Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels

by

Silas Boyd-Wickizer

B.S., Stanford University (2004)
M.S., Stanford University (2006)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

Author ....
Department of Electrical Engineering and Computer Science
September 5, 2013

Certified by. [illegible] ....
Robert Morris
Professor
Thesis Supervisor

Accepted by .... [illegible] ....
Leslie A. Kolodziejski
Professor
Chair of the Committee on Graduate Students

Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels

by
Silas Boyd-Wickizer

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One difficulty of programming multicore processors is achieving performance that scales with the number of cores in the system. A common performance optimization is to increase inter-core parallelism. If the application is sufficiently parallelized, developers might hope that performance would scale as core count increases. Unfortunately for some applications, such as operating system kernels, parallelization reveals inter-core communication as a performance bottleneck. When data is updated on one core and read or written on other cores, the cache coherence protocol serializes accesses to the data. The result is that each access to the shared data can take hundreds to thousands of cycles, depending on how many cores are contending for the data.

This dissertation focuses on optimizing communication bottlenecks caused by update-heavy workloads, where a data structure is frequently updated but rarely read. Such data structures are commonly used for operating system kernel bookkeeping, such as LRU lists, reverse maps in virtual memory, and file system notification queues. This dissertation identifies bottlenecks in the Linux kernel caused by update-heavy data structures, presents a general approach for optimizing communication in update-heavy data structures, and presents a library called OpLog that embodies this approach and helps developers achieve good scalability for update-heavy data structures. OpLog achieves scalability by logging update operations in per-core logs, and combining the logs only when required by a read to the data structure. Measurements on a 48-core AMD server show that applying OpLog to update-heavy data structures in the Linux kernel significantly improves application performance under certain workloads.

Thesis Supervisor: Robert Morris
Title: Professor

Contents

List of Figures

Chapter 1

Introduction

Cache-coherent shared-memory multiprocessor hardware has become the default in modern PCs as chip manufacturers have adopted multicore architectures. Most PCs use multicore processors and manufacturers build 16-core $x86$ processors and motherboards that support up to eight multicore processors. Trends suggest that the the number of cores per chip will increase in the foreseeable future.

The dissertation investigates the challenge of optimizing inter-core communication in multiprocessor operating system kernels. One difficulty of programming multicore processors is achieving performance that scales with the number of cores in the system. A common performance optimization is to increase inter-core parallelism by, for example, refactoring the application to use many threads synchronized with locks. If the application is sufficiently parallelized, developers might hope that performance would scale as core count increases. Unfortunately for some applications, such as operating system kernels, parallelization reveals inter-core communication as a performance bottleneck.

An *inter-core communication bottleneck* is when cores spend time waiting for the hardware interconnect to process communication (*i.e.*, waiting for a cache line transfer), instead of executing instructions. For example, if multiple cores modify the same data structure, cores must wait for the cache coherence hardware to transfer the cache lines holding the data structure. Transferring cache lines between cores can take thousands of cycles, which is longer than many system calls take to complete.

The main topics of this dissertation are the discussion of the inter-core communication bottleneck; an approach for optimizing a common class of communication bottlenecks; the design and implementation of programming library, called OpLog, that embodies this approach; and an evaluation of OpLog applied to a multiprocessor operating system kernel.

1.1 Context

Operating system performance is important. Many applications spend a significant amount of time executing in the kernel. For example, when running on a single core, the Apache web server and the Exim mail server each spend more than 60% of their execution time in the kernel. Unfortunately, even if an application spends a small amount of execution time in the kernel on a single core, an unoptimized kernel subsystem can bottleneck performance when running on multiple cores. For example, the PostgreSQL database spends 1.5% of its execution time in the kernel on one core, but this grows to 82% when running a recent version of the Linux kernel on 48 cores. The performance of many applications is dependent on the operating system kernel. If kernel performance does not scale, neither will application performance.

Early multiprocessor kernel implementations provided almost no parallelism. This was partly because early multiprocessor PCs had very few cores, and thus there was little benefit from implementing a highly parallel kernel; and partly because allowing concurrent access to shared resources is difficult to implement correctly. Initial versions of multiprocessor operating systems protected the entire kernel with a single lock. This required little developer effort to understand data structure invariants, but provided poor performance as the number of cores increased because only one core could execute in the kernel at a time.

Multiprocessor kernels have evolved to expose more and more parallelism: from a single lock forcing serial execution of the whole kernel, to a lock for each subsystem, to a separate lock for each kernel object such as a file or a network socket. This progression is driven by increases in the number of cores: as more cores execute inside the kernel, locks must protect smaller pieces of code or data so that cores execute without waiting for each others' locks. Many critical sections are already as short as they can be (a few instructions) in kernels such as Linux. The logical next step has started to appear: lock-free[1] data structures, as well as replicated or partitioned per-core data that is locked but rarely contended.

1.2 Problem

Increases in core counts and the elimination of lock contention has revealed a different problem: the cost of communicating shared data among cores. It is common for a critical section, or its lock-free equivalent, to read and write shared data. When data is written on one core and read or written on other cores, the cache coherence protocol serializes accesses to the data. The result is that each access to the shared data can take hundreds to thousands of cycles, depending on how many cores are contending for the data. That expense is of the same order as an entire system call. Chapter 4 quantifies these costs on a modern multicore cache-coherent computer.

Since contention for shared data is very expensive, the next step in improving performance, after lock contention has been eliminated, must be reducing inter-core communication. This means, for example, that lock-free data structure designs must pay as much attention to optimizing communication patterns as they do to eliminating lock costs.

This dissertation focuses on reducing inter-core communication for data structures that the operating system updates frequently, but reads less frequently. Operating systems often maintain bookkeeping data structures that fit this access pattern. For example, LRU lists are usually updated with every access, but read only when the system needs to evict an object. As another example, when the virtual memory system maps a physical page, it updates a *reverse map* that tracks the set of page tables mapping a given physical page. However, the kernel consults the reverse map only when it swaps a page out to disk or truncates a mapped file. These actions are usually infrequent. Chapters 3 identifies and analyzes parts of the Linux kernel that are update intensive and bottlenecked by communication.

The observations in this dissertation apply only in certain situations. They are most relevant for traditional kernel designs with many shared data structures. The shared data must be written often enough to have an effect on performance. The sharing must be necessary, since if it can conveniently be eliminated altogether, that is usually the best approach.

[1]We mean "lock-free" in the informal sense of manipulating shared data without locks.

1.3 Challenges with current approaches

Techniques to reduce inter-core communication are often quite different from those aimed at increasing parallelism. For example, replacing locked reference count updates with atomic increment instructions improves parallelism, but that improvement is limited by the movement of the counter between cores.

A common approach for reducing inter-core communication is to apply updates to a per-core data structure, instead of a single shared one. Using a separate data structure for each core, however, requires the programmer to design a strategy for reconciling updates to different data structures and keeping them consistent. Consider replacing a contended global LRU list with per-core LRU lists. Instead of evicting the first item from the LRU list, the programmer would have to come up with a strategy to select one object from n per-core lists, which is complex because the oldest object on one core may have been recently accessed on another core. Several naïve approaches, such as keeping the most recent access time in the object itself, or strictly partitioning objects between the n LRU lists, do not solve the problem either, because they require cache line transfers when the same object is accessed by many cores.

Using techniques such as batching, absorption, and other classic communication-optimizing techniques can improve performance substantially, however, it is not always straightforward to apply these techniques. For example, if the kernel adds a mapping to the shared reverse map, then removes it before ever reading the reverse map, the kernel could have avoided modifying the shared data structure. Delaying the add operation would make it possible for the kernel to cancel the add operation when trying to remove the mapping. For this optimization to work correctly, the kernel must ensure that it applies the delayed add operation before reading the reverse map. Unfortunately we know of no general infrastructure or API for writing these types of communication optimizations.

1.4 Approach

This dissertation presents an approach for scaling data structures under update-heavy workloads, and an implementation of this approach in a library called OpLog. By "update-heavy" we mean that the callers of most operations do not need to know the result, so that the work can be deferred. At a high level, our approach is to log each update to the shared data structure in a per-core log, rather than immediately updating the data structure, and before the shared data structure is read, bring it up-to-date by applying all of the updates from the per-core logs. The order of updates matters in some data structures; for example, in an LRU list, the order of updates determines which object was used more recently, and in a reverse map for a virtual memory system, inserting an entry must happen before removing that entry (even if the removal happens on a different core due to process migration). To ensure that updates are applied in the correct order, our idea is to record a timestamp in each log entry, and merge logs by timestamp before applying them.

The OpLog approach has two main benefits: it ensures that updates scale, since each update need only append to a per-core log, incurring no lock contention or cache line transfers, and it ensures that the semantics of the data structure remain the same, because logged updates are applied before the shared data structure is read. However, OpLog must overcome three challenges in order to work well in practice:

- first, existing data structures must be adapted to a logging implementation;

- second, replaying a long log on read operations may be too expensive; and
- third, storing all the log entries, for every object in the system, would require a large amount of memory.

OpLog addresses these challenges using the following ideas.

First, OpLog provides an API that is easily layered on top of a data structure implementation, to transform it into a logged implementation. Instead of updating an underlying data structure, an update function uses the OpLog `queue` function to append a closure with the update code to the current core's per-core log. Prior to reading the underlying data structure, read functions must first call the OpLog `synchronize` function, which merges the per-core logs by timestamp and executes each closure. The most basic usage of OpLog essentially requires the programmer to specify which operations read the data structure and which operations update the data structure. The data structure API and the code that invokes the API do not require changes.

Second, OpLog absorbs logged updates, if allowed by the semantics of the data structure. For example, instead of logging an update that removes a particular item from a list, OpLog can instead drop a previously logged update that added the item to the list in the first place. This keeps the logs short, reducing the cost of the next read operation.

Third, OpLog mitigates memory overhead by dynamically allocating log space. If a data structure receives many updates, OpLog stores the updates in a per-core log; otherwise, it maintains only the shared data structure, and immediately applies the updates to it.

1.5 Results

To demonstrate that OpLog can improve the performance of real systems, we implemented two prototypes of OpLog: one in C++ for user-space applications, and one in C for the Linux kernel. We applied OpLog to several data structures in the Linux kernel: the global list of open files, the virtual memory reverse map, the inotify update notification queue, and the reference counter in the directory name cache.

An evaluation on a 48-core AMD server shows that these uses of OpLog improve the performance of two real applications. OpLog removes contention on reference counts in the directory name cache, encountered by the Apache web server, allowing Apache to scale perfectly to 48 cores when serving a small number of static files. OpLog also removes contention on the virtual memory reverse map, encountered by the Exim mail server, improving its performance by 35% at 48 cores over a lock-free version of the reverse map under the workload from MOSBENCH [6].

A comparison of optimizations implemented with OpLog to optimizations implemented by hand using per-core data structures shows that OpLog reduces the amount of code the programmer must write without sacrificing performance.

1.6 Contributions

The main contributions of this dissertation are as follows:

- The identification of a common class of update-heavy communication bottlenecks in modern multiprocessor operating system kernels.
- The OpLog approach for optimizing update-heavy data structures.

- The design and implementation of the OpLog API and programming library.
- An evaluation demonstrating that communication bottlenecks affect application performance and that applying OpLog removes the bottlenecks and improves performance.

1.7 Overview

This dissertation is organized as follows. Chapter 2 provides some context on the current state of modern multiprocessor kernels by describing techniques kernel developers have used to increase parallelism and ultimately uncover communication bottlenecks. Chapter 3 provides several examples of update-heavy data structures in the Linux kernel, along with performance measurements to demonstrate that they are scalability bottlenecks. Chapter 4 presents results showing that the bottlenecks in Chapter 3 are communication bottlenecks. Chapter 5 outlines OpLog's approach, Chapter 6 describes OpLog's library interface, and Chapter 7 describes its implementation. Chapter 8 evaluates OpLog and Chapter 9 discusses related work. Chapter 10 speculates on the future importance of OpLog, identifies areas of future work, and concludes.

Chapter 2

Parallelism in Multiprocessor Kernels

Communication tends to bottleneck performance once an operating system is well parallelized. Many early versions of multiprocessor operating system kernels had limited parallelism and developers devoted much effort to increasing parallelism before uncovering communication bottlenecks. To help provide context for the current state of highly parallel multiprocessor kernels, this chapter gives an overview of techniques Linux kernel developers have used for optimizing parallelism. By applying the techniques described in this chapter, kernel developers have started to uncovered communication bottlenecks, some of which are detailed in the next chapter.

The multiprocessor Linux kernel evolved from using a single lock to serialize kernel execution to a highly parallel kernel with many short serial sections. In some cases developers removed coarse-grained locks by refactoring the implementation to use many finer grained locks. In other cases developers applied alternatives to locks, like Read-Copy-Update, to shorten serial sections. We give an overview of locking in Linux, describe three popular parallelization techniques, (per-core locking, read-write locks, and Read-Copy-Update) that provide alternatives to fine grained locking, and walk through an application case study to illustrate the parallelization techniques. We focus on Linux as an example of a highly parallel kernel, because Linux developers have spent much energy optimizing it for multicore and large-scale SMP machines [6].

2.1 Fine grained locking

Early multiprocessor versions Linux used coarse-grained locking and supported limited amounts of parallel execution in the kernel. Coarse-grained locking caused performance bottlenecks because CPUs would spend most of their time waiting to acquire locks. One approach Linux developers have taken to eliminate these bottlenecks is fine-grained locking. Replacing a single coarse-grained lock protecting many data objects with many fine-grained locks that each protect a single object increases parallelism because CPUs can access different objects in parallel.

One interesting issue with spinlock usage in the Linux kernel is that contended spinlocks protecting short serial sections can cause performance to collapse. The time it takes to acquire a Linux spinlock is proportional to the square of the number of contending CPUs. For short serial sections, the time to acquire the spinlock dominates the cost of actually executing the serial section. If more than a few CPUs are waiting to acquire a spinlock, performance often collapses. There are situations where application throughput initially scales with the number CPUs, but suddenly collapses with the addition of another CPU. Appendix A provides a model that explains this behavior.

It may seem reasonable to replace the Linux spinlock implementation with a non-collapsing spinlock implementation such as MCS [21]. This would help to guarantee that performance would not suddenly collapse with the addition of CPUs. One problem, however, with non-collapsing spinlock implementations is that they perform poorly in the absence of contention compared to the Linux spinlock implementation. Acquiring and releasing an uncontended Linux spinlock takes 32 cycles, while an MCS lock, for example, takes 50 cycles [7].

Uncontended spinlock performance is important for data structures optimized to use per-core locks, because per-core locks are rarely contended. Even if data structures are not optimized to use per-core locks, it is often possible for applications to arrange for different cores to access different kernel data structures. For example, in Linux each directory has a list of directory entries that is protected by a spinlock. Concurrently creating or removing files on different cores can lead to performance collapse; however, it is usually possible for the application to arrange for different cores to access different directories.

2.2 Per-core locks

One way developers increase parallelism without shortening serial section length is by using per-core locks. Converting a data structure protected by a single lock into per-core data structures protected by per-core locks improves parallelism if cores frequently access their per-core data structures.

For example, per-core free page lists perform well because a core can usually allocate a page by using its free page list. Occasionally a core's free page list is empty and the core will refill it by acquiring another core's per-core free page list lock and removing pages from that core's free page list. OpLog also uses per-core locks to protect operation logs.

2.3 Read-write locks

Read-write locks can increase parallelism by allowing concurrent readers. Linux uses read-write locks in a some cases where a data structure is updated infrequently, but read frequently. For example, the mount table is protected by a read-write lock, because the kernel often accesses the mount table during pathname lookup, but is rarely required to modify it. Since the introduction of Read-Copy-Update, developers have frequently used Read-Copy-Update instead of read-write locks.

2.4 Read-Copy-Update

Read-Copy-Update (RCU) [19] is a synchronization primitive in the Linux kernel that allows readers to execute concurrently with writers. In addition to improved parallelism, RCU typically has lower storage overhead and execution overhead than read-write locks. RCU does not provide synchronization among writers, so developers use spinlocks or lock-free algorithms to synchronize writers.

2.5 Parallelism Case Study: Exim

The Exim [1] mail server provides an interesting case study for illustrating use of parallel techniques in the Linux kernel. With many concurrent client connections, Exim has a good deal of application parallelism. Exim is also system intensive, spending 69% of its time in the kernel on a single core, stressing process creation, file mapping, and small file creation and deletion. Exim exercises all the parallelization techniques discussed in this chapter.

We operate Exim in a mode where a single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which in turn accepts the incoming mail, queues it in a shared set of spool directories, appends it to the per-user mail file, deletes the spooled mail, and records the delivery in a shared log file. Each per-connection process also forks twice to deliver each message.

The rest of this section describes the subsystems Exim stresses and what techniques the subsystems use to achieve high parallelism.

Scheduling run queues Exim stresses the scheduler because it creates and deletes many processes that the kernel must balance among CPUs. The Linux scheduler uses per-core run queues. Each run queue is protected by a per-core spinlock. A core acquires its per-core runqueue lock before adding or removing processes from the local runqueue. If a core's runqueue is empty it will acquire the runqueue lock of another core and try to remove processes from that runqueue and add them to the local runqueue. If there are more runnable processes than cores it a core will likely manipulate its runqueue without interference from other cores.

Virtual file system Exim creates several files for each message it delivers. For each file Exim creates, the kernel must resolve the file pathname string into a kernel directory entry object. Linux resolves a pathname by walking the pathname and mapping each component into a directory entry object. Linux implements pathname walking using a hash table called the directory entry cache. The directory entry cache maps a tuple composed of the pathname component and a parent directory entry object to another directory entry object representing the pathname component. Lookup in the directory cache uses RCU and a combination of lock-free techniques.

Reverse page mapping The kernel maintains a reverse page mapping to locate all the virtual address spaces that map a physical page. Each time Exim forks a process the kernel adds all the virtual address mappings from the new process to the reverse map. When the process exits the kernel must remove the mappings from the reverse map. Linux implements a parallel reverse map using fine grained locking. Every file inode has an associated interval tree protected by a lock. Each interval tree maps physical pages caching file content to page table entries mapping the physical pages. If Exim processes map many files, cores simultaneously executing fork or exit will usually manipulate the reverse map in parallel. As the number of cores increases relative to the number of mapped files, cores will eventually contend on the locks protecting each interval tree.

Figure 2-1 shows the throughput of Exim on a 48-core machine running Linux version 3.9. Exim scales relatively well, delivering 10$\times$ as many messages on 48 cores than on one core. As the number of cores increases, however, each additional core improves throughput by a smaller amount and performance gradually flattens out. The locks in the Linux reverse mapping implementation that protect each interval tree bottleneck Exim's performance.

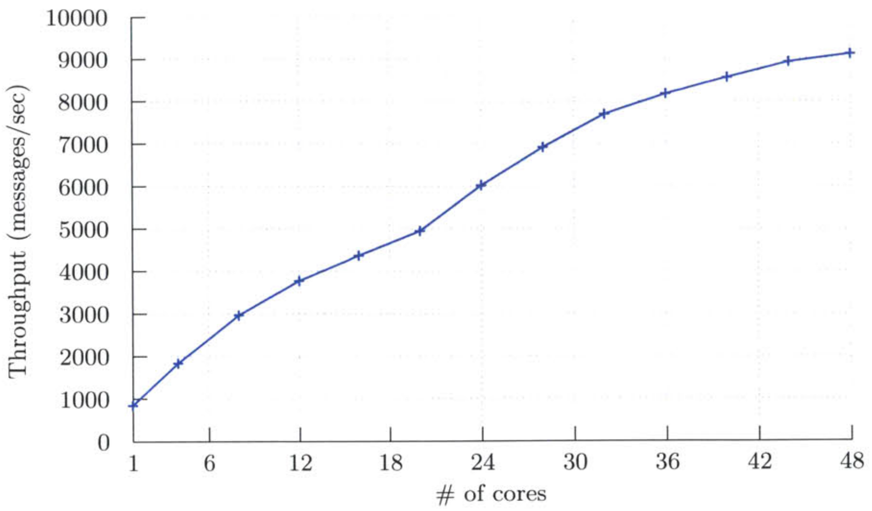


Figure 2-1: Performance of Exim.

2.6 Conclusion

This chapter provided context for the current state of highly parallel multiprocessor kernels. The first Linux kernel to support multiprocessor hardware used a single kernel lock to serialize all execution in the kernel. Kernel developers replaced the single kernel lock with highly parallel subsystems implemented using fine grained locking, per-core locks, read-write locks, and RCU. Increasing inter-core parallelism has uncovered performance bottlenecks due to inter-core communication.

The next two chapters present examples of communication bottlenecks. Chapter 3 presents example of Linux subsystems that are well parallelized, but still scale poorly on a multicore machine. Chapter 4 presents evidence suggesting that the bottlenecks in Chapter 3 are due to inter-core communication and shows how expensive communication costs are on a modern multicore computer.

Chapter 3

Problem

This chapter presents examples of Linux subsystems that are potential candidates for optimizing with OpLog. The examples are data structures that are update-heavy, whose multicore performance limited by contention, and where the full work of the update need not be completed immediately. However, all the examples are bottlenecked by lock contention, instead of contention on the underlying data structure. The Chapter 4 demonstrates that removing the locks still results in poor scaling due to contention on the data.

3.1 Examples in Linux

We present performance results from experiments on a 48-core $x86$ machine, composed of eight 6-core AMD Opteron Istanbul chips. Although the numbers are specific to the 48-core AMD machine, experiments on an Intel machine generate similar results. We focus on the Linux kernel because it has been extensively optimized for multicore and large-scale SMP machines [6]. We use a recent version (3.9) of the Linux kernel. For all numbers reported in this dissertation, we report the average of three runs of an experiment on our machine; there was little variation for all experiments.

The examples are heavily used and heavily optimized data structures: the name cache, containing name to inode mappings, the reverse map, containing physical page to virtual page mappings, and the inotify queue, containing notifications of file system changes. All of these data structures experience frequent updates: the kernel updates a reference count in the name cache on each pathname lookup, the reverse map must insert or remove a mapping each time a page is mapped or unmapped, and the kernel appends to the inotify queue every time a file or directory is changed. In the case of inotify, the data structure is also frequently read by an application that watches for file system updates. To isolate the scalability bottlenecks due to concurrent updates we measure these data structures with microbenchmarks; Chapter 8 shows that some of these bottlenecks can limit the scalability of real applications.

The inotify and name cache implementations use spinlocks to protect heavily updated data structures. The Linux spinlock implementation causes performance collapse when a spinlock is contended. Appendix A explains this behavior with a performance model and demonstrates that the collapse is due to the particular spinlock implementation. The rmap uses Linux's mutexes that sleep, instead of spin, when contended. To ensure that scalability bottlenecks are due to contention on the data structures, instead of poorly performing locks, Chapter 4 explores the performance of lock-free rmap and name cache implementations.

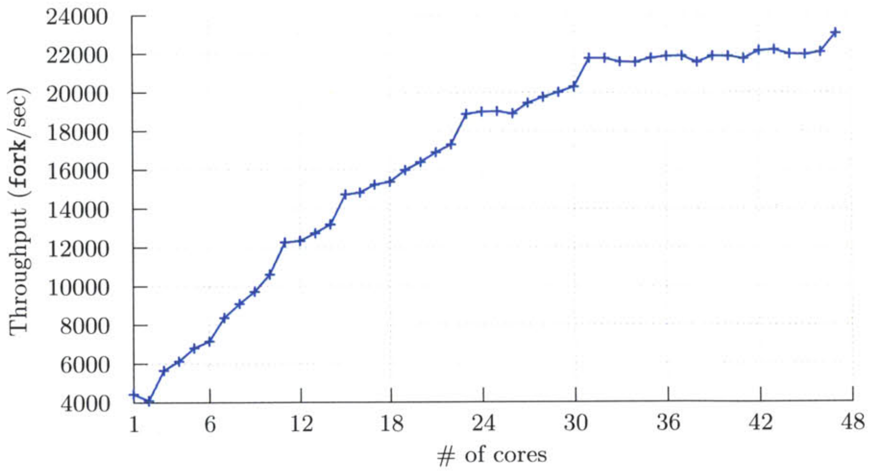


Figure 3-1: The **fork** benchmark performance.

3.2 Example: fork and exit

Creating and destroying processes efficiently is important. Applications fork and exit all the time (*e.g.*, Exim [6]), and processes often map the same files (*e.g.*, libc). The kernel must track which processes are mapping each file's pages, and it does so using a reverse map (rmap).

The rmap records for each physical page all page table entries that map that page. When a process truncates a file, the operating system must unmap the truncated pages from every process that maps the file; Linux uses the rmap to find those processes efficiently. Similarly, if the kernel evicts file pages from the disk buffer cache, it must be able to unmap the pages if they are mapped by any processes.

The Linux designers have heavily optimized the implementation of the rmap using interval trees [16–18]. Each file has an associated interval tree protected by a Linux mutex. A file's interval tree maps intervals of the file to processes that map that portion of the file. The Linux rmap implementation can become a bottleneck when many processes simultaneously try to map the same file. For example, a workload that creates many processes is likely to cause contention for the mutexes protecting the interval trees of commonly mapped files (*e.g.*, libc), because each call to **fork** and **exit** will acquire the interval tree mutex of popular files and update the interval trees. **fork** duplicates every file mapping in the parent process and insert each duplicated mapping into an interval tree, and **exit** removes each file mapping in the exiting process from an interval tree.

We wrote a benchmark to measure the performance of the rmap. The benchmark creates one process on each core. Each process repeatedly calls **fork** to create a child process that immediately calls **exit**. This stresses the rmap, because **fork** and **exit** update the rmap. Figure 3.2 shows the result. The x-axis shows the numbers of cores and y-axis shows the throughput in forks per second. The reason why the performance fails to scale is that multiple cores simultaneously try to update the same interval tree and contend on the lock protecting the tree.

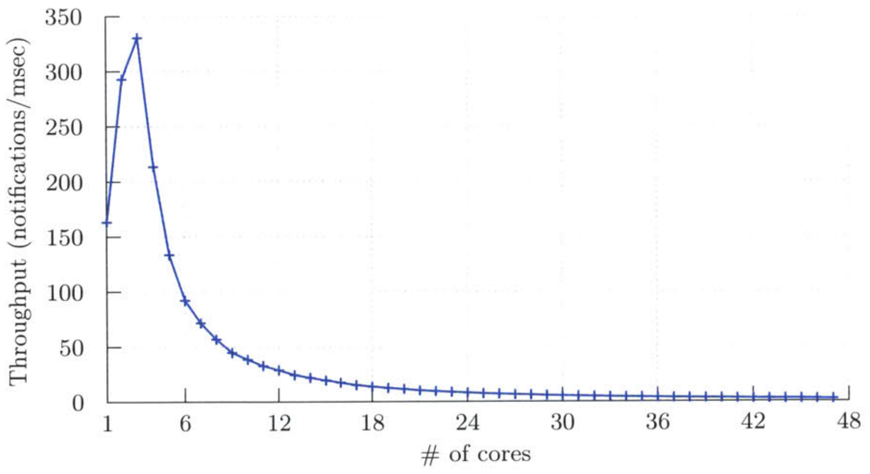


Figure 3-2: Performance of the `inotify` benchmark.

3.3 Example: inotify

Inotify is a kernel subsystem that reports changes to the file system to applications. An application registers a set of directories and files with the kernel. Each time one of the files or directories in this set is modified, the inotify subsystem appends a notification to a queue, which the application reads. File indexers, such as Recoll [2], rely on inotify in order to re-index new or modified files.

The inotify subsystem is interesting because the notification queue is both updated and read from frequently. The kernel maintains a single queue of notifications, and serializes updates to the queue with a spinlock, to ensure that notifications are delivered to the application in the correct order. For instance, if the queue contains both a creation and a deletion event for the same file name, the order determines whether the file still exists or not.

We wrote a benchmark to measure the performance of an update-and-read-heavy inotify workload. The benchmark creates a set of files, registers the files with inotify, and creates a process that continuously dequeues notifications. The benchmark then creates one process on each core. The benchmark assigns each process a different subset of files and each process continuously opens, modifies, and closes every file in its subset. Each file modification causes the kernel to append a notification to the queue.

Figure 3-2 presents the performance of the benchmark. Throughput increases up to three cores, but then collapses due to spinlock contention caused by multiple cores simultaneously trying to queue notifications.

3.4 Example: pathname lookup

In order to speed up pathname lookups, the Linux kernel maintains a mapping from directory identifier and pathname component to cached file/directory metadata. The mapping is called the `dcache`, and the entries are called `dentry`s. Each file and directory in active or recent use has a `dentry`. A `dentry` contains a pointer to the parent directory, the file's name within that directory, and metadata such as file length.

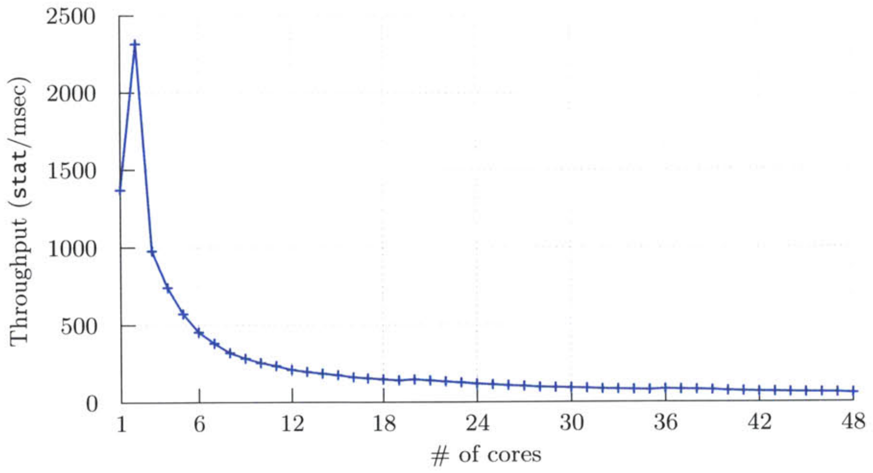


Figure 3-3: **stat** benchmark performance.

The **dcache** is generally well parallelized: **open**, **stat**, and so on, perform a pathname lookup lock-free using RCU [15], with one exception. For each **dentry** returned from a pathname lookup, the kernel checks the per-**dentry** generation count and increments a reference count in the **dentry**. When a kernel thread reads a **dentry** for the first time, it records its generation count, and then reads other fields of the **dentry**. If the thread decides it needs the **dentry**, it updates the **dentry**'s reference count, but it must first ensure that the generation count has not been changed by a remove operation. If it has been changed, the kernel must restart the lookup. To ensure that the check of the generation count and the increment of the reference count is atomic, the kernel uses a spinlock. The kernel performs this check only for the final component in a pathname. For example, calling **stat("/tmp/foo")** will cause the kernel to execute the generation count check only on the **dentry** associated with "foo".

We wrote a microbenchmark to evaluate the performance of the **dcache**. The benchmark creates one process per core; all of the processes repeatedly call **stat** on the same file name. Figure 3-3 shows the performance of this stat microbenchmark. The x-axis shows the number of cores and the y-axis shows throughput in terms of stat operations per millisecond. The performance does not increase with the number of cores because of contention for the spinlock used to atomically access the generation count and reference count associated with the argument to **stat**.

3.5 Conclusion

The Linux implementations of our examples use locks (spinlocks or Linux mutexes). Linux's spinlocks can suffer from performance collapse under contention, so their cost dominates performance, and Linux mutexes can cause processes to sleep for long periods. However, the fundamental problem is not lock contention: even without lock contention, the Linux examples would still have poor scaling due to contention over the data itself, and thus would still be promising targets for OpLog. To demonstrate this point, the next chapter presents results from re-implementing several examples to use lock-free data structures.

Chapter 4

Cost of communication

This chapter demonstrates that communication is a performance bottleneck in multiprocessor operating system kernels, and presents the cost of contended inter-core communication on a modern multicore.

We re-implement the rmap and dcache to use lock-free data structures. The results show that even lock-free implementations do not scale well. Using results from kernel instrumentation, we show that the reason for poor scalability is that the amount of time spent waiting for cache line transfers increases with the number of cores. We show there is room for improvement by presenting the performance of our OpLog implementations (described in Chapters 5–7), which perform much better.

Using microbenchmarks, we measure the cost of contended inter-core communication as the number of cores varies. To put the cost of inter-core communication in perspective, we present the cost of several system calls, and show that the cost of a single inter-core cache line transfer can exceed the cost of commonly execute system calls.

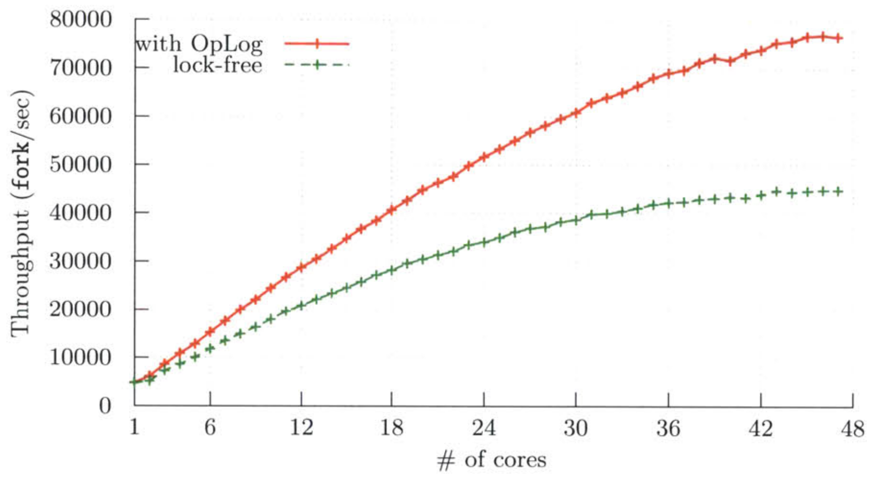
4.1 Example: rmap

We wrote a version of the rmap that replaces the interval trees with lock-free lists. Figure 4-1(a) shows the performance. The performance of this lock-free rmap reaches a plateau around 40 cores. The reason is not due to repeated failure and retry of the compare-exchange instructions, as one might fear under high contention. Instead, the reason is the cost of fetching a cache line that has been updated by another core. Figure 4-1(b) shows the percent of total execution cycles the **fork** benchmark spends waiting for cache lines in the lock-free rmap code. On 40 cores, roughly 35% of the execution time is spent waiting for cache line transfers.

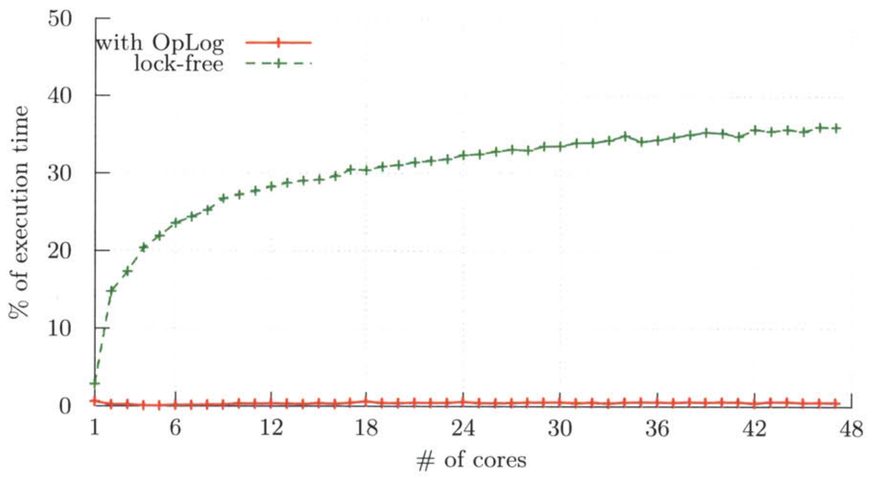
To show there is room for improvement, we also ran the benchmark on a modified version of the Linux kernel that implements the rmap using OpLog. The "with OpLog" line shows that eliminating communication can provide further benefit even after locks have been removed. Chapter 8 explores the performance of the OpLog-based rmap further.

4.2 Example: pathname lookup

To avoid performance collapse due to the spinlock, we modified the kernel to remove the spinlock in favor of a lock-free implementation. The lock-free version packs the generation count and reference count into a single cache line, and uses an atomic compare-and-exchange



(a) Performance.



(b) Percent of execution time spent waiting for cache line transfers in the rmap.

Figure 4-1: The **fork** benchmark performance.

instruction to conditionally increment the reference count. This lock-free implementation also scales poorly (see the "lock-free" line). The reason is that many cores try to modify the reference count at the same time, which causes contention on the corresponding cache line (i.e., increasingly high latencies for the compare-and-exchange). Figure 4-2(b) shows the percentage of total execution cycles the `stat` benchmark spends waiting for cache line transfers in the reference counting code. On six cores the lock-free implementation spends almost 80% of the time waiting for cache line transfers. On 48 cores the lock-free implementation spends 99% of the time waiting for cache line transfers.

This result shows that the lock-free implementation does not scale well, but one might wonder how much room for improvement is available. To answer this question we also plot the performance of an implementation that uses OpLog to implement the reference counter. This version uses a distributed reference counter [8] that is optimized using OpLog (which we describe in Chapter 6). The OpLog version increments a local reference count, and then checks if the generation count changed. If the generation count changed, the OpLog version decrements the local reference count and retries. The "with OpLog" line in Figure 4-2(a) scales nearly perfectly with the number of cores.

The large gap between the "with OpLog" line and the "lock-free" lines highlights the opportunity for improving the scalability of update-heavy data structures by avoiding contention on either locks or shared cache lines.

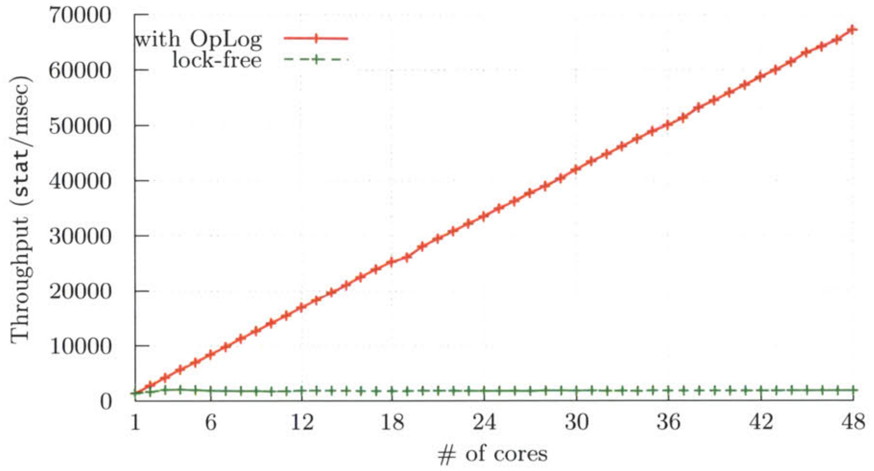
4.3 The cost of cache line contention

As we have shown, several workloads in Linux spend much of their time fetching cache lines, and the proportion of time spent waiting for cache lines increases with the number of cores. In this section we demonstrate that contended accesses to a single cache line result in a significant increase in the latency to access that cache line, and this latency can easily exceed the cost of a single system call.

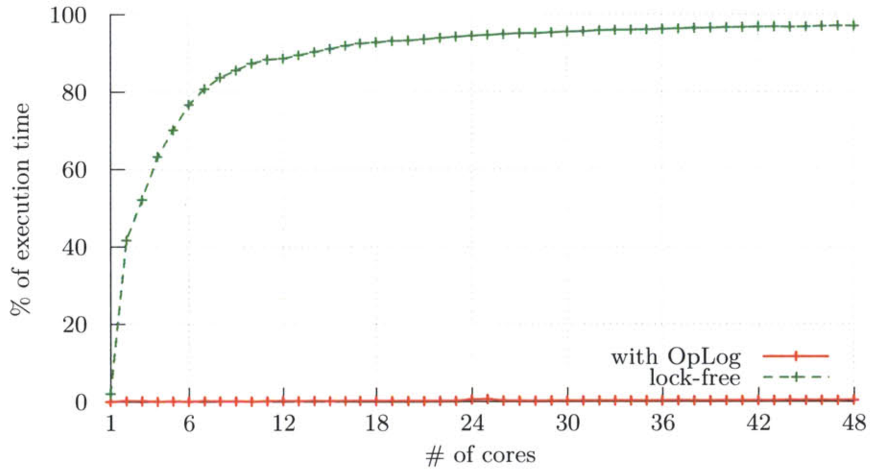
To understand how updating a single cache line scales, we measure its performance in isolation using a microbenchmark. This microbenchmark arranges for each core to simultaneously store an integer to the same memory location. The benchmark reports the average time it takes a core to execute the store, using a `cpuid` instruction to wait for completion. The results in Figure 4-3 show that as the number of writers increases from 2 to 48, the latency to execute the store increases from 182 to 8182 cycles. We also measured the average time to execute an atomic increment instead of a `mov` instruction. The atomic increment latencies were at most 1.5% longer than the `mov` latencies.

When several cores write the same cache line, the coherence protocol moves the line between cores. Conway et al. provide a detailed description of the caching subsystem of the Istanbul chips [9], which we summarize. Each core uses a local multi-level cache to provide fast access to recently used cache lines. A core can read from and write to its local L1 cache in a few cycles. Before a core modifies a cache line that other cores also cache, the cache coherence protocol invalidates the other copies. When a core tries to access a cache line that was recently updated by another core, cache coherence ensures the core accesses the most up-to-date version of the cache line, by copying the cache line from the other core's cache. Modern multicore computers implement directory-based coherence to track the location of cache lines [9, 14].

The cost of an inter-core cache miss depends on distance in the interconnect and on contention. Each chip has an internal interconnect for cache coherence within the chip, and



(a) Performance.



(b) Percent of execution time spent waiting for cache line transfers in reference counting.

Figure 4-2: The `stat` benchmark results.

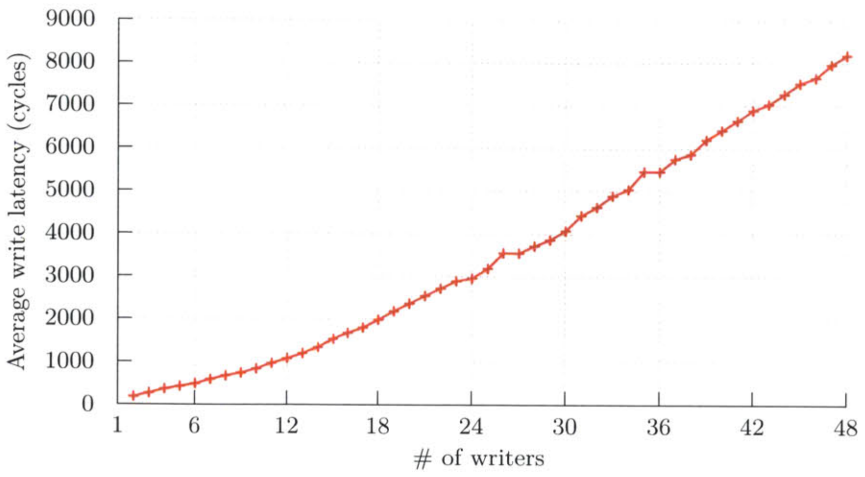


Figure 4-3: Average latency for writing a shared cache line, as a function of the number of simultaneously active cores.

System call	**Latency**
open	2618 cycles
close	770 cycles
mmap	849 cycles
dup	387 cycles

Figure 4-4: Single core latency of several Linux system calls.

an external interconnect that carries inter-chip cache coherence messages. When there is no other traffic, a core can read a cache line from L1 cache of a core on the same chip in 124 cycles. A cache line transfer between cores on different chips takes between 250 and 550 cycles depending on the distance. This architecture imposes relatively high latencies for cache line transfers between distant cores. We expect lower latencies when all cores are located on the same chip, but it is likely that communicating with a distant core will always take longer than communicating with a nearby one.

When multiple cores simultaneously try to write the same cache line, the cache line's home chip serializes the requests for ownership of the cache line. If the line is clean but cached by other cores, the home chip broadcasts an invalidate message; all chips must finish invaliding before the write can proceed. If the line is dirty, the home chip forwards the request to the cache line's previous owner. The previous owner sends the up-to-date value to the requesting core and invalidates the cache line from its own cache. When the requesting core receives the cache line, it sends a confirmation message to cache line's home chip. Once the home chip receives the confirmation it will process subsequent requests for the same cache line. The main variable factor in inter-core cache miss latency is serial request processing. The caching subsystem serializes inter-core cache misses for the same cache line (or even requests for different cache lines that happened to be issued at the same time).

To put the cost of contending for a single cache line in perspective, we compare the cost of accessing a remote cache line with the cost of several systems calls (see Figure 4-4). We measured these numbers by executing the system call many times on a single core running

Linux 3.9. A call to **dup**, which performs little work, is only 3.1 times more expensive than the 124 cycles for reading an uncontended cache line from another core's L1 on the same chip, and in the same ballpark as the 250–550 cycles for reading an uncontended cache line from another core on a different chip. The **open** call system call, which performs a lookup of /bin/sh, costs about 4.7–21 times as much as a single uncontended cache miss, depending whether it is a local or remote cache miss. An **open** call costs considerably less than writing a cache line when a dozen cores simultaneously update the same cache line.

Chapter 5

OpLog Design

The OpLog design aims to reduce communication for update intensive data structures. The main goal is to provide a general approach for reducing communication that is applicable to any data structure, without specific knowledge of the data structure, or the requirements for reconciling updates from different cores. This chapter describes the OpLog approach to achieving this goal and how this approach facilitates several useful optimizations. To help clarify the OpLog approach, the chapter concludes with an example of applying the approach the Linux rmap.

5.1 The OpLog approach

The general approach behind OpLog's design is to timestamp each update operation, append the operation to a per-core log, and to reconcile those logs when reading. Timestamps provide a solution to ordering updates from different cores. Logging updates makes OpLog agnostic to the data structure being optimized and allows developers to apply OpLog without writing any data structure specific code.

In the general case, OpLog must maintain the causal order of updates to a data structure. For example, consider a linked list data structure. If a process logs an insert operation on one core, migrates to another core, and logs a remove operation on that other core, the remove should execute after the insert. Instead of tracking precise causal dependencies through shared memory, OpLog relies on a system-wide synchronized clock to order updates,[1] by recording a timestamp for each log entry, and applying log updates in timestamp order. This allows OpLog to provide linearizability, thereby making it compatible with existing data structure semantics.

OpLog provides a general approach for optimizing communication that only requires developers to classify operations as read or update. The OpLog approach is based on a single shared data structure and per-core operation logs. Instead of executing an update operation immediately, OpLog appends the operation and a timestamp to a per-core log. Before a core reads the shared data structure, OpLog merges the per-core operation logs according to timestamps and then executes every operation in the merged list, producing an up-to-date shared data structure. In order to use OpLog, a developer only needs to specify which operations update the data structure, so OpLog knows to log them, and which operations read the data structure, so OpLog knows to merge and execute the logs.

[1]Modern Intel and AMD processors provide synchronized clocks via the RDTSC and RDTSCP instructions.

5.2 OpLog optimizations

The OpLog design facilitates several optimizations that reduce inter-core communication.

5.2.1 Batching updates

By deferring the execution of updates until a read operation, OpLog can apply multiple updates in a single batch, which is often less expensive than applying updates one-by-one. Moreover, batching can improve locality, since the data structure being updated remains in the cache of the processor that applies the updates, and for most data structures, executing an update accesses more cache lines than transferring the log entry for the update. This is similar to the locality exploited by flat combining [13], but OpLog can batch more operations by deferring updates for longer.

5.2.2 Absorbing updates

Explicitly logging each update is required for the general case in which the order of updates is significant, and in which update operations on each core cannot be aggregated. However, for many specific uses of OpLog, it is possible to take advantage of the semantics of the target data structure to absorb log operations, as we describe next. These optimization reduce the number of updates that have to be performed on the shared data structure, and reduce the space required by OpLog for storing per-core logs.

First, if two operations in a log cancel each other out—that is, applying both of them is equivalent to not applying any operations at all—OpLog removes both operations from the log. For example, consider a linked list. If OpLog is about to log a remove operation, but that core's log already contains an insert operation for that same item, OpLog can simply drop the insert log entry and not log the remove at all. Note that if any read operation was executed between the insert and the remove, that read would have applied all pending log operations, and remove would not find a pending insert in the log for absorption.

Second, if multiple operations can be equivalently represented by a single operation, OpLog combines the log entries into a single update. For example, consider a counter. Two operations that increment the counter by 1 are equivalent to a single operation that increments the counter by 2; thus, if OpLog is about to log an inc(1) operation, and the per-core log already contains an inc(n) operation, OpLog replaces inc(n) by inc($n+1$), and does not log the inc(1) at all.

5.2.3 Allocating logs

For many data structures, only a few instances of the data structure are contended and will benefit from queuing operations; the memory overhead of OpLog's per-core log for other instances of the data structure will be wasted. To avoid this memory overhead, OpLog dynamically allocates per-core logs only for recently used objects. If an object has not been accessed recently on a given core, OpLog revokes its log space on that core, and reuses the log space for other objects. To revoke the log space of an object, OpLog must apply all updates to that object, including updates from other cores, much like a read operation. After an object's log has been applied, it is safe to reuse its log space.

5.3 Example: a logging rmap

For concreteness we illustrate the OpLog approach using the rmap example from Chapter 3. There are three rmap operations that are relevant: adding a mapping to the rmap when a process maps a region into its address space, removing a mapping from the rmap when a process unmaps a region from its address space, and truncating a file, which reads the rmap to find all processes that map it. The most common rmap operations are processes adding and deleting mappings. Using OpLog, a programmer can achieve good scalability for these operations without modifying the rmap's interface for the rest of the system or the rmap's shared data structure.

With OpLog, a logging rmap consists of a shared rmap plus a per-core log of recent operations; there is one lock protecting the shared rmap, plus a lock for each per-core log. When a process maps or unmaps a page, it appends a map or unmap timestamped record to its local per-core log. As long as a process executes on the same core, map and unmap operations use the local core's log and lock, and do not require any cache line transfers.

When a process truncates a file, it acquires all of the per-core locks and the rmap lock, merges all per-core logs into a single log sorted by timestamps, applies all log records to the shared rmap in chronological order, and then uses the shared rmap to look up all of the processes that map the truncated file. Timestamped logs are a natural fit for this situation because they preserve the order of operations on different cores. Suppose core c_1 maps virtual address v_1 to physical address p_1, and then core c_2 unmaps v_1 from the same address space. When the rmap is later read, it is important that the state seen by the reader reflects the unmap occurring after the map.

By applying many operations in a single batch, OpLog amortizes the expense of acquiring exclusive use of the shared rmap's cache lines over many logged operations; this is faster than having each core separately write the shared rmap. Moreover, a programmer can take advantage of absorption to reduce the size of the log and the cost of applying log entries. In particular, if a core is about to log a removal of a region in its local log, it can check if there is an earlier addition of the same region in its log. If there is, it can remove the addition from its log and not log the removal. These two operations do not have to be applied to the shared rmap at all.

Chapter 6

The OpLog Library

Realizing the potential scaling benefits of the logging approach described in Chapter 5, along with its potential optimizations, requires designing a convenient programmer interface to address the following two challenges:

- How can developers apply OpLog without changing existing interfaces or data structures?
- How can developers expose the optimization opportunities like absorption to OpLog without having to customize OpLog for each use case?

The OpLog library allows programmers to take an implementation of a data structure, and automates most of the transformation to an update-scalable implementation with per-core logs. The programmer implements type-specific optimizations to benefit, for example, from absorption. The rest of this section describes the OpLog library in more detail.

6.1 OpLog API

To use OpLog, a developer starts with the uniprocessor implementation of a data structure and determines which operations update the data structure and which operations read the data structure. The programmer must specify how to execute an update operation by overloading the `Op::exec()` method shown in Figure 6-1, and must modify the implementation to `queue` the update operation. The programmer must also modify read operations to first call `synchronize` before executing the read operation. Programmers can also overload other methods shown in that Figure to implement type-specific optimizations.

For clarity, the examples in this dissertation are shown in C++, because it provides clear syntax for overloading. In practice, OpLog supports both C and C++, and we use the C version in the Linux kernel.

Figure 6-2 shows how OpLog can be used to implement the logging rmap design of Chapter 3. The interface between the kernel and an rmap consists of three methods: `add` to add a mapping to an rmap, `rem` to remove a mapping from an rmap, and `truncate`. The kernel invokes the `truncate` method, for example, when truncating a file or evicting it from the buffer cache. The methods operate on a shared data structure, which in this example is the interval tree `itree_`. The kernel developer implements the `add` and `rem` methods by queuing them in OpLog.

Method call	Semantics
`Object::queue(Op* op)`	add `op` to a per-core log, implemented by a Queue object
`Object::synchronize()`	acquire a per-object lock, and call apply on each per-core Queue object
`Object::unlock()`	release the per-object lock acquired by `synchronize()`
`Queue::append(Op* op)`	append `op` to the per-core Queue object
`Queue::apply()`	sort and execute the operations from Queues
`Queue::try_absorb(Op* op)`	try to absorb an op
`Op::exec()`	execute the operation

Figure 6-1: OpLog interface overview.

When an application must perform a read operation on the shared object,[1] it uses the `synchronize` method, which returns a locked version of the shared object. The `truncate` method invokes `synchronize` to ask OpLog to produce a locked and up-to-date version of the shared `itree_` interval tree object. Once the interval tree is up-to-date, `truncate` can perform truncate on the interval tree as usual.

The `synchronize` method is provided by OpLog, and hides from the developer the per-core logs and the optimizations to apply per-core logs. By default, the `synchronize` method acquires locks on all per-core queues, sorts queued operations by timestamp, calls `try_absorb` on each operation, and calls `exec` for each operation. The implementation of `synchronize` ensures that only one core applies the logs when several invoke `synchronize`. To avoid consuming too much memory, OpLog invokes `synchronize` if the length of a core's local log exceeds a specified threshold. In the base `Queue` class `try_absorb` does nothing. §6.2 describes how it is used to implement absorption.

Figure 6-2 illustrates that OpLog hides most of the logging infrastructure from the programmer: the programmer does not have to worry about per-core logs, per-core locks, iterating over per-core logs to apply operations, etc. The developer implements an `Op` sub-type for a mapping addition and a mapping removal operation, and creates these when adding a mapping in `add` and when removing a mapping in `rem`.

6.2 Type-specific optimizations

Programmers can optimize the space and execution overhead of queues further, for example, for operations that do not need to be executed in order. OpLog enables such optimizations by supporting type-specific queues through overloading.

Consider a reference counter supporting 3 operations: increment (`inc`), decrement (`dec`), and reading the count (`read`). If OpLog queues increment and decrement operations, the order that OpLog executes the operations does not matter. Furthermore the synchronized value of the reference counter would be the same if, instead of maintaining per-core queues, OpLog executed operations on per-core counters. To exploit these properties a developer can implement type-specific queues.

The programmer chooses what data structure to use to represent a per-core log by specifying the `Queue` class that OpLog should use. By default OpLog uses an overloaded version of the C++ queue class to represent a log. OpLog calls the `queue` method to insert an `Op`. The `queue` method acquires a per-core lock, invokes `append` on the `Queue` class, and releases the lock. The default `Queue` class used by OpLog timestamps each `Op`, to be able to

[1]We will use "object" to refer to an instance of a data structure type.

```
struct Rmap : public QueuedObject<Queue> {
public:
  void add(Mapping* m) { queue(AddOp(m)); }
  void rem(Mapping* m) { queue(RemOp(m)); }

  void truncate(off_t offset) {
    synchronize();
    // For each mapping that overlaps offset..
    interval_tree_foreach(Mapping* m, itree_, offset)
      // ..unmap from offset to the end of the mapping.
      unmap(m, offset, m->end);
    unlock();
  }

private:
  struct AddOp : public Op {
    AddOp(Mapping* m) : m_(m) {}
    void exec(Rmap* r) { r->itree_.add(m_); }
    Mapping* m_;
  }

  struct RemOp : public Op {
    RemOp(Mapping* m) : m_(m) {}
    void exec(Rmap* r) { r->itree_.rem(m_); }
    Mapping* m_;
  }

  IntervalTree<Mapping> itree_;
}
```

Figure 6-2: Using OpLog to implement a communication-efficient rmap.

sort the operations in the per-core queues later to form a serial log of all operations. OpLog timestamps an entry using the cycle counter on each core, which modern hardware keeps synchronized.

To implement counters efficiently using OpLog, the programmer implements the `CounterQueue` class as a subtype of `Queue`, as shown in Figure 6-3. When an application invokes `inc` or `dec` to queue an operation, OpLog will invoke the `append` method of the `Queue` class. The `CounterQueue` overloads this method to invoke `exec` directly on the operation, which then updates a per-core value. When `read` invokes `synchronize`, OpLog invokes the `apply` method for each per-core queue. The `CounterQueue` overloads `apply` to add the per-core values to a shared counter. Executing operations directly on per-core counters reduces storage overhead and in most cases performs better than queuing operations. Furthermore, the dynamic allocation of queues ensure that if a counter is not heavily used, the space overhead is reduced to just the shared counter.

The OpLog version of a distributed counter has the same performance benefits as previous distributed counters, but is more space efficient when it is not contended. This OpLog distributed counter is the one that Chapter 4 measured (see Figure 3-3).

The logging rmap design can benefit from type-specific queues to realize the benefit of absorption. To add regions, the developer overloads `append` to append the operation to a per-core queue. To remove a region, the overloaded `append` method checks if the region was added to the current core's list and if so removes region directly from the list; otherwise, it queues the operation. Thus, in the common case, if files are opened and closed on the same core, those operations are absorbed.

For rmap, it is not uncommon that one core maps a file and another core unmaps the file; for example, the parent maps a file and a child unmaps that file from its address space on another core. To deal with this case, the rmap use of OpLog also performs absorption during `synchronize` by overriding `try_absorb`. If the operation passed to `try_absorb` is a `RemOp` and OpLog has not yet executed the `AddOp` that inserts the `Mapping`, `try_absorb` removes both operations from the queue of operations. `try_absorb` can tell if OpLog executed a particular `AddOp` by checking if the `Mapping` member variable has been inserted into an interval tree.

```
struct Counter : public QueuedObject<CounterQueue> {
  struct IncOp : public Op {
    void exec(int* v) { *v = *v + 1; }
  }

  struct DecOp : public Op {
    void exec(int* v) { *v = *v - 1; }
  }

  void inc() { queue(IncOp()); }
  void dec() { queue(DecOp()); }

  int read() {
    synchronize();
    int r = val_;
    unlock();
    return r;
  }

  int val_;
}

struct CounterQueue : public Queue {
  void push(Op* op) { op->exec(&val_); }

  static void apply(CounterQueue* qs[], Counter* c) {
    for_each_queue(CounterQueue* q, qs)
      c->val_ += q->val_;
  }

  int val_;
}
```

Figure 6-3: Using OpLog to implement a distributed reference counter.

Chapter 7

OpLog Implementation

We implemented two versions of OpLog. One is a user-level implementation in C++ and the other is a kernel implementation in C. Both implementations are approximately 1000 lines of code. The C version of OpLog takes advantage of GNU C nested functions to simplify the process of defining update operations. A developer defines an update operation by delimiting existing update code with two OpLog macros that define a closure using nested functions and pass the closure to `queue`. We report on the changes to use OpLog in individual Linux subsystems in Section 8.4.

Both implementations use a maximum length of 128 entries for a per-core queue. Performance is not very sensitive to the maximum queue length, unless it is very large or very small. Large maximum queue lengths can lead to memory exhaustion. Short maximum queue lengths reduce the performance benefits of batching.

To implement dynamic log space allocation, OpLog maintains a cache for each type (*e.g.*, one for objects of type rmap), implemented as a per-core hash table. The hash table acts like a direct-mapped cache: only one object with a given hash value can be present in the cache. Only an object present in a core's hash table can have a per-core queue. The net effect is that heavily used objects are more likely to have per-core queues than infrequently used objects.

To queue an operation, OpLog hashes the object (*e.g.*, its pointer) to determine the hash table bucket, and checks if the queue currently in the bucket is for that object. If it is, OpLog queues the operation. If there is a hashing conflict, OpLog first synchronizes the object currently associated with the bucket, which removes any queues associated with the object. Once the object is synchronized, it can be removed from the hash table. This scheme works well because recently accessed objects will be in the hash table, perhaps replacing less-recently-accessed objects.

For every object, OpLog tracks which cores have a per-core queue for that object. It does so using a bitmap stored in the object; if a core's bit is set, then that core has a per-core queue. The bitmap is protected by a lock. To synchronize an object, OpLog acquires the bitmap's lock, merges the queues for each core ID in the bitmap, applies the ordered operations, and clears the bitmap.

Chapter 8

Evaluation

This chapter evaluates the OpLog approach and the implementation of OpLog library by answering the following questions:

- Does OpLog improve whole-application performance? Answering this question is challenging because full applications stress many kernel subsystems; even if some kernel data structures are optimized using OpLog, other parts may still contain scalability bottlenecks. Nevertheless, we show for two applications that OpLog reduces inter-core communication and improves performance. (Section 8.1)
- How does OpLog affect performance when a data structure is read frequently? We answer this question by analyzing the results of a benchmark that stresses the rmap with calls to `fork` and `truncate`. (Section 8.2)
- How important are the individual optimizations that OpLog uses? To answer this question we turn on optimizations one by one, and observe their effect. (Section 8.3)
- How much effort is required of the programmer to use OpLog compared to using per-core data structures? We answer this question by comparing the effort required to apply OpLog and the per-core approach to three subsystems in the Linux kernel. (Section 8.4)

8.1 Application performance

Since OpLog is focused on scaling updates to data structures that have relatively few reads, we focus on workloads that generate such data structure access patterns; not every application and workload suffers from this kind of scalability bottleneck. In particular, we use two applications from MOSBENCH [6]: the Apache web server and the Exim mail server. These benchmarks stress several different parts of the kernel, including the ones that the microbenchmarks in Chapter 3 stress.

8.1.1 Apache web server

Apache provides an interesting workload because it exercises the networking stack and the file system, both of which are well parallelized in Linux. When serving an HTTP request for a static file, Apache `stat`s and `open`s the file, which causes two pathname lookups in the kernel. If clients request the same file frequently enough and Apache is not bottlenecked by

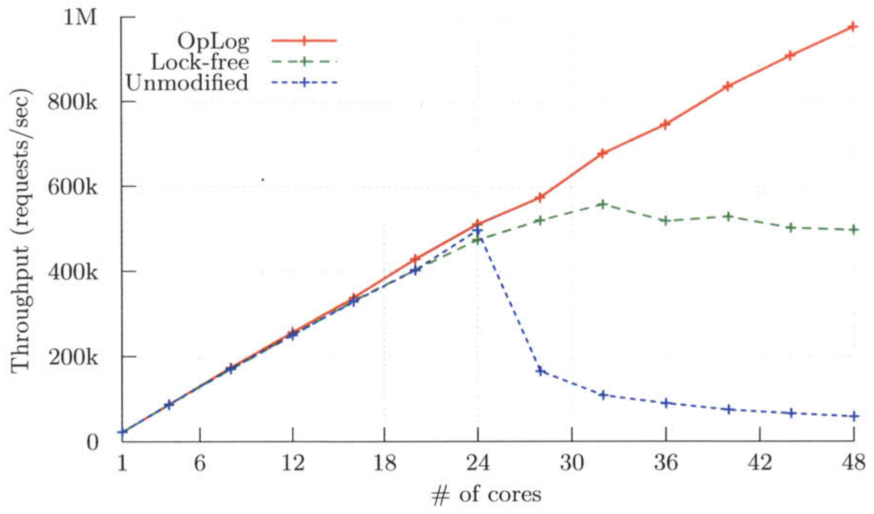


Figure 8-1: Performance of Apache.

the networking stack, we expect performance to eventually bottleneck on reference counting the file's **dentry**, in the same way as the pathname microbenchmark in Chapter 3 (see Figure 3-3).

We configured the Apache benchmark to run a separate instance of Apache on each core and benchmarked Apache with HTTP clients running on the same machine, instead of over the network. This configuration eliminates uninteresting scalability bottlenecks and excludes drivers. We run one client on each core. All clients request the same 512-byte file.

Figure 8-1 presents the throughput of Apache on three different kernels: the unmodified 3.9 kernel and the two kernels used in the pathname example in Chapter4. The x-axis shows the number of cores and the y-axis shows the throughput measured in requests per second. The line labeled "Unmodified" shows the throughput of Apache on Linux 3.9. The kernel acquires a spinlock on the **dentry** of the file being requested in order to atomically check a generation count and increment a reference counter (see Chapter 3). This line goes down after 24 cores due to contention on the spinlock for a **dentry**.

The line labeled "Lock-free" shows the throughput after we refactored the code to replace the lock with a conditional compare-and-exchange instruction (as described in Chapter 3). This line levels off for the same reasons as the line in Figure 4-2, but at more cores because Apache does other work in addition to checking the generation count and incrementing the reference count of a **dentry**.

The line labeled "OpLog" shows the throughput when using reference counters implemented with OpLog (as described in Chapter 5), which scales perfectly with increasing core counts. The distributed counters built with OpLog apply increments and decrements to per-core counters, avoiding any inter-core communication when incrementing and decrementing a reference counter.

To verify that OpLog reduces inter-core communication for the Apache benchmark we counted inter-core cache misses using hardware event counters. On 48 cores, unmodified Linux has 360 inter-core cache misses per request. The lock-free implementation of pathname lookup reduces inter-core cache misses to 30 per request. Using OpLog to implement pathname lookup eliminates all inter-core cache misses for the Apache benchmark.

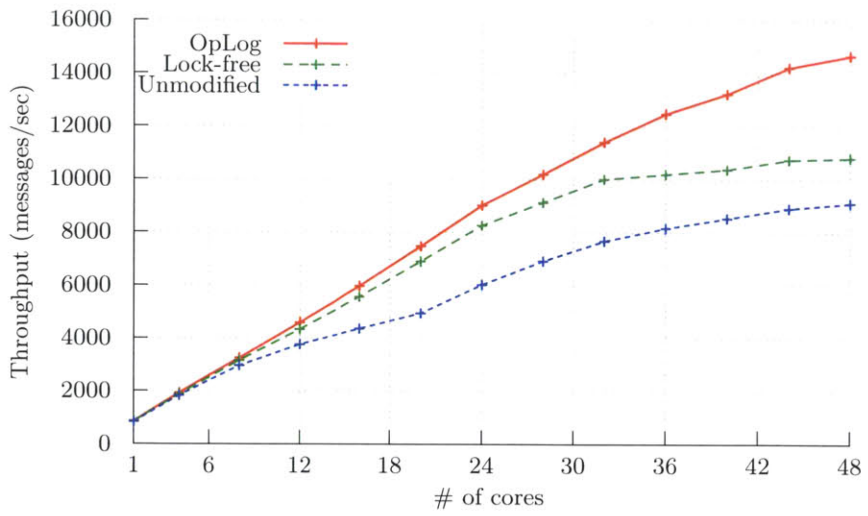


Figure 8-2: Performance of Exim.

8.1.2 Exim mail server

We configure Exim [1] to operate in a mode where a single master process listens for incoming SMTP connections via TCP and forks a new process for each connection. The new process accepts the incoming mail, queues it in a shared set of spool directories, appends it to the per-user mail file, deletes the spooled mail, and records the delivery in a shared log file. Each per-connection process also forks twice to deliver each message.

The authors of MOSBENCH found that Exim was bottlenecked by per-directory locks in the kernel when creating files in spool directories. They suggested avoiding these locks and speculated that in the future Exim might be bottlenecked by cache misses on the rmap in `exit` [6]. We run Exim with more spool directories to avoid the bottleneck on per-directory locks.

Figure 8-2 shows the results for Exim on three different kernels: the unmodified 3.9 kernel and the two kernels that the fork and exit microbenchmark uses in Chapter 3 (see Figure 4-1). The results in Figure 8-2 show that the performance of Exim plateaus on both the unmodified kernel and the kernel with a lock-free rmap. Exim scales better on the kernel with OpLog, but at 42 cores its performance also starts to level off. At 42 cores, the performance of Exim is becoming bottlenecked by zeroing pages that are needed to create new processes.

We used hardware event counters to measure how much OpLog reduces inter-core communication. The unmodified kernel incurs 23358 inter-core cache misses per message on 48 cores. The lock-free rmap implementation reduces inter-core cache misses by about 15%, or 19695 per message. The OpLog rmap reduces inter-core core cache misses to 12439, or almost half the number of inter-core cache misses the unmodified kernel incurs.

8.2 Read intensive workloads

One downside of using OpLog for a data structure is that read operations can be slower, because each read operation will have to execute `synchronize` to collect updates from all cores, and OpLog will not be able to take advantage of batching or absorption.

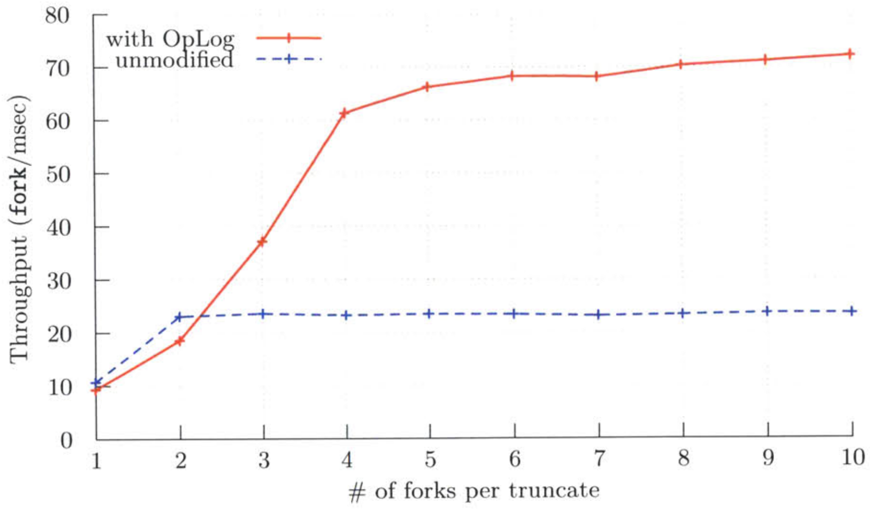


Figure 8-3: Performance of the **fork-truncate** benchmark for varying truncate frequencies.

To help understand how reads affect the performance of OpLog, we wrote a version of the **fork** benchmark that calls **truncate** to trigger reads of the shared rmap. This **fork-truncate** benchmark creates 48 processes, one pinned to each core, and 48 files, and each process **mmap**s all of the 48 files. Each process calls **fork** and the child process immediately calls **exit**. After a process calls **fork** a certain number of times, it truncates one of the **mmap**ed files by increasing the file length by 4096 bytes, and then decreasing the file length by 4096 bytes. A runtime parameter dictates the frequency of truncates, thus controlling how often the kernel invokes **synchronize** to read a shared rmap. The benchmark reports the number of **fork**s executed per millisecond. We ran the benchmark using truncate frequencies ranging from once per fork to once per ten forks.

Figure 8-3 shows the results of the **fork-truncate** benchmark. The unmodified version of Linux outperforms OpLog by about 10% when the benchmark truncates files after one call to **fork** and by about 25% when truncating after two calls to **fork**. However, we find that even when **truncate** is called once every three forks, OpLog outperforms the unmodified kernel. This suggests that OpLog improves performance even when the data structure is read periodically.

8.3 Breakdown of techniques

We evaluate the individual OpLog optimizations from Chapter 5 with user-level microbenchmarks and with measurements from a running Linux kernel.

8.3.1 Absorbing updates

We evaluate the benefit of implementing absorbing optimizations with user-level microbenchmarks using the OpLog C++ implementation. We implement a singly-linked list using OpLog and stress the list using a benchmark that adds and removes entries from concurrently. Benchmarking a singly-linked list is interesting because it allows us to compare OpLog versions of the linked list with a lock-free design [12].

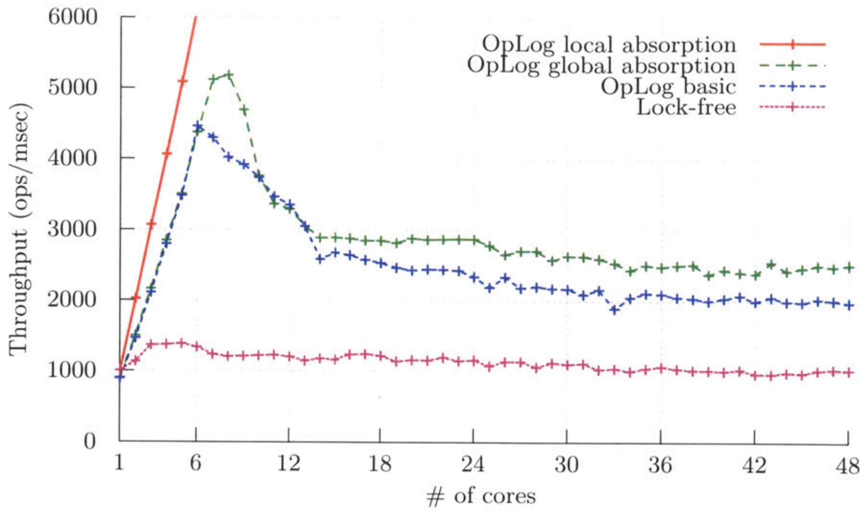


Figure 8-4: Performance of a lock-free list implementation and three OpLog list implementations. "OpLog local absorption" scales linearly and achieves 49963 operations per millisecond on 48 cores.

We study how much each technique from Chapter 5 affects performance using three different OpLog list implementations. The first is an OpLog list without any type-specific optimizations ("OpLog basic"). Comparing throughput of OpLog basic to the throughput of the lock-free list demonstrates how much batching list operations can improve performance. The other two list implementations demonstrate how much absorption improves performance. The second OpLog list uses type-specific information to perform absorption during `synchronize`. `synchronize` executes an add operation only if there is not a remove operation queued for the same element ("OpLog global absorption"). The third list takes advantage of the fact that the operations do not have to be executed in order: it performs absorption immediately when a thread queues an operation ("OpLog local absorption").

We wrote a microbenchmark to stress each list implementation. The benchmark instantiates a global list and creates one thread on each core. Each thread executes a loop that adds an element to the list, spins for 1000 cycles, removes the element from the list, and spins for another 1000 cycles. We count each iteration of the loop as one operation. We chose to delay 1000 cycles to simulate a list that is manipulated once every system call.

If operations on the lock-free list cause cache contention, we would expect OpLog basic to provide some performance improvement. OpLog global absorption should perform better than basic OpLog, because it ends up performing fewer operations on the shared list. OpLog local absorption should scale linearly, because when a thread queues a remove operation, OpLog should absorb the preceding add operation executed by the thread.

Figure 8-4 presents the results. The lock-free line shows that the throughput of the lock-free list peaks at 3 cores with 1364 operations per millisecond. OpLog basic throughput increases up to 4463 operations per millisecond on six cores, then begins to decrease until 1979 operations per millisecond on 48 cores. Throughput starts to decrease for more than 6 cores because the benchmark starts using more than one CPU node and inter-core communication becomes more expensive. OpLog global absorption throughput peaks at 5186 operations per millisecond on eight cores. Similar to OpLog basic, OpLog global absorption throughput decreases with more cores and is 2516 operations per millisecond on 48 cores. OpLog local

	Shared counter	**Per-core counters**	**OpLog counters**
`dentry` size	192 bytes	576 bytes	196 bytes
Total `dentry` storage	2.6 Gbytes	8.0 Gbytes	2.7 Gbytes

Figure 8-5: Storage overhead of `dentry`s using a single shared reference counter, per-core reference counters, and OpLog log caching. Measurements are for a 48-core machine with 15 million cached `dentry`s.

absorption throughput scales linearly and achieves 49963 operations per millisecond on 48 cores.

8.3.2 Allocating logs

OpLog allocates per-core data structures dynamically to avoid allocating them when an object is not heavily used. To gauge the importance of this technique we compared the space overhead for `dentry`s with per-core reference counters to OpLog's reference counters with log caching.

Figure 8.3.2 summarizes the measurements for a 48-core machine after it had been running for several days and the kernel filled the `dcache` with 15 million entries. An unmodified `dentry` with a single shared counter is 192 bytes. Per-core 8-byte counters require 384 bytes of storage and increase the `dentry` size by 200% to 576 bytes. The OpLog reference counter adds an extra 4 bytes of metadata per-`dentry`. The OpLog log cache size is limited to 4096 entries per-core.

Per-core counters use 5.4 more Gigabytes than the shared counter, while OpLog counters used only 0.1 more Gigabytes. These results suggest OpLog's log caching is an important optimization for conserving storage space.

8.4 Programmer effort

To compare the programmer effort OpLog requires to the effort per-core data structures require we counted the number of lines of code added or modified when applying OpLog and per-core data structures to three Linux kernel subsystems. We implemented per-core versions of the rmap and inotify, and used the existing per-core version of the open files list. The lines of code for the OpLog implementations do not include type-specific optimizations, only calls to `synchronize` and `queue`.

Figure 8.4 presents the number of lines of code. The rmap required the most lines of code for both the per-core and OpLog implementations, because the rmap is not very well abstracted, so we had to modify much of the code that uses the rmap. The inotify and the open-files implementations required fewer code changes because they are well abstracted, so we did not need to modify any code that invoked them.

The OpLog implementations of all three subsystems required fewer changes than the per-core versions. The OpLog implementations do not modify operations, except for adding calls to `synchronize` or `queue`. After calling synchronize, the existing code for read operations can safely access the shared data structure. The per-core implementations, on the other hand, replace every update operation on the shared data structure with an operation on a per-core operation and every read operation with code that implements a reconciliation policy.

Subsystem	Per-core LOC	OpLog LOC
rmap	473	45
inotify	242	11
open files list	133	8

Figure 8-6: The number of lines of code (LOC) required by per-core implementations and OpLog implementations.

The number of lines of code required to use OpLog is few compared to using per-core data structures. This suggests that OpLog is helpful for reducing the programmer effort required to optimize communication in update-heavy data structures.

Chapter 9

Related work

OpLog targets cases where shared data structures are necessary. One way to achieve scalability is to avoid this problem in the first place, by implementing kernel subsystems in a way that avoids sharing, or changing the kernel interface in a way that allows an implementation not to share. For example, by default the threads in a Linux process share file descriptors, which can cause contention when threads create or use file descriptors [5]. However, if a process creates a thread with the `CLONE_FILES` flag, the new thread has its own space of file descriptors; this can reduce sharing and the associated contention. This paper focuses on the cases where the Linux kernel developers have decided that sharing is required.

OpLog is in spirit similar to read-copy update (RCU) [19], which is heavily used in the Linux kernel [20]. OpLog, however, targets update-intensive data structures, while RCU targets read-intensive data structures. OpLog, like RCU, provides a convenient infrastructure to optimize the performance of kernel data structures. Like RCU, OpLog does not solve a problem directly, but provides programmers with a helpful library. Like RCU, OpLog is focused on certain workloads; OpLog's target is write-dominated workloads, which RCU does not address.

Flat combining is an approach to improve performance of data structures that are heavily updated [13], but, unlike OpLog, it applies *all* operations, and applies them *eagerly*, which causes cache line contention, and thus scales less well than OpLog. In flat combining, a thread posts its operation in a per-core set of pending operations. The thread that succeeds in acquiring the lock becomes the "combiner." The combiner applies all pending operations to the locked data structure, posts the result of each operation, and releases the lock. By having a single thread perform all operations, flat combining avoids lock contention and can improve locality because one thread applies all operations to the data structure. OpLog delays applying updates until needed by a read operation, which gives it a higher degree of batching and locality than flat combining. The delaying also enables OpLog to perform several optimizations, such as absorption (e.g., if a `remove` cancels out an earlier `insert` in the same per-core log) and batching, and therefore scale better (e.g., near-ideal scaling if most operations are absorbed). In flat combining, every operation forces all pending updates to be applied (or waits for another core to apply them), making it a poor fit for update-heavy workloads. OpLog can defer updates in part because it uses timestamps to establish the global order in which operations must be applied.

Tornado's Clustered Objects [11], which were also used in K42 [3], provide a convenient programming abstraction that allows programmers to think in terms of single virtual object, but which can be instantiated as a partitioned, distributed, or replicated object at runtime, depending on the workload. In particular, a clustered object can instantiate a per-core *representative*, which mirrors the approach of using per-core instances described in the introduction, to improve scalability. Clustered objects require programmers to manually implement a strategy for keeping the per-core representative data structures consistent, such as an invalidation and update protocol [11]. OpLog, on the other hand, implements a consistency strategy for any data structure, based on logging and applying updates, which works well for update-heavy workloads. By focusing on a specific consistency strategy, OpLog is able to implement several important optimizations, such as batching and absorption. OpLog shares some implementation techniques with Clustered Objects; for example, OpLog's per-core hash table to avoid using per-core logs for all object instances bears similarities to the per-processor translation tables in Clustered Objects, which serve a similar purpose.

OpLog can be viewed as a generalization of distributed reference counter techniques. In fact, applying OpLog to a reference counter creates a distributed counter similar to Refcache [8]. OpLog makes it easy for programmers to apply the same techniques to data structures other than counters.

OpLog borrows techniques from distributed systems, but applies them in the context of shared-memory multiprocessor kernels. OpLog's approach of having a per-core log of operations, which must be merged to provide a synchronized global view of a data structure, has similarities with Bayou's per-device operation logs [22] for disconnected applications. OpLog also borrows Bayou's observation that certain logged operations can be re-ordered.

Also taking a page from distributed systems, the designers of Barrelfish [4] and Fos [23] argue that multicore processors should be viewed as distributed systems and all communication should be made explicit because it can be costly. They have proposed kernel designs where by default each core runs an independent kernel that does not share data with other cores. The Barrelfish multikernel optimizes communication between kernels using a fast message-passing library that can batch, and using a multicast tree for TLB shoot down. OpLog is a library for traditional shared-memory multiprocessor kernels, and optimizes inter-core communication when sharing is required.

Chapter 10

Discussion and Conclusion

This dissertation describes the communication bottleneck and presents a novel technique for optimizing a class of update-heavy communication bottlenecks that occur in multiprocessor operating system kernels. The OpLog library is an implementation of this technique and demonstrates that optimizing update-heavy communication bottlenecks improves application performance and requires minimal programmer effort. This chapter speculates on the future importance of OpLog; how the OpLog implementation and API might be extended and improved; and considerations for hardware architects trying to optimize communication.

10.1 Future importance of OpLog

To explore whether it is likely that there will be more opportunities to apply OpLog we performed two studies on the Linux kernel source code. The first study measures the length of serial sections in the Linux kernel and the second study looks at the usage of RCU in the Linux kernel over time.

To measure serial section length, we compiled the 3.8 kernel with the default 64-bit $x86$ configuration, and searched the kernel object code for calls to spin lock acquisition functions. When we located a call to acquire a lock, we stepped forward through the object code and counted instructions until the next call to an unlock function.

Figure 10-1 is a histogram presenting the length of critical sections in Linux version 3.8. The x-axis is the length of the serial section measured in instruction count and the y-axis is the number of serial sections that have that length. The bar labeled "call" is a count of serial sections that contain a call instruction and therefore are relatively expensive. About 70% of all serial sections contain call instructions, which indicates Linux still has many long serial sections. With many long serial sections it is likely that as kernel developers parallelize them they will uncover communication bottlenecks, some of which could be optimized with OpLog. For example, Linux maintains a list of directory entries for each directory. The kernel updates a directory's list when a file is created or deleted, and reads the list only when an application reads the directory entries. The creation and deletion code paths modify the list within relatively serial sections.

Figure 10-2 shows that the number of uses of the RCU API has increased from none in kernel version 2.5.0, to 10268 in kernel version 3.8. The results show that developers actively applying RCU which is an indication that they are implementing highly parallel new subsystems or refactoring old subsystems to use RCU-style lock-free algorithms. This

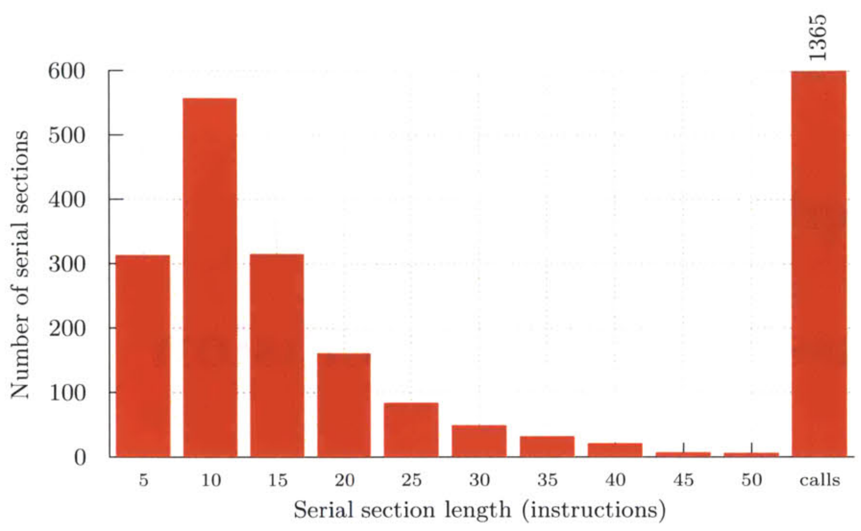


Figure 10-1: Serial section length in the Linux kernel.

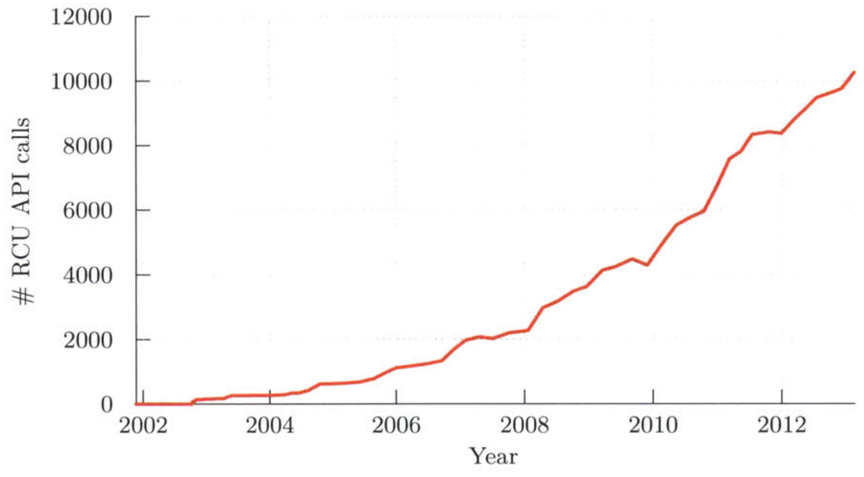


Figure 10-2: RCU API usage in the Linux kernel.

suggests that kernel developers will continue to parallelize the kernel and possibly uncover more communication bottlenecks.

The number of long serial sections and active usage of RCU by kernel developers suggests that Linux developers will continue to parallelize the kernel. This effort could uncover communication bottlenecks and we expect OpLog to be a useful tool in optimizing the bottlenecks caused by update-heavy data structures.

10.2 Extending OpLog

A future direction of research is to extend OpLog with other communication optimizing techniques. For example, OpLog could exploit locality by absorbing state in near-by cores first, and arranging the cores in a tree topology to aggregate the state hierarchically. This design can reduce the latency of executing `synchronize`.

Extending the OpLog API presented in Chapter 6 might improve performance of some uses of OpLog. For example, a version of `synchronize` that allows concurrent updates to per-core logs could be useful to inotify, because it would allow cores to append notifications operations to per-core logs while a reading core was executing synchronize and popping the first notification from the shared notification queue.

10.3 Optimizing communication with hardware

The machine used in the paper implements the widely used MOESI directory-based cache coherence protocol. The exact details of the cache-coherence protocol don't matter that much for the main point of the paper. For example, using an update protocol instead of an invalidation protocol just shifts the inter-core communication cost from a load to a store. Improvements in cache-coherence protocols, however, could change the exact point at which the techniques are applicable. One interesting area of research to explore is to have the cache-coherence protocol expose more state or new operations to allow software to optimize inter-core communication. For example, it would be useful to have a load instruction that only peeks at a cache-line and leaves it in exclusive mode in the remote core's cache, instead of changing the cache line from exclusive to shared.

10.4 Conclusion

Data structures that experience many updates can pose a scalability bottleneck to multicore systems, even if the data structure is implemented without locks. Prior approaches to solving this problem require programmers to change their application code and data structure semantics to achieve scalability for updates. This paper presented OpLog, a generic approach for scaling an update-heavy workload using per-core logs along with timestamps for ordering updates. Timestamps allow OpLog to preserve linearizability for data structures, and OpLog's API allows programmers to preserve existing data structure semantics and implementations. Results with a prototype of OpLog for Linux show that it improves throughput of real applications such as Exim and Apache for certain workloads. As the number of cores in systems continues to increase, we expect more scalability bottlenecks due to update-heavy data structures, which programmers can address easily using OpLog.

Appendix A

Spinlock Model

Some operating systems rely on spinlocks for serialization that can collapse when contended. For example, the Linux kernel uses ticket spin locks that can cause dramatic collapse in the performance of real workloads, especially for short critical sections. This appendix explains the nature and sudden onset of collapse with a Markov-based performance model.

A.1 The Linux spinlock implementation

For concreteness we discuss the ticket lock used in the Linux kernel, but any type of non-scalable lock will exhibit the problems shown in this section. Figure A-1 presents simplified C code from Linux. The ticket lock is the default lock since kernel version 2.6.25 (released in April 2008).

An acquiring core obtains a ticket and spins until its turn is up. The lock has two fields: the number of the ticket that is holding the lock (`current_ticket`) and the number of the next unused ticket (`next_ticket`). To obtain a ticket number, a core uses an atomic increment instruction on `next_ticket`. The core then spins until its ticket number is current. To release the lock, a core increments `current_ticket`, which causes the lock to be handed to the core that is waiting for the next ticket number.

If many cores are waiting for a lock, they will all have the lock variables cached. An unlock will invalidate those cache entries. All of the cores will then read the cache line. In most architectures, the reads are serialized (either by a shared bus or at the cache line's home or directory node), and thus completing them all takes time proportional to the number of cores. The core that is next in line for the lock can expect to receive its copy of the cache line midway through this process. Thus the cost of each lock hand off increases in proportion to the number of waiting cores. Each inter-core operation takes on the order of a hundred cycles, so a single release can take many thousands of cycles if dozens of cores are waiting. Simple test-and-set spin locks incur a similar $O(N)$ cost per release.

A.2 Performance

We exercised the Linux spinlock implementation using two microbenchmarks. The first microbenchmark uses a single lock, spends a fixed number of cycles inside a serial section protected by the lock, and a fixed number of cycles outside of that serial section. The serial section always takes 400 cycles to execute, but the non-serial section varies from 12.5 Kcycles to 200 Kcycles. When a lock becomes contended we expect parallel speedup to collapse

```
struct spinlock_t {
  int current_ticket;
  int next_ticket;
}

void spin_lock(spinlock_t *lock)
{
  int t =
    atomic_fetch_and_inc(&lock->next_ticket);
  while (t != lock->current_ticket)
    ;   /* spin */
}

void spin_unlock(spinlock_t *lock)
{
  lock->current_ticket++;
}
```

Figure A-1: Pseudocode for ticket locks in Linux.

because the cost of acquiring the lock will increase with the number of contending cores and come to dominate the cost of executing the 400 cycle serial section.

Figure A-2 shows the speedup of the ticket spinlock microbenchmark. Speedup is calculated by dividing the throughput on N cores by the throughput on one core. Speedup of all three benchmark configurations eventually collapse. One surprising result is that collapse occurs for fewer cores than expected. For example, with a 3.2% of execution in a serial section, one might expect collapse to start around 31 cores (1.0/0.032) when there is significant chance that many cores try to acquire the lock at the same time. Another surprising result is that performance collapse happens so quickly.

The second benchmark we used to exercise the Linux spinlock implementation is similar to the first. Instead of a 400 cycle serial section, the serial section is always 2% of the total time to execute the serial section and the non-serial section on one core. We ran the benchmark with serial section lengths of 400 cycles, 1600 cycles, and 25600 cycles. The results from this benchmark should provide some insight into the affect of using collapsing locks on serial sections of different lengths. We might expect that the speedup of longer serial sections will suffer less than shorter serial sections because the cost of acquiring the lock is a smaller fraction of the total execution cost.

Figure A-3 shows the speedup of the second spinlock microbenchmark. The speedup of the 400 cycle and 1600 cycle serial section configurations collapse at 13 and 24 cores respectively. The 25600 cycle serial section configuration does not collapse.

A.3 Questions

The performance results raise three questions that a model should explain.

- Why does collapse start as early as it does? One would expect collapse to start when there is a significant chance that many cores need the same lock at the same time.

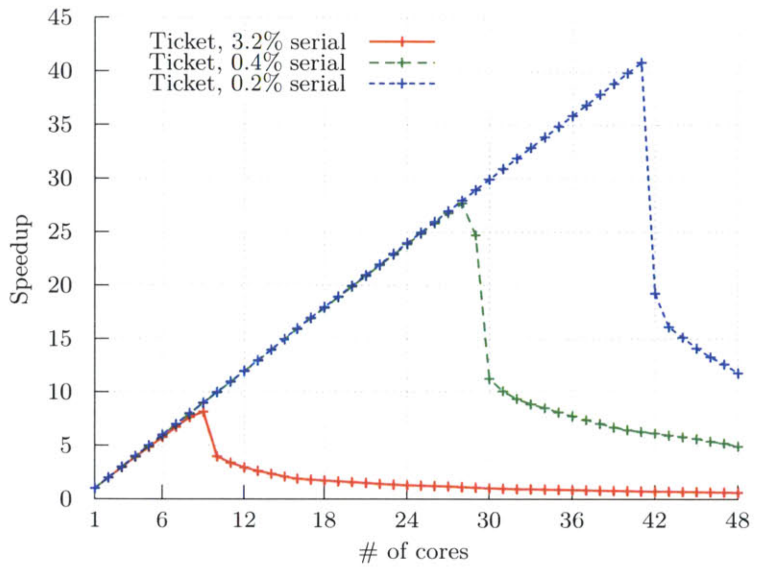


Figure A-2: Speedup of the ticket spinlock microbenchmark with a 400-cycle serial section and different lengths of non-serial sections.

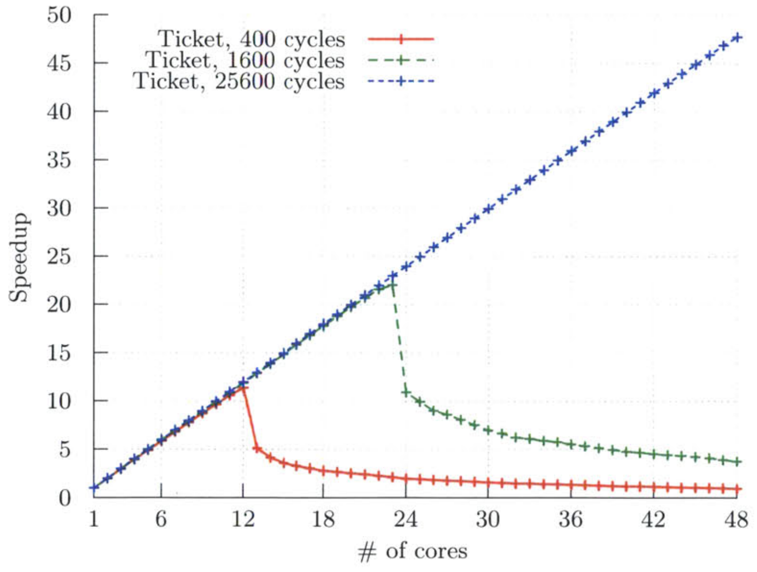


Figure A-3: Speedup of the ticket spinlock microbenchmark with 2% of execution time in the serial section and different amounts of total execution time.

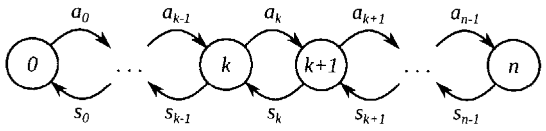


Figure A-4: Markov model for a ticket spin lock for n cores. State i represents i cores holding or waiting for the lock. a_i is the arrival rate of new cores when there are already i cores contending for the lock. s_i is the service rate when $i + 1$ cores are contending.

Thus one might expect speedup of the configuration with a 400 cycle serial section that is 3.2% of total execution to collapse around 31 cores.

- Why does performance ultimately fall so far?
- Why does performance collapse so rapidly? One might expect a gradual decrease with added cores, since each new core should cause each release of the bottleneck lock to take a little more time. Instead, adding just a single core can cause a sharp drop in total throughput. This is worrisome; it suggests that a system that has been tested to perform well with N cores might perform far worse with $N + 1$ cores.

A.4 Model

To understand the collapse observed in ticket-based spin locks, we construct a model. One of the challenging aspects of constructing an accurate model of spin lock behavior is that there are two regimes of operation: when not contended, the spin lock can be acquired quickly, but when many cores try to acquire the lock at the same time, the time taken to transfer lock ownership increases significantly. Moreover, the exact point at which the behavior of the lock changes is dependent on the lock usage pattern, and the length of the critical section, among other parameters. Recent work [10] attempts to model this behavior by combining two models—one for contention and one for uncontended locks—into a single model, by simply taking the max of the two models' predictions. However, this fails to precisely model the point of collapse, and doesn't explain the phenomenon causing the collapse.

To build a precise model of ticket lock behavior, we build on queuing theory to model the ticket lock as a Markov chain. Different states in the chain represent different numbers of cores queued up waiting for a lock, as shown in Figure A-4. There are $n + 1$ states in our model, representing the fact that our system has a fixed number of cores (n).

Arrival and service rates between different states represent lock acquisition and lock release. These rates are different for each pair of states, modeling the non-scalable performance of the ticket lock, as well as the fact that our system is closed (only a finite number of cores exist). In particular, the arrival rate from k to $k + 1$ waiters, a_k, should be proportional to the number of remaining cores that are not already waiting for the lock (i.e., $n - k$). Conversely, the service rate from $k + 1$ to k, s_k, should be inversely proportional to k, reflecting the fact that transferring ownership of a ticket lock to the next core takes linear time in the number of waiters.

To compute the arrival rate, we define a to be the average time between consecutive lock acquisitions on a single core. The rate at which a single core will try to acquire the lock, in the absence of contention, is $1/a$. Thus, if k cores are already waiting for the lock, the arrival rate of new contenders is $a_k = (n - k)/a$, since we need not consider any cores that are already waiting for the lock.

To compute the service rate, we define two more parameters: s, the time spent in the serial section, and c, the time taken by the home directory to respond to a cache line request. In the cache coherence protocol, the home directory of a cache line responds to each cache line request in turn. Thus, if there are k requests from different cores to fetch the lock's cache line, the time until the winner (pre-determined by ticket numbers) receives the cache line will be on average $c \cdot k/2$. As a result, processing the serial section and transferring the lock to the next holder when k cores are contending takes $s + ck/2$, and the service rate is $s_k = \frac{1}{s+ck/2}$.

Unfortunately, while this Markov model accurately represents the behavior of a ticket lock, it does not match any of the standard queuing theory that provides a simple formula for the behavior of the queuing model. In particular, the system is closed (unlike most open-system queuing models), and the service times vary with the size of the queue.

To compute a formula, we derive it from first principles. Let $P_0, \ldots, P_n$ be the steady-state probabilities of the lock being in states 0 through n respectively. Steady state means that the transition rates balance: $P_k \cdot a_k = P_{k+1} \cdot s_k$. From this, we derive that $P_k = P_0 \cdot \frac{n!}{a^k(n-k)!} \cdot \prod_{i=1}^{k}(s+ic)$. Since $\sum_{i=0}^{n} P_i = 1$, we get $P_0 = 1 / \left(\sum_{i=0}^{n}\left(\frac{n!}{a^i(n-i)!}\prod_{j=1}^{i}(s+jc)\right)\right)$, and thus:

$$P_k = \frac{\frac{1}{a^k(n-k)!} \cdot \prod_{i=1}^{k}(s+ic)}{\sum_{i=0}^{n}\left(\frac{1}{a^i(n-i)!}\prod_{j=1}^{i}(s+jc)\right)} \tag{A.1}$$

Given the steady-state probability for each number of cores contending for the lock, we can compute the average number of waiting (idle) cores as the expected value of that distribution, $w = \sum_{i=0}^{n} i \cdot P_i$. The speedup achieved in the presence of this lock and serial section can be computed as $n - w$, since on average that many cores are doing useful work, while w cores are spinning.

A.5 Validating the model

To validate our model, Figures A-5 and A-6 show the predicted and actual speedup of a microbenchmark configuration from Section A.2. As we can see, the model closely matches the real hardware speedup for all configurations.

One difference between the predicted and measured speedup is that the predicted collapse is slightly more gradual than the collapse observed on real hardware. This is because the ticket lock's performance is unstable near the collapse point, and the model predicts the average steady-state behavior. Our measured speedup reports the throughput for a relatively short-running microbenchmark, which has not had the time to "catch" the instability.

A.6 Implications of model results

The behavior predicted by our model has several important implications. First, the rapid collapse of ticket locks is an inherent property of their design, rather than a performance problem with our experimental hardware. Any cache-coherent system that matches our basic hardware model will experience similarly sharp performance degradation. The reason behind the rapid collapse can be understood by considering the transition rates in the Markov

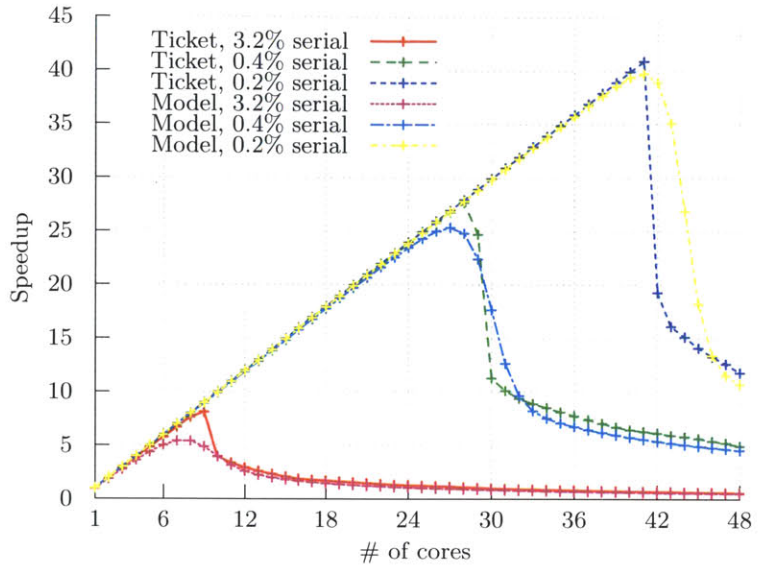


Figure A-5: Speedup and predicted speedup of the ticket spinlock microbenchmark with a 400-cycle serial section and different lengths of non-serial sections.

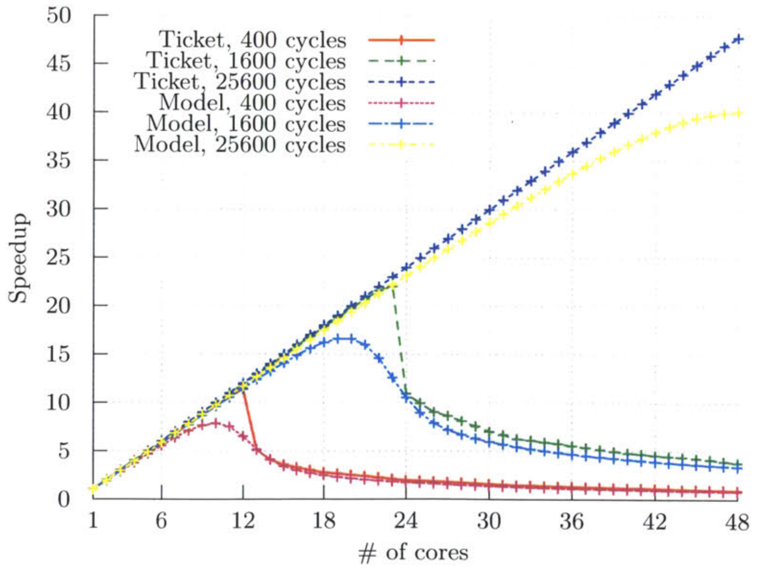


Figure A-6: Speedup and predicted speedup of the ticket spinlock microbenchmark with 2% of execution time in the serial section and different amounts of total execution time.

model from Figure A-4. If a lock ever accumulates a large number of waiters (e.g., reaches state n in the Markov model), it will take a long time for the lock to go back down to a small number of waiters, because the service rate s_k rapidly decays as k grows, for short serial sections. Thus, once the lock enters a contended state, it becomes much more likely that more waiters arrive than that the current waiters will make progress in shrinking the lock's wait queue.

A more direct way to understand the collapse is that the time taken to transfer the lock from one core to another increases linearly with the number of contending cores. However, this time effectively increases the length of the serial section. Thus, as more cores are contending for the lock, the serial section grows, increasing the probability that yet another core will start contending for this lock.

The second implication is that the collapse of the ticket lock only occurs for short serial sections, as can be seen from Figure A-6. This can be understood by considering how the service rate s_i decays for different lengths of the serial section. For a short serial section time s, $s_k = \frac{1}{s+ck/2}$ is strongly influenced by k, but for large s, s_k is largely unaffected by k. Another way to understand this result is that, with fewer acquire and release operations, the ticket lock's performance contributes less to the overall application throughput.

The third implication is that the collapse of the ticket lock prevents the application from reaching the maximum performance predicted by Amdahl's law (for short serial sections). In particular, Figure A-6 shows that a microbenchmark with a 2% serial section, which may be able to scale to 50 cores under Amdahl's law, is able to attain less than 10× scalability when the serial section is 400 cycles long.

Bibliography

[1] Exim, May 2010. `http://www.exim.org/`.

[2] Recoll, July 2013. `www.recoll.org/`.

[3] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3):6, 2007.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Haris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc of the 22nd SOSP*, Big Sky, MT, USA, Oct 2009.

[5] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proc. of the 8th OSDI*, December 2008.

[6] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proc. of the 9th OSDI*, Vancouver, Canada, October 2010.

[7] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference (EuroSys 2013)*, Prague, Czech Republic, April 2013.

[9] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, March 2010.

[10] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of ISCA*, pages 262–370, 2010.

[11] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of the 3rd OSDI*, pages 87–100, 1999.

[12] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

[13] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, 2010.

[14] Intel. An introduction to the Intel QuickPath Interconnect, January 2009.

[15] Jonathan Corbet. Dcache scalability and RCU-walk, April 23, 2012. `http://lwn.net/Articles/419811/`.

[16] Jonathan Corbet. The case of the overly anonymous anon_vma, March 28, 2013. `http://lwn.net/Articles/383162/`.

[17] Jonathan Corbet. The object-based reverse-mapping VM, March 28, 2013. `http://lwn.net/Articles/23732/`.

[18] Jonathan Corbet. Virtual memory ii: the return of objrmap, March 28, 2013. `http://lwn.net/Articles/75198/`.

[19] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf`.

[20] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Linux Symposium*, pages 338–367, 2002.

[21] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[22] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, and Alan J. Demers. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.

[23] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.