Veracity: Declarative Multicore Programming with Commutativity

ADAM CHEN, Stevens Institute of Technology, USA
PARISA FATHOLOLUMI, Stevens Institute of Technology, USA
ERIC KOSKINEN, Stevens Institute of Technology, USA
JARED PINCUS, Stevens Institute of Technology, USA

There is an ongoing effort to provide programming abstractions that ease the burden of exploiting multicore hardware. Many programming abstractions (*e.g.*, concurrent objects, transactional memory, etc.) simplify matters, but still involve intricate engineering. We argue that some difficulty of multicore programming can be meliorated through a declarative programming style in which programmers directly express the independence of fragments of sequential programs.

In our proposed paradigm, programmers write programs in a familiar, sequential manner, with the added ability to explicitly express the conditions under which code fragments sequentially commute. Putting such commutativity conditions into source code offers a new entry point for a compiler to exploit the known connection between commutativity and parallelism. We give a semantics for the programmer's sequential perspective and, under a correctness condition, find that a compiler-transformed parallel execution is equivalent to the sequential semantics. Serializability/linearizability are not the right fit for this condition, so we introduce scoped serializability and show how it can be enforced with lock synthesis techniques.

We next describe a technique for automatically verifying and synthesizing commute conditions via a new reduction from our commute blocks to logical specifications, upon which symbolic commutativity reasoning can be performed. We implemented our work in a new language called Veracity, implemented in Multicore OCaml. We show that commutativity conditions can be automatically generated across a variety of new benchmark programs, confirm the expectation that concurrency speedups can be seen as the computation increases, and apply our work to a small in-memory filesystem and an adaptation of a crowdfund blockchain smart contract.

CCS Concepts: • **Software and its engineering** → Automated static analysis; • **Theory of computation** → **Parallel computing models**; *Program specifications*; **Program analysis**; Operational semantics; *Pre- and post-conditions*; • **Computing methodologies** → *Concurrent programming languages*.

Additional Key Words and Phrases: commutativity analysis, commutativity conditions, pre-condition synthesis, serializability, parallelization

ACM Reference Format:
Adam Chen, Parisa Fathololumi, Eric Koskinen, and Jared Pincus. 2022. Veracity: Declarative Multicore Programming with Commutativity. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 186 (October 2022), 31 pages. https://doi.org/10.1145/3563349

Authors' addresses: Adam Chen, Stevens Institute of Technology, USA, achen19@stevens.edu; Parisa Fathololumi, Stevens Institute of Technology, USA, pfathol1@stevens.edu; Eric Koskinen, Stevens Institute of Technology, USA, eric.koskinen@stevens.edu; Jared Pincus, Stevens Institute of Technology, USA, jpincus@stevens.edu.

2475-1421/2022/10-ART186
https://doi.org/10.1145/3563349

1 INTRODUCTION

Writing concurrent programs is difficult. Researchers and practitioners, seeking to make life easier, have developed better paradigms for concurrent programming, such as concurrent objects [Herlihy and Wing 1990], transactional memory [Harris and Fraser 2003; Herlihy et al. 2003; Herlihy and Moss 1993; Saha et al. 2006], actors [Armstrong 1997], parallel for [Allen et al. 2005], goroutines [Prabhakar and Kumar 2011], ownership [Matsakis and Klock 2014], composable atomicity [Golan-Gueta et al. 2015] and others. As another strategy, compiler designers have instead sought to automatically parallelize programmers' sequential programs (see, *e.g.* [Blume et al. 1994; Blume and Eigenmann 1992; Li et al. 1990]). Others have devised more *declarative* and/or domain-specific programming models such as Fortress [Steele Jr. 2005]. Some have extended concurrent collection programming with graph-based languages for GPUs [Grossman et al. 2010]; some have introduced DSLs for grid computing [Orchard et al. 2010]; some delay side effects in blocks [Lindley 2007]; others aim to further separate high-level programming from low-level concurrent programming (*e.g.* the Habanero project [Barik et al. 2009]). (See also the DAMP Workshop [Various 2012].) Still others aim to improve proof techniques (*e.g.* [Brookes and O'Hearn 2016; Elmas 2010; Jones 1983; Owicki and Gries 1976; Vafeiadis 2010]) so that verification tools can complement the error-prone task of concurrent programming.

Meanwhile, it has been long known that *commutativity* can be used for scalability in concurrent programming, dating back to the database community [Bernstein 1966; Korth 1983; Weihl 1988, 1983] and, more recently, for system building [Clements et al. 2015]. There have thus been a growing collection of works aimed at exploiting commutativity [Dickerson et al. 2017, 2019; Hassan et al. 2014; Herlihy and Koskinen 2008; Kulkarni et al. 2011, 2008, 2007; Ni et al. 2007], synthesizing/verifying commutativity [Aleen and Clark 2009; Bansal et al. 2018; Gehr et al. 2015; Kim and Rinard 2011; Koskinen and Bansal 2021], and using commutativity analysis for parallelization in compilers [Rinard and Diniz 1997] and blockchain smart contracts [Pîrlea et al. 2021]. Despite works on declarative programming models for concurrency and separate works that exploit commutativity, to our knowledge none have yet sought to combine the two by giving the programmer (or automated analysis) agency in specifying commutativity directly in the programming language.

Programming with commutativity. In this paper, we introduce a new sequential programming paradigm in which *explicit* commutativity conditions are expressions that are part of the programming language, to be exploited by the compiler for concurrent execution. Specifically, we introduce `commute` statements of the form:

$$\texttt{commute (expr) \{\{ stmt}_1\texttt{ \} \{ stmt}_2\texttt{ \}\}}$$

The idea is to allow programmers to continue to write programs using sequential reasoning and, still from that sequential perspective, express the conditions under which statements commute. For example, a programmer may specify that statements incrementing and decrementing a counter `c` commute when the counter's value is above 0. Putting commutativity directly into the programming language has three key benefits:

(1) *Sequential programming*: Programmers need only reason sequentially, which is far more approachable than concurrent reasoning. The semantics are simply that non-deterministic behavior is permitted when `commute` conditions hold and, otherwise, resort to sequencing in the order written.
(2) *Sequential verification/inference*: These commutativity conditions can be automatically verified or even synthesized without the need to consider all interleavings.

(3) *Parallel execution*: Once commutativity conditions are available, they can be used *within the compiler* and combined with lock synthesis (existing techniques,*e.g.* [Cherem et al. 2008; Vechev et al. 2010] or new, more targeted techniques) to generate parallel compiled programs.

Since programmers write with the sequential semantics in mind, we naturally would like to ensure a concurrent semantics that is equivalent to the sequential semantics, but interestingly, the *de facto* standards of serializability [Papadimitriou 1979] and linearizability [Herlihy and Wing 1990] do not quite fit the bill. Parallelizing programs with arbitrarily nested (and sequentially composed) `commute` blocks, involves a recursive sub-threading shape. We introduce *scoped serializability* to describe a sufficient condition under which the concurrent semantics is equivalent to the sequential semantics when used with valid commutativity conditions. Scoped serializability is *stronger* than serializability; though that may at first seem distasteful, it is necessary to capture the structure of nested commute blocks, and it is not bad given that the program can be written by only thinking of its sequential semantics in contrast to, say, transactional memory where the programmer must decide how to organize their program into threads and atomic sections. Scoped serializability can then be enforced through lock synthesis techniques, another area of ongoing research (*e.g.* [Cherem et al. 2008; Flanagan and Qadeer 2003; Golan-Gueta et al. 2015; Vechev et al. 2010]). While such existing techniques can be applied here, they are geared to a more general setting. We describe alternative locking and re-write strategies that incur less mutual exclusion and are more geared toward scoped serializability.

Although `commute` conditions can be written by programmers, we next focus on correctness and describe how such conditions can be automatically verified and even inferred. We describe a novel translation from `commute` programs to an embedding as a logical specification, for which recent techniques [Bansal et al. 2018] and tools [Servois 2018] can verify or synthesize commutativity conditions. The pieces do not immediately align: `commute` blocks occur in program contexts (with local/global variables) and our language provides support for nested commute blocks as well as builtin primitives such as dictionaries. We show that, nonetheless, these program pieces can be translated to logical specifications and those specifications can then be used to synthesize `commute` conditions via abstraction-refinement [Bansal et al. 2018]. Synthesized conditions can sometimes be overly complex and/or describe only trivial cases (due to incompleteness in those algorithms) so a programmer may instead wish to manually-provide a `commute` condition and, in such cases, we verify the provided condition.

We implement our work in a front-end compiler/interpreter for a new language called Veracity[1], built on top of Multicore OCaml [mul 2014] 4.12.0, and a new commutativity condition verifier/synthesizer [Servois2 2022], using a variety of underlying SMT solvers. Veracity uses the recent highly concurrent hashtable implementation `libcuckoo` [libcuckoo 2013]. We publicly released Veracity, as well as 30 benchmark programs which are, as far as we know, the first to include `commute` statements directly in the programming language. We provide a preliminary experimental evaluation, offering promising evidence that (i) `commute` conditions can be automatically verified/inferred, (ii) as expected, speedups can be seen as the computation size grows, and (iii) `commute` blocks could be used in applications such as in-memory file systems and blockchain smart contracts.

Contributions. In summary, our contributions are:

- `commute` statements, a new sequential language statement that weakens sequential composition, to explicitly express commutativity conditions, which are then used for parallelization.
- (Sec. 4) Sequential and concurrent semantics for the language.

[1]A portmanteau of "verified" and "commutativity."

- (Sec. 5) A correctness condition for parallelization called *scoped serializability* that, with commutativity, implies that the concurrent semantics are equal to the sequential semantics. We also present methods for automatically enforcing scoped serializability.
- (Sec. 6) A method for automatic verification and synthesis of `commute` conditions, via an embedding into ADT specifications.
- (Sec. 7) A frontend compiler/interpreter for a new language Veracity, built in Multicore OCaml with libcuckoo and a new implementation of Servois [Servois2 2022].
- (Sec. 8) A preliminary evaluation demonstrating synthesized `commute` conditions, scaling speedups, and relevant applications.

As indicated throughout this paper, some formal detail is deferred to our technical report [Chen et al. 2022a].

Limitations. While we have implemented a multicore interpreter, we reserve back-end compilation questions for future research: there is much to be explored there, particularly with respect to optimization. When commute blocks contain loops, we infer/verify commute conditions by first instrumenting loop summaries [Ernst 2020, 2022; Kroening et al. 2008; Silverman and Kincaid 2019; Xie et al. 2017] with havoc/assume. We used Korn[2] [Ernst 2022] and some manual reasoning to increase Korn's precision, but we plan to fully automate loop support in the future. Scoped serializability is formalized in Sec. 5.5, and we argue that it can be automatically enforced. Because automated lock synthesis is a research area largely orthogonal to our work, and would involve implementing other orthogonal components (such as alias analysis), our prototype implementation does not perform lock synthesis automatically. Rather, we manually applied the procedure of Sec. 5.5 and detail how it applied to our benchmarks in Sec. 8.

2 OVERVIEW

We begin with some simple examples to illustrate how a programmer may use `commute` statements and, from these examples, outline the challenges addressed in this paper. The following is perhaps the most trivial example: commuting blocks that increment and decrement a value.

```
    commute(true) { { c = c-x; }  { c = c+y; } }
```

These two operations clearly commute: the final value of `c` is the same regardless of the order in which the blocks are executed, due to the natural commutativity of integer arithmetic. Moreover, this holds for any initial value of `c`, justifying the trivial *commutativity condition* of `true`. Compiler optimizations can exploit simple situations like this where commutativity always holds and parallelize with a lock to protect the data race [Rinard and Diniz 1997]. In this example, the overhead of parallelization is not worth it, so let us move to more examples. Now consider a case where the condition is not simply `true`, and must be specified.

Example 2.1 (Conditional commutativity - `simple.vcy`).

```
commute (c>a) {
  f1: { t = foo(c>b); a = a - |t|; }
  f2: { u = bar(c>a); }
}
```

Above we refer to the two blocks labeled $\mathfrak{f}_1$ and $\mathfrak{f}_2$ as the two "fragments" (or co-fragments) of the `commute` statement[3]. We assume that `foo` and `bar` are pure but costly computations. In this case, the programmer has written the commutativity condition `c>a`. Although $\mathfrak{f}_2$ reads the variable `a` which is written by $\mathfrak{f}_1$, it only observes whether or not `c>a`. Moreover, $\mathfrak{f}_1$ may modify `a`, but will

[2]https://github.com/gernst/korn
[3]The terminology is chosen to avoid nebulous words like "blocks," "statements," and "nested."

only decrease its value, which does not impact the boolean observation `c>a` made by $\mathfrak{f}_2$. Hence, the fragments commute whenever `c>a` initially. It is necessary to include a condition because when the condition does not hold, the operations may not always commute (they only commute if `foo` returns a value with magnitude greater than `a-c`, which may be computationally difficult to determine *a priori*) and we must resort to sequential execution. Note that, throughout this paper, we mostly use `commute` statements with only two fragments, although our implementation supports N-ary `commute` statements.

We emphasize that a programmer needs to reason only in the sequential setting. However, as detailed below, our compiler can transform the above program into a concurrent one, permitting `foo` and `bar` to execute in parallel when `c>a` and resort to sequential execution otherwise. Note that it is not possible to parallelize these computations with a simple dataflow analysis. The challenge is that `bar(c>a)` cannot be moved above `a = a - |t|` in the absence of protection from the commutativity condition. More generally, the intuition is that each fragment may make observations about the state, and also perform mutations to the state. These fragments can be parallelized if the mutations made by one do not change the observations made by the other, which is captured by the notion of commutativity and specified in the `commute` expression. Furthermore, the compiler does not need to introduce any locks because there is a read-write in one fragment, but only a single read in the other, and the `commute` condition ensures that reading before or after the write does not matter.

Example 2.2 (Counter control flow - `calc.vcy`*).*

```
commute (c>0) {
  𝔣1: { x = calc1(a); c = c + (x*x); }
  𝔣2: { if (c>0 && y<0)
          { c = c-1; z = calc2(y); }
        else
          { z = calc3(y); } }
}
```

Example 2.2 considers `commute` blocks with fragments that have control flow. `calc1`/`calc2`/`calc3` are again pure calculations. Here, again using only sequential reasoning, the programmer has provided the `commute` condition `c>0`. This is necessary because, *e.g.*, if `c` is 0 initially, then in one order `c` will be `x*x` and in the other order, `c` will be `(x*x)-1`. A compiler can use this sequential commutativity condition to parallelize these operations, provided that it also ensures a new form of serializability (detailed in the next subsection) through lock synthesis techniques. In this case, a single lock must be synthesized to protect access to `c`, yet allow parallel execution of `calc1` with `calc2`/`calc3` (*i.e.*, $\mathfrak{f}_1$ holds the lock only during the `c` read/write, and $\mathfrak{f}_2$ holds the lock from the beginning until before the call to `calc2`/`calc3`). This enforces serializability of the fragments, so we may treat them as atomic points in the execution, and then, due to the commutativity condition, it does not matter which point occurs first.

This example also shows that other orthogonal strategies such as future/promises [Chatterjee 1989; Liskov and Shrira 1988] do not subsume `commute` blocks. A promise for the value of `x` must be evaluated at the end of $\mathfrak{f}_1$ before beginning $\mathfrak{f}_2$ because `x` is used in $\mathfrak{f}_1$. Promises cannot avoid the data-flow dependency across $\mathfrak{f}_1$ into $\mathfrak{f}_2$. By contrast, commutativity allows a compiler to execute the fragments out of order, effectively relaxing the data dependency (*e.g.* although `c` may change, `c>0` will not). (In future work one could combine promises with `commute` blocks.)

Commutative computation appears in many contexts beyond counters, *e.g.*, vector or matrix multiplication:

Example 2.3 (Multiplication - `matrix.vcy`*).*

```
commute (y == 0) {
  { int sc = scale(y); int y0 = y;
    x = x*sc; y = 3*y; z = z - 2*y0; }
  { int y₀ = y;
    x = 5*x; y = 4*y; z = 3*z - y₀;
    out = summarize(z); }}
```

Here the resulting vector (x,y,z) can be the same in either order of the blocks, but only in certain cases. This is nearly a trivial diagonal matrix multiplication, except for a non-diagonal contribution to z. Under the condition that y=0 initially, however, the contribution vanishes. Our compiler can exploit this commutativity and execute these fragments in parallel, synthesizing a lock for the read/writes to x/y/z. If the `scale` and `summarize` computations are time-consuming, then there is a large payoff for parallelizing: locks are held for only small windows, and execution of the costly `scale` and `summarize` can overlap.

It is increasingly common for programming languages to have *builtin abstract datatypes* (ADTs), such as dictionaries/hashtables, queues, stacks, etc. These present further opportunities for sequential commutativity-based programming and automatic parallelization through highly concurrent implementations of those ADTs. Consider the following example:

Example 2.4 (Builtin dictionaries - `dict.vcy`*).*

```
commute (res ≠ input) {
  𝔣₁:  { t = calc(x); stats[res] = t; }
  𝔣₂:  { y = stats[input]; y = y + x; }
}
```

Above, `stats` is a dictionary/hashtable, with a `put` operation in fragment $\mathfrak{f}_1$, and a `get` in $\mathfrak{f}_2$. The programmer (or our analysis) has provided a commutativity condition that the keys are distinct, using sequential reasoning.

Hashtables have appealing commutativity properties [Bronson et al. 2010; Herlihy and Koskinen 2008; Ni et al. 2007], well understood sequential semantics and recent high-performance concurrent implementations [Li et al. 2014; Liu et al. 2014] such as `libcuckoo` [libcuckoo 2013]. For this example, the memory accesses do not conflict except on the hashtable. Thus, the compiler can translate the `commute` block to be executed concurrently simply by using a hashtable with a concurrent implementation like `libcuckoo`, with no other locking needed. That linearizable implementation ensures the atomicity of the operations on `stats` and the commutativity condition (`res ≠ input`) ensure that $\mathfrak{f}_1$ will commute with $\mathfrak{f}_2$ regardless of the interleaving. `commute` blocks that use other builtin ADTs such as Sets, Stacks or Queues can similarly benefit from this strategy (locking may be necessary when there are multiple such operations in each fragment).

As seen in the examples above, our work is particularly suited to settings with long pure computations mixed with commutative updates to shared memory. Examples might include in-memory filesystems, machine learning, and smart contracts. In this paper we focus on the theoretical foundations, algorithms, and verification/synthesis with some predictable empirical results. We defer real-world applications and integration to future work.

2.1 Semantics of `commute`

As our proposed language has no explicit `fork` or `clone` command, a user cannot explicitly express concurrency. Instead, a user (and, later, our automated techniques) can express *sequential* nondeterminism of code fragments through `commute` statements. Parallelism is only introduced by our compiler, which exploits the `commute`-specified allowance of nondeterminism. Incorporating such `commute` statements into a programming language has semantic implications. A natural goal is to provide a concurrent semantics that somehow corresponds to a sequential semantics, *e.g.* trace inclusion, simulation, etc. However, the programming model is also distinct from many prior models that involve explicit fork/clone, like transactional memory or concurrent objects and their respective correctness conditions: serializability/opacity [Guerraoui and Kapalka 2008; Papadimitriou 1979] or linearizability [Herlihy and Wing 1990].

Because of this, there is also a distinction in what correctness condition is appropriate. To see the distinction from transactional memory and serializability, consider the following `commute` blocks:

Example 2.5 (Nested commute blocks – `nested.vcy`*).*

```
y := 0; x := 1;
commute(true) {
   { commute(true)
      𝔣1: { x := 0; }
      𝔣2: { x := x * 2; } }
 𝔣3:{ if (x>=2) y := 1; }
}
```

A notion of serializability adapted for this setting might require that any execution be equivalent to some serial order of $\mathfrak{f}_1$, $\mathfrak{f}_2$, and $\mathfrak{f}_3$. This would permit the interleaved execution:

$$\texttt{y:=0},\ \ \texttt{x:=1},\ \ \mathfrak{f}_2\texttt{:x:=x*2},\ \ \mathfrak{f}_3\texttt{:y:=1},\ \ \mathfrak{f}_1\texttt{:x:=0}$$

ending in a final state where x=0 and y=1. However, the semantics of `commute` in our language do not permit a serial order where $\mathfrak{f}_3$ occurs between ($\mathfrak{f}_1$;$\mathfrak{f}_2$) nor between ($\mathfrak{f}_2$;$\mathfrak{f}_1$).

In Section 4 we introduce sequential and concurrent semantics for a simple language augmented with `commute` statements. From the programmer's perspective, the semantics of `commute` statements are naturally captured through nondeterminism. At runtime however, our interpreter uses a parallel execution semantics through recursively nested threading. We define a correctness condition called *scoped serializability* which is stronger than serializability, yet ensures an equivalence between the concurrent semantics and the sequential semantics. This equivalence means that the program can be automatically parallelized. We then give some techniques for *enforcing* scoped serializability. One *can* naïvely use mutual exclusion or existing lock synthesis techniques (which were designed for programs with explicit forking). We then discuss improved locking techniques that are more geared to this nested, pair-wise setting. We show that in certain cases, some sound reordering of fragments can minimize or even eliminate the need for locking.

2.2 Automated Verification and Inference of `commute` Conditions

A key benefit of `commute` statements is that we can use *sequential reasoning* to discover commutativity conditions (*i.e.* we do not have to consider interleavings). Commute conditions are often simple enough for programmers to write, though they must be correct in order for parallel execution to be safe. In Sec. 6 we turn to this correctness issue and discuss how to automatically verify or even infer these conditions.

A collection of recent techniques and tools have emerged pertaining to commutativity reasoning. Bansal et al. [2018] used a counterexample-guided abstraction refinement (CEGAR) strategy in the Servois tool [Servois 2018] to synthesize commutativity conditions of object methods' pre/post specifications, but not the *ad hoc* code that appears in our `commute` blocks. Koskinen and Bansal [2021] verified commutativity conditions of source code through reachability. We found that in this setting, the reachability approach introduced many unnecessary, extraneous paths and variables (reducing performance and complicating conditions), and that SMT solvers struggled in many cases. Finally, Pîrlea et al. [2021] described an abstract interpretation that tracks the linearity of assignments to determine when operations *always* commute. Yet our `commute` blocks need *conditions* for commutativity, and also have non-linear behaviors such as matrix multiplication, modulus, calls to builtin ADTs, etc.

We enable automated verification or inference of `commute` conditions through a novel embedding of `commute` block fragments into logical specifications, fit for verification with SMT or inference with the abstraction-refinement of Bansal et al. [2018]. The translation is not immediate: `commute` statements occur within the context of a program, mutate global/local variables, invoke builtin ADTs such as dictionaries, and may involve nested `commute` statements. The idea is thus to capture the program context for the given `commute` block as the pre-state, then translate each fragment's code into a logical postcondition, describing how the fragment mutates the context. Builtin operations are treated by "inlining" their specification, and nested `commute` blocks can be treated as sequential composition, thanks to their sequential semantics. Synthesized `commute` conditions can then be reverse translated back into the source language. Sometimes these conditions may be overly complex or too trivial; thus, a programmer still may wish to provide their own condition, which can be verified via the same embedding.

2.3 The Veracity Programming Language

We implemented our work in a prototype front-end compiler and interpreter for a new language called Veracity[4], built on top of Multicore OCaml [mul 2014] and Servois2 [Servois2 2022], a new implementation of the Bansal et al. [2018] algorithm[5]. Our interpreter performs the embedding to verify and synthesize `commute` conditions, and uses Multicore OCaml domains and a foreign function interface to `libcuckoo` [libcuckoo 2013] for concurrent execution of `commute` statements.

We created a suite of 30 benchmark programs featuring `commute` blocks, which are available in the benchmarks directory of the Veracity repository[6]. We first show that Veracity can synthesize commutativity conditions for most of these programs, which include a variety of program features: linear expressions, non-linear expressions, vector multiplication, builtin ADTs, modulus, nesting, arrays, and loops. When these synthesized conditions are too complex, we show that better manual ones can be verified.

We next confirm the expectation that concurrent execution offers a speedup over sequential execution, and that speedup increases as non-conflicting computations increases. Although no existing programs are written with `commute` statements, we examine two case studies in which we adapted programs to use `commute` statements, including an Algorand smart contract and an in-memory file system.

3 PRELIMINARIES

We provide some basic background and definitions used throughout this paper.

States. A state $\sigma \in \Sigma$ is a partial mapping $Varnames \sqcup MemLocs \rightharpoonup V$, *i.e.* the disjoint union of variable names and memory locations, to program values. We assume a notion of scope, and that each program variable statically refers to a unique element of the state. We assume that states can be composed and decomposed by scope, *i.e.*, $\sigma = \sigma_0 \oplus \sigma_1$, and that relations on Σ are also relations over the decomposed states. We work with equality on states $\simeq$ in the usual way (*e.g.*, equivalence of valuation of accessible, primitive variables).

Syntax. We begin with syntax and contextual semantics for a standard programming language. The language involves constants c, references, mutable variables and declarations. Here, we use t to refer to the language of types, v to refer to the language of variables, and e to refer to the language of expressions. We omit them here for brevity as we assume a standard definition for them.

$$\begin{array}{lll@{\qquad\qquad}lll} c & ::= & int \mid string \mid bool \mid () \mid ref & v\text{-}decls & ::= & \epsilon \mid v\text{-}decls' \\ lval & ::= & \text{varname} \mid ref & v\text{-}decls' & ::= & t\ v = e \mid t\ v = e, v\text{-}decls' \end{array}$$

[4]Available at https://github.com/veracity-lang/veracity.
[5]Available at https://github.com/veracity-lang/servois2.
[6]https://github.com/veracity-lang/veracity/tree/main/benchmarks

$$\begin{array}{rcl}
\langle v, \sigma\rangle & \rightsquigarrow & \langle \sigma(v), \sigma\rangle \\
\langle c_0[c_1], \sigma\rangle & \rightsquigarrow & \langle [\![c_0[c_1] : ref]\!]\sigma, \sigma\rangle \\
\langle \texttt{new } t[c], \sigma\rangle & \rightsquigarrow & \langle p, \sigma[p \mapsto \texttt{new } t[c] : \text{ref}]\rangle \\
\langle \texttt{new hashtable}[t_0, t_1], \sigma\rangle & \rightsquigarrow & \langle p, \sigma[p \mapsto \texttt{new hashtable}[t_0, t_1] : \text{ref}]\rangle \\
\langle c_0 \text{ bop } c_1, \sigma\rangle & \rightsquigarrow & \langle [\![c_0 \text{ bop } c_1]\!]\sigma, \sigma\rangle \\
\langle lval = c, \sigma\rangle & \rightsquigarrow & \langle \texttt{skip}, \sigma[lval \mapsto c]\rangle \\
\langle \texttt{while}(e)\{s\}, \sigma\rangle & \rightsquigarrow & \langle \texttt{if}(e)\{s; \texttt{while}(e)\{s\}\}\texttt{else}\{\texttt{skip}\}, \sigma\rangle \\
\langle \texttt{skip}; s, \sigma\rangle & \rightsquigarrow & \langle s, \sigma\rangle \\
\langle \texttt{if(true)}\{s_0\}\texttt{else}\{s_1\}, \sigma\rangle & \rightsquigarrow & \langle s_0, \sigma\rangle \\
\langle \texttt{commute(false)}\{\{s_0\}\{s_1\}\}, \sigma\rangle & \rightsquigarrow & \langle s_0; s_1, \sigma\rangle
\end{array}$$

Fig. 1. Selection of reduction rules for the grammar.

We next define redexes in the usual way, with added redexes for our `commute` statements. We have included a selection of the redexes.

$$\begin{array}{rcl}
r & ::= & v \mid c_0[c_1] \mid \texttt{new } t[c] \mid \texttt{new hashtable}[t_0, t_1] \mid c_0 \text{ bop } c_1 \\
& & \mid lval = c \mid \texttt{while}(e)\{s\} \mid \texttt{skip}; s \mid \texttt{if(true)}\{s_0\}\texttt{else}\{s_1\} \\
& & \mid \ldots \boxed{\mid \texttt{commute(false)}\{\{s_0\}\{s_1\}\} \mid \texttt{commute(true)}\{\{s_1\}\{s_2\}\}}
\end{array}$$

Our language includes typical constructors, arrays, structures, as well as “builtin” ADTs. (Our theory applies to general ADTs, but for illustration here we use hashtables.) The new feature of the language is the `commute` statement, which we discuss in the next section. We next define contexts in the usual way, with contexts allowing for reductions inside commutativity conditions. The notation $H[r]$ means “H with the • replaced by r”.

$$\begin{array}{rcl}
H & ::= & \bullet \mid H; s \mid \text{uop } H \mid c \text{ bop } H \mid H \text{ bop } e \mid H?e_0 : e_1 \\
& & \mid H[e] \mid c[H] \mid H.\texttt{fieldname} \mid lval = H \mid t\ v = H \\
& & \mid \texttt{if}(H)\{s_0\}\texttt{else}\{s_1\} \mid \boxed{\texttt{commute}(H)\{\{s_0\}\{s_1\}\}}
\end{array}$$

The language of programs is that of all such $H[r]$, with the trivial program skip.

Small step semantics. Redex reduction rules are denoted $\langle r, \sigma\rangle \rightsquigarrow \langle r', \sigma'\rangle$ and several are given in Fig. 1 to give an idea of how they operate. Most of the reductions are straight-forward. The first two reductions merely look up the requested variable or memory in the state. The next two create a new data structure within the state. A boolean operation between constants gets reduced to the result of that operation on the constants. An assignment statement updates the state so that the value assigned to now refers to the constant being assigned. A `while` loop unrolls one iteration. Sequential composition with `skip` reduces to the next statement. An `if` statement predicated on a constant `true` reduces to the then branch (likewise, a constant false reduces to the else branch). Finally, a `commute` statement on a constant `false` reduces to a sequential composition. We defer the interesting case of a `commute` block on a constant `true` to Sec. 4. The full list of redexes and redex reductions can be found in our technical report [Chen et al. 2022a].

For simplicity, these are all atomically reducible expressions. Note that memory cannot be read and written in one atomic step. Intuitively, one can think of a redex as a simplest possible program, and the above reductions define the simplest possible steps that such a program can take. However, not all programs are redexes, so for each redex reduction Redex, we use the SS rule below:

$$\frac{}{\langle r, \sigma\rangle \rightsquigarrow \langle r', \sigma'\rangle}\ \text{Redex} \qquad \frac{\langle r, \sigma\rangle \rightsquigarrow \langle r', \sigma'\rangle}{\langle H[r], \sigma\rangle \rightsquigarrow \langle H[r'], \sigma'\rangle}\ \text{SS}$$

For a program $s = H[r]$, a small step reduction is an application of the SS rule above. We shall show later that every valid program may be decomposed into a unique redex and context, thus the redex reductions generalize into a small step semantics for the whole language. Note if s is a redex, we recover the redex rule by taking $H = \bullet$.

Suppose we are given program statements s_0 and s_1, and a predicate on states $\varphi_{s_1}^{s_0} : \Sigma \longrightarrow \mathbb{B}$.

Definition 3.1 (Sufficient Commutativity Condition). $\varphi_{s_1}^{s_0}$ is a sufficient commutativity condition for s_0 and s_1 when

$$\begin{aligned} \forall \sigma_0, \sigma_{f_0}, \sigma_{f_1} : \quad & \langle s_0; s_1, \sigma_0 \rangle \rightsquigarrow^* \langle \texttt{skip}, \sigma_{f_0} \rangle \wedge \\ & \langle s_1; s_0, \sigma_0 \rangle \rightsquigarrow^* \langle \texttt{skip}, \sigma_{f_1} \rangle \implies \\ & \varphi_{s_1}^{s_0}(\sigma_0) = \texttt{true} \implies \sigma_{f_0} = \sigma_{f_1} \end{aligned}$$

That is $\varphi_{s_1}^{s_0}$ tells when the order of execution has no effect on the final state.

4 SEMANTICS OF COMMUTE BLOCKS

In this section we define semantics for the language, which has been extended with the addition of `commute` statements. The full formalization of the semantics is given in Chen et al. [2022a], but we here summarize the key rules. We define three semantics, denoted $sem ::= seq \mid nd \mid par$ which represent (resp.) sequential, nondeterministic, and parallel semantics.

Sequential semantics. For sequential semantics $\rightsquigarrow_{seq}$ in the language extended with `commute` blocks, we add a simple reduction that treats `commute` as sequential composition:

$$\langle \texttt{commute(true)}\{\{s_0\},\{s_1\}\}, \sigma \rangle \quad \rightsquigarrow_{seq} \quad \langle s_0; s_1, \sigma \rangle$$

Nondeterministic semantics. We next provide another form of sequential (non-interleaved) semantics. The nondeterministic semantics $\rightsquigarrow_{nd}$ has two possible reductions that could take place when a commutativity condition has been reduced to `true`:

$$\begin{aligned} \langle \texttt{commute(true)}\{\{s_0\},\{s_1\}\}, \sigma \rangle \quad & \rightsquigarrow_{nd} \quad \langle s_0; s_1, \sigma \rangle \\ \langle \texttt{commute(true)}\{\{s_0\},\{s_1\}\}, \sigma \rangle \quad & \rightsquigarrow_{nd} \quad \langle s_1; s_0, \sigma \rangle \end{aligned}$$

The two rules for `commute(true)` apply to the same syntax. This allows for non-determinism when reducing this redex.

Concurrent semantics. The next semantics permits the bodies of (possibly nested) `commute` blocks to execute concurrently. To this end, we define a *configuration* $\mathfrak{C}$ that expresses the nested threading nature of the semantics. Formally, there are two constructors of a configuration:

$$\mathfrak{C} \quad ::= \quad \langle s, \sigma \rangle \quad \mid \quad \langle (\mathfrak{C}_0, \mathfrak{C}_1), s, \sigma \rangle$$

Configurations are either a top-most configuration (code and state), or a nested configuration, executing in the context of an outer remaining code s and outer state σ. For convenience, we also define $\mathfrak{C}.st$, the state of a configuration, by: $\langle s, \sigma \rangle.st = \sigma$ and $\langle (\mathfrak{C}_0, \mathfrak{C}_1), s, \sigma \rangle.st = \sigma$. The statements of a `commute`, which we will call *fragments*, execute code that naturally has access to variables defined in outer scopes. Thus when fragments step, they need to access to outer scopes. To allow for this, we define appending a state to a configuration ($\mathfrak{C} \oplus \sigma$) as performing the $\oplus$ operation on the outermost state in the configuration. Nested configurations arise when reaching a `commute` block. That is, start with the following redex and corresponding reduction:

$$\langle \texttt{commute(true)}\{\{s_0\},\{s_1\}\},\sigma\rangle \rightsquigarrow_{par} \langle(\langle s_0,\emptyset\rangle,\langle s_1,\emptyset\rangle),\texttt{skip},\sigma\rangle$$

where $\emptyset$ represents new local (and empty) scopes of variables for each fragment. This is then included in the *par* semantics with the following rule:

$$\frac{\langle r,\sigma\rangle \rightsquigarrow_{par} \langle(\langle s_0,\sigma_0\rangle,\langle s_1,\sigma_1\rangle),r',\sigma\rangle}{\langle H[r],\sigma\rangle \rightsquigarrow_{par} \langle(\langle s_0,\sigma_0\rangle,\langle s_1,\sigma_1\rangle),H[r'],\sigma\rangle}\ \text{Fork-Step}$$

Execution in the concurrent semantics involves (possibly recursively) descending into sub-components to find a leaf fragment that can take a step. This is done using the following left (or, similarly, right) projection rule:

$$\frac{\mathfrak{C}_0 \oplus \sigma \rightsquigarrow_{par} \mathfrak{C}_0' \oplus \sigma'}{\langle(\mathfrak{C}_0,\mathfrak{C}_1),s,\sigma\rangle \rightsquigarrow_{par} \langle(\mathfrak{C}_0',\mathfrak{C}_1),s,\sigma'\rangle}\ \text{Left-Proj} \qquad \frac{\mathfrak{C}_1 \oplus \sigma \rightsquigarrow_{par} \mathfrak{C}_1' \oplus \sigma'}{\langle(\mathfrak{C}_0,\mathfrak{C}_1),s,\sigma\rangle \rightsquigarrow_{par} \langle(\mathfrak{C}_0,\mathfrak{C}_1'),s,\sigma'\rangle}\ \text{Right-Proj}$$

When the two fragments of a `commute` have both reduced their statements to `skip`, the Join rule can be used. Updates to the inner scope are lost, but updates to the outer scope σ_2 are preserved:

$$\frac{}{\langle(\langle\texttt{skip},\sigma_0\rangle,\langle\texttt{skip},\sigma_1\rangle),s,\sigma_2\rangle \rightsquigarrow_{par} \langle s,\sigma_2\rangle}\ \text{Join}$$

Note that join is blocking; both threads must reach `skip` before s begins to reduce.

Labels. As we detail below, concurrent executions involve nested `commute` blocks' threads taking interleaved steps together. These steps will involve multiple occurrences of the Left-Proj and Right-Proj rules, used to drill down to specific nested `commute` block fragments, to allow them to take a step, and we now define labels to uniquely refer to these fragments. This will later be used in our correctness condition. A label $Lab : \{L_n, R_n \mid n \in \mathbb{N}\}^*$ is a sequence of choices to be made (corresponding to Left- or Right-Proj) to access the sub-configuration of a given `commute` block's transitions. Ignoring the subscripts, L indicates a Left-Proj and R indicates a Right-Proj. As an example, if there are no sequentially composed `commute` blocks, applying Proj-Left twice then Proj-Right once may yield a fragment label of $L_0L_0R_0$. Subscripts are then needed to cope with the fact that `commute` blocks may be sequentially composed, which naturally creates a forced order on the execution. Consider the labels given for the program below:

Example 4.1 (Fragment labels).

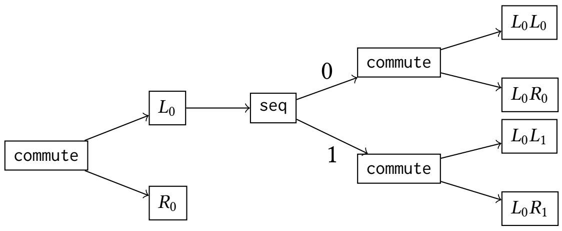
```
commute (...) {
L0: {
  commute (...) {
    L0L0: { ... }
    L0R0: { ... } }
  commute (...) {
    L0L1: { ... }
    L0R1: { ... } }
}
R0: { ... }
}
```

The first subscripts on L_0 and R_0 are simply 0 because the outer `commute` block is not sequentially composed. However, within L_0, there are sequentially composed `commute` blocks. Each one's fragments begin with the prefix L_0, but then the first `commute` block's fragments use subscripts 0, while the second `commute` block's fragments use subscripts 1. These labels uniquely identify fragments and we next use these labels in the definition of execution to identify the transitions associated with each fragment's statement.

Executions. An *execution* ϱ (w.r.t. a semantics *sem*) is a sequence of configurations with labeled transitions. That is, let $\mathfrak{C}$ be the set of configurations and T be the set of transitions. Then an execution ϱ is an element of the set $\mathfrak{C} \times (T \times \mathfrak{C})^*$, such that every transition is valid ($c_n \rightsquigarrow_{sem}^{t_{n+1}} c_{n+1}$). We assume all executions terminate and reach a final configuration $\langle \texttt{skip}, \sigma_f \rangle$ for some σ_f. A transition label ℓ has two parts: $Lab \times (\{\epsilon\} \cup (Var \times Val))$.

The first component is the fragment label, which as previously described, uniquely identifies some `commute` block's fragment. For example, during an execution of Ex. 4.1, the transitions that happen within each of the fragments would have the corresponding fragment label. The second part of an execution's transition label is either ϵ or $v \mapsto c$. The small steps only at most modify the value of one state element at a time, so this is sufficient to capture all effects. We write $\mathfrak{C} \Downarrow_{sem} \varrho$ if the initial configuration of ϱ is $\mathfrak{C}$, and let ϱ_f notate the final configuration of the execution ϱ.

Execution Trees. As we will discuss in detail later, it is natural to think of executions as being sequences of transitions made on nodes of a (labeled) tree, where branching arises due to the nesting and composition of `commute` blocks. Fragment labels may be associated with a path in such trees, with each element of the label, directing the path's choices at branches down the tree. For the above example, the tree might pictorially be represented as follows:



Above `seq` is used for sequential composition, with branches labeled 0 and 1, referring to the composition sequence. `commute` block fragments are identified by their label. Each execution step of this example can be thought of as addressing a specific node (*e.g.* L_0L_1) and having that node take a step.

5 A CONDITION FOR SAFE PARALLELIZATION

As discussed in Sec. 2.1, our programming model is distinct from, say, transactional memory in which users explicitly fork threads. Our goal is instead to safely interchange the parallel semantics for the sequential semantics, *i.e.*, show that $[\![s]\!]_{par} = [\![s]\!]_{seq}$. We now define this notion of safe parallelization and then introduce a new correctness condition—*scoped serializability*—which is distinct and slightly stronger than serializability. Intuitively this new correctness condition is still a relationship between interleaved executions and serial executions, however there are restrictions on which serial executions are permitted, based on the nested structure of commute blocks. We then show that, when combined with valid commutativity conditions, scoped-serializability achieves the goal.

5.1 Definitions and Properties

We work with a semantics based on *post state equivalence* rather than traces (projecting actions out of executions) because, although the latter works well for serializability, the former is simpler in our setting. Our notion of post-state equivalence also permits weaker notions of state equivalence (*i.e. observational* equivalence), and corresponds more closely to standard definitions of commutativity [Bansal et al. 2018; Dimitrov et al. 2014; Kim and Rinard 2011; Pîrlea et al. 2021]. Toward defining the goal, we begin with some definitions:

Definition 5.1 (Big-Step Semantics). For program s, let $[\![s]\!]_{sem} : \Sigma \to \mathbb{P}(\Sigma)$ be defined as:

$$[\![s]\!]_{sem} \;\equiv\; \lambda\, \sigma_0.\ \{\sigma_f \mid \exists \varrho : \langle s, \sigma_0 \rangle \Downarrow_{sem} \varrho \wedge \varrho_f.st = \sigma_f\}$$

Intuitively, the semantics $[\![s]\!]_{sem}$ is the set of final states of executions. Naturally, $[\![s_0]\!]_{sem_0}(\sigma_{0_0}) = [\![s_1]\!]_{sem_1}(\sigma_{0_1})$ when they are equal as sets, and we say that $[\![s_0]\!]_{sem_0}$ *is equivalent to* $[\![s_1]\!]_{sem_1}$ when, $\forall \sigma$, $[\![s_0]\!]_{sem_0}(\sigma) = [\![s_1]\!]_{sem_1}(\sigma)$. A statement s is *deterministic* in *sem* when $\forall \sigma : [\![s]\!]_{sem}(\sigma)$ is singleton. Finally, we say that s is *parallelizable* if $[\![s]\!]_{seq} = [\![s]\!]_{par}$, *i.e.*, defined in terms of end states of a program under different semantics. In the next subsection we will discuss a condition that ensures this goal.

There are a few key properties of the semantics, given below, that are later used in proving other lemmas or the main theorem. These properties are formalized and proved in [Chen et al. 2022a].

(1) Redex Uniqueness: Every nontrivial program has a unique redex and context. We use this property to be able to show that, in every non-final execution step of the program, there is always a unique successor (determinism of the sequential semantics).
(2) Conditional Determinism: If the redex is not a `commute(true)`, then the next step is unique. In other words, the only source of nondeterminism is from `commute` statements.
(3) Commutativity Correctness: If every `commute` block guard in s is a sufficient commutativity condition then s is deterministic in *nd*. This draws a connection between commutativity ensuring determinism.
(4) Inclusion: End states obtained through the *seq* semantics are a subset of those of *nd* are a subset of those of *par*. Due to this property, proving correctness of parallelization later will only require the opposite direction.

5.2 Serializability

Before we introduce our correctness condition, we would like to show that a traditional notion of serializability is insufficient for parallelization. Recall that a serializable execution is one for which there exists a semantically equivalent execution that has no interleaving of threads [Papadimitriou 1979]. For a definition of serializability in our notation, see our technical report [Chen et al. 2022a].

To see that serializability is insufficient, consider Ex. 2.5 from Sec. 2. Suppose that the sub-statement `x = x * 2` locks x so that it appears to execute atomically (we defer formalization of locks). An acceptable serial execution (omitting the interleaved configurations) of this program is:

$$\begin{array}{lcl}
(\epsilon, y \mapsto 0), & & \\
(\epsilon, x \mapsto 1), & & \\
(L_0R_0, x \mapsto x * 2), & \longleftarrow & \textit{Here, } x * 2 = 2 \\
(R_0, \epsilon), & \longleftarrow & \textit{Reads x's value as 2} \\
(R_0, y \mapsto 1), & & \\
(L_0L_0, x \mapsto 0) & &
\end{array}$$

After this execution we have $\sigma_f = \{x \mapsto 0, y \mapsto 1\}$. However, no pairwise re-ordering of the commutative blocks can result in such a state; if statement `if (x>=2) { y=1; }` executes first then y will be 0, and if it executes second, then y will still be 0.

In other words, serializability permits interleavings of outer, sibling `commute` statements with inner, nested `commute` statements. We thus seek a stronger correctness condition that constrains which serializations are permissible, with order requirements that account for which pairs of (nested, composed) fragments can be permuted.

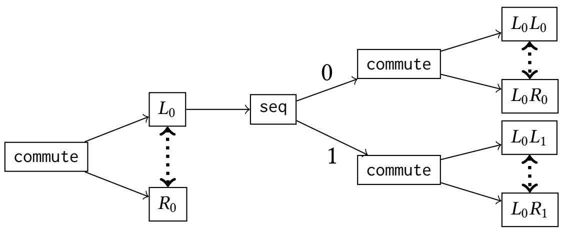


Fig. 2. Diagram showing the structure of Ex. 4.1. Dotted lines indicate which nodes can be re-ordered to obtain possible serialization orders. For example, a serial order with R_0 between L_0L_1 and L_0R_1 is *not* permitted.

5.3 Scoped Serializability

We now define *scoped* serializability, which restricts interleavings so that every execution in *par* is semantically equivalent to one in *nd*. Scoped serializability is still, like serializability, a property specifying that interleaved traces must be equivalent to some serial traces, but that set of possible serial traces is restricted by the nesting/composition of commute block pairs. For the aforementioned Ex. 2.5, we will only allow interleavings that are equivalent to one of the following cases:

$$(\mathfrak{f}_1;\mathfrak{f}_2);\mathfrak{f}_3,\quad (\mathfrak{f}_2;\mathfrak{f}_1);\mathfrak{f}_3,\quad \mathfrak{f}_3;(\mathfrak{f}_1;\mathfrak{f}_2),\quad\text{and}\quad \mathfrak{f}_3;(\mathfrak{f}_2;\mathfrak{f}_1).$$

In the first two cases, the nested first pair of fragments proceed in one or the other order and complete before the outer second fragment $\mathfrak{f}_3$ whereas, in the second two cases, $\mathfrak{f}_3$ occurs first.

We will refer to these possible serialization orders in terms of "swapping" fragments in the definitions below. To intuit the structure of permissible swaps in serial executions, it is helpful to use a tree, whose shape is determined by the nesting/composition of `commute` statements. Consider Fig. 2, showing the structure of the fragment nesting in Ex. 4.1. There is a node for every `commute` block in the program. The permissible serial orders are those obtained by swapping siblings (*i.e.* we may change the order of children of any `commute` node). Then, the set of permissible serial orders are those obtained from an in-order walk of a tree in which children have been swapped arbitrarily.

We formalize this by using fragment labels to track our current path in the tree. We begin with a single execution:

Definition 5.2 (Scoped serial execution). Execution ϱ is scoped serial if:

$$\begin{aligned}&\forall p \in \{L_n, R_n \mid n \in \mathbb{N}\}^* :\\ &((\forall \ell, \ell' \in \varrho : \quad \ell.\mathit{fr} \text{ has prefix } p \cdot L_k \wedge \ell'.\mathit{fr} \text{ has prefix } p \cdot R_k \implies \ell \text{ precedes } \ell' \text{ in } \varrho)\\ &\vee(\forall \ell, \ell' \in \varrho : \quad \ell.\mathit{fr} \text{ has prefix } p \cdot L_k \wedge \ell'.\mathit{fr} \text{ has prefix } p \cdot R_k \implies \ell' \text{ precedes } \ell \text{ in } \varrho))\end{aligned}$$

Above, $\ell.\mathit{fr}$ is the fragment label of ℓ. The key idea here is to identify the *scope* of commute fragments through labels, and then require a serializability condition for the L/R pair of the given scope. Consider, *e.g.*, a single `commute` block, possibly with children. For an execution to be scoped-serial, we require all of the transitions from one of the fragments to execute prior to all the transitions from its co-fragment (the other statement in the `commute`). Next, when there are nested `commute` blocks, the quantification over prefixes requires that we expect this same property to hold locally for all nested `commute` blocks. Without nesting, we recover the standard notion of serial.

The permissible scoped serial execution trees are defined by letting the fragment labels describe the path from the root to the given fragment. If two fragments have matching prefixes, then that prefix defines a common ancestor of the fragments. Since the ordering is decided prior to the quantification of transitions, we avoid the interleaving of nested fragments, as all nested fragments must adhere to the same ordering relative to an ancestor's sibling. Unsurprisingly, if an execution is scoped-serializable, then it is also serializable.

As a reminder, we do not restrict every execution to be non-interleaved. Rather, we only require that interleaved executions must be equivalent to some such (scoped) serial alternative. We define scoped serializability as such a weakening of scoped serial:

Definition 5.3 (Scoped Serializability). For execution ϱ such that $\mathfrak{C} \Downarrow_{par} \varrho$, ϱ is *scoped serializable*, wrt $\mathfrak{C}$ if and only if

$$\exists \varrho' : \mathfrak{C} \Downarrow_{par} \varrho' \wedge \varrho' \text{ is scoped serial} \wedge \varrho_f.st = \varrho'_f.st.$$

A program s is scoped serializable (or s-serializable) if every execution of s is scoped serializable.

5.4 Parallelizability

As our setting allows for a nested structure of `commute` statements, and also sequentially composed `commute` statements, these must be reflected in the parallel semantics. To ensure parallelizability, this structure must be preserved. While serializability is insufficient, we prove that scoped serializability is a sufficient condition.

Lemma 5.4. *S-Serializable(s)* $\implies [\![s]\!]_{nd} = [\![s]\!]_{par}$

By adding commutativity correctness, we conclude our main theorem:

Theorem 5.5 (Sufficient Condition for Parallelizability). *If every commutativity condition in s is valid and S-Serializable(s), then s is parallelizable.*

Proof. By induction on maximum fragment label length of transitions in *par* executions of s. Full proof in Chen et al. [2022a]. □

5.5 Enforcing Scoped Serializability

Note that any sufficient condition for pairwise serializability admits a condition for s-serializability by inductively applying it on each `commute` statement. Imagine that we have first enforced s-serializability for a sub-configuration. Then we can use a strategy for enforcing pairwise serializability as an inductive step for enforcing s-serializability in outer scopes. We thus now discuss techniques for enforcing pairwise serializability (and thus s-serializability).

Pattern 0: Naïve Locking. To our knowledge, prior works have not focused on pairwise serializability, perhaps because it has not been useful in other settings (*e.g.*, parallel composition). However, we may still take the naïve approach of locking from the first read/write on a common variable to the last. For only two blocks, this likely would not yield performance gains, as it leads to mutual exclusion, but in our nested `commute` setting, we can still aim for above 2× speedup due to the nesting of `commute` statements.

Indeed because of the structure of our setting we can *use and improve* the possible gains from pairwise serializability.

Pattern 1: Write/ReadOnly Intersection. In some cases, although both threads are reading/writing multiple variables, it may be that, among the *commonly* accessed variables, one thread is only reading from that set.

Let $wr(s)$ be the set of variables that s writes to, and $rd(s)$ be the set of variables that s reads from. For each instance of a builtin ADT, we include that builtin's variable name (accounting for aliasing) in both $wr(s)$ and $rd(s)$ if a method is invoked on it. Then let the set of conflicting variables be as follows:

$$con(s_0, s_1) \quad \equiv \quad [wr(s_0) \cap wr(s_1)] \cup [wr(s_0) \cap rd(s_1)] \cup [rd(s_0) \cap wr(s_1)]$$

Now let us consider the case where that $(wr(s_0) \cup rd(s_0)) \cap wr(s_1) = \emptyset$. Then $con(s_0, s_1) = wr(s_0) \cap rd(s_1)$. We may consider the symmetrical case (with s_0 and s_1 swapped) as well. We propose a simple program transformation can be used to enforce s-serializability in cases where this pattern holds. Suppose we are in the case where the commutativity condition holds (it may be necessary, *e.g.*, to copy the original code for when the condition does not hold). For each (conflicting) variable $v \in con(s_0, s_1)$,

(1) Replace all reads of v in s_1 to that of a new var v_0.
(2) Before execution of the `commute` block, add a statement $v_0 = v$.

Then all of s_1's reads will refer to the state prior to execution of s_0, regardless of whether s_0 executes first or not. So every interleaving will be as if it were the case $s_1; s_0$. One can think of the above process as snapshotting the state prior to potential modifications that may cause issues when interleaved.

Pattern 2: Narrowing mutual exclusion. Now we consider the more general case, when there are variables written by both threads. We can still use a single lock to enforce scoped serializability. (For a formal treatment of locking in our semantics, refer to Chen et al. [2022a].) If the threads lock on a common lock, they necessarily mutually exclude each other (and thus have no gains from parallelization), so the goal is to merely minimize the amount of time that the lock is held. Intuitively, this means we want all of the operations on conflicting variables as temporally close as possible in each thread. As is done elsewhere [Černý et al. 2013], this can be done via program transformations.

We describe a method for reordering statements in commutative blocks so that each block may be treated as an atomic point with respect to one another, *i.e.* the point when it acquires the singular lock. Once again, this is for the case where the commutativity condition holds. Suppose we are provided a data-flow graph for s_0 and s_1. Identify all instructions (nodes) that refer to the set of conflicting variables $con(s_0, s_1)$. Call this set $\mathcal{CI}_{s_0}$ (resp. $\mathcal{CI}_{s_1}$). Next, reorder instructions to be in the following form:

(1) Instructions that are only ancestors of nodes in $\mathcal{CI}_{s_0}$ (*i.e.*, instructions whose data only flows into the conflicting variables);
(2) $lock()$;
(3) Instructions that are in, or that are both ancestors and descendants of (possibly distinct) nodes in, $\mathcal{CI}_{s_0}$.
(4) Snapshot $con(s_0, s_1)$;
(5) $unlock()$;
(6) Instructions that are only descendants of nodes in $\mathcal{CI}_{s_0}$ (*i.e.*, instructions that are only dependent on the conflicting variables).
(7) Instructions that are neither ancestors nor descendants of nodes in $\mathcal{CI}_{s_0}$.

We use the same notion of snapshotting as in Pattern 1. This transformation is correct (assuming proper instruction ordering within each of the four groups), as it preserves all data dependencies, and it enforces pairwise serializability, as we may treat the fragments as executing sequentially in the order of reaching the lock. Indeed, we can re-order most transitions in an execution, as:

- The statements in (1) for s_0 cannot have a dependency to the statements in (3) for s_1, as otherwise they would cause writes in s_0 that are read in s_1, a contradiction as then they would be in $\mathcal{CI}_{s_0}$, and thus in (3).
- The statements in (6) for s_0 cannot have dependencies from the statements in (3) for s_1, as they have been modified to refer only to the local snapshot.
- All data dependencies across the blocks must involve a statement from (3). So, for example, (1) for s_1 does not depend on (6) for s_0.

Thus, we can reorder all transitions in (1) in the second thread to after (3) of the first, and all statements in (6) and (7) of the first thread to before the (3) of the second, obtaining a serial execution. In other words, the blocks may have some instructions that have read/write conflicts. We first perform non-conflicting dependencies in parallel, so that we can minimize the time where the conflict is locked. Then we synchronously perform the critical section, and finally we perform any remaining actions in parallel. Thus we effectively minimize, as far as practical, the length of the critical section, without violating any dataflow constraints.

For load balancing reasons, it may be practical to put (7) at the beginning of the block when reordering s_1. If fact, even when locking is non-trivial, using this technique can ensure correctness at minimal performance cost with proper load balancing.

Toward implementing enforcement. As noted above, enforcement can be implemented using existing techniques for lock synthesis. To implement pattern 0, one must employ a static analysis to determine (perhaps an over-approximation of) the read and write sets of each fragment, and introduce lock acquisition before each access of a common variable, as well as lock release at the end of the fragment. To implement pattern 1, one must once again determine the read and write sets of each fragment, in order to see if the transformation may be applied. Then, the snapshotting may be implemented as a syntactic rewrite rule. Pattern 2 requires—in addition to the read/write sets—a dataflow analysis, perhaps with an alias analysis for references. Then one must reorder instructions and introduce locks appropriately.

Much of the above analyses are commonplace within compilers. As discussed in Sec. 7, Veracity is currently implemented as an interpreter and we defer challenges involved in back-end compilation of `commute` blocks to future work. We believe that the back-end stage of a compiler— where such static analyses are widespread—is a better setting for implementing enforcement. Enforcing serializability involves small-step atomic steps (*e.g.*, lock acquisition, memory read/write, etc) and thus in the back-end, where each IR instruction represents fewer steps than an original program statement, it may be more effective and easier to perform the instruction reordering of pattern 2.

6 VERIFICATION AND INFERENCE OF COMMUTATIVITY CONDITIONS

For the parallelized version of the program to be correct, the commutativity conditions must be sound. Although there are some recent techniques for verifying or inferring commutativity conditions, these techniques do not quite fit our setting, as discussed in Sec. 9. For example, the abstraction-refinement technique of Bansal et al. [2018] applies to logically specified abstract datatypes (ADTs) and not directly to code. In this section we will describe an embedding of programs with `commute` statements (which may program context, involve nested `commute` statements, builtin hashtables, loops, etc.) into such logical specifications, thus enabling us to use such abstraction-refinement to synthesize conditions that can be used in our `commute` blocks, or to verify manually provided conditions.

Since we target the abstraction-refinement synthesis algorithm of Bansal et al. [2018], we briefly recall the input to this algorithm, which is a logically specified ADT O, defined as:

Definition 6.1 (Logical ADT specification).

$$O = \left\{ \begin{array}{llllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & : & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

These objects include state defined as a list of variables, an equality relation on states, a finite list of method *signatures* (without implementations), and logical specifications for each method. The *commutativity condition synthesis problem* is then, for a given pair of method signatures $m(\bar{x})$: Meth and $n(\bar{y})$: Meth, to find a logical condition φ in terms of the object's state, and the arguments $\bar{x}$ and

$\bar{y}$, such that m and n commute according to the method specifications from every state satisfying φ. Bansal et al. [2018] give a method for doing so based on abstraction-refinement.

6.1 Embedding `commute` Blocks as Logical ADT Specifications

The challenge we now address is how to translate programs with `commute` blocks into logical object specifications. For a given `commute` block with statements s_1 and s_2 and their program context E, our translation Tr is defined below and yields an object O with two methods m_{s_1} and m_{s_2}.

Definition 6.2 (Embedding Tr). For `commute` fragment statements s_1 and s_2 in program context E, $Tr(E, \texttt{commute}\ s_1\ s_2) \equiv O$ where

- $\underline{O.\mathsf{state}}$: We create an object variable for every global or local variable in environment E (that is visible to the scope containing s_1 and s_2).
- $\underline{O.\mathsf{eq}}$: o_1 is equal to o_2 provided that they agree on the values of all the variables in O.state.
- $\underline{O.\mathsf{methods}}$: Nullary methods m_{s_1} and m_{s_2}.
- $\underline{O.\mathsf{spec}}$: Associating these nullary methods with true preconditions and post-conditions given by specOf in Fig. 3, *i.e.*, $[m_{s_1} \mapsto (\mathsf{true}, \mathsf{specOf}(s_1)), m_{s_2} \mapsto (\mathsf{true}, \mathsf{specOf}(s_2))]$

Intuitively, the idea is to embed the current program context (including variables in scope and builtins) as O's state, and $m_{s_1}()$ and $m_{s_2}()$ as nullary methods corresponding to s_1 and s_2, respectively, whose pre/post specifications describe the changes to that context.

The last spec component translates `commute` fragment source code to a logical method specification. The overall shape of the translation is akin to verification condition generation, introducing indexed let-binding v_i each time program variable v is mutated and, ultimately, constraining the value of v in the postcondition (with respect to the 0th bindings) in terms of the most recent let-binding. (Programs in our language are total and thus, preconditions are simply true.) The output of our translation is a logical formula, written in SMTLIBv2 [Barrett et al. 2010]:

$$\varphi \quad ::= \quad \varphi \vee \varphi \mid \neg\varphi \mid \underline{ite}\ \varphi\ \varphi\ \varphi \mid \underline{f}\ \bar{\varphi} \mid \underline{let}\ v\ \varphi\ \underline{in}\ \varphi \mid \underline{exists}\ v\ \underline{in}\ \varphi \mid \varphi \underline{\otimes} \varphi \mid \underline{true}$$

Greek letters are used for logical terms. Φ will be used for the type of terms. A formula φ is a well-sorted term with sort boolean. The terms we use include $\underline{let}$, $\underline{ite}$, and $\underline{exists}$, underlining formulae constructors to distinguish from our language and algorithm.

The translation algorithm specOf is given in Fig. 3. First we convert any sequential composition of statements into a list of statements (s_{flat}), where `skip` is the empty list. We proceed in an SSA-like manner, creating an initial mapping id0 of indices for each variable and construct a $\underline{let}$ term binding v_0 with v for each variable that is defined in the context E and accessed in the body of the `commute` fragment. This binding will let us ultimately define the final value of v, denoted v_{new}, in terms of its initial value v. We then begin with translation Tr_S on s_{flat}. A call to $(Tr_S\ s\ \mathsf{id})$ recursively constructs the translation to a logical expression representing the postcondition of the block. Within a given call to $(Tr_S\ s\ \mathsf{id})$, if s is a final skip statement (i.e., if s is the empty list), then an innermost formula is constructed that relates each v_{new} to the most recent binding for v in the ID map. In the assignment case, e is first translated, which may accumulate additional variable bindings, namely when e contains a call to a builtin ADT. Then a fresh binding for variable v is created with the new value. In the `if/else` case, we recursively translate the condition expression e to obtain a logical condition φ_e. We then represent branching via $\underline{ite}$, with condition φ_e and the remainder of the block tl appended to both the branches, recursively translated. In the case of a nested `commute` block with fragments s_1' and s_2', our semantics permits a sequential interpretation of `commute`, so we arbitrarily choose $s_1'; s_2'$ and recursively translate this sequential composition. Loops are supported by instrumenting loop summaries with `havoc` and `assume`, as detailed at the

```
type idmap = $Var \rightarrow \mathbb{N}$

let rec $Tr_E$ ($e$ : expr) (id : idmap) :
            idmap $\times\ \Phi \times$ (id $\times\ \Phi$) list =
  match $e$ with
  | $v \rightarrow$ (id, $\underline{v_{(\mathrm{id}\ v)}}$, [])
  | $e' \otimes e'' \rightarrow$
      let (id$'$, $\varphi_L$, binds$_L$) = $Tr_E\ e'$ id in
      let (id$''$, $\varphi_R$, binds$_R$) = $Tr_E\ e''$ id$'$ in
      (id$''$, $\varphi_L \underline{\otimes} \varphi_R$, binds$_L$ ++ binds$_R$)
  ...
  | builtin adt $v_{adt}$ m $\bar{e} \rightarrow$
      let (id_ret :: $\overline{\varphi_{args}}$, $\overline{\text{binds}}$) =
          map ($Tr_E \bullet$ id) $\bar{e}$ in
      let adt_id = id $v_{adt}$ in
      let ($\varphi$, spec_binds) =
          $\boxed{\text{spec adt } v_{adt} \text{ m } \overline{\varphi_{args}} \text{ adt\_id}}$ in
      (id[id$_{\mathrm{ret}} \mapsto$ id[id$_{\mathrm{ret}}$] + 1],
       $\varphi$, $\overline{\text{binds}}$ ++ spec_binds)

let specOf ($s$ : stmt) : $\Phi$ =
  let $s_{flat}$ = flatten $s$ in
  let $s'$ = summarizeLoops($s_{flat}$) in
  let id0 = [$v \mapsto 0 \mid v \in Vars(s')$] in
  let inits = $\{(v_0^i\ v^i) \mid v^i \in Var(s')\}$ in
  $\underline{let}$ inits $\underline{in}$ $Tr_S\ s_{flat}$ id0
```

```
let rec $Tr_S$ ($stmts$ : stmt list)
           (id : idmap) : $\Phi$ =
  match $stmts$ with
  | [] $\rightarrow \bigwedge_{v\ \in\ \mathrm{dom(id)}} v_{new} = v_{(\mathrm{id}\ v)}$
  | ($v = e$) :: tl $\rightarrow$
      let (id$'$, $\varphi_e$, bindings) = $Tr_E\ e$ id in
      $\underline{let}$ $\underline{\text{bindings}}$, $\underline{v_{(\mathrm{id}'\ v+1)}\ \varphi_e}$ $\underline{in}$
        $Tr_S$ tl (id$'$[$v \mapsto$ (id$'$ $v$) + 1])
  | (if $e$ then $s_1$ else $s_2$) :: tl $\rightarrow$
      let (id$'$, $\varphi_e$, bindings) = $Tr_E\ e$ id in
      $\underline{let}$ $\underline{\text{bindings}}$ $\underline{in}$
        ($\underline{ite}$ $\varphi_e$ ($Tr_S$ ($s_1$ ++ tl) id$'$)
                ($Tr_S$ ($s_2$ ++ tl) id$'$))
  | (commute $e$ $s_1$ $s_2$) :: tl $\rightarrow$
      $\boxed{Tr_S \text{ id } (s_1 +\!\!+\ s_2 +\!\!+ \text{ tl})}$
  | (assume $e$) :: tl $\rightarrow$
      let (_, $\varphi$, _) = $Tr_E\ e$ id in
      $\varphi \wedge Tr_S$ tl id
  | (havoc $hid$) :: tl $\rightarrow$
      let $hid_{havoc}$ = freshvar() in
      $\underline{exists}$ $hid_{havoc}$ $\underline{in}$
        $\underline{let}$ $\underline{hid_{(\mathrm{id}\ hid+1)}\ hid_{havoc}}$ $\underline{in}$
          $Tr_S$ tl id[$hid \mapsto$ (id $hid$) + 1]
```

Fig. 3. Reducing a `commute` fragment's statement s to a logical method specification.

end of this subsection. As usual, `havoc` existentially quantifies a new value and `assume` introduces a new specified constraint, both with respect to the current binding IDs.

The translation of an expression $Tr\ e$ id returns a triple (an updated id map, a translation of e into logic, and possibly new variable bindings), and is mostly straightforward, except in the case of a builtin ADT. This case requires us to instantiate the ADT's logical specification for the current calling context, so we first recursively translate each of the arguments in $\bar{e}$, obtaining a set of local bindings $\overline{\text{binds}}$. We then use a helper method spec (omitted) to instantiate the arguments into the ADT's postcondition, and construct an SMT expression φ describing the return value and additional bindings for the return value(s) (on a logical level, we may consider the ADT's updated state to be a return value, *e.g.* a hashtable has a set of keys, an integer size, and a finite map of keys to values). In this way, the specification internals are "ferried" through the nested $\underline{let}$ statements and constraints in the final embedding. The translation of builtins is illustrated (among other things) in the following example:

Example 6.3 (Illustrating the embedding).

```
commute (c>0) {
  s₁: { c = c+1; }
  s₂: { if (c>0) { tbl[c] = 5; c = c-1; } }
  }
```

We label the fragments above by their statements. We describe how specOf converts s_2 to a formula. A map id0=$[c \mapsto 0, tbl \mapsto 0]$ is created, as well as let-bindings $((c_0\ c)\ (tbl_0\ tbl))$. The first Tr_S call is for the if-then-else, and it recursively translates the condition: Tr_E `(c>0)` id0, which takes the $c \otimes 0$ case, recursively translates c to $c_{\text{id0 c}}$, and returns (id0, $(\underline{gt}\ c_0\ 0)$, []). Now in the then branch, Tr_S recursively translates the sequential composition. The assignment is syntactic sugar for `ht_put(tbl, c, 5)`. Here the recursive translation of the first argument translates argument `c` to c_0, and then a call is made to spec `ht` tbl `put` c_0 `5 (id` tbl`)`. This helper function instantiates the specification to `put` as follows:

$$
\begin{array}{ll}
\varphi_{put} \equiv & (\underline{ite}\ (mem\ c_0\ tblK_0)\ (\neq 5\ (select\ tblM_0\ c_0))\ \underline{true}) \\
binds \equiv & \left(\begin{array}{lll}
(tblK_1(\underline{ite} & (mem\ c_0\ tblK_0)\ tblK_0\ (ins\ c_0\ tblK_0)) \\
(tlbS_1(\underline{ite} & (mem\ c_0\ tblK_0)\ tlbS_0\ (tlbS_0 + 1)) \\
(tblM_1(\underline{ite} & (mem\ c_0\ tblK_0) \\
 & (\underline{ite}\ ((select\ tblM_0\ c_0) = 5) \quad tblM_0 \quad (store\ tblM_0\ c_0\ 5)) \\
 & (store\ tblM_0\ c_0\ 5)))
\end{array}\right.
\end{array}
$$

The above bindings involve three variables $tblK_1, tblS_1, tblM_1$ representing, resp., the set of keys, the size, and the finite map of the hashtable after the get operation, with respect to the corresponding previous variables $tblK_0, tblS_0, tblM_0$. These variables model the standard hashtable semantics. Returning to the Tr_S of the assignment statement, the new value of the state is bound to, then the increment of `c` is translated to $\lambda\ k.\underline{let}\ (c_1(c_0 - 1))\ \underline{in}\ k$ by a recursive call. After the assignment a skip will be introduced, so an innermost constraint of the form $(c_{new} = c_1 \wedge tblK_{new} = tblK_1 \wedge ...)$ is constructed. Returning to the original Tr call on the if-then-else, the full translation is assembled:

$$
\begin{array}{l}
\underline{let}\ ((c_0\ c) \ldots (tblK_0\ tblK) \ldots)\underline{in}\ \underline{true} \wedge \\
\quad (\underline{ite}\ (c_0 > 0) \\
\qquad\quad (\underline{let}\ binds\ \underline{in}\ (\underline{let}\ (b_1\ \varphi_{put})\ \underline{in} \\
\qquad\qquad (\underline{let}\ (c_1(c_0 - 1))\ \underline{in} \\
\qquad\qquad\quad (c_{new} = c_1 \wedge tblK_{new} = tblK_1 \wedge \ldots))
\end{array}
$$

Loop summaries. Loops are supported through summaries that can be computed by other techniques (*e.g.* [Ernst 2020, 2022; Kroening et al. 2008; Silverman and Kincaid 2019; Xie et al. 2017]). We calculated loop summaries using the Korn tool [Ernst 2022]. Details can be found in the appendix of Chen et al. [2022a]. In some cases, Korn was not precise enough, so we manually strengthened the summaries, but we believe this work could be fully automated, and that it is orthogonal to our main contributions. We then replace loops in `commute` blocks with their summaries using `havoc` and `assume` statements. Note that in general, approximation of loops may not be precise enough to ensure commutativity [Koskinen and Bansal 2021], however, this is not a problem in our setting because our commute conditions are verified/inferred with the requirement that post-states be exactly equal. Thus, we would fail to verify/infer any conditions in the presence of insufficiently precise loop summaries.

6.2 Verifying and Inferring Commutativity Conditions

Our embedding allows us to use SMT tools to verify commutativity conditions and the abstraction-refinement algorithm of Bansal et al. [2018] to synthesize them.

Verification. To verify a provided commutativity condition φ, we construct a series of constraints, modeling the execution of $o.m_{s_1}$ and $o.m_{s_2}$ in either order denoted $o.m_{s_1} \bowtie o.m_{s_2}$ and assert the negation of $\varphi \implies o.m_{s_1} \bowtie o.m_{s_2}$. The correctness of our technique is based on the soundness of our embedding, as described in the following theorem:

Theorem 6.4 (Embedding soundness). *For every* `commute` $s_1\ s_2$, *let* $o = Tr(E,$ `commute` $s_1\ s_2)$. *Then if valid(*$\varphi \implies o.m_{s_1} \bowtie o.m_{s_2}$*), then* φ *is a valid commutativity condition for* `commute` $s_1\ s_2$.

Proof Sketch. We first assume a standard semantics $[\![\varphi]\!]^{\mathsf{smt}}$ for SMT expressions. The crux of the proof is to correlate the ADT state $o.\mathsf{state}$ and transition functions $o.m_1, o.m_2$ on $o.\mathsf{state}$ with the Veracity statements s_1 , s_2 as functions on Veracity state σ. Thus, if the SMT transition functions commute, then we can conclude that the Veracity statements commute (Def. 3.1). Statements in the Veracity semantics, *e.g.*, $\langle \texttt{x:=x+1}, \sigma\rangle \rightsquigarrow_{seq} \langle \texttt{skip}, \sigma[x \mapsto \sigma(x) + 1]\rangle$ are transitions and translated to SMT formulae of the form $x_{new} = x + 1$, necessitating a relational SMT semantics $[\![\varphi]\!]^{\mathsf{smt}}_R$ over two vocabularies of variables: the original and those subscripted with new. For example, $[\![x_{new} = x + 1]\!]^{\mathsf{smt}}_R$ represents a pair of states (encoded as a single state) where x in the second state is one greater than x in the first state.

We use structural induction on the expressions/statements, correlating SMTLIB semantics with Veracity semantics. As part of the induction, we maintain the invariant that the value of each Veracity variable is equivalent to the value of a corresponding SSA-style indexed SMTLIB variable, according to an accumulated “id-map” between variables and index numbers.

For expressions, we show that if a Veracity variable is equal to its corresponding SMT indexed variable, then the Veracity expression’s evaluation is equivalent to the corresponding SMT expression’s evaluation. This holds because of the similarity between Veracity semantics and SMT semantics. For statements, the id-map could be updated. We show that, if all Veracity variables in a state are equal to their corresponding SMT indexed variables before the statement, then after the statement, Veracity variables are equivalent to corresponding SMT indexed variables in a possibly updated id-map. For example, with an assignment statement, the update id-map will have a fresh indexed variable for the lvalue. We proceed by structural induction.

Since the Veracity-SMT equivalence is maintained, the final indexed variables in the innermost SMT constraints for $o.m_1$ and $o.m_2$ correspond to relationships between the initial and final values of the variables in the Veracity statements s_1 and s_2 (resp.). Thus, validity of ADT commutativity ($o.m_1 \bowtie o.m_2$) is equivalent to commutativity of Veracity statements. We also rely on the soundness of specifications provided for built-in data types. □

Inference. Often it is even more convenient for commute conditions to be automatically synthesized. To this end our embedding also allows us to use abstraction-refinement [Bansal et al. 2018], which takes as input the kinds of objects that we construct. As discussed in Sec. 7 we re-implemented this algorithm, adding some improvements to that algorithm (*e.g.* in the predicate selection process) that are beyond the scope of this paper. The resulting commutativity conditions we infer are sound, again due to the soundness of our embedding (Thm. 6.4).

Completeness. Commute conditions are akin to Hoare logic preconditions and, as such, they are subject to expressibility concerns of the assertion language. However, commute conditions need not be complete for two reasons: (i) as opposed to preconditions, when commute conditions do not hold we default to sequential execution, rather than undefined behavior and (ii) consequently, it is always fine to over-approximate commute conditions by assuming that fragments only commute in cases where they have been proved to commute. As we show in Sec. 8, in our 30 benchmarks, in every case either a precise commute condition or a sensible sound approximation could be used.

7 IMPLEMENTATION: VERACITY

Our implementation comprises an interpreter capable of parallel execution (and sequential execution for testing), as well as analyses for verification and synthesis of commute conditions (Sec 6). For simplicity we focused on building a concurrent interpreter, deferring backend compilation matters to future work. The source code for our implementation is publicly available[4].

Interpreter. Threads are implemented with Multicore OCaml, as OCaml does not natively support true multi-threading. Considerations are made in the interpreter to minimize shared memory between threads. When a commute statement is reached and its guard evaluates to true, threads are spawned for each parallel block. Using thread pools did not substantially improve performance over spawning as needed. Each thread is given a copy of the global environment and call stack, permitting thread-local method calls. Sibling threads' environments will only share references to variables in shared scope, namely global variables, arguments of the current method, and variables defined within the method in scope of the current block. When a commute statement's guard evaluates to false, the blocks are executed in sequence. The Veracity executable includes a flag to force all commute guards to evaluate to false, allowing us to compare parallel-vs-sequential execution times.

A key benefit of commute blocks is that hashtable operations can be implemented with a linearizable hashtable (*e.g.* [Liu et al. 2014; Purcell and Harris 2005]) in the concurrent semantics. To confirm the benefit, we built a foreign function interface to the high-performance NSDI'13/EuroSys'14 concurrent hashtable [Fan et al. 2013; Li et al. 2014] called libcuckoo [libcuckoo 2013].

There are minor differences and syntactic sugar between the formal semantics and the implemented Veracity language. (See our technical report [Chen et al. 2022a] for details.)

Verifying and inferring commutativity conditions. We implemented the embedding described in Sec. 6. This translation then generates a specification for an object O, with commute blocks embedded as logical specifications, given as OCaml SMTLIBv2 expressions. For verification and abstraction-refinement inference, we implemented Servois2 [Servois2 2022] a new version of Servois [Servois 2018] as an OCaml library/module for a few reasons:

- Improve performance and closer integration. O is now an OCaml type, constructed by Veracity's compiler and passed to our new abstraction-refinement library.
- Support more/other SMT solvers. We used CVC5 [Barbosa et al. 2022] for all of our experiments, but can also use CVC4, Z3, and plug in other solvers.
- Improve/tune the predicate generation and predicate choice techniques.
- Add support for verifying user-provided commutativity conditions.

To improve abstraction-refinement we also provide possible terms from which candidate predicates are constructed. We extract these terms from the syntax of the Veracity program, as well as practically selected constants which helped reduce the size of the synthesized conditions. We also extract terms from the builtin ADT specifications. Finally, after commutativity conditions are synthesized, we then reverse translate back to Veracity expressions.

8 EVALUATION

We evaluated our work against four goals:

(1) Determine whether commute conditions could be automatically generated (Sec. 6);
(2) Determine whether scoped serializability can be enforced using our locking patterns (Sec. 5.5);
(3) Confirm that speedups can be seen when the duration of sufficiently independent commute fragments increase; and
(4) Whether, despite introducing a new programming paradigm, existing applications could be adapted and exploit commute statements.

Experimental setup. All experiments below were run on a machine with a machine with an AMD EPYC 7452 32-Core CPU, 125GB RAM, Ubuntu 20.04, and Multicore OCaml 4.12.0.

Benchmarks. We created a total of 30 example programs with commute blocks that use a variety of program features including linear arithmetic, nonlinear arithmetic, arrays, builtin hashtables, nested commute blocks, loops and some procedures. The complete set of benchmarks are available in the benchmarks directory of the Veracity repository[6].

Table 1. Inference and/or verification of commute conditions. Group 1 contains benchmarks with inferable conditions. Group 2 is a subset of cases from Group 1 where manually providing a condition was desirable. Group 3 contains programs with structures that we could not translate into logic.

Correctness of Commute Conditions

Group 1: Automatically Inferred Commute Conditions. All benchmarks, except those below in group (3).

Program	Time (s)	Inferred Conditions
`array-disjoint`	0.63	`i != j && x != y || x == y`
`array1`	0.75	`1 != r[0] && r[0] + 1 != y && r[0] <= 1 || r[0] + 1 == y && r[0] <= 1`
`array2`	1.11	`0 > a[0] && 1 != x || 1 == x`
`array3`	1.13	`d != e && a != b || a == b`
`calc`	✗	`1 == y && 0 != y && 1 > c && 1 != c || ... || 1 == c`
`conditional`	0.18	`x > 0`
`counter`	0.20	`0 != c`
`dict`	3.82	`i != r && c + x != y || c + x == y`
`dot-product`	0.24	`true`
`even-odd`	1.18	`x % 2 == x + y && 0 != y || 0 == y`
`ht-add-put`	2.24	`tbl[z] == u + 1 && u + 1 != z`
`ht-cond-mem-get`	1.54	`tbl[x] == tbl[z] && x != z || x == z`
`ht-cond-size-get`	0.83	`ht_size(tbl) <= 0 && 0 != z || 0 == z`
`ht-simple`	30.64	`x + a != z && 3 == tbl[z] && y != z`
`linear-bool`	3.62	`0 <= y && 3 == x && 2 != x && 1 != x && x > 0 && 0 != x || 0 > y + 3 * x && 2 == x && 1 != x && x > 0 && 0 != x`
`linear-cond`	2.65	`2 <= y && 2 != y && 1 != y || ... || 1 == y`
`linear`	0.25	`true`
`loop-amt`	✗	`0 == i && amt == i_pre && ctr - 1 > i_pre && i_pre <= amt && 0 != i_pre && i_pre <= ctr && amt != amt_pre && ctr - 1 > amt_pre && amt_pre <= amt && 0 != amt_pre && amt_pre <= ctr && ctr - 1 != 1 && 1 != ctr && 1 != amt && 1 == ctr + amt || ... || amt == i && 1 == ctr && 1 != amt && 1 == ctr + amt`
`loop-disjoint`	0.02	`true`
`loop-inter`	4.63	`0 == x && 0 != y || 0 == y`
`loop-simple`	0.06	`true`
`matrix`	0.71	`0 == y`
`nested-counter`	6.25	`0 != c && c != t || c == t; c != x && c <= x && 1 != x && t == x || ... || 1 == x && t == x`
`nested`	0.25	`true; 0 == x`
`nonlinear`	1.42	`0 == y`
`simple`	3.49	`a <= c && a != b && a != c || a == b && a != c`

Group 2: Automatically Verified Commute Conditions. Manually provided conditions for verification.

Program	Time (s)	Verified?	Complete?	Provided Condition
`array1`	0.02	✓	✓	`r[0] <= 0 || r[0] == 1 && y == 2`
`calc`	0.07	✓	?	`c > 0`
`counter`	0.02	✗	–	`true`
`even-odd`	0.04	✓	✓	`y % 2 == 0`
`linear-bool`	0.02	✓	✓	`y < 0 - 3 * x && x == 2 || y >= 0 && x == 3`
`linear-cond`	0.02	✓	✓	`y > 0 || 0 == y && x + 2 == z`
`loop-amt`	0.04	✓	✗	`i % 2 == 0 && (ctr > 0 && amt > 0 || ctr <= 0 && amt < -ctr)`
`nested-counter`	0.05	✓	✓	First commute block: `0 != c && c != t || c == t`
(cont.)		✓	✗	Second commute block: `x == t && (x > c || x == c && x > 1)`
`simple`	0.04	✓	✗	`c > a`

Group 3: Unverified Examples. Including case studies.

Program	Manual commute condition
`crowdfund`	`true`
`dihedral`	`(!s1 && !s2) || (s1 && !s2 && (r2 == 0 || (n % 2 == 0 && r2 == n / 2))) || (!s1 && s2 && (r1 == 0 || (n % 2 == 0 && r1 == n / 2))) || (s1 && s2 && r1 == r2)`
`filesystem`	`fname1 != fname2`
`ht-fizz-buzz`	`true; true`

Inferring/verifying `commute` *conditions.* Table 1 shows the results of inference/verification among the benchmarks, classified into 3 groups. We first applied our procedure to infer commute conditions for all benchmarks. Benchmarks for which inference succeeded are given in Group 1. A few benchmarks featured complicated loops or interprocedural calls that we could not translate into logic. For these cases, we manually provide conditions, as listed in Group 3.

For those benchmarks in Group 1, we list the inference computation time and the generated condition. Cases for which inference timed out (at 120 seconds) are marked with ✗. Cases featuring multiple `commute` blocks list the conditions generated for each block. In most benchmarks, we successfully inferred concise and useful commute conditions within a few seconds. Very long generated conditions are abridged; the unabridged results are provided in [Chen et al. 2022a].

After inferring conditions for the Group 1 benchmarks, we performed condition verification for a subset of these cases, applying our Sec. 6 embedding to manually provided commutativity conditions. These benchmarks were selected in order to (i) manually simplify a needlessly large generated condition containing redundant terms; (ii) manually adjust a correct, but obviously incomplete, generated condition to become complete; or (iii) demonstrate the result of attempting to verify an intentionally false condition. These cases are reported in Group 2 of Table 1, along with the time, verification result, and completeness result. Verification or rejection took fractions of a second in all cases, and (in)completeness was reported in all but one case.

Enforcing scoped serializability. In Sec. 5.5 we describe how to enforce scoped serializability. We applied these methodologies to the benchmarks, and believe it could be automated in the future. As discussed in Sec. 5.5, enforcement is likely better implemented as part of the back-end IR of a compiler for `commute` blocks, which we defer to future work. For many benchmarks (*e.g.* `counter`), a single lock could be naïvely synthesized to protect against updates to a shared set of variables (Pattern 0). We used our snapshot approach (Pattern 1) on `array3` and `simple` to avoid the need for locks at all. Other programs (`conditional`, `matrix`, and `loop-disjoint`) did not need any locks. For `dict`, the only conflict is a single access to an already linearizable concurrent hashtable. `dihedral` and `ht-fizz-buzz` are serializable due to loop disjointness. All other benchmarks had manual locks added naively around the critical sections.

Speedup. We next sought to confirm that, depending on the size of the computation, parallelization offers a speedup over sequential execution. We executed the programs and collected execution time measurements. Our benchmarks all involve some pure computation(s) of parametric size N. We varied this problem size, and recorded the speedup ratio (sequential execution time divided by parallel execution time).

We took the geometric mean of the speedup ratios across 4 trials and plotted it against problem size. The graph is shown in Fig. 4. On the x-axis we increased the problem size N exponentially. Many benchmarks had very similar performance. For legibility of the overall plot, we grouped and colored those with similar performance characteristics. (The group labels do not imply any particular semantic connection between the benchmarks.) In most of the benchmarks, speedups were observed and asymptotically approached the expected 2× mark. Most of the payoff occurred when the problem size was above 1,000,000, with some payoff occurring when the problem size was above 1,000. In `dihedral`, the two threads can have differently sized workloads.

For the four benchmarks involving loops, we initially found no speedup. While this is expected for `loop-inter` (because it only commutes in trivial cases) and for `loop-simple` (because threads had to hold the lock for the entire loop), we discovered that the interpreter exhibited contention when concurrently accessing the shared (and pure) context. Consequently, the `loop-disjoint` and `loop-amt` examples also had no speedup. This could be a subtle bug in Multicore OCaml. To confirm that this is merely a bug and that our approach should yield speedup, we edited `loop-disjoint`

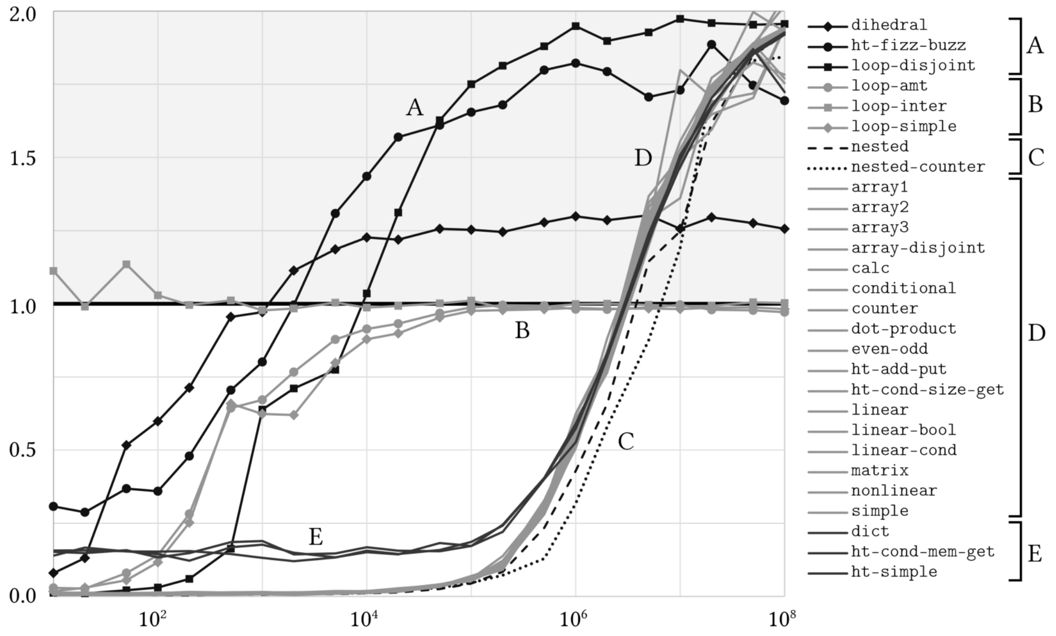


Fig. 4. Parallel-to-sequential speedup for benchmark programs versus computation size. The computation size x-axis is logarithmically scaled. Each trendline is for a different program. Many benchmarks had similar performance characteristics. For legibility of the overall plot, we color them all (Group D) in medium gray. The other group labels also indicate similar performance characteristics.

so that the fragments only accessed local variables (we do not do this in the original benchmark because the translation does not translate scope), and indeed we observed speedup with increasing problem size, as plotted in Fig. 4.

Case studies. As `commute` blocks are novel, there are no existing programs with them. Thus, we explored two case studies whose concurrent properties could be reformulated as Veracity programs with `commute` statements.

Crowdfund Smart Contract. Miners and validators in blockchain systems repeatedly execute enormous workloads of smart contract transactions. These transactions are currently executing sequentially, and recent proposals have been made to leverage multicore architectures through parallelization [Dickerson et al. 2017; Saraph and Herlihy 2019; Tran et al. 2021] or through sharding [Pîrlea et al. 2021]. We modeled parts of an Algorand crowdfund smart contract[7]

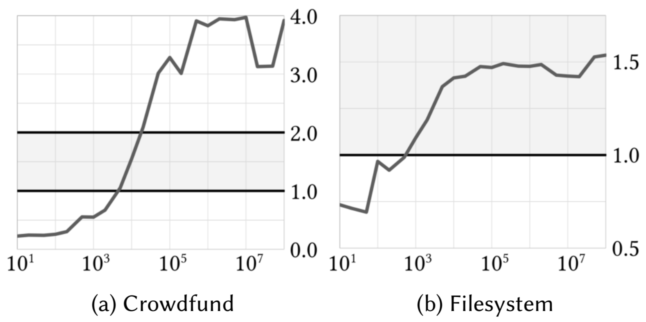


Fig. 5. Benchmark results for case studies.

[7]https://developer.algorand.org/solutions/example-crowdfunding-stateful-smart-contract-application/

as a case study. For performance, we tested four `donate` operations in parallel. Busy waits were added at the end of each donate. As expected, we saw significant speedups; the plot of speedup against problem size can be found in Fig. 5a. This example also illustrates that even better speedups can be achieved using `commute` blocks with more than 2 fragments.

Filesystem operations. A common low-conflict use case of hashtables is mapping directories to files. This could be in a user application, but also potentially in the implementation of a filesystem (*e.g.* ScaleFS [Clements et al. 2015; Eqbal 2014]). We modeled filesystem operations as veracity programs. Writing to files commutes when different files are being written to. The results are shown in Fig. 5b. When writing sufficiently large files, we achieved large speedups (approaching about 1.85×).

9 RELATED WORK

To our knowledge no prior works have explored bringing commutativity conditions *into* the programming language syntax, and the ramifications thereof. We now survey other related works.

It is long known that there is a fundamental connection between commutativity and concurrency, dating back to work from the database communities describing how to ensure atomicity in concurrent executions with locking protocols based on commutativity (*e.g.* [Bernstein 1966; Korth 1983; Weihl 1988]). Compiler-based parallelization of such arithmetic operations was introduced by Rinard and Diniz [1997]. Later commutativity *conditions* have been used in many concurrent programming contexts such as optimistic parallelism in graph algorithms [Kulkarni et al. 2008], transactional memory [Dickerson et al. 2019; Herlihy and Koskinen 2008; Ni et al. 2007], dynamic analysis [Dimitrov et al. 2014], and blockchain smart contracts [Bansal et al. 2020; Dickerson et al. 2017; Pîrlea et al. 2021]. Closed or open nested transactions [Ni et al. 2007] permit syntactically nested transactions. Although nested transactions (NT) might seem akin to nested `commute` statements, NTs still fall within the realm of a programming model in which users write a concurrent program (explicitly forking and so on), as opposed to our sequential `commute` statements.

In more recent years several works have focused on *reasoning* about commutativity. Gehr et al. [2015] describe a method based on black-box sampling. Aleen and Clark [2009] and Tripp et al. [2011] identify sequences of actions that commute (via random interpretation and dynamic analysis, resp.). Kim and Rinard [2011] *verify* commutativity conditions from specifications. Bansal et al. [2018] synthesize commutativity conditions from ADT specifications in the tool Servois [Servois 2018] (which we build on top of in Sec. 6). Koskinen and Bansal [2021] verify commutativity conditions from source code. Pîrlea et al. [2021] discover commutativity through a static analysis based on linear types and cardinality constraints. Najafzadeh et al. [2016] describe a tool for weak consistency that checks commutativity formulae. Houshmand and Lesani [2019] describe commutativity checking for replicated data types.

Numerous works are focused on synthesizing locks. Flanagan and Qadeer [2003] described a type-based approach, and Vechev et al. [2010] used abstraction-refinement. Cherem et al. [2008] and Golan-Gueta et al. [2015] describe automatic locking techniques for ensuring atomicity of, respectively, transactions and multi-object atomic sections.

Others such as Vakilian et al. [2009] and Bocchino et al. [2009] have worked on type-and-effect systems for parallelization, employing some commutativity, in Deterministic Parallel Java. Bocchino et al. [2009] describe a "commuteswith" annotation to inform their compiler that certain collection class operations (always) commute, despite there being read/write conflicts. These annotations do not involve commute conditions, nor do they use scoped serializability or our synthesis reduction. Outside of the formal setting, Prabhu et al. [2011] describe compiler pragmas that allow a programmer to define groups of operations in disparate regions of the code, and write commute conditions for members of those groups. The authors use this information as part of the program dependence graph to obtain speedups, but do not provide a formal semantics, nor verification/synthesis for commute conditions.

10 DISCUSSION AND FUTURE WORK

To our knowledge we have presented the first work on introducing `commute` statements into the programming language. We have given a formal semantics, a correctness condition, methods for ensuring parallelizability, and techniques for inferring/verifying `commute` conditions. This work is embodied in the new Veracity language and front-end compiler.

Future Work. On the practical side, top priorities for future work are to explore back-end compilation strategies to emit, *e.g.*, concurrent IR. Our work can also be combined with other parallelization strategies such as promises/futures [Chatterjee 1989; Liskov and Shrira 1988]. In our ongoing implementation we seek to integrate existing data flow analyses to reduce locking while enforcing scoped serializability. On the theoretical side we could generalize to N-way `commute` statements and some notion of object-oriented encapsulation, *i.e.*, arbitrary concurrent objects. With more encapsulated expert-written concurrent objects, more sequential `commute` programs can be written to exploit them.

Our work can also be combined with invariant generation, which can inform commutativity inference, as seen in the following example:

Example 10.1 (Combine with static analysis).

```
{ y < 0 } /* Example invariant */
commute _ {
  { y = y + 3*x; }  /* if x is negative, this will
      reduce y */
  { if (y<0) { x=2; } else { x=3; } } /* sensitive to
      whether y went below 0 */
}
```

Without the static knowledge of the invariant `y < 0` before the commute condition, an excessively complex commute condition would be inferred. However, if we first perform a static analysis, there are fewer choices to be made during abstraction-refinement, quickly leading to a simpler and more context-sensitive commute condition (in this case: `0 > y + 3*x && 2 == x`). Our embedding already supports such context information so in the future we aim to integrate Veracity with an abstract interpreter.

DATA AVAILABILITY STATEMENT

Our implementation of Veracity and all benchmark programs are available at https://github.com/veracity-lang/veracity, and an archived artifact is also available[8] [Chen et al. 2022b]. The artifact contains a docker image that builds and runs the interpreter and contains all benchmark programs. The source for the interpreter, along with all benchmark programs, is also provided.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and Maurice Herlihy for the suggestion of a smart contracts case study. This work was partially supported by the National Science Foundation award #2008633.

[8] https://doi.org/10.5281/zenodo.7058421

REFERENCES

2014. Multicore OCaml. https://github.com/ocaml-multicore/ocaml-multicore.

Farhana Aleen and Nathan Clark. 2009. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, Mary Lou Soffa and Mary Jane Irwin (Eds.). ACM, 241–252. https://doi.org/10.1145/1508244.1508273

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. 2005. The Fortress language specification. *Sun Microsystems* 139, 140 (2005), 116.

Joe Armstrong. 1997. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 196–203. https://doi.org/10.1145/258948.258967

Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018 (Lecture Notes in Computer Science, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 115–132. https://doi.org/10.1007/978-3-319-89960-2_7

Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2020. Synthesizing Precise and Useful Commutativity Conditions. *J. Autom. Reason.* 64, 7 (2020), 1333–1359. https://doi.org/10.1007/s10817-020-09573-w

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. 2009. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 735–736. https://doi.org/10.1145/1639950.1639989

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, Vol. 13. 14.

Arthur Bernstein. 1966. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers* 15, 5 (1966), 757–763. https://doi.org/10.1109/PGEC.1966.264565

Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, et al. 1994. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*. Citeseer, 141–154. https://doi.org/10.1007/BFb0025876

William Blume and Rudolf Eigenmann. 1992. Performance analysis of parallelizing compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems* 3, 6 (1992), 643–656. https://doi.org/10.1109/71.180621

Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 97–116. https://doi.org/10.1145/1640089.1640097

Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 6–15. https://doi.org/10.1145/1835698.1835703

Stephen Brookes and Peter W O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

Pavol Černỳ, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2013. Efficient synthesis for concurrency by semantics-preserving transformations. In *International Conference on Computer Aided Verification*. Springer, 951–967. https://doi.org/10.1007/978-3-642-39799-8_68

Arunodaya Chatterjee. 1989. Futures: a mechanism for concurrency among objects. In *Supercomputing'89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. IEEE, 562–567. https://doi.org/10.1145/76263.76326

Adam Chen, Parisa Fathololumi, Eric Koskinen, and Jared Pincus. 2022a. Veracity: Declarative Multicore Programming with Commutativity. https://doi.org/10.48550/ARXIV.2203.06229

Adam Chen, Parisa Fathololumi, Eric Koskinen, and Jared Pincus. 2022b. Veracity: Declarative Multicore Programming with Commutativity. *PACMPL* 6, 10 (Sept. 2022), 186:1 – 186:31. https://doi.org/10.5281/zenodo.7058421 Funded by NSF Grant #2008633..

Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. 2008. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)*. 304–315. https://doi.org/10.1145/1375581.1375619

Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)* 32, 4 (2015), 1–47. https://doi.org/10.1145/2699681

Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding Concurrency to Smart Contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. ACM, New York, NY, USA, 303–312. https://doi.org/10.1145/3087801.3087835

Thomas D. Dickerson, Eric Koskinen, Paul Gazzillo, and Maurice Herlihy. 2019. Conflict Abstractions and Shadow Speculation for Optimistic Transactional Objects. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*. 313–331. https://doi.org/10.1007/978-3-030-34175-6_16

Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. 2014. Commutativity Race Detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. https://doi.org/10.1145/2594291.2594322

Tayfun Elmas. 2010. QED: a proof system based on reduction and abstraction for the static verification of concurrent software. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 507–508. https://doi.org/10.1145/1810295.1810454

Rasha Eqbal. 2014. *ScaleFS: A multicore-scalable file system.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Gidon Ernst. 2020. A Complete Approach to Loop Verification with Invariants and Summaries. https://doi.org/10.48550/ARXIV.2010.05812

Gidon Ernst. 2022. Loop Verification with Invariants and Contracts. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 69–92. https://doi.org/10.1007/978-3-030-94583-1_4

Bin Fan, David G Andersen, and Michael Kaminsky. 2013. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 371–384. https://doi.org/10.5555/2482626.2482662

Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 338–349. https://doi.org/10.1145/781131.781169

Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning Commutativity Specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 307–323. https://doi.org/10.1007/978-3-319-21690-4_18

Guy Golan-Gueta, G Ramalingam, Mooly Sagiv, and Eran Yahav. 2015. Automatic scalable atomicity via semantic locking. *ACM SIGPLAN Notices* 50, 8 (2015), 31–41. https://doi.org/10.1145/2688500.2688511

Max Grossman, Alina Simion Sbirlea, Zoran Budimlić, and Vivek Sarkar. 2010. CnC-CUDA: declarative programming for GPUs. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 230–245. https://doi.org/10.1007/978-3-642-19595-2_16

Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, 175–184. https://doi.org/10.1145/1345206.1345233

Tim Harris and Keir Fraser. 2003. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)* (Anaheim, California, USA). ACM Press, 388–402. https://doi.org/10.1145/949305.949340

Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 387–388. https://doi.org/10.1145/2555243.2555283

Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly Concurrent Transactional Objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)* (Salt Lake City, Utah, United States). https://doi.org/10.1145/62678.62684

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC'03)*. 92–101. https://doi.org/10.1145/872035.872048

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)* (San Diego, California, United States). ACM Press, 289–300. https://doi.org/10.1145/165123.165164

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: replication coordination analysis and synthesis. *Proc. ACM Program. Lang.* 3, POPL (2019), 74:1–74:32. https://doi.org/10.1145/3290387

Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

Deokhwan Kim and Martin C. Rinard. 2011. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 528–541. https://doi.org/10.1145/1993498.1993561

Henry F. Korth. 1983. Locking Primitives in a Database System. *J. ACM* 30, 1 (1983), 55–79. https://doi.org/10.1145/322358.322363

Eric Koskinen and Kshitij Bansal. 2021. Decomposing Data Structure Commutativity Proofs with *mn*-Differencing. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 81–103. https://doi.org/10.1007/978-3-030-67067-2_5

Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. 2008. Loop summarization using abstract transformers. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 111–125. https://doi.org/10.1007/978-3-540-88387-6_10

Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 542–555. https://doi.org/10.1145/1993498.1993562

Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L Paul Chew. 2008. Optimistic parallelism benefits from data partitioning. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 233–243. https://doi.org/10.1145/1346281.1346311

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*. 211–222. https://doi.org/10.1145/1250734.1250759

Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14. https://doi.org/10.1145/2592798.2592820

Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. 1990. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Trans. Parallel Distributed Syst.* 1, 1 (1990), 26–34. https://doi.org/10.1109/71.80122

libcuckoo 2013. libcuckoo: A high-performance, concurrent hash table. http://efficient.github.io/libcuckoo/.

Sam Lindley. 2007. Implementing deterministic declarative concurrency using sieves. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. 45–49. https://doi.org/10.1145/1248648.1248657

Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM Sigplan Notices* 23, 7 (1988), 260–267. https://doi.org/10.1145/960116.54016

Yujie Liu, Kunlong Zhang, and Michael Spear. 2014. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 242–251. https://doi.org/10.1145/2611462.2611495

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE tool: proving weakly-consistent applications correct. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*. 2:1–2:3. https://doi.org/10.1145/2911151.2911160

Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. 68–78. https://doi.org/10.1145/1229428.1229442

Dominic A Orchard, Max Bolingbroke, and Alan Mycroft. 2010. Ypnos: declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*. 15–24. https://doi.org/10.1145/1708046.1708053

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta informatica* 6, 4 (1976), 319–340. https://doi.org/10.1007/BF00268134

Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653. https://doi.org/10.1145/322154.322158

George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical smart contract sharding with ownership and commutativity analysis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1327–1341. https://doi.org/10.

1145/3453483.3454112

Raghu Prabhakar and Rohit Kumar. 2011. *Concurrent programming with Go.* Technical Report. Citeseer.

Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P Johnson, and David I August. 2011. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation.* 1–11. https://doi.org/10.1145/1993316.1993500

Chris Purcell and Tim Harris. 2005. Non-blocking hashtables with open addressing. In *International Symposium on Distributed Computing*. Springer, 108–121. https://doi.org/10.1007/11561927_10

Martin C. Rinard and Pedro C. Diniz. 1997. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (November 1997), 942–991. https://doi.org/10.1145/267959.269969

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06).* 187–197. https://doi.org/10.1145/1122971.1123001

Vikram Saraph and Maurice Herlihy. 2019. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376* (2019). https://doi.org/10.48550/arXiv.1901.01376

Servois 2018. Servois: Synthesizing Commutativity Conditions. https://github.com/kbansal/servois.

Servois2 2022. Servois2: An Extended Commutativity Condition Synthesizer. https://github.com/veracity-lang/servois2.

Jake Silverman and Zachary Kincaid. 2019. Loop summarization with rational vector addition systems. In *International Conference on Computer Aided Verification.* Springer, 97–115. https://doi.org/10.1007/978-3-030-25543-5_7

Guy L. Steele Jr. 2005. Parallel Programming and Parallel Abstractions in Fortress. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), 17-21 September 2005, St. Louis, MO, USA*. IEEE Computer Society, 157. https://doi.org/10.1109/PACT.2005.34

Thi Hong Tran, Hoai Luan Pham, Tri Dung Phan, and Yasuhiko Nakashima. 2021. BCA: A 530-mW Multicore Blockchain Accelerator for Power-Constrained Devices in Securing Decentralized Networks. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 10 (2021), 4245–4258. https://doi.org/10.1109/TCSI.2021.3102618

Omer Tripp, Greta Yorsh, John Field, and Mooly Sagiv. 2011. HAWKEYE: effective discovery of dataflow impediments to parallelization. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011.* 207–224. https://doi.org/10.1145/2048066.2048085

Viktor Vafeiadis. 2010. Automatically proving linearizability. In *International Conference on Computer Aided Verification.* Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. 2009. Inferring Method Effect Summaries for Nested Heap Regions. In *2009 IEEE/ACM International Conference on Automated Software Engineering.* 421–432. https://doi.org/10.1109/ASE.2009.68

Various. 2007–2012. Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP). https://doi.org/10.1145/1481839

Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 327–338. https://doi.org/10.1145/1706299.1706338

William Weihl. 1988. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Comput.* 37, 12 (1988), 1488–1505. https://doi.org/10.1109/12.9728

William E. Weihl. 1983. Data-dependent concurrency control and recovery (Extended Abstract). In *Proceedings of the 2nd annual ACM symposium on Principles of Distributed Computing (PODC'83).* ACM Press, 63–75. https://doi.org/10.1145/800221.806710

Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. 2017. Automatic loop summarization via path dependency analysis. *IEEE Transactions on Software Engineering* 45, 6 (2017), 537–557. https://doi.org/10.1109/TSE.2017.2788018