transmitting a request to store data
  [PASS] Processing and responding to a retrieval (GET) request
  [PASS] Response formatted in accordance with protocol
  [PASS] No corrupt response packets
  [PASS] All packets correctly ordered
  [PASS] No packet loss
  [PASS] Integrity check: verifying data matches in GET response

Test suite completed.

   7  of  7  tests passed.
   0  failed tests.

Run complete: Wednesday 16 April 2014 15:04:38 BST

All tests passed successfully.
r2d2@quorum201:~/unbuckle$ 
```

Figure 4.1: *Successful test output from the* Unbuckle *unit and regression testing suite, testing the multi-threaded iteration of the store using the low-level UDP server.*

4.3 Approach to empirical evaluation

In this section, I will briefly outline the high-level approach to analysing *Unbuckle*'s performance that I take for the remainder of this chapter. This evaluation is split into two distinct components:

1. **Internal optimisations** are evaluated to demonstrate their benefits and confirm the investment of engineering effort was worthwhile.
2. **External comparisons** are then made with competing state-of-the-art key-value stores to determine how *Unbuckle* fares in relation to widely deployed systems.

4.3.1 Test environment

The collection of data in these experiments took place on the SRG's high-performance 10 Gbps test network. The specification of the test machines and networking equipment is provided in appendix A.3.1.

Experiments involving the commercial memcached key-value store use release 1.4.17 of this product and are optimised according to recommendations laid out in appendix A.3.3. It is worth noting that an out-of-the-box memcached installation exhibits significantly poorer performance (2–3× lower throughput).

Test harness

Except as otherwise noted in the text, the memaslap load simulation and performance evaluation utility, described in §2.4.1, was used to stress-test each key-value store. This introduced two problems which required resolution before work could proceed:

- memaslap has tunable parameters for varying the simulated workload, including the ability to specify the number of internal worker threads and the size of each batch of requests issued to the system. To select appropriate values, I ran calibration experiments sweeping the parameter space and picked the values at which memaslap performed best.
- The user-space nature of memaslap is problematic. Since the performance metrics are measured for round-trip requests, user-space overheads at the *client* side are part of the result. Consequently, a memaslap client was often unable to saturate the *Unbuckle* server; hence, I made sure to use sufficient concurrent load generators to fully load the server in all experiments.

4.3.2 Data collection methodology

To permit a fair comparison of datasets, each machine was rebooted prior to each test run to ensure no cumulative effects due to processor cache behaviour. The same machines were used to collect data in each experiment, each with a uniform system image and configuration.

This is a tedious and error-prone task which I automated with a set of bash scripts to co-ordinate the machines and automate the reboots. A sample of one of the scripts is provided in appendix A.3.2.

Minor modifications were also made to memaslap to incorporate a raw data reporting module to work around the limited scope of its internal statistics engine. This permitted, in particular, the computation of the percentiles of each run, which were necessary for request latency analysis. The raw output was analysed *in situ* by a set of Python scripts to produce graphs for inclusion in this report.

4.4 Evaluation of internal optimisations

4.4.1 Request pipeline breakdown

To understand the divisions of work in processing a request to the key-value store, I added instrumentation to measure the processor cycles expended in each major stage of the request pipeline. The results are presented in figure 4.2, evaluated against the in-kernel version using the low-level UDP server (§3.2.2).

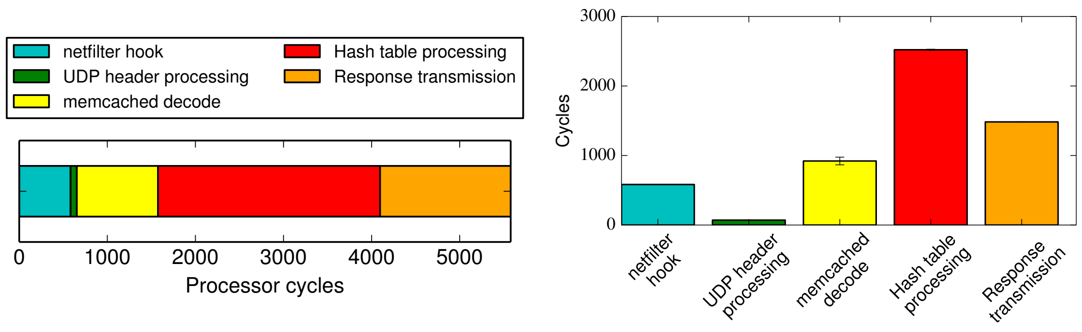


Figure 4.2: A breakdown of the processor cycles consumed by a request in each stage of the Unbuckle *pipeline (per §3.1.1). Error bars on the bar plot show the standard deviation from the mean, but the only errors of significance are in the* memcached decode *stage. Cycle counts were averaged over three runs using a fixed request size of 1024 bytes.*

The total cost of a request is around 5,800 cycles. For comparison, merely sending a UDP datagram from user-space takes around 20,000 cycles. Most stages perform a deterministic amount of work, typically operating on the same quantity of data in the header or request, as demonstrated by the absence of variance. The request processing stage is the most costly, as we would hope for in a well-tuned system.

While the 2,500 cycles spent processing the request may afford opportunities for further optimisation, it is worth noting that this only corresponds to approximately 1μs of "wall clock" time.

Header decoding

Substantial differences between the decoding of the UDP header and parsing of the memcached header can be observed. The former is binary, whereas the latter is an ASCII header. Binary headers adhere to a fixed layout in memory, which permits immediate decode to extract data – indeed, the UDP decode constitutes just 1% of the overall processor time. Unfortunately, string processing for parsing the ASCII header produces many data-dependent branches, which do not interact well with speculative processors.[2]

4.4.2 Network server

As described previously in §3.2, *Unbuckle* supports two distinct approaches to network connectivity when operating in the kernel: the socket interface at the top of the network stack, and low-level interaction in the IP protocol processing layer.

The low-level optimisation was one of the most time consuming to implement correctly, but it improves performance and opens possibilities for other optimisations, such as multi-threading.

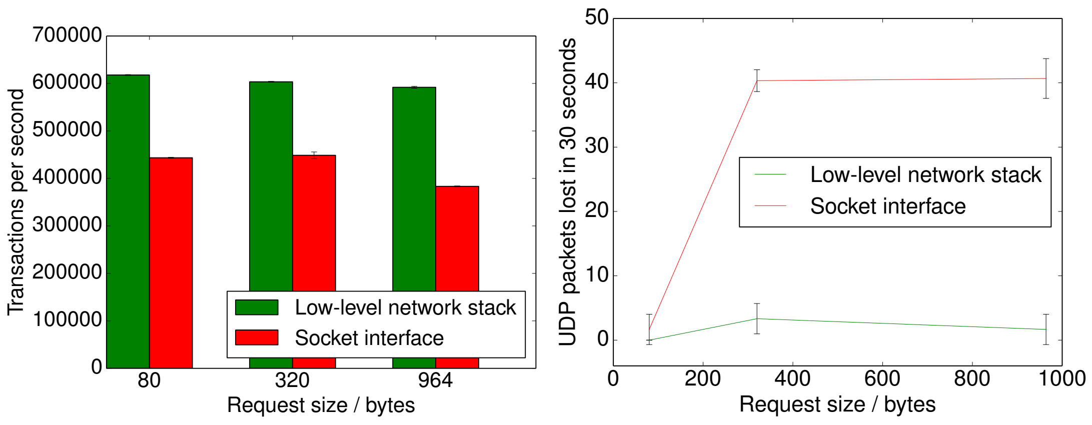
Figure 4.3 shows the transactions processed per second for a single threaded version of the key-value store with each of the two network servers. A range of small, medium and large request sizes was used corresponding to figures from Facebook's memcached infrastructure [41]; these data points will appear in several experiments throughout this chapter.

Transaction throughput

As demonstrated in figure 4.3a, the custom low-level network stack achieves approximately a 50% gain over the socket interface for large requests.

I should note this figure may well be an underestimate of the network server's full potential. We would expect the throughput in transactions per second to reduce with increasing request size due to the overheads in memory copy operations for larger requests. It is likely that the capacity of a single thread of execution is exhausted before the network server's processing capability, producing the effect shown. I investigate the effects of multi-threading the key-value store in a later section and find a 4× increase in throughput (§4.5.3). I cannot fairly compare a multi-threaded

[2]memcached does have support for a binary protocol variant. However, the ASCII protocol is most widely used, and load testing tools do not currently support the combination of UDP and the binary protocol.



(a) Transactions per second achieved by each network server.

(b) UDP packets lost, due to timeout or corruption, measured during each run for each network server.

Figure 4.3: Comparison of the two approaches for network connectivity in Unbuckle*: the in-kernel socket interface and the custom low-level UDP server.*

version of the network server here because the socket interface is not itself multi-threaded.

Packet loss

The rate of UDP packet loss[3] for each network server is given in figure 4.3b. UDP, by its nature as a connectionless protocol, is susceptible to some intrinsic loss of packets due to queueing delay in switch buffers and corruption in memory copy operations.

The low-level network server achieves a much lower rate of loss across the spectrum of request sizes than the socket interface, despite identical networking infrastructure, load and higher packet throughput. This suggests a high rate of contention in the socket interface's packet queue (see the top of the network stack diagram in figure 3.4), whereas loss in the network itself is not of significance. The queue rejects packets at a rate of more than one per second for medium and large request sizes, while the custom network server loses no more than five packets across an entire simulation run.

An empirical evaluation of this effect in terms of the queue blocking probability might be possible [30, pp. 269–ff.], but would require extensive modifications to the kernel to add the necessary instrumentation and is beyond the scope of this project.

[3]For the purposes of the ensuing discussion, "loss" subsumes any event which caused a UDP packet to fail to reach its destination successfully, including a packet dropped in a queue, a packet dropped due to checksum failure/corruption, and other similar events.

4.4.3 Bucket allocator

The concept of memory pre-allocation was introduced in §3.3. This optimisation seeks to amortise the cost of memory allocation in `SET` request processing over many requests.

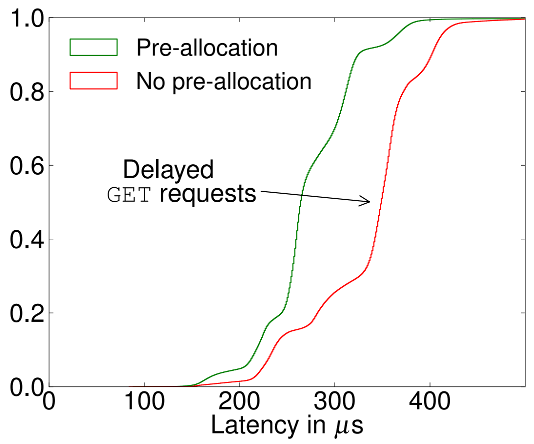


Figure 4.4: A CDF to compare the latency of request processing in Unbuckle *before and after the introduction of the bucket allocator for memory pre-allocation.*

The latency profile for the key-value store with and without the memory pre-allocation optimisation is shown in figure 4.4. Improvements can be seen across the whole latency distribution, with a 100μs improvement at the median.

An interesting side-effect of memory pre-allocation is an indirect improvement for the performance of `GET` requests in addition to `SET` requests. This effect is observed above the 20th percentile latency, where the two plots diverge. For the key-value store without memory pre-allocation, `GET` requests experience additional queuing latency while the request pipeline is blocked on the memory allocator in an earlier `SET` request.

4.5 Comparative evaluation

In this section, I will perform a thorough analysis of *Unbuckle*'s performance to demonstrate the success of the project in comparison to existing commercial systems. In particular, I will confirm the hypothesis I sought to investigate in §1.1 and §A.4, namely:

Hypothesis:
"it might be possible to extract further performance gains from a key-value store by... building the code directly within the system kernel... where [no] system calls are necessary, as the [key-value store] already has the highest level of privilege, including direct access to... the file system, physical memory and network..."

Prior to commencing data collection, it was necessary to identify the comparisons to perform and the results I wished to analyse. The various possibilities are shown in figure 4.5 and elaborated in the following text.

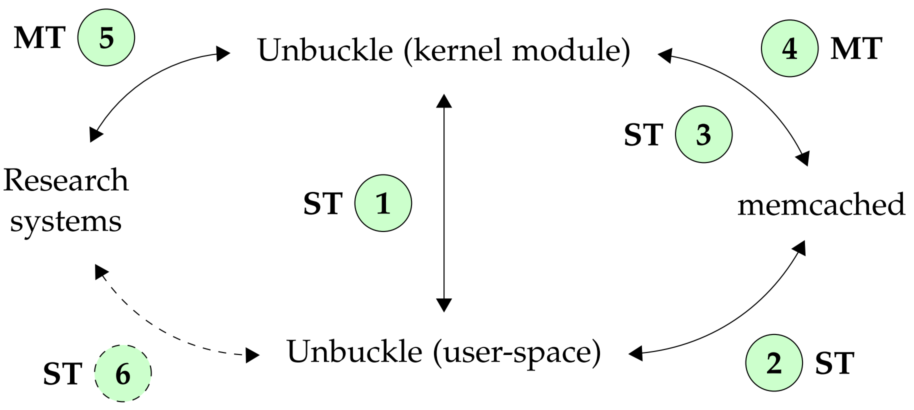


Figure 4.5: Comparing Unbuckle *with existing commercial and research systems.* ***ST*** *refers to a single threaded comparison;* ***MT*** *to a multi-threaded comparison.*

1. ***Unbuckle* (user-space) *vs.* *Unbuckle* (kernel) (single threaded[4])**
   Analyses the overhead of the system call interface in running a network I/O-bound software system independent of any other modifications, as the core algorithms in both versions of *Unbuckle* are identical, save for the necessary changes to interact with different external APIs (per §3.6).

2. ***Unbuckle* (user-space) *vs.* memcached[5] (single threaded)**
   Evaluates the performance of *Unbuckle*'s algorithms and data structures against an equivalent state-of-the-art user-space system, independent of the system context. This provides a baseline for the comparison when *Unbuckle* is in the kernel.

3. ***Unbuckle* (kernel) *vs.* memcached (single threaded)**
   Follows from 2 to determine the performance speedup of *Unbuckle* in its native context against an existing state-of-the-art product in the absence of any concurrency.

4. ***Unbuckle* (kernel) *vs.* memcached (multi-threaded)**
   Compares the performance of *Unbuckle* against an optimised, concurrent, widely-deployed user-space key-value store. Determines the prospect of performance gains in web infrastructure by replacing memcached with

[4] *Unbuckle* in user-space is only provided single threaded. This is due to the infeasibility of porting kernel threads to user-space, as described in §3.6. In any case, a multi-threaded version would tend towards the performance of memcached, as both systems incur similar performance overheads in the system call interface and network stack.

[5] memcached is a user-space only product. Hence, no qualification is given as to user or kernel context.

Unbuckle.

5. ***Unbuckle*** **(kernel)** ***vs.*** **optimised research systems (multi-threaded)**
   Provides supporting data to determine how *Unbuckle* fares against recently published research systems, which have been specifically optimised for performance.

6. ***Unbuckle*** **(user-space)** ***vs.*** **optimised research systems (single threaded)**
   This comparison, while possible, is pointless; the analysis would not be original and would simply re-produce the results of other papers, such as those in the MICA paper [35].

4.5.1 The system call overhead

One of the core tenets of the project is the assertion that the system call interface of traditional operating systems has a negative effect on data centre software whose predominant task is to communicate on a network. The versatility of the *Unbuckle* platform allows this hypothesis to be evaluated.

An experiment was performed against the same code compiled in both a kernel and user-space context. Service latency as a function of request size was measured. In the kernel, I used the socket-based network server to ensure the tests only varied in whether they had to perform system call interface crossings for privileged system tasks.

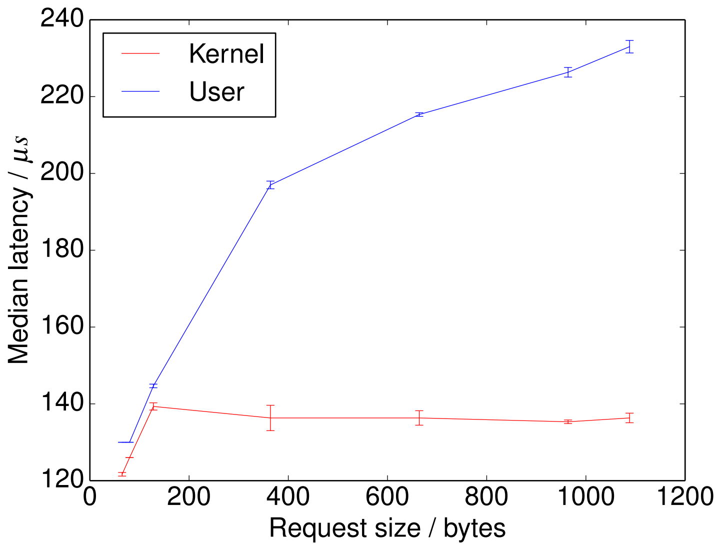


Figure 4.6: A demonstration of the overheads imposed by the system call interface on the latency of packets as a function of request size. Plotted values are the mean of the median latency from five test runs. Error bars show the standard deviation of this quantity. The key size was fixed at 64 bytes to ensure the hash function overhead remained constant as request size changed.

Figure 4.6 makes the system call overhead evident. The user-space store performs worse than the in-kernel one for all request sizes and the gap widens as request size increases. This behaviour is caused by the confluence of two time-sensitive operations and merits some further explanation.

The test environment and network stack of the client machines impose an intrinsic lower bound on the latency. Use of the socket interface in both tests meant all packets reaching the store were required to traverse the full extent of the network stack illustrated in figure 3.4. This leads to the baseline latency of just over 100μs for both versions. However, there is additional overhead to the user-space version.

Crossing the system call interface

As previously described, interacting with network hardware in user-space requires interrupting the processor to invoke the kernel. This overhead dominates on the lower end of the request size spectrum, where user-space incurs a 10μs overhead. This corresponds to an average of two system calls per request – once upon ingress and again upon egress – but a fraction of requests will make more system calls for tasks such as memory allocation in the bucket allocator. These data corroborate the SRG's earlier findings of a 5μs overhead per system call.

Memory copy operations

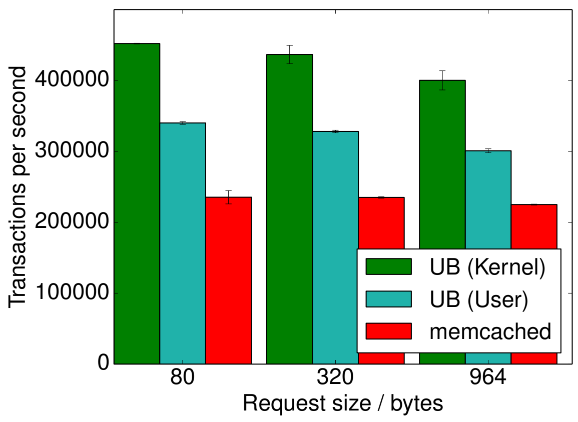
In user-space, additional memory copy operations are required between memory buffers when a system call is made to access the network. This overhead is not necessary in the kernel, because the kernel has direct access to the `sk_buff` data structures which represent network packets. Such memory copies are linear in the size of the data to be copied, making the overheads of user-space increase with larger requests. This effect appears to dominate beyond request sizes of 150 bytes.

Meanwhile, the kernel store does not need to perform such memory copy operations, for it already has access to read the `sk_buff`'s contents in kernel-space physical memory when a socket interface operation is invoked. This explains the constant request latency of 140μs in the kernel version – once network driver and network stack overhead is accounted for, the only remaining work for the kernel is the processing of the request, which is independent of request size.[6]

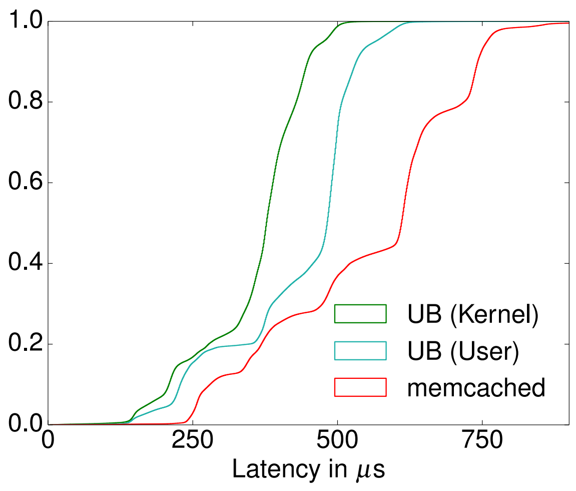
4.5.2 Comparison with memcached

In order to ascertain the performance differential between *Unbuckle* and a state-of-the-art competitor, I performed several experiments to compare the project in various guises against memcached. As the user-space version of *Unbuckle* is not

[6]Of course, the copy cost still exists on the load test client machine, but memaslap's batched receiving amortises this over many requests.



(a) *Transactions per second achieved across the range of small, medium and large request sizes.*



(b) *A CDF comparing the latency distribution. Request size was 320 bytes, corresponding to the "medium"-sized request above and the median in the Facebook paper [41].*

Figure 4.7: A comparison of Unbuckle *with memcached on various parameters.*

multi-threaded, I used the single threaded, socket interface versions (both user-space and kernel varieties) to facilitate a like-for-like comparison. Later experiments will demonstrate the full extent of scalability when the network server and multi-threading in the kernel are taken into account.

Figure 4.7 presents the results of the comparison of the systems, in terms of the mean transaction rate per second (figure 4.7a) and the cumulative distribution of request service latency (figure 4.7b). Once again, the request sizes at which the systems were evaluated were drawn from the observations made in a commercial deployment at Facebook [41].

It is apparent from the results that even before extensive optimisation, *Unbuckle* is successful in two ways:

- ✓ **Improved user-space performance** is observed, even before the optimisation of moving into the kernel is accommodated.
- ✓ **Further improved kernel performance** is delivered, which demonstrates benefits to moving into the kernel context.

Moreover, an increase in the request size leads to reduced throughput, owing to the increased overheads in network transmission and memory copy operations. While this trend is observable in *Unbuckle*, the throughput of memcached does not change between experiments, demonstrating it is not particularly efficient by comparison.

User-space comparison

I will first consider the performance differences observed when operating wholly in user-space. In this context, *Unbuckle* performs significantly better than memcached for all metrics. For small data sizes, *Unbuckle* achieves approximately a 40% throughput increase and a 25% reduction in latency at the median. This is an impressive result for a store which was originally intended for operation in the kernel. The port to user-space was made with no performance optimisations in mind, demonstrating that the robustness of the code extends into other environments.

By consideration of the latency CDF in figure 4.7b, it is clear that at the first percentile, the performance of the two versions of *Unbuckle* is comparable. This demonstrates similar overheads due to common code paths being executed, processor cache lookups and systematic overheads in network communication. Meanwhile, memcached's first percentile latency is 100μs greater. This is likely because memcached is intended for multi-threaded operation and performs synchronisation even when run with just one thread.

Kernel comparison

The key goal of the project was to investigate kernel overheads inflicted on most data-intensive, network-bound applications. The user-space code evaluated in figure 4.7 is functionally equivalent to the kernel code for the purposes of this test. Both versions use the socket interface and are single threaded.

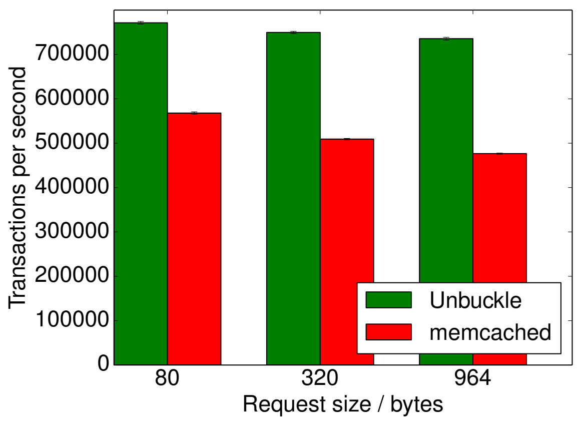
Although the two versions of *Unbuckle* were investigated earlier (in §4.5.1), it is instructive to study the results of the kernel version against memcached to appreciate the magnitude of the gains achieved in the kernel context.

The improvements in request throughput are readily visible across the request size spectrum, as is the change to the latency distribution. However, for small requests, the reduction in system call overhead delivers the greatest improvement: 90% higher throughput in *Unbuckle* (*vs.* memcached). This is a highly significant result; the skewed request size distribution places emphasis on support for small request sizes, which would make *Unbuckle* very competitive in a real-world setting.

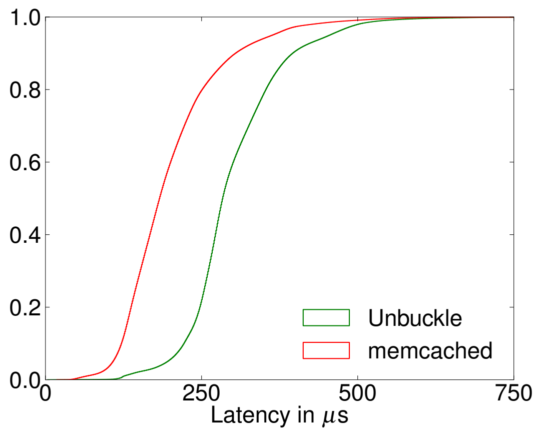
4.5.3 Two highly optimised key-value stores

To determine the true extent of *Unbuckle*'s performance, I will now compare it against memcached in full operation. Both key-value stores are multi-threaded for the purposes of the following tests, and all optimisations previously described in this dissertation are enabled in *Unbuckle*.

The results are presented in figure 4.8. Once again, *Unbuckle* comes top on all metrics. Its latency distribution is particularly attractive in comparison to memcached, saving over 100μs at the median latency. In terms of throughput, *Unbuckle* is consistently



(a) *Transactions per second for the optimised key-value stores in terms of small, medium and large requests.*



(b) *Another CDF comparing the latency distribution for the median request size of 320 bytes, for the multi-threaded key-value stores.*

Figure 4.8: Performance of multi-threaded Unbuckle *in comparison to memcached.*

better, achieving up to 50% higher throughput than memcached at large request sizes.

In comparison to figure 4.7, it becomes clear that memcached scales somewhat when deployed in multi-threaded mode. However, its internal overheads and the system call interface's inherent latency overhead prevent it from utilising all the available processor cores to their full extent, as demonstrated by *Unbuckle*'s superior performance on the same hardware.

4.5.4 *Unbuckle vs.* optimised research systems

I compared *Unbuckle* with the state-of-the-art research systems in this field, including Masstree [39], MICA [35] and RAMCloud [42]. For comparison purposes, I evaluated the requests processed per second by *Unbuckle* using the settings provided in the MICA paper. Small requests consisted of 16 byte keys and 64 byte values. Large requests used 128 byte keys and 1024 byte values. In addition, the experiments were run on a server system comparable to the one used in the MICA paper.

	Requests per second		**Code size**	**Commodity**
System	**Small requests**	**Large requests**	**(lines)**	**hardware**
RAMCloud	2.3M	3.1M	96k	✗[7]
Masstree	4.9M	10.8M	25k	✓
MICA	9.4M	64.4M	8.7k[8]	✗
Unbuckle	1.9M	2.5M	3.3k[9]	✓

Table 4.1: Indicative performance comparison of Unbuckle *with research systems.*

Of course, any comparison of these systems is only indicative, as they all use different approaches and perform subtly different tasks. Masstree, for instance, is a persistent data store and uses a specialist, lock-free data structure, whereas *Unbuckle* is constrained by the kernel's built-in hash table being unoptimised for concurrency. RAMCloud supports replication and contains logic for fast recovery from machine failures, neither of which is supported by *Unbuckle* or memcached. Finally, MICA requires specific system hardware to provide a user-space network stack, as previously discussed in §1.3.2. By contrast, *Unbuckle* does not have such hardware constraints.

Table 4.1 shows *Unbuckle* compares favourably to other research systems. Its throughput approaches that of the more complex RAMCloud system, while continuing to support commodity hardware. Masstree achieves a 2.5× higher throughput for small requests. However, it is considerably more complex, and its data structures have been specially optimised for cache affinity and multicore scalability.

MICA makes extensive use of hardware optimisations present in Intel NICs, motherboards and CPUs while also carefully fine-tuning the back-end data structures. The MICA paper was published close to the completion of this project. In the future, I hope to apply some of the suggested optimisations to *Unbuckle*. It is worth pointing out, however, that MICA only works on an end-to-end Intel stack, while I have evaluated *Unbuckle* on a mixture of AMD and Intel systems, and preserve compatibility with any NIC.

4.6 Scalability analysis

To determine how well *Unbuckle* scales, I ran experiments in a more elaborate test environment. The test machine used for this analysis uses a modern six-core Ivy Bridge-EP processor and several 10 Gbps NICs. Six independent load test clients were used to concurrently generate load against the key-value store.

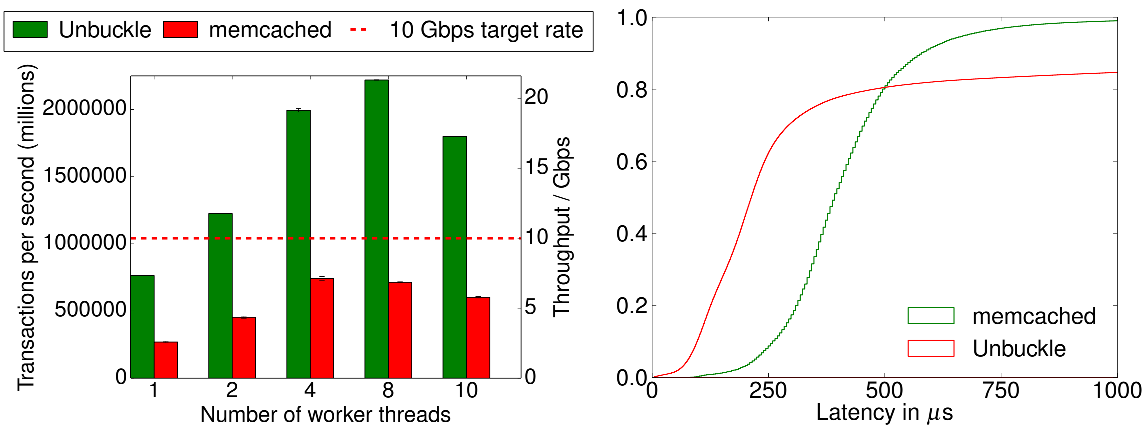
Figure 4.9 shows how *Unbuckle* scales as the number of worker threads increases. Using eight worker threads, it achieves in excess of 2 million transactions per second, corresponding to a 22 Gbps throughput. This result more than twice exceeds the original goal of operating at 10 Gbps, marked by the dashed line in figure 4.9a. memcached, the competing system, does not scale beyond 8 Gbps while simultaneously delivering a far worse latency profile.

The drop in performance for ten worker threads is consistent and can be explained by the experimental setup. The machine used for this test has six physical processor

[7] RAMCloud was originally designed for special-purpose Infiniband networks, although it has since been shown to function on Ethernet [35].

[8] Only the kernel portion of the *Unbuckle* source code is counted.

[9] This figure is an underestimate, due to MICA's dependence on the Intel Data Plane Development Kit (DPDK) to provide a user-space network stack. The DPDK is 1.3 million lines of code.



(a) System throughput performance *(b) A latency CDF for the two systems*

Figure 4.9: An evaluation of Unbuckle *running at scale to demonstrate the extent to which the original aim of the project is exceeded. This key size used in this experiment was 64 bytes. Values were 1024 bytes.*

cores, each of which has two logical threads. An increase in thread-level parallelism pays off for 8 worker threads, but increasing this further decreases performance due to increased scheduling contention for interrupts from the NICs.

Figure 4.9b shows that *Unbuckle*'s latency profile exhibits a long tail at the 80th percentile. This long tail (not shown) runs to a 99th percentile of 4ms. This is an expected side-effect of the key-value store's increased throughput which fills some of the 10 Gbps links to capacity. Consequently, some packets spend time waiting in switch queues and NIC buffers. As memcached cannot fill any of the links to capacity, it does not suffer from this contention.

4.7 Summary

Through a variety of experiments, the original hypothesis I set out to investigate has been confirmed: moving code into the system kernel *does* produce observable performance gains.

However, naïvely bypassing the system call interface only partially contributes to the reported improvement. The remaining gains are obtained by the investment of considerable effort to harness additional opportunities presented in the kernel; for example, the specialisation of commonly traversed code paths to reduce dependence on overly general system interfaces. A variety of such optimisations were shown to contribute noticeably, especially in the network stack.

The *Unbuckle* key-value store outperforms as state-of-the-art system, memcached, in all experiments. The latency of requests to *Unbuckle*, even when placed under significant load, remains below that of memcached. Sound software design and engineering practice is demonstrated by the framework's impressively scalability. It continues to outperform its original design remit in user-space.

Chapter 5

Conclusions

This dissertation has described the design, implementation and evaluation of the *Unbuckle* in-kernel, high-performance key-value store. To my knowledge, *Unbuckle* is the first system of this kind. The system delivered outperforms a state-of-the-art competitor by up to a 3× margin, while retaining hardware independence. More generally, the evidence presented provides further support for the hypothesis that traditional operating system design principles limit performance in modern data centre environments.

5.1 Achievements

I have surpassed the initial project requirements by implementing a number of challenging optimisations, which collectively provided demonstrable performance gains over existing systems in commercial use. These are delivered in a platform independent design using commodity x86 server hardware, despite recent published work claiming this feat was impossible (per §1.3.1).

Personally, I have enjoyed the opportunity to enhance my knowledge of operating system design. I undertook this project to understand further how the operating systems principles I have studied are applied in practice. I hope to do further research at this intersection of distributed systems, networking and operating system design, a pursuit in which the knowledge I have acquired on this project will be indispensable.

Finally, I am encouraged by the potential for immediate real-world impact this project possesses. It could readily replace existing memcached clusters in the core of the modern web to deliver performance improvements using existing infrastructure. While this has the potential to transform the use of key-value stores, it also provides practical confirmation that further research into operating system principles would be extremely beneficial for these data centre systems.

5.2 Lessons learnt

The process of transforming an initial idea into a fully fledged key-value store was extremely enjoyable. However, I was perhaps naïve as to how time consuming working with kernel code could be. Its complexity means many hours are required to understand a seemingly simple subsystem, demanding a rigorous application of operating system theory rather than the mere investment of engineering effort. This issue is only compounded by the rapid pace of kernel development and the difficulty of debugging when a bug will typically hard-crash the machine.

5.3 Further work

The *Unbuckle* key-value store is complete in the sense that it satisfies the initial project specification. Nevertheless, there is scope for further implementation and investigative work to evaluate the impact of operating systems on data centre workloads.

I am planning to refine the implementation and hopefully adapt this dissertation for publication. I will also release the source code free-of-charge on the internet. The project will be licensed under the GNU General Public License (GPL) in accordance with the licensing requirements of the Linux kernel.[1]

Further possible improvements include:

- **Reliable delivery:** some memcached users may wish to offset performance against reliable delivery guarantees. The canonical protocol for this purpose is TCP, but the complexity of its state machine would make a custom TCP server a project in its own right. However, there is potential to layer the Stream Control Transmission Protocol (SCTP) atop this project's UDP server to obtain similar semantics [53].
- **multi-`GET` support:** the memcached protocol permits the batching of `GET` requests to reduce protocol overhead in large systems, where hundreds of keys may be retrieved to service one user request.
- **Port Masstree to the kernel:** this lock-free data structure delivers performance gains in user-space by careful interoperation with processor caches. The beta release is written in C++, so a kernel port in C would be a non-trivial undertaking. However, further performance improvements are possible.
- **Q-Jump integration:** the SRG's Q-Jump project, formerly the *Resilient Realtime Data Distributor* (R2D2), provides bufferless, bounded latency networking [23]. Integration with this framework would allow *Unbuckle* to offer a tight performance guarantee.

[1]See `http://www.tldp.org/HOWTO/Module-HOWTO/copyright.html`.

Bibliography

[1] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference* (1967), ACM, pp. 483–485.

[2] Aniszczyk, C. Caching with Twemcache. https://blog.twitter.com/2012/caching-twemcache, 2012.

[3] Arista Networks. 7150 series data sheet. http://www.aristanetworks.com/media/system/pdf/Datasheets/7150S_Datasheet.pdf, 2012.

[4] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), ACM, pp. 53–64.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), ACM, pp. 164–177.

[6] Benvenuti, C. *Understanding Linux Network Internals*. O'Reilly Media, 2006.

[7] Boehm, B. W. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes 11*, 4 (Aug. 1986), 14–24.

[8] Bonwick, J. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference* (1994), USENIX Association, pp. 6–6.

[9] Burbeck, S. Applications programming in Smalltalk-80: How to use Model-View-Controller (MVC). http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html, 1992.

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. BigTable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

[11] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (June 1970), 377–387.

[12] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud Computing* (2010), ACM, pp. 143–154.

[13] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[14] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (2007), ACM, pp. 205–220.

[15] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (2014), USENIX, pp. 401–414.

[16] Edwards, A., and Muir, S. Experiences implementing a high performance TCP in user-space. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (1995), ACM, pp. 196–205.

[17] Ely, D., Savage, S., and Wetherall, D. Alpine: A user-level infrastructure for network protocol development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (2001), USENIX Association, pp. 15–15.

[18] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), ACM, pp. 251–266.

[19] Fan, B., Andersen, D. G., and Kaminsky, M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (Apr. 2013), USENIX Association, pp. 371–384.

[20] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004).

[21] Gandhi, R., Gupta, A., Povzner, A., Belluomini, W., and Kaldewey, T. Mercury: bringing efficiency to key-value stores. In *Proceedings of the 6th International Systems and Storage Conference* (2013), ACM, pp. 6:1–6:6.

[22] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Symposium on Operating System Design & Implementation* (2000), USENIX Association, pp. 22–22.

[23] Grosvenor, M. P., Schwarzkopf, M., and Moore, A. W. R2D2: Bufferless, switchless data center networks using commodity Ethernet hardware. In *Proceedings of the 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2013), ACM, pp. 507–508. Poster session.

[24] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach*, 5th ed. Elsevier, 2012.

[25] Istvan, Z., Alonso, G., Blott, M., and Vissers, K. A flexible hash table design for 10 Gbps key-value stores on FPGAs. In *The 23rd International Conference on Field Programmable Logic and Applications* (Sept 2013), pp. 1–8.

[26] Iyengar, A., Challenger, J., Dias, D., and Dantzig, P. High performance web site design techniques. *Internet Computing, IEEE 4*, 2 (Mar 2000), 17–26.

[27] Jacobson, V., and Felderman, B. Speeding up networking. In *Ottawa Linux Symposium (July 2006)* (2006).

[28] Joubert, P., King, R. B., Neves, R., Russinovich, M., and Tracey, J. M. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference (General Track)* (2002), USENIX Association, pp. 175–187.

[29] Kant, K. TCP offload performance for front-end servers. In *IEEE Global Telecommunications Conference* (2003), vol. 6, pp. 3242–3247 vol.6.

[30] Keshav, S. *Mathematical Foundations of Computer Networking*. Addison-Wesley Professional, 2012.

[31] Krieger, M. Storing hundreds of millions of simple key-value pairs in Redis. `http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value-pairs`, 2012.

[32] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[33] Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th European Conference on Computer Systems* (2014), ACM, pp. 27:1–27:14.

[34] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[35] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (2014), USENIX, pp. 429–444.

[36] Love, R. *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[37] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ACM, pp. 461–472.

[38] Madhavapeddy, A., Mortier, R., Sohan, R., Gazagnaire, T., Hand, S., Deegan, T., McAuley, D., and Crowcroft, J. Turning down the LAMP: Software specialisation for the cloud. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (2010), USENIX Association, pp. 11–11.

[39] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), ACM, pp. 183–196.

[40] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference* (2013), USENIX Association, pp. 103–114.

[41] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX, pp. 385–398.

[42] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review 43*, 4 (Jan. 2010), 92–105.

[43] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144.

[44] Patton, R. *Software Testing*, 2nd ed. Sams Publishing, 2005.

[45] Royce, W. W. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering* (1987), IEEE Computer Society Press, pp. 328–338.

[46] Schwarzkopf, M. Personal communication, 2013.

[47] Schwarzkopf, M., Grosvenor, M. P., and Hand, S. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (2013), ACM, pp. 9:1–9:7.

[48] Silberschatz, A., Korth, H. F., and Sudarshan, S. *Database System Concepts*, 6th ed. McGraw-Hill, 2011.

[49] SolarFlare Communications. Filling the pipe: A guide to optimising memcache performance on SolarFlare hardware. Tech. Rep. SF-110694-TC, SolarFlare Communications, August 2013.

[50] Stevens, W. R., Fenner, B., and Rudoff, A. M. *UNIX Network Programming: the sockets networking API*, 3rd ed., vol. 1. Addison-Wesley Professional Computing, 2003.

[51] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001), ACM, pp. 149–160.

[52] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. Serving large-scale batch computed data with Project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), USENIX Association, pp. 18–18.

[53] Tuexen, M., and Stewart, R. UDP encapsulation of Stream Control Transmission Protocol (SCTP) packets for end-host to end-host communication. RFC 6951 (Proposed Standard), May 2013.

[54] Vogels, W. Eventually consistent. *Communications of the ACM 52*, 1 (Jan. 2009), 40–44.

[55] Wilcox, M. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Proceedings of the 4th Australian Linux conference* (Jan. 2003).

[56] Wilkes, M. V. The memory gap and the future of high performance memories. *SIGARCH Computer Architecture News 29*, 1 (Mar. 2001), 2–7.

[57] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (Nov. 2006), 393–423.

Appendix A

Appendices

A.1 Kernel privilege separation on x86 systems

The most widely supported instruction set architecture (ISA) in modern data centre environments is the x86, upon which most modern desktop processors are based. As a result of its ubiquity, the ensuing discussion is specialised to this ISA.

The separation of kernel and user-space is realised in hardware on an x86-compliant processor by implementation of four protection rings. These rings are arranged hierarchically in order of decreasing trust of code executing therein. The x86 protection rings are depicted in figure A.1.

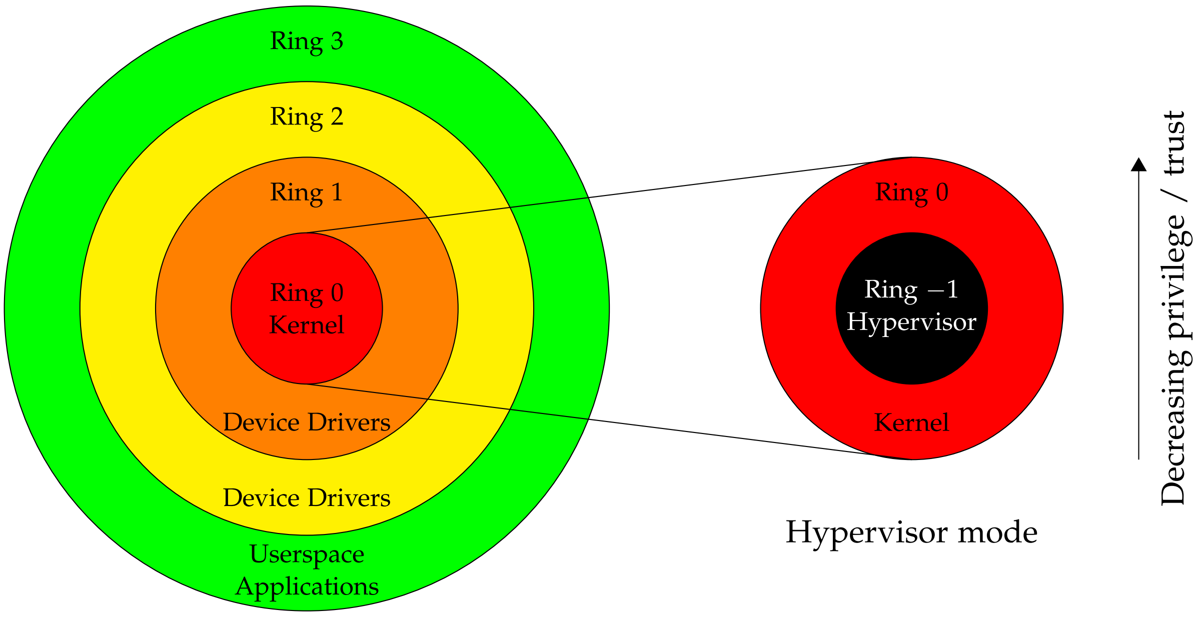


Figure A.1: Protection rings of a modern x86 processor.

Ring 0 represents the most privileged kernel mode with unrestricted access to and control of the system. Each ring layers over the interface provided by the previous ring. Rings with lower trust are restricted in their ability to execute privileged instructions or write to critical memory regions. Certain rings may also enforce a virtual memory abstraction, to prevent applications reading each others' memory segments. Most user-space code executes in the lowest ring, ring 3, which is characterised by the requirement that faulty code should never have the ability to cause system instability or the failure of other running applications.

The configuration of the protection rings takes place when the kernel first gains control of the machine at system boot. A set of system descriptor tables is configured at a well-known memory location such that the kernel is free to dynamically re-configure the level of protection enforced by the processor.

Further discussion of protection rings is available in the literature [24, pp. B-50–ff.].

Hypervisor-based virtualisation

Bare-metal hypervisors, such as Xen [5], have recently become popular in data centre design by permitting underutilised machines to execute multiple guest operating systems without compromising on system performance or security.

Running multiple operating systems under a single host machine traditionally required a software-based emulation platform to intercept issued processor instructions from guest operating systems, sanity check them and pass to the processor for execution. This interface was a bottleneck and impractical for server-based workloads.

To support hardware-assisted virtualisation, x86 processors are enriched with new hardware primitives to permit guest virtual machines to safely share system resources. In particular, the protection rings are augmented with a new ring -1, below ring 0 in the hierarchy, as illustrated in the inset in figure A.1. In this design, existing operating systems are provided with direct access to program the hardware as if they were executing physically, including permitting a guest operating system's kernel to execute natively within ring 0. System functions shared between virtual machines are implemented by the hypervisor, which executes with unrestricted access to system hardware in the new ring -1.

A.2 Evaluation of load simulators

Section 2.4.1 describes the approach followed for performing testing and evaluation of the key-value store. In order for this to take place, it was necessary to select a suitable test harness for simulating load and collecting statistics of the store's performance.

Test Harness	**Evaluation**	
Yahoo! Cloud Serving Benchmark	Not a test harness *per se*, but rather a widely used generic package of real-world workload simulations against which key-value store performance can be evaluated [12]. Unfortunately, the only implementation for memcached utilises spymemcached,[1] which has no UDP protocol support.	✗
memaslap[2]	Uniquely among memcached test harnesses, memaslap supports the memcached UDP protocol. The number of worker threads and the batch sizes to be simulated can be varied, so the simulation can be tuned to approximate a real-world setup with multiple application servers. It is capable of 10 Gbps operation.	✓
memtier benchmark[3]	This is one of the best test harnesses available which offers substantial support for tuning the request sizes and distributions. Unfortunately, it also has no support for the UDP protocol.	✗

Table A.1: An evaluation of memcached-compliant test harnesses and their suitability for performance analysis.

Several options were evaluated, as shown in table A.1. *memaslap* was deemed the only suitable system which aligned with the requirements of the project, and was thus selected as the harness for the remainder of this work.

[1] `https://code.google.com/p/spymemcached/`

[2] Component of libmemcached, available at `http://www.libmemcached.org`

[3] `http://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached`

A.3 Test machines

A.3.1 Specification

In this section, I give the specification of the test machines and networking equipment used for the purposes of evaluating the performance of *Unbuckle* at scale.

Test Machines

The primary test machines followed a uniform specification, as follows:

- Intel® Xeon® E5-2643 processors:
  - A chip multiprocessor with four processor cores, each running at 3.30 GHz.
  - Capable of hyperthreading (Intel's name for simultaneous multi-threading), but this was disabled for performance reasons in line with other research work in this field.
- 128 GB of system memory, all of which is accessible at equal cost from all processor cores (there is only a single NUMA node).
- Linux kernel version 3.10.2 on the Ubuntu 12.04 *Precise Pangolin* operating system.

In order to assess the maximum scalability of *Unbuckle* as-is, it was necessary to use machines with a larger number of processor cores to spawn more worker threads:

- Intel® Xeon® E5-2643 v2 processors, with six hyperthreading-enabled cores running at 3.50 GHz.
- 64 GB of system memory.
- Linux kernel version 3.10.2 on the Ubuntu 13.10 *Saucy Salamander* operating system.

Network Connectivity

To interconnect the test machines together, all machines were connected via 10 Gbps optic connections to a fast cut-through 10 Gbps switch from Arista Networks [3]. This switch is specifically designed for data centre environments; unlike most switches, which could introduce significant packet delay, this switch has a best-case switching latency below 800 ns.

A.3.2 Automated data collection scripts

A set of custom scripts were implemented to co-ordinate the process of collecting and analysing the large datasets from each simulation run. The scripts used to compare the performance of *Unbuckle* between user-space and the kernel implementation, to determine the overhead of the system call interface on a network-intensive application, are presented here. This corresponds to evaluation 1 in figure 4.5.

A variation of these scripts was used for the other analyses, with appropriate changes to the simulation configuration to exercise the parameters under test and collect the required data.

Simulation

This simulation analysed a large parameter space of seven different request sizes. Each such request size was tested against the key-value store in its kernel and user-space contexts.

The script used for this task is given in listing A.1. This runs on the machine which performs the simulation. The worker machine is automatically configured over the network via SSH to allow a completely hands-off simulation, even handling reboots and the subsequent configuration to prepare for another run. The total simulation time was 4 hours and 23 minutes, producing a total of 21 GB of output for analysis.

```bash
#!/bin/bash

eval `ssh-agent -s`
ssh-add

# Manage the reboot process of the quorum206 machine,
# and automatically re-configure without manual intervention.
function q206_reboot {
  ssh 128.232.33.16 "sudo␣reboot"
  while ! ping -c1 128.232.33.16 &>/dev/null; do :; done
  ssh 128.232.33.16 \
    "~/setup_r2d2.sh␣&&␣sudo␣~/ub_opt.sh"
}

function run_kernel_test {
  ssh 128.232.33.16 "~/unbuckle/setup_kernel.sh"
  ~/libmemcached-1.0.15/clients/memaslap -s 10.10.0.3:11211 \
    --udp -t 30s -T ${4} -c ${5} -S 15s > \
    ~/ubdata/userspace_data/key-${1}_value-${2}_krnl-${3}.dat
  ssh 128.232.33.16 "~/unbuckle/setup_rmkernel.sh"
}

function run_user_test {
  ssh -f 128.232.33.16 "~/unbuckle/setup_user.sh"
  ~/libmemcached-1.0.15/clients/memaslap -s 10.10.0.3:11211 \
    --udp -t 30s -T ${4} -c ${5} -S 15s > \
    ~/ubdata/userspace_data/key-${1}_value-${2}_user-${3}.dat
  ssh 128.232.33.16 "~/unbuckle/setup_rmuser.sh"
}

function run_test {
  # Set up .memslap.cnf file
  echo -e "key\n"$1"␣"$1"␣1\n" \
    "value\n"$2"␣"$2"␣1\n" \
    "cmd\n0␣0.1\n1␣0.9" > \
    ~/.memslap.cnf

  # Run against user-space three times to take an average.
  for i in 1 2 3
  do
    run_user_test $1 $2 $i $3 $4
    q206_reboot
  done

  # Now run the same value size against the kernel version.
  for i in 1 2 3
  do
    run_kernel_test $1 $2 $i $3 $4
    q206_reboot
  done
}

echo Why not watch a video while you wait?
echo https://www.youtube.com/watch?v=dQw4w9WgXcQ
mkdir userspace_data/
# Value size distribution to run tests against
for s in 1 16 64 300 600 900 1024
do
  run_test 64 $s $t $c
done
```

Listing A.1: Shell script used to run the Unbuckle *user-space* vs. *kernel simulation.*

Analysis

The analysis of the data was conducted using a Python script, producing the graphs presented in chapter 4. For brevity, the script is not reproduced here. It can be found in the project source code repository at `test/consolidate.py`.

A.3.3 Optimisation of memcached

In order to ensure the evaluation was as fair as possible, I optimised memcached on the test machines in accordance with a paper by *SolarFlare*, which documents various techniques to extract performance from such installations [49]. A shell script was implemented to automate this process during the experiments. This script:

- Disables Hyperthreading
- Disables swapping memory to disk
- Pins the memcached worker thread(s) to particular CPU cores
- Stops non-essential system services
- Migrates system threads and other programs to a single CPU core to provide maximum CPU time on the other cores to the memcached process

The script which performs these tasks is given in listing A.2.

```
#!/bin/bash

# Optimise machine for running memcached

# Shutdown non-critical services
for s in \
  acpid cgconfig irqbalance atd auditd \
  avahi-daemon cpuspeed crond haldaemon \
  iscsid kdump ksm ksmtuned libvirt-guests \
  libvirtd lvm2-monitor mdmonitor microcode_ctl \
  openct pcscd portreserve sysstat
do
  service $s stop
done

# Disable hyperthreading if not already disabled in BIOS
# WARNING: this operation is not idempotent
for c in `cat /sys/devices/system/cpu/cpu*/topology/ \
  thread_siblings_list | cut -f2 -d ',' | sort -n | uniq`
do
  echo 0 > /sys/devices/system/cpu/cpu$c/online
```

```
done

# Disable swapping
swapoff -a

# Hugepage support
echo 128 > /proc/sys/vm/nr_hugepages

# Migrate non-memcache threads to CPU 0
cset set --cpu=0 system-tasks

# Move all other programs into the system task shield
cset proc --move / system-tasks --kthread

# memcache shield
cset set --cpu=0-3 --mem=0 mc-node0

ethtool -C eth4 adaptive-rx off
ethtool -C eth4 rx-usecs 0

# Start up memcached with 65GB memory in UDP mode
cset proc --set=mc-node0 --exec -- \
  /home/r2d2/memcached-1.4.17/memcached -t 1 -U 11211 \
  -u r2d2 -m 65536 &

# Set processor affinity to pin worker threads
MCID=`ps aux | grep [m]emcache | awk '{print $2}'`
# Print threads to allow them to be pinned to CPUs
ps -p $MCID -o tid= -L | sort -n
```

Listing A.2: Shell script used for optimising a test machine for memcached performance

A.4 Project proposal

Matthew Huxtable
St John's College
mjh233

Part II Project Proposal

An in-kernel high-performance key-value store

22nd October 2013

Project Originators: Malte Schwarzkopf & Dr Steven Hand

Resources Required: No special resources

Project Supervisors: Malte Schwarzkopf & Dr Steven Hand

Signatures:

Director of Studies: Malte Schwarzkopf

Signature:

Overseers: Dr Anil Madhavapeddy & Dr Simone Teufel

Signatures:

Introduction & Description

In recent years, web-based software platforms have grown and now demand high-performance methods for storing datasets of unprecedented size. As these systems become more pervasive and their user densities increase, so too does the volume of data and the rate at which it must be stored, retrieved, generated and transmitted to end-users. It has become apparent that conventional methods of data storage, such as the relational database model [11], are unable to scale effectively to meet today's demands. Key-value stores have received considerable attention as an alternative solution. This is exemplified by sustained real-world use in social media [2, 31, 32], web search [10], and e-commerce [14].

The key-value store is a simple, unstructured and often distributed database system in which the schema is encoded in external applications rather than the database itself. A simple mapping is made between a key and its value, with no further formalisation of the structure of these data. In trivial cases, the value may consist of a simple alphanumerical string; however, it is common to store serialised objects, making the store a simple extension of an object-oriented programming environment.

The simple nature of the key-value store is by contrast to the conventional relational database management system (RDBMS). Key-value stores achieve their performance by sacrificing many of the guarantees offered by the database engine in an RDBMS. Lack of an internally enforced data model was already described; other changes include: the ACID properties,[4] data normalisation, and disk storage. This overcomes many issues: relaxing consistency guarantees makes it simpler to replicate data among multiple nodes, maintain high transactional throughput and respond to system failures; avoiding data normalisation eliminates costly in-memory joins to reconstruct a useful view of a dataset; in-memory storage delivers data more efficiently and with less overhead than complex disk subsystems but at the expense of volatility.

Popular key-value stores in active use today as database systems include Apache Cassandra,[5] Redis,[6] and HBase.[7] There also exists distributed object caching systems based upon the same principle; these act as front-end caches to other systems, storing pre-computed instances of data for a suitable period of time. This avoids the otherwise inevitable load on back-end systems were these data to be fetched for every request. memcached[8] is a popular implementation.

Unfortunately, the existing implementations suffer performance issues at scale. This may be due to implementation choices, particularly for Cassandra and HBase, both of which are implemented in Java and suffer the overhead of a Java Virtual Machine.

[4]Atomicity, Consistency, Isolation and Durability – common properties offered to users of an RDBMS which are implemented internally within the database engine.

[5]`http://cassandra.apache.org/`

[6]`http://redis.io/`

[7]`http://hbase.apache.org/`

[8]`http://www.memcached.org/`

However, these systems also run as processes in userspace, under the jurisdiction of an operating system. Each request within the application to perform a privileged task requires invoking the kernel by means of a system call ("syscall"). This includes such tasks as: the transmission of a packet of data in response to a query; a request to read data from, or write data to, a file on disk; the spawning and scheduling of a new thread of execution. The relatively small data sizes leads to a high request throughput which results in a relatively large number of syscalls.

I propose taking the existing notion of a key-value store, implement it as a kernel module, and test the performance of the implementation against an equivalent store implemented in userspace. In this methodology, fewer transitions into the kernel via syscalls are necessary, as the system already has the highest level of privilege including direct access to system resources – the file system, physical memory and network. Cycles previously wasted changing the mode of the processor can be used to serve additional requests, potentially leading to improvements in system throughput. Traditionally, separation of privilege between user and kernel mode serves a vital purpose in abstracting low-level system details from application code. However, it is widely acknowledged that interrupting the CPU in order to elevate to supervisor mode leads to wasted cycles and poor performance during the privilege switch.[9]

An in-kernel implementation will naturally remove many of the kernel's security benefits, particularly those intended to maintain order on multi-user systems. However, this is not a major concern. Nodes which operate a key-value store are typically dedicated to the task, so concerns over process isolation and fairness are not relevant as the system environment is not shared. Furthermore, it is common to operate nodes in a virtualised environment in which additional security benefits are provided by the underlying hypervisor. Work on unikernels has shown additional benefits in optimising these workloads in the form of specialised microkernels, despite the security concerns [37].

Resources Required

The resources required for this project are minimal:

- My personal Linux-based laptop, for development purposes. This is an Intel Core i5 machine with 8 GB RAM.
- The Linux kernel source code, freely available.[10]

My personal laptop fulfils the development requirements, as I have full control over the system and its configuration. Testing will be performed initially using a running Linux instance in a virtual machine. This provides the flexibility to install and test

[9] Recent work in the Systems Research Group at the Computer Laboratory observed an average penalty incurred for each syscall of $5 \mu s$ on an Intel Core i7 machine.

[10] `www.kernel.org`

code without the prospect of a bug crashing my primary development machine. It also permits the use of snapshots to effortlessly roll back the test environment to an earlier state, if necessary.

In order to ensure the source code and project report are securely stored, automated and manual backups will be made on a regular basis from my revision control system to several locations: the MCS filestore, my own personal file server (outside Cambridge) and an offline hard disk.

Starting Point

To complete this project, I will be drawing heavily on prior knowledge gained from:

- **Parts IA and IB of the Computer Science Tripos**
  One of the reasons for my pursuing this project is a desire to understand the low-level operation of a complex system kernel in more depth based on the theory I have covered so far in the Tripos. This includes the theory of OS, computer architecture and kernel design taught in the Part IA *Operating Systems* course and Part IB courses in *Algorithms*, *Computer Design*, *Concurrent and Distributed Systems* and *Programming in C and C++*.
- **Existing research**
  A wealth of research is being published at present on the topic of high-performance key-value stores, as they grow in importance with large internet companies. It may be possible to draw ideas from those being published for my kernel-based implementation, especially where optimisations are concerned [21,34,39].
- **Own systems and programming experience**
  This project will build heavily on previous programming work in earlier years of the Tripos and elsewhere. I have spent some of my vacation time over recent years building networking and database systems, so I am familiar with many of the low-level details of these protocols. Additionally, in the Part IB Group Projects, my team's project concerned the real-time analysis of network log files using Hadoop, which depended heavily on key-value concepts, so I already have some exposure to the systems I will be using as part of this project.

To the best of my knowledge, there is no equivalent open-source product of this nature in widespread usage today. However, I am motivated by similar work done in the area of HTTP web servers. Operating some aspects of these systems in-kernel has been shown to be advantageous [28].

Structure of the Project

My intention is to split this project into multiple phases for manageability. These phases work towards a common purpose of obtaining an implementation of a key-value store with in-memory data storage. In order for a fair performance comparison to be performed, it must be possible to compile the store into two modes: a kernel module and a userspace equivalent. This will require good software engineering and dual implementations of some low-level operations. Good code abstraction will ensure the build process simply links with kernel or userspace implementations depending on the mode of compilation.

Interaction with the store will need to adhere to a suitable protocol. A subset of operations implemented by the memcached protocol[11] is attractive. This is a relatively simple and widely used protocol with the additional benefit of a binary mode of operation, removing the need to implement string handling over ASCII text input. Using an existing protocol also promises seamless compatibility between applications developed for memcached and those developed for the kernel equivalent, which may be useful in testing or for future use of this project.

Phase 1 – Core Implementation

In the first phase, a naïve implementation will be made within a Linux kernel module. This will serve several purposes. Firstly, it keeps the complexity of the required code relatively low at the beginning of the project, allowing me to focus on familiarising myself with the intricacies of kernel programming. Secondly, by postponing potential optimisations until later, it should be possible to observe positive progress being made at a relatively early milestone. It will be necessary to investigate a number of areas at this early stage and answer several high-level questions, including:

- The network protocol to support – should the implementation support communication over UDP, TCP (potentially incurring additional latency but with guarantees of data integrity), or both?
- Selection of suitable data structures to maximise efficiency in key-value pair and index storage – existing work includes the use of tries [34], distributed hash tables and Masstree [39], a proposed combination of B$^+$-trees and tries. Exploration of suitable libraries which already implement these structures will be necessary in order to decide whether they can be used as-is or must be re-implemented. There is the added constraint that such libraries cannot make use of the C Standard Library because this is unavailable in the Linux kernel, so I expect it will be necessary to implement many of them from scratch.

[11]https://github.com/memcached/memcached/blob/master/doc/protocol.txt

Phase 2 – Testing and Performance Evaluation

On completion of a working module, it will be necessary to perform suitable tests to gather performance data. In order to perform a comparison, each test will be performed twice: against an in-kernel instance and an instance operating in userspace. This will provide a definitive, real-world test which determines whether moving this task into the kernel shows an appreciable performance change. I will investigate whether existing test frameworks are suitable for this purpose, or whether construction of my own will be necessary. The tool must be capable of benchmarking the system through the simulation of a workload, followed by the collection of statistics for an analysis to be performed, such as request throughput (in terms of requests per second) and latency (in terms of μs/req). Several tools exist already, including the *Yahoo! Cloud Serving Benchmark* [12] and *memaslap*.[12]

Phase 3 – Enhancements and Optimisations

The final phase will be devoted to introducing optimisations, as extensions in a second iteration of the kernel module. See Possible Extensions on this page for ideas and further discussion.

Possible Extensions

Working within the system kernel provides a much greater degree of insight and control over system resources than is possible in userspace. It is possible to conceive a great many possible extensions to the initial kernel module which exploit the additional information and control interfaces in a bid to further optimise the system's performance:

- **Direct Memory Access (DMA)** – access to the physical system hardware makes it possible to request DMA data transfers from main memory to other system devices, freeing the processor to service other requests. The network interface card (NIC) is a candidate for the use of DMA. In a system which spends much of its time communicating with the network, a considerable amount of work is done transferring data between the NIC's buffers and the data structures in main memory.
- **Process level optimisations** – awareness of, and direct access to, the scheduler opens the potential to "bias" the scheduler in the key-value store module's favour. Thus, smarter decisions may be made with regards to when and upon which core key-value store threads will be executed, with the potential on multiprocessor systems that cores can be dedicated to the execution of the database.

[12]http://docs.libmemcached.org/bin/memaslap.html

- **Network communication optimisations** – the `sk_buff` data structure represents a socket buffer in kernel networking code. All data packets currently queued in the kernel are represented by `sk_buff` objects in a doubly linked list, where the data structure contains pointers to the data and associated information necessary for properly processing the packet (such as protocol headers). Allocating memory for instances of the `sk_buff` data structure may constitute a time consuming process, which could hinder query throughput to the key-value store. A point of investigation would involve storing instances of the `sk_buff` data structure within the key-value store, allowing these to be directly read out and spliced into the in-memory packet queue with only minimal effort to make the structure consistent by setting pointers to the appropriate protocol header structures. Similarly, previous work has shown it may be possible to reduce the requirement for memory-to-memory copies by working directly in the NIC buffers [29].
- **Core data storage optimisations** – Masstree [39], introduced in Phase 1 – Core Implementation on page 81, is a proposed algorithm for storing key-value pairs in-memory which respects the operations taking place in low-level hardware on symmetric multiprocessor systems. This is primarily focussed on the behaviour surrounding the creation and eviction of cache lines. Furthermore, Masstree's protocol addresses specific concerns over contention when reading or writing the data structure, and implements techniques such as Optimistic Concurrency Control (OCC) to enforce consistency without significantly sacrificing performance. The data structure aims to work alongside caching – rather than fight against or simply neglect its behaviour, both actions which could have significant performance impacts. Many of the optimisations discussed in the paper are relevant to this project. If the decision to go with this advanced data structure is not made earlier in the project, it should almost certainly be investigated and implemented as an extension in phase 3 if time permits.

Timetable: Workplan and Milestones

In accordance with recommended practice, I have split the entire project into approximately fortnightly work packages. I intend to complete the work by the middle of March. This allows time to write the dissertation over the Easter vacation, revise for the forthcoming exams during Easter term, and adds buffer time should this be necessary.

- **8th October to 22nd October**
  - Investigate existing work in this area, including academic papers and open-source key-value store implementations
  - Work on the project proposal (deadline: 25th October)

  - Read around the kernel in suitable literature and experiment with dummy code snippets and modules as familiarisation with this particular form of programming

  Milestone: Complete and submit project proposal

- **23rd October to 5th November**
  - Further research and familiarisation with kernel concepts. Study the memcached protocol in more depth, perhaps even its current implementation. Experiment with an existing memcached instance to demonstrate operation of the current system on a small scale.
  - Investigate existing libraries which may be of use, particularly any with a focus on algorithm implementation in-kernel.
  - Configure and test the development environment and disaster recovery procedures
  - Set up the skeleton kernel module structure and begin to implement the core code, particularly the core algorithm for storing key-value pairs which will be used in the first iteration.
  - Build suitable unit test frameworks for code written *(ongoing from this point)*

- **6th November to 19th November**
  - Complete back-end data storage.
  - Implement the memcached protocol in the key-value store and test input/output accordingly.
  - Begin investigating test frameworks, evaluate YCSB (as introduced in Phase 2 – Testing and Performance Evaluation on page 82) and work on adapting this or implementing my own if necessary.

- **20th November to 3rd December**
  - Integrate network communications with the back-end data storage components.
  - Finalise phase 1 of the key-value store.
  - Complete test frameworks, build a test workload and run this on the key-value store in both kernel and userspace modes.

***6th December** – end of full Michaelmas term*

- **4th December to 17th December**

  Further time for testing.

  - Continue running test workloads in order to evaluate the system.
  - Implement bug fixes for any identified issues.

  Milestone: Phases 1 & 2 completed by end of timeslot

- **18th December to 31st December**

  Progress report and obligatory catchup time.

  Milestone: Draft of the progress report by end of timeslot.

- **1st January to 14th January**

  Implementation time for phase 3 (optional extensions) and wrap up of any outstanding work from phases 1 and 2.

  - Begin the implementation of the optional extensions as outlined in Possible Extensions on page 82.
  - Investigate further extension possibilities which may have come to light during the development stage of the project.
  - Continually test and evaluate the effect of extensions as they are completed *(ongoing from this point)*
  - Consider any additional minor enhancements to the phase 1 implementation.

***14th January** – start of full Lent term*

- **15th January to 29th January**

  Completion and hand-in of progress report.

  - Incorporate any necessary modifications to the progress report, update with new developments, and hand-in.

  - Work on the presentation to overseers and other students, including the preparation of any required materials *(scheduled to take place on one of 6th, 7th, 10th or 11th February)*
  - Continue implementation of optional extensions or performance enhancing features.

  Deadline *(hard)*: Hand-in the progress report.

  Milestone: Have a presentation ready for early February.

  Milestone: Have a fully working key-value store with kernel-level optimisations implemented. Evaluate using the testing framework.

- **30th January to 12th February**

  Further extension implementation and catchup time.

  - Continue implementing optional extensions to the project.
  - Deliver the progress presentation.

- **13th February to 24th February**

  Final opportunity for the implementation of any last minute enhancements, additions or tweaks.

  - Heavily test the key-value store as it now stands, with a view to generating evaluation data suitable for the dissertation write-up which is about to commence.
  - Wrap up any final extensions and any other code in development.

  Milestone: The substance of the key-value store kernel module – including the optional extensions as previously selected – implemented and tested with full simulated production workloads.

- **25th February to 11th March**

  Start writing dissertation. Buffer time for bug fixes and any outstanding issues.

- **11th March to 8th April** (note: double length slot)

  Complete dissertation to draft standard.

  - Write the main sections of the dissertation. Focus on generating appropriate figures and writing an evaluation based on the results produced.
  - Typeset the dissertation to a standard suitable for handing in for review.

  Milestone: Dissertation draft complete by end of timeslot.

  ***14th March** – end of full Lent term (falls within timeslot)*

- **8th April to 22nd April**

  Finish dissertation.

  - Finish off the dissertation, adding additional evaluation results where necessary.
  - Incorporate feedback on the draft as received from supervisor and any additional proofreaders.

  ***22nd April** – start of full Easter term*

- **22nd April to 16th May** (note: extra long slot)

  Buffer time. Dissertation deadline at end of timeslot.

  - Address any outstanding issues with the dissertation.
  - Hand the dissertation in and upload a tarball containing the source code.
  - Revision for the exams.

  Deadline *(hard)*: complete administrivia to finalise the project, preferably in good time – hand in dissertation, upload source code and the dissertation in electronic form. Momentarily breathe a sigh of relief before thinking about exams again.