The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler[†]
MIT CSAIL and [†]Harvard University

Abstract

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

To help developers apply the rule, a new tool named COMMUTER accepts high-level interface models and generates tests of operations that commute and hence could scale. Using these tests, COMMUTER can evaluate the scalability of an implementation. We apply COMMUTER to 18 POSIX calls and use the results to guide the implementation of a new research operating system kernel called sv6. Linux scales for 68% of the 13,664 tests generated by COMMUTER for these calls, and COMMUTER finds many problems that have been observed to limit application scalability. sv6 scales for 99% of the tests.

1 Introduction

The state of the art for evaluating the scalability of multicore software is to choose a workload, plot performance at varying numbers of cores, and use tools such as differential profiling [29] to identify scalability bottlenecks. This focuses developer effort on real issues, but has several drawbacks. Different workloads or higher core counts often exhibit new bottlenecks. It's unclear which bottlenecks are fundamental, so developers may give up without realizing that a scalable solution is possible. Finally, this process happens so late in the development process that design-level solutions such as improved interfaces may be impractical.

This paper presents a new approach to scalability that starts at a higher level: the software interface. This makes reasoning about scalability possible before an implementation exists and before the necessary hardware is available to measure the implementation's scalability. It can highlight inherent scalability problems, leading to alternate interface designs. And it sets a clear scaling target for the implementation of a scalable interface.

Scalability is often considered an implementation property, not an interface property, not least because what scales depends on hardware. However, if we assume a shared-memory multicore processor with caches kept coherent by a MESI-like protocol [33], general scalability arguments are possible. On such processors, a core can scalably read and write data it has cached exclusively, and scalably read data it has cached in shared mode. Writing a cache line that was last read or written by another core is not scalable, however, since the coherence protocol serializes ownership changes for each cache line, and because the shared interconnect may serialize unrelated transfers [7: §4.3].

We therefore say that a set of operations scales if their implementations have *conflict-free* memory accesses, where no core writes a cache line that was read or written by another core. When memory accesses are conflict-free, adding more cores will produce a linear increase in capacity. This is not a perfect model of the complex realities of modern hardware, but it is a good approximation.

At the core of our approach is this *scalable commutativity rule*: In any situation where several operations *commute*—meaning there's no way to distinguish their execution order using the interface—they have an implementation whose memory accesses are conflict-free during those operations. Or, more concisely, **whenever interface operations commute, they can be implemented in a way that scales.**

Connections between commutativity and concurrency are well established in the literature. Previous work, however, has focused on using commutativity to reason about the safety of executing operations concurrently (see §2). Our work is complementary: we use commutativity to reason about scalability.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522712

The commutativity rule makes intuitive sense: when operations commute, their results (return value and effect on system state) are independent of order. Hence, communication between commutative operations is unnecessary, and eliminating it yields conflict-free implementations. This intuitive version of the rule is useful in practice, but not precise enough to reason about formally. §3 formally defines the commutativity rule and proves the correctness of the formalized rule.

An important consequence of this presentation is a novel form of commutativity we call *SIM commutativity*. The usual definition of commutativity (e.g., for algebraic operations) is so stringent that it rarely applies to the complex, stateful interfaces common in systems software. SIM commutativity, in contrast, is *state-dependent* and *interface-based*, as well as *monotonic*. When operations commute in the context of a specific system state, specific operation arguments, and specific concurrent operations, we show that an implementation exists that is conflict-free *for that state and those arguments and concurrent operations*. This exposes many more opportunities to apply the rule to real interfaces—and thus discover scalable implementations—than a more conventional notion of commutativity would. Despite its logical 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.

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

Complex interfaces can make it difficult to spot and reason about all commutative cases even given the rule. To address this challenge, §5 introduces a tool named COMMUTER that automates this reasoning. COMMUTER takes an interface model in the form of a simplified, symbolic implementation, computes precise conditions under which sets of operations commute, and tests an implementation for conflict-freedom under these conditions. This tool can be integrated into the development process to drive initial design and implementation, to incrementally improve existing implementations, or to help developers understand the commutativity of an interface.

This paper demonstrates the value of the commutativity rule and COMMUTER in two ways. In §4, we explore the commutativity of POSIX and use this understanding both to suggest guidelines for designing interfaces whose operations commute and to propose specific modifications to POSIX that would allow for greater scalability.

In §6, we apply COMMUTER to a simplified model of 18 POSIX file system and virtual memory system calls. From this model, COMMUTER generates 13,664 tests of commutative system call pairs, all of which can be made conflict-free according to the rule. We use these tests to guide the implementation of a new research operating system kernel named sv6. sv6 has a novel virtual memory system (RadixVM [15]) and in-memory file system (named ScaleFS). COMMUTER determines that sv6 is conflict-free for 13,528 of the 13,664 tests, while Linux is conflict-free for 9,389 tests. Some of the commutative cases where Linux doesn't scale are important to applications, such as commutative mmaps and creating different files in a shared directory. §7 confirms that commutative conflict-free system calls translate to better application scalability on an 80-core machine.

2 Related work

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

2.1 Thinking about scalability

Israeli and Rappoport introduce the notion of disjoint-access-parallel memory systems [26]. Roughly, if a shared memory system is disjoint-access-parallel and a set of processes access disjoint memory locations, then those processes scale linearly. Like the commutativity rule, this is a conditional scalability guarantee: if the application uses shared memory in a particular way, then the shared memory implementation will scale. However, where disjoint-access parallelism is specialized to the memory system interface, our work encompasses any software interface. Attiya et al. extend Israeli and Rappoport's definition to additionally require conflict-free operations to scale [1]. Our work builds on the assumption that memory systems behave this way, and we indirectly confirm that real hardware closely approximates this behavior (§7).

Both the original disjoint-access parallelism paper and subsequent work, including the paper by Roy et al. [36], 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 (§7).

The Laws of Order [2] explore the relationship between an interface's *strong non-commutativity* and whether its implementation requires atomic instructions or fences (e.g., mfence on the x86) for correct concurrent execution. These instructions slow down execution by interfering with out-of-order execution, even if there are no memory access conflicts. The Laws of Order resemble the commutativity rule, but draw conclusions about sequential performance, rather than scalability. Paul McKenney explores the Laws of Order in the context of the Linux kernel, and points out that the Laws of Order may not apply if linearizability is not required [30].

It is well understood that cache-line contention can result in bad scalability. A clear example is the design of the MCS lock [32], which eliminates scalability collapse by avoiding contention for a particular cache line. Other good examples include scalable reference counters [16, 21]. 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 [12]. 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 paper identifies situations where POSIX permits or limits scalability and points out specific interface modifications that would permit greater implementation scalability.

Multikernels for multicore processors aim for scalability by avoiding shared data structures in the kernel [3, 43]. These systems implement shared abstractions using distributed systems techniques (such as name caches and state replication) on top of message passing. It should be possible to generalize the commutativity rule to distributed systems, and relate the interface exposed by a shared abstraction to its scalability, even if implemented using message passing.

The designers of the Corey operating system [8] argue for putting the application in control of managing the cost of sharing without providing a guideline for how applications should do so; the commutativity rule could be a helpful guideline for application developers.

2.3 Commutativity

The use of commutativity to increase concurrency has been widely explored. Steele describes a parallel programming discipline in which all operations must be either causally related or commutative [40]. His work approximates commutativity as conflict-freedom. Our work shows that commutative operations always have a conflict-free implementation, making Steele's model more broadly applicable. Rinard and Diniz describe how to exploit commutativity to automatically parallelize code [35]. They allow memory conflicts, but generate synchronization code to ensure atomicity of commutative operations. Similarly, Prabhu et al. describe how to automatically parallelize code using manual annotations rather than automatic commutativity analysis [34]. 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: 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 [6]. Weihl extends this work to abstract data types by deriving lock conflict relations from operation commutativity [42]. Transactional boosting applies similar techniques in the context of software transactional memory [23]. Shapiro et al. extend this to a distributed setting, leveraging commutative operations in the design of replicated data types that support updates during faults and network partitions [38, 39]. Like Rinard and Prabhu's work, the work in databases and its extensions focuses on the *safety* of executing commutative operations concurrently, not directly on scalability.

2.4 Test case generation

Prior work on concolic testing [22, 37] and symbolic execution [10, 11] generates test cases by symbolically executing a specific implementation. Our COMMUTER tool uses a combination of symbolic and concolic execution, but generates test cases for an *arbitrary* implementation based on a model of that implementation's interface. This resembles QuickCheck's [13] or Gast's [27] model-based testing, but uses symbolic techniques. Furthermore, while symbolic execution systems often avoid reasoning precisely about symbolic memory accesses (e.g., accessing a symbolic offset in an array), COMMUTER's test case generation aims to achieve *conflict* coverage (§5.2), which tests different access patterns when using symbolic addresses or indexes.

3 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. The formalism relies on SIM commutativity, whose generality broadens the rule's applicability to complex software interfaces. Our constructive proof of the commutativity rule also sheds some light on how real commutative implementations might be built, though the actual construction is not very practical.

3.1 Actions

Following earlier work [24], 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 result (a PID or a file descriptor). Invocations and responses are paired. Each invocation is made by a specific thread, and the corresponding response is returned to the same thread. An action thus comprises (1) an operation class (e.g., which system call is being invoked); (2) operation arguments (for invocations) or a return value (for responses); (3) the relevant thread; and (4) a tag for uniqueness. We'll write invocations as left half-circles A ("invoke A") and responses as right half-circles A ("respond A"), where the letters match invocations and their responses. Color and vertical offset differentiate threads: A and B are invocations on different threads.

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

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

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 formed by selecting t's actions from H alternates invocations and responses as one would want. In a well-formed history, each thread has at most one outstanding invocation at every point.

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

3.2 Commutativity

Commutativity should capture the idea that the order of a set of actions "doesn't matter." This happens when *later* actions can't tell which order actually occurred. The specification helps make this precise: a set of operations commutes in some context when the specification is indifferent to the execution order of that set. This means that any response valid for one order of the commutative set is valid for any order of the commutative set, and likewise any response invalid for one order is invalid for any order. But the right definition for commutativity is a little tricky, so we build it up in two steps.

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

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,

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

This definition captures the interface basis and state dependence we need. The action sequence X puts the system into the state we wish to consider; switching regions Y and Y' requires that the return values from Y be valid according to the specification regardless of the actions' order; and the presence of region Z in both histories requires that reorderings of actions in region Y are indistinguishable by future operations.

Unfortunately, SI commutativity doesn't suffice for the proof because it is *non-monotonic*. Given an action sequence $X \,||\, Y_1 \,||\, Y_2$, it is possible for $Y_1 \,||\, Y_2$ to SI-commute after region X even though Y_1 on its own does not. For example, consider a get/set interface and $Y = [$A $=$ set(1), A, B $=$ set(2), B, C $=$ set(2), C$]$. Y SI-commutes in any history (every order sets the underlying value to 2), but its prefix $Y_1 = [$A $=$ set(1), A, B $=$ set(2), B$]$ does not (some orders set the value to 1 and some to 2). Whether or not Y_1 will ultimately form part of a commutative region thus depends on *future* operations! This is usually incompatible with scalability: operations in Y_1 must "plan for the worst" by remembering their order in case execution diverges from Y_2.

A monotonic version of commutativity eliminates this problem. An action sequence Y *SIM-commutes* in a history $H = X \,||\, Y$ when for any *prefix* P of some reordering of Y (including $P = Y$), P SI-commutes in $X \,||\, P$. Like SI commutativity, SIM commutativity captures interface basis and state dependence; unlike SI commutativity, it is monotonic and, as we show below, suffices to prove the commutativity rule.

State dependence means that SIM commutativity captures operations that commute, and therefore can scale, in some states, but not others. This allows us to greatly expand the situations that commute, and that therefore can scale. For example, few OS system calls unconditionally commute in every state and history. (One that does is getpid(), since its result is constant over a process's lifetime.) But many system calls *conditionally* commute. Consider Unix's open system call. Two calls to open("a", O_CREAT|O_EXCL) often don't commute: one call will create the file and the other will fail because the file already exists. However, two calls to open("a", O_CREAT|O_EXCL) *do* commute if called from processes with different working directories. And even if the processes have the same working directory, two calls to open("a", O_CREAT|O_EXCL) will commute if the file already exists (both calls will return the same error). SIM commutativity allows us to distinguish these cases, even though the operations are the same in each. This, in turn, means the commutativity rule can tell us that scalable implementations exist in the commutative cases.

SIM commutativity is also interface-based. It evaluates the consequences of execution order using only the specification. Furthermore, it doesn't say that every reordering has indistinguishable results on a *given* implementation; it requires instead that every reordering *is allowed by the specification* to have indistinguishable results. This is important because any given implementation might have unnecessary scalability bottlenecks that show through the interface. The SIM commutativity of an interface can be considered even when no implementation exists. This in turn makes it possible to use the commutativity rule early in software development, during interface design and initial implementation.

3.3 Implementations

To reason about implementation scalability, we need to model implementations in enough detail to tell whether different threads' "memory accesses" are conflict-free. (As discussed in §1, conflict freedom is our proxy for scalability.) We define an implementation as a step function: given a state and an invocation, it produces a new state and a response. Special CONTINUE actions enable concurrent overlapping operations and blocking.

We begin by defining three sets:

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

An implementation m is a function in $S \times I \mapsto S \times R$. Given an old state and an invocation, the implementation produces a new state and a response (where the response must have the same thread as the invocation). A CONTINUE response indicates that a real response for that thread is not yet ready, and allows the implementation to effectively switch to another thread. CONTINUE invocations give the implementation an opportunity to complete an outstanding request (or further delay its response); however, the response must be for the thread matching the CONTINUE invocation.[1]

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

- $m(s_0, \boxed{A}) = \langle s_1, \text{CONTINUE} \rangle$
- $m(s_1, \boxed{B}) = \langle s_2, \text{CONTINUE} \rangle$
- $m(s_2, \text{CONTINUE}) = \langle s_3, \text{CONTINUE} \rangle$
- $m(s_3, \text{CONTINUE}) = \langle s_4, \boxed{B} \rangle$
- $m(s_4, \text{CONTINUE}) = \langle s_5, \boxed{A} \rangle$

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 CONTINUEs removed.

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

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

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

Two implementation steps have an *access conflict* when they are on different threads and one writes a state component that the other either writes or reads. A set of implementation steps is *conflict-free* when no pair of steps in the set has an access conflict. This notion of access conflicts maps directly onto read and write access conflicts on real shared-memory machines. Since modern MESI-based cache-coherent machines usually provide good scalability on conflict-free access patterns, we can loosely say that a conflict-free set of implementation steps "scales."

[1] There are restrictions on how implementation arguments are chosen—we assume, for example, that CONTINUE invocations are passed only when a thread has an outstanding request. Since implementations are functions, they must be deterministic. We could model implementations instead as relations, allowing non-determinism, though this would complicate later arguments somewhat.

```
$m_{ns}(s,a) \equiv$
  If $\text{head}(s.h) = a$:
    $r \leftarrow$ CONTINUE
  else if $a =$ CONTINUE and $\text{head}(s.h)$ is a response
          and $\text{thread}(\text{head}(s.h)) = \text{thread}(a)$:
    $r \leftarrow \text{head}(s.h)$                          // replay s.h
  else if $s.h \neq$ EMULATE:    // H complete or input diverged
    $H' \leftarrow$ an invocation sequence consistent with $s.h$
    For each invocation $x$ in $H'$:
      $\langle s.refstate,\ \_\rangle \leftarrow M(s.refstate, x)$
    $s.h \leftarrow$ EMULATE                // switch to emulation mode
  If $s.h =$ EMULATE:
    $\langle s.refstate,\ r\rangle \leftarrow M(s.refstate, a)$
  else:                                              // replay mode
    $s.h \leftarrow \text{tail}(s.h)$
  return $\langle s,\ r\rangle$
```

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

3.4 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.

3.5 Proof

A constructive proof for the commutativity rule can be obtained by building a scalable implementation m from the 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. This means that, within the SIM-commutative region, m scales.

To understand the construction, it helps to first imagine constructing a *non-scalable* implementation m_{ns} from the reference M. This non-scalable implementation begins in *replay mode*. While the input invocations match H, m_{ns} responds exactly according to H, without invoking the reference. When the input invocations diverge from H, however, m_{ns} no longer knows how to respond, so it enters *emulation mode*. This requires feeding M all previously received invocations to prepare its state.

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

```
$m(s,a) \equiv$
  $t \leftarrow \text{thread}(a)$
  If $\text{head}(s.h[t]) =$ COMMUTE:   // enter conflict-free mode
    $s.commute[t] \leftarrow$ TRUE;  $s.h[t] \leftarrow \text{tail}(s.h[t])$
  If $\text{head}(s.h[t]) = a$:
    $r \leftarrow$ CONTINUE
  else if $a =$ CONTINUE and $\text{head}(s.h[t])$ is a response
          and $\text{thread}(\text{head}(s.h[t])) = t$:
    $r \leftarrow \text{head}(s.h[t])$                          // replay s.h
  else if $s.h[t] \neq$ EMULATE:    // H complete/input diverged
    $H' \leftarrow$ an invocation sequence consistent with $s.h[*]$
    For each invocation $x$ in $H'$:
      $\langle s.refstate,\ \_\rangle \leftarrow M(s.refstate, x)$
    $s.h[u] \leftarrow$ EMULATE for each thread $u$
  If $s.h[t] =$ EMULATE:
    $\langle s.refstate,\ r\rangle \leftarrow M(s.refstate, a)$
  else if $s.commute[t]$:                         // conflict-free mode
    $s.h[t] \leftarrow \text{tail}(s.h[t])$
  else:                                              // replay mode
    $s.h[u] \leftarrow \text{tail}(s.h[u])$ for each thread $u$
  return $\langle s,\ r\rangle$
```

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

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

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

- $s.h[t]$—a per-thread history. Initialized to $X \,||$ 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 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 commutativity rule is proved. The construction uses SIM commutativity when initializing the reference implementation's state via H'. If the observed invocations diverge before the commutative region, then just as in m_{ns}, H' will exactly equal the observed invocations. If the observed invocations diverge in or after the commutative region, however, there's not enough information to recover the order of invocations. (The $s.h[t]$ components track which invocations have happened per thread, but not the order of those invocations 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]

3.6 Discussion

The commutativity rule and proof construction push state and history dependence to an extreme: the proof construction is specialized for a single commutative region. Repeated application of the construction can build an implementation that scales over multiple commutative regions in a history, or for the union of many histories. (This is 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.) Nevertheless, the proof construction is impractical, and real implementations usually achieve scalability using different techniques.

We believe it is easier to create practical scalable implementations for operations that commute in more situations. 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.

It is also often the case that a set of operations commutes in more than one class of situation, but no single implementation scales for all classes. Consider, for example, an interface with two calls: put(x) records a sample with value x, and max() returns the maximum sample recorded so far (or 0). Suppose

$$H = [\text{A} = \text{put}(1), \text{A}, \text{B} = \text{put}(1), \text{B}, \text{C} = \text{max}(), \text{C} = 1].$$

An implementation could store per-thread maxima reconciled by max and be conflict-free for A A B B in H. Alternatively, it could use a global maximum that put checked before writing. This is conflict-free for B B C C in H. But no correct implementation can be conflict-free across all of H. In the end, a system designer must decide which situations involving commutative operations are most important, and find practical implementation strategies that scale in those situations. In §6 we show that many operations in POSIX have implementations that scale quite broadly, with few cases of incompatible scalability classes.

The commutativity rule shows that SIM-commutative regions have conflict-free implementations. It does not show the converse, however: commutativity *suffices* for conflict-free accesses, but it may not be *necessary*. Some non-commutative interfaces may have scalable implementations—for instance, on machines that offer scalable access to strictly increasing sources of time, or when the core interconnect allows certain communication patterns to scale. Furthermore, some conflict-free access patterns don't scale on real machines; if an application overwhelms the memory bus with memory accesses, scalability will suffer regardless of whether those accesses have conflicts. We hope to investigate these problems in future, but as we show below, the rule is already a good guideline for achieving practical scalability.

4 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, we explore changes that make operations commute in more situations, enabling more scalable implementations. Already, many POSIX operations commute with many other operations, a fact we will quantify in the next section; this section focuses on problematic cases to give a sense of the subtler issues of commutative interface design.

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

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

mask, and several other properties. As a result, fork fails to commute with most other operations in the same process, such as memory writes, address space operations, and many file descriptor operations. However, applications often follow fork with exec, which undoes most of fork's sub-operations. With only fork and exec, applications are forced to accept these unnecessary sub-operations that limit commutativity.

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

Another example, stat, retrieves and returns many different attributes of a file simultaneously, which makes it non-commutative with operations on the same file that change any attribute returned by stat (such as link, chmod, chown, write, and even read). In practice, applications invoke stat for just one or two of the returned fields. An alternate API that gave applications control of which field or fields were returned would commute with more operations and enable a more scalable implementation of stat, as we show in §7.2.

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.

Embrace specification non-determinism. POSIX's "lowest available FD" rule is a classic example of overly deterministic design that results in poor scalability. Because of this rule, open operations in the same process (and any other FD allocating operations) do not commute, since the order in which they execute determines the returned FDs. This constraint is rarely needed by applications and an alternate interface that could return any unused FD would allow FD allocation operations to commute and enable implementations to use well-known scalable allocation methods. We will return to this example, too, in §7.2. 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.

Permit weak ordering. Another common source of limited commutativity is strict ordering requirements between operations. For many operations, ordering is natural and keeps interfaces simple to use; for example, when one thread writes data to a file, other threads can immediately read that data. Synchronizing operations like this are naturally non-commutative. Communication interfaces, on the other hand, often enforce strict ordering, but may not need to. For instance, most systems order all messages sent via a local Unix domain socket, even when using SOCK_DGRAM, so any send and recv system calls on the same socket do not commute (except in error conditions). This is often unnecessary, especially in multi-reader or multi-writer situations, and an alternate interface that does not enforce ordering would allow send and recv to commute as long as there is both enough free space and enough pending messages on the socket, which would in turn allow an implementation of Unix domain sockets to support scalable communication (which we use in §7.3).

Release resources asynchronously. A closely related problem is that many POSIX operations have global effects that must be visible before the operation returns. This is generally good design for usable interfaces, but for operations that release resources, this is often stricter than applications need and expensive to ensure. For example, writing to a pipe must deliver SIGPIPE immediately if there are no read FDs for that pipe, so pipe writes do not commute with the last close of a read FD. This requires aggressively tracking the number of read FDs; a relaxed specification that promised to eventually deliver the SIGPIPE would allow implementations to use more scalable read FD tracking. Similarly, munmap does not commute with memory reads or writes of the unmapped region from other threads. Enforcing this requires non-scalable remote TLB shootdowns before munmap can return, even though depending on this behavior usually indicates a bug. An munmap (perhaps an madvise) that released virtual memory asynchronously would let the kernel reclaim physical memory lazily and batch or eliminate remote TLB shootdowns.

5 Analyzing interfaces using COMMUTER

Fully understanding the commutativity of a complex interface is tricky, and achieving an implementation that avoids sharing when operations commute adds another dimension to an already difficult task. However, by leveraging the formality of the commutativity rule, developers can automate much of this reasoning. This section presents a systematic, test-driven approach to applying the commutativity rule to real implementations embodied in a tool named COMMUTER, whose components are shown in Figure 3.

First, ANALYZER takes a symbolic model of an interface and computes precise conditions under which that interface's operations commute. Second, TESTGEN takes these conditions and generates concrete test cases 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.

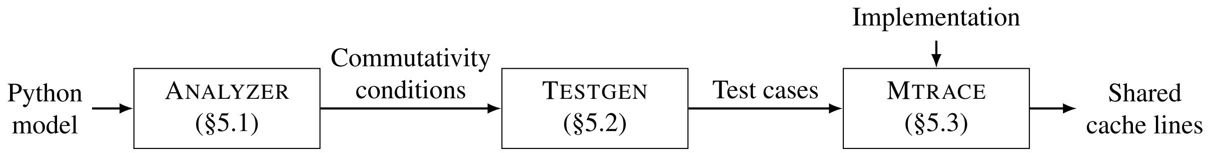


Figure 3: The components of COMMUTER.

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

5.1 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 state for exactly when sets of operations commute. A developer can inspect these expressions to understand an interface's commutativity or pass them to TESTGEN (§5.2) 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 under-constrained symbolic values.

ANALYZER considers every set of operations of a certain size (typically we use pairs). For each set of operations o, it constructs an unconstrained symbolic system state s and unconstrained symbolic arguments for each operation in o, and executes all permutations of o, each starting from a copy of s. This execution forks at any branch that can go both ways, building up *path conditions* that constrain the state and arguments that can lead to each code path. At the end of each code path, ANALYZER checks if its path condition yields an initial state and arguments that make o commute by testing if each operation's return value is equivalent in all permutations and if the system states reached by all permutations are equivalent (or can be equivalent for some choice of nondeterministic values like newly allocated inode numbers). For sets larger than pairs, ANALYZER must also check that the intermediate states are equivalent for every permutation of each subset of o.

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

def __init__(self, ...):
  self.fname_to_inum = SymDir.any()
  self.inodes = SymIMap.any()

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

Figure 4: A simplified version of our rename model.

This test codifies the definition of SIM commutativity from §3.2, except that (1) it assumes the specification is sequentially consistent, and (2) instead of considering all possible future operations (which would be difficult in symbolic execution), it substitutes state equivalence. 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 §6, only a few types need special handling beyond what the standard data types provide automatically.

Figure 4 gives an example of how a developer could model rename. The first five lines declare symbolic types (tuninterpreted declares a type whose values support only equality), and _init_ instantiates the file system state. 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.

Given two rename operations, rename(a, b) and rename(c, d), ANALYZER outputs that they 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 exist.
- Both calls are self-renames (a=b and c=d).
- One call is a self-rename of an existing file (a exists and a=b, or c exists and c=d) and it's not the other call's source (a≠c).
- Two hard links to the same inode are renamed to the same new name (a and c point to the same inode, a≠c, and b=d).

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.

5.2 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 4. 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 5. The test case includes setup code that configures the initial state of the system and a set of functions to run on different cores. Every TESTGEN test case should have a conflict-free implementation.

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

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

Figure 5: An example test case for two rename calls generated by TESTGEN for the model in Figure 4.

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

These two forms of coverage ensure that the test cases generated by TESTGEN will cover all possible paths and data structure access patterns in the model, and to the extent that the implementation is structured similarly to the model, should achieve good coverage for the implementation as well. As we demonstrate in §6, TESTGEN produces a total of 13,664 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.

5.3 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 5 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 [4]. 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 [20] information and logs of every dynamic allocation's type) and stack traces for each conflicting access.

5.4 Implementation

We built a prototype implementation of COMMUTER's three components. ANALYZER and TESTGEN consist of 3,050 lines of Python code, including the symbolic execution engine, which uses the Z3 SMT solver [19] 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.

6 Finding scalability opportunities

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 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?
- What techniques are necessary to achieve scalability for cases where current file and virtual memory systems do not scale?
- What situations might be too difficult or impractical to make scale, despite being commutative?

6.1 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; some sv6 data structures are conflict-free for offsets on different pages, but offsets within a page conflict. COMMUTER generates a total of 13,664 test cases from our model. Generating the test cases and running them on both Linux and sv6 takes a total of 8 minutes on the machine described in §7.1.

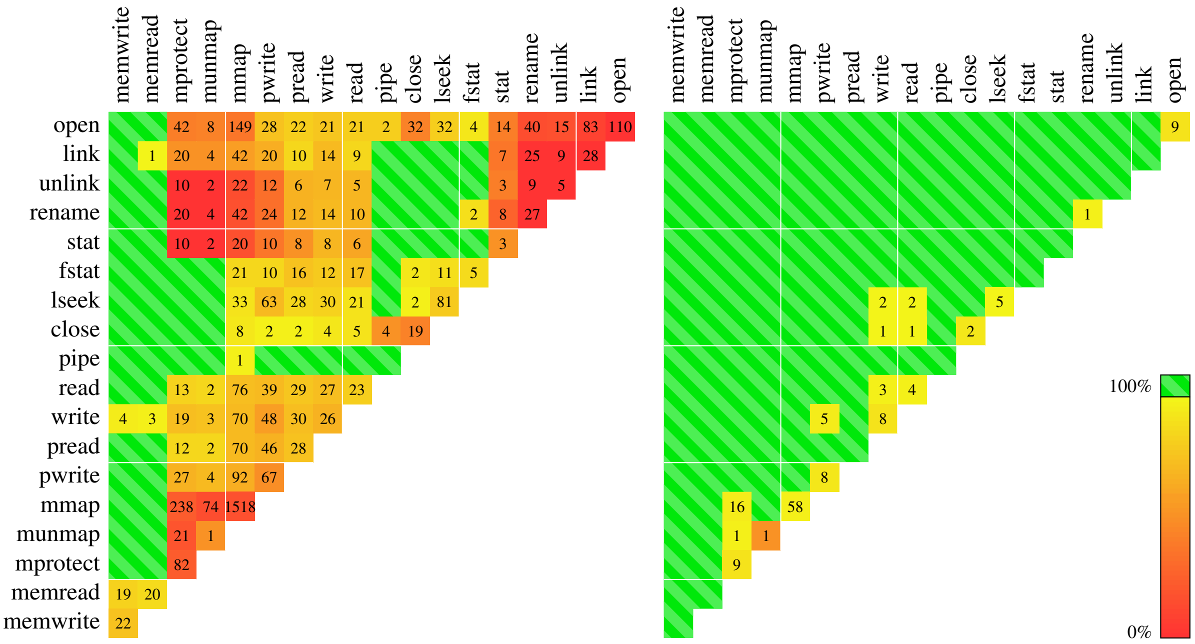
The model implementation and its model-specific test code generator are 596 and 675 lines of Python code, respectively. Figure 4 showed a part of our model, and Figure 5 gave an example test case generated by COMMUTER. We verified that all test cases return the expected results on both Linux and sv6.

6.2 Current implementation scalability

To evaluate the scalability of existing file and virtual memory systems, we used MTRACE to check the above test cases against Linux kernel version 3.8. Linux developers have invested significant effort in making the file system scale [9], 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 [17]. We evaluated the ramfs file system because ramfs is effectively a user-space interface to the Linux buffer cache. Since exercising ramfs is equivalent to exercising the buffer cache and the buffer cache underlies all Linux file systems, this represents the best-case scalability for a Linux file system. Linux's virtual memory system, in contrast, involves process-wide locks that are known to limit its scalability and impact real applications [9, 14, 41].

The left half of Figure 6 shows the results. Out of 13,664 test cases, 4,275 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



	memwrite	memread	mprotect	munmap	mmap	pwrite	pread	write	read	pipe	close	lseek	fstat	stat	rename	unlink	link	open
open	🟩	🟩	42	8	149	28	22	21	21	2	32	32	4	14	40	15	83	110
link	🟩	1	20	4	42	20	10	14	9	🟩	🟩	🟩	🟩	7	25	9	28	
unlink	🟩	🟩	10	2	22	12	6	7	5	🟩	🟩	🟩	🟩	3	9	5		
rename	🟩	🟩	20	4	42	24	12	14	10	🟩	🟩	🟩	2	8	27			
stat	🟩	🟩	10	2	20	10	8	8	6	🟩	🟩	🟩	🟩	3				
fstat	🟩	🟩	🟩	🟩	21	10	16	12	17	🟩	2	11	5					
lseek	🟩	🟩	🟩	🟩	33	63	28	30	21	🟩	2	81						
close	🟩	🟩	🟩	🟩	8	2	2	4	5	4	19							
pipe	🟩	🟩	🟩	🟩	1	🟩	🟩	🟩	🟩	🟩								
read	🟩	🟩	13	2	76	39	29	27	23									
write	4	3	19	3	70	48	30	26										
pread	🟩	🟩	12	2	70	46	28											
pwrite	🟩	🟩	27	4	92	67												
mmap	🟩	🟩	238	74	1518													
munmap	🟩	🟩	21	1														
mprotect	🟩	🟩	82															
memread	19	20																
memwrite	22																	

Linux (9,389 of 13,664 cases scale)

	memwrite	memread	mprotect	munmap	mmap	pwrite	pread	write	read	pipe	close	lseek	fstat	stat	rename	unlink	link	open
open	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	9
link	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
unlink	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩		
rename	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	1			
stat	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩				
fstat	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩					
lseek	🟩	🟩	🟩	🟩	🟩	🟩	🟩	2	2	🟩	🟩	5						
close	🟩	🟩	🟩	🟩	🟩	🟩	🟩	1	1	🟩	2							
pipe	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩								
read	🟩	🟩	🟩	🟩	🟩	🟩	🟩	3	4									
write	🟩	🟩	🟩	🟩	🟩	5	🟩	8										
pread	🟩	🟩	🟩	🟩	🟩	🟩	🟩											
pwrite	🟩	🟩	🟩	🟩	🟩	8												
mmap	🟩	🟩	16	🟩	58													
munmap	🟩	🟩	1	1														
mprotect	🟩	🟩	9															
memread	🟩	🟩																
memwrite	🟩																	

sv6 (13,528 of 13,664 cases scale)

Figure 6: Scalability for system call pairs, showing the fraction and 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 5.

reference count on a struct file, making commutative operations such as two fstat calls on the same file descriptor not scale. Coarse-grained locks are another source of access conflicts. For instance, Linux locks the parent directory for any operation that creates file names, even though operations that create distinct names generally commute. Similarly, we see that coarse-grained locking in the virtual memory system severely limits the conflict-freedom of address space manipulation operations. This agrees with previous findings [9, 14, 15], which demonstrated these problems in the context of several applications.

6.3 Making test cases scale

Given that Linux does not scale in many cases, how hard is it to implement scalable file systems and virtual memory systems? To answer this question, we designed and implemented a ramfs-like in-memory file system called ScaleFS and a virtual memory system called RadixVM for sv6, our research kernel based on xv6 [18]. RadixVM appeared in previous work [15], so we focus on ScaleFS here. Although it is in principle possible to make the same changes in Linux, we chose not to implement ScaleFS in Linux because ScaleFS's design would have required modifying code 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). The right half of Figure 6 shows the result of applying COMMUTER to sv6.

ScaleFS makes extensive use of existing techniques for scalable implementations, such as per-core resource allocation, double-checked locking, lock-free readers using RCU [31], scalable reference counts using Refcache [15], and seqlocks [28: §6]. These techniques lead to several common patterns, as follows; we illustrate the patterns with example test cases from COMMUTER that led us to discover these situations:

Layer scalability. ScaleFS uses data structures that themselves naturally satisfy the commutativity rule, such as linear arrays, radix arrays [15], 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 array, so that concurrent reads or writes to different file pages scale, even in the presence of operations extending or truncating the file. Many operations also use this radix array to determine if some offset is within the file's bounds without risking conflicts with operations that change the file's size.

Defer work. Many kernel resources are shared, such as files and pages, 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 [15] 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. 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 scalable per-core inode allocation.

Precede pessimism with optimism. Many operations in ScaleFS have an optimistic check stage followed by a pessimistic update stage, a generalized sort of double-checked locking. The optimistic stage checks conditions for the operation and returns immediately if no updates are necessary (this is often the case for error returns, but can also happen for success returns). This stage does no writes or locking, but because no updates are necessary, it is often easy to make atomic. If updates are necessary, the operation acquires locks or uses lock-free protocols, re-verifies its conditions to ensure atomicity of the update stage, and performs updates. For example, lseek first computes the new offset using a lock-free read-only protocol and returns early if the new offset is invalid or equal to the current offset. Otherwise, lseek locks the file offset, and re-computes the new offset to ensure consistency.

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.

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.

6.4 Difficult-to-scale cases

As Figure 6 illustrates, there are a few (136 out of 13,664) commutative test cases for which ScaleFS is not conflict-free. The majority of these tests involve idempotent updates to internal state, such as two lseek operations that both seek a file descriptor to the same offset, or two anonymous mmap operations with the same fixed base address and permissions. While it is possible implement these scalably, every implementation we considered significantly impacted the performance of more common operations, so we explicitly chose to favor common-case performance over total scalability. Even though we decided to forego scalability in these cases, the commutativity rule and COMMUTER forced us to consciously make this trade-off.

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

7 Performance evaluation

The previous section showed that ScaleFS and RadixVM achieve conflict-freedom for nearly all commutative operations, which should result in perfect scalability in theory. This section shows that these results translate to scalability on real hardware for a complete operating system by answering the following questions:

- Do non-commutative operations limit performance on real hardware?
- Do conflict-free implementations of commutative operations scale on real hardware?
- Does optimizing for scalability sacrifice sequential performance?

7.1 Experimental setup

To answer these questions, we use sv6. In addition to the operations analyzed in §6, we scalably implemented other commutative operations (e.g., posix_spawn) and many of the modified POSIX APIs from §4. All told, sv6 totals 51,732 lines of code, including user space and library code.

We ran experiments on an 80-core machine with eight 2.4 GHz 10-core Intel E7-8870 chips and 256 GB of RAM. When varying the number of cores, benchmarks enable whole sockets at a time, so each 30 MB socket-level L3 cache is shared by exactly 10 enabled cores. We also report single-core numbers for comparison, though these are expected to be higher because one core can use the entire 30 MB cache.

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

As a single core performance baseline, we compare against the same benchmarks running on Linux 3.5.7 from Ubuntu Quantal. Direct comparison is difficult because Linux implements many features sv6 does not, but this comparison indicates sv6's performance is sensible.

7.2 Microbenchmarks

We evaluate scalability and performance on real hardware using two microbenchmarks and an application-level benchmark. Each benchmark has two variants, one that uses standard, non-commutative POSIX APIs and another that accomplishes the same task using the modified, more broadly commutative APIs from §4. By benchmarking the standard interfaces against their commutative counterparts, we can isolate the cost of non-commutativity and also examine the scalability of conflict-free implementations of commutative operations.

We run each benchmark three times and report the mean. Variance from the mean is always under 4% and typically under 1%.

statbench. In general, it's difficult to argue that an implementation of a non-commutative interface achieves the best possible scalability for that interface and that no implementation could scale better. However, in limited cases, we can do exactly this. We start with statbench, which measures the scalability of fstat with respect to link. This benchmark creates a single file that $n/2$ cores repeatedly fstat. The other $n/2$ cores repeatedly link this file to a new, unique file name, and then unlink the new file name. As discussed in §4, 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 [25]).

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 [15] for the link count so that the links and unlinks do not conflict, and place it on its own cache line to avoid false sharing. Figure 7(a) shows the results. With the commutative fstatx, statbench scales perfectly and experiences zero L2 cache misses in fstatx, while fstat severely limits the scalability of statbench.

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 the expense of link and unlink), but still does not scale. With a shared link count, each fstat call experiences exactly one L2 cache miss (for the cache line containing the link count), which means this is the most scalable that fstat can possibly be in the presence of concurrent links and unlinks. Yet, despite sharing only a single cache line, this seemingly innocuous non-commutativity limits the implementation's scalability. One small tweak to make the operation commute by omitting st_nlink eliminates the barrier to scaling, demonstrating the cost of non-commutativity.

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 cores, fstat is 3.9× more expensive than on Linux. In contrast, fstatx can avoid this overhead unless link counts are requested; like fstat with a shared count, it performs similarly to Linux's fstat on a single core.

openbench. Figure 7(b) shows the results of openbench, which stresses the file descriptor allocation performed by open. In openbench, n threads concurrently open and close per-thread files. These calls do not commute because each open must allocate the lowest unused file descriptor in the process. For many applications, it suffices to return any unused file descriptor (in which case the open calls commute), so sv6 adds an O_ANYFD flag to open, which it implements using per-core partitions of the FD space. Much like statbench, the standard, non-commutative open interface limits openbench's scalability, while openbench with O_ANYFD scales linearly. Furthermore, there appears to be 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.

7.3 Application performance

Finally, we perform a similar experiment using a simple mail server to produce a system call workload more representative of a real application. Our mail server uses a se-

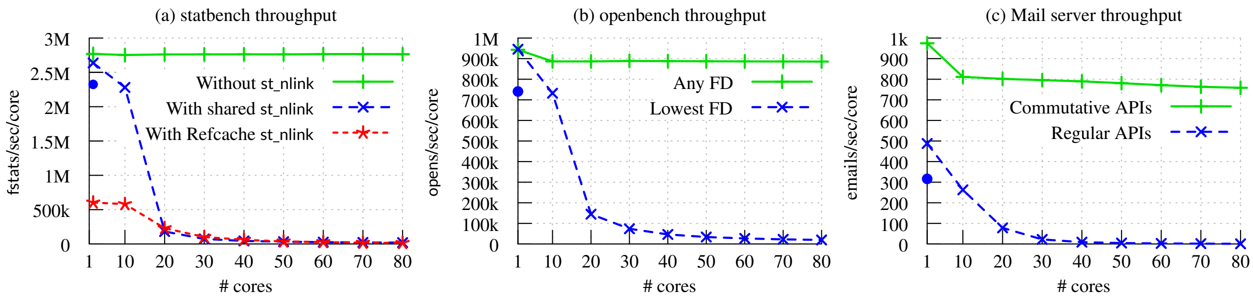


Figure 7: Benchmark throughput in operations per second per core with varying core counts on sv6. The blue dots indicate single core Linux performance for comparison.

quence of separate, communicating processes, each with a specific task, roughly like qmail [5]. mail-enqueue takes a mailbox name and a single message on stdin, 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 stdin 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 §4. For queue notifications, we use a Unix domain datagram socket; sv6 implements this with a single shared queue in ordered mode and with per-core message queues with scalable load balancing in unordered mode. 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 7(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 non-commutative operations, the results are quite similar. Non-commutative operations cause the benchmark's throughput to collapse at a small number of cores, while the configuration that uses commutative APIs achieves 7.5× scalability from 1 socket (10 cores) to 8 sockets.

8 Conclusion

The scalable commutativity rule provides a new approach for software developers to understand and exploit multicore scalability starting at the software interface. 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. We hope that programmers will find the commutativity rule helpful to produce software that is scalable by design.

COMMUTER, sv6, and a browser for the data in this paper are available at `http://pdos.csail.mit.edu/commuter`.

Acknowledgments

We thank Silas Boyd-Wickizer, Marc Shapiro, Rachid Guerraoui, Butler Lampson, the anonymous reviewers, and our shepherd, Jon Howell, for their feedback. Additionally, Silas Boyd-Wickizer contributed to much of the implementation of sv6. 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

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

[2] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order:

Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages*, Austin, TX, January 2011.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[4] F. Bellard et al. QEMU. `http://www.qemu.org/`.

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

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

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

[8] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[9] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

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

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

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

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

[15] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference*, Prague, Czech Republic, April 2013.

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

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

[18] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system. `http://pdos.csail.mit.edu/6.828/2012/xv6.html`.

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

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

[21] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

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

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

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

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

[27] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, Madrid, Spain, September 2002.

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

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

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

[31] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2002.

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

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

[34] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, CA, June 2011.

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

[36] A. Roy, S. Hand, and T. Harris. Exploring the limits of disjoint access parallelism. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

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

[38] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, Grenoble, France, October 2011.

[39] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, June 2011.

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

[41] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. *SIGPLAN Notices*, 46(11):79–88, June 2011.

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

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