Conflict Abstractions and Shadow Speculation for Optimistic Transactional Objects

Thomas Dickerson[1], Eric Koskinen[2]*, Paul Gazzillo[3]**, and Maurice Herlihy[1]

[1] Brown University***
[2] Stevens Institute of Technology
[3] University of Central Florida

Abstract. Concurrent data structures implemented with software transactional memory (STM) perform poorly when operations which do not conflict in the definition of the abstract data type nonetheless incur conflicts in the concrete state of an implementation. Several works addressed various aspects of this problem, yet we still lack efficient, general-purpose mechanisms that allow one to readily integrate black-box concurrent data-structures into existing STM frameworks.
In this paper we take a step further toward this goal, by focusing on the challenge of how to use black-box concurrent data structures in an *optimistic* transactional manner, while exploiting an off-the-shelf STM for transaction-level conflict detection. To this end, we introduce two new enabling concepts. First, we define data-structure conflict in terms of commutativity but, unlike prior work, we introduce a new format called *conflict abstractions*, which is kept separate from the object implementation and is fit for optimistic conflict detection. Second, we describe *shadow speculation* for wrapping off-the-shelf concurrent objects so that updates can be speculatively and opaquely applied—and even return values observed—but then later dropped (on abort) or else atomically applied (on commit). We have realized these concepts in a new open-source transactional system called ScalaProust, built on top of ScalaSTM and report encouraging experimental results.

Further detail and experimental results can be found in the extended version of this paper. [8]

1 Introduction

Modern software transactional memory (STM) systems typically perform synchronization on the basis of *read-write* conflicts: two transactions conflict if they access the same memory location, and at least one access is a write. It is well understood that this technique works poorly for contended data objects because operations that could have correctly executed concurrently are deemed to conflict, causing unnecessary rollbacks and serialization.

Some prior works were aimed at this problem and found solutions to some cases. Transactional Boosting [17] centers around constructing a transactional

* Partially supported by NSF Award #1618542.
** Partially supported by NSF Awards #1618542 and #1840934.
*** Partially supported by NSF Awards #1561807 and #1908806

"wrapper" for legacy thread-safe concurrent data structures. Designing a boosting wrapper requires identifying which operations commute, as well as providing operation inverses. Boosting can take advantage of existing thread-safe libraries, so there is no need to re-invent the wheel, but is limited to pessimistic treatment of object operations. Hassan *et al.* [14] provide an optimistic strategy, but requires white-box access to the data-structure. Transactional Predication [4] maps semantic conflicts onto read-write conflicts handled by an underlying STM. Predication can exploit highly-optimized mechanisms provided by off-the-shelf STM systems, but applies only to sets and maps. Software Transactional Objects (STO) [18] is an STM design that provides built-in primitives to track conflicts among arbitrary operations, not just read-write conflicts. Similarly, Transactional Data Structure Libraries [30] describes techniques for building libraries of transaction-aware data structures. The latter two works do not readily support existing concurrent ADT implementations (*e.g.* `java.util.concurrent`), which would be appealing because these implementations are highly optimized.

Despite the advances noted above, we still lack general approaches to building transactional systems that exploit both the conflict resolution of state-of-the-art STM systems, as well as the high performance of off-the-shelf concurrent abstract data type (ADT) implementations. Here is an example: imagine we wanted to use an off-the-shelf concurrent priority queue that supported efficient (copy-on-write) snapshots, but had no efficient inverse for `insert`. These seemingly simple requirements escape all prior techniques. Predication [4] doesn't quite fit the bill because it is limited to sets/maps. Optimistic boosting [14] requires white-box access to the data-structure. Boosting [17] could be made to work with an inefficient synthetic inverse; however, it would still require pessimistic synchronization, which isn't a good fit for most STMs. Thus, new abstractions must be developed to support efficient use of ADTs with STM systems.

This paper takes a step further toward this goal: we address the challenge of how to allow updates to black-box highly concurrent objects to be performed optimistically, while exploiting off-the-shelf STMs. To this end, we introduce two new key concepts—*conflict abstractions* and *shadow speculation*—which, together, enable programmers to build such transactional systems.

We first consider the challenge of defining and detecting conflict. Conflict between ADT operations is typically understood in terms of commutativity specifications [2,31,23,17] which are implementation-independent, but aren't easily translated into code. To resolve this tension, we introduce an approximation of commutativity, called *conflict abstractions* specifically fit for *optimistic* synchronization. We build upon the idea that commutativity-based conflict specifications can be kept separate from object itself, by extending the concept to keep *implementations* of commutativity-based conflict detection separate from the implementation of the data structure. In fact, we can even use the STM itself to detect non-commutativity, even when the data structure implementation doesn't use the STM at all. The principal advantage is that a programmer can readily integrate an off-the-shelf concurrent object into a transactional setting, without knowing the complex implementation details of the object. Instead, the programmer simply needs to understand the abstract type.

Conflict abstractions can be used with optimistic STMs (such as ScalaSTM) to enable optimistic commutativity-based conflict detection. But now how do we cope with operations being speculatively applied to the objects themselves? One could potentially delay the application of operations to commit time, but what about operations that involve return values that are needed by the transaction to continue? This requires the ability to predict the effects of operations which have not yet been applied. To this end, we introduce the idea of *shadow speculation*, allowing a transaction to speculatively apply ADT operations to an object while ensuring the updates cannot be viewed by concurrent transactions. These updates can be atomically applied at commit time or else discarded in the case of an abort. This is achieved by first tracking operation replay logs. We then describe two strategies (based on *snapshots* and *memoization*) that each allow transactions to maintain their own shadow of a shared data structure and observe return values of their speculative operations. By combining these shadow copies with commutativity-based conflict abstractions, we enable non-commutative operations to be applied speculatively to off-the-shelf ADTs and in a way that is opaque to concurrent transactions.

We have incorporated these ideas into a new transactional object system called ScalaProust[4], built on top of ScalaSTM. ScalaProust, unlike predication, goes beyond sets/maps and can support objects of arbitrary abstract type such as priority queues and non-zero indicators. Meanwhile, unlike boosting, ScalaProust allows optimistic synchronization and integrates with the underlying STM, to take advantage of well-engineered STM conflict-detection mechanisms. While the ScalaProust tool also supports pessimistic updates, this paper focuses on contributions pertaining to optimistic updates.

In summary, we make the following contributions:

1. *Conflict abstractions* provide a novel way to concretely realize an abstract data type's semantic notions of conflict so it can efficiently cooperate with a generic software transactional memory run-time (Section 3).
2. *Shadow speculation* allows individual transactions to make private speculative updates to highly-concurrent black-box objects (Section 4).
3. *The* ScalaProust *transactional system*,[5] built on top of ScalaSTM and combines off-the-shelf ADTs with existing STMs (Section 5).
4. *An experimental evaluation* demonstrates scalability competitive with existing specialized approaches such as transactional predication, but with a wider range of applicability (Section 6).

Limitations. The mechanisms described in this paper are designed for transactional objects and currently don't support mixtures between transactional

[4] This name is a portmanteau of *predication* and *boosting*, both influential prior works. The name is also an *hommage* to Marcel Proust, an author famous for his exploration of the complexities of memory.

[5] `www.github.com/ScalaProust/ScalaProust/`

objects and STM-managed read/write operations. We leave this to future work. This paper makes conceptual contributions and experimental demonstrates their impact. A proof of opacity could perhaps be achieved by adapting existing theoretical models (*e.g.* [22]), another important step for future work.

2 Overview

We now highlight the key ideas of this paper with two example concurrent ADTs—a priority queue and a non-zero indicator (NZI)—and describe how *conflict abstractions* and *shadow speculation* allow black box implementations of the ADTs to be used optimistically with an off-the-shelf STM.

2.1 From Commutativity to Conflict Abstractions

Let us first consider the priority queue ADT, supporting the three operations `min()/x`, `removeMin()/x`, and `insert(x)`. We assume that the programmer already has a concurrent implementation (*e.g.* from `java.util.concurrent`). Moreover, like in transactional boosting [17], we will first require the programmer to be aware which operations commute under which circumstances. (Recent work has shown that commutativity can be synthesized from the ADT's specification [1].) We say that two ADT operations *commute* provided that they lead to the same final state and return the same values, regardless of the order in which they are applied. As a reminder, the table to the right summarizes sound commutativity conditions for pairs of priority queue operations. For a more complete collection of commutativity conditions of ADTs, see [19,1]. In the above example, `insert(42)` always commutes with `removeMin()/1` because the value inserted (42) was greater than the minimum value (1) in the priority queue. Also, insertions always commute because the internal order of the inserted elements will be dictated by their values.

	`min()/`x	`removeMin()/`x	`insert(`x`)`
`min()/`y	true	false	$y \leq x$
`removeMin()/`y	false	$x = y$	$y \leq x$
`insert(`y`)`	$y \geq x$	$y \geq x$	true

While commutativity specifications benefit from being independent of the implementation, they are difficult to translate into program source code. In the pessimistic setting, transactional boosting [17] uses so-called abstract locking and, for priority queues, gives an example of a single read/write lock to approximate commutativity. The challenge remains: how can we use commutativity specifications as the basis for *optimistic* conflict detection and, moreover, can we exploit black-box optimistic STMs to perform this abstract conflict detection?

Toward this challenge, we begin by introducing *conflict abstractions*. The idea is to approximate commutativity-based conflict detection by using the STM itself, keeping the implementation of conflict detection separate from the implementation of the ADT (which may not even use the STM at all). Let's say thread T_1 would like to perform `min()` and thread T_2 would like to perform `removeMin()`. The (logical) commutativity of these operations tells us that we should assume these operations conflict. The idea of a conflict abstraction is to represent this logical notion of conflict by introducing *concrete* STM-managed variables and rules for when those variables should be read/written so that the STM will detect a conflict when these two transactions try to proceed with non-commutative operations. As a trivial example, we could create a new variable v, and require T_1 to read v and T_2 to write (some random fresh value) to v. In this way, the `read`(v) summarizes the logical "read-only" nature of `min`, while the `write`(v) summarizes the logical update made by `removeMin()`. Notice that we have now (in a limited way) tricked the STM to perform commutativity-based conflict detection and have not had to touch the internals of the priority queue. Note that, in some cases, this new variable v could potentially be removed by a compiler, and so we must protect v with an annotation such as `volatile`.

Let's now generalize beyond this single-variable example. The idea is that threads summarize the ADT operations they plan to perform—a sort of *digest*—through a few read/write operations on some freshly-introduced STM-managed variables. The primitives in this digest are chosen to reflect various conceptual aspects of the object's abstract state (*e.g.* a priority queue's minimum value, size, and multiset). This digest, if written correctly, is such that whenever the ADT operations being performed by two threads do not commute, operations on the digest primitives will be found to conflict. This mapping of abstract state to STM variables, and the rules for which to read and which to write—as a function of the ADT operation being performed—is what we call a *conflict abstraction*. Here is a conflict abstraction for the priority queue ADT:

Conflict Abstraction for Priority Queue
CA STM vars: $v_{min}, v_{incr}, v_{decr}$, with CA operation rules:

`min()`$/x$	$: rd(v_{min})$
`removeMin()`$/x$	$: wr(v_{decr}); wr(v_{min})$
`insert`(x)	$: wr(v_{incr});$ `if` $(x <$ `min()`$)$ $wr(v_{min})$ `else` $rd(v_{min})$
`size()`$/n$	$: rd(v_{decr}); rd(v_{incr})$

In this conflict abstraction (CA), we use STM-managed variables v_{min}, v_{incr}, and v_{decr}. Intuitively, v_{min} summarizes whether operations are somehow dependent upon the minimum element. Writing to variable v_{incr} summarizes whether the operation increases the size of the queue, while reading from v_{incr} indicates that the operation is sensitive to whether the size will increase. v_{decr} is similar. Notice that if we take *any* initial state, and consider *any* pair of ADT operations, if the CA operation rules are followed, then the STM will detect some kind of conflict on at least one of the memory locations v_{min}, v_{incr} or v_{decr}.

As an example, let's say that we have operations T_1 : `removeMin()`$/42$ and T_2 : `insert`(1). In general these operations do not commute because the element being inserted is less than the current minimum value so, depending on the order of the operations, T_1 will observe different values (and the final state of the ADT will be different). Following the CA operation rules, T_1 will write v_{decr} and write v_{min}. Meanwhile, T_2 will write v_{incr} and either read or write v_{min}, depending on the ADT's current `min`. (Here `min` is a another ADT method, which will itself perform a read on v_{min}.) Off-the-shelf STMs will detect some kind of conflict, *e.g.*, a write/read or write/write conflict on v_{min}, effectively doing the work of non-commutativity detection. On the other hand, let's assume that initially 33 is the minimal element of the priority queue and consider two commutative operations: T_1 : `insert`(42) and T_2 : `min`/33. In this case T_1 will write v_{incr} and read v_{min}, while T_2 will read v_{min}. An off-the-self STM won't detect any conflicts (two reads on v_{min} don't conflict), correctly reflecting that these abstract ADT operations commute.

This approach is not limited to priority queues. Let's consider a second example: a Counter that is capable of non-zero indication (NZI), as inspired by Ellen *et al.* [10]. Like the priority queue, this is a standard ADT, but not a map/set-like structure required by predication [4]. NZI provides three operations: `inc()`, `dec()`/p, `zero()`/p, where `dec()` returns a flag indicating if the operation failed because the NZI was already zero. The commutativity is to the right. Two `inc()` operations are independent, as are two `zero()` operations.

	`inc()`	`dec()`/p	`zero()`/p
`inc()`	true	$\neg p$	$\neg p$
`dec()`/q	$\neg q$	$q = p$	$q = p$
`zero()`/q	$\neg q$	$q = p$	true

Naturally, an `inc()` may alter the return value of `zero()` and `dec()` which further complicates matters. In these cases, commutativity depends on the return values of `dec()` and `zero()`. Once again, we cannot directly use this commutativity specification, because it is not in a format readily understood by STMs. The following corresponding conflict abstraction can be:

Conflict Abstraction for Non-Zero Indicator (NZI)
CA STM vars: v_{zero}, with CA operation rules:

```
inc()      : if (zero()) $wr(v_{zero})$ else $rd(v_{zero})$
dec()/$q$  : if (willBeZero()) $wr(v_{zero})$else $rd(v_{zero})$
zero()/$q$ : $read(v_{zero})$
```

For NZI, one can use a *single* STM memory location v_{zero} to summarize the abstract conflict. As we discuss in Section 3, one can construct a CA differently, depending on the ADT and how finely grained one would like to characterize conflict. Taking an example of T_1 : `inc()` and T_2 : `zero()`, it is easy to see that an STM will detect conflict on v_{zero}, depending on whether the NZI is zero. Notice that we have used `zero()` which, itself is an operation. The ScalaProust system, outlined in Section 5, is able to support CAs that, themselves contain other method calls, by collecting transitive dependencies. Our conflict abstraction above also used another helper method `willBeZero()`. This function depends not only the NZI ADT's current state, but also on potential future states, to characterize its commutativity. In the Section 4 we will describe how we support such helper functions to examine aspects of the state (and even predicted state) and enable more precise conflict abstractions.

While this paper focuses on optimism, as a side node, conflict abstractions can also be used for pessimistic conflict detection, by defining boosting-like abstract locks. Each CA variable can instead be a lock and the conflict abstraction indicates whether the lock should be acquired in read or write mode.

2.2 Support for Shadow Speculation

While conflict abstractions provide a route to optimistic, commutativity-based conflict detection, the question remains: is it safe to perform the ADT operations optimistically? The answer is, of course, no. Optimistic transactions may abort and, to ensure opacity, their uncommitted effects must not be observed by concurrent transactions. The next idea of this paper—called *shadow speculation*—allow one to take an off-the-shelf ADT implementation and perform speculative updates on it, and even view return values. Our strategy makes these speculative updates invisible to concurrent transactions and permits them to be either discarded (on abort) or atomically applied (on commit). In Section 4 we describe how to achieve shadow speculation using a combination of wrappers, operation replay logs, and one of two techniques to predict values: *fast snapshots* and *memoization*.

2.3 The ScalaProust Transactional System

With conflict abstractions and shadow speculation, we now have a path to use black-box ADTs in an optimistic setting, with black-box optimistic STMs. In Section 5 we discuss ScalaProust, built on top of ScalaSTM [5].

In Section 6 we conclude with an evaluation, demonstrating that black-box ADT implementations can be used on top of high-performance STMs with optimistic read/write conflict detection. Moreover, we can obtain performance that is on the order of transactional predication, yet permits a more expressive class of objects (beyond map/set-like structures).

2.4 Related Work

In Section 1, we noted prior works including transactional boosting [17], transactional predication [4], optimistic boosting [14], software transactional objects [18], and transactional data structure libraries [30]. While these prior works were sources of inspiration, each of them tackled slightly different problems. The concepts of conflict abstractions and shadow speculation described here are novel, as well as our new ScalaProust transactional system.

Two aforementioned recent works aimed at developing data-structure *implementations* from the ground-up so that they are amenable to a transactional setting. Herman *et al.* [18] build on top of a core infrastructure that provides operations on version numbers and abstract tracking sets that can be used to make object-specific decisions at commit time. Spiegelman *et al.* [30] describe how to build data-structure libraries using traditional STM read/write tracking primitives. In this way, the implementation can exploit these STM internals. Unlike these prior works, our aim is to reuse existing linearizable objects and exploit the decades of hard-work and ingenuity that went into their implementations.

In recent years, it has been shown the commutativity can be verified [19] or even synthesized [1] from ADT specifications. Early work on exploiting commutativity for concurrency control includes Korth [20], Weihl [32], CRDTs [29], and Galois [24]. Some false conflicts in STMs can be alleviated by other escape mechanisms such as open nesting [25], elastic transactions [11], and transactional collection classes [6]. Other mechanisms that exploit commutativity include automatic semantic locking [13] and dynamic race detection [9].

3 Conflict Abstractions

The principal challenge for any type-specific transactional object implementation is how to map type-specific notions of conflict into a low-level synchronization framework. Like others [4,17,21,22], we identify type-specific synchronization conflicts with a *failure to commute*: two operations commute if applying them in either order yields the same return values and the same final object state. In this section, we describe *conflict abstractions* which permit optimistic transactional conflict detection, without exposing the internals of a black-box object. Our approach symbolically represents aspects of the object's abstract state as STM-managed memory locations, kept separate from the ADT implementation itself.

We will use the following definitions. $\mathcal{M}$ are the set of object *methods* $o.m, o.n$, etc. A method *signature* is denoted $o.m(\bar{x})$ where $\bar{x}$ represents the vector of arguments to method m. $\mathcal{A}$ are method argument *values*. We denote a vector of argument values as $\bar{\alpha}$ where each element α is the value for the corresponding element in $\bar{x}$ (as denoted earlier in this paper). An *invocation* is an application of a method to a vector of arguments, $o.m(\bar{\alpha}), o.n(\bar{\beta})$, etc. Σ_o is the *abstract* state space for object o; we do not need to model the implementation of o. We also assume that the object provides (or can be extended to provide) various read-only methods that permit a transaction to query aspects of the object's abstract state, such as $o.\texttt{size}()$, etc. Finally, we write $P : \Sigma \to \mathbb{B}$ to be the type of a state predicate.

As discussed in Section 2.1, a conflict abstraction (CA) is a way of approximating commutativity by summarizing the effects of black-box object methods using a series of memory operations. More precisely,

Definition 1 (Conflict abstraction). *A conflict abstraction is a pair* (X, f) *where* X *is a finite set of variables and* $f : \mathcal{M} \to \mathcal{A} \to \Sigma \to (P \times X \times \{\mathsf{rd}, \mathsf{wr}\})\,\mathsf{list}$.

Intuitively, a conflict abstraction first has a set of abstract locations X, representing STM-managed memory (or locks if used pessimistically). For a given object method $o.m(\bar{\alpha})$ with arguments $\bar{\alpha}$ and object state σ_o, the conflict abstraction function f returns a list of $(p, x, mode)$ tuples. Each tuple consists of a condition p, a location x and a mode (read or write). For each tuple, if the condition p holds, then the thread is instructed to access location x with the given read-vs-write mode. Recall from Section 2.1 the priority queue example. We can now define the conflict abstraction so that $f(o.insert, [1], \sigma_{pq}) = \{(\mathsf{true}, v_{incr}, \mathsf{wr}), (1 < o.\mathsf{min}(), v_{min}, \mathsf{wr}), (1 \geq o.\mathsf{min}(), v_{min}, \mathsf{rd})\}$. That is, the transaction is instructed to write to v_{incr} and then read or write to v_{min} depending on whether 1 is less than the current minimum value. (Note $o.\mathsf{min}()$ appears in the CA of $o.\mathsf{insert}()$, so $o.\mathsf{insert}()$'s CA depends on $o.\mathsf{min}()$'s.) Similarly, we let $f(o.min, [], \sigma_{pq}) = \{(true, v_{min}, \mathsf{rd})\}$. In Section 5 we describe how these conflict abstractions are used inside "wrappers" so that transactions perform these STM read/write operations just before the corresponding operation and again before commit.

The impact of conflict abstractions is that we can leverage an STM to perform transactional conflict detection, even though the ADT is treated as black-box. In the above example, the STM will detect a read/write conflict on v_{min} and we have enabled efficient STMs to do the work of conflict detection.

Conflict abstractions have several benefits over conflict strategies based on abstract locks [17] or commutativity alone. The format of a conflict abstraction is more algorithmic and less declarative than prior strategies. A programmer will already have at least an intuitive understanding of the black-box object's abstract state, and it is easier to translate this into a series of STM locations and read/write operations. This avoids the need to think about pair-wise reasoning (as in commutativity or abstract locks) upfront: one instead simply considers the effects of each operation independently. Later, one can verify the correctness of their conflict abstraction through pair-wise reasoning (see discussion below).

Notice that a conflict abstraction can be more fine-grained or more coarse-grained with respect to how it represents the object's abstract state. A trivial coarse-grained conflict abstraction would have cardinality 1 and use a single STM location x, and map all read-oriented object methods to read x and map all object mutator methods to write x. While simple and correct, the downside is of course that concurrency may be lost. The choice of granularity (cardinality) is often specific to the data structure and the workload. Regardless, it is important that the conflict abstraction be correct:

Definition 2 (Correctness). *A conflict abstraction* (X, f) *is* correct *provided that for every* $m(\bar{\alpha})$ *and* $n(\bar{\beta})$ *that do not commute, and every* σ_o*, there exists some* $(p, v, m_1) \in f(o.m, \bar{\alpha}, \sigma_o)$ *and* $(q, v, m_2) \in f(o.n, \bar{\beta}, \sigma_o)$ *such that* $p(\sigma_o)$ *and* $q(\sigma_o)$ *and either* $m_1 = \mathsf{wr}$ *or* $m_2 = \mathsf{wr}$*.*

A CA is correct if, for any pair of non-commutative method invocations, there will be some location with either a read/write or write/write conflict.

Verifying Conflict Abstractions. Existing software verification tools can verify the correctness of a conflict abstraction. Specifically, the question of correctness can be reduced to satisfiability, fit for reasoning with SAT/SMT tools. We do not need the ADT's implementation; instead, it is sufficient to work with a model (or sequential implementation) of the abstract data type. As done previously [1], it is easy to model a variety of ADTs in SMT.

Once we have modeled object methods m and n, we further model conflict abstractions. SMT reasoning then proceeds by asserting the following series of constraints: (1) Method m performs its conflict abstraction reads/writes. (2) Method m performs its data-structure operation. (3) Method n performs its conflict abstraction reads/writes. (4) No read/write or write/write conflict occurs. (5) Method n performs its data-structure operation. We now need to ensure that the resulting state is the same as it would have been if the operations executed in the opposite order. Using different variable names for the intermediate states, we then assert the other order (n before m). Finally, we assert that the results (return values and final state) were different and check whether this is satisfiable. If it is not satisfiable, then the conflict abstraction is correct.

Other ADTs. To highlight the generality of our approach, we now describe conflict abstractions for some other ADTs.

- *Stack.* Since stack operations are typically focused only on the top element, most operations conflict. A suitable conflict abstraction can consist of a single variable v, where both `push` and `pop` write to v. If the stack supports a `peek` operation (*i.e.* inspecting the top element without removing it), then `peek` can simply perform a read of v, enabling concurrent `peek` operations. A more sophisticated conflict abstraction could take into account the *values* on the stack.
- *Sets and Map.* A conflict abstraction for a Set or Map can use a strategy similar to boosting [17]. Since the number of elements/keys could be large, one may not want a CA that separately tracks each element or key. Instead, some smaller number N of CA locations can be used and, when an element e is accessed, the CA can instead read or write location $v_{e\%N}$. The choice of N can depend on the workload. Naturally, `put(k,v)`, for example, would write to location $v_{k\%N}$, while `get(k)` would read.
- *Directed Graph.* Consider a directed graph with methods `addNode(nid)`, `addEdge(nid,nid')`, and `getNext(nid)/nids`. As with Sets and Hashtables, the number of nodes n may be large so we may want to only have some $N << n$ CA locations. We can thus define a conflict abstraction (X, f) where $X = \{v_0, \ldots, v_N\}$ and

$$\begin{aligned} f(\texttt{addNode}, [\texttt{nid}], \sigma) &= \{(\mathsf{true}, v_{\texttt{nid}\%N}, \mathsf{wr})\} \\ f(\texttt{getNext}, [\texttt{nid}], \sigma) &= \{(\mathsf{true}, v_{\texttt{nid}\%N}, \mathsf{rd})\} \\ f(\texttt{addEdge}, [\texttt{nid,nid'}], \sigma) &= \{(\mathsf{true}, v_{\texttt{nid}\%N}, \mathsf{wr})\} \end{aligned}$$

  The idea is to approximate conflict by focusing on the *nodes*. This node-based notion of conflict is one approach but one could imagine an alternative CA that uses edges as a basis for conflict. Each strategy is an approximation of conflict and the choice of strategy may depend on the specific semantics of the graph, methods, and/or workload. Indeed, one could even use edges *and* nodes as a basis for a CA, if a very fine-grained notion of conflict is needed. Notice that this CA places no restrictions on how the directed graph is actually implemented.

4 Shadow Speculation

Transactional Boosting [17] performed ADT operations *eagerly* and used inverse operations to apply an *operation undo log* to cleanup an aborted transaction. Unfortunately, this approach was coupled with pessimistic conflict resolution, where execution blocks when a conflict is detected. In an optimistic setting, transactions execute as if they will not encounter conflicts, and abort/retry if conflicts are detected. The key challenge is that a transaction must be able to observe the results of its own speculative updates to shared objects, without those updates becoming visible to other transactions until a successful commit occurs.

This is where *shadow speculation* helps. Shadow speculation is a technique for transactional objects, where updates are made on a separate local copy of a data-structure and then later merged with the master copy at commit time, similar to version control. This is conceptually similar to the thread-local copies used by lock-free and wait-free universal constructions [15,16]; however, conflict abstractions allow our approach to support finer grained concurrency, and we describe several techniques which allow our shadow copies to incur a lower memory overhead.

Replay logs. We begin by creating wrappers around a black-box ADT implementation so that we can replace the default behavior of a method invocation (*i.e.* immediately applying it to the object) with a more speculative strategy.

To support commit, we maintain an *operation replay log*, tracking the method names and arguments of all operations performed by a transaction rather than applying operations directly on the object (as seen in Boosting). Then, at commit, we can use a single data-structure lock to atomically replay the log of operations onto the shared object.

Unfortunately, this is insufficient for most applications as once an active transaction enqueues an operation to the log, it may need to know the return value in order to continue.

Speculative wrapper. Conceptually, the natural next step is for active transactions to be able to operate on their own local or *shadow* copy of the shared data-structure. This allows those transactions to perform operations that are invisible to concurrent transactions, and in particular, it also allows transactions to view the return values of these uncommitted operations. Our shadow copies are implemented inside the wrapper and designed so that when client code (speculatively) calls ADT operations, it predicts the result of each operation, intuitively reaching forward in time to see what return value would be generated if the transaction were to commit. The predicted values for a transaction T is calculated based on the committed state of the (black-box) object, combined with the uncommitted operations performed thus far by T. When a transaction aborts, a wrapper of this variety has no further work, because the underlying data structure has not been altered, and the shadow copy can be discarded. On the other hand, when a transaction commits, the wrapper must use the operation replay log to ensure that every speculative operation is finally applied to the underlying object.

While shadow copies are conceptually simple, they are not practical if implemented naïvely. Therefore, we must consider how to efficiently implement shadow copies for a variety of data structures.

Efficient shadow copies. Here we describe two approaches for efficient shadow copies.

1. *Memoization.* For some data-structures, the results of an operation (even an update) can be computed purely from the initial state of the wrapped data-structure, or from the arguments of other pending operations. In these cases, we may implement shadow copies by memoization. Repeated operations to the same key can be cached in a transaction-local table, and queried, to determine the results of the next operation on that key. If the key is not present, it's state can be determined by reading the unmodified backing data structure. Then, when the transaction commits, we can replay a single synthetic operation for each key in the table, to capture its final state.
   Memoization works particularly well for ADTs such as maps and sets: the result of `m.set(a,x)` followed by `m.get(a)` (a read-only operation) is `x`. We implemented this approach in our LazyHashMap, using Java's ConcurrentHashMap as the underlying data-structure.
2. *Snapshots.* For many data structures, memoization will be insufficient.
   A more general approach uses the fast-snapshot semantics provided by many concurrent data structures [26,27,3,28] to support shadow speculation. Such snapshots typically employ a lazy copy-on-write strategy to allow snapshots to initially share their internal structure with the original, and only copy as much data as is needed to perform each subsequent modification.
   Using snapshots for our shadow copies, the first time a transaction attempts to perform an update, a snapshot is made, and all further updates are performed on that snapshot. Whenever a transaction commits, any changes to the snapshot are replayed onto the shared copy.
   Snapshot implementations of shadow copies are also helpful in providing the "peek" methods such as `min` (priority queue) and `zero`/`willBeZero` (NZI) discussed above. For example, shadow copies let us determine ahead of time if the operation in question will change the result of `zero()` before and after the invocation.
   We implemented two data-structures this way: LazyTrieMap (based on Scala'a TrieMap) and LazyPriorityQueue (based on a concurrent Braun heap [7]).

5 The ScalaProust Transactional System

In this section we describe our implementation ScalaProust, an open source transactional system, available at the following URL:

`www.github.com/ScalaProust/ScalaProust/`

Our implementation includes support for both *pessimistic* operations on black-box highly concurrent ADTs (similar to boosting) as well as *optimistic* operations, as discussed in this paper. In this way the tool generalizes both boosting and predication. In this paper, however, we focus on how ScalaProust is used for optimistic operations, based on conflict abstractions (Sections 3) and shadow speculation (Section 4).

We first define a conflict abstraction (X, f) for the ADT, as discussed in Section 3. Next, we create a ScalaProust wrapper and decide whether the wrapper will manage shadow speculation via snapshots or via a memoization table, as discussed in Section 4. The wrapper can then be constructed, as defined to the right, to invoke each supported operation. proust_apply must execute in the context of a transaction T in order to register onCommit events.

```
1  let proust_apply($T, o.m, \bar{\alpha}$) =
2    let locs = $f(o.m, \bar{\alpha}, \sigma_o)$ in
3    foreach (fun ($p$, v, mode) →
4      if $p(\bar{\alpha}, \sigma_o)$ then match mode with
5      | rd → stm_read(v)
6      | wr → stm_write(v)
7    ) locs
8    let rv = Predict($o.m$,$\bar{\alpha}$) in
9    $T$.onCommit(fun () → { $o.m(\bar{\alpha})$;
10       foreach (fun (_,v,_) → stm_read(v) )
11         locs });
12   return rv
```

The wrapper proceeds as follows. First, the conflict abstraction f is consulted for the given method m and arguments $\bar{\alpha}$, returning the list of $(p, v, mode)$ tuples (Line 2). Next, the wrapper follows the instructions of each tuple of the conflict abstraction: if $p(\bar{\alpha}, \sigma_o)$ holds, then location v is either read or written (Line 3). The shadow speculation facility is used next (Line 8) to speculatively apply the method and obtain a return value rv. Finally, the wrapper registers an onCommit handler (Line 9) which will invoke the method on the shared object and once again read all of the conflict abstraction memory locations to guard against opacity violations.

ScalaProust includes a library API implementing conflict abstractions, as well as replay logs for both shadow speculation techniques. ScalaProust also provides a number of wrapped data structures out of the box, including both transactional maps and transactional priority queues, which can be used as-is, or serve as example code for developers to create their own wrappers.

6 Evaluation

Our goal in this section is to evaluate whether our optimistic treatment of black-box ADTs is efficient. Notably, our evaluation includes experiments to determine whether ScalaProust is competitive with the state-of-the-art specialized optimistic treatment of set/map-like structures found in predication [4]. Note that, for lack of space, we summarize the experimental results here. In the extended version of this paper [8], we have included additional experimental results and a discussion of how ScalaProust can be used pessimistically.

We focus our evaluation on time-efficiency rather than memory-efficiency for several reasons. First, it is difficult to reproducibly measure memory usage on the JVM due to its weak guarantees concerning garbage collection. Second, the memory usage is likely to be dependent on the specific implementation of the shadow copy, as well as the workload. However, we expect that for tree-like data-structures which exploit structural sharing for fast snapshots, the first modification will introduce $O(\log(n))$ memory overhead, and gradually saturate towards $O(n)$.

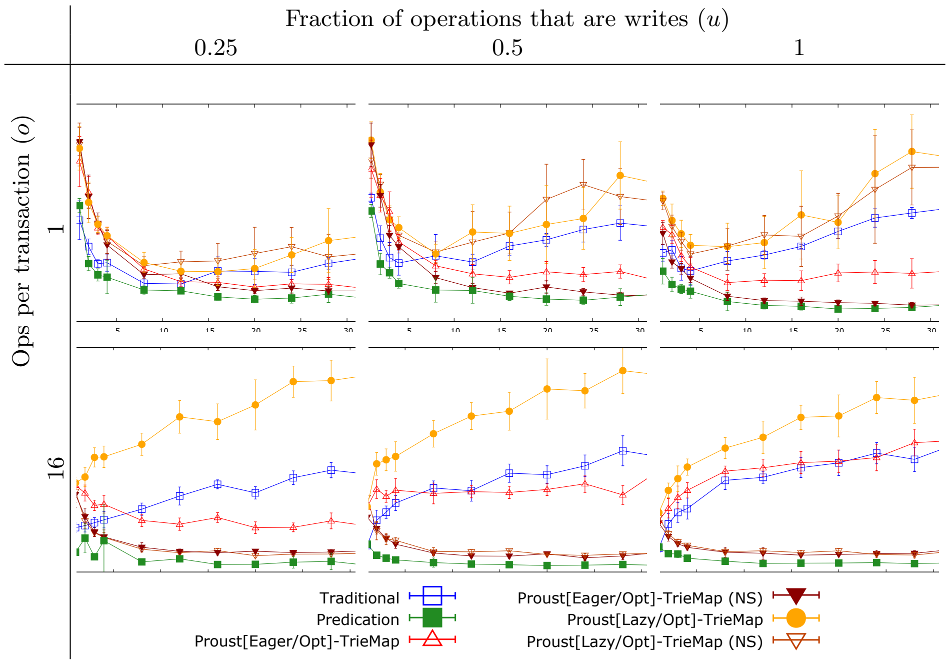

Fig. 1: Time to process 10^6 operations on concurrent maps (*smaller is better*), varying %-updates and #ops/txn. For each chart, the x-axis is the number of threads from 0 to 32 and the y-axis is the average time in milliseconds from 0 to 250. The (NS) variants disabled `size()`.

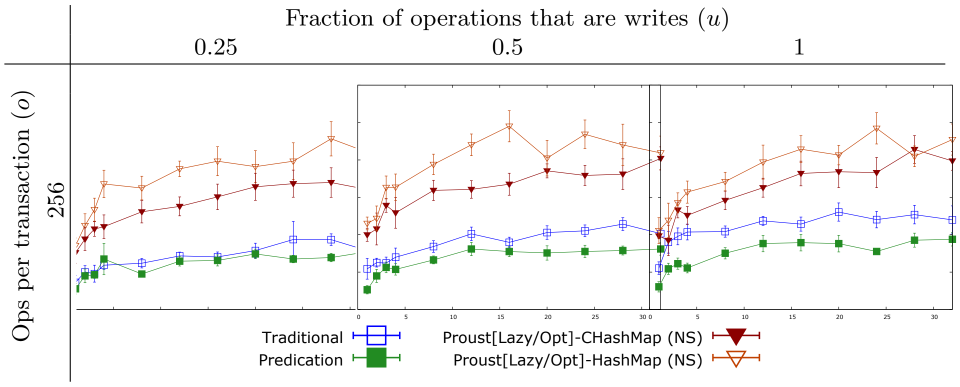

Fig. 2: Memoizing shadow copies allow updates of the same entry to be combined, providing a substantial decrease in execution time. *Smaller is better.*

We classify data structure wrappers based on conflict abstractions along two axes: their choice of synchronization strategy (optimistic or pessimistic), and their choice of update strategy (lazy or eager). Optimistic synchronization

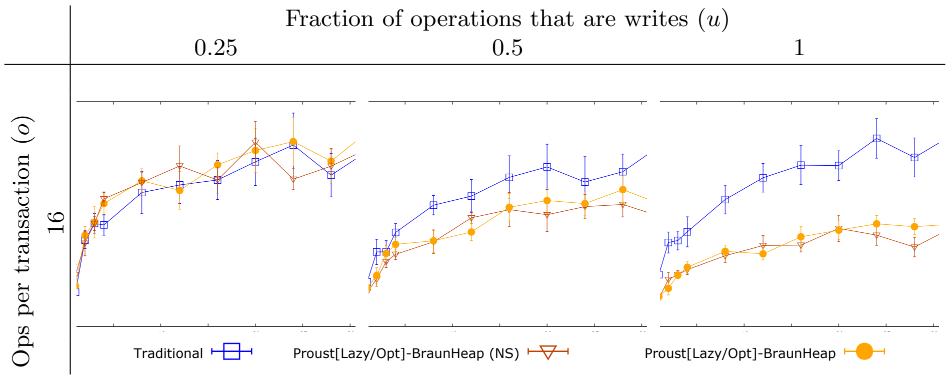

Fig. 3: Time to process 10^6 operations on concurrent priority queues (*smaller is better*), varying %-updates. For each chart, the x-axis is the number of threads from 0 to 32 and the y-axis is the average time in milliseconds from 0 to 2400. The (NS) variants disabled `size()`.

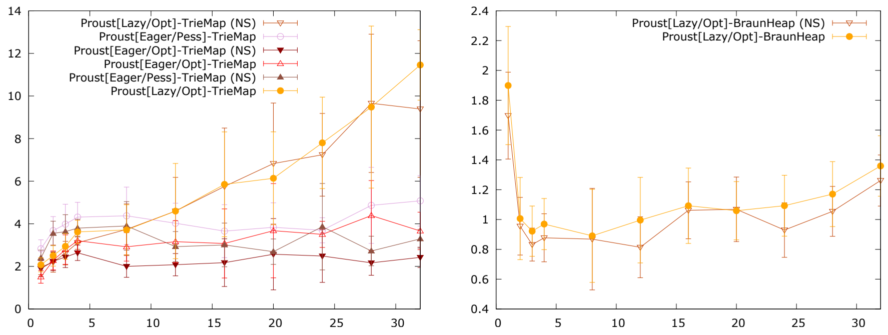

Fig. 4: Overhead of the transactional wrapper (relative to the base data structure) for different configurations of ScalaProust (vertical axis) vs thread count (horizontal axis), with $o = 1$. On the left are transactional maps and on the right are transactional priority queues. *Smaller is better.*

has been the primary focus of this work; however, pessimistic synchronization has been used in transactional boosting [17]. Similarly, lazy updates based on shadow copies and replay logs have been the primary focus this work; however, the conflict abstraction methodology can also be applied to eager updates based on inverses and undo logs à la boosting. We present results for three of the four possible quadrants: lazy/optimistic, eager/optimistic, and eager/pessimistic. Lazy/pessimistic wrappers are possible, but it seems unlikely that the extra memory overhead for lazy updates will pay off when pessimistic synchronization already ensures exclusive access to the relevant portions of the shared state.

Maps. We benchmarked several ScalaProust map wrappers (including variants with and without a `size()` operation) against both predication and a traditional pure-STM hash map, with a setup similar to that used by Bronson, et al. for predication [4]. We ran our experiments on an Amazon EC2 `m4.10xlarge` instance,[6] which has 40 vCPUs and 160 GB of RAM. For each experiment, we performed 10^6 random operations on a shared map, split across t threads, with o operations per transaction. A fraction u of the operations were writes (evenly split between `put` and `remove`), and the remaining $(1 - u)$ were `get` operations. We varied t, o, and u to achieve different levels of contention[7]. For each configuration, we warmed up the JVM for 10 executions, then timed each of the following 10 executions, garbage collecting in between to reduce jitter, and reported the mean and standard deviation.

Notes on experimental setup. First, our implementation was limited in its communication with the CCSTM contention manager. ScalaProust can communicate conflicts with CCSTM but currently does not provide the reason for the conflict. Consequently, CCSTM is limited in its ability to intelligently schedule retries. In particular, we found that under the artificially high contention seen in these experiments, longer transaction times could lead to livelock, as the STM lacked required information about the instigating (non-STM) memory accesses. For this reason, we only show the pessimistic results in the initial $o = 1$ experiments. Second, though the Eager/Optimistic configuration does not satisfy opacity under the CCSTM backend for ScalaSTM, we benchmarked it anyway, and did not observe any instances where this violated correctness (notably our benchmark makes no explicit control flow decisions based on the results of map accesses, and ScalaSTM performs an abort and retry if it ever observes an unchecked exception). It seems likely that a performance penalty was paid for late detection of inconsistent memory accesses, and we believe this speaks well to the potential performance of Eager/Optimistic wrappers on STMs where they satisfy opacity. Third, substantial performance differences between the standard and `(NS)` wrapper variants illustrate the previously discussed impedance mismatch between "pure" writes in a conflict abstraction and "impure" writes provided by STMs. We note that disabling the `size` operation for the `(NS)` variants did not require modifications to the underlying data structure, merely that we control which operations of the underlying data structure are exposed through the wrapper.

Results. The experimental results depicted in Figure 1 display the effects of several competing trends. Intuitively, ScalaProust's performance scales much better than the traditional STM implementation as contention increases, due to varying t and u (though we are consistently outperformed by the highly en-

[6] `https://aws.amazon.com/blogs/aws/the-new-m4-instance-type-bonus-price-reduction-on-m3-c4/`

[7] We did not vary key range as in the predication paper, as garbage collection was not a focus of this implementation

gineered predication implementation[8]); however, increasing values of o have a negative influence on the relative performance of the ScalaProust wrappers. Intuitively, this is to be expected, as our log sizes (either to undo or replay) are proportional to the number of updates performed, whereas predication and traditional implementations replay with time proportional to the number of unique memory locations updated, and as o increases, so does the probability that multiple writes will alter the same location. An optimization for memoization-, rather than snapshot-, based shadow speculation is to apply only the final state of each abstract state element; resulting in Figure 2. The overhead of the wrapper, relative to the base data structure can be seen in Figure 4.

Priority Queues. We used a nearly identical experimental setup to compare the runtimes of two priority queues based on Braun heaps (one traditional STM implementation and one wrapper around the snapshot-able concurrent implementation referenced earlier [7]). The writes were split evenly between `insert` and `removeMin` operations.

The experimental results in Figure 3 show that across a variety of conditions, the queue was competitive with, or outperformed, the traditional implementation. In general, run times were substantially longer than for the map throughput test, as the min-element is subject to heavy contention; however, unlike for map, the effects of additional operations per transaction were less pronounced, as most contention is discovered early in the transaction. The overhead of the wrapper, relative to the base data structure is shown in Figure 4.

7 Conclusions & Future Work

We introduced conflict abstractions and shadow speculation, permitting us to use existing highly-concurrent objects in an optimistic transactional manner, separately using off-the-shelf STMs for performing commutativity-based conflict detection. Benchmarks show we outperform, or are competitive with fine-tuned STM techniques (*i.e.* predication), while we are able to leverage existing ADT libraries and avoid implementing them from scratch. While we are outperformed by predication on the map throughput tests, we believe that our utility as a tool for wrapping arbitrary data structures will encourage use beyond sets and maps.

One important direction forward is to integrate pessimistic and optimistic treatment of black-box ADTs with standard STM memory operations. This brings with it some opacity challenges. To further improve performance, one could also explore an extension of our log-combining optimization from memoized replays to snapshot replays and undo logs. Alternatively, shadow copies based on confluently persistent data structures could even be merged without an explicit log [12]. In another direction, the use of conflict abstractions to describe commutativity and synchronization reveals a use-case for STMs to support "pure writes", allowing them to match the expressivity of handcrafted locks. Finally,

[8] Predication as a technique is specialized to maps and sets, in essence embedding their conflict abstraction as the member elements of the backing collection and allowing frequent updates to the same element to avoid updating the concrete state of the backing data structure.

automatic verification techniques (such as those mentioned in Section 3) might be used as a building-block for an automatic synthesis technique, perhaps along the lines of recent techniques for synthesizing commutativity conditions [1].

References

1. Bansal, K., Koskinen, E., and Tripp, O. Automatic generation of precise and useful commutativity conditions. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I* (2018), pp. 115–132.
2. Bernstein, A. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers 15*, 5 (1966), 757–763.
3. Bronson, N. G., Casper, J., Chafi, H., and Olukotun, K. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 257–268.
4. Bronson, N. G., Casper, J., Chafi, H., and Olukotun, K. Transactional predication: High-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (New York, NY, USA, 2010), PODC '10, ACM, pp. 6–15.
5. Bronson, N. G., Chafi, H., and Olukotun, K. Ccstm: A library-based stm for scala. *def 9* (2010), 10.
6. Carlstrom, B. D., McDonald, A., Carbin, M., Kozyrakis, C., and Olukotun, K. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2007), PPoPP '07, ACM, pp. 56–67.
7. Dickerson, T. D. Fast snapshottable concurrent braun heaps. *arXiv preprint arXiv:1705.06271* (2017).
8. Dickerson, T. D., Gazzillo, P., Koskinen, E., and Herlihy, M. Proust: A design space for highly-concurrent transactional data structures. *CoRR abs/1702.04866* (2017).
9. Dimitrov, D., Raychev, V., Vechev, M., and Koskinen, E. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, ACM, pp. 305–315.
10. Ellen, F., Lev, Y., Luchangco, V., and Moir, M. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (2007), PODC '07, ACM, pp. 13–22.
11. Felber, P., Gramoli, V., and Guerraoui, R. Elastic transactions. In *Proceedings of the 23rd International Conference on Distributed Computing* (Berlin, Heidelberg, 2009), DISC'09, Springer-Verlag, pp. 93–107.
12. Fiat, A., and Kaplan, H. Making data structures confluently persistent. *Journal of Algorithms 48*, 1 (2003). 12th ACM-SIAM Symposium on Discrete Algorithms.
13. Golan-Gueta, G., Ramalingam, G., Sagiv, M., and Yahav, E. Automatic semantic locking. *SIGPLAN Not. 49*, 8 (Feb. 2014), 385–386.
14. Hassan, A., Palmieri, R., and Ravindran, B. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), PPoPP '14, ACM, pp. 387–388.
15. Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (1991), 124–149.

16. Herlihy, M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 15*, 5 (1993), 745–770.
17. Herlihy, M., and Koskinen, E. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symp. Principles and prac. of parallel programming* (2008), PPoPP '08, ACM.
18. Herman, N., Inala, J. P., Huang, Y., Tsai, L., Kohler, E., Liskov, B., and Shrira, L. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 31:1–31:16.
19. Kim, D., and Rinard, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), pp. 528–541.
20. Korth, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (Jan. 1983).
21. Koskinen, E., Parkinson, M., and Herlihy, M. Coarse-grained transactions. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2010), POPL '10, ACM, pp. 19–30.
22. Koskinen, E., and Parkinson, M. J. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), Portland, OR, USA* (2015), ACM.
23. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)* (2007), pp. 211–222.
24. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. *Commun. ACM 52*, 9 (Sept. 2009), 89–97.
25. Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2007), PPoPP '07, ACM, pp. 68–78.
26. Petrank, E., and Timnat, S. Lock-free data-structure iterators. In *International Symposium on Distributed Computing* (2013), Springer, pp. 224–238.
27. Prokopec, A. Snapqueue: Lock-free queue with constant time snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala* (New York, NY, USA, 2015), SCALA 2015, ACM, pp. 1–12.
28. Prokopec, A., Bronson, N. G., Bagwell, P., and Odersky, M. Concurrent tries with efficient non-blocking snapshots. 151–160.
29. Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.
30. Spiegelman, A., Golan-Gueta, G., and Keidar, I. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), ACM, pp. 682–696.
31. Steele, Jr, G. L. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'90)* (New York, NY, USA, 1990), ACM Press, pp. 218–231.
32. Weihl, W. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers 37*, 12 (1988), 1488–1505.