

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

By Austin T. Clements, M. Frans Kaashoek, Eddie Kohler, Robert T. Morris, and Nickolai Zeldovich

Abstract

Developing software that scales on multicore processors is an inexact science dominated by guesswork, measurement, and expensive cycles of redesign and reimplementa-tion. Current approaches are workload-driven and, hence, can reveal scalability bottlenecks only for known workloads and available software and hardware. This paper introduces an *interface-driven* approach to building scalable software. This approach is based on the *scalable commutativity rule*, which, informally stated, says that whenever interface operations commute, they can be implemented in a way that scales. We formalize this rule and prove it correct for any machine on which conflict-free operations scale, such as current cache-coherent multicore machines. The rule also enables a better design process for scalable software: programmers can now reason about scalability from the earliest stages of interface definition through software design, implementation, and evaluation.

1. INTRODUCTION

Until the mid-2000s, continuously rising CPU clock speeds made sequential software perform faster with each new hardware generation. But higher clock speeds require more power and generate more heat, and around 2005 clock speeds reached the thermal dissipation limits of a few square centimeters of silicon. CPU architects have not significantly increased clock speeds since, but the number of transistors that can be placed on a chip has continued to rise. Architects now increase parallelism by putting more CPU cores on each chip. *Total* cycles per second continues to grow exponentially, but software must *scale*—must take advantage of parallel CPU resources—to benefit from this growth.

Unfortunately, scaling is still an untamed problem. Even with careful engineering, software rarely achieves the holy grail of linear scalability, where doubling hardware parallelism doubles software performance.

Engineering scalable systems software is particularly challenging. Systems software, such as operating system kernels and databases, presents services to applications through well-defined interfaces. Designers rarely know ahead of time how applications will use these interfaces, and thus often cannot predict what bottlenecks to multicore scalability will arise. Furthermore, scaling bottlenecks may be a consequence of the definition of the interface itself; such problems are particularly difficult to address once many applications depend on the interface.

Lack of a principled way to reason about scalability hampers all phases of systems software development: defining an interface, implementing the interface, and testing its scalability.

When defining an interface, developers lack a systematic way of deciding whether a given definition will allow for scalable implementations. Demonstrating a scalability bottleneck requires a complete implementation and a workload. By the time these are available, interface changes may no longer be practical: many applications may rely on the existing interface, and applications that trigger the bottleneck may not be important enough to warrant an interface change.

During design and implementation, developers lack a systematic way to spot situations in which perfect scalability is achievable. This makes it hard to design an implementation to be scalable from the start. Instead, over time developers must iteratively improve the software's parallel performance as specific workloads uncover bottlenecks, often re-implementing the software multiple times.

While testing, developers lack a systematic way of evaluating scalability. The state of the art for testing the scalability of multicore software is to choose a workload, plot performance at varying numbers of cores, and use tools such as differential profiling¹³ to identify scalability bottlenecks exhibited by that workload. Each new hardware model or workload, however, may expose new scalability bottlenecks.

This paper presents a new approach to designing scalable software that starts with the design of scalable software interfaces. This approach makes it possible to reason about multicore scalability 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 is 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

The original version of this paper was published in the *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

by another core). Since conflict-free operations empirically scale (as we argue in Section 2), 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 avoiding it yields a conflict-free implementation. Conflict-free operations can execute on different cores without mutual interference via inter-core cache coherence invalidation requests, allowing total throughput to scale linearly with the number of cores.

The intuitive version of the rule is useful in practice, but is not precise enough to reason formally about interfaces or to build automated tools that evaluate scalability. This paper formalizes the scalable commutativity rule and illustrates its usefulness in the context of several examples, and for entire operating systems that support POSIX, a complicated, widely used interface.

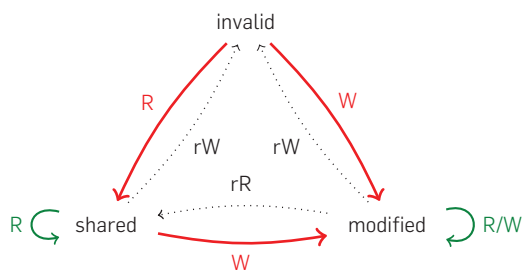
2. SCALABILITY AND CONFLICT-FREEDOM

The scalable commutativity rule assumes that code with conflict-free memory accesses—that is, code in which no cache line written by one core is read or written by any other core—is scalable. This section argues that, under reasonable assumptions, conflict-free operations do scale linearly on shared-memory multicore computers.

Multicores maintain a unified, globally consistent view of memory using MESI-like coherence protocols.¹⁵ MESI protocols coordinate ownership of cached memory at the level of cache lines. Their key invariant is that a line with a mutable copy in one core's cache cannot be present in any other caches: obtaining a mutable copy invalidates any other caches' immutable copies. This requires coordination, which affects scalability.

Figure 1 shows the basic state machine implemented by each cache for each cache line. This maintains the 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, and AMD's “owned” state—but these do not change the basic communication required to maintain cache coherence.

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



Roughly, a set of operations scales when maintaining coherence does not require ongoing communication. There are two memory access patterns that fit:

- Multiple cores reading and/or writing different cache lines. This scales because no further communication is required once each cache line is in the relevant core's cache, so further accesses can proceed independently of concurrent operations.
- 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 to N . Therefore, given a perfect implementation of MESI, conflict-free operations scale linearly.

Conflict-freedom is not a perfect predictor of scalability. Limited cache capacity and associativity cause caches to evict cache lines (later resulting in cache misses) even in the absence of coherence traffic, and a core's first access to a cache line will always 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; the resulting communication may contend with concurrent operations for interconnect resources or memory controller bandwidth. But applications with good cache behavior are unlikely to have such problems, while applications with poor cache behavior usually have sequential performance problems that outweigh scalability concerns. We have verified on real hardware that conflict-free operations actually do scale linearly under reasonable workload assumptions.⁶

3. THE SCALABLE COMMUTATIVITY RULE

Connections between commutativity and scalability have been explored before, especially in the context of operations on abstract data types.^{2, 16, 17, 19, 21, 22} For instance, commutative replicated data types¹⁹ are distributed objects whose operations always commute, allowing scalable, synchronization-free implementation. Abstract data type operations commute if they always produce the same result, regardless of order. For example, set member insertion commutes with itself, but not with removal: `set.insert(i)` and `set.insert(j)` produce the same results in either order, `set.insert(i)` and `set.remove(j)` has order-dependent results if $i = j$. But the systems interfaces we care about are richer, more granular, and more state- and context-dependent than typical data type operations. Consider the POSIX `creat` system call, which creates a file. The calls `creat("/d1/x")` and `creat("/d2/y")` seem to commute: their results are the same, regardless of the order they are applied. But if the disk is almost full and only one inode remains, then the calls

do not commute—the second creat call will fail. (Unless, that is, one or more of the files already exists, in which case the calls commute after all!) Special cases like this can dominate analyses that use a strong notion of commutativity. If commutative operations had to commute in *all* contexts, then only trivial systems operations could commute, and commutativity would not help us explore interface scalability.

Our work relies on a new definition of commutativity, called *SIM commutativity* (State-dependent, Interface-based, and Monotonic), that captures state- and context-dependence, and conditional commutativity, independent of any implementation. SIM commutativity lets us prove the scalable commutativity rule, which says that scalable implementations exist whenever operations commute. Even if an interface is commutative only in a restricted context, there exists an implementation that scales in that context.

The rest of this section explains this formalism, gives the rule precisely, and lays out some of its consequences for system designers.

3.1. Specifications

We represent specifications using *actions*, where an action is either an *invocation* (representing an operation call with arguments) or a *response* (representing the return value). Splitting each operation into an invocation and a response lets us model blocking interfaces and concurrent operations.¹¹ Each invocation is made by a specific thread and the corresponding response is returned to the same thread. We will write invocations as $\text{creat}("/x")_1$ and responses as OK_1 , where an overbar marks responses and subscript numbers are thread IDs.

A particular execution of a system is a *history* or *trace*, which is just a sequence of actions. For example,

$$H = [A_1, B_3, C_2, \bar{A}_1, \bar{C}_2, \bar{B}_3, D_1, \bar{D}_1, E_2, F_3, G_1, \bar{E}_2, \bar{G}_1, \bar{F}_3],$$

consists of seven invocations and seven corresponding responses across three different threads. In a *well-formed* history, each thread's actions alternate invocations and responses, so each thread has at most one outstanding invocation at any point. H above is well-formed; for instance, in the thread-restricted sub-history $H|1 = [A_1, \bar{A}_1, D_1, \bar{D}_1, G_1, \bar{G}_1]$, which selects 1's actions from H , invocations and responses alternate as expected.

A *specification* models an interface's behavior as a set of system histories—specifically, a prefix-closed set of well-formed histories. A system execution is “correct” according to the specification if its trace is included in the specification. For instance, if \mathcal{S} corresponded to the POSIX specification, then $[\text{getpid}_1, \bar{92}_1] \in \mathcal{S}$ (a process may have PID 92) but $[\text{getpid}_1, \text{ENOENT}_1] \notin \mathcal{S}$ (the `getpid()` system call may not return that error). A specification constrains both invocations and responses: $[\text{NtAddAtom}_1]$ is not in the POSIX specification because `NtAddAtom` is not a POSIX system call.

An *implementation* is an abstract machine that takes invocations and calculates responses. Our constructive proof of the scalable commutativity rule uses a class of machines on which conflict-freedom is defined⁶; a good analogy is a Turing-type machine with a random-access tape, where conflict-freedom follows if the machine's operations on behalf of different threads access disjoint portions of the tape. An implementation may “stutter-step,” taking multiple rounds to

finish calculating a response, and it does not have to generate responses in the order invocations were received.

An implementation M *exhibits* a history H if, when fed H 's invocations at the appropriate times, M can produce H 's responses (so that its external behavior equals H overall). An implementation M is *correct* for a specification \mathcal{S} if M 's responses always obey the specification. This means that every history exhibited by M is either in \mathcal{S} , or contains some invalid invocation.

3.2. Commutativity

SIM commutativity, which we define here, aims to capture state dependence at the interface level. State dependence means SIM commutativity must capture when operations commute in some states, even if those same operations do not commute in other states; however, we wish to capture this contextually, without reference to any particular implementation's state. To reason about *possible* implementations, we must 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.

Commutativity states that actions may be reordered without affecting eventual results. We say a history H' is a *reordering* of H when $H|t = H'|t$ for every thread t . This allows actions to be reordered across threads, but not within them. For example, if $H = [A_1, B_2, \bar{A}_1, C_1, \bar{B}_2, \bar{C}_1]$, then $[B_2, \bar{B}_2, A_1, \bar{A}_1, C_1, \bar{C}_1]$ is a reordering of H , but $[B_2, C_1, \bar{B}_2, \bar{C}_1, A_1, \bar{A}_1]$ is not, since it does not respect the order of actions in $H|1$.

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

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

This definition captures state dependence at the interface level. 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 cannot be distinguished by future operations, which is an interface-based way of saying that Y and Y' leave the system in the same state.

Unfortunately, SI commutativity is not sufficient to prove the scalable commutativity rule. To avoid certain degenerate cases, we must further strengthen the definition of commutativity to be *monotonic* (the M in SIM). An action sequence Y *SIM-commutes* in a history $H = X \parallel Y$ when for any *prefix* P of any reordering of Y (including $P = Y$), P SI-commutes in $X \parallel P$. 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 ,

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

Both SI commutativity and SIM commutativity capture state dependence and interface basis. Unlike SI commutativity, SIM commutativity excludes cases where the commutativity of a region changes depending on future operations. SIM commutativity is what we need to state and prove the scalable commutativity rule.

3.3. Rule

We can now formally state the scalable commutativity rule.

Assume an interface specification \mathcal{S} that has a correct implementation and a history $H = X \parallel Y$ exhibited by that implementation. Whenever Y SIM-commutes in H , there exists a correct implementation of \mathcal{S} whose steps in Y are conflict-free. Since, given reasonable workload assumptions, conflict-free operations empirically scale on modern multicore hardware, this implementation is scalable in Y .

Our proof of the rule constructs the scalable implementation from the correct reference implementation, and relies on our abstract machine definition and our definition of conflict-freedom.⁶

3.4. Example

Consider a reference counter interface with four operations. `reset(v)` sets the counter to a specific value v , `inc` and `dec` increment and decrement the counter and return its new value, and `isz` returns Z if the counter value is zero and NZ otherwise. The caller is expected to never decrement below zero, and once the counter reaches zero, the caller should not invoke `inc`.

Consider the counter history

$$H = [\text{reset}(2)_{3,}, \overline{OK}_3, \underbrace{\text{isz}_1, \overline{NZ}_1, \text{isz}_2, \overline{NZ}_2}_{H_1}, \underbrace{\text{dec}_3, \overline{1}_3, \text{dec}_4, \overline{0}_4}_{H_2}].$$

The region H_1 SIM-commutes in H , so the rule tells us that a correct implementation exists that is conflict-free for H_1 . In fact, this is already true of a simple shared-counter implementation: its `isz` reads the shared counter, but does not write it.

But H_2 does not SIM-commute in H , so no scalable implementation is implied—and, in fact, none is possible. The problem is that the caller can reason about order via the `dec` return values. Only a degenerate implementation, such as one that refused to respond to certain requests, could avoid tracking this order in a nonconflict-free way.

We can make `dec` commute by eliminating its return value. If we modify the specification so that `inc` and `dec` return nothing, then any region consisting exclusively of these operations commutes in any history. A version of H with this modified specification is

$$H' = [\text{reset}(2)_{3,}, \overline{OK}_3, \underbrace{\text{isz}_1, \overline{NZ}_1, \text{isz}_2, \overline{NZ}_2}_{H'_1}, \underbrace{\text{dec}_3, \overline{OK}_3, \text{dec}_4, \overline{OK}_4}_{H'_2}].$$

H'_2 , unlike H_2 , SIM-commutes, so there must be an implementation that is conflict-free there. Per-thread counters give us such an implementation: each `dec` can modify its local counter, while `isz` 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'_3 also SIM-commutes in H . However, the per-thread counter implementation is not conflict-free for H'_3 : `dec` will write one component of the state that is read and summed by

`isz`₁ and `isz`₂. But, again, there is a conflict-free implementation based on adding a Boolean “zeroness” snapshot as well as per-thread counters. `isz` simply returns this snapshot. When `dec` reduces a per-thread value to zero or below, it reads and sums all per-thread values and updates the snapshot if necessary.

3.5. Discussion

The rule pushes state and history dependence to an extreme: it makes a statement about a *single* history. 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. For instance, in our modified reference counter, H'_1 , H'_2 , and H'_3 all SIM-commute in H' , and we described a scalable implementation for each situation. However, H'_4 does not SIM-commute, even though it is a union of SIM-commutative pieces: if the two `dec` operations were reordered to the start of the region, then the `isz` operations would have to return different values. Any reasonable counter implementation must fail to scale in H'_4 , because `isz` must return different values depending on whether it ran before or after the `dec` invocations, and this requires communication between the cores that ran `dec` and `isz`. This can be proved using a converse of the rule: when a history contains a non-SIM-commutative region, no non-degenerate implementation can be scalable in that region.⁶ (The non-degeneracy condition eliminates implementations that, for example, never respond to any invocation, or always respond with an error return value.)

In our experience, real-world interface operations rarely demonstrate such mutually exclusive implementation choices. For example, the POSIX implementation in Section 5 scales quite broadly, with only a handful of cases that would require incompatible implementations.

4. DEFINING COMMUTATIVE INTERFACES

This section demonstrates more situations of interface-level reasoning enabled by the rule, using POSIX, the standard interface for Unix-like operating systems.

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

4.1. 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, including 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 `posix_spawn` call that addresses this problem by creating a process and loading an image directly (CreateProcess in Windows is similar). This is equivalent to `fork` followed by `exec`, eliminating the need for many of `fork`'s 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 non-commutative with operations on the same file that change any attribute returned by `stat` (such as `link`, `chmod`, `chown`, `write`, and even `read`). In practice, applications invoke `stat` for just one or two of the returned fields. An alternate API that gave applications control of which field or fields were returned would commute with more operations and enable a more scalable implementation of `stat`.⁶

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 of them.

4.2. Embrace specification nondeterminism

POSIX requires that the open system call returns the lowest-numbered unused file descriptor (FD) for the newly opened file. This 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. 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.

4.3. 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 multi-reader or multi-writer situations, and an alternate interface that does not enforce ordering would allow `send` and `recv` to commute as long as there is both enough free space and enough pending messages on the socket. This in turn would allow an implementation of Unix domain sockets to support scalable communication.

4.4. Release resources asynchronously

A closely related problem is that many POSIX operations have global effects that must be visible before the operation returns. This is generally good design for usable interfaces,

but for operations that release resources, this is often stricter than applications need and expensive to ensure. For example, writing to a pipe must deliver `SIGPIPE` immediately if there are no read FDs for that pipe, so pipe writes do not commute with the last close of a read FD. This requires aggressively tracking the number of read FDs; a relaxed specification that promised to eventually deliver the `SIGPIPE` would allow implementations to use more scalable read FD tracking. Similarly, `munmap` does not commute with memory reads or writes of the unmapped region from other threads, because other threads should not be able to write to the unmapped region after `munmap` returns (even though depending on this behavior usually indicates a bug). Indeed, enforcing this requires remote TLB shootdowns, which do not scale on today's hardware. 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.

As another example, to build a scalable reference counter, we start with the interface described in Section 3.4: `inc` and `dec` both return nothing and hence always commute. In place of the `isz` operation, we introduce a new `review` operation that finds all objects whose reference counts recently reached zero; this frees the developer from having to periodically call `isz` on their own. `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 user can choose how often to invoke `review`. More frequent calls clean up freed memory more quickly, but cause more conflicts. In our implementation of this scheme, called `Refcache`,⁷ `review` is called at 10 ms intervals, which is several orders of magnitude longer than the time required by even the most expensive conflicts on current multicores.

5. DESIGNING FOR CONFLICT-FREEDOM

To evaluate the implementation difficulty of the previous section's commutative interfaces, we built `sv6`, a research operating system that aims to provide a POSIX-like interface with as much scalability as is reasonably possible. `sv6` includes a `ramfs`-like in-memory file system called `ScaleFS`⁸ and a virtual memory system called `RadixVM`.⁷ In designing and implementing `sv6`, the rule told us that conflict-free implementations were possible in many cases, which forced us to come up with designs that achieved conflict-freedom. Without the rule, we would have given up too soon, deciding that some corner cases simply cannot be made to scale.

Problems in achieving conflict-freedom fell into two broad categories. On the one hand, we found situations where a single logical object (such as a reference counter, a pool of memory, or the scheduler queue) was accessed from many cores. Here, we typically used per-core data structures for the commutative parts of the API, and tried to ensure that noncommutative parts of the API (such as reconciling per-core reference counts, or stealing free memory pages or runnable threads from other cores when one core runs out) are invoked rarely and minimize cache-line movement when they are invoked. In some cases this required designing new algorithms, such as `Refcache`.

On the other hand, we also encountered situations that accessed logically distinct objects (e.g., files in a directory, or

pages in a virtual address space), but the data structures typically used to access these objects induced unnecessary conflicts. In particular, we discovered that many sophisticated data structures like red-black trees, splay trees, AVL trees, concurrent lock-free skip lists, etc., are a poor fit for the scalable commutativity rule. For example, balancing operations on binary trees have nonlocal effects: an operation on one branch can cause conflicts over much of the tree. Lock-free skip lists 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. A frequent solution involved switching to array-based data structures, which tend to naturally lend themselves to avoiding conflicts for commutative operations. For example, using an array to represent the open file descriptors for a process naturally provides conflict-freedom for operations on distinct file descriptors, because those operations access different addresses in the array.

Naive arrays are not great for situations where the key space is large. One solution for medium-size keys is to use a radix tree. For instance, we use radix trees in the sv6 virtual memory system, RadixVM,⁷ to implement the mapping from virtual addresses to the corresponding mapped objects. Since radix trees have no balancing operations, accesses to different addresses tend to not conflict. At the same time, simple compression techniques in the radix tree allow for a compact representation that's much more efficient than a single flat array.

For large or variable-sized keys, hash tables are a natural choice. For example, in the sv6 file system, we use a hash table to represent each directory. This means that concurrent operations on different file names in a single directory are unlikely to conflict (unless they map to the same hash table bucket). This is in contrast to traditional file system designs that take out a single lock to ensure that operations do not modify the same directory entry at the same time.

6. TESTING FOR CONFLICT-FREEDOM

Fully understanding the commutativity of a complex interface is tricky, and checking if an implementation achieves conflict-freedom whenever operations commute adds another dimension to an already difficult task. To help developers apply the rule during testing, we developed a tool called COMMUTER that automates this process.⁶ First, COMMUTER takes a symbolic model of an interface and computes precise conditions for when that interface's operations commute. Second, COMMUTER 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, COMMUTER checks whether a particular implementation is conflict-free for each test case. A developer can use these test cases to understand the commutative cases they should consider, to iteratively find and fix scalability bottlenecks in their code, and to perform regression tests to ensure scalability bugs do not creep into the implementation over time.

To illustrate how COMMUTER can help with testing for scalability, we wrote a symbolic model of the POSIX interface

covering file system and virtual memory operations, and checked the resulting test cases against Linux and the sv6 operating system. The results are shown in Figures 2 and 3, respectively. Each square represents a pair of system calls. The color of each square represents the fraction of test cases that fail to be conflict-free despite being commutative.

In the case of Linux, we can see that the kernel is already quite scalable: many pairs of system calls are conflict-free for all tests generated by COMMUTER. However, there are also many pairs that commute but are not conflict-free. This indicates that even a mature and reasonably scalable operating system implementation misses many cases that can be made

Figure 2. Conflict-freedom of commutative system call pairs in Linux 3.8, showing the fraction and absolute number of test cases generated by COMMUTER that are not conflict-free for each system call pair.

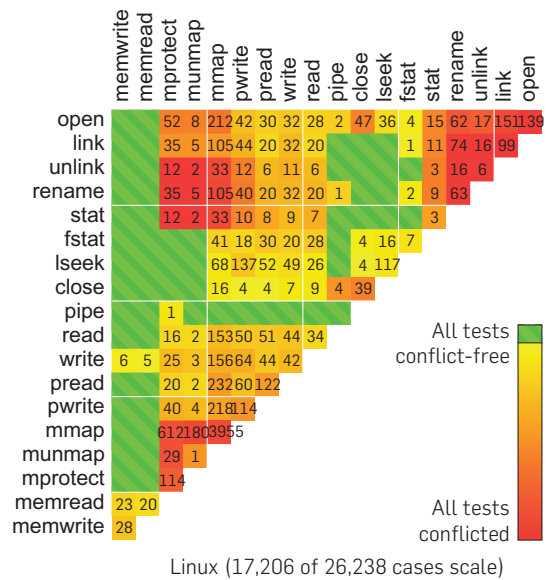
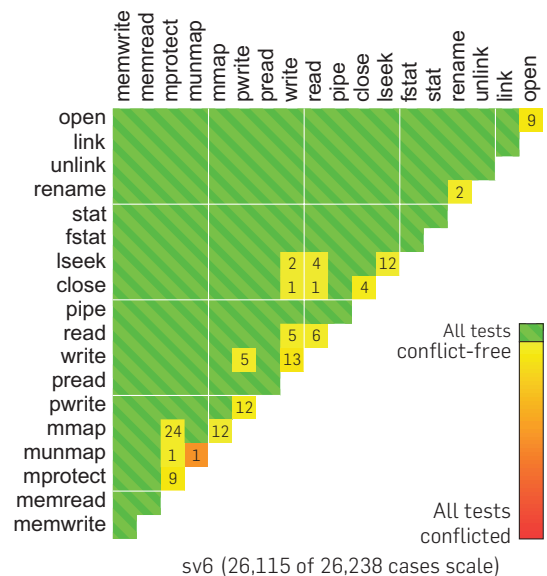


Figure 3. Conflict-freedom of commutative system call pairs in sv6.



to scale according to the commutativity rule. Some of these correspond to well-known scalability problems in Linux, such as concurrent operations on different file names in the same directory (which conflict on a per-directory lock) or concurrent operations on the virtual memory subsystem (which conflict on a per-address-space lock⁷). Others are new bottlenecks that may not have been previously discovered: COMMUTER has systematically discovered latent scalability problems.

In contrast with Linux, sv6 is conflict-free for nearly every commutative test case. In part this is due to our choice of data structures that are naturally conflict-free, as described in the previous section. While testing sv6, COMMUTER also discovered many commutative corner cases that we would not have thought of by ourselves. For example, consider the rename system call and the access system call, which can be used to check if a file exists. Suppose there are two existing files, a and b. COMMUTER discovered that rename(a, b) commutes with access(b), because in either order, rename succeeds and access indicates that b exists. However, our initial implementation was not conflict-free, because access used an internal function that not only checked if the file exists, but also looked up the file's inode. To make this case conflict-free, we introduced a separate function to check whether a file name exists in a directory hash table, without actually reading its corresponding value. During testing, we discovered a number of other common design patterns, such as deferring work whenever possible, preceding pessimism (i.e., writes to memory locations) with optimistic read-only checks, and avoiding reads unless absolutely necessary.

For a small number of commutative operations, sv6 is not conflict-free. The majority of these cases 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 reduced the performance of more common operations, so we explicitly chose to favor common-case performance over total scalability. Other 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 exception. However, the presence of the rule forced us to explicitly recognize, evaluate, and justify where we made such trade-offs.

7. DISCUSSION

One surprising aspect of the rule is that it allows us to reason about scalability without having to measure the throughput of a system as a function of the number of cores. Indeed, this paper contains no such graph. To be sure that our rule works in practice, we measured the scalability of a mail server running on sv6, using commutative system calls. The result was perfect scalability. On the one hand, this demonstrates the power of the rule: even for a previously untested hardware system and workload, we are able to confidently predict scalability. On the other hand, scalability is not the same as performance, and a perfectly scalable implementation could have lower total performance than an implementation tuned for efficiency on a small number of cores.

One potential way to expand the reach of the rule and create more opportunities for scalable implementations is to find ways in which *nonconflict-free* operations can scale. For example, while streaming computations are in general not linearly scalable because of interconnect and memory contention, we have 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. On one 80-core x86 system, 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. Mapping computations to this model might be difficult, and given the varying structures of multicore interconnects, the model itself may not generalize. 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.

8. RELATED WORK

This section briefly explains the relation between the scalable commutativity rule and previous work that explores thinking about scalability and commutativity. For a more in-depth discussion of related work that also covers scalable operating systems and testing approaches we refer the reader to Clements's thesis.⁶

8.1. Scalability

Israeli and Rappoport¹² 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.³ extend Israeli and Rappoport's definition to additionally require non-disjoint reads to scale. Our work builds on the assumption that memory systems behave this way and we have confirmed that real hardware closely approximates this behavior.⁶

Both the original disjoint-access parallelism paper and subsequent work¹⁸ explore the scalability of processes that have some amount of non-disjoint sharing, such as compare-and-swap instructions on a shared cache line or a shared lock. Our work takes a black-and-white view because we have found that, on real hardware, a single modified shared cache line can wreck scalability.

The Laws of Order² explore the relationship between the "strong noncommutativity" of an interface and whether any implementation of that interface must contain atomic and/or fence instructions 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.

It is well understood that cache-line contention can result in bad scalability. A clear example is the design of the MCS lock,¹⁴ which eliminates scalability collapse by avoiding contention for a particular cache line. Other good examples include scalable reference counters.^{1, 5, 9} The commutativity rule builds on this understanding and identifies when arbitrary interfaces can avoid conflicting memory accesses.

8.2. Commutativity


The use of commutativity to increase concurrency has been widely explored. Steele describes a parallel programming discipline in which all operations must be either causally related or commutative.²¹ His work approximates commutativity as conflict-freedom. We show that commutative operations always have a conflict-free implementation, implying that Steele's model is broadly applicable. Rinard and Diniz¹⁷ 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.¹⁶ 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: we show that any concurrent, commutative operations 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.⁴ Weihl extends this work to abstract data types by deriving lock conflict relations from operation commutativity.²² Transactional boosting applies similar techniques in the context of software transactional memory.¹⁰ Shapiro et al.^{19, 20} 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 focuses on the safety of executing commutative operations concurrently, not directly on scalability.

9. CONCLUSION

The scalable commutativity rule helps developers to reason about scalability in all three phases of software design: defining an interface, designing and implementing the software, and testing its scalability properties. The rule does not require the developer to have a target workload or a physical machine to reason about scalability. We hope that programmers will find the commutativity rule helpful in producing software that is scalable by design.

Acknowledgments

This research was supported by NSF awards SHF-964106 and CNS-1301934, by Quanta, and by Google. Eddie Kohler was partially supported by a Microsoft Research New Faculty Fellowship and a Sloan Research Fellowship. 

References

- Appavoo, J., da Silva, D., Krieger, O., Auslander, M., Ostrowski, M., Rosenberg, B., Waterland, A., Wisniewski, R.W., Xenidis, J., Stumm, M., Soares, L. Experience distributing objects in an SMP OS. *ACM Trans. Comput. Syst.* 25, 3 (August 2007).
- Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M.M., Vechev, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages* (Austin, TX, January 2011), 487–498.
- Attiya, H., Hillel, E., Milani, A. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Calgary, Canada, August 2009), 69–78.
- Bernstein, P.A., Goodman, N. Concurrency control in distributed database systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221.
- Boyd-Wickizer, S., Clements, A., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., Zeldovich, N. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, October 2010).
- Clements, A.T. The scalable commutativity rule: Designing scalable software for multicore processors. PhD thesis, Massachusetts Institute of Technology (June 2014).
- Clements, A.T., Kaashoek, M.F., Zeldovich, N. RadixVM: Scalable address spaces for multithreaded applications (revised 2014-08-05). In *Proceedings of the ACM EuroSys Conference* (Prague, Czech Republic, April 2013), 211–224.
- Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.* 32, 4 (January 2015), 10:1–10:47.
- Ellen, F., Lev, Y., Luchango, V., Moir, M. SNZI: Scalable nonzero indicators. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Portland, OR, August 2007), 13–22.
- Herlihy, M., Koskinen, E. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, February 2008), 207–216.
- Herlihy, M.P., Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
- Israeli, A., Rappoport, L. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Los Angeles, CA, August 1994), 151–160.
- McKenney, P.E. Differential profiling. *Softw. Pract. Exp.* 29, 3 (1999), 219–234.
- Mellor-Crummey, J.M., Scott, M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65.
- Papamarcos, M.S., Patel, J.H. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (Ann Arbor, MI, June 1984), 348–354.
- Prabhu, P., Ghosh, S., Zhang, Y., Johnson, N.P., August, D.I. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, CA, June 2011), 1–11.
- Rinard, M.C., Diniz, P.C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.* 19, 6 (November 1997), 942–991.
- Roy, A., Hand, S., Harris, T. Exploring the limits of disjoint access parallelism. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism* (Berkeley, CA, March 2009).
- Shapiro, M., Prego, N., Baquero, C., Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France, October 2011), 386–400.
- Shapiro, M., Prego, N., Baquero, C., Zawirski, M. Convergent and commutative replicated data types. *Bull. EATCS* 104 (June 2011), 67–88.
- Steele, G.L., Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages* (San Francisco, CA, January 1990), 218–231.
- Weihl, W.E. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.* 37, 12 (December 1988), 1488–1505.

Austin T. Clements, M. Frans Kaashoek, Robert T. Morris, and Nikolai Zeldovich ([aclements, kaashoek, rtm, zeldovich]@csail.mit.edu), MIT CSAIL, Cambridge, MA.

Eddie Kohler (kohler@seas.harvard.edu), Harvard University School of Engineering and Applied Sciences, Computer Science Area, Cambridge, MA.