Exploiting the Commutativity Lattice *

Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
milind@purdue.edu

Donald Nguyen[1], Dimitrios Prountzos[1],
Xin Sui[1], Keshav Pingali[1,2]
[1]Department of Computer Science, [2]Institute for
Computational Engineering and Sciences
The University of Texas at Austin
{ddn, dprountz, xinsui, pingali}@cs.utexas.edu

Abstract

Speculative execution is a promising approach for exploiting parallelism in many programs, but it requires efficient schemes for detecting conflicts between concurrently executing threads. Prior work has argued that checking semantic commutativity of method invocations is the right way to detect conflicts for complex data structures such as kd-trees. Several ad hoc ways of checking commutativity have been proposed in the literature, but there is no systematic approach for producing implementations.

In this paper, we describe a novel framework for reasoning about commutativity conditions: the *commutativity lattice*. We show how commutativity specifications from this lattice can be systematically implemented in one of three different schemes: abstract locking, forward gatekeeping and general gatekeeping. We also discuss a disciplined approach to exploiting the lattice to find different implementations that trade off precision in conflict detection for performance. Finally, we show that our novel conflict detection schemes are practical and can deliver speedup on three real-world applications.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent Programming Structures

General Terms Languages

Keywords Commutativity lattice, transactions, optimistic parallelism

1. Introduction

Speculative execution of high-level language programs has emerged as an important concept in multicore programming [12, 19, 22]. A central concern in implementing such systems is detecting conflicts accurately and with low overhead. These two goals are usually contradictory. For example, a single global lock on all of shared-memory will detect conflicts with low overhead, but it prevents parallel execution of speculative computations. Transactional memory (TM) is a better approach that tracks the set of locations (or objects) read and written by each transaction and reports conflicts if a location written by one transaction is read or written by another concurrently executing transaction [12]. This results in more fine-grained conflict detection than the first scheme and permits much more concurrency but at the expense of potentially higher overhead, especially if the TM is implemented in software.

Though the TM approach works well for many programs, it has been pointed out in the literature [10, 18, 25] that conflict detection at the memory level can be overly conservative for programs that use complex abstract data types (ADTs). Consider the disjoint-set union-find data structure. Two transactions might both invoke the find method, and these invocations may result in modifications to the concrete (internal) representation of the data structure because of path compression, causing memory-level conflicts[1]. However, from the ADT perspective, find operations do not modify the disjoint sets; they are read-only operations that commute with each other and do not conflict. Intuitively, there may be several *concrete states* of the data structure that represent the same *abstract state*; if the client of the ADT only cares about the abstract state, memory-level conflict detection can be overly conservative, reducing concurrency. This problem can also arise with other ADTs such as sets and kd-trees, as we discuss later in the paper.

Although issues surrounding the behavior of commutativity based conflict detection have been explored in prior work [10, 15, 25], no work has discussed how such semantic conflict checks should be *implemented*. It is straightforward to see how memory-level conflict checking can be performed for a complex data structure, but it is far less apparent how to implement a correct and efficient conflict detection scheme based on the semantics of union-find. Worse, every abstract data type potentially requires a different conflict-detection scheme. Prior work has relied on *ad hoc* implementations for the specific data structures they focus on [4, 10, 18, 20], but what is missing is a *systematic* way of generating correct conflict detectors.

Complicating matters further is that for a given abstract data type, there may be multiple correct conflict detection schemes that nevertheless provide different amount of parallelism at different overheads. For example, Ni *et al.* present a conflict checker for sets based on read/write locks on keys inserted into the set [20]. In contrast, Herlihy and Koskinen propose a conflict checker for sets based on exclusive locks on keys [10]. Given two conflict checkers, how do we know which one to use? Which one will expose more parallelism? Which one will have higher overhead? Intuitively, a more complex scheme may be able to expose more parallelism at the cost of additional run-time overhead; is this always the case? Can we produce multiple conflict checkers for a data structure in a disciplined way?

This paper addresses these issues through a novel way of reasoning about commutativity in abstract data types: a lattice of *commutativity specifications*. A commutativity specification is a set of predicates associated with pairs of methods in a data structure; a

* This work is supported in part by NSF grants 0923907, 0833162, 0719966, and 0702353 and by grants from IBM, NEC and Intel.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

[1] Ironically, a simpler ADT implementation for union-find that does not perform path compression would let the two invocations proceed concurrently!

pair of methods commute if the predicate associated with them evaluates to `true`. There are many possible commutativity specifications for a data structure, and Section 2 shows that they can be arranged into a *commutativity lattice*, ordered by the amount of parallelism they permit.

Section 3 shows how commutativity specifications can be systematically turned into commutativity checking implementations. We present three novel constructions: one that generates abstract lock-based schemes and two that generate novel conflict detection schemes called *gatekeepers*. Section 4 shows that different commutativity checking implementations for a data structure are actually implementations of different specifications from the commutativity lattice, and describes a disciplined way of producing different implementations for a given data structure. Section 5 explores the tradeoff between performance and precision, and shows that our systematically constructed conflict detectors can deliver scalable performance. Section 6 concludes and presents related work.

Contributions This paper makes the following contributions:

1. We give a definition of commutativity conditions and specifications, and describe a lattice-theoretic approach to reasoning about various commutativity specifications for a data structure.
2. We present three approaches for implementing commutativity specifications, *abstract locking*, *forward gatekeeping* and *general gatekeeping*. We develop algorithms for systematically constructing commutativity checkers of each type given a specification. We discuss the expressivity of each scheme, and their overheads.
3. We show for three real-world applications that our commutativity schemes are practical. We quantify the potential tradeoffs between the parallelism exposed and the overhead incurred by different schemes. We also show, paradoxically, that for some benchmarks the higher-parallelism scheme in fact has lower overhead.

2. Commutativity

We begin in Section 2.1 by briefly explaining how commutativity of method invocations can be used to check serializability. Section 2.2 next defines commutativity conditions, which are predicates a run-time system can use to guarantee serializability. Section 2.3 describes the language used to specify commutativity conditions. We then present a lattice-theoretic view of commutativity specifications in Section 2.4. Section 2.5 provides example specifications for a few data structures, illustrating the potential complexity of commutativity properties.

2.1 Serializability through commutativity

Our goal is to ensure that two transactions, A and B, that execute concurrently maintain transactional semantics. In particular, we want to ensure that A and B are serializable: that the parallel execution of A and B produce the same results as if A and B were executed sequentially in some order[2].

We consider data structures that provide a set of methods $m \in \mathcal{M}$. The data structure's state, σ, consists of any information needed to capture the *abstract* behavior of the structure. For example, a set data structure's abstract state is the set of elements contained in the data structure. In general, many concrete representations of a data structure have the same abstract state. For example, a set represented as a linked list and one represented as a hash set will have different concrete states, but, if they contain the same elements, would have the same abstract state.

[2] The following discussion will assume that there are only two transactions executing, and that the transactions access a single data structure; the approach to guaranteeing serializability generalizes straightforwardly.

The program consists of a series of method invocations performed by transactions on the data structure. Transaction T invoking a method m with argument v in state σ will be written $\langle m^T(v)\rangle_\sigma$. We will omit arguments, states and transaction labels unless necessary for clarity.

Each of these method invocations is atomic: when a method m is invoked, it appears to execute instantaneously at some point between when m was invoked and when it returned. More formally, we consider *linearizable* data structures [13]. Note that this means we need only consider interleavings of transactions at the granularity of method invocations [15].

A *history* is a series of method invocations performed by one or more transactions, starting from a particular state. The history records method invocations as well as their return values[3]. Note that because the invocations are atomic, the return values are available immediately after the method is invoked. Histories will be written as: $\langle m_1(v_1), m_2(v_2), \ldots, m_k(v_k)\rangle_\sigma$, indicating a series of method invocations starting from state σ. We can also discuss sub-histories, which are subsequences of larger histories; as these sub-histories also have some initial state, they can be considered histories in their own right.

Two histories H and H' are equivalent, denoted $H \equiv H'$, if their initial states are the same, they contain of the same method invocations (albeit potentially in a different order), each invocation returns the same result, and the states at the end of the histories are the same. Note that if we consider two equivalent sub-histories h and h', where h is a sub-history of some larger history H, replacing h with h' in H produces an equivalent history H'.

We can define *commutativity* in terms of history equivalence:

DEFINITION 1. *Two invocations $m_1(v_1)$ and $m_2(v_2)$ commute with respect to a state σ if and only if $\langle m_1(v_1), m_2(v_2)\rangle_\sigma \equiv \langle m_2(v_2), m_1(v_1)\rangle_\sigma$.*

In other words, two method invocations commute with respect to a given state if they are invoked consecutively starting in that state, and interchanging them produces an equivalent history[4].

We can use the commutativity of method invocations performed by A and B to check if their execution is serializable. Intuitively, if every method invocation in A commutes with every method invocation in B, then A and B are serializable, and B can be seen as executing before A. This type of serializability can be thought of as an extension of *d-serializability* [21] from non-interfering method invocations to commuting method invocations. As this approach has been used by several prior works ([10, 15, 18]), we present only an informal inductive argument here.

Assume that the first invocation performed by transaction A is $\langle m_1(v_1)^A\rangle_{\sigma_1}$, producing some new state σ'. When transaction B later performs its *last* invocation $\langle m_2(v_2)^B\rangle_{\sigma_2}$, a run-time check must be performed. If σ_2 is σ' (meaning each transaction has invoked one method), it suffices to show that $m_1(v_1)^A$ and $m_2(v_2)^B$ commute with respect to σ_1. This check allows the methods to be "interchanged" in the history, producing a history where $m_2^B(v_2)$ executes before $m_1^A(v_1)$.

In most cases, however, σ_2 will be separated from σ_1 by some intervening invocations by both A and B. It is apparent that we can move $m_1^A(v_1)$ and $m_2^B(v_2)$ towards each other by proving that the invocations commute with the intervening methods and swapping their positions in the history (hence pushing all later invocations

[3] Because return values are determined by method arguments and state, we do not explicitly represent the return values in the history.

[4] Prior work has distinguished between left-moving and right-moving commutativity, which constrain in which directions method invocations can be moved [15]. In this paper, we consider *both-moving* commutativity, where invocations can be moved in either direction.

from A after $m_2^B(v_2)$, and all earlier invocations from B before $m_1^A(v_1)$). At this point, let σ_1' be the state $m_1^A(v_1)$ in which is actually invoked, while $m_2^B(v_2)$ is invoked in the next state, σ_2'. If we can then prove that $m_1^A(v_1)$ and $m_2^B(v_2)$ commute with respect to σ_1', we can swap the two invocations and produce a serial history where all invocations from B occur before any invocations from A.

2.2 Commutativity conditions

It is apparent from the previous discussion that, if we could find predicates $\psi_{m_1;m_2}(\sigma, v_1, v_2)$, for all pairs of methods m_1 and m_2, that were true for histories $\langle m_1(v_1), m_2(v_2)\rangle_\sigma$ if and only if the two method invocations commuted in that history, we could check this formula as part of a run-time system that verified that two concurrently executing transactions were serializable. Furthermore, such predicates clearly exist (even if they are not easily expressible in a particular logic, or, for method invocations that don't commute, they are merely **false**).

Unfortunately, in a realistic run-time system, method invocations from two transactions will not appear back to back in a program's execution history. Instead, transaction A might invoke $\langle m_1(v_1)\rangle_{\sigma_1}$ and, much later, transaction B might invoke $\langle m_2(v_2)\rangle_{\sigma_2}$. What we desire is a *commutativity condition*: a predicate over method invocations that could occur in *any* two states that nevertheless lets us prove the commutativity of method invocations when they appear back-to-back and must be swapped. To make the definition of commutativity conditions more clear, we first define a more refined equivalence properties for histories:

DEFINITION 2. *Two histories H and H' are* C-EQUIVALENT*, denoted $H \equiv_C H'$, iff H can be transformed into H' by replacing sub-histories of the form $\langle m_1(v_1), m_2(v_2)\rangle_\sigma$ with equivalent sub-histories of the form $\langle m_2(v_2), m_1(v_1)\rangle_\sigma$.*

In other words, two histories are C-EQUIVALENT if one can be transformed into the other by swapping commuting method invocations.

We can now define commutativity conditions, which are over two method invocations that occur in two non-consecutive states in some history H:

DEFINITION 3. *A* commutativity condition*, ϕ, is a predicate on two method invocations $\langle m_1(v_1)\rangle_{\sigma_1}$ and $\langle m_2(v_2)\rangle_{\sigma_2}$ in an execution history H that is* **true** *only if, for all histories $H' \equiv_C H$ that contain a sub-history $h = \langle m_1(v_1), m_2(v_2)\rangle_{\sigma'}$, $m_1(v_1)$ and $m_2(v_2)$ commute with respect to σ'.*

In other words, a commutativity condition is a predicate ϕ that, when true in one history, means that in any C-EQUIVALENT history where the two method invocations happen back to back, the invocations commute. We say that a predicate is a *valid* commutativity condition if it satisfies the requirements of Definition 3.

In practice, these conditions can be expressed as a particular formula for a pair of methods, parameterized on the arguments, return value and states of the method invocations. Such commutativity formulae are thus written $\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2)$ (where r_1 and r_2 are the return values of $\langle m_1(v_1)\rangle_{\sigma_1}$ and $\langle m_2(v_2)\rangle_{\sigma_2}$, respectively, and can be read informally as:

$$\langle m_1(v_1)\rangle_{\sigma_1}/r_1 \text{ commutes with } \langle m_2(v_2)\rangle_{\sigma_2}/r_2 \text{ if } \phi$$

If, for each pair of method invocations $\langle m_1(v_1)\rangle_{\sigma_1}$ and $\langle m_2(v_2)\rangle_{\sigma_2}$ invoked by concurrently executing transactions, the run-time system ensures that $\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2)$ is **true**, then the current execution is serializable (a proof of this claim is given in Appendix A).

If one of these *commutativity checks* does not succeed, then the run-time can preserve serializability by undoing one transaction and re-executing it later when it will not be concurrent with the conflicting transaction. The implementation of commutativity checks is the subject of Section 3.

Discussion

Commutativity conditions are somewhat complex, and producing a valid commutativity condition for a pair of methods may seem daunting. While this paper does not address the construction and correctness of commutativity conditions (the verification problem, though not the construction problem, is addressed by techniques such as [14]), we discuss some interesting properties of commutativity conditions that can inform their specification and verification.

First, most commutativity conditions are expressible solely in terms of the arguments and return values of an invocation. While the return values are dependent on the states the invocations occur in, return values have a distinguished place in the definition of commutativity (specifically, return values must remain the same when two commuting method invocations are swapped). This means that such a predicate on method invocations in one history is true if and only if it is true in all C-EQUIVALENT histories, as the arguments and return values are invariant up to commutative swaps. Thus it suffices to consider the effects of two method invocations appearing back-to-back when creating such commutativity conditions.

As we will see in Section 2.5, some commutativity conditions are based on more than just arguments and return values. Instead, they also make use of relations over the abstract state of the data structure. It is less obvious how to reason about valid commutativity conditions in such cases. One approach is to consider the relations over abstract state to be merely auxiliary, "hidden," return values of the method invocations. As such, commuting methods must preserve these hidden return values as well as the actual return values. This is complicated by the complex conditions we see in some data structures, where the state relations are over abstract state as well as information from *future* method invocations. In such cases, determining that a commutativity condition is valid requires a more substantial proof; we leave this investigation to future work.

Note that a particular commutativity condition need not allow the run-time to detect that two transactions can execute in parallel, even if a different commutativity condition would. If the commutativity condition for two method invocations is simply **false**, the run-time system will determine that the two transactions cannot execute concurrently and will delay the execution of one transaction until the other completes. This reduces parallelism by preventing concurrent execution, but preserves serializability. The implications of the mismatch between schedules that can be proved serializable given certain commutativity conditions and those that could be proved serializable given better commutativity conditions is explored in Section 2.4.

2.3 Commutativity specifications

A *commutativity specification* is a set of logical formulae that represent commutativity conditions for each pair of methods in a data structure. The logic, L_1, of these formulae is given in Figure 1. The vocabulary of the logic includes the arguments and return values of method invocations. The specification can also use arbitrary functions over the arguments as well as the abstract states in which the methods are invoked. The logic allows boolean connectives, equality and arithmetic, but no quantifiers. In Section 3, we present some restricted logics that are subsets of L_1, and show that if a specification is expressible in those restricted logics we can derive efficient commutativity checking implementations.

S	:=	$\sigma_1 \mid \sigma_2$	*Abstract states*
V	:=	$v_1 \mid v_2 \mid r_1 \mid r_2$	*Arguments, return values,*
	|	$\mathbf{Z} \mid \mathbf{B}$	*constants*
F	:=	$f(S, V, V, ...)$	$(S \times V \times V \times \cdots) \rightarrow \mathbf{Z} \cup \mathbf{B}$
O	:=	$+ \mid - \mid * \mid / \mid < \mid >$	*Arithmetic connectives*
	|	$\wedge \mid \vee$	*Boolean connectives*
	|	$=$	*Equality*
P	:=	$V \mid F$	*Primitive formula*
C	:=	$P\ O\ P \mid (C) \mid \neg C \mid C\ O\ C$	*Formula*

$$\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2) = C$$

Figure 1. L_1: Logic to express commutativity conditions

(1)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{add}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee$ $(r_1 = \mathbf{false} \wedge r_2 = \mathbf{false})$
(2)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{remove}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee$ $(r_1 = \mathbf{false} \wedge r_2 = \mathbf{false})$
(3)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{contains}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee r_1 = \mathbf{false}$
(4)	$\langle \mathsf{remove}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{remove}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee$ $(r_1 = \mathbf{false} \wedge r_2 = \mathbf{false})$
(5)	$\langle \mathsf{remove}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{contains}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee r_1 = \mathbf{false}$
(6)	$\langle \mathsf{contains}(a) \rangle_{\sigma_1} / r_1$	commutes with	$\langle \mathsf{contains}(b) \rangle_{\sigma_2} / r_2$

Figure 2. Commutativity specification for sets.

An example commutativity specification for a set is given in Figure 2[5]. Note that add and remove return a boolean that indicates whether the invocation modified the set. Each condition is relatively straightforward: methods commute with each other if (i) neither modifies the set or (ii) their arguments are different. For example, condition (1) states that $\mathsf{add}(x)$ commutes with $\mathsf{add}(y)$ if $x \neq y$ or neither invocation of add modified the set (as indicated by their returning false).

For linearizable data structures, the relevant state of the data structure is the *abstract* state; the *concrete* implementation of the data structure is not relevant to commutativity. For example, a set data structure implemented concretely using a linked list has the same commutativity properties as a set data structure implemented using a red-black tree. If a commutativity specification *does* refer to the concrete (*i.e.*, memory-level) state of a data structure, then it will only be valid for particular implementations of the ADT.

2.4 A Commutativity Lattice

While Figure 2 presents a single commutativity specification for sets, it is by no means the only one. Recall that a commutativity condition is a predicate that satisfies Definition 3. However, as discussed in Section 2.2, for each pair of methods in an ADT, there may be multiple valid commutativity conditions (for example, **false** is always a valid commutativity condition).

Despite there being multiple valid conditions for a pair of methods, there is only one *precise* condition, ϕ^*, which is true if and only if the conditions of Definition 3 hold. Note that this means that all commutativity conditions $\phi_{m_1;m_2} \Rightarrow \phi^*_{m_1;m_2}$. The commutativity specification given in Figure 2 is precise, meaning that all the formulae capture precise commutativity conditions.

If we consider the set of all valid commutativity conditions for a pair of methods, S, we can build a partially ordered set (poset), $P = (S, \prec)$ based on logical implication, such that $\phi_1 \preceq \phi_2$ iff $\phi_1 \Rightarrow \phi_2$. This poset has a natural least element, $\bot = \mathbf{false}$,

[5] The commutativity formulae presented in this paper are symmetric: $\phi_{m_1;m_2}$ is the same as $\phi_{m_2;m_1}$. A full specification also includes these symmetric conditions, which have been excluded for brevity.

(1)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{add}(b) \rangle_{\sigma_2} / r_2$ $a \neq b$
(2)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{remove}(b) \rangle_{\sigma_2} / r_2$ $a \neq b$
(3)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{contains}(b) \rangle_{\sigma_2} / r_2$ $a \neq b$
(4)	$\langle \mathsf{remove}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{remove}(b) \rangle_{\sigma_2} / r_2$ $a \neq b$
(5)	$\langle \mathsf{remove}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{contains}(b) \rangle_{\sigma_2} / r_2$ $a \neq b$
(6)	$\langle \mathsf{contains}(a) \rangle_{\sigma_1} / r_1$	commutes with	$\langle \mathsf{contains}(b) \rangle_{\sigma_2} / r_2$

Figure 3. A strengthened commutativity specification for sets.

which is obviously less than all the other elements and is also trivially a valid commutativity condition. More interestingly, ϕ^* is the greatest element ($\top$) of the poset. This follows directly from the definitions of commutativity conditions and ϕ^*.

We can then define two binary operators, $\sqcup$ (least upper bound, or join) and $\sqcap$ (greatest lower bound, or meet) for every pair of conditions in S, as follows:

$$a \sqcap b \quad \Leftrightarrow \quad a \wedge b$$
$$a \sqcup b \quad \Leftrightarrow \quad a \vee b$$

The structure $(S, \sqcup, \sqcap, \top, \bot)$ forms a *bounded lattice*. Note that $a \preceq b \Leftrightarrow (a \sqcup b = b) \Leftrightarrow (a \sqcap b = a)$.

Note that a particular commutativity condition (in particular, ϕ^*) may not be expressible in a given logic L. We can create a restriction of the set of valid commutativity conditions to those expressible in L, S_L. Provided that L is closed under disjunction (as L_1 is), we can build a lattice over S_L with its $\top$ as the *weakest* commutativity condition expressible in L.

While the lattice has been defined only for a pair of methods of an ADT, it can be readily extended to create a bounded lattice of commutativity specifications, with elements of the lattice representing specifications that are distinct up to logical equivalence. For two commutativity specifications Φ_1 and Φ_2, $\Phi_1 \preceq \Phi_2$ if and only if, for each pair of methods m_1, m_2, where ϕ_1 is the commutativity condition for m_1 and m_2 from Φ_1 and ϕ_2 is the corresponding condition from Φ_2, $\phi_1 \preceq \phi_2$. We can define $\sqcup$ and $\sqcap$ in a similar manner (joining or meeting corresponding pairs of commutativity conditions from the two specifications). $\bot$ is the specification where all conditions are **false**, and $\top$ is the specification where all conditions are precise: Φ^*.

Moving up and down the commutativity lattice

Different points in the commutativity specification lattice have different properties; as you move down in the lattice, the commutativity conditions become *stronger*, and hence less likely to prove that two methods commute (and $\bot$ does not allow any methods to commute). Recall that two transactions are allowed to proceed in parallel if we can prove that the method invocations from those transactions commute. As a consequence, *stronger commutativity conditions permit fewer transactions to proceed in parallel, potentially reducing the amount of parallelism an application exhibits.*

For example, consider the specification for sets, Φ', given in Figure 3, which is derived by dropping some of the clauses from Φ^*, the precise specification of Figure 2. It is obvious that the new specification is lower in the commutativity lattice than the precise specification. If two transactions A and B, each execute $\mathsf{add}(x)$ on a set that already contains x, the adds commute according to Φ^*, but not according to Φ'. A conflict detection scheme based on the former specification will allow A and B to execute in parallel, while one based on the latter will not, reducing parallelism.

The choice of which commutativity specification from the lattice to use as the basis for a conflict detector clearly has implications for parallelism. However, as we will see in Section 3, different commutativity specifications have properties that allow them to be

Definitions:
$\mathsf{dist}(a, b)$ is an algorithm defined-distance metric such that $\mathbf{nearest}(a)$ returns the nearest point according to dist.

(1)	$\langle \mathsf{nearest}(a) \rangle_{\sigma_1} / r_1$	commutes with	$\langle \mathsf{nearest}(b) \rangle_{\sigma_2} / r_2$
(2)	$\langle \mathsf{nearest}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{add}(b) \rangle_{\sigma_2} / r_2$ $r_2 = \mathbf{false} \vee$ $\mathsf{dist}(a, b) > \mathsf{dist}(a, r_1)$
(3)	$\langle \mathsf{nearest}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{remove}(b) \rangle_{\sigma_2} / r_2$ $(a \neq b \wedge r_1 \neq b) \vee$ $r_2 = \mathbf{false}$
(4)	$\langle \mathsf{remove}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{remove}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee$ $(r_1 = \mathbf{false} \wedge r_2 = \mathbf{false})$
(5)	$\langle \mathsf{remove}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{add}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee$ $(r_1 = \mathbf{false} \wedge r_2 = \mathbf{false})$
(6)	$\langle \mathsf{add}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{add}(b) \rangle_{\sigma_2} / r_2$ $a \neq b \vee$ $(r_1 = \mathbf{false} \wedge r_2 = \mathbf{false})$

Figure 4. Commutativity specification for kd-trees.

implemented using different approaches. Hence, different choices of specifications also affect the *overhead* of a conflict detector. In fact, it may be worth accepting less parallelism in exchange for lower overhead. Although the precise commutativity specification will expose the most parallelism, conflict detectors based on less precise schemes may achieve low enough overhead to deliver superior performance. Sections 4 and 5 investigate this tradeoff.

2.5 Example specifications

In this section, we provide the commutativity specifications for two complex data structures, the kd-tree and the union-find structure, that have particularly interesting commutativity properties. As we will see in the next section, the complexity of these specifications constrains their implementation choices.

Kd-tree Kd-trees are used to find the nearest point to a given point among a set of points in space. They operate by recursively partitioning a high-dimensional space along one dimension at a time. The abstract state of a kd-tree is the set of points in the tree (in this respect, kd-trees are similar to sets) as well as a relation dist over pairs of points that captures the distance (usually Euclidean) between two points. The basic operations of a kd-tree are $\mathsf{add}(a)$, which adds point a to the kd-tree, and $\mathsf{remove}(a)$, which removes point a from the kd-tree. The query $\mathsf{nearest}(a)$ returns the nearest point to a.

One implementation of a kd-tree is as follows. Points are stored only in the leaf nodes, and each interior node records the location of its splitting plane. To accelerate $\mathsf{nearest}$ calls, each node also stores the bounding box of all the points contained in the subtree below it. When a node is added, the tree is walked to find the appropriate leaf node and the point is added there. The bounding box of each node from the root to this leaf node is then updated to account for the new point. Removing a node functions similarly. When $\mathsf{nearest}$ is invoked, we use the splitting planes and the bounding boxes to determine which subtrees could potentially contain a nearby point. These bounding boxes allow us to avoid traversing some subtrees of the kd-tree, allowing $\mathsf{nearest}$ queries to be executed in expected time that is logarithmic in the number of points.

The commutativity conditions for kd-trees are given in Figure 4. The commutativity conditions (4–6), are similar to those for sets. The conditions dealing with $\mathsf{nearest}$ however are more complex. Recall that the $\mathsf{nearest}(a)$ returns the point closest to a according to a distance metric dist. As $\mathsf{nearest}$ is a read-only operation, it clearly commutes with itself, as stated in condition (1). Condition (3) states that $\mathsf{nearest}$ commutes with remove as long as remove does not remove either the argument or return value of $\mathsf{nearest}$. Condition (2) states that $\mathsf{nearest}(a)$ commutes with $\mathsf{add}(b)$ as long as $\mathsf{add}(b)$ does not modify state, or b is not closer to the a than the return value of $\mathsf{nearest}(a)$ is.

Definitions:

$\mathsf{rep}(\sigma, x) =$ return value of $\mathsf{find}(x)$ invoked in state σ

$\mathsf{rank}(\sigma, x) =$ the "rank" of x in state σ (as defined in union-by-rank).

$$\mathsf{loser}(\sigma, x, y) = \begin{cases} \mathsf{rep}(\sigma, x) & \text{if } \mathsf{rank}(\mathsf{rep}(\sigma, x)) < \mathsf{rank}(\mathsf{rep}(\sigma, y)) \\ \mathsf{rep}(\sigma, y) & \text{otherwise} \end{cases}$$

(1)	$\langle \mathsf{union}(a, b) \rangle_{\sigma_1}$	commutes with if	$\langle \mathsf{union}(c, d) \rangle_{\sigma_2}$ $\mathsf{rep}(\sigma_1, c) \neq \mathsf{loser}(\sigma_1, a, b) \wedge$ $\mathsf{rep}(\sigma_1, d) \neq \mathsf{loser}(\sigma_1, a, b)$
(2)	$\langle \mathsf{union}(a, b) \rangle_{\sigma_1}$	commutes with if	$\langle \mathsf{find}(c) \rangle_{\sigma_2} / r_2$ $\mathsf{rep}(\sigma_1, c) \neq \mathsf{loser}(\sigma_1, a, b)$
(3)	$\langle \mathsf{union}(a, b) \rangle_{\sigma_1}$	commutes with if	$\langle \mathsf{create}(c) \rangle_{\sigma_2} / r_2$ $\mathbf{false}$
(4)	$\langle \mathsf{find}(a) \rangle_{\sigma_1} / r_1$	commutes with	$\langle \mathsf{find}(b) \rangle_{\sigma_2} / r_2$
(5)	$\langle \mathsf{find}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{create}(b) \rangle_{\sigma_2} / r_2$ $\mathbf{false}$
(6)	$\langle \mathsf{create}(a) \rangle_{\sigma_1} / r_1$	commutes with if	$\langle \mathsf{create}(b) \rangle_{\sigma_2} / r_2$ $\mathbf{false}$

Figure 5. Commutativity specification for union-find structures.

Union-find The abstract state of a union-find data structure is the set of disjoint sets. Each set also has an associated *representative* element, and a *rank* that determines how two sets should be merged. Union-find data structures support two operations on disjoint sets: $\mathsf{union}(a, b)$, which merges the subset containing a with that of b, and $\mathsf{find}(a)$, which determines which subset a belongs to, returning its representative element. The most efficient union-find implementation is a disjoint-set forest. Each subset is represented as a tree rooted at the set's representative element, which is used to identify the entire subset. Each element in the subset points to some other element as its parent. To find which subset an element belongs to, we start at the element and traverse parent pointers until we reach the root, and then return the representative element. To merge two subsets, the representative element of the set with larger rank is made the parent of the representative element of the other.

For asymptotic efficiency, it is necessary to use path compression. When invoking find, the parent pointer of every element traversed to find the representative element is updated to point to that representative element, "flattening" that portion of the tree. An interesting side effect of path compression is that find operations update the data structure. While the *semantic* state of the data structure does not change (invoking find on an element does not change the membership of any subset), the *concrete* state of the data structure can change due to path compression.

The union-find data structure has the most complex commutativity conditions. As expected, find operations commute with each other (see condition (4)). More complex are the conditions dealing with union. We use three helper functions that return properties of the abstract state: $\mathsf{rep}(\sigma, a)$, which returns a's representative node; $\mathsf{rank}(\sigma, x)$, which returns the rank of the subset containing x and $\mathsf{loser}(\sigma, a, b)$ which finds the representative nodes of a and b in state σ and returns the one with lowest rank or $\mathsf{rep}(\sigma, b)$ if the ranks are equal. Condition (1) states that two unions commute as long as the second doesn't operate on the loser from the first. Note that this is defined by evaluating rep on the arguments to the second union *in the state that the first union executed in*. Similarly, condition (2) states that union commutes with find provided that find *would not have returned the loser of the union had it executed in* σ_1. The dependence of commutativity on evaluating functions in an earlier state using information from a later method invocation introduces subtle difficulties, as we discuss in the following section. For simplicity, conditions (3), (5) and (6) state that create does not commute with anything (more precise conditions are possible, but they are quite complex).

3. Implementing commutativity specifications

As discussed in the previous section, for a given ADT there is a range of commutativity specifications to choose from, and different choices can expose different amounts of parallelism. However, what is not clear is *which* specification to choose. This choice is influenced by the *implementation* of the commutativity specification: the code that actually detect conflicts between concurrently executing transactions according to the commutativity specifications of the data structures they access.

Most prior work that has discussed using high-level conflict detection has provided *ad hoc* conflict detection schemes for a few chosen data structures [10, 18, 20]. However, previous work has left unanswered whether there are *systematic* approaches to building efficient, correct conflict detectors. In this section, we show that conflict detectors can be built systematically (albeit not automatically) given a commutativity specification. We present three such techniques, one based on abstract locks and two based on logging, that we call *gatekeepers*. We also discuss how the properties of a commutativity specification constrain its implementation and relate these constraints to the lattice-theoretic view of commutativity.

3.1 Soundness and Completeness

In general, there are many ways to implement a commutativity specification. For simple data structures like sets, abstract locks are one popular approach [10, 20], but it is not clear that they can implement the commutativity specifications for more complex data structures like kd-trees and union-find data structures. In fact, we show in Section 3.2 that abstract locks are too "coarse" to implement commutativity conditions for these data structures. To formalize the notions of correctness and coarseness, we introduce the following definitions.

DEFINITION 4. *A commutativity checker is* SOUND *with respect to a commutativity specification Φ if it allows two concurrent method invocations, m_1 and m_2, to execute without detecting a conflict only if the methods commute according to the Φ.*

DEFINITION 5. *A commutativity checker is* COMPLETE *with respect to a commutativity specifcation Φ if it permits every pair of methods invocations m_1 and m_2 that commute according to the Φ to execute without conflict.*

A sound and complete implementation of a commutativity specification therefore guarantees safe parallel execution and will permit the maximum amount of parallelism allowed by the specification. It is interesting to note that if a commutativity checker is sound but not complete with respect to a specification Φ, it *must* be sound and complete with respect to some specification Φ' such that $\Phi' \Rightarrow \Phi$ (in other words, the checker is sound and complete with respect to some specification lower in the commutativity lattice). Similarly, if a commutativity checker is sound with respect to a specification Φ, it is sound with respect to all other specifications Φ' such that $\Phi \Rightarrow \Phi'$.

3.2 Abstract Locking Schemes

The first commutativity checking scheme we discuss is *abstract locking*. An abstract lock is a lock with a number of *modes*. When attempting to acquire a lock in a particular mode, the acquisition succeeds if no other entity holds the lock in an incompatible mode. The compatibility of modes is determined by a lock's *compatibility matrix*. These abstract locks are a generalization of database mode locks [8] and subsume the abstract locks used in [20].

An *abstract locking scheme* for a data structure operates as follows: the data structure associates an abstract lock with each of its data members (defined as any arguments or return values to methods of the data structure). The data structure also has an abstract lock which represents whole data structure. When a transaction invokes a method, the abstract locks on its arguments and on the data structure *may* be acquired in some mode, and when the method returns, the abstract lock on the return value may be acquired in some mode. If an abstract lock cannot be acquired, it must be held in a conflicting mode by another transaction. The abstract locking scheme triggers a conflict, and a compensating action must be taken (either the current transaction or the conflicting transaction must be aborted). All abstract locks are released when a transaction ends[6].

$$
\begin{array}{lcll}
V_1 & := & v_1 \mid r_1 & \textit{Arguments, return values of } m_1 \\
V_2 & := & v_2 \mid r_2 & \textit{Arguments, return values of } m_2 \\
P & := & (V_1 \neq V_2) \mid P \wedge P & \textit{Conjunctive formula} \\
C & := & P \mid \mathbf{true} \mid \mathbf{false} & \textit{Formula}
\end{array}
$$

$$\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2) = C$$

Figure 6. L_2: Logic for SIMPLE commutativity conditions

(1)	$\langle \mathsf{increment}(a) \rangle_{\sigma_1}$	commutes with	$\langle \mathsf{increment}(b) \rangle_{\sigma_2}$
(2)	$\langle \mathsf{increment}(a) \rangle_{\sigma_1}$	commutes with if	$\langle \mathsf{read}() \rangle_{\sigma_2} / r_2$ **false**
(3)	$\langle \mathsf{read}() \rangle_{\sigma_1} / r_1$	commutes with	$\langle \mathsf{read}() \rangle_{\sigma_2} / r_2$

Figure 7. Commutativity specification for accumulators.

To create a locking scheme for a particular data structure, we must choose which locks are acquired, in which modes, and the particular mode compatibility matrix for the abstract locks.

Abstract locks are well-suited for providing conflict checking for collections [4, 20] and other basic data structures [10]. However, in prior work, programmers had to carefully consider the semantics of a data structure to build ad hoc abstract locking schemes. Two key questions were left unanswered: (i) *can abstract locks provide a sound and complete implementation of a commutativity specification?* and (ii) *can an abstract lock-based conflict detector be systematically constructed?*

Constructing abstract locking schemes

We now present an algorithm for constructing sound and complete abstract locking schemes for commutativity specifications whose conditions satisfy the following property:

DEFINITION 6. *A commutativity condition $\phi_{m_1;m_2}$ is* SIMPLE *if one of three conditions holds: (i) $\phi_{m_1;m_2} = \mathbf{false}$ (the methods do not commute); (ii) $\phi_{m_1;m_2} = \mathbf{true}$ (the methods always commute); or (iii) $\phi_{m_1;m_2}$ is a conjunction of clauses of the form $a \neq b$ where every a is an argument or return value of m_1 and every b is an argument or return value of m_2.*

More formally, a commutativity specification is SIMPLE if all of its conditions are expressed in the language L_2, given in Figure 6.

To produce an abstract locking scheme, we must (i) define the abstract locks, and their modes; (ii) determine which abstract locks should be acquired when a method is invoked, and in which mode; and (iii) construct the compatibility matrix between the various abstract lock modes. We provide a systematic procedure for accomplishing these steps. As a running example, we will explain how the procedure applies to an accumulator data structure, using the commutativity specification given in Figure 7.

1. **Define the abstract locks and their modes:** An abstract lock is associated with each data member and with the data structure itself. Each lock supports several modes: one mode per method to represent the method's access to the data structure as a whole, and one mode for each argument and return value of a method.

[6] One can imagine a more liberal abstract locking scheme that allows simple predicates to be evaluated before acquiring a lock. We leave the precise definition and investigation of such schemes to future work.

	inc:ds	*inc:x*	*read:ds*	*read:ret*
inc:ds	√	√	×	√
inc:x	√	√	√	√
read:ds	×	√	√	√
read:ret	√	√	√	√

(a) Compatibility matrix for accumulator abstract locks

	inc:ds	*read:ds*
inc:ds	√	×
read:ds	×	√

(b) Reduced compatibility matrix

Figure 8. Compatibility matrices for accumulator abstract locks

For example, the accumulator data structure associates abstract locks with the data structure as a whole, as well as every object that can be passed as an argument or returned by a method (these object locks can be allocated as needed, rather than being created ahead of time). Each lock supports four modes: $\mathsf{increment}(x)$ requires two modes, one for its argument (called *inc:x*) and one for the data structure (called *inc:ds*); $\langle \mathsf{read}() \rangle_{\sigma_\sigma} / r_\sigma$ requires two modes, one for its return value (called *read:ret*) and one for the data structure (called *read:ds*).

2. **Define which abstract locks should be acquired when a method is invoked:** Any time a method is invoked it acquires the data structure lock in its mode, as well as locks on all its arguments in their appropriate modes.

   For example, when increment is invoked, it acquires the data structure lock in *inc:ds* mode and the lock on its argument in *inc:x* mode. Similarly, when read is invoked, it acquires the data structure lock in *read:ds* mode and the lock on its return value in *read:ret* mode.

3. **Construct the compatibility matrix:** To this point, our definitions of abstract locks and modes did not refer to the commutativity specification. The specification is used to determine the compatibility between various abstract lock modes.

   To derive the mode compatibility matrix, we consider each pair of methods m_1 and m_2 and their commutativity condition, $\phi_{m_1;m_2}$, and applying the following three rules to determine the compatibility of the methods' modes:

   1. If $\phi_{m_1;m_2} =$ **false**, modes m_1:*ds* and m_2:*ds* are incompatible, but all other modes are left undefined. Hence, *inc:ds* is incompatible with *read:ds*.
   2. For each conjunct $x \neq y$ in $\phi_{m_1;m_2}$, where x is an argument or return value of m_1 and y is an argument or return value of m_2, mode m_1:x is incompatible with mode m_2:y.
   3. Any pair of lock modes whose compatibility is left undefined by rules 1 and 2 are assumed to be compatible.

   The compatibility matrix thus generated for the accumulator is shown in Figure 8(a). Note that this compatibility matrix, as well as the rules of abstract locks acquisition, prevents reads from being executed if another transaction has executed increment, while transactions can simultaneously execute read.

Given the full compatibility matrix, we note that if a lock mode is compatible with all other lock modes, acquiring it is superfluous. We can eliminate any such lock modes and remove the corresponding lock acquisitions from the relevant methods. This optimization yields the reduced compatibility matrix in Figure 8(b). The reduced abstract locking scheme acquires the data structure lock in *inc:ds* mode whenever increment is called, allowing other calls to increment to proceed without conflict, and read equivalently acquires the lock in *read:ds* mode.

While it is possible to produce a sound abstract locking scheme for any commutativity specification (a single, global exclusive lock constitutes a sound abstract locking scheme), not all specifications have a complete abstract locking implementation. For example, the kd-tree specification in Figure 4 relies on evaluating a function (to find the distance between two points) to determine whether nearest commutes with add, which cannot be captured by abstract locks. In fact, we can show the following:

THEOREM 1. *Given a commutativity specification Φ, a sound and complete abstract locking scheme exists iff Φ is* SIMPLE. □

The proof of Theorem 1 is given in Appendix B. This theorem means that while the strengthened commutativity specification for sets in Figure 3 can be implemented using an abstract locking scheme, the precise specification of Figure 2 cannot.

3.3 Gatekeeping

In this section, we present a new conflict detection paradigm called *gatekeeping*. A *gatekeeper* is a special object associated with a particular data structure whose role is to ensure that methods invoked by concurrently executing transactions on a data structure respect the specified commutativity conditions. At a high level, gatekeepers operate as follows. When an transaction A invokes a method m on a data structure, the gatekeeper "intercepts" the invocation and determines if the method invocation commutes with all other active method invocations (*i.e.*, methods invoked by all transactions B which have not yet ended). If the invocation commutes with all active method invocations, it is allowed to proceed and the transaction receives the result. If the invocation does not commute, then the gatekeeper triggers a conflict.

To determine if a method m_1 commutes with an active method m_2, the gatekeeper is allowed to evaluate predicates on the arguments and return values of m_1 and m_2. Additionally, the gatekeeper can invoke methods on the data structure it protects. The ability to evaluate arbitrary predicates and methods makes the gatekeeping approach strictly more expressive than abstract locking.

Because a gatekeeper interacts with a data structure only by invoking methods on it, the data structure is effectively a "black box." This means that the gatekeeper is agnostic to the actual implementation details of the data structure, and a gatekeeper constructed to protect one abstract data type (*e.g.*, a gatekeeper which protects sets) can protect *all* implementations of that abstract data type. However, the gatekeeper's behavior must appear atomic. In other words, the entire sequence of events: (i) intercepting a method invocation, (ii) checking commutativity; (iii) executing the method on the data structure, and (iv) returning the result to the invoking transaction, should appear atomic. There are two types of gate-keepers. The first type, *forward* gatekeepers, can only implement a restricted set of commutativity specifications, while the second, *general* gatekeepers, can implement more general commutativity conditions.

3.3.1 Forward gatekeepers

A forward gatekeeper operates by building up information about method invocations as they are invoked, storing that information in logs associated with the invocation, and using those logs to verify that every invocation commutes with all preceding invocations from other transactions. Consider the operation of a forward gatekeeper for kd-trees:

Whenever $\mathsf{nearest}(x)$ is invoked, returning r, the gatekeeper must both ensure that nearest commutes with preceding invocations and also record enough information to ensure that later method invocations can check their commutativity with $\mathsf{nearest}(x)$. It does this by storing the tuple $\langle x, \mathsf{dist}(x, r) \rangle$ in a log associated with the invocation. When a transaction later invokes $\mathsf{add}(y)$, the information from this log is used to evaluate $\mathsf{dist}(x, r) < \mathsf{dist}(x, y)$ to check commutativity. The point y is then stored in a log associated with the invocation $\mathsf{add}(y)$ to facilitate later commutativity checks. As transactions commit or abort, the logs associated with their invocations are cleared.

S	:=	$\sigma_1 \mid \sigma_2$	*Abstract states*
V	:=	$v_1 \mid v_2 \mid r_1 \mid r_2$	*Arguments, return values,*
	|	$\mathbf{Z} \mid \mathbf{B}$	*constants*
V_1	:=	$v_1 \mid r_1$	Arguments, return values of m_1
F_1	:=	$f(\sigma_1, V_1, V_1, ...)$	Function of σ_1 and m_1 information
F_2	:=	$f(\sigma_2, V, V, ...)$	Function of σ_2 and any value
O	:=	$+ \mid - \mid * \mid / \mid < \mid >$	*Arithmetic connectives*
	|	$\wedge \mid \vee$	*Boolean connectives*
	|	$=$	*Equality*
P	:=	$V \mid F_1 \mid F_2$	*Primitive formula*
C	:=	$P\ O\ P \mid (C) \mid \neg C \mid C\ O\ C$	*Formula*

$$\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2) = C$$

Figure 9. L_3: Logic for ONLINE-CHECKABLE conditions

Constructing forward gatekeepers

Forward gatekeepers can be constructed for commutativity specifications whose conditions are ONLINE-CHECKABLE:

DEFINITION 7. $\phi_{m_1;m_2}(v_1, \sigma_1, r_1, v_2, \sigma_2, r_2)$ *is* ONLINE-CHECKABLE *if any function of* v_2, σ_2 *or* r_2 *in* $\phi_{m_1;m_2}$ *is* not *a function of* σ_1.

In other words, a condition is ONLINE-CHECKABLE if there is no function that must be invoked in the state of the first method invocation using information from the second method invocation. More formally, an ONLINE-CHECKABLE condition is one that can be expressed using the logic L_3, given in Figure 9. Note that L_3 is very similar to L_1 (Figure 1), with the exception that functions that deal with abstract state σ_1 cannot also use v_2 or r_2 as arguments. This restriction means that conditions (1–2) of the union-find specification in Figure 5 are not ONLINE-CHECKABLE.

Before describing forward gatekeepers, let us define the set C_m of primitive functions associated with each method, m. For each commutativity condition $\phi_{m_1;m_2}$, collect any functions from F_1 (which only operate over the arguments, return value and abstract state of m_1) that appear in the formula. Place these into C_{m_1}. Note that in general, for a given method m its associated set C_m will have functions from *every* commutativity condition that m participates in.

Given the function sets constructed as above, the operation of a forward gatekeeper proceeds as follows:

1. When a gatekeeper sees a method invocation $m(v)$, it evaluates every function in C_m and records the result in a result log $L_{m(v)}$. It also records the return value of $m(v)$ in $L_{m(v)}$. Note that this result log is keyed by both the method and its arguments.
2. The gatekeeper records $m(v)$ as an *active* invocation (*i.e.*, a method invoked by a currently executing transaction).
3. The gatekeeper checks the commutativity of $m(v)$ by evaluating, for every active method invocation $m_a(v_a)$ made by other transactions, the predicate $\phi_{m_a;m}$. This can be done efficiently using results stored in $L_{m_a(v_a)}$ as well as the arguments and return values of $m(v)$. If any such commutativity condition evaluates to **false**, $m(v)$ cannot proceed, and one of the conflicting transactions must be aborted.

   Crucially, because all the commutativity conditions are ONLINE-CHECKABLE, $L_{m_a(v_a)}$ contains all the information necessary from $m_a(v_a)$ for evaluating $\phi_{m_a;m}$. Thus, the gatekeeper can determine commutativity simply by recording information about method invocations *as they happen*, rather than well after they execute.
4. When a transaction completes, the gatekeeper removes the result sets L associated with the methods invoked by that transaction, and removes the methods from the active invocation list.

Because the forward gatekeeper explicitly evaluates $\phi_{m_1;m_2}$ when determining the commutativity of m_1 and m_2, it is obviously sound and complete. Furthermore, it is apparent that if the online-checkability condition is violated, there will be a function f that uses σ_1 as well as arguments or return values from m_2. This function cannot be evaluated when m_1 is executed, and hence the forward gatekeeper will be unable to record information in its logs to evaluate $\phi_{m_1;m_2}$.

3.3.2 General gatekeepers

General gatekeepers are conflict detection schemes that can capture any commutativity specification expressible in L_1 (Figure 1). For conditions which are not ONLINE-CHECKABLE, the gatekeeper will perform a rollback by executing undo methods to evaluate the commutativity condition in the appropriate state and then restore the data structure state by re-executing the forward actions. This means that the gatekeeper must keep a log of all actions in case they need to be rolled back to check commutativity. Note that this entire process must appear atomic. General gatekeepers can be thought of as forward gatekeepers that perform state rollback to evaluate any commutativity conditions that are *not* ONLINE-CHECKABLE.

A general gatekeeper for union-find: We now describe the design of a general gatekeeper that fully captures the commutativity conditions of Figure 5 for the operations union and find. The union-find gatekeeper maintains two logs: *find-reps* which stores all the active elements that have been returned by a call to find as representatives of some set and *loser-rep*, which records the result of evaluating $\mathsf{loser}(a, b)$ whenever $\mathsf{union}(a, b)$ is invoked.

For each invocation, $\mathsf{union}(a, b)$, the gatekeeper first computes $\mathsf{loser}(a, b)$. If the loser has been recorded in the *find-reps* log as the return value of an active find, then a conflict is detected. Additionally, if either of these representatives has been recorded in the *loser-rep* log as the loser representative involved in a previous union, then a conflict is detected. The gatekeeper subsequently performs the call to union and updates *loser-rep* appropriately.

In the case of an invocation $\mathsf{find}(a)$, the gatekeeper first executes the actual call on the data structure, getting the result r_a. A subtle, but important point is that if $\mathsf{find}(a)$ had executed earlier, before some active invocation of union, it may have returned a different result. This is captured in condition (2) of Figure 5 as commutativity dependent on the results of calling find in a different state. To ensure that commutativity holds, the gatekeeper undoes the effects of all potentially interfering calls to union, and re-executes $\mathsf{find}(a)$ to ensure it still returns r_a. If so, invoking $\mathsf{find}(a)$ does not violate commutativity. If not, a conflict is detected. The gatekeeper then restores the state of the data structure by re-invoking the unions. Finally, r_a is stored in *find-reps* to aid in conflict checking.

3.4 Discussion of expressivity and overheads

The three conflict detection schemes we present, abstract locking, forward gatekeeping and general gatekeeping, form a hierarchy, ranked by expressivity and overhead. The lowest-overhead scheme is abstract locking, as it merely requires acquiring a small number of locks per method invocation. Forward gatekeeping incurs more overhead as it can require a substantial amount of logging. General gatekeepers have the highest overhead, as they may perform rollbacks to evaluate complex commutativity conditions. The hierarchy of expressivity is identical: abstract locking schemes can be generated for SIMPLE specifications, while forward gatekeepers can be generated for more complex ONLINE-CHECKABLE specifications and general gatekeepers can be generated for all specifications.

Given this hierarchy of performance and precision, a natural question becomes: which scheme should we use for a particular data structure? A particular specification can be implemented using one particular scheme; can we produce different conflict detectors

that use different schemes? The next section discusses how to leverage the commutativity lattice to create different commutativity implementations in a disciplined manner.

4. Exploiting the commutativity lattice

4.1 The precision/performance tradeoff

Up to this point, we have focused on producing sound and complete implementations of commutativity specifications using a variety of conflict detection schemes. However, for a given application, a given conflict detection scheme for a data structure may have unacceptably high overheads or enable too little parallelism.

Prior work on high-level conflict detection has implicitly accepted this point. For example, in [10] the conflict detection scheme for a set explicitly does not permit concurrent adds when the arguments are identical, even though, as discussed in Section 2, doing so may allow more parallelism. Similar tradeoffs are made in [4, 18, 20]. In other words, incomplete conflict detection schemes are used in order to sacrifice parallelism for overhead. Unfortunately, it is difficult to directly produce incomplete yet sound commutativity implementations or to safely modify existing sound implementations to create other implementations.

To tackle this problem, we turn to the commutativity lattice of Section 2.4. Rather than change the conflict detection implementation directly, we can choose a less precise specification from the commutativity lattice that can be implemented in a more efficient manner. Recall that if a commutativity checker is sound with respect to this less precise specification, it must be sound with respect to the original specification.

As a simple example of this, consider using the specification $\bot$ to build a commutativity checker for any data structure. This specification is SIMPLE and trivially correct. The abstract locking synthesis algorithm will produce the correct implementation: a single, global lock that prevents transactions from accessing the data structure concurrently. This example also illustrates the precision/performance tradeoff, as the conflict checker has very low overhead but admits no parallelism.

A more compelling example is to consider the set specifications from Figures 2 and 3. While the former specification is not SIMPLE and must therefore be implemented using a forward gatekeeper, the latter *is*, and can be implemented using the abstract locking construction algorithm (which will use read/write locks on the method arguments). We can move still further down the commutativity lattice by allowing contains to commute with contains only if the arguments differ; this new SIMPLE specification will induce an abstract locking scheme that uses cheaper exclusive locks on method arguments. Crucially, *it is far easier to reason about the relationships between specifications in the commutativity lattice than it is to determine whether an abstract locking scheme constitutes a sound but incomplete refinement of a forward gatekeeper.*

While we can generate different conflict detection schemes for a given data structure in a disciplined manner using the commutativity lattice, it is unclear how to *select* the appropriate scheme. This is because the overhead and parallelism of a particular scheme is dependent on application characteristics. In fact, as we demonstrate in Section 5, in certain cases the higher parallelism schemes based on gatekeeping have lower overhead than less precise schemes!

4.2 Disciplined lock coarsening

The commutativity lattice approach also allows us to reason in a systematic way about other conflict detection optimizations. One such optimization is *lock coarsening* [17], where a data structure such as a graph is partitioned, and locks are acquired on entire partitions when any object in the partition is accessed. While this optimization seems complex, it can be formulated in a straightforward manner in terms of the commutativity lattice.

We begin with a set data structure that has been *logically partitioned*: every element that may appear in the set is assigned a partition. A helper method, part, returns the partition an object is assigned to. We can then produce a stronger commutativity specification where every clause in the specification of Figure 3 of the form $a \neq b$ is replaced by $\mathsf{part}(a) \neq \mathsf{part}(b)$. This new specification is very similar to a SIMPLE specification, except that it deals in partitions rather than individual data elements. The commutativity lattice lets us immediately see that this partition-based scheme is stronger than the original scheme, as $\mathsf{part}(a) \neq \mathsf{part}(b) \Rightarrow a \neq b$. It can be implemented in an analogous manner. Rather than acquiring locks on arguments and return values, an implementation of this new specification will acquire locks on partitions.

Partition-based specifications can be created from *any* SIMPLE specification, allowing us to automatically derive partition-based conflict checkers for many data structures. Note that these conflict checkers again trade off parallelism for overhead (intuitively, the parallelism becomes bounded by the number of partitions). Section 5 discusses this issue in more detail and quantifies the tradeoff.

4.3 Reasoning about transactional memory

The precision of transactional memory [9, 12, 19] can be reasoned about in terms of the commutativity lattice. For a particular concrete instantiation of an ADT, we can define a commutativity specification, Φ_C, that refers to the memory-level state of the data structure. By referencing memory, Φ_C can be based on establishing *concrete commutativity* rather than semantic commutativity. Two methods will commute according to this specification if executing them in either order produces the same *concrete* state. Because concrete commutativity implies semantic commutativity, it is apparent that Φ_C can be found in the commutativity lattice, but that $\Phi_C \preceq \Phi^*$.

If two transactions can execute in parallel according to a TM, they must leave the data structure in the same concrete state regardless of the order in which they execute, effectively enforcing concrete commutativity. As TM constitutes a sound (though not necessarily complete) implementation of Φ_C, we can see that a sound and complete implementation of the precise specification will always expose at least as much parallelism as TM (although potentially with higher overhead).

5. Experimental Evaluation

In this section, we present several experiments. First, we discuss a microbenchmark based on sets that allows us to explore the effects of moving between the various set commutativity specifications (and hence implementations) discussed in this paper. Then we present three case studies of real-world applications. The first, preflow push, shows the effects of switching between systematically constructed conflict detection schemes in a realistic application. The second, agglomerative clustering, shows that forward gatekeepers can deliver scalable performance. The third, Boruvka's algorithm, shows the same for general gatekeepers.

For each case study, we estimate the two parameters: the average parallelism and the overhead of conflict detection. This allows us to study the potential tradeoff between parallelism and overhead demonstrated by implementations based on different points in the commutativity lattice. These results are shown in Table 1.

We used ParaMeter [16] to estimate the average amount of parallelism in an application when using a particular conflict detection scheme. We parallelized our benchmarks using the Galois system [18][7]. We estimate the overhead of conflict detection to be the

[7] While our evaluation uses the Galois system, the techniques given for constructing conflict detectors and using the commutativity lattice are general.

Application	Variant	Path length	Parallelism	Overhead
Preflow-push	part	2789217	25.69	1.14
	ex	51978	1894.88	1.80
	ml	47558	2072.52	5.62
Boruvka	uf-ml	3678	271.89	2.5
	uf-gk	3681	271.67	1.31
Clustering	kd-ml	2209	115.88	58.76
	kd-gk	123	2018.15	2.32

Table 1. Critical path lengths, average parallelism and overheads for different applications and conflict detection schemes.

ratio between the run time of the parallelized application running on a single thread and the runtime of the original sequential version.

As a baseline, we also implemented our benchmarks using a software transactional memory, DSTM2 (version 2.1b) [11]. For each benchmark we used boosted objects [10] wherever possible (for example, the worklist), except for our target data structures, which used object-based conflict detection. Boosting allowed us to isolate any performance or parallelism differences to only the data structures under study. Each benchmark also used the best available contention manager (determined empirically). Parallelism metrics were estimated by tracking memory-level read/write conflicts in ParaMeter.

Our performance evaluations were performed on a Nehalem server running Ubuntu Linux version 8.06. The system consists of two 2.93 GHz quad-core processors. Each core has two 32KB L1 caches and a unified 256 KB L2 cache. Each processor has a shared 8 MB L3 cache, and the system contains 24 GB of main memory. The applications and the Galois system are written in Java and run on the Sun HotSpot 64-bit virtual machine, version 1.6.0.

Set microbenchmark To study the effects of using different conflict detection schemes of varying overheads and permittivity, we built a microbenchmark based on a set. In the microbenchmark, a number of threads concurrently query and update a global set. Each thread picks an arbitrary object from a shared pool and, at random, inserts it to the set (with add) or checks if it is already there (with contains). The benchmark performs 1 million operations, using one of two inputs: in the first, all examined objects are distinct, and in the second, objects are in 10 equivalence classes.

We consider four conflict detection schemes generated from specifications drawn from the set's commutativity lattice: (i) a global lock generated from the $\perp$ specification; (ii) exclusive abstract locks placed on set elements generated from the specification discussed in Section 4; (iii) r/w abstract locks generated from the specification in Figure 3; and (iv) a forward gatekeeper generated from the precise specification of Figure 2.

Table 2 shows the results of running the microbenchmark on 4 threads. For both inputs, the global lock provides poor performance: the low single thread overhead is outweighed by the non-existent parallelism (and resulting high abort rate). We note that the advantage of the gatekeeping and read/write lock schemes over exclusive locks arises when multiple transactions access the same key in the set. As this does not occur in the distinct-element input, all variants have no aborts and the low-overhead exclusive lock solution performs best. In contrast, in the repeated-element input, the advantage of allowing multiple threads to access the same element (when invoking contains, or add on an element already in the set) manifests, and gatekeeping performs the best, followed by the read/write scheme. Gatekeeping has a lower abort ratio because non-mutating adds can proceed in parallel.

Preflow-push We used preflow-push [7] to investigate the performance of different conflict detection schemes for a single benchmark. Preflow push finds the maximum flow of a network by pushing excess flow from the source towards the A worklist is initialized with the source node. In each iteration, a node with excess flow is removed from the worklist. It is *relabeled* and flow is *pushed* to

Program	(a) Distinct		(b) Repeats	
	Abort Ratio %	Time (sec)	Abort Ratio %	Time (sec)
Global Lock	48.68	4.644	44.07	3.935
Abs. Lock (Ex.)	0	1.097	1.53	1.538
Abs. Lock (RW)	0	1.365	0.09	0.818
Gatekeeper	0	1.191	0	0.697

Table 2. 4-thread results for set microbenchmark. (a) 1 million distinct elements. (b) 1 million elements in 10 equivalence classes.

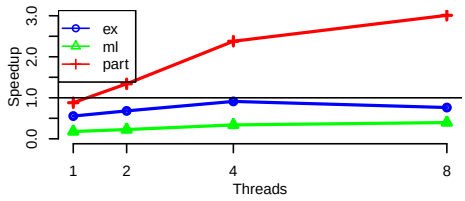

Figure 10. Preflow-push speedup (serial time 53 sec.)

neighboring nodes along edges that have remaining capacity. This increases the excess flow at the destination node, decreases the excess flow at the source, and decrements the capacity of the edge. Any nodes that gained excess flow are added back to the worklist. The algorithm terminates when no more flow can be pushed.

This algorithm is implemented in terms of a graph-like ADT, which supports relabel, pushFlow and getNeighbors. The methods relabel and pushFlow do not commute with other methods that access the same nodes, while getNeighbors commutes with itself. All the conditions in this specification are SIMPLE, and can be implemented with read/write abstract locks on nodes and edges. Note that this conflict detection strategy is identical to the conflict detection performed by a transactional memory.

We also investigate two schemes based on strengthened specifications from lower in the commutativity lattice. The first does not allow getNeighbors to commute with methods that access the same nodes; in effect, this converts read/write locks on nodes to exclusive locks. The second uses the partition-based scheme derived in Section 4.2, using 32 partitions. Table 1 shows parallelism and overhead data for the three versions of preflow push: memory level locking (*ml*), exclusive locking (*ex*) and partition-locking (*part*). As input, we use a GENRMF "challenge" input from [1].

We see the expected behavior: implementations based on specifications from higher in the commutativity lattice expose more parallelism. However, the performance results in Figure 10 show that run-time is inversely correlated with the amount of parallelism: in this application, moving down in the commutativity lattice produces lower parallelism, but lower overhead conflict checkers. This is especially interesting in the case of the partition-based conflict detector, as the critical-path length is a factor of 50 longer than the precise scheme. However, parallelism beyond the number of processors available for execution is superfluous. Since the average parallelism of partition-based conflict detection is well above the number of processors, trading parallelism for overhead is effective.

Agglomerative Clustering To study the behavior of forward gatekeeping, we implement an algorithm called *agglomerative clustering* that uses the kd-tree data structure (described in Section 2.5). The input to agglomerative clustering is (i) a set of points and (ii) a metric which measures the distance between two points. The algorithm builds a hierarchical clustering of points, called a *dendrogram*, in a bottom-up manner. The structure of the dendrogram exposes the similarity between points.

One approach to building the dendrogram was proposed by Walter *et al.* [24]. We first build a kd-tree that contains all the points. At each step of the algorithm, an arbitrary point p is chosen, and its nearest neighbor n is determined by querying the kd-tree. We then

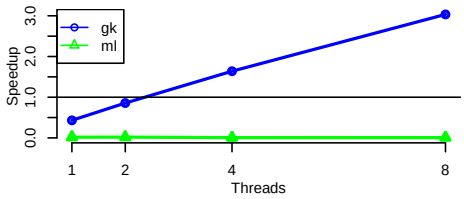

Figure 11. Agglomerative clustering speedup (serial time 7.8 sec.)

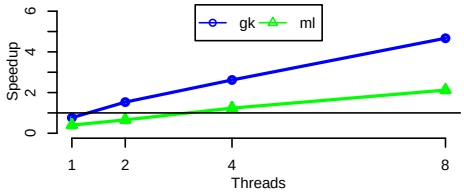

Figure 12. Boruvka's algorithm speedup (serial time 3.7 sec.)

find the nearest neighbor of n; if that is p, the two points are clustered together by removing them from the kd-tree and inserting a new point representing that cluster. The algorithm terminates when there is a single cluster; by convention, the point at infinity is the closest point if the data set contains a single point.

Table 1 shows the critical path length and average parallelism on an input of 100,000 randomly generated points for a forward-gatekeeping implementation (*kd-gk*) and the baseline implementation (*kd-ml*). The gatekeeping implementation exposes significantly more parallelism than the baseline; although the same amount of work is done, the critical path of *kd-ml* is an order of magnitude larger than that of *kd-gk*. This is because the operations on the kd-tree manipulate the bounding boxes of interior kd-tree nodes, resulting in memory-level conflicts even when the operations commute. We do not investigate an abstract locking implementation because there is no straightforward SIMPLE specification that does not merely prevent add and nearest from executing concurrently.

Figure 11 shows the performance of the gatekeeping version of agglomerative clustering as well as the transactional memory baseline, using an input of 500,000 randomly generated points. The gatekeeping conflict detector is built using the construction procedure of Section 3.3.1. Interestingly, despite the precision of the commutativity specification implemented by the forward gatekeeper (and the resulting high parallelism), it has lower overhead and better scalability than the memory-level conflict checker. This is because the gatekeeper need only track semantic information about the data structure, rather than each memory access. Clearly, forward gatekeeping, despite its complexity, is a viable approach to conflict detection for complex commutativity specifications.

Boruvka's algorithm Finally, to study the behavior of general gatekeeping, we evaluated an algorithm built around the union-find data structure (see Section 2.5), Boruvka's algorithm. Boruvka's algorithm builds minimal spanning trees (MST's) of undirected graphs. Initially, each node of the graph is in a component by itself. At each step, the algorithm (i) picks an arbitrary component, (ii) determines the lightest weight edge e connecting that component to another component, (iii) merges the two components, and (iv) adds e to the MST. The algorithm terminates when the entire graph is in one component.

At each point during the execution of the algorithm, nodes in different components are in disjoint sets. When two components are merged, the corresponding sets of nodes are as well. Therefore, a union-find data structure can be used in step (ii) to find whether or not the end points of an edge are in the same component.

Table 1 shows the parallelism and overhead of Boruvka's algorithm when run on a randomly generated 1000×1000 mesh using both a general gatekeeper (*uf-gk*), built using our construction, and our baseline TM (*uf-ml*). Curiously, we see that general gatekeeping does not offer a parallelism benefit over memory level locking. Recall that although stronger commutativity specifications will typically permit less parallelism than specifications from higher in the lattice, this is entirely dependent on application behavior. In the case of union-find, additional parallelism is exposed by a gatekeeper if multiple interfering finds are performed (as path compression prevents a memory-level approach from performing them concurrently). However, Boruvka's does not perform such finds, obviating the gatekeeper's parallelism advantage.

Interestingly, however, the general gatekeeper, despite its complexity, has fairly low overhead ($\sim 31\%$), and far less than a TM, because it need only track semantic changes to the data structure (an especially large advantage considering all of the reads and writes involved in path compression). This advantage is further pressed when scaling up, as Figure 12 shows that the general gatekeeper provides better performance than the baseline. It is apparent that, for some applications, even a conflict detection scheme as involved as general gatekeeping can provide acceptable performance.

Putting it all together

The results for our example applications demonstrate that different conflict detection schemes can provide vastly different performance. To understand the tradeoffs between different schemes, consider a simple model in terms of the following three factors:

- T = sequential runtime of algorithm,
- o_d = overhead of conflict detection scheme d, and
- α_d = average parallelism enabled by d.

$T * o_d$ is the single-threaded runtime while performing conflict detection, and $T * o_d/\alpha_d$ is the best-case parallel runtime, assuming perfect load-balance. Note that α_d and o_d correspond to the "parallelism" and "overhead" columns, respectively, in Table 1.

Both α and o are application specific. However, this model can help illuminate the choice of conflict detection. Consider two conflict detection schemes l and h for which $o_l < o_h$ and $\alpha_l < \alpha_h$ (*i.e.*, scheme l has less overhead but also permits less parallelism).

- The lower overhead, lower parallelism conflict detection scheme l offers better performance if $o_l/\alpha_l < o_h/\alpha_h$ (*i.e.*, lower overhead makes up for lower parallelism). This explains why exclusive locking outperforms read/write locking in preflow push.
- Most current systems have only a small number of processors, so the actual runtime of an algorithm on a machine with p processors may be bounded above by $T * o_d/p$. If $\alpha_l \gg p$, we should use the lower overhead, lower parallelism scheme l, since the relative amount of parallelism exposed by different conflict detection schemes ceases to be the dominant factor in performance. This explains why partition locking for preflow push is the best-performing conflict detection scheme.
- If $\alpha_h > \alpha_l$ but $o_h < o_l$ then the higher parallelism scheme actually has lower overhead. Our model suggests that in such situations, we should prefer the higher parallelism scheme. This situation arises when considering the gatekeeping implementations of kd-trees and union-find.

In short, the best conflict checker for an ADT can depend on the characteristics of both the application and the parallel machine; a single choice of conflict detector is likely not ideal for every application. However, measuring simple parameters can guide the selection of conflict checker.

While the construction techniques of Section 3 are intended to permit library writers to develop commutativity checkers for their ADTs, the commutativity lattice provides a useful interface between the library writer and the programmers who use the ADTs.

The library writer can produce a range of sound checkers for a given ADT, using the construction techniques of Section 3 and the strengthening techniques of Section 4. The lattice then allows the library writer to rank the checkers according to expressiveness. The user can then make an informed decision to move between implementations if his/her application demands more parallelism or less overhead. Alluringly, the ability to rank checkers by permittivity can allow an automated system to adaptively and dynamically select from these implementations as run-time needs change, given observations of parallelism and overhead, though we leave the design and development of such a system to future work.

6. Related work and conclusions

Related work Commutativity conditions can be seen as a highly generalized form of predicate locks as used in databases [6], where locks on arbitrary predicates are used to determine when transactions can successfully execute in parallel. Database research has been the source of a considerable number of techniques to exploit the high level properties of abstract data types, including commutativity, for concurrency control [2, 23, 25]. That work laid the foundation for subsequent work that focused on exploiting data structure semantics in speculative parallelization and synchronization systems on shared memory systems [5, 10, 18, 20].

Ni *et al.* addressed the shortcomings of memory-level locking in transactional memory systems by introducing *open nesting*, which uses abstract locks to synchronize access to data structures by exploiting their semantics [20]. However, the work focuses mainly on the mechanism of integrating abstract locking with a transactional memory, rather than on how to use abstract locks in data structures in a disciplined manner. In fact, the authors note that improper use of open nesting could lead to deadlock situations.

Kulkarni *et al.* described an approach to checking commutativity [18]. This technique applies to ONLINE-CHECKABLE specifications, but requires logging entire method invocations, and hence is not as efficient as forward gatekeeping, which can benefit from common sub-clauses in commutativity conditions.

Herlihy and Koskinen proved that commutativity information informs a disciplined, safe approach for implementing open nesting [10]. The key difference between their work and our work is that the former focused on how commuting methods could be used to enforce serializability, while we focus on implementation approaches for conflict detection. Rather than using *ad hoc* conflict detectors, we provide systematic approaches for constructing commutativity checkers, each with well-defined expressive power. Furthermore, we provide a disciplined approach to systematically generating the simpler abstract locking implementations used in prior work.

Recently, Demsky and Lam proposed using object-level *views* to synthesize high-level concurrency control [5]. Views are similar in spirit to our SIMPLE specifications, but allow for more complex interactions. As a result, their lock-generation algorithm is greedy, rather than precise. An interesting avenue of future research is to determine how a view specification is related to a commutativity specification, and whether the former can be derived from the latter.

Other recent work from Burnim *et al.* discusses verifying that method invocations on a data structure that are meant to be atomic indeed are [3]. Their notion of atomicity is semantic: two methods are atomic if they preserve certain semantic *bridge predicates* that refer to the data structure's semantic state, rather than its concrete state. This results in a notion of atomicity that is similar to semantic commutativity. Their atomicity checks are performed at compile time by performing a limited search through a space of possible parallel executions. Unlike in our work, their aim is to show that a particular pair of method invocations (with particular arguments) are atomic; they do not consider commutativity conditions, which constrain the kinds of method invocations that will appear atomic.

Kim and Rinard look at *verifying* commutativity conditions [14]. Given a definition of the abstract state of a data structure, how methods use that abstract state, and commutativity conditions between methods, Kim and Rinard's technique can verify that the commutativity conditions are, indeed, correct. This technique is complementary to the goals of our paper: we have not considered the correctness of commutativity conditions, instead relying on external techniques such as theirs to ensure that they are valid.

Future work and conclusions There are a number of promising directions for future research. First, for data structures such as union-find, determining appropriate commutativity conditions is difficult; can we *generate* commutativity conditions given a data structure specification? Second, there are interesting avenues of implementation research: although gatekeepers are expressive, they can be quite inefficient. Are there more efficient conflict detection schemes that do not overly sacrifice expressibility?

In Section 1, we posed three questions: (i) how should semantic conflict checks be implemented? (ii) can we trade off conflict checking precision for performance? and (iii) which conflict detection scheme should be used? Through our commutativity lattice framework, we showed that conflict detectors can be reasoned about in terms of a lattice of commutativity specifications. We showed how different points in the lattice can be implemented systematically by three different conflict detection schemes. We also showed how the lattice can be exploited to derive lower parallelism/lower overhead detectors by strengthening commutativity specifications. Finally we studied the tradeoffs between precision and performance and showed that our systematically generated conflict detectors can provide acceptable performance.

Acknowledgements

We would like to thank Roman Manevich for his insights and comments regarding the presentation of this paper. We would also like to thank the anonymous reviewers for their helpful feedback.

References

[1] Synthetic maximum flow families. `http://www.avglab.com/andrew/CATS/maxflow_synthetic.htm`.

[2] A. Bondavalli, N. De Francesco, D. Latella, and G. Vaglini. Shared abstract data type: an algebraic methodology for their specification. In *MFDBS 89*, 1989.

[3] J. Burnim, G. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *ASPLOS'11: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[4] B. D. Carlstrom, A. McDonald, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPoPP*, 2007.

[5] B. Demsky and P. Lam. Views: object-inspired concurrency control. In *ICSE*, 2010.

[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[7] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*. Oct 2003.

[10] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In

PPoPP, 2008.

[11] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, 2006.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[14] D. Kim and M. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *PLDI*, 2011.

[15] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, 2010.

[16] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP*, 2009.

[17] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS*, 2008.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[19] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[20] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

[21] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[22] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI*, 1995.

[23] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, 1984.

[24] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast agglomerative clustering for rendering. In *Interactive Ray Tracing*, 2008.

[25] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12), 1988.

A. Proof of serializability

Here we prove the serializability claim made in Section 2.1. The theorem is stated in terms of an execution history consisting of two transactions A and B, which each start at the beginning of the history and finish at the end of the history (*i.e.*, the transactions fully overlap and are coextensive with the history itself). The theorem can easily be extended to longer histories, with multiple transactions that only partially overlap. The theorem proceeds by showing how to construct a serial schedule when commutativity conditions have been shown true for all pairs of method invocations.

We begin by showing that if we can show a commutativity condition to be true in one history H, then in any C-EQUIVALENT history H' there exists some commutativity condition over the same method invocations that is true. This allows us to claim that even as we rearrange invocations while constructing a serial schedule, we can assume that the commutativity conditions are true.

LEMMA 1. *If the following three conditions hold:*

- *History H contains the invocations $\langle m_1(v_1)\rangle_{\sigma_1}$ and $\langle m_2(v_2)\rangle_{\sigma_2}$*
- *There exists a* C-EQUIVALENT *history H' with invocations $\langle m_1(v_1)\rangle_{\sigma_3}$ and $\langle m_2(v_2)\rangle_{\sigma_4}$*
- *$\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2)$ is a commutativity condition in H*

then there exists a commutativity condition $\phi'_{m_1;m_2}(\sigma_3, v_1, r_1, \sigma_4, v_2, r_2)$ in H' such that $\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2) \Leftrightarrow \phi'_{m_1;m_2}(\sigma_3, v_1, r_1, \sigma_4, v_2, r_2)$.

PROOF. We begin by assuming that H and H' exist according to the conditions of the lemma, and that $\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2)$ is a commutativity condition. There clearly exists a predicate $\phi'_{m_1;m_2}(\sigma_3, v_1, r_1, \sigma_4, v_2, r_2)$ that is true if and only if $\phi_{m_1;m_2}$ is true (though it may not be expressible in a logic such as L_1). All that remains is to show that this predicate is, indeed, a commutativity condition (in other words, that when it is **true**, the conditions of Definition 3 are satisfied).

We begin by constructing the set of histories that are C-EQUIVALENT to H and in which $m_1(v_1)$ and $m_2(v_2)$ appear consecutively:

$$S(H) = \{H'' | H'' \equiv_C H \wedge \exists \text{ sub-history } h \text{ of } H'' = \langle m_1(v_1), m_2(v_2)\rangle_\sigma\}$$

We have assumed that $\phi'_{m_1;m_2}$ is **true**, and hence so is $\phi_{m_1;m_2}$. It follows from the definition of commutativity conditions that if $\phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2)$ holds, then all sub-histories of the form $\langle m_1(v_1), m_2(v_2)\rangle_\sigma$ found in the histories of $S(H)$ are equivalent to their swapped sub-histories, $\langle m_2(v_2), m_1(v_1)\rangle_\sigma$. Because the C-EQUIVALENCE relation forms an equivalence class, the set $S(H') = S(H)$. Thus, $\phi'_{m_1;m_2}$ is a valid commutativity condition in H'. □

We can now prove the following theorem:

THEOREM 2. *If, in an execution history, H, consisting of method invocations from two transactions, A and B, for each pair of method invocations $\langle m_a^A(v_a)\rangle_{\sigma_a}$ (that returns r_a) and $\langle m_b^B(v_b)\rangle_{\sigma_b}$ (that returns r_b), the commutativity condition $\phi_{m_a;m_b}(\sigma_a, v_a, r_a, \sigma_b, v_b, r_b)$ holds, then there is an equivalent history H' where all the invocations of B appear before all the invocations of A.*

PROOF. The proof proceeds by induction on the length of the sub-history, h, starting with the first method invoked by A in H, $m_1^A(v_1)$, and ending with the last method invoked by B in H, $m_2^B(v_2)$. We can disregard invocations from B that occur before $m_1^A(v_1)$, as they already appear before all invocations from A, and likewise for invocations from A that occur after $m_2^B(v_2)$. If there is no such sub-history, then H is already serialized.

Base case: When a sub-history $h = \langle m_1^A(v_1), m_2^B(v_2)\rangle_\sigma$, we can directly apply the commutativity condition $\phi_{m_1;m_2}$ to produce a C-EQUIVALENT history $h' = \langle m_2^B(v_2), m_1^A(v_1)\rangle_\sigma$.

Inductive hypothesis: When a sub-history h contains n or fewer invocations from A and m or fewer invocations from B, we can construct a sub-history $h' \equiv_C h$ that has all invocations from B before all invocations from A.

Inductive step: We will look at two cases: one where there are $n+1$ invocations from A and m invocations from B in h, and a second where there are n invocations from A and $m+1$ invocations from B in h.

Case 1: Consider the sub-history of h, i, that is the suffix of h starting with the last invocation from A in h:

$$i = \langle m_{n+1}^A(v_{n+1}), m_k^B(v_k), \ldots, m_m^B(v_m)\rangle_\sigma$$

i has at most m invocations from B and 1 invocation from A, and hence the inductive hypothesis applies. We can construct $i' \equiv_C i$ where all the invocations from B appear first. Substituting i' for i in h gives us a C-EQUIVALENT history h'.

The commutativity conditions provided by the conditions of the theorem only apply to h, not to h'. By appealing to Lemma 1, we know there are commutativity conditions between the pairs of methods in h' that are all **true**. Note that h' has at least one method invocation from A at the very end. We can now consider the sub-history j of h' that excludes that last method invocation. This new sub-history has n invocations from A and m invocations from B, allowing us to apply the inductive hypothesis again, constructing an equivalent j' in which all the invocations from A are at the end.

Substituting j' for j in h' gives us a new sub-history h'' that is equivalent to h but has all the invocations from B occurring before any invocations from A, proving this case.

Case 2: This case proceeds analogously to Case 1.

We have thus shown that there exists a sub-history h' that is C-EQUIVALENT to sub-history h for sub-histories of any length. Substituting h' for h in the complete history H produces a C-EQUIVALENT history H' that has all invocations from A after all invocations from B, and hence is serialized. □

B. Proof of Theorem 1

In this section, we provide the proof of Theorem 1 (that a sound a complete abstract locking scheme exists for a specification iff that specification is SIMPLE). The proof proceeds in two parts. First, we show that the construction algorithm presented in Section 3.2 produces a sound and complete implementation of a SIMPLE commutativity specification. Then, we show that if a specification is not SIMPLE, there does not exist a sound and complete abstract locking scheme for it.

B.1 Soundness and completeness

LEMMA 2. *The abstract locking construction presented in the paper produces an abstract locking scheme that is sound and complete with respect to a* SIMPLE *commutativity specification.*

PROOF. We will use the contrapositive definitions of soundness and completeness in this lemma, as described below.

- *Soundness.* For an abstract locking scheme to be a sound implementation of a commutativity specification, if two method invocations are allowed to proceed by the scheme (*i.e.*, both can successfully acquire their abstract locks), then the methods must commute according to the specification. By contrapositive, if two method invocations *conflict* according to the specification, then one or more abstract lock acquires performed by the scheme must fail, triggering a conflict.
- *Completeness.* For an abstract locking scheme to be complete, we must show that if two method invocations commute according to the specification, the abstract locking scheme will not detect a conflict. By contrapositive, if the abstract locking scheme detects a conflict between two method invocations (because one cannot acquire its abstract locks), then the methods must conflict according to the specification.

Because each method acquires its abstract locks in separate modes, it suffices to address the soundness and completeness of each commutativity predicate in isolation. For a commutativity specification to be SIMPLE, all commutativity predicates must satisfy one of three conditions. We address the three cases separately.

Case 1: $\phi_{m_1;m_2} = \mathbf{false}$. In this case, the m_1 and m_2 always conflict. We see that our lock compatibility table requires that the lock modes on the data structure lock used by the two methods interfere, hence preventing the second method from successfully acquiring the data structure lock. Thus the implementation is sound. Completeness is trivial, as the methods always conflict.

Case 2: $\phi_{m_1;m_2} = \mathbf{true}$. In this case, m_1 and m_2 *never* conflict. For soundness, this case is uninteresting; regardless of the behavior of the abstract locking scheme, the implementation will be sound. Completeness requires that the abstract locking scheme allow m_1 and m_2 to be successfully invoked by concurrently executing transactions. Because every method acquires locks in its own modes, and lock modes are compatible by default, we see that our abstract locking scheme lets m_1 and m_2 proceed.

Case 3: $\phi_{m_1;m_2} = ... \wedge (x \neq y) \wedge ...$. To show soundness, consider the negation of this condition: a disjunction of equalities of the form $x = y$. If any of the equality clauses are true, our abstract locking scheme triggers a conflict. The equality of arguments means that the two method invocations will attempt to acquire locks on the same data item, and our compatibility matrix construction guarantees that the modes in which that lock will be acquired are incompatible.

To show completeness, we note that if our scheme detects a conflict, then two method invocations must have acquired the same argument lock in conflicting modes. By our construction, this can only happen if the commutativity condition contains a clause of the form $x \neq y$. Hence, the *conflict* condition contains the clause $x = y$, and the two methods conflict according to the specification.

Because the abstract locking scheme instantiated by our algorithm for a SIMPLE commutativity specification is sound and complete with respect to each commutativity predicate, the overall scheme is sound and complete. □

B.2 Expressivity of abstract locking schemes

Lemma 2 shows that if a commutativity specification is SIMPLE, we can produce a sound and complete abstract locking scheme for it by applying our construction algorithm. However, it is unclear whether sound and complete abstract locking schemes exist for non-SIMPLE commutativity specifications. The following lemma shows that they do not.

LEMMA 3. *If a commutativity specification is* not SIMPLE*, then there does not exist a sound and complete abstract locking scheme that implements it.*

PROOF. We proceed by contradiction. Assume that there is a non-SIMPLE commutativity specification, Φ, with a sound and complete abstract locking implementation. From this abstract locking scheme, construct Φ' as follows: For each pair of methods m_1 and m_2, consider at the compatibility sub-matrix induced by the locks acquired when invoking each method. If two argument locks, a and b are incompatible, the commutativity condition $\phi_{m_1;m_2}$ contains the conjunct $a \neq b$. If data structure locks acquired by the methods are incompatible, $\phi_{m_1;m_2}$ is simply $\mathbf{false}$.

Note that Φ' is clearly SIMPLE, and the abstract locking scheme is a sound and complete implementation of Φ' by construction (specifically, applying the abstract locking construction algorithm to Φ' will produce the original abstract locking scheme). As the locking scheme is also sound and complete with respect to Φ, Φ and Φ' must be logically equivalent, contradicting the assumption that Φ is non-SIMPLE. □

Theorem 1 follows directly from Lemmas 2 and 3.