The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

AUSTIN T. CLEMENTS, M. FRANS KAASHOEK, NICKOLAI ZELDOVICH,
and ROBERT T. MORRIS, MIT CSAIL
EDDIE KOHLER, Harvard University

What opportunities for multicore scalability are latent in software interfaces, such as system call APIs? Can scalability challenges and opportunities be identified even before any implementation exists, simply by considering interface specifications? To answer these questions, we introduce the scalable commutativity rule: *whenever interface operations commute, they can be implemented in a way that scales.* This rule is useful throughout the development process for scalable multicore software, from the interface design through implementation, testing, and evaluation.

This article formalizes the scalable commutativity rule. This requires defining a novel form of commutativity, *SIM commutativity*, that lets the rule apply even to complex and highly stateful software interfaces.

We also introduce a suite of software development tools based on the rule. Our COMMUTER tool accepts high-level interface models, generates tests of interface 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 highlights Linux kernel problems previously observed to limit application scalability and identifies previously unknown bottlenecks that may be triggered by future workloads or hardware.

Finally, we apply the scalable commutativity rule and COMMUTER to the design and implementation sv6, a new POSIX-like operating system. 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.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

General Terms: Design, Performance, Theory

Additional Key Words and Phrases: Commutativity, conflict freedom, multicore, software interfaces

ACM Reference Format:
Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. ACM Trans. Comput. Syst. 32, 4, Article 10 (January 2015), 47 pages.
DOI: http://dx.doi.org/10.1145/2699681

This article extends and adapts the conference version that appeared in the *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)* [Clements et al. 2013b] and Austin Clements' Ph.D. dissertation [Clements 2014].
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. Eddie Kohler was partially supported by a Microsoft Research New Faculty Fellowship and a Sloan Research Fellowship.
Authors' addresses: Computer Science and Artificial Intelligence Laboratory, MIT, 32 Vassar St, Cambridge, MA; emails: aclements@csail.mit.edu, kaashoek@mit.edu, nickolai@mit.edu, rtm@mit.edu, kohler@seas.harvard.edu.

0734-2071/2015/01-ART10
DOI: http://dx.doi.org/10.1145/2699681

1. INTRODUCTION

This article presents a formal and pragmatic approach to the design and implementation of scalable multicore software, spanning the earliest stages of software interface design through to the testing and maintenance of complete implementations.

The rest of this section 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

The mid-2000s saw a fundamental shift in programming techniques for high-performance software. In prior decades, CPU clock speeds, which rode the exponential curve of Moore's Law, made sequential software faster automatically. Unfortunately, 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. *Total* cycles per second continues to grow exponentially, but software must *scale*—must take advantage of parallel CPU resources—to take advantage of this growth. Parallel programming has gone from niche to necessary. Unfortunately, scaling with parallelism is still 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. But the kernel must cope with diverse and unknown workloads, and its role as the arbiter of shared resources makes it particularly susceptible to scalability problems. 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 [McKenney 1999] to identify scalability bottlenecks. This approach focuses developer effort on demonstrable issues but is ultimately nearsighted. Each new hardware model or workload restarts a Sisyphean cycle of finding and fixing scalability 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 these.

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 [IEEE 2013]. 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 few applications care about file descriptor values, POSIX—the standard for the Unix interface—requires that open return the numerically lowest file descriptor available in the calling process. This forces the kernel to coordinate file descriptor allocation

across all threads, even when many threads are opening files in parallel. This choice simplified the kernel interface during the early days of Unix, but it 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 [Boyd-Wickizer et al. 2008], 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 article 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 even before the necessary hardware is available. It can highlight inherent scalability problems, leading to better interface designs. It sets a clear scaling target for the implementation of a scalable interface. Finally, it enables systematic testing of an implementation's scalability.

At the core of our approach is what we call the *scalable commutativity rule*: in any situation where several operations *commute* (meaning there's no way to distinguish their execution order using the interface), there exists an implementation that is *conflict free* during those operations (meaning no core writes a cache line that was read or written by another core). Since conflict-free operations empirically scale, this implementation scales. Thus, 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 article 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 a specific context of system state, operation arguments, and 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 would a more conventional notion of commutativity. 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.

This article also shows that in certain situations, we can reason about when operations must conflict; that is, SIM commutativity is not only sufficient but also necessary

for operations to be conflict free. This *converse scalable commutativity rule* is more limited in scope than the scalable commutativity rule but helps to broaden our understanding of the relationship between SIM commutativity and conflict freedom.

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, imagine that multiple processes are creating 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 the *implementations* we could think of. This process—difficult, unguided, and appropriate only for simple interfaces—motivated us to reason about scalability in terms of interfaces.

Real-world interfaces and implementations are complex. Even with the rule, it can be difficult to spot and reason about all commutative cases. To address this challenge, this article 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 found by COMMUTER represent opportunities for the developer to improve the scalability of the implementation. The 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 [Boyd-Wickizer et al. 2010]. 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 article makes the following 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.
—sv6, an OS kernel that demonstrates the application of these techniques to a POSIX virtual memory system and file system.

Together, these ideas are the basis for a new approach to building scalable software, one where interface-based reasoning guides design, implementation, and testing.

We validate sv6, and our design methodology, by evaluating its performance and scalability on an 80-core x86 machine.

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

1.5. Outline

The rest of this article 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 Section 2.

We then turn to formalizing and proving the scalable commutativity rule, which we approach in two steps. First, Section 3 establishes experimentally that conflict-free operations are generally scalable on modern, large multicore machines. Section 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. Section 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.

Section 6 presents COMMUTER, which uses the rule to automate reasoning about interface commutativity and the conflict freedom of implementations. Section 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. Section 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. Section 9 confirms that theory translates into practice by evaluating the performance and scalability of sv6 on real hardware for several microbenchmarks and application benchmarks.

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

2. RELATED WORK

This section relates the rule and its use in sv6 and COMMUTER to prior work.

2.1. Thinking About Scalability

Israeli and Rappoport [1994] introduce the notion of disjoint-access-parallel memory systems. 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. [2009] extend Israeli and Rappoport's definition to additionally require nondisjoint reads to scale. Our work builds on the assumption that memory systems behave this way, and we confirm that real hardware closely approximates this behavior (Section 3).

Both the original disjoint-access parallelism paper and subsequent work, including the paper by Roy et al. [2009], explore the scalability of processes that have some amount of nondisjoint 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 (Sections 3 and 9).

The Laws of Order [Attiya et al. 2011] explore the relationship between the *strong noncommutativity* 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 [2011] 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.

It is well understood that cache-line contention can result in bad scalability. A clear example is the design of the MCS lock [Mellor-Crummey and Scott 1991], which eliminates scalability collapse by avoiding contention for a particular cache line. Other good examples include scalable reference counters [Corbet 2010; Ellen et al. 2007]. 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 [Cantrill and Bonwick 2008]. 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 article identifies situations where POSIX permits or limits scalability and points out specific interface modifications that would permit greater implementation scalability.

Scalable kernels such as K42 [Appavoo et al. 2007], Tornado [Gamsa et al. 1999], and Hurricane [Unrau et al. 1995] have introduced and used design patterns such as clustered objects and locality-preserving IPC. 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 noncommutative operations.

Multikernels for multicore processors aim for scalability by avoiding shared data structures in the kernel [Baumann et al. 2009; Wentzlaff and Agarwal 2009]. 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 message-passing systems as well as shared-memory systems.

The designers of the Corey operating system [Boyd-Wickizer et al. 2008] argue that applications should manage the cost of sharing but do not provide a guideline for how to 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 [1990] describes a parallel programming discipline in which all operations must be either causally related or commutative. His work approximates commutativity as conflict freedom. This article shows that commutative operations always have a conflict-free implementation, showing that Steele's model is broadly applicable. Rinard and Diniz [1997] describe how to exploit commutativity to automatically parallelize code. They allow memory conflicts but generate synchronization code to ensure atomicity of commutative operations. Similarly, Prabhu et al. [2011] describe how to automatically parallelize code using manual annotations rather than automatic commutativity analysis. 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 [Bernstein and Goodman 1981]. Weihl [1988] extends this work to abstract data types by deriving lock conflict relations from operation commutativity. Transactional boosting applies similar techniques in the context of software transactional memory [Herlihy and Koskinen 2008]. Shapiro et al. [2011a, 2011b] extend this to a distributed setting, leveraging commutative operations in the design of replicated data types that support updates during faults and network partitions. Like Rinard and Prabhu's work, the work in databases and its extensions focus on the *safety* of executing commutative operations concurrently, not directly on scalability.

2.4. Test Case Generation

Concolic testers [Godefroid et al. 2005; Sen et al. 2005] and symbolic execution systems [Cadar et al. 2006, 2008] generate 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 model-based testing in QuickCheck [Claessen and Hughes 2000] or Gast [Koopman et al. 2002] 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.

3. SCALABILITY AND CONFLICT FREEDOM

Understanding multicore scalability requires first understanding the hardware. This section shows that, under reasonable assumptions, conflict-free operations scale linearly on modern multicore hardware. The following section will use conflict freedom to establish the scalable commutativity rule.

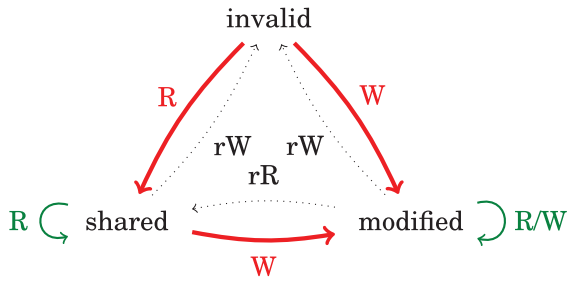


Fig. 1. A basic cache-coherence state machine. "R" and "W" indicate local read and write operations, while "rR" and "rW" indicate remote read and write operations, respectively. Thick red lines show operations that cause communication. Thin green lines show operations that occur without communication.

3.1. Conflict Freedom and Multicore Processors

The connection between conflict freedom and scalability mustn't be taken for granted. Indeed, some early multiprocessor architectures such as the Intel Pentium depended on shared buses with global lock lines [Intel 2013, §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 [Papamarcos and Patel 1984] 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 1 shows the basic state machine implemented by each cache for each cache line. This maintains the previous 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 [Goodman and Hum 2009], and AMD's "owned" state [Advanced Micro Devices 2012, §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 ongoing communication. There are three memory access patterns that do not require 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. This scales for much the same reason.
—Multiple cores reading the same cache line. A copy of the line can be kept in each core's cache in shared mode; further reads from those cores can access the line 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

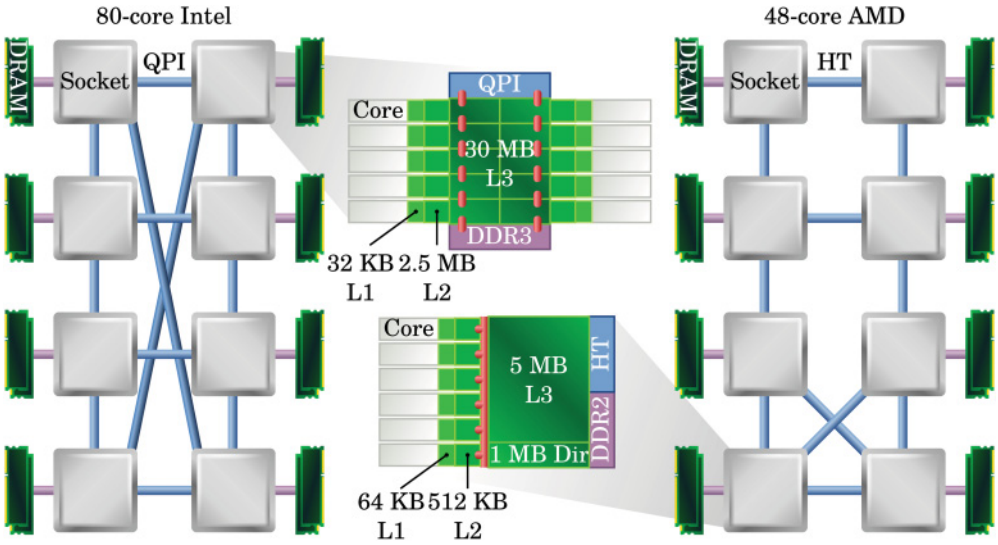


Fig. 2. Organization of Intel and AMD machines used for benchmarks [Super Micro Computer 2012; Tyan Computer Corporation 2006a, 2006b].

to N. Therefore, given a perfect implementation of MESI, conflict-free operations scale linearly. The following sections verify that 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.

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 $\times$ 10 cores) Intel Xeon E7-8870 (the same machine used for evaluation in Section 9) and, to show that our conclusions generalize, a 48-core (8 sockets $\times$ 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 2 shows how the two machines are broadly organized.

Figure 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, 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 shows the results of stressing conflict-free writes. Each of N cores is assigned a different cache line, which it repeatedly writes. In this case, each

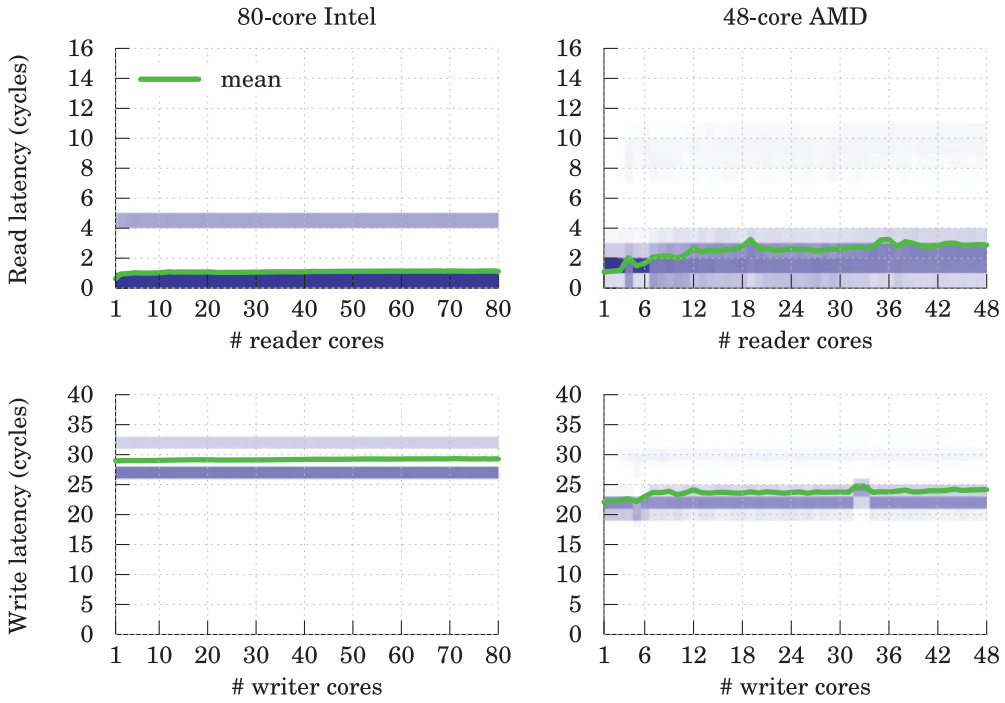


Fig. 3. Each graph shows the cycles required to perform a conflict-free read or write from N cores. Shading indicates the latency distribution for each N (darker shading indicates higher frequency).

cache line enters a "modified" state but then remains in that state: 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 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 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 to 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 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.

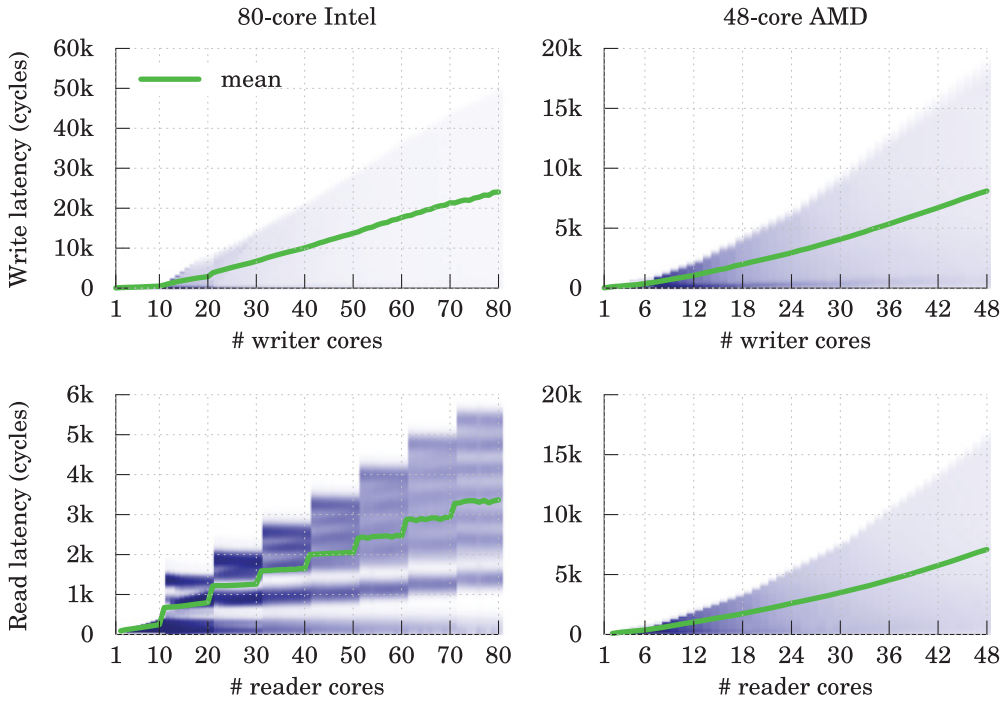


Fig. 4. Conflicting accesses do not scale. Each graph shows the cycles required to perform a conflicting read or write from N cores. Shading indicates the latency distribution for each N (estimated using kernel density estimation).

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 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 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 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.

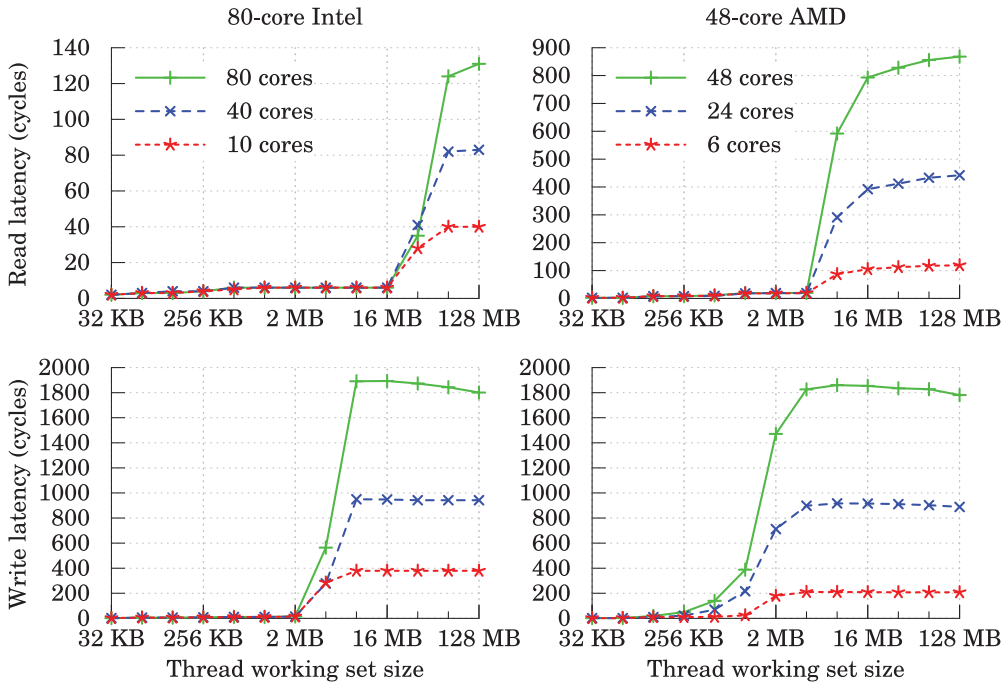


Fig. 5. Operations scale until they exceed cache or directory capacity. Each graph shows the latency for repeatedly reading a shared region of memory (top) and writing separate per-core regions (bottom) as a function of region size and number of cores.

We observe a similar effect for writes, shown in the bottom row of Figure 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 1MB 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.

Section 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.

4. THE SCALABLE COMMUTATIVITY RULE

This section 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 section. 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. Finally, we consider the *converse scalable commutativity rule*, which lets us argue in limited ways about when conflicts must arise.

4.1. Actions

Following earlier work [Herlihy and Wing 1990], 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,

$$H = \text{A}\ \text{B}\ \text{C}\ \text{A}\ \text{C}\ \text{B}\ \text{D}\ \text{D}\ \text{E}\ \text{E}\ \text{F}\ \text{G}\ \text{H}\ \text{F}\ \text{H}\ \text{G}$$

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 $H|t =$ A A D D H H, which selects t's actions from H, alternates invocations and responses as expected. 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 $\mathcal{S}$ is a prefix-closed set of well-formed histories. The set's contents depend on the system being modeled; for example, if $\mathcal{S}$ specified a Unix-like OS, then [A = getpid(), A = 0] $\notin \mathcal{S}$, 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 means that an interface's caller cannot distinguish the order in which concurrent actions actually occurred, either by the actions' responses or through any possible future actions. Thus, the order of a set of commutative actions "doesn't matter": the specification is indifferent to the execution order of that set. 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 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 H' is a *reordering* of an action sequence H when $H|t = H'|t$ for every thread t.

If $H =$  , then  is a reordering of H, but  is not, since it doesn't respect the order of actions in H's red thread. A reordering of H contains the same invocations and responses as H, and within each thread the operation order is unchanged, but the reordering may change the way different threads' operations interleave.

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

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

This definition captures the state dependence and interface basis we need. The action sequence X puts the system into a specific state, without specifying a 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 can be *nonmonotonic*: later operations in a history can change whether prior operations SI-commute. This surprising property seems incompatible with scalable implementation. 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, the prefix Y_1 does not SI-commute on its own: since some orders set the value to 1 and some to 2, future get operations could distinguish them. Whether or not Y_1 will ultimately form part of a commutative region thus depends on *future* operations! Real implementations cannot predict what operations will be called in the future, so operations in a region like Y_1 would "plan for the worst" by remembering their order. In our machine models, tracking a precise order of operations induces conflicts and limits

scalability. To draw a connection between conflict freedom and commutativity, we must require monotonicity.

Definition. An action sequence Y *SIM-commutes* in a history $H = X||Y$ when for any *prefix* P of any reordering of Y (including $P = Y$), P SI-commutes in $X||P$. Equivalently, Y SIM-commutes in H when, given any prefix P of any reordering of Y, any reordering P' of P, and any action sequence Z,

$$X \,||\, P \,||\, Z \in \mathscr{S} \quad \text{if and only if} \quad X \,||\, P' \,||\, Z \in \mathscr{S}.$$

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 does not SI-commute.

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 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, or scheduler, that repeatedly picks a thread to step forward and passes state from step to step. Special YIELD responses let the step function request that the driver "run" a different thread and help represent concurrent overlapping operations and blocking operations.

We begin by defining S, the set of implementation states. To help us reason about conflicts, we divide states into *components* indexed by a component set C. Two implementation steps will conflict if they access at least one shared component and at least one of those accesses is a write. Given $s \in S$ and $c \in C$, we write $s.c$ for the value of the cth component of s. These values are left opaque, except that we assume they can be compared for equality. We write $s\{c \leftarrow x\}$ for component replacement; $s\{c \leftarrow x\}$ is a state so that, given any $c' \in C$,

$$(s\{c \leftarrow x\}).c' = \begin{cases} x & \text{if } c' = c, \\ s.c' & \text{otherwise.} \end{cases}$$

We extend these notations to component sequences in the natural way. For example, if $\mathbf{c} = [c_1, \ldots, c_n]$ is a sequence of components and $\mathbf{x} = [x_1, \ldots, x_n]$ a sequence of values, then $s\{\mathbf{c} \leftarrow \mathbf{x}\} = s\{c_1 \leftarrow x_1\} \cdots \{c_n \leftarrow x_n\}$, and, given another state s',

$$s\{\mathbf{c} \leftarrow s'.\mathbf{c}\} = s\{c_1 \leftarrow s'.c_1\} \cdots \{c_n \leftarrow s'.c_n\}.$$

In addition, let I be the set of valid implementation invocations, which is the set of specification invocations plus a special CONTINUE invocation; and let R be the set of valid implementation responses, which is the set of specification responses plus a special YIELD response.

Definition. An *implementation* m is a function in $S \times I \mapsto S \times R \times \mathscr{P}(C)$: given an old state and an invocation, the implementation produces a new state, a response, and a set of components called the *access set*. An implementation must obey four restrictions. For any step $m(s, i) = \langle s', r, \mathbf{a} \rangle$, we must have:

(1) *Response consistency.* The implementation returns the response to the same thread as the invocation: $\text{thread}(i) = \text{thread}(r)$.

(2) *Write inclusion.* Any state component changed by the step is in the access set: for any $c \notin \mathbf{a}$, $s.c = s'.c$.
(3) *Access restriction.* The implementation's behavior is indifferent to state components not in the access set: for any $c \notin \mathbf{a}$ and any value x,

$$m(s\{c \leftarrow x\}, i) = \langle s'\{c \leftarrow x\}, r, \mathbf{a}\rangle.$$

(4) *Interruptibility.* The implementation responds with YIELD to invocations that are not CONTINUE: if $i \neq$ CONTINUE, then $r =$ YIELD. This last requirement is not fundamental (an implementation that does not satisfy this can be transformed into one that does), but it simplifies some arguments.

We say an implementation step $m(s, i) = \langle s', r, \mathbf{a}\rangle$ *accesses* (reads and/or writes) all components $c \in \mathbf{a}$ and specifically *writes* a component $c \in \mathbf{a}$ if $s'.c \neq s.c$.

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 (including CONTINUE invocations) produce the corresponding history. The particular sequence of invocations to the implementation that generates a history is a *witness* of that implementation generating that history. For example, this sequence shows an implementation m, given the invocations [A, B, CONTINUE, CONTINUE, CONTINUE], generating the history A B B A:

— $m(s_0, \text{A}) = \langle s_1, \text{YIELD}, \mathbf{a_1}\rangle$
— $m(s_1, \text{B}) = \langle s_2, \text{YIELD}, \mathbf{a_2}\rangle$
— $m(s_2, \text{CONTINUE}) = \langle s_3, \text{YIELD}, \mathbf{a_3}\rangle$
— $m(s_3, \text{CONTINUE}) = \langle s_4, \text{B}, \mathbf{a_4}\rangle$
— $m(s_4, \text{CONTINUE}) = \langle s_5, \text{A}, \mathbf{a_5}\rangle$

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.

Two implementation steps have an *access conflict* when they are on different threads and one writes a state component that the other accesses (reads or writes). This notion of access conflicts maps directly onto the read and write access conflicts on real shared-memory machines explored in Section 3. A set of implementation steps is *conflict free*

[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 nondeterminism, though this would complicate later arguments somewhat.

when no pair of steps in the set has an access conflict; that is, no thread's steps access 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 $H = X \,||\, Y$ 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 (Section 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 $\mathscr{S}_{\text{refctr}}$. $\mathscr{S}_{\text{refctr}}$ has a simple implementation with a single shared counter component "ctr." We can represent this implementation as

$$m(\{\text{ctr} \leftarrow x\}, \text{inc}) \equiv \langle \{\text{ctr} \leftarrow x+1\}, x+1, \{\text{ctr}\} \rangle;$$
$$m(\{\text{ctr} \leftarrow x\}, \text{dec}) \equiv \langle \{\text{ctr} \leftarrow x-1\}, x-1, \{\text{ctr}\} \rangle;$$
$$m(\{\text{ctr} \leftarrow x\}, \text{iszero}) \equiv \langle \{\text{ctr} \leftarrow x\}, x = 0, \{\text{ctr}\} \rangle.$$

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

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

The region H_{AB} SIM-commutes in H. Thus, by the rule, there is an implementation of $\mathscr{S}_{\text{refctr}}$ that is conflict free for H_{AB}. In fact, this is already true of the shared-counter implementation: its iszero reads ctr but does not write 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 way to make H_{CD} conflict free: if we modify the specification so that inc and dec return nothing, then these modified operations commute (more precisely: any region consisting exclusively of these operations commutes in any history). With this modified specification, $\mathscr{S}'_{\text{refctr}}$, 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 tradeoff [Boyd-Wickizer et al. 2010; Clements et al. 2013a; DeTreville 1990].

The equivalent history with this modified specification is

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

Unlike H_{CD}, H'_{CD} SIM-commutes; accordingly, there is an implementation of $\mathscr{S}'_{\text{refctr}}$ 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 previously for H'_{CD} is not conflict free for H'_{ABC}: 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.

These two implementations of $\mathscr{S}'_{\mathsf{refctr}}$ 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). An implementer must determine the scaling opportunities that exist, decide which are likely to be the most valuable, and choose the implementation that scales in those situations.

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_{rule} from an arbitrary reference implementation M and history $H = X \,||\, Y$. 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 $X \,||\, P$ 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_{rule} scales.

To understand the construction, it helps to first imagine deriving a *nonscalable* implementation m_{replay} from the reference M. This nonscalable implementation begins in *replay mode*. As long as each invocation matches the next invocation in H, m_{replay} simply replays the corresponding responses from H, without invoking the reference implementation. If the input invocations diverge from H, m_{replay} 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_{replay} passes all invocations to the reference implementation and returns its responses.

A state s for m_{replay} 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, $s.refstate$ is the state of the reference implementation and starts as the value of the reference implementation's initial state. Figure 6 shows how the simulated implementation works. We make several simplifying assumptions, including that m_{replay} receives CONTINUE invocations in a restricted way; these assumptions aren't critical for the argument. One line requires expansion, namely, the choice of a witness 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_{replay} 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: the specification doesn't distinguish among orders. Thus, it is safe to initialize the reference implementation with the commutative actions *in a different order than that in which they were received*. All future responses will still be valid according to the specification.

```
m_replay(s, i) ≡
  If head(s.h) = i:
    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' ← a witness 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, i)
    a ← {h, refstate}
  else:                                                    // replay mode
    s.h ← tail(s.h)
    a ← {h}
  Return ⟨s, r, a⟩
```

Fig. 6. Constructed nonscalable implementation m_{replay} for history H and reference implementation M.

```
m_rule(s, i) ≡
  t ← thread(i)
  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]) = i:
    r ← CONTINUE
  else if i = 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' ← a witness 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, i)
    a ← C                                                  // all state components
  else if s.commute[t]:                                    // conflict-free mode
    s.h[t] ← tail(s.h[t])
    a ← {commute[t], h[t]}
  else:                                                    // replay mode
    s.h[u] ← tail(s.h[u]) for each thread u
    a ← C
  Return ⟨s, r, a⟩
```

Fig. 7. Constructed scalable implementation m_{rule} for history H and reference implementation M.

Figure 7 shows the construction of m_{rule}, a version of M that scales over Y in $H = X||Y$. m_{rule} is similar to m_{replay} 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 || COMMUTE || $(Y|t)$, where the special COMMUTE action indicates the commutative region has begun.

—$s.commute[t]$—a per-thread flag indicating whether the commutative region has been reached. Initialized to FALSE.
—$s.refstate$—the reference implementation's state.

Each step of m_{rule} 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_{replay}, 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 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 constructed by the proof is impractical and real implementations achieve broad scalability using different techniques, such as the ones this article explores in Section 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: put(x) records a sample with value x, and max() returns the maximum sample recorded

[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_{rule} 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_{rule} 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.

so far (or 0). Suppose

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

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 Section 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 Section 10. However, as the rest of this article shows, the rule is already an effective guideline for achieving practical scalability.

4.8. Converse Rule

The scalable commutativity rule shows that SIM-commutative regions have conflict-free implementations. However, it does not show the converse—that given a noncommutative region, no correct implementation can be conflict free there.

The converse is, in fact, not strictly true. Consider a counter whose interface consists of inc, which increments the counter value, and maybe-get, which either returns the value or returns "retry." inc and maybe-get do not SIM-commute. For instance, consider the history

$$H = [\underbrace{\mathsf{A} = \text{maybe-get}, \mathsf{B} = \text{inc}, \mathsf{B}}_{Y}, \underbrace{\mathsf{A} = 0}_{Z}].$$

Y does not SIM-commute, and the suffix Z shows why. $H = Y \,||\, Z \in \mathscr{S}$—the maybe-get and inc operations overlap, so maybe-get may return the old value—but with the reordering

$$Y' = [\mathsf{B} = \text{inc}, \mathsf{B}, \mathsf{A} = \text{maybe-get}],$$

$Y' \,||\, Z \notin \mathscr{S}$, since in this order the maybe-get must return 1 or "retry," but Z gives a response of 0. Nevertheless, a correct implementation m_{retry} exists that is conflict free in *every* history. m_{retry} simply does nothing for inc and always returns "retry" for maybe-get.

Although this disproves the rule's converse, it does so on an unsatisfying technicality. A version of the converse *is* true for specifications *generated by an implementation*. That is, given an implementation m of some specification $\mathscr{S}$, we consider only the subset $\mathscr{S}|m$ of histories that m can generate. In our example, $\mathscr{S}|m_{\text{retry}}$ does not contain histories where maybe-get returns a value, so *all* histories in $\mathscr{S}|m_{\text{retry}}$ SIM-commute and the existence of a conflict-free implementation is no surprise.

Given an implementation m, SIM commutativity with respect to $\mathscr{S}|m$ is both sufficient *and necessary* for m's steps in a SIM-commutative region to be conflict free. The

forward case is a direct consequence of the scalable commutativity rule. The reverse case—the *converse scalable commutativity rule*—is a consequence of the restriction to $\mathscr{S}|m$. Stated precisely: given a specification $\mathscr{S}$, a history $H = X||Y \in \mathscr{S}$, and a correct implementation m of $\mathscr{S}$ that can generate H, if Y does not SIM-commute in $X||Y$ with respect to $\mathscr{S}|m$, then m's steps in Y must conflict.

The full proof is given in Appendix A and builds on the fact that reordering a region of operations with conflict-free implementation steps does not change the implementation's responses or final state or the conflict freedom of those steps. It follows that, if m's steps in Y are conflict free, Y must SIM-commute with respect to $\mathscr{S}|m$. Hence, if Y does *not* SIM-commute with respect to $\mathscr{S}|m$, m's steps in Y must have a conflict.

The restriction to $\mathscr{S}|m$ limits the scope of the converse rule somewhat. However, many real implementations achieve broad coverage of their specifications. This is simply a consequence of engineering; if it is not true, the specification is probably too underspecified to be useful, or the implementation interprets the specification too narrowly to be useful. Many specifications (especially those involving network communication) have a notion of retrying—allowing trivially conflict-free implementations like m_{retry}—but any worthwhile implementation should be capable of returning the actual result eventually.

The converse rule also gives us insight into how an implementation can achieve broader conflict freedom and hence better scalability. Any implementation that can generate the same set of histories is bound by the reverse rule to have conflicts in the same situations. However, an implementation that generates a strictly smaller subset of the same specification may be able to achieve conflict freedom in situations where a more "general" implementation of the same specification cannot. If $\mathscr{S}|m_1 \supsetneq \mathscr{S}|m_2$, regions that do not SIM-commute in $\mathscr{S}|m_1$ may SIM-commute in $\mathscr{S}|m_2$, so m_2 may be conflict free where m_1 cannot be.

We can gain further insight by revisiting the assumptions of the reverse rule. In particular, the reverse rule depends on the implementation model described in Section 4.3. This is not the only possible model. Section 10 considers other implementation models that let us escape the constraints of the reverse rule and potentially achieve broader conflict freedom.

5. 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 section 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 sections; this section 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 suboperations. With only fork and exec, applications are forced to accept these unnecessary suboperations 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 suboperations. 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 noncommutative 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 Nondeterminism

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 could use scalable allocation methods, which are well known. We will return to this example 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 noncommutative. 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 multireader or multiwriter 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. This in turn would allow an implementation of Unix domain sockets to support scalable communication. We return to this example 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 a 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 nonscalable remote TLB shootdowns before munmap can

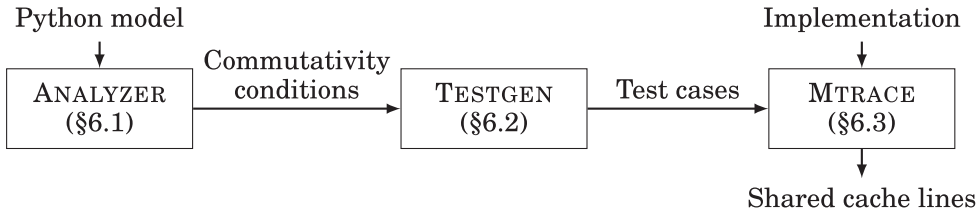


Fig. 8. The components of COMMUTER.

return, even though depending on this behavior usually indicates a bug. An munmap (or an madvise) that released virtual memory asynchronously would let the kernel reclaim physical memory lazily and batch or eliminate remote TLB shootdowns.

6. 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 scalable commutativity rule, developers can automate much of this reasoning. This section presents a systematic, test-driven approach to applying the rule to real implementations embodied in a tool named COMMUTER, whose components are shown in Figure 8.

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.

Developers 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 that 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, written in a symbolic variant of Python, and outputs *commutativity conditions*: expressions in terms of arguments and states 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 nondeterminism (like creat's ability to choose any free inode) as underconstrained 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 9. To begin with, we can think of this

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

  # Generate all (non-empty) prefixes of all reorderings of ops
  for prefixlen in range(1, 1 + len(ops)):
    for hist in permutations(ops, prefixlen):
      # Execute next operation in this history
      past, op = hist[:-1], hist[-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(hist) not in states:
        states[frozenset(hist)] = s
      elif s != states[frozenset(hist)]:
        return False
  return True
```

Fig. 9. The SIM commutativity test algorithm in Python. 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.

test in concrete (nonsymbolic) 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 $H = X \,||\, Y$ when, given any reordering Y' of Y and any action sequence Z,

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

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 Section 7, only a few types need special handling beyond what ANALYZER's high-level data types already provide.

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

```
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
```

Fig. 10. A simplified version of our rename model.

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

Figure 10 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 11, 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 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).

```
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
```

Fig. 11. The SIM commutativity test algorithm specialized to two operations.

—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 earlier except the self-rename case depends on state and would not have been revealed by the traditional algebraic definition of commutativity.

Figure 12 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 12 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. 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 noncommutative (Figure 12 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

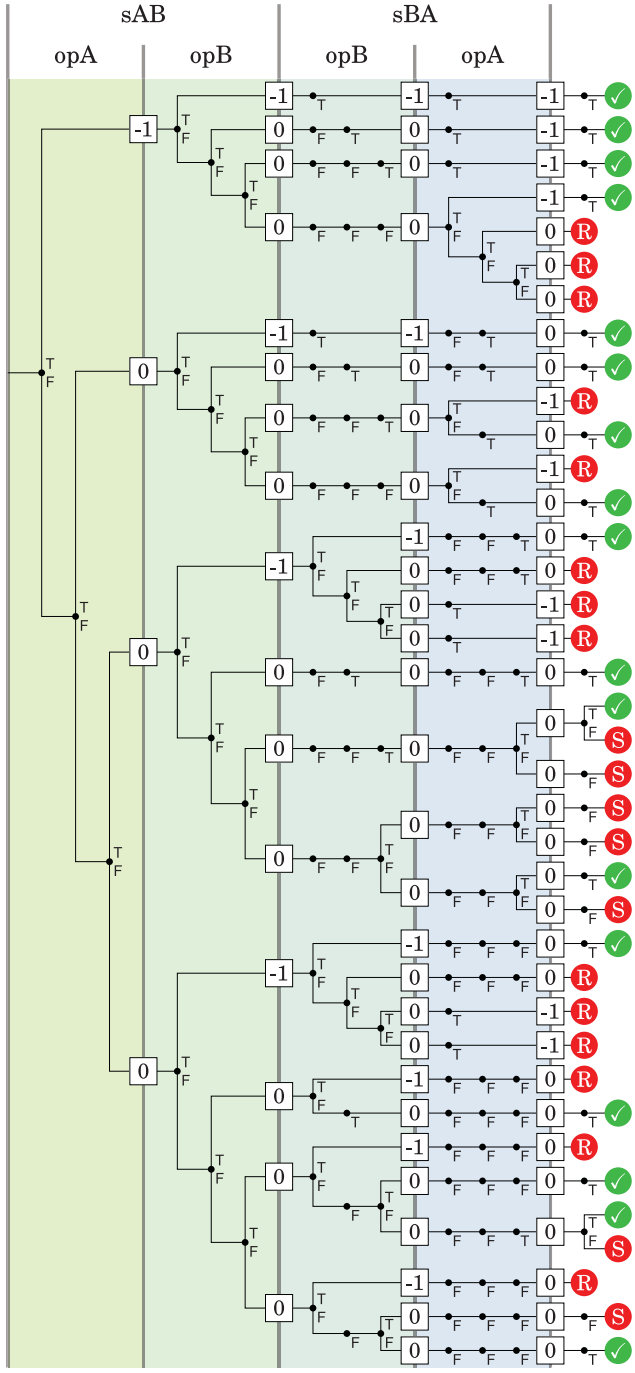


Fig. 12. Symbolic execution tree of commutes2 (Figure 11) for rename/rename. Each node represents a conditional branch on a symbolic value. The terminals at the right indicate whether each path constraint yields a commutative execution of the two operations (✓), or, if not, whether it diverged on return values (R) or final state (S).

```
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");
}
```

Fig. 13. An example test case for two rename calls generated by TESTGEN for the model in Figure 10.

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 10. 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 13. 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.

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, and so forth. TESTGEN approximates conflict coverage by

concolically executing the model-specific test code generator 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 Section 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 13 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 [Bellard et al. 2011]. 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 debug information [DWARF Debugging Information Format Committee 2010] 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 4,387 lines of Python code, including the symbolic execution engine, which uses the Z3 SMT solver [de Moura and Bjørner 2008] 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.

7. 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 section 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 section, 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 operations conflict for any offsets within a page. 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 693 and 835 lines of Python code, respectively. Figure 10 showed a part of our model, and Figure 13 gave an example test case generated by COMMUTER. We verified that all test cases return the expected results on both Linux and sv6.

7.2. Linux Conflict Freedom

To evaluate the scalability of existing file and virtual memory systems, we used MTRACE to check the previous test cases against Linux kernel version 3.8. Linux developers have invested significant effort in making the file system scale [Boyd-Wickizer et al. 2010], 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 [Corbet 2012]. We evaluated the ramfs file system because it is effectively a minimal wrapper for the Linux inode, directory, and file caches, which underlie all Linux file systems. Linux's virtual memory system, in contrast, involves process-wide locks that are known to limit its scalability and impact real applications [Boyd-Wickizer et al. 2010; Clements et al. 2012; Tene et al. 2011].

Figure 14 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 and thus could be conflict free. 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 [Boyd-Wickizer et al.

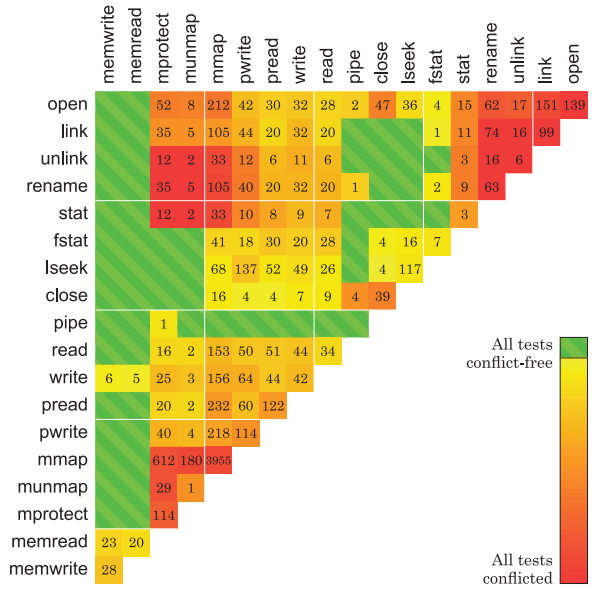


Fig. 14. Conflict freedom of commutative system call pairs in Linux, showing the fraction and absolute number of test cases generated by COMMUTER that are *not* conflict free for each system call pair. One example test case was shown in Figure 13.

2010; Clements et al. 2012, 2013a], which demonstrated these problems in the context of several applications.

Figure 14 also reveals many previously unknown bottlenecks that may be triggered by future workloads or hardware.

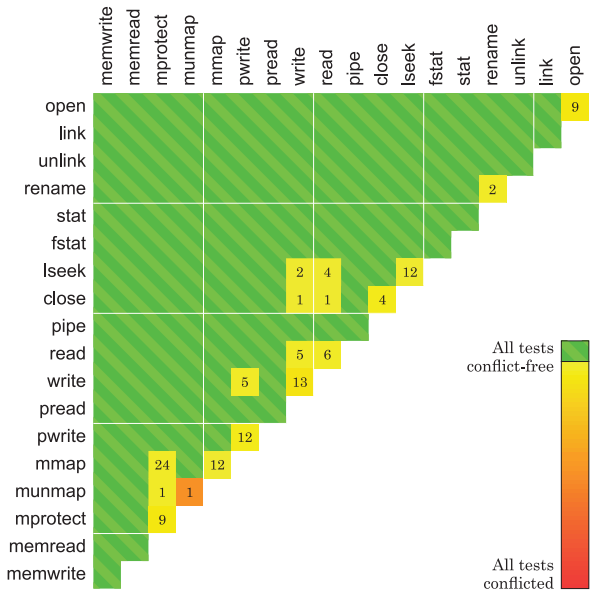
The next section shows how these current and future bottlenecks can be eliminated in a practical implementation of POSIX.

8. 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 [Clements et al. 2013a] for sv6, our research kernel based on xv6 [Cox et al. 2011]. Both these systems make use of Refcache [Clements et al. 2013a], a scalable reference counter. Although it is in principle possible to make the same changes in Linux, 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 15 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



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

Fig. 15. Conflict freedom of commutative system call pairs in sv6.

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 tradeoffs.

The rest of this section describes the high-level design of ScaleFS, RadixVM, and Refcache; discusses operations that are difficult to make conflict free without sacrificing other practical concerns; and briefly describes the implementation of sv6.

8.1. ScaleFS: Conflict-Free File System Operations

ScaleFS implements 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. 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 [McKenney et al. 2002], and seqlocks [Lameter 2005, §6]. It also employs Refcache to track internal resources and inode link counts. ScaleFS is 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.

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, reverifies 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 recomputes 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.

8.2. RadixVM: Conflict-Free Virtual Memory Operations

The POSIX virtual memory interface is rife with commutativity. VM operations from different processes (and hence address spaces) trivially commute, but VM operations from the same process also often commute—in particular, operations on *nonoverlapping* regions of the address space commute. Many multithreaded applications exercise exactly this 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 [Evans 2006; Ghemawat 2007; Liu and Chen 2012]. But 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 [Boyd-Wickizer et al. 2010; Clements et al. 2012].

The RadixVM virtual memory system makes commutative operations on nonoverlapping address space regions almost always conflict free [Clements et al. 2013a]. 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, was challenging, and our experience with developing RadixVM led to our search for a more general understanding of commutativity and scalability in multicore systems. Again, several design patterns arose.

Prefer data structures that localize conflicts to key regions. Most operating system kernels represent address spaces as balanced trees of mapped memory regions. Unfortunately, balancing operations on red-black trees (the representation in Linux), splay trees (FreeBSD), and AVL trees (Solaris and Windows) have nonlocal effects: an operation on one branch can cause conflicts over much of the tree. As a result, these kernels use coarse-grained locking. Lock-free skip lists [Herlihy and Shavit 2008] and other lock-free balanced lookup data structures avoid locking but still induce conflicts on operations that should commute: inserts and removes make nonlocal memory writes to preserve balance (or an equivalent), and those writes conflict with commutative lookups. The effect of these conflicts on performance can be dramatic. RadixVM, instead, adopts a data structure that is organized around the structure of its keys (memory addresses), namely, a multilevel compressed radix tree. Thanks to this structure, changes in one region of the address space almost always modify different cache lines than those accessed for changes in another region.

Precise information tracking reduces conflicts. A major impediment to scaling address space changes is the need to explicitly invalidate cached TLB (translation lookaside buffer) entries on other cores. POSIX specifies that when one core changes an address mapping, the new mapping (or, in the case of munmap, the new lack of mapping) becomes instantly visible to all threads. This in turn requires explicitly invalidating any cached mappings in the TLBs for those threads' cores. Typical Unix VM systems conservatively broadcast TLB shootdown interrupts to *all* cores in the process [Black et al. 1989], inducing cache line conflicts and limiting scalability. RadixVM addresses this problem by precisely tracking the set of CPUs that have accessed each page mapping. On the x86 architecture, RadixVM achieves this using *per-core page tables*. Each core maintains its own page table for the process, and that page table is filled on demand with just those memory regions actually accessed by threads on that core. Per-core page tables require more memory than standard shared page tables; however,

when an application thread allocates, accesses, and frees memory on one core, with no other threads accessing the same memory region, RadixVM will perform no TLB shootdowns.

8.3. Refcache: Conflict-Free Reference Counting

Reference counting is critical to many OS functions. 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; ScaleFS must reference-count file descriptions, FD tables, and entries in various caches and scalably maintain other counts, such as for hard links. sv6 uses Refcache for these reference counts and others. Refcache implements space-efficient, lazy, scalable reference counting using per-core *reference delta caches* [Clements et al. 2013a]. Refcache users trade off latency in releasing reference-counted resources for scalability, which makes Refcache particularly suited to uses where increment and decrement operations often occur on the same core (e.g., the same thread that allocated a page also frees it). 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.

A common reference-counter interface consists of inc and dec functions, where inc returns nothing and dec returns whether the count is now zero. (When zero is reached, the caller must free the object.) In this interface, a sequence of inc and dec operations SIM-commutes if—and only if—the count does not reach zero in any reordering. In this case, the operations' results are the same in all reorderings, and since reordering does not change the final sum, no future operations can distinguish different orders. By the scalable commutativity rule, any such sequence has some conflict-free implementation. Sequences where zero is reached, however, might not. Refcache extends commutativity to such sequences by allowing some latency between when the count reaches zero and when the system *detects* that it's reached zero. Refcache's inc and dec both return nothing and hence always commute. A new review operation finds all objects whose reference counts recently reached zero; the Refcache user now frees objects after review, not after dec. 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 Refcache user can choose how often to invoke review. More frequent calls clean up freed memory more quickly but cause more conflicts. sv6 invokes review at 10ms intervals, which is several orders of magnitude longer than the time required by even the most expensive conflicts on current multicores.

By separating count manipulation and zero detection, Refcache can batch increments and decrements and reduce cache line conflicts. 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 prior scalable counters [Appavoo et al. 2007; Boyd-Wickizer et al. 2010; Corbet 2010; Ellen et al. 2007], but unlike them, it requires space proportional to the *sum* of the number of counters and the number of cores, rather than the product. Space overhead is particularly important for RadixVM, which must reference count every physical page; at large core counts, other counters could use more than half of physical memory simply to track page references.

8.4. Difficult-to-Scale Cases

As Figure 15 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 to 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 tradeoff.

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.

8.5. sv6 Implementation

sv6 is derived from xv6 [Cox et al. 2011] but ports xv6 to the x86-64 architecture and the C++ language and adds support for hardware features necessary to run on modern multicores such as NUMA, the x2APIC, ACPI, and PCI Express. sv6 implements key POSIX file system and virtual memory system interfaces (those shown in Figure 15 plus others), as well as interfaces for process management (fork, exec, wait, etc.), threads (pthread_*), and sockets (connect, bind, send, recv, etc.).

In addition to the operations analyzed in this section, sv6 also scalably implements many of the modified, more broadly commutative POSIX APIs from Section 5.

All told, sv6 totals 51,732 lines of code, including kernel proper, device drivers, and user space and library code (but not including the lwIP [Dunkels et al. 2012] and ACPICA [Intel 2012] libraries). sv6 is implemented primarily in C++11 [ISO 2011] and makes extensive use of the C++11 memory model and language-based atomics as well as object-oriented programming, generic programming, and automatic resource management.

9. PERFORMANCE EVALUATION

Given that nearly all commutative ScaleFS and RadixVM operations are conflict free, applications built on these operations should in principle scale perfectly. This section confirms this, completing a pyramid whose foundations were set in Section 3 when we demonstrated that conflict-free memory accesses scale in most circumstances on real hardware. This section 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 noncommutative 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 sublinear) scalability of noncommutative operations, and acceptable memory use?

To answer these questions, we use sv6. We focus on the file system; benchmark results for the virtual memory system appeared in previous work [Clements et al. 2013a].

9.1. Experimental Setup

We ran experiments on an 80-core machine with eight 2.4GHz 10-core Intel E7-8870 chips and 256GB of RAM (detailed earlier in Figure 2). When varying the number of cores, benchmarks enable whole sockets at a time, so each 30MB 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 30MB 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 that 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. Microbenchmarks

Each file system benchmark has two variants, one that uses standard, noncommutative POSIX APIs and another that accomplishes the same task using the modified, more broadly commutative APIs from Section 5. By benchmarking the standard interfaces against their commutative counterparts, we can isolate the cost of noncommutativity 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 noncommutative 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 Section 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 [Howells 2010]).

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 [Clements et al. 2013a] 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 16(a) shows the 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

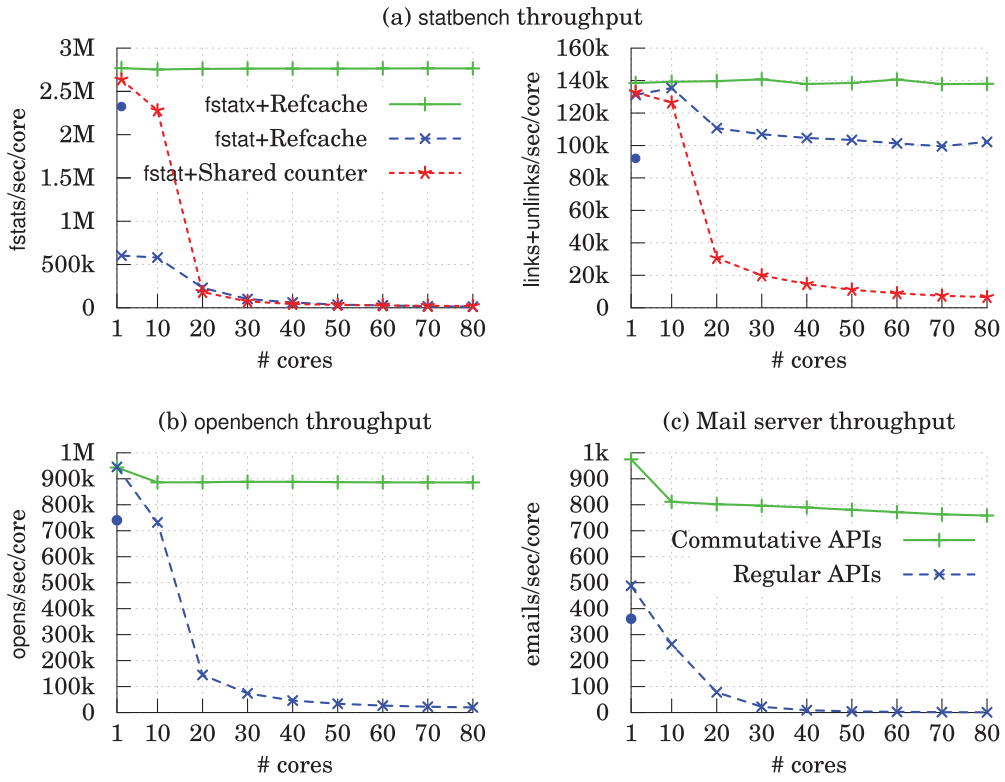


Fig. 16. Microbenchmark and mail server throughput in operations per second per core with varying core counts on sv6. The blue dots indicate baseline Linux performance for comparison.

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 noncommutative 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 noncommutative operation can have severely limited scalability.

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 16(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, noncommutative 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. 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 [Bernstein 2007]. mail-enqueue takes a mailbox name and a single message on the standard input, 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 the standard input 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 Section 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 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 16(c) 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 noncommutative operations, the results are quite similar. Noncommutative 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 one socket (10 cores) to eight sockets.

9.4. Discussion

This section completes our trajectory from theory to practice. Our benchmark results demonstrated that commutative operations can be implemented to scale, confirming the scalable commutativity rule for complex interfaces and real hardware and validating our sv6 design and design methodologies. Furthermore, all of this can be accomplished at little cost to sequential performance. We have also demonstrated the importance of commutativity and conflict freedom by showing that even a single contended cache line in a complex operation can severely limit scalability.

10. FUTURE DIRECTIONS

This section takes a step back and reviews some of the questions we have only begun to explore.

10.1. 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 that may expand the reach of the rule and enable scalable implementations of broader classes of interfaces: *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 and the proof of the reverse rule no longer hold.

Recent work by Boyd-Wickizer [2014] developed a technique for scalable implementations called OpLog based on using synchronized timestamp counters. OpLog does not work well for all interfaces: although logs can be collected in a conflict-free way, resolving logs may require sequential processing, which can limit scalability and performance. The characterization of interfaces amenable to OpLog implementation is worth exploring.

10.2. 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 Section 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.

10.3. Not Everything Can Commute

We have advocated fixing scalability bottlenecks by making interfaces as commutative as possible. Unfortunately, some interfaces are set in stone, and others, such as synchronization interfaces, are fundamentally noncommutative. 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 article addressed this problem only in ad hoc ways. Most notably, resource reclamation is inherently non commutative but, of course, unavoidable in any real system. However, resource reclamation is often not time sensitive. Hence, ScaleFS makes extensive use of Refcache [Clements et al. 2013a], which focuses all of the noncommutativity and nonscalability inherent in resource reclamation into a periodic, non-critical-path operation. This periodic operation allows Refcache to batch and eliminate many conflicts and amortize the cost of these conflicts. However, whether there is a general interface-driven approach to the scalability of noncommutative operations remains an open question.

10.4. Broad Conflict Freedom

As evidenced by sv6 in Section 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 tradeoffs 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.

11. CONCLUSION

We are in the midst of a sea change in software performance, as systems from top-tier servers to embedded devices turn to parallelism to maintain a performance edge. This article 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.

APPENDIX

A. PROOF OF CONVERSE RULE

We begin by showing that the order of two conflict-free machine invocations can be exchanged without affecting the responses or conflict freedom of the corresponding steps or the machine state following the two steps.

LEMMA A.1. *Consider an initial state s_0 and two invocations i_1 and i_2 on different threads. If*

$$m(s_0, i_1) = \langle s_1, r_1, \mathbf{a}_1 \rangle \quad \textit{and} \quad m(s_1, i_2) = \langle s_2, r_2, \mathbf{a}_2 \rangle$$

and $m(s_0, i_1)$ and $m(s_1, i_2)$ are conflict free, then for some s_1',

$$m(s_0, i_2) = \langle s_1', r_2, \mathbf{a}_2 \rangle \quad \textit{and} \quad m(s_1', i_1) = \langle s_2, r_1, \mathbf{a}_1 \rangle$$

and $m(s_0, i_2)$ and $m(s_1', i_1)$ are conflict free. That is, i_1 and i_2 can be reordered without changing their responses, the final state, or their conflict freedom.

Intuitively, this makes sense: since i_1 and i_2 are conflict free, the execution of each step is unaffected by the other. Proving this is largely an exercise in transforming s_0, s_1, and s_2 in the appropriate ways.

PROOF. Let $\mathbf{w}_1$ and $\mathbf{w}_2$ be the sets of state components written by $m(s_0, i_1)$ and $m(s_1, i_2)$, respectively. Because these steps are conflict free and on different threads, $\mathbf{w}_1 \cap \mathbf{a}_2 = \mathbf{w}_2 \cap \mathbf{a}_1 = \varnothing$. This implies $\mathbf{w}_1 \subseteq \mathbf{a}_2{}^c$ and $\mathbf{w}_2 \subseteq \mathbf{a}_1{}^c$ (where $\mathbf{s}^c$ denotes the set complement of $\mathbf{s}$). Furthermore, because of the write inclusion restriction, $\mathbf{w}_1 \subseteq \mathbf{a}_1$ and $\mathbf{w}_2 \subseteq \mathbf{a}_2$.

Consider the step $m(s_0, i_2)$, which applies i_2 to state s_0 instead of s_1. We can write $s_0 = s_1\{\mathbf{w}_1 \leftarrow s_0.\mathbf{w}_1\}$; or, since $\mathbf{w}_1 \subseteq \mathbf{a}_2{}^c$, $s_0 = s_1\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\}$. Since $m(s_1, i_2) = \langle s_2, r_2, \mathbf{a}_2 \rangle$, we can apply the access set restriction to see that

$$m(s_0, i_2) = m(s_1\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\}, i_2) = \langle s_2\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\}, r_2, \mathbf{a}_2 \rangle.$$

So the response and access set are unchanged from the original order.

Let s_1' be the new state. We expand it as follows:

$$\begin{aligned}
s_1' &= s_2\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\} && \text{(from above)} \\
&= s_0\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\}\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\} && \text{(def. of } s_2\text{)} \\
&= s_0\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\}\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\} && \text{(since } \mathbf{a}_2{}^c \cap \mathbf{w}_2 = \varnothing\text{)} \\
&= s_0\{\mathbf{a}_2{}^c \leftarrow s_0.\mathbf{a}_2{}^c\}\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\} && \text{(since } \mathbf{w}_1 \subseteq \mathbf{a}_2{}^c\text{)} \\
&= s_0\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\}. && \text{(self-assignment)}
\end{aligned}$$

Now consider the step $m(s_1', i_1)$, which applies i_1 to s_1'. Since $\mathbf{w}_2 \subseteq \mathbf{a}_1{}^c$, we can write $s_1' = s_0\{\mathbf{a}_1{}^c \leftarrow s_1'.\mathbf{a}_1{}^c\}$, and, again using the access set restriction, conclude that

$$m(s_1', i_1) = m(s_0\{\mathbf{a}_1{}^c \leftarrow s_1'.\mathbf{a}_1{}^c\}, i_1) = \langle s_1\{\mathbf{a}_1{}^c \leftarrow s_1'.\mathbf{a}_1{}^c\}, r_1, \mathbf{a}_1 \rangle.$$

Again, the response and access set are unchanged from the original order.

Let s_2' be the resulting state $s_1\{\mathbf{a}_1{}^c \leftarrow s_1'.\mathbf{a}_1{}^c\}$. Then

$$\begin{aligned}
s_2' &= s_0\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{a}_1{}^c \leftarrow s_1'.\mathbf{a}_1{}^c\} \\
&= s_0\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{w}_2 \leftarrow s_1'.\mathbf{w}_2\}\{(\mathbf{a}_1{}^c - \mathbf{w}_2) \leftarrow s_1'.(\mathbf{a}_1{}^c - \mathbf{w}_2)\} \\
&= s_0\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\}\{(\mathbf{a}_1{}^c - \mathbf{w}_2) \leftarrow s_0.(\mathbf{a}_1{}^c - \mathbf{w}_2)\}.
\end{aligned}$$

Since the three assignment groups affect disjoint components, we may reorder them. That shows

$$\begin{aligned}
s_2' &= s_0\{(\mathbf{a}_1{}^c - \mathbf{w}_2) \leftarrow s_0.(\mathbf{a}_1{}^c - \mathbf{w}_2)\}\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\} \\
&= s_0\{\mathbf{w}_1 \leftarrow s_1.\mathbf{w}_1\}\{\mathbf{w}_2 \leftarrow s_2.\mathbf{w}_2\} = s_2 :
\end{aligned}$$

the new final state equals the original final state. □

As a corollary to Lemma A.1, we can reorder not only adjacent invocations but also larger sequences of conflict free invocations.

COROLLARY A.2. *Given a history $H = X||Y||Z$, an implementation m that can generate H whose steps in Y are conflict free, and any reordering Y' of Y, m can generate $X||Y'||Z$ and m's steps in Y' are conflict free.*

PROOF. Let $W = i_1, \ldots, i_n$ be a witness of m generating H where m's steps corresponding to Y are conflict free. Through repeated applications of Lemma A.1 to machine invocations in W, we can construct a witness W' of m generating $X || Y' || Z$. By Lemma A.1, m's steps in Y' will be conflict free. □

Now that we have several facts about conflict freedom, we can begin to consider commutativity. As the definition of SIM commutativity builds on the definition of SI commutativity, we begin by examining SI commutativity.

Consider a specification $\mathscr{S}$ and a history $H = X || Y \in \mathscr{S}$. Let m be a correct implementation of $\mathscr{S}$ that can generate H. Let $\mathscr{S}|m$ be the subset of $\mathscr{S}$ that m can generate.

LEMMA A.3. *If m's steps in Y are conflict free, then Y SI-commutes in $X || Y$ with respect to $\mathscr{S}|m$.*

PROOF. Let Y' be any reordering of Y and Z be any action sequence. We want to show that $X || Y || Z \in \mathscr{S}|m$ if and only if $X || Y' || Z \in \mathscr{S}|m$. Since $H \in \mathscr{S}|m$ if and only if m can generate H, we can equivalently show that m can generate $X || Y || Z$ if and only if m can generate $X || Y' || Z$; but this follows directly from Lemma A.2. □

Finally, we can extend this result to SIM commutativity.

THEOREM A.4. *If m's steps in Y are conflict free, then Y SIM-commutes in $X || Y$ with respect to $\mathscr{S}|m$.*

PROOF. Let Y' be any reordering of Y. By Corollary A.2, m can generate $X || Y'$ and m is conflict free in Y'. Let P be any prefix of Y'. Because m is a step function, m can also generate $X || P$ and, by extension, is conflict free in P. By Lemma A.3, P SI-commutes in $X || P$ and, therefore, Y SIM-commutes in $X || Y$. □

COROLLARY A.5. *If Y does not SIM-commute in $X || Y$ with respect to $\mathscr{S}|m$, then m's steps in Y must conflict.*

REFERENCES

Advanced Micro Devices. 2012. *AMD64 Architecture Programmer's Manual*. Vol. 2. Advanced Micro Devices.

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. 2007. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems* 25, 3 (Aug. 2007), 1–52.

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

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

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.

Fabrice Bellard and others. 2011. QEMU. Retrieved August 1, 2014, from http://www.qemu.org/.

Daniel J. Bernstein. 2007. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the ACM Workshop on Computer Security Architecture*.

Philip A. Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *Computer Surveys* 13, 2 (June 1981), 185–221.

David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. 1989. Translation lookaside buffer consistency: A software approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*. 113–122.

Silas Boyd-Wickizer. 2014. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. Ph.D. Dissertation. Massachusetts Institute of Technology.

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. 2008. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*.

Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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'08)*.

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

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

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

Austin T. Clements. 2014. *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Concurrent address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013a. RadixVM: Scalable address spaces for multithreaded applications (revised 2014-08-05). In *Proceedings of the ACM EuroSys Conference*.

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2013b. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

Jonathan Corbet. 2010. The Search for Fast, Scalable Counters. Retrieved August 1, 2014, from http://lwn.net/Articles/170003/.

Jonathan Corbet. 2012. Dcache scalability and RCU-walk. (April 23, 2012). Retrieved August 1, 2014, from http://lwn.net/Articles/419811/.

Russ Cox, M. Frans Kaashoek, and Robert T. Morris. 2011. Xv6, a simple Unix-like teaching operating system. (February 2011). Retrieved August 1, 2014, from http://pdos.csail.mit.edu/6.828/xv6/.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

John DeTreville. 1990. *Experience with Concurrent Garbage Collectors for Modula-2+*. Technical Report 64. DEC Systems Research Center.

Adam Dunkels and others. 2012. Lightweight IP. Retrieved August 1, 2014, from http://savannah.nongnu.org/projects/lwip/.

DWARF Debugging Information Format Committee. 2010. DWARF debugging information format, version 4. Retrieved from http://www.dwarfstd.org/doc/DWARF4.pdf.

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

Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*. Ottawa, Canada.

Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. 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'99)*. 87–100.

Sanjay Ghemawat. 2007. TCMalloc: Thread-caching Malloc. Retrieved from http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

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

J. R. Goodman and H. H. J. Hum. 2009. *MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects*. Technical Report. University of Auckland and Intel.

Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

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

David Howells. 2010. Extended File Stat Functions, Linux Patch. Retrieved August 1, 2014, from https://lkml.org/lkml/2010/7/14/539.

IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group. 2013. The Open Group base specifications issue 7, 2013 edition (POSIX.1-2008/Cor 1-2013). Retrieved from http://pubs.opengroup.org/onlinepubs/9699919799/.

Intel. 2012. The ACPI Component Architecture Project. Retrieved August 1, 2014, from http://www.acpica.org/.

Intel. 2013. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3. Intel Corporation.

ISO. 2011. *ISO/IEC 14882:2011(E): Information technology – Programming languages – C++*. Geneva, Switzerland.

Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*.

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

Christoph Lameter. 2005. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*. Retrieved August 1, 2014, from http://lameter.com/gelato2005.pdf.

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

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

Paul E. McKenney. 2011. Concurrent Code and Expensive Instructions. Retrieved August 1, 2014, from https://lwn.net/Articles/423994/.

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

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

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

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

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

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

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

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

Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. 2011b. Convergent and commutative replicated data types. *Bulletin of the EATCS* 104 (June 2011), 67–88.

Guy L. Steele, Jr. 1990. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*.

Super Micro Computer. 2012. X8OBN-F manual. Retrieved from http://www.supermicro.com/manuals/motherboard/7500/X8OBN-F.pdf.

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

Tyan Computer Corporation. 2006a. M4985 manual. (2006).

Tyan Computer Corporation. 2006b. S4985G3NR manual. (2006).

R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. 1995. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing* 9, 1–2 (March 1995), 105–134.

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

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

Received October 2014; accepted October 2014