The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

by

Austin T. Clements

S.B., Massachusetts Institute of Technology (2006)
M.Eng., Massachusetts Institute of Technology (2008)

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

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2014

Certified by . . . .
M. Frans Kaashoek
Charles Piper Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

The Scalable Commutativity Rule:
Designing Scalable Software for Multicore Processors

by

Austin T. Clements

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

What fundamental opportunities for multicore scalability are latent in software interfaces, such as system call APIs? Can scalability opportunities be identified even before any implementation exists, simply by considering interface specifications? To answer these questions this dissertation introduces the scalable commutativity rule: *Whenever interface operations commute, they can be implemented in a way that scales.* This rule aids developers in building scalable multicore software starting with interface design and carrying on through implementation, testing, and evaluation.

This dissertation formalizes the scalable commutativity rule and defines a novel form of commutativity named *SIM commutativity* that makes it possible to fruitfully apply the rule to complex and highly stateful software interfaces.

To help developers apply the rule, this dissertation introduces an automated method embodied in a new tool named Commuter, which accepts high-level interface models, generates tests of operations that commute and hence could scale, and uses these tests to systematically evaluate the scalability of implementations. We apply Commuter to a model of 18 POSIX file and virtual memory system operations. Using the resulting 26,238 scalability tests, Commuter systematically pinpoints many problems in the Linux kernel that past work has observed to limit application scalability and identifies previously unknown bottlenecks that may be triggered by future hardware or workloads.

Finally, this dissertation applies the scalable commutativity rule and Commuter to the design and implementation of a new POSIX-like operating system named sv6. sv6's novel file and virtual memory system designs enable it to scale for 99% of the tests generated by Commuter. These results translate to linear scalability on an 80-core x86 machine for applications built on sv6's commutative operations.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Computer Science and Engineering

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

This work would not have been possible without the combined efforts and determination of my advisors Frans Kaashoek, Nickolai Zeldovich, Robert Morris, and Eddie Kohler. Frans' unwavering focus on The Technical Nugget led to our simplest and most powerful ideas. Nickolai's raw energy and boundless breadth are reflected in every corner of this dissertation. Robert's keen vision and diligent skepticism pushed the frontiers of what we thought possible. And Eddie's inexhaustible enthusiasm and remarkable intuition reliably yielded solutions to seemingly impossible problems. I am honored to have had them as my mentors.

This research was greatly improved by the valuable feedback of Silas Boyd-Wickizer, Yandong Mao, Xi Wang, Marc Shapiro, Rachid Guerraoui, Butler Lampson, Paul McKenney, and the twenty three anonymous reviewers who pored over the many drafts of this work.

Silas and Nickolai contributed substantially to the implementation of sv6 and sv6 would not have been possible without xv6, which was written by Frans, Robert, and Russ Cox.

I thank my parents, Hank and Robin, for their encouragement, guidance, and support. Finally, I thank my wife, Emily, for her love and understanding, and for being my beacon through it all.

* * *

This research was supported by NSF awards SHF-964106 and CNS-1301934, by grants from Quanta Computer and Google, and by a VMware graduate fellowship.

* * *

This dissertation incorporates and extends work previously published in the following papers:

> Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, Pennsylvania, November 2013.
>
> Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference*, Prague, Czech Republic, April 2013.
>
> Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Concurrent address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

Contents

1 Introduction **11**

1.1 Parallelize or perish 11
1.2 A rule for interface design 12
1.3 Applying the rule 13
1.4 Contributions 15
1.5 Outline 16

2 Related work **17**

2.1 Thinking about scalability 17
2.2 Designing scalable operating systems 18
2.3 Commutativity 18
2.4 Test case generation 19

3 Scalability and conflict-freedom **21**

3.1 Conflict-freedom and multicore processors 21
3.2 Conflict-free operations scale 23
3.3 Limitations of conflict-free scalability 25
3.4 Summary 27

4 The scalable commutativity rule **29**

4.1 Actions 29
4.2 SIM commutativity 30
4.3 Implementations 32
4.4 Rule 33
4.5 Example 33
4.6 Proof 35
4.7 Discussion 37

5 Designing commutative interfaces **39**

5.1 Decompose compound operations 39
5.2 Embrace specification non-determinism 40

5.3 Permit weak ordering . . . 40
5.4 Release resources asynchronously . . . 40

6 Analyzing interfaces using COMMUTER **43**
6.1 ANALYZER . . . 43
6.2 TESTGEN . . . 49
6.3 MTRACE . . . 51
6.4 Implementation . . . 51

7 Conflict-freedom in Linux **53**
7.1 POSIX test cases . . . 53
7.2 Linux conflict-freedom . . . 54

8 Achieving conflict-freedom in POSIX **57**
8.1 Refcache: Scalable reference counting . . . 58
8.2 RadixVM: Scalable address space operations . . . 64
8.3 ScaleFS: Conflict-free file system operations . . . 71
8.4 Difficult-to-scale cases . . . 73

9 Performance evaluation **75**
9.1 Experimental setup . . . 75
9.2 File system microbenchmarks . . . 76
9.3 File system application performance . . . 78
9.4 Virtual memory microbenchmarks . . . 79
9.5 Virtual memory application benchmark . . . 82
9.6 Memory overhead . . . 83
9.7 Discussion . . . 84

10 Future directions **85**
10.1 The non-scalable non-commutativity rule . . . 85
10.2 Synchronized clocks . . . 86
10.3 Scalable conflicts . . . 87
10.4 Not everything can commute . . . 87
10.5 Broad conflict-freedom . . . 87

11 Conclusion **89**

Figures and tables

3-1 A basic cache-coherence state machine. 22
3-2 Organization of benchmark machines. 23
3-3 Conflict-free accesses scale. 24
3-4 Conflicting accesses do not scale. 25
3-5 Operations scale until they exceed cache or directory capacity. 26

4-1 Constructed *non-scalable* implementation m_{ns} for history H. 35
4-2 Constructed scalable implementation m for history H. 37

6-1 The components of COMMUTER. 43
6-2 The SIM commutativity test algorithm. 45
6-3 A simplified version of our rename model. 46
6-4 The SIM commutativity test algorithm specialized to two operations. 46
6-5 Symbolic execution tree of commutes2 for rename/rename. 48
6-6 An example test case for two rename calls generated by TESTGEN. 50

7-1 Conflict-freedom of commutative system call pairs in Linux. 54

8-1 Conflict-freedom of commutative system call pairs in sv6. 58
8-2 Refcache example showing a single object over eight epochs. 61
8-3 Refcache algorithm. 63
8-4 Key structures in a POSIX VM system. 65
8-5 Throughput of skip list lookups with concurrent inserts and deletes. 67
8-6 A radix tree containing a three page file mapping. 68

9-1 File system microbenchmark throughput. 77
9-2 Mail server benchmark throughput. 79
9-3 Virtual memory microbenchmark throughput. 80
9-4 Metis application scalability. 83
9-5 Memory usage for alternate VM representations. 84

One

Introduction

This dissertation presents a pragmatic and formal approach to the design and implementation of scalable multicore software that spans from the earliest stages of software interface design through testing and maintenance of complete implementations.

The rest of this chapter introduces the multicore architectures that now dominate general-purpose computing, the problematic ways in which software developers are coping with these new architectures, and a new interface-driven approach to the design and implementation of software for multicore architectures.

1.1 Parallelize or perish

In the mid-2000s, there was a fundamental shift in the construction of high performance software. For decades, CPU clock speeds had ridden the exponential curve of Moore's Law and rising clock speeds naturally translated to faster software performance. But higher clock speeds require more power and generate more heat, and around 2005 CPUs reached the thermal dissipation limits of a few square centimeters of silicon. CPU architects could no longer significantly increase the clock speed of a single CPU core, so they began to increase parallelism instead by putting more CPU cores on the same chip. Now, with the widespread adoption of multicore architectures for general-purpose computing, parallel programming has gone from niche to necessary. *Total* cycles per second continues to grow exponentially, but now software must be increasingly parallel to take advantage of this growth. Unfortunately, while software performance naturally scaled with clock speed, scaling with parallelism is an untamed problem. Even with careful engineering, software rarely achieves the holy grail of *linear scalability*, where doubling hardware parallelism doubles the software's performance.

Operating system kernels exemplify both the importance of parallelism and the difficulty of achieving it. Many applications depend heavily on the shared services and resources provided by the kernel. As a result, if the kernel doesn't scale, many applications won't scale. At the same time, the kernel must cope with diverse and unknown workloads while supporting the combined parallelism of all applications on a computer. Additionally, the kernel's role as the arbiter of shared resources makes it particularly susceptible to scalability problems.

Yet, despite the extensive efforts of kernel and application developers alike, scaling software performance on multicores remains an inexact science dominated by guesswork, measurement, and expensive cycles of redesign. The state of the art for evaluating and improving the scalability of multicore software is to choose some workload, plot performance at varying numbers of cores, and use tools such as differential profiling [47] to identify scalability bottlenecks.

This approach focuses developer effort on demonstrable issues, but is ultimately a near-sighted approach. Each new hardware model and workload powers a Sisyphean cycle of finding and fixing new bottlenecks. Projects such as Linux require continuous infusions of manpower to maintain their scalability edge. Worse, scalability problems that span layers—for example, application behavior that triggers kernel bottlenecks—require cross-layer solutions, and few applications have the reach or resources to accomplish this.

But the deeper problem with this workload-driven approach is that many scalability problems lie not in the implementation, but in *the design of the software interface*. By the time developers have an implementation, a workload, and the hardware to demonstrate a bottleneck, interface-level solutions may be impractical or impossible.

As an example of interface design that limits implementation scalability, consider the POSIX open call [63]. This call opens a file by name and returns a *file descriptor*, a number used to identify the open file in later operations. Even though this identifier is opaque, POSIX—the standard for the Unix interface—requires that open return the numerically lowest available file descriptor for the calling process, forcing the kernel to allocate file descriptors one at a time, even when many parallel threads are opening files simultaneously. This simplified the kernel interface during the early days of Unix, but this interface design choice is now a burden on implementation scalability. It's an unnecessary burden, too: a simple change to allow open to return *any* available file descriptor would enable the kernel to choose file descriptors scalably. This particular example is well-known [10], but myriad subtler issues exist in POSIX and other interfaces.

Interface design choices have implications for implementation scalability. If interface designers could distinguish interfaces that definitely have a scalable implementation from those that don't, they would have the predictive power to design *scalable interfaces* that enable scalable implementations.

1.2 A rule for interface design

This dissertation presents a new approach to designing scalable software that starts with the design of scalable software interfaces. This approach makes reasoning about multicore scalability possible before an implementation exists and before the necessary hardware is available to measure the implementation's scalability. It can highlight inherent scalability problems, leading to better interface designs. It sets a clear scaling target for the implementation of a

scalable interface. And it enables systematic testing of an implementation's scalability.

At the core of this dissertation's approach is this *scalable commutativity rule*: In any situation where several operations *commute*—meaning there's no way to distinguish their execution order using the interface—they have a implementation that is *conflict-free* during those operations—meaning no core writes a cache line that was read or written by another core. Empirically, conflict-free operations scale, so this implementation scales. Or, more concisely, **whenever interface operations commute, they can be implemented in a way that scales.**

This rule makes intuitive sense: when operations commute, their results (return values and effect on system state) are independent of order. Hence, communication between commutative operations is unnecessary, and eliminating it yields a conflict-free implementation. On modern shared-memory multicores, conflict-free operations can execute entirely from per-core caches, so the performance of a conflict-free implementation will scale linearly with the number of cores.

The intuitive version of the rule is useful in practice, but not precise enough to reason about formally. Therefore, this dissertation formalizes the scalable commutativity rule, proves that commutative operations have a conflict-free implementation, and demonstrates experimentally that, under reasonable assumptions, conflict-free implementations scale linearly on modern multicore hardware.

An important consequence of this presentation is a novel form of commutativity we name *SIM commutativity*. The usual definition of commutativity (e.g., for algebraic operations) is so stringent that it rarely applies to the complex, stateful interfaces common in systems software. SIM commutativity, in contrast, is *state-dependent* and *interface-based*, as well as *monotonic*. When operations commute in the context of a specific system state, specific operation arguments, and specific concurrent operations, we show that an implementation exists that is conflict-free *for that state and those arguments and concurrent operations*. SIM commutativity exposes many more opportunities to apply the rule to real interfaces—and thus discover scalable implementations—than a more conventional notion of commutativity would. Despite its state dependence, SIM commutativity is interface-based: rather than requiring all operation orders to produce identical internal states, it requires the resulting states to be indistinguishable via the interface. SIM commutativity is thus independent of any specific implementation, enabling developers to apply the rule directly to interface design.

1.3 Applying the rule

The scalable commutativity rule leads to a new way to design scalable software: analyze the interface's commutativity; if possible, refine or redesign the interface to improve its commutativity; and then design an implementation that scales when operations commute.

For example, consider file creation in a POSIX-like file system. Imagine that multiple processes create files in the same directory at the same time. Can the creation system calls

be made to scale? Our first answer was "obviously not": the system calls modify the same directory, so surely the implementation must serialize access to the directory. But it turns out these operations commute if the two files have different names (and no hard or symbolic links are involved) and, therefore, have an implementation that scales for such names. One such implementation represents each directory as a hash table indexed by file name, with an independent lock per bucket, so that creation of differently named files is conflict-free, barring hash collisions. Before the rule, we tried to determine if these operations could scale by analyzing all of the *implementations* we could think of. This process was difficult, unguided, and itself did not scale to complex interfaces, which motivated our goal of reasoning about scalability in terms of interfaces.

Real-world interfaces and their implementations are complex. Even with the rule, it can be difficult to spot and reason about all commutative cases. To address this challenge, this dissertation introduces a method to automate reasoning about interfaces and implementations, embodied in a software tool named COMMUTER. COMMUTER takes a symbolic interface model, computes precise conditions under which sets of operations commute, generates concrete tests of commutative operations, and uses these tests to reveal conflicts in an implementation. Any conflicts COMMUTER finds represent an opportunity for the developer to improve the scalability of their implementation. This tool can be integrated into the software development process to drive initial design and implementation, to incrementally improve existing implementations, and to help developers understand the commutativity of an interface.

We apply COMMUTER to a model of 18 POSIX file system and virtual memory system calls. From this model, COMMUTER generates 26,238 tests of commutative system call pairs, all of which can be made conflict-free according to the rule. Applying this suite to Linux, we find that the Linux kernel is conflict-free for 17,206 (65%) of these cases. Many of the commutative cases where Linux is not conflict-free are important to applications—such as commutative mmaps and creating different files in a shared directory—and reflect bottlenecks found in previous work [11]. Others reflect previously unknown problems that may become bottlenecks on future machines or workloads.

Finally, to demonstrate the application of the rule and COMMUTER to the design and implementation of a real system, we use these tests to guide the implementation of a new research operating system kernel named sv6. sv6 doubles as an existence proof showing that the rule can be applied fruitfully to the design and implementation of a large software system, and as an embodiment of several novel scalable kernel implementation techniques. COMMUTER verifies that sv6 is conflict-free for 26,115 (99%) of the tests generated by our POSIX model and confirms that sv6 addresses many of the sources of conflicts found in the Linux kernel. sv6's conflict-free implementations of commutative system calls translate to dramatic improvements in measured scalability for both microbenchmarks and application benchmarks.

1.4 Contributions

This dissertation's broad contribution is a new approach to building scalable software using interface-based reasoning to guide design, implementation, and testing. This dissertation makes the following intellectual contributions:

- The scalable commutativity rule, its formalization, and a proof of its correctness.
- SIM commutativity, a novel form of interface commutativity that is state-sensitive and interface-based. As we demonstrate with COMMUTER, SIM commutativity enables us to identify myriad commutative cases in the highly stateful POSIX interface.
- A set of guidelines for commutative interface design based on SIM commutativity. Using these guidelines, we propose specific enhancements to POSIX and empirically demonstrate that these changes enable dramatic improvements in application scalability.
- An automated method for reasoning about interface commutativity and generating implementation scalability tests using symbolic execution. This method is embodied in a new tool named COMMUTER, which generates 26,238 tests of commutative operations in our model of 18 POSIX file system and virtual memory system operations. These tests cover many subtle cases, identify many substantial scalability bottlenecks in the Linux kernel, and guide the implementation of sv6.

This dissertation also contributes several operating system implementation techniques:

- Refcache, a novel scalable reference counting scheme that achieves conflict-free increment and decrement operations with efficient periodic reconciliation and $O(1)$ per-object space overhead.
- RadixVM, a POSIX virtual memory system based on radix trees that is conflict-free for commutative mmap, munmap, and pagefault operations.
- ScaleFS, a POSIX file system that is conflict-free for the vast majority of commutative file system operations.

We validate RadixVM and ScaleFS, and thus their design methodology based on the rule and COMMUTER, by evaluating their performance and scalability on an 80-core x86 machine.

The source code to all of the software produced for this dissertation is publicly available under an MIT license from http://pdos.csail.mit.edu/commuter.

1.5 Outline

The rest of this dissertation presents the scalable commutativity rule in depth and explores its consequences from interface design to implementation to testing.

We begin by relating our thinking about scalability to previous work in chapter 2.

We then turn to formalizing and proving the scalable commutativity rule, which we approach in two steps. First, chapter 3 establishes experimentally that conflict-free operations are generally scalable on modern, large multicore machines. Chapter 4 then formalizes the rule, develops SIM commutativity, and proves that commutative operations can have conflict-free (and thus scalable) implementations.

We next turn to applying the rule. Chapter 5 starts by applying the rule to interface design, developing a set of guidelines for designing interfaces that enable scalable implementations and proposing specific modifications to POSIX that broaden its commutativity.

Chapter 6 presents COMMUTER, which uses the rule to automate reasoning about interface commutativity and the conflict-freedom of implementations. Chapter 7 uses COMMUTER to analyze the Linux kernel and demonstrates that COMMUTER can systematically pinpoint significant scalability problems even in mature systems.

Finally, we turn to the implementation of scalable systems guided by the rule. Chapter 8 describes the implementation of sv6 and how it achieves conflict-freedom for the vast majority of commutative POSIX file system and virtual memory operations. Chapter 9 confirms that theory translates into practice by evaluating the performance and scalability of sv6 on real hardware for several microbenchmarks and application benchmarks.

Chapter 10 takes a step back and explores some promising future directions for this work. Chapter 11 concludes.

Two

Related work

The scalable commutativity rule is to the best of our knowledge the first observation to directly connect scalability to interface commutativity. This chapter relates the rule and its use in sv6 and Commuter to prior work.

2.1 Thinking about scalability

Israeli and Rappoport introduce the notion of disjoint-access-parallel memory systems [41]. Roughly, if a shared memory system is disjoint-access-parallel and a set of processes access disjoint memory locations, then those processes scale linearly. Like the commutativity rule, this is a conditional scalability guarantee: if the application uses shared memory in a particular way, then the shared memory implementation will scale. However, where disjoint-access parallelism is specialized to the memory system interface, our work encompasses any software interface. Attiya et al. extend Israeli and Rappoport's definition to additionally require non-disjoint reads to scale [3]. Our work builds on the assumption that memory systems behave this way, and we confirm that real hardware closely approximates this behavior (chapter 3).

Both the original disjoint-access parallelism paper and subsequent work, including the paper by Roy et al. [56], explore the scalability of processes that have some amount of non-disjoint sharing, such as compare-and-swap instructions on a shared cache line or a shared lock. Our work takes a black-and-white view because we have found that, on real hardware, a single modified shared cache line can wreck scalability (chapters 3 and 9).

The Laws of Order [2] explore the relationship between the *strong non-commutativity* of an interface and whether any implementation of that interface must have atomic instructions or fences (e.g., mfence on the x86) for correct concurrent execution. These instructions slow down execution by interfering with out-of-order execution, even if there are no memory access conflicts. The Laws of Order resemble the commutativity rule, but draw conclusions about sequential performance, rather than scalability. Paul McKenney explores the Laws of Order in the context of the Linux kernel, and points out that the Laws of Order may not apply if linearizability is not required [48].

It is well understood that cache-line contention can result in bad scalability. A clear example is the design of the MCS lock [50], which eliminates scalability collapse by avoiding contention for a particular cache line. Other good examples include scalable reference counters [19, 29]. The commutativity rule builds on this understanding and identifies when arbitrary interfaces can avoid conflicting memory accesses.

2.2 Designing scalable operating systems

Practitioners often follow an iterative process to improve scalability: design, implement, measure, repeat [14]. Through a great deal of effort, this approach has led kernels such as Linux to scale well for many important workloads. However, Linux still has many scalability bottlenecks, and absent a method for reasoning about interface-level scalability, it is unclear which of the bottlenecks are inherent to its system call interface. This dissertation identifies situations where POSIX permits or limits scalability and points out specific interface modifications that would permit greater implementation scalability.

Venerable scalable kernels like K42 [1], Tornado [31], and Hurricane [65] have developed design patterns like clustered objects and locality-preserving IPC as general building blocks of scalable kernels. These patterns complement the scalable commutativity rule by suggesting practical ways to achieve conflict-freedom for commutative operations as well as ways to cope with non-commutative operations.

Multikernels for multicore processors aim for scalability by avoiding shared data structures in the kernel [4, 68]. These systems implement shared abstractions using distributed systems techniques (such as name caches and state replication) on top of message passing. It should be possible to generalize the commutativity rule to distributed systems, and relate the interface exposed by a shared abstraction to its scalability, even if implemented using message passing.

The designers of the Corey operating system [10] argue for putting the application in control of managing the cost of sharing without providing a guideline for how applications should do so; the commutativity rule could be a helpful guideline for application developers.

2.3 Commutativity

The use of commutativity to increase concurrency has been widely explored. Steele describes a parallel programming discipline in which all operations must be either causally related or commutative [61]. His work approximates commutativity as conflict-freedom. This dissertation shows that commutative operations always have a conflict-free implementation, making Steele's model more broadly applicable. Rinard and Diniz describe how to exploit commutativity to automatically parallelize code [55]. They allow memory conflicts, but generate synchronization code to ensure atomicity of commutative operations. Similarly, Prabhu et al. describe how to automatically parallelize code using manual annotations rather than

automatic commutativity analysis [53]. Rinard and Prabhu's work focuses on the *safety* of executing commutative operations concurrently. This gives operations the opportunity to scale, but does not ensure that they will. Our work focuses on scalability directly: given concurrent, commutative operations, we show they have a scalable implementation.

The database community has long used logical readsets and writesets, conflicts, and execution histories to reason about how transactions can be interleaved while maintaining serializability [7]. Weihl extends this work to abstract data types by deriving lock conflict relations from operation commutativity [67]. Transactional boosting applies similar techniques in the context of software transactional memory [35]. Shapiro et al. extend this to a distributed setting, leveraging commutative operations in the design of replicated data types that support updates during faults and network partitions [59, 60]. Like Rinard and Prabhu's work, the work in databases and its extensions focuses on the *safety* of executing commutative operations concurrently, not directly on scalability.

2.4 Test case generation

Prior work on concolic testing [33, 58] and symbolic execution [12, 13] generates test cases by symbolically executing a specific implementation. Our COMMUTER tool uses a combination of symbolic and concolic execution, but generates test cases for an *arbitrary* implementation based on a model of that implementation's interface. This resembles QuickCheck's [15] or Gast's [42] model-based testing, but uses symbolic techniques. Furthermore, while symbolic execution systems often avoid reasoning precisely about symbolic memory accesses (e.g., accessing a symbolic offset in an array), COMMUTER's test case generation aims to achieve *conflict* coverage (section 6.2), which tests different access patterns when using symbolic addresses or indexes.

Three

Scalability and conflict-freedom

Understanding multicore scalability requires first understanding the hardware. This chapter shows that, under reasonable assumptions, conflict-free operations scale linearly on modern multicore hardware. The following chapter will use conflict-freedom as a stepping-stone in establishing the scalable commutativity rule.

3.1 Conflict-freedom and multicore processors

The connection between conflict-freedom and scalability mustn't be taken for granted. Indeed, some early multi-processor architectures such as the Intel Pentium depended on shared buses with global lock lines [40: §8.1.4], so even conflict-free operations did not scale.

Today's multicores avoid such centralized components. Modern, large, cache-coherent multicores utilize peer-to-peer interconnects between cores and sockets; partition and distribute physical memory between sockets (NUMA); and have deep cache hierarchies with per-core write-back caches. To maintain a unified, globally consistent view of memory despite their distributed architecture, multicores depend on MESI-like coherence protocols [52] to coordinate ownership of cached memory. A key invariant of these coherence protocols is that either 1) a cache line is not present in any cache, 2) a mutable copy is present in a single cache, or 3) the line is present in any number of caches but is immutable. Maintaining this invariant requires coordination, and this is where the connection to scalability lies.

Figure 3-1 shows the basic state machine implemented by each cache for each cache line. This maintains the above invariant by ensuring a cache line is either "invalid" in all caches, "modified" in one cache and "invalid" in all others, or "shared" in any number of caches. Practical implementations add further states—MESI's "exclusive" state, Intel's "forward" state [34], and AMD's "owned" state [27: §7.3]—but these do not change the basic communication required to maintain cache coherence.

Roughly, a set of operations scales when maintaining coherence does not require communication in the steady state. There are three memory access patterns that do not require communication:

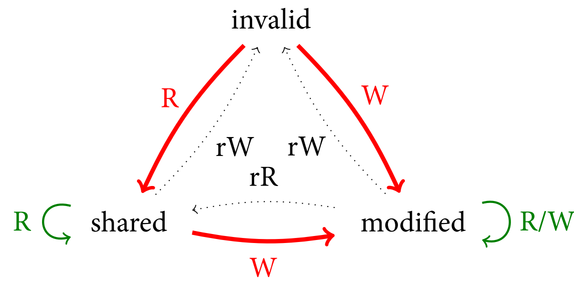

Figure 3-1: A basic cache-coherence state machine. "R" and "W" indicate local read and write operations, while "rR" and "rW" indicate remote read and write operations. Thick red lines show operations that cause communication. Thin green lines show operations that occur without communication.

- Multiple cores *reading different* cache lines. This scales because, once each cache line is in each core's cache, no further communication is required to access it, so further reads can proceed independently of concurrent operations.
- Multiple cores *writing different* cache lines scales for much the same reason.
- Multiple cores *reading the same* cache line scales. A copy of the line can be kept in each core's cache in shared mode, which further reads from those cores can access without communication.

That is, when memory accesses are conflict-free, they do not require communication. Furthermore, higher-level operations composed of conflict-free reads and writes are themselves conflict-free and will also execute independently and in parallel. In all of these cases, conflict-free operations execute in the same time in isolation as they do concurrently, so the total throughput of N such concurrent operations is proportional to N. Therefore, given a perfect implementation of MESI, conflict-free operations scale linearly. The following sections verify this assertion holds on real hardware under reasonable workload assumptions and explore where it breaks down.

The converse is also true: conflicting operations cause cache state transitions and the resulting coordination limits scalability. That is, if a cache line written by one core is read or written by other cores, those operations must coordinate and, as a result, will slow each other down. While this doesn't directly concern the scalable commutativity rule (which says only when operations can be conflict-free, not when they must be conflicted), the huge effect that conflicts can have on scalability affirms the importance of conflict-freedom. The following sections also demonstrate the effect of conflicts on real hardware.

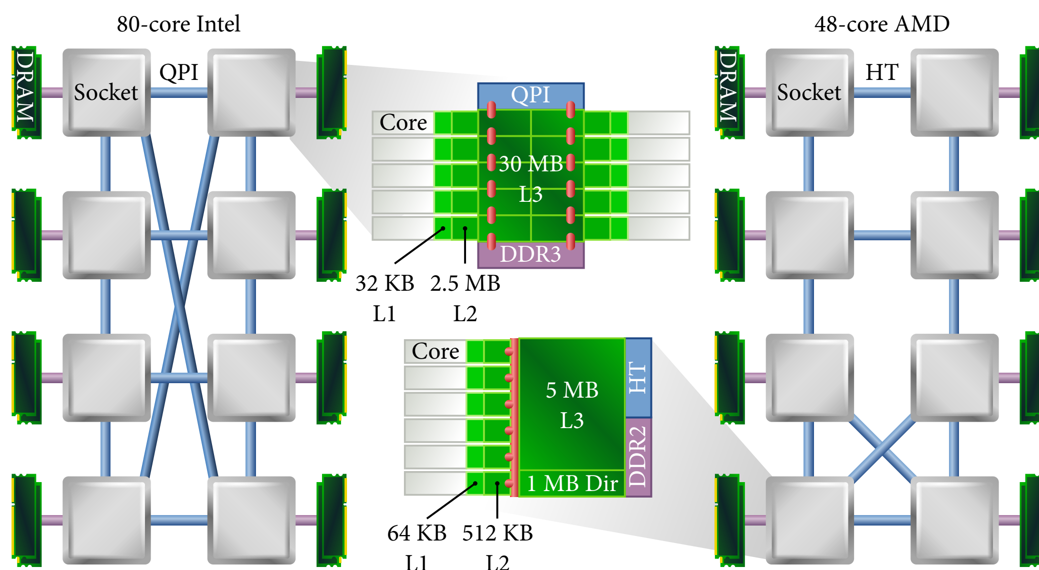

Figure 3-2: Organization of Intel and AMD machines used for benchmarks [18, 21, 22].

3.2 Conflict-free operations scale

We use two machines to evaluate conflict-free and conflicting operations on real hardware: an 80-core (8 sockets × 10 cores) Intel Xeon E7-8870 (the same machine used for evaluation in chapter 9) and, to show that our conclusions generalize, a 48-core (8 sockets × 6 cores) AMD Opteron 8431. Both are cc-NUMA x86 machines with directory-based cache coherence, but the two manufacturers use different architectures, interconnects, and coherence protocols. Figure 3-2 shows how the two machines are broadly organized.

Figure 3-3 shows the time required to perform conflict-free memory accesses from varying numbers of cores. The first benchmark, shown in the top row of Figure 3-3, stresses read/read sharing by repeatedly reading the same cache line from N cores. The latency of these reads remains roughly constant regardless of N. After the first access from each core, the cache line remains in each core's local cache, so later accesses occur locally and independently, allowing read/read accesses to scale perfectly. Reads of different cache lines from different cores (not shown) yield identical results to reads of the same cache line.

The bottom row of Figure 3-3 shows the results of stressing conflict-free writes by assigning each core a different cache line and repeatedly writing these cache lines from each of N cores. In this case these cache lines enter a "modified" state at each core, but then remain in that state, so as with the previous benchmark, further writes can be performed locally and independently. Again, latency remains constant regardless of N, demonstrating that conflict-free write accesses scale.

Figure 3-4 turns to the cost of conflicting accesses. The top row shows the latency of N cores writing the same cache line simultaneously. The cost of a write/write conflict grows

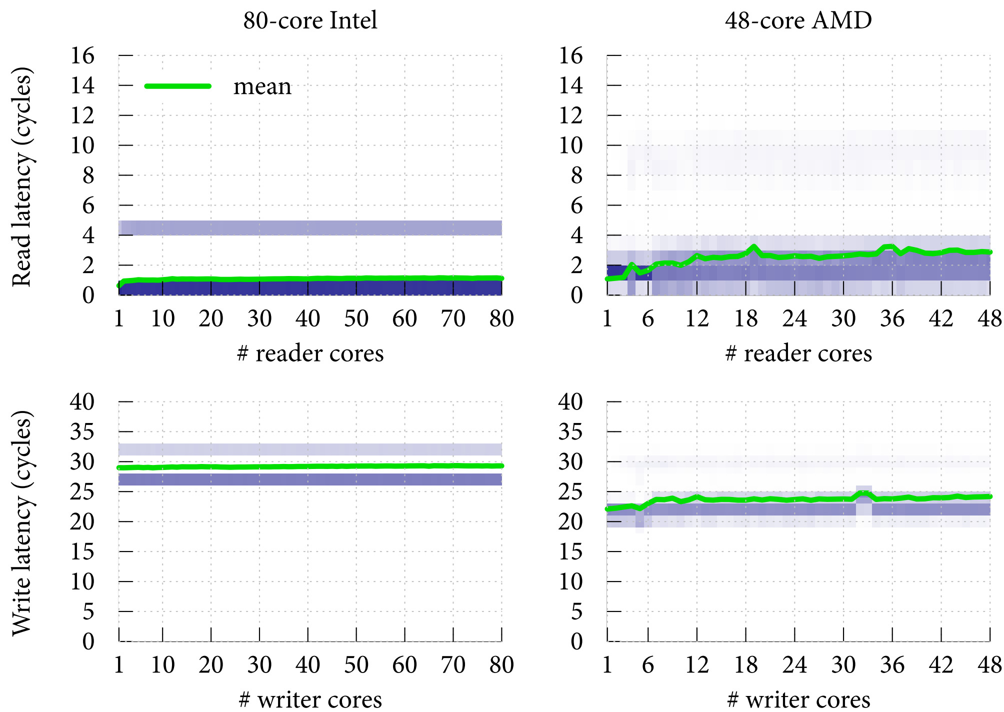

Figure 3-3: Conflict-free accesses scale. Each graph shows the cycles required to perform a conflict-free read or write from N cores. Shading indicates the latency distribution for each N (darker shading indicates higher frequency).

dramatically as the number of writing cores increases because ownership of the modified cache line must pass to each writing core, one at a time. On both machines, we also see a uniform distribution of write latencies, which further illustrates this serialization, as some cores acquire ownership quickly, while others take much longer.

For comparison, an operation like open typically takes 1,000–2,000 cycles on these machines, while a single conflicting write instruction can take upwards of 50,000 cycles. In the time it takes one thread to execute this one machine instruction, another could open 25 files.

The bottom row of Figure 3-4 shows the latency of N cores simultaneously reading a cache line last written by core 0 (a read/write conflict). For the AMD machine, the results are nearly identical to the write/write conflict case, since this machine serializes requests for the cache line at CPU 0's socket. On the Intel machine, the cost of read/write conflicts also grows, albeit more slowly, as Intel's architecture aggregates the read requests at each socket. We see this effect in the latency distribution, as well, with read latency exhibiting up to eight different modes. These modes reflect the order in which the eight sockets' aggregated read requests are served by CPU 0's socket. Intel's optimization helps reduce the absolute latency of reads, but nevertheless, read/write conflicts do not scale on either machine.

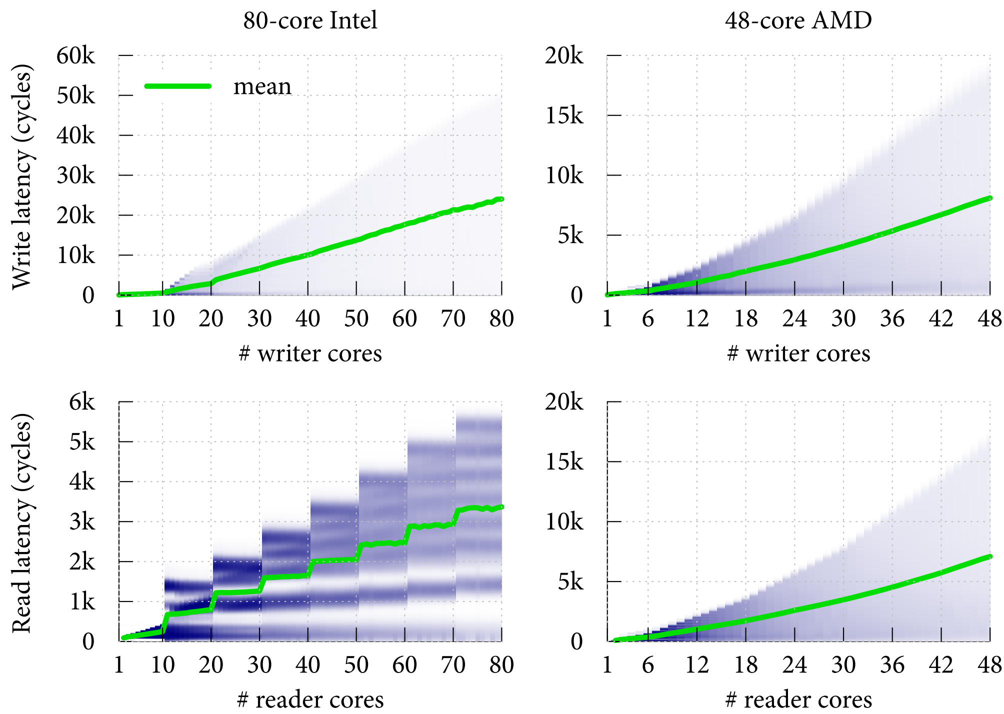

Figure 3-4: Conflicting accesses do not scale. Each graph shows the cycles required to perform a conflicting read or write from N cores. Shading indicates the latency distribution for each N (estimated using kernel density estimation).

3.3 Limitations of conflict-free scalability

Conflict-freedom is a good predictor of scalability on real hardware, but it's not perfect. Limited cache capacity and associativity cause caches to evict cache lines (later resulting in cache misses) even in the absence of coherence traffic. And, naturally, a core's very first access to a cache line will miss. Such misses directly affect sequential performance, but they may also affect the scalability of conflict-free operations. Satisfying a cache miss (due to conflicts or capacity) requires the cache to fetch the cache line from another cache or from memory. If this requires communicating with remote cores or memory, the fetch may contend with concurrent operations for interconnect resources or memory controller bandwidth.

Applications with good cache behavior are unlikely to exhibit such issues, while applications with poor cache behavior usually have sequential performance problems that outweigh scalability concerns. Nevertheless, it's important to understand where our assumptions about conflict-freedom break down.

Figure 3-5 shows the results of a benchmark that explores some of these limits by performing conflict-free accesses to regions of varying sizes from varying numbers of cores. This

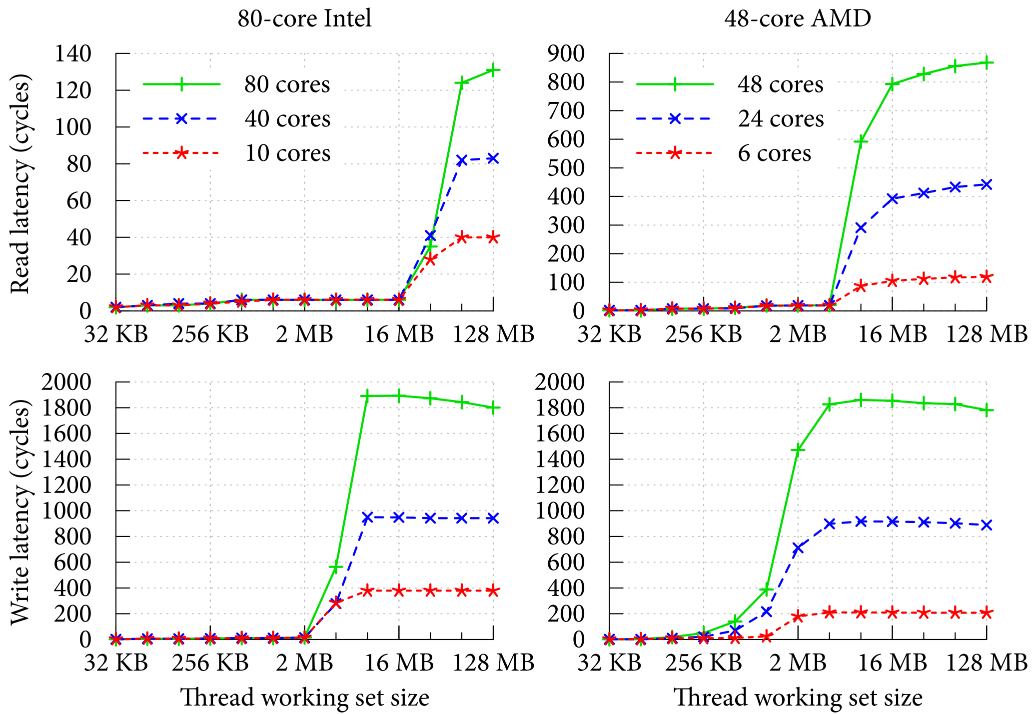

Figure 3-5: Operations scale until they exceed cache or directory capacity. Each graph shows the latency for repeatedly reading a shared region of memory (top) and writing separate per-core regions (bottom), as a function of region size and number of cores.

benchmark stresses the worst case: each core reads or writes in a tight loop and all memory is physically allocated from CPU 0's socket, so all misses contend for that socket's resources. The top row of Figure 3-5 shows the latency of reads to a shared region of memory. On both machines, we observe slight increases in latency as the region exceeds the L1 cache and later the L2 cache, but the operations continue to scale until the region exceeds the L3 cache. At this point the benchmark becomes bottlenecked by the DRAM controller of CPU 0's socket, so the reads no longer scale, despite being conflict-free.

We observe a similar effect for writes, shown in the bottom row of Figure 3-5. On the Intel machine, the operations scale until the combined working set of the cores on a socket exceeds the socket's L3 cache size. On the AMD machine, we observe an additional effect for smaller regions at high core counts: in this machine, each socket has a 1 MB directory for tracking ownership of that socket's physical memory, which this benchmark quickly exceeds.

These benchmarks show some of the limitations to how far we can push conflict-freedom before it no longer aligns with scalability. Nevertheless, even in the worst cases demonstrated by these benchmarks, conflict-free operations both perform and scale far better than conflicted operations.

3.4 Summary

Despite some limitations, conflict-freedom is a good predictor of linear scalability in practice. Most software has good cache locality and high cache hit rates both because this is crucial for sequential performance, and because it's in the interest of CPU manufacturers to design caches that fit typical working sets. For workloads that exceed cache capacity, NUMA-aware allocation spreads physical memory use across sockets and DRAM controllers, partitioning physical memory access, distributing the DRAM bottleneck, and giving cores greater aggregate DRAM bandwidth.

Chapter 9 will return to hard numbers on real hardware to show that conflict-free implementations of commutative interfaces enable software to scale not just at the level of memory microbenchmarks, but at the level of an entire OS kernel and its applications.

Four

The scalable commutativity rule

This chapter addresses two questions: What is the precise definition of the scalable commutativity rule, and why is the rule true? We answer these questions using a formalism based on abstract actions, histories, and implementations, combined with the empirical result of the previous chapter. This formalism relies on a novel form of commutativity, *SIM commutativity*, whose generality makes it possible to broadly apply the scalable commutativity rule to complex software interfaces. Our constructive proof of the scalable commutativity rule also sheds some light on how real conflict-free implementations might be built, though the actual construction is not very practical.

4.1 Actions

Following earlier work [37], we model a system execution as a sequence of *actions*, where an action is either an *invocation* or a *response*. In the context of an operating system, an invocation represents a system call with arguments (such as getpid() or open("file", O_RDWR)) and a response represents the corresponding return value (a PID or a file descriptor). Invocations and responses are paired. Each invocation is made by a specific thread, and the corresponding response is returned to the same thread. An action thus comprises (1) an operation class (e.g., which system call is being invoked); (2) operation arguments (for invocations) or a return value (for responses); (3) the relevant thread; and (4) a tag for uniqueness. We'll write invocations as left half-circles A ("invoke A") and responses as right half-circles A ("respond A"), where the letters match invocations and their responses. Color and vertical offset differentiate threads: A and B are invocations on different threads.

A system execution is called a *history*. For example

$$H = A\ B\ C\ A\ C\ B\ D\ D\ E\ E\ F\ G\ H\ F\ H\ G$$

consists of eight invocations and eight corresponding responses across three different threads. We'll consider only *well-formed* histories, in which each thread's actions form a sequence of invocation–response pairs. H above is well-formed; checking this for the red thread t, we see

that the thread-restricted subhistory $H|t$ = Ⓐ Ⓐ Ⓓ Ⓓ Ⓗ Ⓗ formed by selecting t's actions from H alternates invocations and responses as one would want. In a well-formed history, each thread has at most one outstanding invocation at every point.

The *specification* distinguishes whether or not a history is "correct." A specification $\mathscr{S}$ is a prefix-closed set of well-formed histories. The set's contents depend on the system being modeled; for example, if $\mathscr{S}$ specified a Unix-like OS, then [Ⓐ = getpid(), Ⓐ = 0] $\notin \mathscr{S}$, since no Unix thread can have PID 0. Our definitions and proof require that some specification exists, but we aren't concerned with how it is constructed.

4.2 SIM commutativity

Commutativity captures the idea that the order of a set of actions "doesn't matter." This happens when the caller of an interface cannot distinguish the order in which events actually occurred, either through those actions' responses or through any possible future actions. To formalize this inability to distinguish orders, we use the specification. A set of actions commutes in some context when the specification is indifferent to the execution order of that set. This means that any response valid for one order of the commutative set is valid for any order of the commutative set, and likewise any response invalid for one order is invalid for any order. The rest of this section formalizes this intuition as *SIM commutativity*.

Our definition has two goals: state dependence and interface basis. *State dependence* means SIM commutativity must to capture when operations commute in some states, even if those same operations do not commute in other states. This is important because it allows the rule to apply to a much broader range of situations than traditional non-state-dependent notions of commutativity. For example, few OS system calls unconditionally commute in every state, but many system calls commute in restricted states. Consider POSIX's open call. In general, two calls to open("a", O_CREAT|O_EXCL) don't commute: one call will create the file and the other will fail because the file already exists. However, two such calls do commute if called from processes with different working directories; or if the file "a" already exists (both calls will return the same error). State dependence makes it possible to distinguish these cases, even though the operations are the same in each. This, in turn, means the scalable commutativity rule can tell us that scalable implementations exist in all of these commutative cases.

Interface basis means SIM commutativity must evaluate the consequences of execution order using only the specification, without reference to any particular implementation. Since our goal is to reason about *possible* implementations, it's necessary to capture the scalability inherent in the interface itself. This in turn makes it possible to use the scalable commutativity rule early in software development, during interface design and initial implementation.

The right definition for commutativity that achieves both of these goals is a little tricky, so we build it up in two steps.

Definition. An action sequence, or region, H' is a *reordering* of an action sequence H when $H|t = H'|t$ for every thread t. This is, regions H and H' contain the same invocations and responses in the same order for each individual thread, but may interleave threads differently. If H = A B A C B C, then B B A A C C is a reordering of H, but B C B C A A is not, since it doesn't respect the order of actions in H's red thread.

Definition. Consider a history $H = X \parallel Y$ (where $\parallel$ concatenates action sequences). Y *SI-commutes* in H when given any reordering Y' of Y, and any action sequence Z,

$$X \parallel Y \parallel Z \in \mathscr{S} \quad \text{if and only if} \quad X \parallel Y' \parallel Z \in \mathscr{S}.$$

This definition captures the state dependence and interface basis we need. The action sequence X puts the system into the state we wish to consider, without specifying any particular representation of that state (which would depend on an implementation). Switching regions Y and Y' requires that the exact responses in Y remain valid according to the specification even if Y is reordered. The presence of region Z in both histories requires that reorderings of actions in region Y are indistinguishable by future operations.

Unfortunately, SI commutativity doesn't suffice for our needs because it is *non-monotonic*. Given an action sequence $X \parallel Y_1 \parallel Y_2$, it is possible for $Y_1 \parallel Y_2$ to SI-commute after region X even though Y_1 without Y_2 does not. For example, consider a get/set interface and the history

$$Y = [\underbrace{\text{A} = \text{set}(1), \text{A}, \text{B} = \text{set}(2), \text{B}}_{Y_1}, \underbrace{\text{C} = \text{set}(2), \text{C}}_{Y_2}].$$

Y SI-commutes because set returns nothing and every reordering sets the underlying value to 2, so future get operations cannot distinguish reorderings of Y. However, its prefix Y_1 alone does not SI-commute because some orders set the value to 1 and some to 2, so future get operations can distinguish them. Whether or not Y_1 will ultimately form part of a commutative region thus depends on *future* operations! This is usually incompatible with scalability: operations in Y_1 must "plan for the worst" by remembering their order, even if they ultimately form part of a SI-commutative region. Requiring monotonicity eliminates this problem.

Definition. An action sequence Y *SIM-commutes* in a history $H = X \parallel Y$ when for any *prefix* P of any reordering of Y (including $P = Y$), P SI-commutes in $X \parallel P$.

Returning to the get/set example, while the sequence Y given in the example SI-commutes (in any history), Y does *not* SIM-commute because its prefix Y_1 is not SI commutative.

Like SI commutativity, SIM commutativity captures state dependence and interface basis. Unlike SI commutativity, SIM commutativity excludes cases where the commutativity of a region changes depending on future operations and, as we show below, suffices to prove the scalable commutativity rule.

4.3 Implementations

To reason about the scalability of an implementation of an interface, we need to model implementations in enough detail to tell whether different threads' "memory accesses" are conflict-free. We represent an implementation as a step function: given a state and an invocation, it produces a new state and a response. We can think of this step function as being invoked by a driver algorithm (a *scheduler* in operating systems parlance) that repeatedly picks which thread to take a step on and passes state from one application of the step function to the next. Special YIELD responses let the step function request that the driver "run" a different thread and let us represent concurrent overlapping operations and blocking operations.

We begin by defining three sets:

- S is the set of implementation states.
- I is the set of valid invocations, including CONTINUE.
- R is the set of valid responses, including YIELD.

Definition. An *implementation* m is a function in $S \times I \mapsto S \times R$. Given an old state and an invocation, the implementation produces a new state and a response. The response must have the same thread as the invocation. A YIELD response indicates that a real response for that thread is not yet ready, and gives the driver the opportunity to take a step on a different thread without a real response from the current thread. CONTINUE invocations give the implementation an opportunity to complete an outstanding request on that thread (or further delay its response).[1]

An implementation *generates* a history when calls to the implementation (perhaps including CONTINUE invocations) could potentially produce the corresponding history. For example, this sequence shows an implementation m generating a history A B B A:

- $m(s_0,$ A$) = \langle s_1,$ YIELD$\rangle$
- $m(s_1,$ B$) = \langle s_2,$ YIELD$\rangle$
- $m(s_2,$ CONTINUE$) = \langle s_3,$ YIELD$\rangle$
- $m(s_3,$ CONTINUE$) = \langle s_4,$ B$\rangle$
- $m(s_4,$ CONTINUE$) = \langle s_5,$ A$\rangle$

[1] There are restrictions on how a driver can choose arguments to the step function. We assume, for example, that it passes a CONTINUE invocation for thread t if and only if the last step on t returned YIELD. Furthermore, since implementations are functions, they must be deterministic. We could model implementations instead as relations, allowing non-determinism, though this would complicate later arguments somewhat.

The state is threaded from step to step; invocations appear as arguments and responses as return values. The generated history consists of the invocations and responses, in order, with YIELDS and CONTINUES removed.

An implementation m is *correct* for some specification $\mathscr{S}$ when the responses it generates are always allowed by the specification. Specifically, let H be a valid history that can be generated by m. We say that m is correct when every such H is in $\mathscr{S}$. Note that a correct implementation need not be capable of generating every possible legal response or every possible history in $\mathscr{S}$; it's just that every response it does generate is legal.

To reason about conflict freedom, we must peek into implementation states, identify reads and writes, and check for access conflicts. Let each state $s \in S$ be a tuple $\langle s.0, \ldots, s.j \rangle$, and let $s_{i \leftarrow x}$ indicate component replacement: $s_{i \leftarrow x} = \langle s.0, \ldots, s.(i-1), x, s.(i+1), \ldots, s.j \rangle$. Now consider an implementation step $m(s, a) = \langle s', r \rangle$. This step *writes* state component i when $s.i \neq s'.i$. It *reads* state component i when $s.i$ may affect the step's behavior; that is, when for some y,

$$m(s_{i \leftarrow y}, a) \neq \left\langle s'_{i \leftarrow y}, r \right\rangle.$$

Two implementation steps have an *access conflict* when they are on different threads and one writes a state component that the other either writes or reads. This notion of access conflicts maps directly onto the read and write access conflicts on real shared-memory machines explored in chapter 3. A set of implementation steps is *conflict-free* when no pair of steps in the set has an access conflict; that is, no thread's steps read or write a state component written by another thread's steps.

4.4 Rule

We can now formally state the scalable commutativity rule. Assume a specification $\mathscr{S}$ with a correct reference implementation M. Consider a history $H = X \parallel Y$ where Y SIM-commutes in H, and where M can generate H. Then there exists a correct implementation m of $\mathscr{S}$ whose steps in the Y region of H are conflict-free. Empirically, conflict-free operations scale linearly on modern multicore hardware (chapter 3), so, given reasonable workload assumptions, m scales in the Y region of H.

4.5 Example

Before we turn to why the scalable commutativity rule is true, we'll first illustrate how the rule helps designers think about interfaces and implementations, using reference counters as a case study.

In its simplest form, a reference counter has two operations, inc and dec, which respectively increment and decrement the value of the counter and return its new value. We'll also consider

a third operation, iszero, which returns whether the reference count is zero. Together, these operations and their behavior define a reference counter specification $\mathscr{S}_{\text{rc}}$. $\mathscr{S}_{\text{rc}}$ has a simple reference implementation using a shared counter, which we can represent formally as

$$m_{\text{rc}}(s, \mathsf{inc}) \equiv \langle s+1,\ s+1\rangle \qquad m_{\text{rc}}(s, \mathsf{dec}) \equiv \langle s-1,\ s-1\rangle \qquad m_{\text{rc}}(s, \mathsf{iszero}) \equiv \langle s,\ s=0\rangle$$

Consider a reference counter that starts with a value of 2 and the history

$$H = [\underbrace{\text{A} = \mathsf{iszero}(), \text{A} = \mathsf{false}, \text{B} = \mathsf{iszero}(), \text{B} = \mathsf{false},}_{H_{\mathsf{AB}}} \underbrace{\text{C} = \mathsf{dec}(), \text{C} = 1, \text{D} = \mathsf{dec}(), \text{D} = 0}_{H_{\mathsf{CD}}}]$$

The region H_{AB} SIM commutes in H. Thus, by the rule, there is an implementation of $\mathscr{S}_{\text{rc}}$ that is conflict-free for H_{AB}. In fact, this is already true of the shared counter reference implementation m_{rc} because iszero reads the state, but does not change it. On the other hand, H_{CD} does not SIM commute in H, and therefore the rule does not apply (indeed, no correct implementation can be conflict-free for H_{CD}).

The rule suggests a direction to make H_{CD} conflict-free: if we modify the specification so that dec (and inc) return nothing, then these modified operations *do* commute (more precisely: any region consisting exclusively of these operations commutes in any history). With this modified specification, $\mathscr{S}'_{\text{rc}}$, the caller must invoke iszero to detect when the object is no longer referenced, but in many cases this delayed zero detection is acceptable and represents a desirable trade-off.

The equivalent history with this modified specification is

$$H' = [\overbrace{\underbrace{\text{A} = \mathsf{iszero}(), \text{A} = \mathsf{false}, \text{B} = \mathsf{iszero}(), \text{B} = \mathsf{false},}_{H'_{\mathsf{AB}}} \text{C} = \mathsf{dec}(), \text{C}}^{H'_{\mathsf{ABC}}}, \underbrace{\text{D} = \mathsf{dec}(), \text{D}}_{H'_{\mathsf{CD}}}]$$

Unlike H_{CD}, H'_{CD} SIM commutes. And, accordingly, there is an implementation of $\mathscr{S}'_{\text{rc}}$ that is conflict-free for H'_{CD}. By using per-thread counters, we can construct such an implementation. Each dec can modify its local counter, while iszero sums the per-thread values. Per-thread and per-core sharding of data structures like this is a common and long-standing pattern in scalable implementations.

The rule highlights at least one more opportunity in this history. H'_{ABC} *also* SIM commutes (still assuming an initial count of 2). However, the implementation given above for H'_{CD} is *not* conflict-free for H'_{ABC} because C will write one component of the state that is read and summed by A (and B). But, again, there is a conflict-free implementation based on adding a Boolean iszero snapshot to the state. iszero simply returns this snapshot. When dec's per-thread value reaches zero, it can read and sum all per-thread values and update the iszero snapshot if necessary.

```
m_ns(s, a) ≡
  If head(s.h) = a:
    r ← CONTINUE
  else if a = YIELD and head(s.h) is a response
            and thread(head(s.h)) = thread(a):
    r ← head(s.h)                                          // replay s.h
  else if s.h ≠ EMULATE:                 // H complete or input diverged
    H′ ← an invocation sequence consistent with s.h
    For each invocation x in H′:
      ⟨s.refstate, _⟩ ← M(s.refstate, x)
    s.h ← EMULATE                          // switch to emulation mode
  If s.h = EMULATE:
    ⟨s.refstate, r⟩ ← M(s.refstate, a)
  else:                                                   // replay mode
    s.h ← tail(s.h)
  Return ⟨s, r⟩
```

Figure 4-1: Constructed *non-scalable* implementation m_{ns} for history H and reference implementation M.

These two implementations of $\mathscr{S}'_{\text{rc}}$ are fundamentally different. Which is most desirable depends on whether the workload is expected to be write-heavy (mostly inc and dec) or read-heavy (mostly iszero). Thus an implementer must determine what opportunities to scale exist, decide which are likely to be the most valuable, and choose the implementation that scales in those situations.

In section 8.1, we'll return to the topic of reference counters when we describe Refcache, a practical and broadly conflict-free implementation of a *lazy reference counter* similar to $\mathscr{S}'_{\text{rc}}$.

4.6 Proof

We derived implementations of the reference counter example by hand, but a general, constructive proof for the scalable commutativity rule is possible. The construction builds a conflict-free implementation m from an arbitrary reference implementation M and history $H = X \parallel Y$. The constructed implementation emulates the reference implementation and is thus correct for any history. Its *performance* properties, however, are specialized for H. For any history $X \parallel P$ where P is a prefix of a reordering of Y, the constructed implementation's steps in P are conflict-free. That is, within the SIM-commutative region, m scales.

To understand the construction, it helps to first imagine constructing a *non-scalable* implementation m_{ns} from the reference M. This non-scalable implementation begins in *replay mode*. As long as each invocation matches the next invocation in H, m_{ns} simply replays the corresponding responses from H, without invoking the reference implementation. If the

input invocations diverge from H, however, m_{ns} can no longer replay responses from H, so it enters *emulation mode*. This requires feeding M all previously received invocations to prepare its state. After this, m_{ns} responds by passing all invocations to the reference implementation and returning its responses.

A state s for m_{ns} contains two components. First, $s.h$ either holds the portion of H that remains to be replayed or has the value EMULATE, which denotes emulation mode. $s.h$ is initialized to H. Second, $s.\mathit{refstate}$ is the state of the reference implementation, and starts as the value of the reference implementation's initial state. Figure 4-1 shows how the simulated implementation works. We make several simplifying assumptions, including that m_{ns} receives CONTINUE invocations in a restricted way; these assumptions aren't critical for the argument. One line requires expansion, namely the choice of H' "consistent with $s.h$" when the input sequence diverges. This step calculates the prefix of H up to, but not including, $s.h$; excludes responses; and adds CONTINUE invocations as appropriate.

This implementation is correct—its responses for any history always match those from the reference implementation. But it isn't conflict-free. In replay mode, any two steps of m_{ns} conflict on accessing $s.h$. These accesses track which invocations have occurred; without them it would be impossible to later initialize the state of M. And this is where commutativity comes in. The action order in a SIM-commutative region doesn't matter by definition. Since the specification doesn't distinguish among orders, it is safe to initialize the reference implementation with the commutative actions *in a different order than they were received*. All future responses will still be valid according to the specification.

Figure 4-2 shows the construction of m, a version of M that scales over Y in $H = X \parallel Y$. m is similar to m_{ns}, but extends it with a *conflict-free mode* used to execute actions in Y. Its state is as follows:

- $s.h[t]$—a per-thread history. Initialized to $X \parallel$ COMMUTE $\parallel (Y|t)$, where the special COMMUTE action indicates the commutative region has begun.
- $s.\mathit{commute}[t]$—a per-thread flag indicating whether the commutative region has been reached. Initialized to FALSE.
- $s.\mathit{refstate}$—the reference implementation's state.

Each step of m in the commutative region *accesses only state components specific to the invoking thread*. This means that any two steps in the commutative region are conflict-free, and the scalable commutativity rule is proved. The construction uses SIM commutativity when initializing the reference implementation's state via H'. If the observed invocations diverge before the commutative region, then just as in m_{ns}, H' will exactly equal the observed invocations. If the observed invocations diverge in or after the commutative region, however, there's not enough information to recover the order of invocations. (The $s.h[t]$ components track which invocations have happened per thread, but not the order of those invocations

```
m(s, a) ≡
  t ← thread(a)
  If head(s.h[t]) = COMMUTE:                                  // enter conflict-free mode
    s.commute[t] ← TRUE;  s.h[t] ← tail(s.h[t])
  If head(s.h[t]) = a:
    r ← CONTINUE
  else if a = YIELD and head(s.h[t]) is a response
          and thread(head(s.h[t])) = t:
    r ← head(s.h[t])                                          // replay s.h
  else if s.h[t] ≠ EMULATE:                                   // H complete/input diverged
    H′ ← an invocation sequence consistent with s.h[*]
    For each invocation x in H′:
      ⟨s.refstate, _⟩ ← M(s.refstate, x)
    s.h[u] ← EMULATE for each thread u
  If s.h[t] = EMULATE:
    ⟨s.refstate, r⟩ ← M(s.refstate, a)
  else if s.commute[t]:                                       // conflict-free mode
    s.h[t] ← tail(s.h[t])
  else:                                                       // replay mode
    s.h[u] ← tail(s.h[u]) for each thread u
  Return ⟨s, r⟩
```

Figure 4-2: Constructed scalable implementation m for history H and reference implementation M.

between threads.) Therefore, H' might reorder the invocations in Y. SIM commutativity guarantees that replaying H' will nevertheless produce results indistinguishable from those of the actual invocation order, even if the execution diverges *within* the commutative region.[2]

4.7 Discussion

The rule and proof construction push state and history dependence to an extreme: the proof construction is specialized for a *single* commutative region. This can be mitigated by repeated application of the construction to build an implementation that scales over multiple commutative regions in a history, or for the union of many histories.[3] Nevertheless, the implementation

[2]We effectively have assumed that M, the reference implementation, produces the same results for any reordering of the commutative region. This is stricter than SIM commutativity, which places requirements on the specification, not the implementation. We also assumed that M is indifferent to the placement of CONTINUE invocations in the input history. Neither of these restrictions is fundamental, however. If during replay M produces responses that are inconsistent with the desired results, m could throw away M's state, produce a new H' with different CONTINUE invocations and/or commutative region ordering, and try again. This procedure must eventually succeed and does not change the conflict-freedom of m in the commutative region.

[3]This is possible because, once the constructed machine leaves the specialized region, it passes invocations directly to the reference and has the same conflict-freedom properties as the reference.

constructed by the proof is impractical and real implementations achieve broad scalability using different techniques, such as the ones this dissertation explores in chapter 8.

We believe such broad implementation scalability is made easier by broadly commutative interfaces. In broadly commutative interfaces, the arguments and system states for which a set of operations commutes often collapse into fairly well-defined classes (e.g., file creation might commute whenever the containing directories are different). In practice, implementations scale for whole classes of states and arguments, not just for specific histories.

On the other hand, there can be limitations on how broadly an implementation can scale. It is sometimes the case that a set of operations commutes in more than one class of situation, but no single implementation can scale for all classes. The reference counter example in section 4.5 hinted at this when we constructed several possible implementations for different situations, but never arrived at a broadly conflict-free one. As an example that's easier to reason about, consider an interface with two calls: $\mathsf{put}(x)$ records a sample with value x, and $\mathsf{max}()$ returns the maximum sample recorded so far (or $\mathsf{0}$). Suppose

$$H = [\overbrace{\mathsf{A} = \mathsf{put}(1), \mathsf{A}, \underbrace{\mathsf{B} = \mathsf{put}(1), \mathsf{B}}^{H_{\mathsf{AB}}}, \mathsf{C} = \mathsf{max}(), \mathsf{C} = 1}_{H_{\mathsf{BC}}}].$$

Both H_{AB} and H_{BC} SIM commute in H, but H overall is not SIM commutative. An implementation could store per-thread maxima reconciled by max and be conflict-free for H_{AB}. Alternatively, it could use a global maximum that put checked before writing. This is conflict-free for H_{BC}. But no correct implementation can be conflict-free across all of H. Since H_{AB} and H_{BC} together span H, that means no single implementation can be conflict-free for *both* H_{AB} and H_{BC}.

In our experience, real-world interface operations rarely demonstrate such mutually exclusive implementation choices. For example, the POSIX implementation in chapter 8 scales quite broadly, with only a handful of cases that would require incompatible implementations.

We hope to further explore this gap between the specificity of the formalized scalable commutativity rule and the generality of practical implementations. We'll return to this question and several other avenues for future work in chapter 10. However, as the rest of this dissertation shows, the rule is already an effective guideline for achieving practical scalability.

FIVE

Designing commutative interfaces

The rule facilitates scalability reasoning at the interface and specification level, and SIM commutativity lets us apply the rule to complex interfaces. This chapter demonstrates the interface-level reasoning enabled by the rule, using POSIX as a case study. Already, many POSIX operations commute with many other operations, a fact we will quantify in the following chapters; this chapter focuses on problematic cases to give a sense of the subtler issues of commutative interface design.

The following sections explore four general classes of changes that make operations commute in more situations, enabling more scalable implementations.

5.1 Decompose compound operations

Many POSIX APIs combine several operations into one, limiting the combined operation's commutativity. For example, fork both creates a new process and snapshots the current process's entire memory state, file descriptor state, signal mask, and several other properties. As a result, fork fails to commute with most other operations in the same process, such as memory writes, address space operations, and many file descriptor operations. However, applications often follow fork with exec, which undoes most of fork's sub-operations. With only fork and exec, applications are forced to accept these unnecessary sub-operations that limit commutativity.

POSIX has a little-known API called posix_spawn that addresses this problem by creating a process and loading an image directly (CreateProcess in Windows is similar). This is equivalent to fork/exec, but its specification eliminates the intermediate sub-operations. As a result, posix_spawn commutes with most other operations and permits a broadly scalable implementation.

Another example, stat, retrieves and returns many different attributes of a file simultaneously, which makes it non-commutative with operations on the same file that change any attribute returned by stat (such as link, chmod, chown, write, and even read). In practice, applications invoke stat for just one or two of the returned fields. An alternate API that

gave applications control of which field or fields were returned would commute with more operations and enable a more scalable implementation of stat, as we show in section 9.2.

POSIX has many other examples of compound return values. sigpending returns all pending signals, even if the caller only cares about a subset; and select returns all ready file descriptors, even if the caller needs only one ready FD.

5.2 Embrace specification non-determinism

POSIX's "lowest available FD" rule is a classic example of overly deterministic design that results in poor scalability. Because of this rule, open operations in the same process (and any other FD allocating operations) do not commute, since the order in which they execute determines the returned FDs. This constraint is rarely needed by applications and an alternate interface that could return any unused FD would allow FD allocation operations to commute and enable implementations to use well-known scalable allocation methods. We will return to this example, too, in section 9.2. Many other POSIX interfaces get this right: mmap can return any unused virtual address and creat can assign any unused inode number to a new file.

5.3 Permit weak ordering

Another common source of limited commutativity is strict ordering requirements between operations. For many operations, ordering is natural and keeps interfaces simple to use; for example, when one thread writes data to a file, other threads can immediately read that data. Synchronizing operations like this are naturally non-commutative. Communication interfaces, on the other hand, often enforce strict ordering, but may not need to. For instance, most systems order all messages sent via a local Unix domain socket, even when using SOCK_DGRAM, so any send and recv system calls on the same socket do not commute (except in error conditions). This is often unnecessary, especially in multi-reader or multi-writer situations, and an alternate interface that does not enforce ordering would allow send and recv to commute as long as there is both enough free space and enough pending messages on the socket, which would in turn allow an implementation of Unix domain sockets to support scalable communication (which we use in section 9.3).

5.4 Release resources asynchronously

A closely related problem is that many POSIX operations have global effects that must be visible before the operation returns. This is generally good design for usable interfaces, but for operations that release resources, this is often stricter than applications need and expensive to ensure. For example, writing to a pipe must deliver SIGPIPE immediately if there are no

read FDs for that pipe, so pipe writes do not commute with the last close of a read FD. This requires aggressively tracking the number of read FDs; a relaxed specification that promised to eventually deliver the SIGPIPE would allow implementations to use more scalable read FD tracking. Similarly, munmap does not commute with memory reads or writes of the unmapped region from other threads. Enforcing this requires non-scalable remote TLB shootdowns before munmap can return, even though depending on this behavior usually indicates a bug. An munmap (perhaps an madvise) that released virtual memory asynchronously would let the kernel reclaim physical memory lazily and batch or eliminate remote TLB shootdowns.

Six

Analyzing interfaces using Commuter

Fully understanding the commutativity of a complex interface is tricky, and achieving an implementation that avoids sharing when operations commute adds another dimension to an already difficult task. However, by leveraging the formality of the commutativity rule, developers can automate much of this reasoning. This chapter presents a systematic, test-driven approach to applying the commutativity rule to real implementations embodied in a tool named Commuter, whose components are shown in Figure 6-1.

First, Analyzer takes a symbolic model of an interface and computes precise conditions for when that interface's operations commute. Second, Testgen uses these conditions to generate concrete tests of sets of operations that commute according to the interface model, and thus should have a conflict-free implementation according to the commutativity rule. Third, Mtrace checks whether a particular implementation is conflict-free for each test case.

A developer can use these test cases to understand the commutative cases they should consider, to iteratively find and fix scalability issues in their code, or as a regression test suite to ensure scalability bugs do not creep into the implementation over time.

6.1 Analyzer

Analyzer automates the process of analyzing the commutativity of an interface, saving developers from the tedious and error-prone process of considering large numbers of interactions between complex operations. Analyzer takes as input a model of the behavior of an interface,

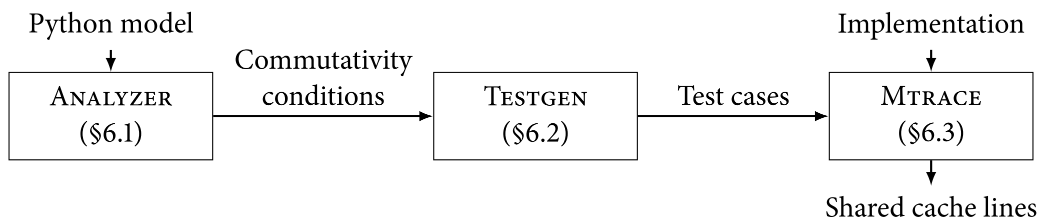

Figure 6-1: The components of Commuter.

written in a symbolic variant of Python, and outputs *commutativity conditions*: expressions in terms of arguments and state for exactly when sets of operations commute. A developer can inspect these expressions to understand an interface's commutativity or pass them to TESTGEN (section 6.2) to generate concrete examples of when interfaces commute.

Given the Python code for a model, ANALYZER uses symbolic execution to consider all possible behaviors of the interface model and construct complete commutativity conditions. Symbolic execution also enables ANALYZER to reason about the external behavior of an interface, rather than specifics of the model's implementation, and enables models to capture specification non-determinism (like creat's ability to choose any free inode) as under-constrained symbolic values.

6.1.1 Concrete commutativity analysis

Starting from an interface model, ANALYZER computes the commutativity condition of each multiset of operations of a user-specified size. To determine whether a set of operations commutes, ANALYZER executes the SIM commutativity test algorithm given in Figure 6-2. To begin with, we can think of this test in concrete (non-symbolic) terms as a test of whether a set of operations commutes starting from a specific initial state s0 and given specific operation arguments args.

This test is implemented by a function called commutes. commutes codifies the definition of SIM commutativity, except that it requires the specification to be sequentially consistent so it needn't interleave partial operations. Recall that Y SI-commutes in $H = X \parallel Y$ when, given any reordering Y' of Y and any action sequence Z,

$$X \parallel Y \parallel Z \in \mathscr{S} \quad \text{if and only if} \quad X \parallel Y' \parallel Z \in \mathscr{S}.$$

Further, for Y to SIM-commute in H, every prefix of every reordering of Y must SI-commute. In commutes, the initial state s0 serves the role of the prefix X: to put the system in some state. ops serves the role of Y (assuming sequential consistency) and the loop in commutes generates every Y', that is, all prefixes of all reorderings of Y. This loop performs two tests. First, the result equivalence test ensures that each operation gives the same response in all reorderings. Finally, the state equivalence test serves the role of the future actions, Z, by requiring all prefixes of all reorderings to converge on states that are indistinguishable by future operations.

Since commutes substitutes state equivalence in place of considering all possible future operations (which would be difficult with symbolic execution), it's up to the model's author to define state equivalence as whether two states are externally indistinguishable. This is standard practice for high-level data types (e.g., two sets represented as trees could be equal even if they are balanced differently). For the POSIX model we present in chapter 7, only a few types need special handling beyond what ANALYZER's high-level data types already provide.

```python
def commutes(s0, ops, args):
  states = {frozenset(): s0}
  results = {}

  # Generate all (non-empty) prefixes of all reorderings of ops
  todo = list((op,) for op in ops)
  while todo:
    perm = todo.pop()

    # Execute next operation in this permutation
    past, op = perm[:-1], perm[-1]
    s = states[frozenset(past)].copy()
    r = op(s, args[op])

    # Test for result equivalence
    if op not in results:
      results[op] = r
    elif r != results[op]:
      return False

    # Test for state equivalence
    if frozenset(perm) not in states:
      states[frozenset(perm)] = s
    elif s != states[frozenset(perm)]:
      return False

    # Queue all extensions of perm
    todo.extend(perm + (nextop,) for nextop in ops if nextop not in perm)
  return True
```

Figure 6-2: The SIM commutativity test algorithm. s0 is the initial state, ops is the list of operations to test for commutativity, and args gives the arguments to pass to each operation. For clarity, this implementation assumes all values in ops are distinct.

The commutes algorithm can be optimized by observing that if two permutations of the same prefix reach the same state, only one needs to be considered further. This optimization gives commutes a pleasing symmetry: it becomes equivalent to exploring all n step paths from $\langle 0,\ 0,\ \ldots\rangle$ to $\langle 1,\ 1,\ \ldots\rangle$ in an n-cube, where each unit step is an operation and each vertex is a state.

6.1.2 Symbolic commutativity analysis

So far, we've considered only how to determine if a set of operations commutes for a specific initial state and specific arguments. Ultimately, we're interested in the entire space of states and arguments for which a set of operations commutes. To find this, ANALYZER executes both the interface model and commutes *symbolically*, starting with an unconstrained symbolic initial state and unconstrained symbolic operation arguments. Symbolic execution lets ANALYZER efficiently consider *all possible* initial states and arguments and precisely determine the

```
SymInode    = tstruct(data  = tlist(SymByte),
                      nlink = SymInt)
SymIMap     = tdict(SymInt, SymInode)
SymFilename = tuninterpreted('Filename')
SymDir      = tdict(SymFilename, SymInt)

class POSIX(tstruct(fname_to_inum = SymDir,
                    inodes        = SymIMap)):
  @symargs(src=SymFilename, dst=SymFilename)
  def rename(self, src, dst):
    if not self.fname_to_inum.contains(src):
      return (-1, errno.ENOENT)
    if src == dst:
      return 0
    if self.fname_to_inum.contains(dst):
      self.inodes[self.fname_to_inum[dst]].nlink -= 1
    self.fname_to_inum[dst] = self.fname_to_inum[src]
    del self.fname_to_inum[src]
    return 0
```

Figure 6-3: A simplified version of our rename model.

```
def commutes2(s0, opA, argsA, opB, argsB):
  # Run reordering [opA, opB] starting from s0
  sAB = s0.copy()
  rA  = opA(sAB, *argsA)
  rAB = opB(sAB, *argsB)

  # Run reordering [opB, opA] starting from s0
  sBA = s0.copy()
  rB  = opB(sBA, *argsB)
  rBA = opA(sBA, *argsA)

  # Test commutativity
  return rA == rBA and rB == rAB and sAB == sBA
```

Figure 6-4: The SIM commutativity test algorithm specialized to two operations.

(typically infinite) set of states and arguments for which the operations commute (that is, for which commutes returns True).

Figure 6-3 gives an example of how a developer could model the rename operation in Analyzer. The first five lines declare symbolic types used by the model (tuninterpreted declares a type whose values support only equality). The POSIX class, itself a symbolic type, represents the *system state* of the file system and its methods implement the interface operations to be tested. The implementation of rename itself is straightforward. Indeed, the familiarity of Python and ease of manipulating state were part of why we chose it over abstract specification languages.

To explore how Analyzer analyzes rename, we'll use the version of commutes given in

Figure 6-4, which is specialized for pairs of operations. In practice, we typically analyze pairs of operations rather than larger sets because larger sets take exponentially longer to analyze and rarely reveal problems beyond those already revealed by pairs.

By symbolically executing commutes2 for two rename operations, rename(a, b) and rename(c, d), ANALYZER computes that these operations commute if any of the following hold:

- Both source files exist, and the file names are all different (a and c exist, and a, b, c, d all differ).
- One rename's source does not exist, and it is not the other rename's destination (either a exists, c does not, and b≠c, or c exists, a does not, and d≠a).
- Neither a nor c exists.
- Both calls are self-renames (a=b and c=d).
- One call is a self-rename of an existing file (a exists and a=b, or c exists and c=d) and it's not the other call's source (a≠c).
- Two hard links to the same inode are renamed to the same new name (a and c point to the same inode, a≠c, and b=d).

Despite rename's seeming simplicity, ANALYZER's symbolic execution systematically explores its hidden complexity, revealing the many combinations of state and arguments for which two rename calls commute. Here we see again the value of SIM commutativity: every condition above except the self-rename case depends on state and would not have been captured by a traditional, non-state-sensitive definition of commutativity.

Figure 6-5 illustrates the symbolic execution of commute2 that arrives at these conditions. By and large, this symbolic execution proceeds like regular Python execution, except when it encounters a conditional branch on a symbolic value (such as any if statement in rename). Symbolic execution always begins with an empty symbolic *path condition*. To execute a conditional branch on a symbolic value, ANALYZER uses an SMT solver to determine whether that symbolic value can be true, false, or either, given the path condition accumulated so far. If the branch can go both ways, ANALYZER logically forks the symbolic execution and extends the path conditions of the two forks with the constraints that the symbolic value must be true or false, respectively. These growing path conditions thereby constrain further execution on the two resulting code paths.

The four main regions of Figure 6-5 correspond to the four calls to rename from commutes2 as it tests the two different reorderings of the two operations. Each call region shows the three conditional branches in rename. The first call forks at every conditional branch because the state and arguments are completely unconstrained at this point; ANALYZER therefore explores every code path through the first call to rename. The second call forks similarly.

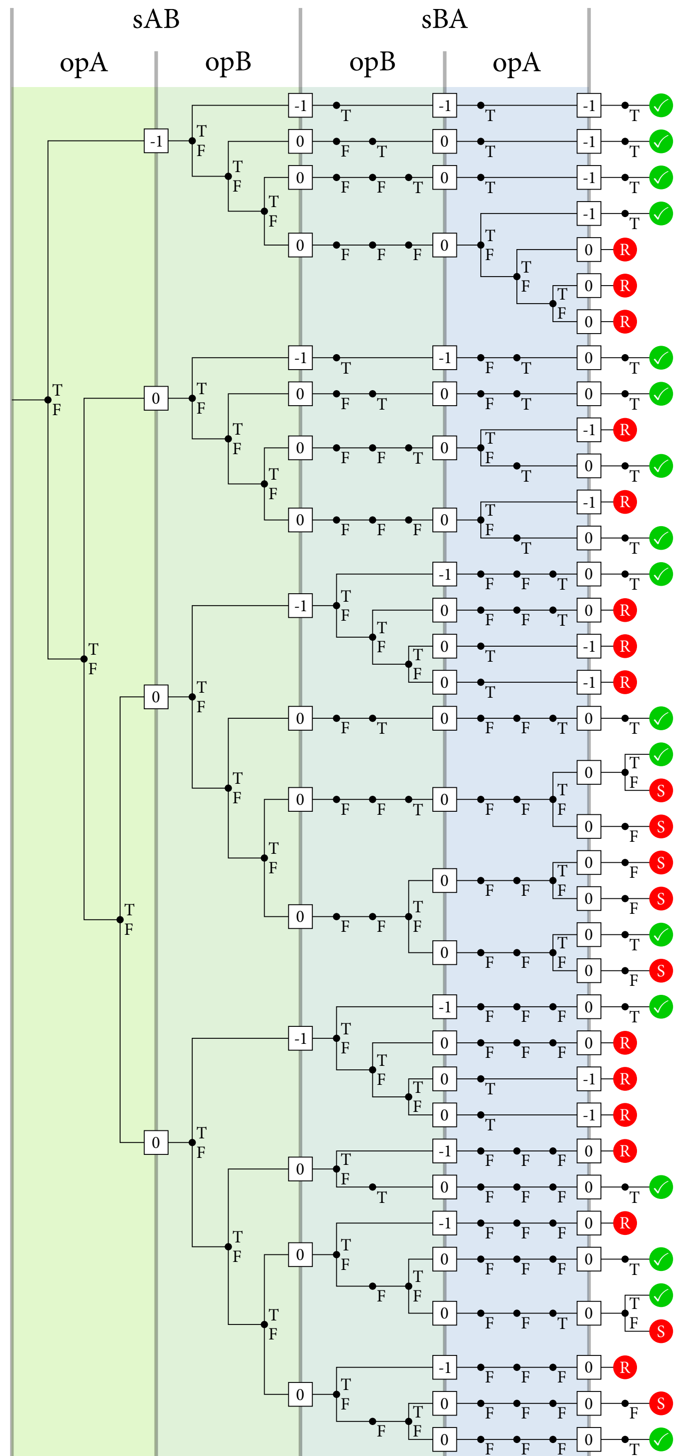

Figure 6-5: Symbolic execution tree of commutes2 for rename/rename. Each node represents a conditional branch on a symbolic value. The terminals at the right indicate whether each path constraint yields a commutative execution of the two operations (✓), or, if not, whether it diverged on return values (R) or final state (S).

The third and fourth calls generally do not fork; by this point, the symbolic values of s0, argsA, and argsB are heavily constrained by the path condition produced by the first two calls. As a result, these calls are often forced to make the same branch as the corresponding earlier call.

Finally, after executing both reorderings of rename/rename, commutes2 tests their commutativity by checking if each operation's return value is equivalent in both permutations and if the system states reached by both permutations are equivalent. This, too, is symbolic and may fork execution if it's still possible for the pair of operations to be either commutative or non-commutative (Figure 6-5 contains two such forks).

Together, the set of path conditions that pass this final commutativity test are the commutativity condition of this pair of operations. Barring SMT solver time-outs, the disjunction of the path conditions for which commutes2 returns True captures the precise and complete set of initial system states and operation arguments for which the operations commute.

As this example shows, when system calls access shared, mutable state, reasoning about every commutative case by hand can become difficult. Developers can easily overlook cases, both in their understanding of an interface's commutativity, and when making their implementation scale for commutative cases. ANALYZER automates reasoning about all possible system states, all possible sets of operations that can be invoked, and all possible arguments to those operations.

6.2 TESTGEN

While a developer can examine the commutativity conditions produced by ANALYZER directly, for complex interfaces these formulas can be large and difficult to decipher. Further, real implementations are complex and likely to contain unintentional sharing, even if the developer understands an interface's commutativity. TESTGEN takes the first step to helping developers apply commutativity to real implementations by converting ANALYZER's commutativity conditions into concrete test cases.

To produce a test case, TESTGEN computes a satisfying assignment for the corresponding commutativity condition. The assignment specifies concrete values for every symbolic variable in the model, such as the fname_to_inum and inodes data structures and the rename arguments shown in Figure 6-3. TESTGEN then invokes a model-specific function on the assignment to produce actual C test case code. For example, one test case that TESTGEN generates is shown in Figure 6-6. The test case includes setup code that configures the initial state of the system and a set of functions to run on different cores. Every TESTGEN test case should have a conflict-free implementation.

The goal of these test cases is to expose potential scalability problems in an implementation, but it is impossible for TESTGEN to know exactly what inputs might trigger conflicting memory accesses. Thus, as a proxy for achieving good coverage on the implementation, TESTGEN aims to achieve good coverage of the Python model.

```
void setup_rename_rename_path_ec_test0(void) {
  close(open("__i0", O_CREAT|O_RDWR, 0666));
  link("__i0", "f0");
  link("__i0", "f1");
  unlink("__i0");
}
int test_rename_rename_path_ec_test0_op0(void) {
  return rename("f0", "f0");
}
int test_rename_rename_path_ec_test0_op1(void) {
  return rename("f1", "f0");
}
```

Figure 6-6: An example test case for two rename calls generated by TESTGEN for the model in Figure 6-3.

We consider two forms of coverage. The first is the standard notion of path coverage, which TESTGEN achieves by relying on ANALYZER's symbolic execution. ANALYZER produces a separate path condition for every possible code path through a set of operations. However, even a single path might encounter conflicts in interestingly different ways. For example, the code path through two pwrites is the same whether they're writing to the same offset or different offsets, but the access patterns are very different. To capture different conflict conditions as well as path conditions, we introduce a new notion called *conflict coverage*. Conflict coverage exercises all possible access patterns on shared data structures: looking up two distinct items from different operations, looking up the same item, etc. TESTGEN approximates conflict coverage by concolically executing *itself* to enumerate distinct tests for each path condition. TESTGEN starts with the constraints of a path condition from ANALYZER, tracks every symbolic expression forced to a concrete value by the model-specific test code generator, negates any equivalent assignment of these expressions from the path condition, and generates another test, repeating this process until it exhausts assignments that satisfy the path condition or the SMT solver fails. Since path conditions can have infinitely many satisfying assignments (e.g., there are infinitely many calls to read with different FD numbers that return EBADF), TESTGEN partitions most values in *isomorphism groups* and considers two assignments equivalent if each group has the same pattern of equal and distinct values in both assignments. For our POSIX model, this bounds the number of enumerated test cases.

These two forms of coverage ensure that the test cases generated by TESTGEN will cover all possible paths and data structure access patterns in the model, and to the extent that the implementation is structured similarly to the model, should achieve good coverage for the implementation as well. As we demonstrate in chapter 7, TESTGEN produces a total of 26,238 test cases for our model of 18 POSIX system calls, and these test cases find scalability issues in the Linux ramfs file system and virtual memory system.

6.3 Mtrace

Finally, Mtrace runs the test cases generated by Testgen on a real implementation and checks that the implementation is conflict-free for every test. If it finds a violation of the commutativity rule—a test whose commutative operations are not conflict-free—it reports which variables were shared and what code accessed them. For example, when running the test case shown in Figure 6-6 on a Linux ramfs file system, Mtrace reports that the two functions make conflicting accesses to the dcache reference count and lock, which limits the scalability of those operations.

Mtrace runs the entire operating system in a modified version of qemu [5]. At the beginning of each test case, it issues a hypercall to qemu to start recording memory accesses, and then executes the test operations on different virtual cores. During test execution, Mtrace logs all reads and writes by each core, along with information about the currently executing kernel thread, to filter out irrelevant conflicts by background threads or interrupts. After execution, Mtrace analyzes the log and reports all conflicting memory accesses, along with the C data type of the accessed memory location (resolved from DWARF [28] information and logs of every dynamic allocation's type) and stack traces for each conflicting access.

6.4 Implementation

We built a prototype implementation of Commuter's three components. Analyzer and Testgen consist of 3,050 lines of Python code, including the symbolic execution engine, which uses the Z3 SMT solver [24] via Z3's Python bindings. Mtrace consists of 1,594 lines of code changed in qemu, along with 612 lines of code changed in the guest Linux kernel (to report memory type information, context switches, etc.). Another program, consisting of 2,865 lines of C++ code, processes the log file to find and report memory locations that are shared between different cores for each test case.

Seven

Conflict-freedom in Linux

To understand whether Commuter is useful to kernel developers, we modeled several POSIX file system and virtual memory calls in Commuter, then used this both to evaluate Linux's scalability and to develop a scalable file and virtual memory system for our sv6 research kernel. The rest of this chapter focuses on Linux and uses this case study to answer the following questions:

- How many test cases does Commuter generate, and what do they test?
- How good are current implementations of the POSIX interface? Do the test cases generated by Commuter find cases where current implementations don't scale?

In the next chapter, we'll use this same POSIX model to guide the implementation of a new operating system kernel, sv6.

7.1 POSIX test cases

To answer the first question, we developed a simplified model of the POSIX file system and virtual memory APIs in Commuter. The model covers 18 system calls, and includes inodes, file names, file descriptors and their offsets, hard links, link counts, file lengths, file contents, file times, pipes, memory-mapped files, anonymous memory, processes, and threads. Our model also supports nested directories, but we disable them because Z3 does not currently handle the resulting constraints. We restrict file sizes and offsets to page granularity; for pragmatic reasons, some kernel data structures are designed to be conflict-free for offsets on different pages, but offsets within a page conflict. Commuter generates a total of 26,238 test cases from our model. Generating the test cases and running them on both Linux and sv6 takes a total of 16 minutes on the machine described in section 9.1.

The model implementation and its model-specific test code generator are 596 and 675 lines of Python code, respectively. Figure 6-3 showed a part of our model, and Figure 6-6 gave an example test case generated by Commuter. We verified that all test cases return the expected results on both Linux and sv6.

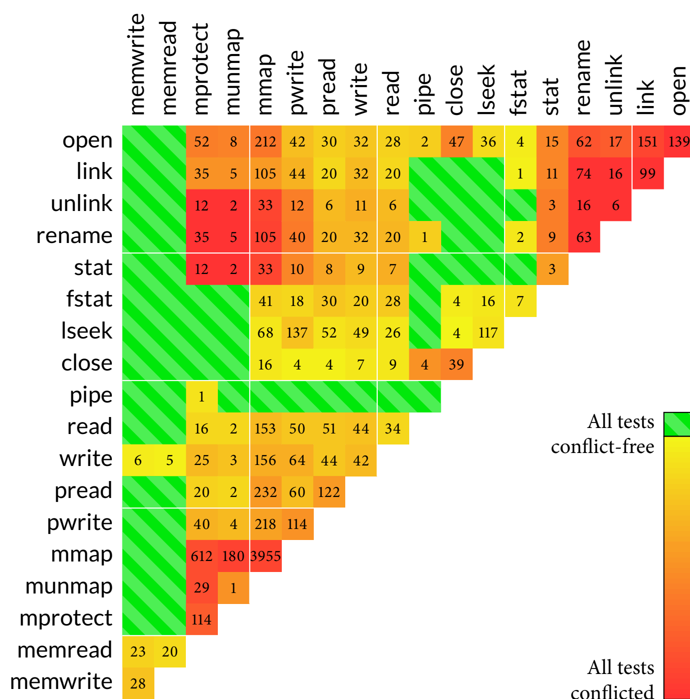

Figure 7-1: Conflict-freedom of commutative system call pairs in Linux, showing the fraction and absolute number of test cases generated by COMMUTER that are *not* conflict-free for each system call pair. One example test case was shown in Figure 6-6.

7.2 Linux conflict-freedom

To evaluate the scalability of existing file and virtual memory systems, we used MTRACE to check the above test cases against Linux kernel version 3.8. Linux developers have invested significant effort in making the file system scale [11], and it already scales in many interesting cases, such as concurrent operations in different directories or concurrent operations on different files in the same directory that already exist [20]. We evaluated the ramfs file system because ramfs is effectively a user-space interface to the Linux buffer cache. Since exercising ramfs is equivalent to exercising the buffer cache and the buffer cache underlies all Linux file systems, this represents the best-case scalability for a Linux file system. Linux's virtual memory system, in contrast, involves process-wide locks that are known to limit its scalability and impact real applications [11, 16, 62].

Figure 7-1 shows the results. Out of 26,238 test cases, 9,032 cases, widely distributed across the system call pairs, were not conflict-free. This indicates that even a mature and reasonably scalable operating system implementation misses many cases that can be made to scale according to the commutativity rule.

A common source of access conflicts is shared reference counts. For example, most file

name lookup operations update the reference count on a struct dentry; the resulting write conflicts cause them to not scale. Similarly, most operations that take a file descriptor update the reference count on a struct file, making commutative operations such as two fstat calls on the same file descriptor not scale. Coarse-grained locks are another source of access conflicts. For instance, Linux locks the parent directory for any operation that creates file names, even though operations that create distinct names generally commute. Similarly, we see that coarse-grained locking in the virtual memory system severely limits the conflict-freedom of address space manipulation operations. This agrees with previous findings [11, 16, 17], which demonstrated these problems in the context of several applications.

Figure 7-1 also reveals many previously unknown bottlenecks that may be triggered by future workloads or hardware.

The next chapter shows how these current and future bottlenecks can be removed in a practical implementation of POSIX.

Eight

Achieving conflict-freedom in POSIX

Given that many commutative operations are not conflict-free in Linux, is it feasible to build file systems and virtual memory systems that do achieve conflict-freedom for commutative operations? To answer this question, we designed and implemented a ramfs-like in-memory file system called ScaleFS and a virtual memory system called RadixVM for sv6, our research kernel based on xv6 [23]. Although it is in principle possible to make the same changes in Linux, we chose not to implement ScaleFS in Linux because ScaleFS's design would have required extensive changes throughout the Linux kernel. The designs of both RadixVM and ScaleFS were guided by the commutativity rule. For ScaleFS, we relied heavily on Commuter throughout development to guide its design and identify sharing problems in its implementation. RadixVM was built prior to Commuter, but was guided by manual reasoning about commutativity and conflicts (which was feasible because of the virtual memory system's relatively simple interface). We later validated RadixVM using Commuter.

Figure 8-1 shows the result of applying Commuter to sv6. In contrast with Linux, sv6 is conflict-free for nearly every commutative test case.

For a small number of commutative operations, sv6 is not conflict-free. Some appear to require implementations that would be incompatible with conflict-freedom in other cases. In these situations, we preferred the conflict-freedom of the cases we considered more important. Other non-conflict-free cases represent intentional engineering decisions in the interest of practical constraints on memory consumption and sequential performance. Complex software systems inevitably involve conflicting requirements, and scalability is no different. However, the presence of the rule forced us to explicitly recognize, evaluate, and justify where we made such trade-offs.

The rest of this chapter describes the design of sv6 and how it achieves conflict-freedom for cases where current file and virtual memory systems do not. This chapter starts with the design of Refcache, a scalable reference counting mechanism used throughout sv6. It then covers the designs of RadixVM and ScaleFS. Finally, it discusses operations that are difficult to make conflict-free without sacrificing other practical concerns.

sv6 (26,115 of 26,238 cases scale)

Figure 8-1: Conflict-freedom of commutative system call pairs in sv6.

8.1 Refcache: Scalable reference counting

Reference counting is critical to many OS functions and used throughout RadixVM and ScaleFS. RadixVM must reference count physical pages shared between address spaces, such as when forking a process, as well as nodes in its internal radix tree. Likewise, ScaleFS must reference count file descriptions, FD tables, and entries in various caches as well as scalably maintain other counts such as link counts. This section introduces Refcache, a novel reference counting scheme used by RadixVM and ScaleFS. Refcache implements space-efficient, lazy, scalable reference counting using per-core *reference delta caches*. Refcache targets uses of reference counting that can tolerate some latency in releasing reference counted resources and is particularly suited to uses where increment and decrement operations often occur on the same core (e.g., the same thread that allocated pages also frees them).

Despite the seeming simplicity of reference counting, designing a practical reference counting scheme that is conflict-free and scalable for most operations turns out to be an excellent exercise in applying the scalable commutativity rule to both implementation and interface design.

We first touched on this problem in section 4.5, where we introduced the notion of a *lazy reference counter* that separates manipulation from zero detection. Simple reference

counters have two operations, inc(*o*) and dec(*o*), which both return the new reference count. These operations trivially commute when applied to different objects, so we focus on the case of a single object. When applied to the same object, these operations *never* commute (any reordering will change their responses) and cannot be made conflict-free.

A better (and more common) interface returns nothing from inc and, from dec, returns only an indication of whether the count is zero. This is equivalent to the Linux kernel's atomic_inc and atomic_dec_and_test interface and is echoed in software ranging from Glib to Python. Here, a sequence of inc and dec operations commutes if (and only if) the count does not reach zero in any reordering. In this case, the results of the operations will be the same in all reorderings and, since reordering does not change the final sum, no future operations can distinguish different orders. Therefore, by the scalable commutativity rule, any such sequence has some conflict-free implementation. Indeed, supposing a *particular* history with a commutative region, an implementation can partition the count's value across cores such that no per-core partition drops to zero in the commutative region. However, a practical, *general-purpose* conflict-free implementation of this interface remains elusive.

Refcache pushes this interface reduction further, targeting reference counters that can tolerate some latency between when the count reaches zero and when the system detects that it's reached zero.

In Refcache, *both* inc and dec return nothing and hence always commute, even if the count reaches zero. A new review() operation finds all objects whose reference counts recently reached zero (and, in practice, calls their destructors). review does not commute in any sequence where *any* object's reference count has reached zero and its implementation conflicts on a small number of cache lines even when it does commute. However, unlike dec, the system can control when and how often it invokes review. sv6 invokes it only at 10ms intervals—several orders of magnitude longer than the time required by even the most expensive conflicts on current multicores. Refcache strikes a balance between broad conflict-freedom, strict adherence to the scalable commutativity rule, and practical implementation concerns, providing a general-purpose, efficient implementation of an interface suitable for many uses of referencing counting.

By separating count manipulation and zero detection, Refcache can batch increments and decrements and reduce cache line conflicts while offering an adjustable time bound on when an object will be garbage collected after its reference count drops to zero. inc and dec are conflict-free with high probability and review induces only a small constant rate of conflicts for global epoch maintenance.

Refcache inherits ideas from sloppy counters [11], Scalable NonZero Indicators (SNZI) [29], distributed counters [1], shared vs. local counts in Modula-2+ [26], and approximate counters [19]. All of these techniques speed up increment and decrement operations using per-core counters, but make different trade-offs between conflict-freedom, zero-detection cost, and space. With the exception of SNZIs, these techniques do not detect when a count reaches

zero, so the system must poll each counter for this condition. SNZIs detect when a counter reaches zero immediately, but at the cost of conflicts when a counter's value is small. Refcache balances these extremes, enabling conflict-free operations and efficient zero-detection, but with a short delay. Furthermore, in contrast with sloppy, SNZI, distributed, and approximate counters, Refcache requires space proportional to the sum of the number of reference counted objects and the number of cores, rather than the product, and the per-core overhead can be adjusted to trade off space and scalability by controlling the reference delta cache collision rate. Space overhead is particularly important for RadixVM, which must reference count every physical page; at large core counts, other scalable reference counters would require more than half of physical memory just to track the remaining physical memory.

8.1.1 Basic Refcache

In Refcache, each reference counted object has a global reference count (much like a regular reference count) and each core also maintains a local, fixed-size cache of deltas to objects' reference counts. Incrementing or decrementing an object's reference count modifies only the local, cached delta and review periodically flushes this delta to the object's global reference count. The true reference count of an object is thus the sum of its global count and any local deltas for that object found in the per-core delta caches. The value of the true count is generally unknown, but we assume that once the true count drops to zero, it will remain zero (in the absence of weak references, which we discuss later). Refcache depends on this stability to detect a zero true count after some delay.

To detect a zero true reference count, Refcache divides time into periodic *epochs* during which each core calls review once to flush all of the reference count deltas in its cache and apply these updates to the global reference count of each object. The last core in an epoch to finish flushing its cache ends the epoch and all of the cores repeat this process after some delay (our implementation uses 10ms). Since these flushes occur in no particular order and the caches batch reference count changes, updates to the reference count can be reordered. As a result, a zero global reference count does not imply a zero true reference count. However, once the true count *is* zero, there will be no more updates, so if the global reference count of an object drops to zero and *remains* zero for an entire epoch, then review can guarantee that the true count is zero and free the object. To detect this, the first review operation to set an object's global reference count to zero adds the object to the local per-core *review queue*. Review then reexamines it two epochs later (which guarantees one complete epoch has elapsed) to decide whether its true reference count is zero.

Figure 8-2 gives an example of a single object over the course of eight epochs. Epoch 1 demonstrates the power of batching: despite six reference count manipulations spread over three cores, all inc and dec operations are conflict-free (as we would hope, given that they commute), the object's global reference count is never written to and cache line conflicts arise only from epoch maintenance. The remaining epochs demonstrate the complications that

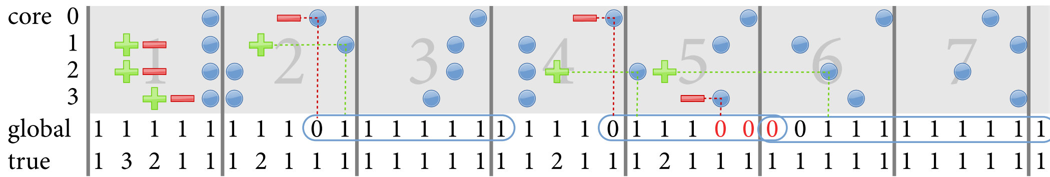

Figure 8-2: Refcache example showing a single object over eight epochs. Plus and minus symbols represent increment and decrement operations, dotted lines show when cores flush these to the object's global count, and blue circles show when each core flushes its local reference cache. The loops around the global count show when the object is in core 0's review queue and the red zeroes indicate dirty zeroes.

arise from batching and the resulting lag between the true reference count and the global reference count of an object.

Because of the flush order, the inc and dec in epoch 2 are applied to the global reference count in the opposite order of how they actually occurred. As a result, core 0 observes the global count temporarily drop to zero when it flushes in epoch 2, even though the true count is non-zero. This is remedied as soon as core 1 flushes its increment, and when core 0 reexamines the object at the beginning of epoch 4, after all cores have again flushed their delta caches, it can see that the global count is non-zero; hence, the zero count it observed was not a true zero and the object should not be freed.

It is necessary but not sufficient for the global reference count to be zero when an object is reexamined; there must also have been no deltas flushed to the object's global count in the interim, even if those changes canceled out or the deltas themselves were zero. For example, core 0's review will observe a zero global reference count at the end of epoch 4, and again when it reexamines the object in epoch 6. However, the true count is not zero, and the global reference count was temporarily non-zero during the epoch. We call this a *dirty* zero and in this situation review will queue the object to be examined again two epochs later, in epoch 8.

8.1.2 Weak references

As described, Refcache is well suited to reference counts that track the number of true references to an object, since there is no danger of the count going back up from zero once the object becomes unreachable. However, operating systems often need untracked references to objects; for example, ScaleFS's caches track objects that may be deleted at any time, and may even need to bring an object's reference count back up from zero. RadixVM's radix tree has similar requirements. To support such uses, we extend Refcache with *weak references*, which provide a tryget operation that will either increment the object's reference count (even if it has reached zero) and return the object, or will indicate that the object has already been deleted.

A weak reference is simply a pointer marked with a "dying" bit, along with a back-reference from the referenced object. When an object's global reference count initially reaches zero,

review sets the weak reference's dying bit. After this, tryget can "revive" the object by atomically clearing the dying bit and fetching the pointer value, and then incrementing the object's reference count as usual. When review decides to free an object, it first atomically clears both the dying bit and the pointer in the weak reference. If this succeeds, it can safely delete the object. If this fails, it reexamines the object again two epochs later. In a race between tryget and review, whether the object is reviewed or deleted is determined by which operation clears the dying bit first.

This protocol can be extended to support multiple weak references per object using a two phase approach in which review first puts all weak references to an object in an intermediate state that can be rolled back if any reference turns out to be revived. Our current implementation does not support this because it is unnecessary for RadixVM or ScaleFS.

8.1.3 Algorithm

Figure 8-3 shows pseudocode for Refcache. Each core maintains a hash table storing its reference delta cache and the review queue that tracks objects whose global reference counts reached zero. A core reviews an object after two epoch boundaries have passed, which guarantees that all cores have flushed their reference caches at least once.

All of the functions in Figure 8-3 execute with preemption disabled, meaning they are atomic with respect to each other on a given core, which protects the consistency of per-core data structures. Fine-grained locks protect the Refcache-related fields of individual objects.

For epoch management, our current implementation uses a barrier scheme that tracks a global epoch counter, per-core epochs, and a count of how many per-core epochs have reached the current global epoch. This scheme suffices for our benchmarks, but schemes with fewer cache-line conflicts are possible, such as the tree-based quiescent state detection used by Linux's hierarchical RCU implementation [46].

8.1.4 Discussion

Refcache trades latency for scalability by batching increment and decrement operations in per-core caches. As a result, except when collisions in the reference delta cache cause evictions, inc and dec are naturally conflict-free with all other operations. Furthermore, because Refcache uses per-core caches rather than per-core counts, it is more space-efficient than other scalable reference counting techniques. While not all uses of reference counting can tolerate Refcache's latency, its scalability and space-efficiency are well suited to the requirements of RadixVM and ScaleFS.

```
inc(obj) ≡
  If local-cache[hash(obj)].obj ≠ obj:
    evict(local-cache[hash(obj)])
    local-cache[hash(obj)] ← ⟨obj, 0⟩
  local-cache[hash(obj)].delta += 1

tryget(weakref) ≡
  Do:
    ⟨obj, dying⟩ ← weakref
  while weakref.cmpxchg(⟨obj, dying⟩, ⟨obj, false⟩) fails
  If obj is not null:
    inc(obj)
  Return obj

evict(obj, delta) ≡
  If delta = 0 and obj.refcnt ≠ 0: return
  With obj locked:
    obj.refcnt ← obj.refcnt + delta
    If obj.refcnt = 0:
      If obj is not on any review queue:
        obj.dirty ← false
        obj.weakref.dying ← true
        Add ⟨obj, epoch⟩ to the local review queue
      else:
        obj.dirty ← true

flush() ≡
  Evict all local-cache entries and clear cache
  Update the current epoch

review() ≡
  flush()
  For each ⟨obj, objepoch⟩ in local review queue:
    If epoch < objepoch + 2: continue
    With obj locked:
      Remove obj from the review queue
      If obj.refcnt ≠ 0:
        obj.weakref.dying ← false
      else if obj.dirty or obj.weakref.cmpxchg(⟨obj, true⟩, ⟨null, false⟩) fails:
        obj.dirty ← false
        obj.weakref.dying ← true
        Add ⟨obj, epoch⟩ to the local review queue
      else:
        Free obj
```

Figure 8-3: Refcache algorithm. Each core calls review periodically. evict may be called by flush or because of a collision in the reference cache. dec is identical to inc except that it decrements the locally cached delta.

8.2 RadixVM: Scalable address space operations

The POSIX virtual memory interface is rife with commutativity, but existing implementations scale poorly. We model a virtual memory system as four key logical operations: mmap adds a region of memory to a process' virtual address space, munmap removes a region of memory from the address space, and memread and memwrite access virtual memory at a particular virtual address or fail if no mapped region contains that address. In reality, the hardware implements memread and memwrite directly and the VM system instead implements a pagefault operation; we discuss this distinction below.

VM operations from *different* processes (and hence address spaces) trivially commute. Most existing implementations use per-process address space representations, so such operations are also naturally conflict-free (and scale well). VM operations from the same process also often commute; in particular, operations on *non-overlapping* regions of the address space commute. Many multithreaded applications exercise exactly this increasingly important scenario: mmap, munmap, and related variants lie at the core of high-performance memory allocators and garbage collectors and partitioning the address space between threads is a key design component of these systems [30, 32, 44]. Applications that frequently map and unmap files also generate such workloads for the OS kernel.

Owing to complex invariants in virtual memory systems, widely used kernels such as Linux and FreeBSD protect each address space with a single lock. This induces both conflicts and, often, complete serialization between commutative VM operations on the same address space, which can dramatically limit application scalability [11, 16]. As a result, applications often resort to workarounds with significant downsides. For example, a Google engineer reported to us that Google's memory allocator is reluctant to return memory to the OS precisely because of scaling problems with munmap and as a result applications tie up gigabytes of memory until they exit. This delays the start of other applications and uses servers inefficiently. Engineers from other companies have reported similar problems to us.

RadixVM is a novel virtual memory system in which commutative operations on non-overlapping address space regions are almost always conflict-free. This ensures that if two threads operate on non-overlapping regions of virtual memory, the VM system does not limit the application's scalability. Furthermore, if multiple threads operate on the same shared region, RadixVM constrains conflicts to the cores executing those threads.

Achieving this within the constraints of virtual memory hardware, without violating POSIX's strict requirements on the ordering and global visibility of VM operations, and without unacceptable memory overhead is challenging. The following sections describe RadixVM's approach. Section 8.2.1 describes the general architecture of POSIX VM systems. Sections 8.2.2 and 8.2.3 describe the data structures that form the core of RadixVM. Finally, section 8.2.4 describes how RadixVM uses these structures to implement standard VM operations.

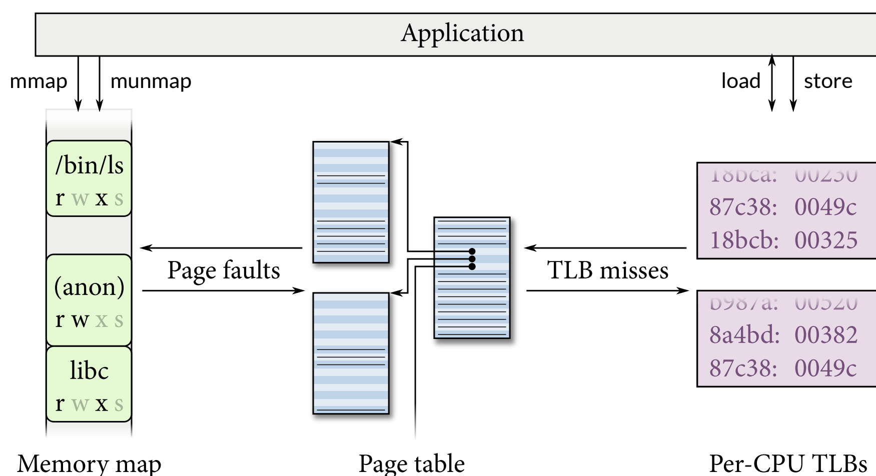

Figure 8-4: Key structures in a POSIX VM system, showing an address space with three mapped regions. Crossed-out page table entries are marked "not present." Some pages have not been faulted yet, and thus are not present, despite being in a mapped region.

8.2.1 POSIX VM architecture

Between the POSIX VM interface on one side and the hardware virtual memory interface on the other, RadixVM's general architecture necessarily resembles typical Unix VM systems. Figure 8-4 shows the principle structures any Unix VM system must manage. On the left is the kernel's internal representation of the address space. Logically, this consists of a set of mapped regions of the virtual address space, where each region may map either a file or "anonymous" heap memory, has various access permissions, and may be either shared or copied across fork. mmap and munmap manipulate this view of the address space.

Translating virtual addresses to physical addresses is performed in hardware by the memory management unit (MMU) and the MMU's representation of virtual memory generally differs significantly from the kernel's internal representation of the address space. Most hardware architectures specify some form of page table data structure, like the x86 page table depicted in Figure 8-4, for the kernel to inform the MMU of the mapping from virtual addresses to physical addresses. These structures map addresses at a page granularity (typically 4 KB, though most architectures also support larger pages) and represent access permissions (which are checked by the MMU), but have little flexibility.

While we model the VM system in terms of mmap, munmap, memread, and memwrite, only the first two are directly implemented by the VM system. The latter two are implemented by the MMU using the kernel-provided page table. However, the VM system has a key hook into the MMU: when the virtual address of a read or a write is marked "not present" in the page table or fails a permission check, the MMU invokes the VM system's pagefault operation. Modern Unix kernels exploit this mechanism to implement *demand paging*, populating page table

entries by allocating pages or reading them from disk only once a page is first accessed, rather than when it is mapped. Hence, mmap and munmap simply clear the page table entries of the region being mapped or unmapped; mmap depends on pagefault operations to later fill these entries with mappings. While demand paging significantly affects the implementation of the VM system, it is nevertheless an implementation issue and does not affect the commutativity of memread or memwrite.

The final key structure the VM system must manage is the hardware translation lookaside buffer (TLB). The TLB is a per-core associative cache of virtual-to-physical mappings. In architectures with a hardware-filled TLB (x86, ARM, PowerPC, and many others), the MMU transparently fills this cache from the page table. Invalidating the TLB, however, is the responsibility of the VM system. Hence, munmap, in addition to removing regions from the kernel's internal address space representation and clearing entries in the page table, must also invalidate the corresponding entries from each core's TLB.

8.2.2 Radix tree

We first tackle RadixVM's internal address space representation. Widely-used operating system kernels represent an address space as a balanced tree of mapped memory regions. For example, Linux uses a red-black tree [64], FreeBSD uses a splay tree [54], and Solaris and Windows use AVL trees [51, 66]. This ensures $O(\log n)$ lookups and modifications for address spaces that can easily contain thousands of mapped regions, but these data structures are poorly suited for concurrent access: not only do they induce cache line conflicts between operations on non-overlapping regions, they force all of these kernels to use coarse-grained locking.

Lock-free data structures seem like a compelling solution to this problem. However, while lock-free (or partially lock-free) data structures such as the Bonsai tree used by the Bonsai VM [16], relativistic red-black trees [38], and lock-free skip lists [36] eliminate the coarse-grained serialization that plagues popular VM systems, their operations are far from conflict-free. For example, insert and lookup operations for a lock-free concurrent skip list can conflict on interior nodes in the skip list—even when the lookup and insert involve different keys and hence commute—because insert must modify interior nodes to maintain $O(\log n)$ lookup time. The effect of these conflicts on performance can be dramatic, as shown by the simple experiment in Figure 8-5. Furthermore, while these data structures maintain their own integrity under concurrent operations, a VM system must maintain higher-level invariants across the address space representation, page tables, and TLBs, and it is difficult to extend the precision design of a single lock-free data structure to cover these broader invariants.

This raises the question: what is a good conflict-free data structure for virtual memory metadata?

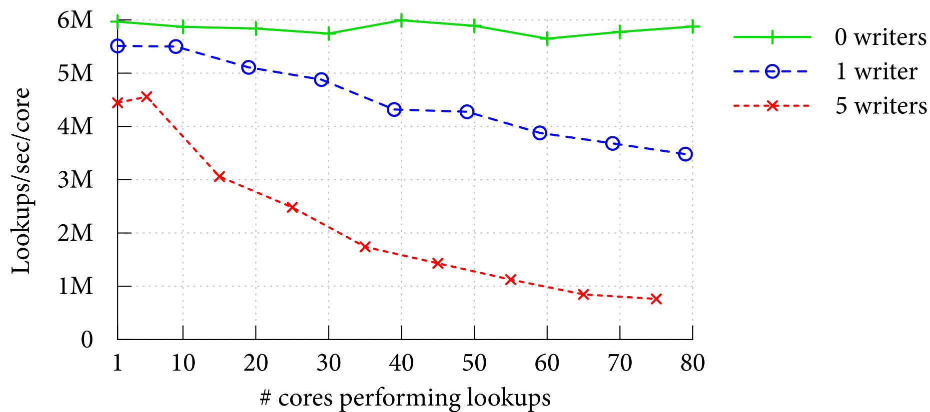

Figure 8-5: Throughput of skip list lookups with concurrent inserts and deletes. The skip list represents ~1,000 virtual memory regions. Lookups commute with modifications, but even a small number of writers significantly limits lookup scalability.

Straw-man solution. A (completely impractical) way to achieve conflict-free commutative address space operations is to represent a process's address space as a large linear array indexed by virtual page number that stores each virtual page's metadata individually. In this linear representation, mmap, munmap, and pagefault can lock and manipulate precisely the range of pages being mapped, unmapped, or faulted. Operations on non-overlapping regions are clearly conflict-free and precise range locking makes maintaining invariants across structures relatively straightforward.

Compressed radix trees. RadixVM follows the same general scheme as this straw-man design, but reins in its memory consumption using a multilevel, compressed radix tree.

RadixVM's internal address space representation resembles a hardware page table structurally, storing mapping metadata in a fixed-depth radix tree, where each level of the tree is indexed by nine or fewer bits of the virtual page number, as shown in Figure 8-6. Like the linear array, the radix tree supports only point queries (not range queries) and iteration, but unlike the linear array, RadixVM can compress repeated entries and lazily allocate the nodes of the radix tree. The radix tree folds any node that would consist entirely of identical values into a single value stored in the parent node. This continues up to the root node of the tree, allowing the radix tree to represent vast swaths of unused virtual address space with a handful of empty slots and to set large ranges to identical values very quickly. This optimization reins in the radix tree's memory use, makes it efficient to set large ranges to the same value quickly, and enables fast range scans. It does come at a small cost: operations that expand a subtree will conflict with other operations in that same subtree, even if their regions do not ultimately overlap. However, after initial address space construction, the radix tree is largely "fleshed out" and these initialization conflicts become rare.

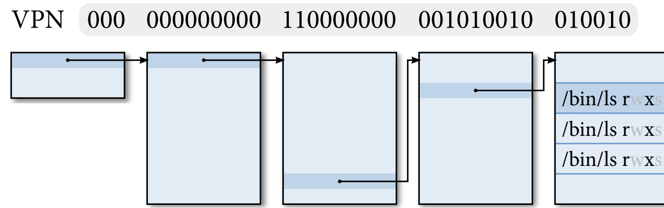

Figure 8-6: A radix tree containing a three page file mapping. This highlights the path for looking up the 36-bit virtual page number shown in binary at the top of the figure. The last level of the tree contains separate mapping metadata for each page.

Mapping metadata. To record each mapping, RadixVM logically stores a separate instance of the mapping metadata in the radix tree for each page in the mapped range. This differs from a typical design that allocates a single metadata object to represent the entire range of a mapping (e.g., *virtual memory areas* in Linux). This is practical because RadixVM's mapping metadata object is small and foldable (the mapping metadata objects of all pages in a mapped range are initially byte-wise identical), so large mappings can be created and stored efficiently in just a few slots in the radix tree.

Also unlike typical virtual memory system designs, RadixVM stores pointers to physical memory pages in the mapping metadata for pages that have been allocated. This is natural to do in RadixVM because, modulo folding, there is a separate mapping metadata object for each page. It's also important to have this canonical representation of the physical memory backing a virtual address space because of the way RadixVM handles TLB shootdown (see section 8.2.3). This does increase the space required by the radix tree, but, asymptotically, it's no worse than the hardware page tables, and it means that the hardware page tables themselves are cacheable memory that can be discarded by the OS to free memory.

Radix node garbage collection. To keep the memory footprint of the radix tree in check, the OS must be able to free nodes that no longer contain any valid mapping metadata. To accomplish this without introducing undue conflicts, we leverage Refcache to scalably track the number of used slots in each node. When this count drops to zero, the radix tree can remove the node from the tree and delete it. Since RadixVM may begin using a node again before Refcache reconciles the used slot count, nodes link to their children using weak references, which allows the radix tree to revive nodes that go from empty to used before Refcache deletes them, and to safely detect when an empty child node has been deleted.

Collapsing the radix tree does potentially introduce conflicts; however, unlike with eager garbage collection schemes, rapidly changing mappings cannot cause the radix tree to rapidly delete and recreate nodes. Since a node must go unused for at least two Refcache epochs before it is deleted, any cost of deleting or recreating it (and any additional contention that results) is amortized.

In contrast with more traditional balanced trees, using a radix tree to manage address space metadata allows RadixVM to achieve near-perfect conflict-freedom (and hence scalability) for metadata operations on non-overlapping regions of an address space. This comes at the cost of a potentially larger memory overhead; however, address space layouts tend to exhibit good locality and folding efficiently compresses large ranges, making radix trees a good fit for a VM system, as we'll confirm in section 9.6.

8.2.3 TLB management

The other major impediment to scaling mmap and munmap operations is the need to explicitly invalidate cached virtual-to-physical mappings in per-core TLBs when a page is unmapped (or remapped). Because TLB shootdowns must be performed on every CPU that may have cached a page mapping that's being modified, and because hardware does not provide information about which CPUs may have cached a particular mapping, typical Unix VM systems conservatively broadcast TLB shootdown interrupts to all CPUs using the modified address space [8], inducing cache line conflicts and limiting scalability.

RadixVM addresses this problem by precisely tracking the set of CPUs that have accessed each page mapping. In architectures with software-filled TLBs (such as the MIPS or UltraSPARC) this is easy, since the MMU informs the kernel of each miss in the TLB. The kernel can use these TLB miss faults to track exactly which CPUs have a given mapping cached and, when a later mmap or munmap changes this mapping, it can deliver shootdown requests only to cores that have accessed this mapping. On architectures with hardware-filled TLBs such as the x86, RadixVM achieves the same effect using per-core page tables. With TLB tracking, if an application thread allocates, accesses, and frees memory on one core, with no other threads accessing the same memory region, then RadixVM will perform no TLB shootdowns.

The downside to this approach is the extra memory required for per-core page tables. We show in section 9.6 that this overhead is small in practice compared to the total memory footprint of an application, but for applications with poor partitioning, it may be necessary for the application to provide hints about widely shared regions so the kernel can share page tables (similar to Corey address ranges [10]) or for the kernel to detect such regions. The kernel can also reduce overhead by simply discarding page table pages when memory is low.

8.2.4 VM operations

With the components described above, RadixVM's implementation of the core VM operations is surprisingly straightforward. One of the most difficult aspects of the POSIX VM interface in a multicore setting is its strict ordering and global visibility requirements [16]. For example, before munmap in one thread can return, the region must appear unmapped to every thread on every core despite caching at both software and hardware levels. Similarly, after mmap

returns, any thread on any core must be able to access the mapped region. Furthermore, though not required by POSIX, it is generally assumed that these operations are atomic.

RadixVM enforces these semantics by always locking, from left to right, the bottom-most radix tree entries for the region of an operation. This simple mechanism ensures that mmap, munmap, and pagefault operations are linearizable [37] without causing conflicts between operations on non-overlapping regions (assuming the tree is already expanded).

An mmap invocation, like all RadixVM operations, first locks the range being mapped. If there are existing mappings within the range, mmap unmaps them, as described later for munmap. mmap then fills each slot in the region with the new mapping metadata (protection level and flags arguments to mmap, as well as what backs this virtual memory range, such as a file or anonymous memory). If parts of the mapping span entire nodes of the radix tree, RadixVM will fold them into individual higher-level slots. Finally, RadixVM unlocks the range. Like in other VM systems, mmap doesn't allocate any physical pages, but leaves that to pagefault, so that pages are allocated only when they are needed.

A pagefault invocation traverses the radix tree to find the mapping metadata for the faulting address, and acquires a lock on it. It then allocates a physical page if one has not been allocated yet (for anonymous memory), or fetches it from the buffer cache (for file mappings) and stores it in the mapping metadata. Finally, pagefault fills in the page table entry in the local core's page table, and adds the local core number to the TLB shootdown list in the mapping metadata for that address. pagefault then releases the lock and returns.

To implement munmap, RadixVM must clear mapping metadata from the radix tree, clear page tables, invalidate TLBs, and free physical pages. munmap begins by locking the range being unmapped, after which it can scan the region's metadata to gather references to the physical pages backing the region, collect the set of cores that have faulted pages in the region into their per-core page tables, and clear each page's metadata. It can then send inter-processor interrupts to the set of cores it collected in the first step. These interrupts cause the remote cores (and the core running munmap) to clear the appropriate range in their per-core page table and invalidate the corresponding local TLB entries. Once all cores have completed this shootdown process, munmap can safely release its lock on the range and decrement the reference counts on the physical pages that were unmapped.

8.2.5 Discussion

By combining Refcache for scalable reference counting, radix trees for maintaining address space metadata, and per-core page tables for precise TLB tracking and shootdown, RadixVM achieves conflict-freedom for the vast majority of commutative VM operations. The clear commutativity properties of the POSIX VM interface, combined with the right data structures makes this possible with a straightforward concurrency plan based on precise range locking.

RadixVM balances practical memory consumption with strict adherence to scalable commutativity rule, allowing conflicts between some commutative operations where doing

so enables far more efficient memory use. However, such conflicts are rare: operations that allocate a radix node will conflict with later operations on that node's range of the key space, but typical address space usage involves few node-allocating operations.

As shown in Figure 8-1, Commuter confirmed that most commutative operations in RadixVM are conflict-free. In chapter 9, we further confirm that RadixVM's design translates into scalable performance for application workloads.

8.3 ScaleFS: Conflict-free file system operations

ScaleFS encompasses sv6's unified buffer cache and VFS layers, providing operations such as read, write, open, and unlink. We focused on the VFS and buffer cache layers because these are the common denominators of all file systems in a Unix kernel. In contrast with RadixVM, ScaleFS makes extensive use of well-known techniques for scalable implementations, such as per-core resource allocation, double-checked locking, lock-free readers using RCU [49], and seqlocks [43: §6]. ScaleFS also employs Refcache for tracking both internal resources and inode link counts. ScaleFS is also structured much like contemporary Unix VFS subsystems, with inode and directory caches represented as concurrent hash tables and per-file page caches. What sets ScaleFS apart is that the details of its implementation were guided by the scalable commutativity rule and, in particular, by Commuter. This led to several common design patterns, which we illustrate with example test cases from Commuter:

Layer scalability. ScaleFS uses data structures that themselves naturally satisfy the commutativity rule, such as linear arrays, radix trees, and hash tables. In contrast with structures like balanced trees, these data structures typically share no cache lines when different elements are accessed or modified. For example, ScaleFS stores the cached data pages for a given inode using a radix tree, so that concurrent reads or writes to different file pages scale, even in the presence of operations extending or truncating the file. Commuter led us to an additional benefit of this representation: many operations also use this radix tree to determine if some offset is within the file's bounds without reading the size and conflicting with operations that change the file's size. For example, pread first probes this radix tree for the requested read offset: if the offset is beyond the last page of the file, it can return 0 immediately without reading (and potentially conflicting on) the file size.

Defer work. Many ScaleFS resources are shared, such as file descriptions and inode objects, and must be freed when no longer referenced. Typically, kernels release resources immediately, but this requires eagerly tracking references to resources, causing commutative operations that access the same resource to conflict. Where releasing a resource is not time-sensitive, ScaleFS uses Refcache to batch reference count reconciliation and zero detection. This way,

resources are eventually released, but within each Refcache epoch commutative operations can be conflict-free.

Some resources are artificially scarce, such as inode numbers in a typical Unix file system. When a typical Unix file system runs out of free inodes, it must reuse an inode from a recently deleted file. This requires finding and garbage collecting unused inodes, which induces conflicts. However, the POSIX interface does not require that inode numbers be reused, only that the same inode number is not used for two files at once. Thus, ScaleFS never reuses inode numbers. Instead, inode numbers are generated by a monotonically increasing per-core counter, concatenated with the core number that allocated the inode. This allows ScaleFS to defer inode garbage collection for longer periods of time, and enables conflict-free and scalable per-core inode allocation.

Precede pessimism with optimism. Many operations in ScaleFS have an optimistic check stage followed by a pessimistic update stage, a generalized sort of double-checked locking. The optimistic stage checks conditions for the operation and returns immediately if no updates are necessary (this is often the case for error returns, but can also happen for success returns). This stage does no writes or locking, but because no updates are necessary, it is often easy to make atomic. If updates are necessary, the operation acquires locks or uses lock-free protocols, re-verifies its conditions to ensure atomicity of the update stage, and performs updates. For example, lseek computes the new offset using a lock-free read-only protocol and returns early if the new offset is invalid or equal to the current offset. Otherwise, lseek locks the file offset and re-computes the new offset to ensure consistency. In fact, lseek has surprisingly complex interactions with state and other operations, and arriving at a protocol that was both correct and conflict-free in all commutative cases would have been difficult without COMMUTER.

rename is similar. If two file names a and b point to the same inode, rename(a, b) should remove the directory entry for a, but it does not need to modify the directory entry for b, since it already points at the right inode. By checking the directory entry for b before updating it, rename(a, b) avoids conflicts with other operations that look up b. As we saw in section 6.1.2, rename has many surprising and subtle commutative cases and, much like lseek, COMMUTER was instrumental in helping us find an implementation that was conflict-free in these cases.

Don't read unless necessary. A common internal interface in a file system implementation is a namei function that checks whether a path name exists, and if so, returns the inode for that path. However, reading the inode is unnecessary if the caller wants to know only whether a path name existed, such as an access(F_OK) system call. In particular, the namei interface makes it impossible for concurrent access(b, F_OK) and rename(a, b) operations to scale when a and b point to different inodes, even though they commute. ScaleFS has a separate internal interface to check for existence of a file name, without looking up the inode, which allows access and rename to scale in such situations.

8.4 Difficult-to-scale cases

As Figure 8-1 illustrates, there are a few (123 out of 26,238) commutative test cases for which RadixVM and ScaleFS are not conflict-free. The majority of these tests involve idempotent updates to internal state, such as two lseek operations that both seek a file descriptor to the same offset, or two anonymous mmap operations with the same fixed base address and permissions. While it is possible implement these scalably, every implementation we considered significantly impacted the performance of more common operations, so we explicitly chose to favor common-case performance over total scalability. Even though we decided to forego scalability in these cases, the commutativity rule and COMMUTER forced us to consciously make this trade-off.

Other difficult-to-scale cases are more varied. Several involve reference counting of pipe file descriptors. Closing the last file descriptor for one end of a pipe must immediately affect the other end; however, since there's generally no way to know a priori if a close will close the pipe, a shared reference count is used in some situations. Other cases involve operations that return the same result in either order, but for different reasons, such as two reads from a file filled with identical bytes. By the rule, each of these cases has some conflict-free implementation, but making these particular cases conflict-free would have required sacrificing the conflict-freedom of many other operations.

Nine

Performance evaluation

Given that nearly all commutative ScaleFS and RadixVM operations are conflict-free, in principle, applications built on these operations should scale perfectly. This chapter confirms this, completing a pyramid whose foundations were set in chapter 3 when we demonstrated that conflict-free memory accesses scale in most circumstances on real hardware. This chapter extends these results, showing that complex operating system calls built on conflict-free memory accesses scale and that, in turn, applications built on these operations scale. We focus on the following questions:

- Do conflict-free implementations of commutative operations and applications built using them scale on real hardware?
- Do non-commutative operations limit performance on real hardware?

Since real systems cannot focus on scalability to the exclusion of other performance characteristics, we also consider the balance of performance requirements by exploring the following question:

- Can implementations optimized for linear scalability of commutative operations also achieve competitive sequential performance, reasonable (albeit sub-linear) scalability of non-commutative operations, and acceptable memory use?

To answer these questions, we use sv6. In addition to the operations analyzed in chapter 8, we scalably implemented other commutative operations (e.g., posix_spawn) and many of the modified POSIX APIs from chapter 5. All told, sv6 totals 51,732 lines of code, including user space and library code. Using sv6, we evaluate two microbenchmarks and one application benchmark focused on file system operations, and three microbenchmarks and one application benchmark focused on virtual memory operations.

9.1 Experimental setup

We ran experiments on an 80-core machine with eight 2.4 GHz 10-core Intel E7-8870 chips and 256 GB of RAM (detailed earlier in Figure 3-2). When varying the number of cores,

benchmarks enable whole sockets at a time, so each 30 MB socket-level L3 cache is shared by exactly 10 enabled cores. We also report single-core numbers for comparison, though these are expected to be higher because without competition from other cores in the socket, the one core can use the entire 30 MB cache.

We run all benchmarks with the hardware prefetcher disabled because we found that it often prefetched contended cache lines to cores that did not ultimately access those cache lines, causing significant variability in our benchmark results and hampering our efforts to precisely control sharing. We believe that, as large multicores and highly parallel applications become more prevalent, prefetcher heuristics will likewise evolve to not induce this false sharing.

As a performance baseline, we run the same benchmarks on Linux 3.5.7 from Ubuntu Quantal. All benchmarks compile and run on Linux and sv6 without modifications. Direct comparison is difficult because Linux implements many features sv6 does not, but this comparison confirms sv6's sequential performance is sensible.

We run each benchmark three times and report the mean. Variance from the mean is always under 5% and typically under 1%.

9.2 File system microbenchmarks

Each file system benchmark has two variants, one that uses standard, non-commutative POSIX APIs and another that accomplishes the same task using the modified, more broadly commutative APIs from chapter 5. By benchmarking the standard interfaces against their commutative counterparts, we can isolate the cost of non-commutativity and also examine the scalability of conflict-free implementations of commutative operations.

statbench. In general, it's difficult to argue that an implementation of a non-commutative interface achieves the best possible scalability for that interface and that no implementation could scale better. However, in limited cases, we can do exactly this. We start with statbench, which measures the scalability of fstat with respect to link. This benchmark creates a single file that $n/2$ cores repeatedly fstat. The other $n/2$ cores repeatedly link this file to a new, unique file name, and then unlink the new file name. As discussed in chapter 5, fstat does not commute with link or unlink on the same file because fstat returns the link count. In practice, applications rarely invoke fstat to get the link count, so sv6 introduces fstatx, which allows applications to request specific fields (a similar system call has been proposed for Linux [39]).

We run statbench in two modes: one mode uses fstat, which does not commute with the link and unlink operations performed by the other threads, and the other mode uses fstatx to request all fields except the link count, an operation that *does* commute with link and unlink. We use a Refcache scalable counter for the link count so that the links and unlinks are conflict-free, and place it on its own cache line to avoid false sharing. Figure 9-1(a) shows the

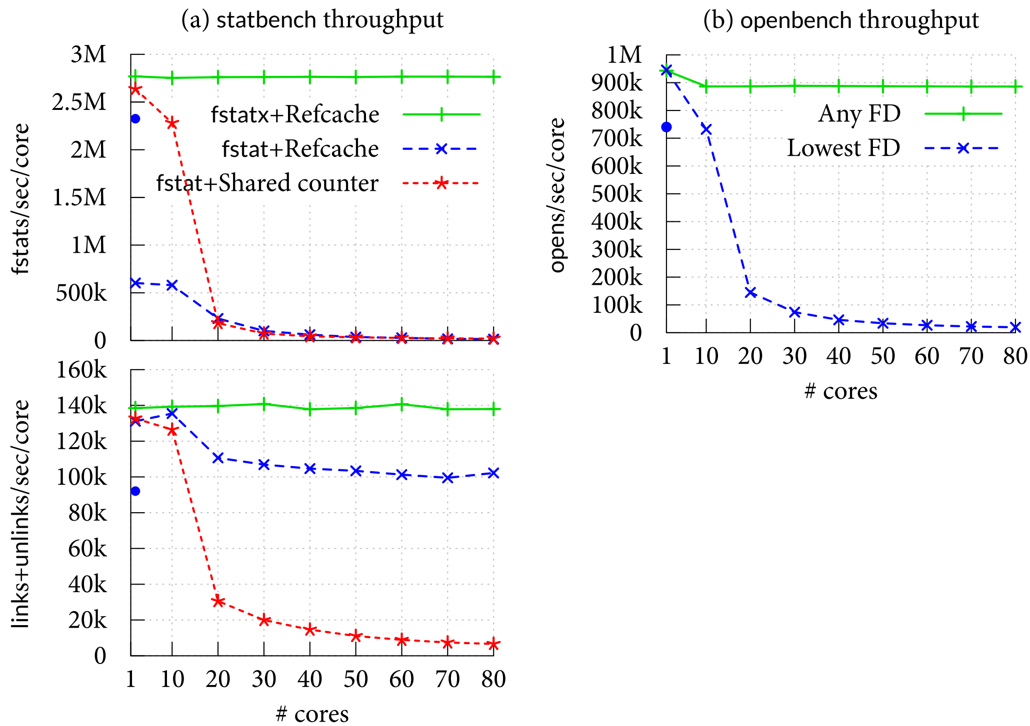

Figure 9-1: File system microbenchmark throughput in operations per second per core with varying core counts on sv6. The blue dots indicate single core Linux performance for comparison.

results. With the commutative fstatx, statbench scales perfectly for both fstatx and link/unlink and experiences zero L2 cache misses in fstatx. On the other hand, the traditional fstat scales poorly and the conflicts induced by fstat impact the scalability of the threads performing link and unlink.

To better isolate the difference between fstat and fstatx, we run statbench in a third mode that uses fstat, but represents the link count using a simple shared counter instead of Refcache. In this mode, fstat performs better at low core counts, but fstat, link, and unlink all suffer at higher core counts. With a shared link count, each fstat call experiences exactly one L2 cache miss (for the cache line containing the link count), which means this is the most scalable that fstat can possibly be in the presence of concurrent links and unlinks. Yet, despite sharing only a single cache line, the seemingly innocuous conflict arising from the non-commutative interface limits the implementation's scalability. One small tweak to make the operation commute by omitting st_nlink eliminates the barrier to scaling, demonstrating that even an optimal implementation of a non-commutative operation can have severely limited scalability, underscoring the results of chapter 3.

In the case of fstat, optimizing for scalability sacrifices some sequential performance. Tracking the link count with Refcache (or some scalable counter) is necessary to make link

and unlink scale linearly, but requires fstat to reconcile the distributed link count to return st_nlink. The exact overhead depends on the core count, which determines the number of Refcache caches, but with 80 Refcache caches, fstat is 3.9× more expensive than on Linux. In contrast, fstatx can avoid this overhead unless the caller requests link counts; like fstat with a shared count, it performs similarly to Linux's fstat on a single core.

openbench. Figure 9-1(b) shows the results of openbench, which stresses the file descriptor allocation performed by open. In openbench, *n* threads concurrently open and close per-thread files. These calls do not commute because each open must allocate the lowest unused file descriptor in the process. For many applications, it suffices to return any unused file descriptor (in which case the open calls commute), so sv6 adds an O_ANYFD flag to open, which it implements using per-core partitions of the FD space. Much like statbench, the standard, non-commutative open interface limits openbench's scalability, while openbench with O_ANYFD scales linearly.

Furthermore, there is no performance penalty to ScaleFS's open, with or without O_ANYFD: at one core, both cases perform identically and outperform Linux's open by 27%. Some of the performance difference is because sv6 doesn't implement things like permissions checking, but much of Linux's overhead comes from locking that ScaleFS avoids.

9.3 File system application performance

We perform a similar experiment using a simple mail server to produce a file system workload more representative of a real application. The mail server uses a sequence of separate, communicating processes, each with a specific task, roughly like qmail [6]. mail-enqueue takes a mailbox name and a single message on standard in, writes the message and the envelope to two files in a mail queue directory, and notifies the queue manager by writing the envelope file name to a Unix domain datagram socket. mail-qman is a long-lived multithreaded process where each thread reads from the notification socket, reads the envelope information, opens the queued message, spawns and waits for the delivery process, and then deletes the queued message. Finally, mail-deliver takes a mailbox name and a single message on standard in and delivers the message to the appropriate Maildir. The benchmark models a mail client with *n* threads that continuously deliver email by spawning and feeding mail-enqueue.

As in the microbenchmarks, we run the mail server in two configurations: in one we use lowest FD, an order-preserving socket for queue notifications, and fork/exec to spawn helper processes; in the other we use O_ANYFD, an unordered notification socket, and posix_spawn, all as described in chapter 5. For queue notifications, we use a Unix domain datagram socket. sv6 implements this with a single shared queue in ordered mode In unordered mode, sv6 uses load-balanced per-core message queues. Load balancing only triggers when a core attempts to read from an empty queue, so operations on unordered sockets are conflict-free as long as

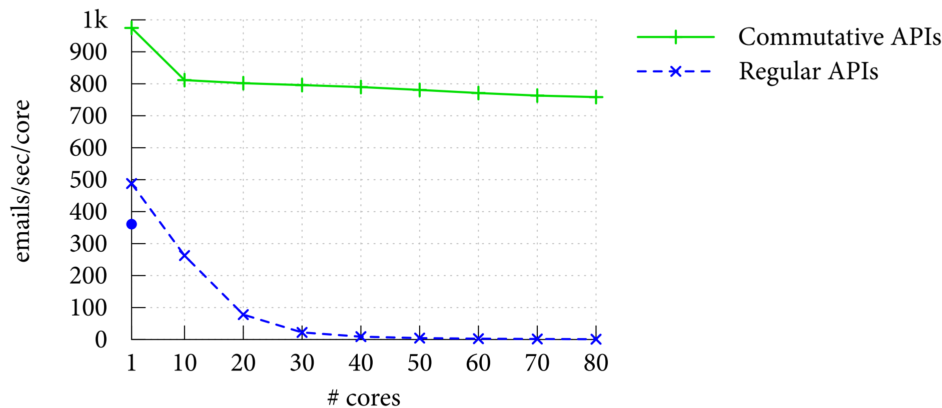

Figure 9-2: Mail server benchmark throughput. The blue dot indicates single core Linux performance for comparison.

consumers don't outpace producers. Finally, because fork commutes with essentially no other operations in the same process, sv6 implements posix_spawn by constructing the new process image directly and building the new file table. This implementation is conflict-free with most other operations, including operations on O_CLOEXEC files (except those specifically duped into the new process).

Figure 9-2 shows the resulting scalability of these two configurations. Even though the mail server performs a much broader mix of operations than the microbenchmarks and doesn't focus solely on non-commutative operations, the results are quite similar. Non-commutative operations cause the benchmark's throughput to collapse at a small number of cores, while the configuration that uses commutative APIs achieves 7.5× scalability from 1 socket (10 cores) to 8 sockets.

9.4 Virtual memory microbenchmarks

To understand the scalability and performance of sv6's RadixVM virtual memory system, we use three microbenchmarks, each of which exercises a specific pattern of address space usage.

As in the earlier benchmarks, we compare RadixVM against Linux 3.5.7. We additionally compare against the Bonsai VM system [16] (based on Linux 2.6.37). As discussed in section 8.2.2, Linux's mmap, munmap, and pagefault all acquire a per-address space lock. This serializes mmap and munmap operations. Linux's pagefault acquires the lock in shared mode, allowing pagefaults to proceed in parallel, but acquiring the address space lock in shared mode still induces conflicts on the cache line storing the lock. Bonsai modifies the Linux virtual memory system so that pagefaults do not acquire this lock. This makes most commutative pagefaults conflict-free in Bonsai, though Bonsai does not modify the locking behavior of mmap and munmap, so all mmap and munmap operations on the same address space conflict.

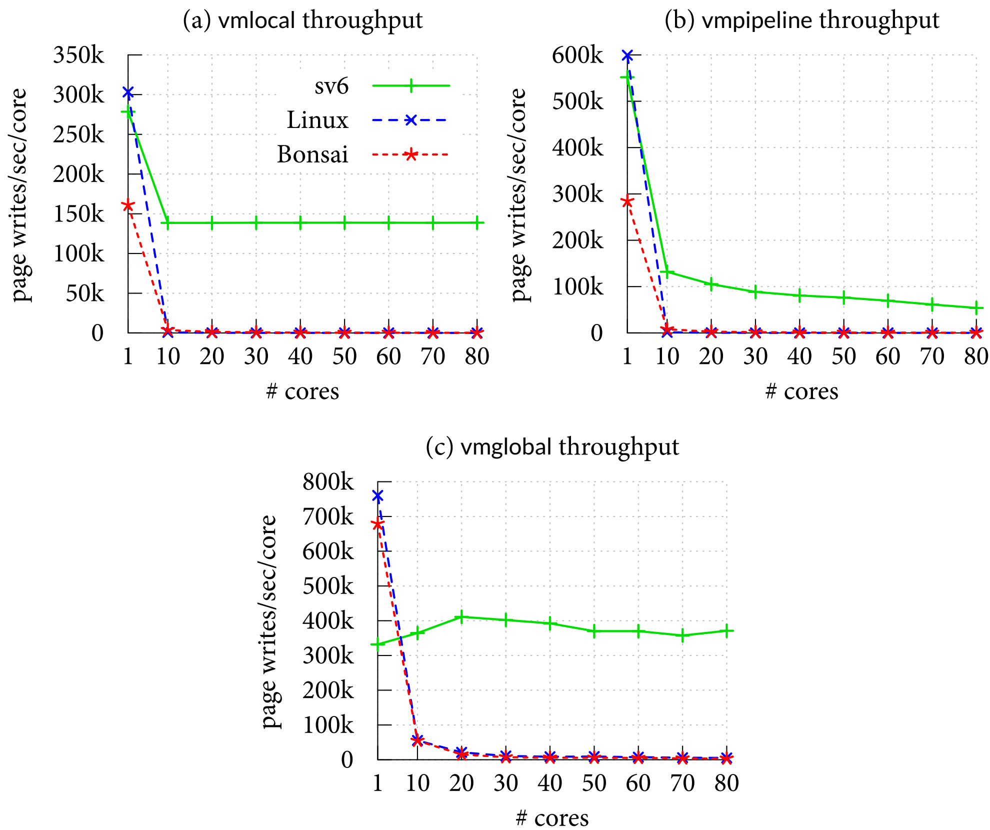

Figure 9-3: Virtual memory microbenchmark throughput in page writes per second per core with varying core counts.

Figure 9-3 shows the throughput of our three microbenchmarks on sv6, Bonsai, and Linux. For consistency, we measure the number of pages written per second per core in all three benchmarks. Because sv6 uses per-core page tables, each of these writes translates into a page fault, even if the page has already been faulted by another core. Linux and Bonsai incur fewer page faults than sv6 for the pipeline and global microbenchmarks because all cores use the same page table.

vmlocal. The vmlocal microbenchmark exercises completely disjoint address space usage. In vmlocal, each thread mmaps a single page in the shared address space, writes to the page (invoking pagefault), and then munmaps the page. Because each thread maps a page at a different virtual address, all operations performed by vmlocal commute.

Many concurrent memory allocators use per-thread memory pools that specifically optimize for thread-local allocations and exhibit this pattern of address space manipulation [30, 32]. However, such memory allocators typically map memory in large batches and conservatively return memory to the operating system to avoid burdening on the virtual

memory system. Our microbenchmark does the opposite, using 4 KB regions to maximally stress the VM system.

The vmlocal microbenchmark scales linearly on sv6. While this benchmark does incur cache misses, none are conflict misses and none require cross-core communication. Regardless of the number of cores, we observe about 75 L2 cache misses per iteration and about 50 L3 cache misses, almost all a result of page zeroing. None of these require cross-socket communication as all are satisfied from the core's local DRAM, which is consistent with COMMUTER's determination that these operations are conflict-free. Likewise, because sv6 can track remote TLBs precisely, the vmlocal microbenchmark sends no TLB shootdowns. Because these operations are conflict-free—there is no lock contention, a small and fixed number of cache misses, no remote DRAM accesses, and no TLB shootdowns—the time required to mmap, pagefault, and munmap is constant regardless of the number of cores.

Linux and Bonsai, on the other hand, slow down as we add more cores. This is not unexpected: Linux acquires the address space lock three times per iteration and Bonsai twice per iteration, effectively serializing the benchmark. sv6 also demonstrates good sequential performance: at one core, sv6's performance is within 8% of Linux, and it is likely this could be improved.

vmpipeline. The vmpipeline benchmark captures the pattern of streaming or pipeline communication, such as a *map* task communicating with a *reduce* task in MapReduce [25]. In vmpipeline, each thread mmaps a page of memory, writes to the page, and passes the page to the next thread in sequence, which also writes to the page and then munmaps it. The operations performed by neighboring cores do not commute and their implementation is not conflict-free. However, the operations of non-neighboring cores do commute, so unlike the non-commutative cases in the file system benchmarks, increasing the number of cores in vmlocal increases the number of conflicted cache lines without increasing the number of cores conflicting on each cache line. As a result, vmpipeline still scales well on sv6, but not linearly. We observe similar cache miss rates as vmlocal, but vmpipeline induces cross-socket memory references for pipeline synchronization, returning freed pages to their home nodes when they are passed between sockets, and cross-socket shootdown IPIs. For Linux and Bonsai, vmpipeline is almost identical to vmlocal except that it writes twice to each page; hence we see almost identical performance, scaled up by a factor of two. Again, sv6's single core performance is within 8% of Linux.

vmglobal. Finally, vmglobal simulates a widely shared region such as a memory-mapped library or a shared data structure like a hash table. In vmglobal, each thread mmaps a 64 KB part of a large region of memory, then all threads access all of the pages in the large region in a random order. These operations commute within each phase—all mmaps are to non-overlapping regions and pagefaults always commute—but not between phases.

Vmglobal scales well on sv6, despite being conceptually poorly suited to sv6's per-core page tables and targeted TLB shootdown. In this benchmark, sv6 performance is limited by the cost of TLB shootdowns: at 80 cores, delivering shootdown IPIs to the other 79 cores and waiting for acknowledgments takes nearly a millisecond. However, at 80 cores, the shared region is 20 MB, so this cost is amortized over a large number of page faults. Linux and Bonsai perform better on this benchmark than on vmlocal and vmpipeline because it has a higher ratio of page faults to mmap and munmap calls, but they still fail to scale.

9.5 Virtual memory application benchmark

To evaluate the impact of RadixVM on application performance, we use Metis, a high-performance single-server multithreaded MapReduce library, to compute a word position index from a 4 GB in-memory text file [25, 45]. Metis exhibits all of the sharing patterns exercised by the microbenchmarks: it uses core-local memory, it uses a globally shared B+-tree to store key-value pairs, and it also has pairwise sharing of intermediate results between *map* tasks and *reduce* tasks. Metis also allows a direct comparison with the Bonsai virtual memory system [16], which used Metis as its main benchmark.

By default Metis uses the Streamflow memory allocator [57], which is designed to minimize pressure on the VM system, but nonetheless suffers from contention in the VM system when running on Linux [16]. Previous systems that used this library avoided contention for in-kernel locks by using super pages and improving the granularity of the super page allocation lock in Linux [11], or by having the memory allocator pre-allocate all memory upfront [45]. While these workarounds do allow Metis to scale on Linux, we wanted to focus on the root scalability problem in the VM system rather than the efficacy of workarounds and to eliminate compounding factors from differing library implementations, so we use a custom allocator on both Linux and sv6 designed specifically for Metis. In contrast with general-purpose memory allocators, this allocator is simple and designed to have no internal contention: memory is mapped in fixed-sized blocks, free lists are exclusively per-core, and the allocator never returns memory to the OS.

Two factors determine the scalability of Metis: conflicts between concurrent mmaps during the *map* phase and conflicts between concurrent pagefaults during the *reduce* phase. If the memory allocator uses a large allocation unit, Metis can avoid the first source of contention by minimizing the number of mmap invocations. Therefore, we measure Metis using two different allocation units: 8 MB to stress pagefault and 64 KB to stress mmap. At 80 cores, Metis invokes mmap 4,145 times in the 8 MB configuration, and 232,464 in the 64 KB configuration. In both cases, it invokes pagefault approximately 10 million times, where 65% of these page faults cause it to allocate new physical pages and the rest bring pages already faulted on another core in to the per-core page tables.

Figure 9-4 shows how Metis scales for the three VM systems.

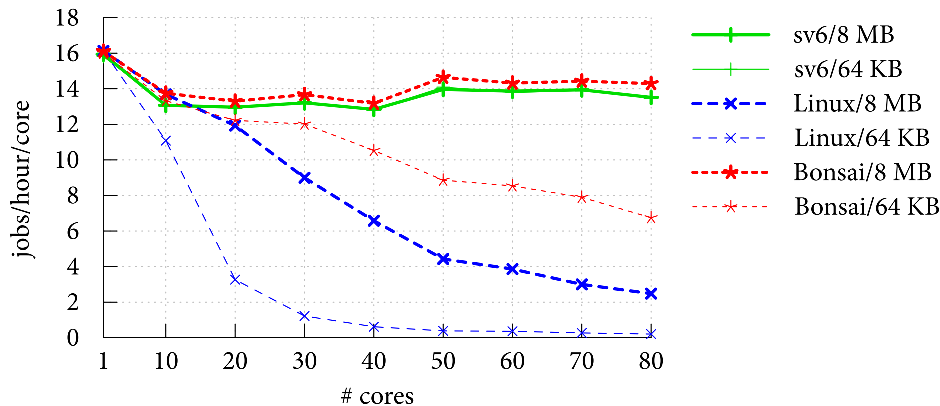

Figure 9-4: Metis application scalability for different VM systems and allocation unit sizes. sv6/8 MB and sv6/64 KB perform identically, so their lines are indistinguishable.

Metis on sv6 not only scales near-linearly, but performs identically in both the 64 KB and 8 MB configurations because virtually all of the virtual memory operations performed by Metis are commutative and thus conflict-free on RadixVM.

In contrast, Metis scales poorly on Linux in both configurations, spending most of its time at high core counts in the virtual memory system acquiring the address space lock rather than performing useful computation. This is true even in the pagefault-heavy configuration because, while pagefaults run in parallel on Linux, acquiring the address space lock in shared mode induces cache line conflicts.

For the 8 MB pagefault-heavy configuration, Bonsai scales as well as RadixVM because it also achieves conflict-free pagefault operations. Furthermore, in this configuration sv6 and Bonsai perform similarly—sv6 is ~ 5% slower than Bonsai at all core counts—suggesting that there is little or no sequential performance penalty to RadixVM's very different design. It's likely we could close this gap with further work on the sequential performance of sv6. Bonsai suffers in the 64 KB configuration because, unlike sv6, Bonsai's mmap and munmap operations are not conflict-free.

9.6 Memory overhead

Thus far, we've considered two measures of throughput: scalability and sequential performance. This section turns to memory overhead, another important performance metric.

While ScaleFS's data structures closely resemble those of traditional Unix kernels, the scalability of RadixVM depends on a less-compact representation of virtual memory than a traditional binary tree of memory regions and per-core page tables.

To quantify the memory overhead of RadixVM's radix tree, we took snapshots of the virtual memory state of various memory-intensive applications and servers running on Linux

		Linux		Radix tree
	RSS	VMA tree	Page table	(rel. to Linux)
Firefox	352 MB	117 KB	1.5 MB	3.9 MB (2.4×)
Chrome	152 MB	124 KB	1.1 MB	2.4 MB (2.0×)
Apache	16 MB	44 KB	368 KB	616 KB (1.5×)
MySQL	84 MB	18 KB	348 KB	980 KB (2.7×)

Table 9-5: Memory usage for alternate VM representations. RSS (resident set size) gives the size of the memory mapped and faulted by the application. The other columns give the size of the address space metadata structures used by Linux and RadixVM.

and measured the space required to represent the address space metadata in both Linux and RadixVM. Linux uses a single object to represent each contiguous range of mapped memory (a *virtual memory area* or VMA), arranged in a red-black tree, which makes its basic address space layout metadata very compact. As a result, Linux must store information about the physical pages backing mapped virtual pages separately, which it cleverly does in the hardware page table itself, making the hardware page table a crucial part of the address space metadata. RadixVM, on the other hand, stores both OS-specific metadata and physical page mappings together in the radix tree. This makes the radix tree larger than the equivalent VMA tree, but means RadixVM can freely discard memory used for per-core hardware page tables.

Table 9-5 summarizes the memory overhead of these two representations for four applications: the Firefox and Chrome web browsers, after significant use, and the Apache web server and MySQL database server used by the EuroSys 2013 paper submission web site. Compared to each application's resident set size (the memory directly allocated by the application), the radix tree incurred at most a 3.7% overhead in total memory footprint.

9.7 Discussion

This chapter completes our trajectory from theory to practice. The benchmarks herein have demonstrated that commutative operations can be implemented to scale, confirming the scalable commutativity rule for complex interfaces and real hardware and validating the designs and design methodologies set out in chapter 8. Several of these benchmarks have also demonstrated the necessity of commutativity and conflict-freedom for scalability by showing that even a single contended cache line in a complex operation can severely limit scalability. Furthermore, all of this can be accomplished at little cost to sequential performance or memory use.

Ten

Future directions

In the course of research, every good result inspires at least two good questions. This chapter takes a step back and reviews some of the questions we have only begun to explore.

10.1 The non-scalable non-commutativity rule

The scalable commutativity rule shows that SIM-commutative regions have conflict-free implementations. It does not show the inverse, however: that given a non-commutative region, there is no implementation that is conflict-free (or linearly scalable) in that region.

In general, the inverse is not true. For example, consider an interface with two operations: increment and get, where get can return either the current value *or* unknown. increment and get operations do not SIM commute because the specification permits an implementation where get always returns the current value and thus the order of these operations is observable. But the specification does permits a trivially conflict-free implementation with *no* state where get always returns unknown. This disproves the inverse of the rule, but seems to do so on an unsatisfying technicality: this example is "too non-deterministic."

What if we swing to the other extreme and consider only interfaces that are *sequentially consistent* (every legal history has a sequential reordering that is also legal) and *sequentially deterministic* (the sequence of invocations in a sequential history uniquely determines the sequence of responses)? Many interfaces satisfy these constraints, such as arrays, ordered sets, and ordered maps, as well as many subsets of POSIX like file I/O (read/write/etc.) For sequentially consistent, sequentially deterministic interfaces, the inverse of the rule *is* true.

To see why, first consider an implementation that is conflict-free for Y in $X \parallel Y$. By definition, the state components read by all of the steps of each thread in Y must be disjoint from the state components written by steps on other threads, so the steps of a thread cannot be affected by the steps of other threads. Since the invocations on each thread will be the same in any reordering of Y and the implementation is conflict-free in Y, it *must* yield the same responses and the same final state for any reordering of Y.

However, if Y is non-SIM commutative in $X \parallel Y$ and the specification is sequentially consistent and sequentially deterministic, an implementation *must* yield some different re-

sponse or a different final state in some reordering of Y. Thus, an implementation cannot be conflict-free in Y.

Many real-world interfaces are sequentially consistent and sequentially deterministic, but many are not (e.g., unsurprisingly, POSIX). Indeed, we consider non-determinism an important tool in the design of broadly commutative interfaces.

We believe, however, that specifications with limited non-determinism are similarly constrained by the inverse of the rule. For example, a specification consisting exclusively of POSIX's open is sequentially consistent and sequentially deterministic (concurrent calls to open appear equivalent to sequential calls and non-concurrent calls have deterministic behavior). Adding *any FD* semantics makes open commute in more regions, but does not fundamentally alter the behavior of regions that did not commute for other reasons (such as concurrent, exclusive creates the same file name).

Understanding the inverse of the rule would shed light on the precise boundaries of scalable implementations. However, the greater value of the inverse may be to highlight ways to escape these boundaries. The following two sections explore hardware features not captured by the rule's formalization that may expand the reach of the rule and enable scalable implementations of broader classes of interfaces.

10.2 Synchronized clocks

Section 4.3 formalized implementations as step functions reflecting a machine capable of general computation and communication through shared memory. Some hardware has at least one useful capability not captured by this model: *synchronized timestamp counters*.

Reads of a synchronized timestamp counter will always observe increasing values, even if the reads occur on different cores. With this capability, operations in a commutative region can record their order *without communicating* and later operations can depend on this order. For example, consider an append operation that appends data to a file. With synchronized timestamp counters, the implementation of append could log the current timestamp and the appended data to per-thread state. Later reads of the file could reconcile the file contents by sorting these logs by timestamp. The appends do not commute, yet this implementation of *append* is conflict-free.

Formally, we can model a synchronized timestamp counter as an additional argument to the implementation step function that must increase monotonically over a history. With this additional argument, many of the conclusions drawn by the proof of the scalable commutativity rule no longer hold.

Recent work by Boyd-Wickizer developed a technique for scalable implementations called OpLog based on using synchronized timestamp counters [9]. Exactly what interfaces are amenable to OpLog implementations remains unclear and is a question worth exploring.

10.3 Scalable conflicts

Another potential way to expand the reach of the rule and create more opportunities for scalable implementations is to find ways in which non-conflict-free operations can scale.

For example, while streaming computations are in general not linearly scalable because of interconnect and memory contention, we've had success with scaling *interconnect-aware* streaming computations. These computations place threads on cores so that the structure of sharing between threads matches the structure of the hardware interconnect and such that no link is oversubscribed. For example, on the 80-core x86 from chapter 9, repeatedly shifting tokens around a ring mapped to the hardware interconnect achieves the same throughput regardless of the number of cores in the ring, even though every operation causes conflicts and communication.

It is unclear what useful computations can be mapped to this model given the varying structures of multicore interconnects. However, this problem has close ties to job placement in data centers and may be amenable to similar approaches. Likewise, the evolving structures of data center networks could inform the design of multicore interconnects that support more scalable computations.

10.4 Not everything can commute

This dissertation advocated fixing scalability problems starting by making interfaces as commutative as possible. But some interfaces are set in stone, and others, such as synchronization interfaces, are fundamentally non-commutative. It may not be possible to make implementations of these scale linearly, but making them scale as well as possible is as important as making commutative operations scale.

This dissertation addressed this problem only in ad hoc ways. Most notably, Refcache's design focused all of the non-commutativity and non-scalability inherent in resource reclamation into a single, non-critical-path operation. This reclamation operation ran only once every 10ms, allowing Refcache to batch and eliminate many conflicts and amortize the cost of this operation. However, whether there is a general interface-driven approach to the scalability of non-commutative operations remains an open question.

10.5 Broad conflict-freedom

As evidenced by sv6 in chapter 8, real implementations of real interfaces can be conflict-free in nearly all commutative situations. But, formally, the scalable commutativity rule states something far more restricted: that for a *specific* commutative region of a *specific* history, there is a conflict-free implementation. In other words, there is some implementation that is conflict-free for each of the 26,238 tests COMMUTER ran, but passing all of them might

require 26,238 different implementations. This strictness is a necessary consequence of SIM commutativity, but, of course, sv6 shows that reality is far more tolerant.

This gap between the theory and the practice of the rule suggests that there may be a space of trade-offs between interface properties and construction generality. A more restrictive interface property may enable the construction of broadly conflict-free implementations. If possible, this "alternate rule" may capture a more practically useful construction; perhaps even a construction that could be applied mechanically to build practical scalable implementations.

Eleven

Conclusion

We are in the midst of a sea change in software performance, as everything from top-tier servers to embedded devices turns to increasing parallelism to maintain a competitive performance edge.

This dissertation introduced a new approach for software developers to understand and exploit multicore scalability during software interface design, implementation, and testing. We defined, formalized, and proved the scalable commutativity rule, the key observation that underlies this new approach. We defined SIM commutativity, which allows developers to apply the rule to complex, stateful interfaces. We further introduced Commuter to help programmers analyze interface commutativity and test that an implementation scales in commutative situations. Finally, using sv6, we showed that it is practical to achieve a broadly scalable implementation of POSIX by applying the rule, and that commutativity is essential to achieving scalability and performance on real hardware. As scalability becomes increasingly important at all levels of the software stack, we hope that the scalable commutativity rule will help shape the way developers meet this challenge.

Bibliography

[1] 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), August 2007.

[2] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages*, Austin, TX, January 2011.

[3] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, Canada, August 2009.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[5] Fabrice Bellard et al. QEMU. http://www.qemu.org/.

[6] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the ACM Workshop on Computer Security Architecture*, Fairfax, VA, November 2007.

[7] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[8] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: a software approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–122, Boston, MA, April 1989.

[9] Silas Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, February 2014.

[10] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. 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 *Proceedings of the 8th Symposium*

on Operating Systems Design and Implementation (OSDI), San Diego, CA, December 2008.

[11] 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 *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[13] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[14] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Communications of the ACM*, 51(11):34–39, 2008.

[15] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada, September 2000.

[16] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Concurrent address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

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

[18] Super Micro Computer. X8OBN-F manual, 2012.

[19] Jonathan Corbet. The search for fast, scalable counters, May 2010. http://lwn.net/Articles/170003/.

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

[21] Tyan Computer Corporation. M4985 manual, 2006.

[22] Tyan Computer Corporation. S4985G3NR manual, 2006.

[23] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2012/xv6.html.

[24] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, March–April 2008.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, November 1990.

[27] Advanced Micro Devices. *AMD64 Architecture Programm's Manual*, volume 2. Advanced Micro Devices, March 2012.

[28] DWARF Debugging Information Format Committee. DWARF debugging information format, version 4, June 2010.

[29] Faith Ellen, Yossi Lev, Victor Luchango, and Mark Moir. SNZI: Scalable nonzero indicators. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[30] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference*, Ottawa, Canada, April 2006.

[31] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–100, New Orleans, LA, February 1999.

[32] Sanjay Ghemawat. TCMalloc: Thread-caching malloc, 2007. http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[34] J. R. Goodman and H. H. J. Hum. MESIF: A two-hop cache coherency protocol for point-to-point interconnects. Technical report, University of Auckland and Intel, 2009.

[35] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[36] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[37] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[38] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. Technical Report 10-06, Portland State University, Computer Science Department, 2010.

[39] David Howells. Extended file stat functions, Linux patch, 2010. https://lkml.org/lkml/2010/7/14/539.

[40] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3. Intel Corporation, 2013.

[41] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Los Angeles, CA, August 1994.

[42] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, Madrid, Spain, September 2002.

[43] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, May 2005. http://www.lameter.com/gelato2005.pdf.

[44] Ran Liu and Haibo Chen. SSMalloc: A low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[45] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT CSAIL, May 2010.

[46] Paul McKenney. Hierarchical RCU, November 2008. https://lwn.net/Articles/305782/.

[47] Paul E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1999.

[48] Paul E. McKenney. Concurrent code and expensive instructions. https://lwn.net/Articles/423994/, January 2011.

[49] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2002.

[50] 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.

[51] Microsoft Corp. Windows research kernel. http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx.

[52] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, Ann Arbor, MI, June 1984.

[53] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, CA, June 2011.

[54] The FreeBSD Project. FreeBSD source code. http://www.freebsd.org/.

[55] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, November 1997.

[56] Amitabha Roy, Steven Hand, and Tim Harris. Exploring the limits of disjoint access parallelism. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

[57] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. of the 2006 ACM SIGPLAN International Symposium on Memory Management*, Ottawa, Canada, June 2006.

[58] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal, September 2005.

[59] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, Grenoble, France, October 2011.

[60] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, June 2011.

[61] Guy L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[62] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *SIGPLAN Notices*, 46(11):79–88, June 2011.

[63] The Institute of Electrical and Electronics Engineers (IEEE) and The Open Group. The Open Group base specifications issue 7, 2013 edition (POSIX.1-2008/Cor 1-2013), April 2013.

[64] Linus Torvalds et al. Linux source code. http://www.kernel.org/.

[65] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 1995.

[66] Landy Wang. Windows 7 memory management, November 2009. http://download.microsoft.com/download/7/E/7/7E7662CF-CBEA-470B-A97E-CE7CE0D98DC2/mmwin7.pptx.

[67] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[68] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating System Review*, 43(2):76–85, 2009.