Transactional Predication: High-Performance Concurrent Sets and Maps for STM

Nathan G. Bronson
Computer Systems Lab,
Stanford University
353 Serra Mall
Stanford, California 94305
nbronson@stanford.edu

Jared Casper
Computer Systems Lab,
Stanford University
353 Serra Mall
Stanford, California 94305
jaredc@stanford.edu

Hassan Chafi
Computer Systems Lab,
Stanford University
353 Serra Mall
Stanford, California 94305
hchafi@stanford.edu

Kunle Olukotun
Computer Systems Lab,
Stanford University
353 Serra Mall
Stanford, California 94305
kunle@stanford.edu

ABSTRACT

Concurrent collection classes are widely used in multi-threaded programming, but they provide atomicity only for a fixed set of operations. Software transactional memory (STM) provides a convenient and powerful programming model for composing atomic operations, but concurrent collection algorithms that allow their operations to be composed using STM are significantly slower than their non-composable alternatives.

We introduce *transactional predication*, a method for building transactional maps and sets on top of an underlying non-composable concurrent map. We factor the work of most collection operations into two parts: a portion that does not need atomicity or isolation, and a single transactional memory access. The result approximates semantic conflict detection using the STM's structural conflict detection mechanism. The separation also allows extra optimizations when the collection is used outside a transaction. We perform an experimental evaluation that shows that predication has better performance than existing transactional collection algorithms across a range of workloads.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – Concurrent programming structures; D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel programming; E.1 [**Data Structures**]: Distributed data structures, trees

General Terms: Algorithms, Performance

Keywords: Transactional predication, software transactional memory, semantic conflict detection, concurrent map

1. INTRODUCTION

Concurrent sets and maps classes have emerged as one of the core abstractions of multi-threaded programming. They provide the programmer with the simple mental model that most method calls are linearizable, while admitting efficient and scalable implementations. Concurrent hash tables are part of the standard library of Java and C#, and are part of Intel's Thread Building Blocks for C++. Concurrent skip lists are also widely available. The efficiency and scalability of these data structures, however, comes from the use of non-composable concurrency control schemes. None of the standard concurrent hash table or skip list implementations provide composable atomicity.

Software transactional memory (STM) provides a natural model for expressing and implementing compound queries and updates of concurrent data structures. It can atomically compose multiple operations on a single collection, operations on multiple collections, and reads and writes to other shared data. Unlike lock-based synchronization, composition does not lead to deadlock or priority inversion.

If all of the loads and stores performed by a hash table, tree, or skip list are managed by an STM, then the resulting data structure automatically has linearizable methods that may be arbitrarily composed into larger transactions. The STM implementation may also provide useful properties such as: optimistic conflict detection with invisible readers (provides the best scalability for concurrent readers on cache-coherent shared memory architectures) [23]; lock- or obstruction-freedom (limits the extent to which one thread can interfere with another) [24]; intelligent contention management (prevents starvation of individual transactions) [9]; and modular blocking using using `retry` and `orElse` (allows composition of code that performs conditional waiting) [11].

We apply the adjective 'transactional' to a data structure if its operations may participate in a transaction, regardless of the underlying implementation. The most straightforward way of implementing such an algorithm is to execute

all shared memory accesses through the STM; the result will automatically be transactional, but it will suffer from high single-thread overheads and false conflicts. For many applications, trading some speed for improved programmability can be a good decision. Maps and sets are such fundamental data structures, however, that the additional internal complexity and engineering effort of bypassing the STM is justified if it leads to improvements in performance and scalability for all users.

This paper introduces *transactional predication*, the first implementation technique for transactional maps and sets that preserves the STM's optimistic concurrency, contention management, and modular blocking features, while reducing the overheads and false conflicts that arise when the STM must mediate access to the internal structure of the collection. We factor each transactional operation into a referentially transparent lookup and a single STM-managed read or write. This separation allows the bulk of the work to bypass the STM, yet leaves the STM responsible for atomicity and isolation. Our specific contributions:

- We introduce transactional predication, the first method for performing semantic conflict detection for transactional maps and sets using an STM's structural conflict detection mechanism. This method leverages the existing research on STM implementation techniques and features, while avoiding structural conflicts and reducing the constant overheads that have plagued STM data structures (Section 3).

- We use transactional predication to implement transactional sets and maps on top of linearizable concurrent maps (Section 3). We add support for iteration in unordered maps (Section 5.1), and describe how to perform iteration and range-based search in ordered maps (Section 5.2).

- We describe two schemes for garbage collecting predicates from the underlying map: one based on reference counting (Section 4.1), and one using soft references (Section 4.2).

- We experimentally evaluate the performance and scalability of maps implemented with transactional predication, comparing them to best-of-breed non-transactional concurrent maps, data structures implemented directly in an STM, and concurrent maps that have been transactionally boosted. We find that predicated maps outperform existing transactional maps, often significantly (Section 6).

2. BACKGROUND

Sets and associative maps are fundamental data structures; they are even afforded their own syntax and semantics in many programming languages. Intuitively, concurrent sets and maps should allow accesses to disjoint elements to proceed in parallel. There is a surprising diversity in the techniques developed to deliver this parallelism. They can be roughly grouped into those that use fine-grained locking and those that use concurrency control schemes tailored to the specific data structure and its operations. Transactional predication is independent of the details of the underlying map implementation, so we omit a complete survey. We refer the reader to [16] for step-by-step derivation of several concurrent hash table and skip list algorithms.

Concurrent collection classes are widely used, but they do not provide a means to compose their atomic operations. This poses a difficulty for applications that need to simultaneously update multiple elements of a map, or coordinate updates to two maps. Consider an application that needs to concurrently maintain both a forward and reverse association between keys and values, such as a map from names to phone numbers and from phone numbers to names. If the forward and reverse maps are implemented using hash tables with fine-grained locks, then changing a phone number while maintaining data structure consistency requires acquiring one lock in the forward map (to change the value that records the phone number), and two locks in the reverse map (to remove the name from one number and add it to another). This would require breaking the clean interface to the concurrent map by exposing its internal locks, because it is not sufficient to perform each of the three updates separately. This example also leads to deadlock if the locks are not acquired following a global lock order, which will further complicate the user's code. Lock-free hash tables don't even have the option of exposing their locks to the caller. Transactional memory, however, provides a clean model for composing the three updates required to change a phone number.

While there has been much progress in efficient execution of STM's high-level programming model, simply wrapping existing map implementations in atomic blocks will not match the performance achievable by algorithm-specific concurrency control. Data structures implemented on top of an STM face two problems:

- **False conflicts** – STMs perform conflict detection on the concrete representation of a data structure, not on its abstract state. This means that operations that happen to touch the same memory location may trigger conflict and rollback, despite the operations being semantically independent.

- **Sequential overheads** – STMs instrument all accesses to shared mutable state, which imposes a performance penalty even when only one thread is used. This penalty is a 'hole' that scalability must climb out of before a parallel speedup is observed. Sequential overheads for STM are higher than those of traditional shared-memory programming [5] and hand-rolled optimistic concurrency [2].

False conflicts between operations on a transactional data structure can be reduced or eliminated by performing *semantic conflict detection* at the level of operations. Rather than computing conflicts based on the reads from and writes to individual memory locations, higher-level knowledge is used to determine whether operations conflict. For example, adding k_1 to a set does not semantically conflict with adding k_2 if $k_1 \neq k_2$, regardless of whether those operations write to the same chain of hash buckets or rotate the same tree nodes. Because semantically independent transactions may have structural conflicts, some other concurrency control mechanism must be used to protect accesses to the underlying data structure. This means that a system that provides semantic conflict detection must break transaction isolation to communicate between active transactions. Isolation can be relaxed for accesses to the underlying structure by performing them in open nested transactions [4, 22], or by performing them outside transactions, using a lineariz-

able algorithm that provides its own concurrency control. The latter approach is used by transactional boosting [13].

Although semantic conflict detection using open nested transactions reduces the number of false conflicts, it exacerbates sequential overheads. Accesses still go through the STM, but additional information about the semantic operations must be recorded and shared. Semantic conflict detection using transactional boosting reduces sequential overheads by allowing loads and stores to the underlying data structure to bypass the STM entirely, but it accomplishes this by adding a layer of pessimistic two-phase locking. These locks interfere with optimistic STMs, voiding useful properties such as opacity [10], obstruction- or lock-freedom, and modular blocking [11]. In addition, boosting must be tightly integrated to the STM's contention manager to prevent starvation and livelock.

The goal of our research into transactional collections is to produce data structures whose non-transactional performance and scalability is equal to the best-of-breed concurrent collections, but that provide all of the composability and declarative concurrency benefits of STM. Transactional predication is a step in that direction.

3. TRANSACTIONAL PREDICATION

Consider a minimal transactional set, that provides only the functions `contains(`e`)` and `add(`e`)`. Semantically, these operations conflict only when they are applied to equal elements, and at least one operation is an `add` [1]:

conflict?	`contains(`e_1`)`	`add(`e_1`)`
`contains(`e_2`)`	no	$e_1 = e_2$
`add(`e_2`)`	$e_1 = e_2$	$e_1 = e_2$

This conflict relation has the same structure as the basic reads and writes in an STM: two accesses conflict if they reference the same location and at least one of them is a write:

conflict?	`stmRead(`p_1`)`	`stmWrite(`p_1, v_1`)`
`stmRead(`p_2`)`	no	$p_1 = p_2$
`stmWrite(`p_2, v_2`)`	$p_1 = p_2$	$p_1 = p_2$

The correspondence between the conflict relations means that we can perform semantic conflict detection in our transactional set by mapping each element e to a location p, performing a read from p during `contains(`e`)`, and performing a write to p during `add(`e`)`.

Of course, conflict detection is not enough; operations must also query and update the abstract state of the set, and these accesses must be done in a transactional manner. Perhaps surprisingly, the reads and writes of p can also be used to manage the abstract state. Transactional predication is based on the observation that membership in a finite set S can be expressed as a predicate $f : U \to \{0, 1\}$ over a universe $U \supseteq S$ of possible elements, where $e \in S \iff f(e)$, and that f can be represented in memory by storing $f(e)$ in the location p associated with each e. We refer to the p associated with e as that element's *predicate*. To determine if an e is in the abstract state of the set, as viewed from the current transactional context, we perform an STM-managed read of p to see if $f(e)$ is true. To add e to the set, we perform an STM-managed write of p to change the encoding for $f(e)$. The set operations are trivial as the complexity has been moved to the $e \to p$ mapping.

[1]We assume a non-idempotent `add` that reports set changes.

```
1  class TSet[A] {
2    def contains(elem: A): Boolean =
3        predForElem(elem).stmRead()
4    def add(elem: A): Boolean =
5        predForElem(elem).stmReadAndWrite(true)
6    def remove(elem: A): Boolean =
7        predForElem(elem).stmReadAndWrite(false)
8
9    private val predicates =
10       new ConcurrentHashMap[A,TVar[Boolean]]
11   private def predForElem(elem: A) = {
12     var pred = predicates.get(elem)
13     if (pred == null) {
14       val fresh = new TVar(false)
15       pred = predicates.putIfAbsent(elem, fresh)
16       if (pred == null) pred = fresh
17     }
18     return pred
19   }
20 }
```

Figure 1: A minimal but complete transactionally predicated set in Scala. Read and write barriers are explicit. *TVar* is provided natively by the STM.

The final piece of *TSet* is the mapping from element to predicate, which we record using a hash table. Precomputing the entire relation is not feasible, so we populate it lazily. The mapping for any particular e never changes, so `predForElem(elem)` is referentially transparent; its implementation can bypass the STM entirely.

Although the mapping for each element is fixed, reads and lazy initializations of the underlying hash table must be thread-safe. Any concurrent hash table implementation may be used, as long as it provides a way for threads to reach a consensus on the lazily installed key-value associations. Figure 1 shows the complete Scala code for a minimal transactionally predicated set, including an implementation of `predForElem` that uses `putIfAbsent` to perform the lazy initialization. `putIfAbsent(`e, p`)` associates p with e only if no previous association for e was present. It returns `null` on success, or the existing p_0 on failure. In Figure 1, Line 15 proposes a newly allocated predicate to be associated with `elem`, and Line 16 uses the value returned from `putIfAbsent` to compute the consensus decision.

3.1 Atomicity and Isolation

Transactional predication factors the work of *TSet* operations into two parts: lookup of the appropriate predicate, and an STM-managed access to that predicate. Because the lookup is referentially transparent, atomicity and isolation are not needed. The lookup can bypass the STM completely. The read or write to the predicate requires STM-provided atomicity and isolation, but only a single access is performed and no false conflicts can result.

Bypassing the STM for the predicate map is similar to Moss' use of open nesting for *String*`.intern(`s`)`, which internally uses a concurrent set to merge duplicate strings [21]. Like strings interned by a failed transaction, lazily installed predicates do not need to be removed during rollback.

Figure 2 shows a simultaneous execution of `add(10)` and `contains(10)` using the code from Figure 1. Time proceeds from the top of the figure to the bottom. Because no predicate was previously present for this key, thread 1 performs

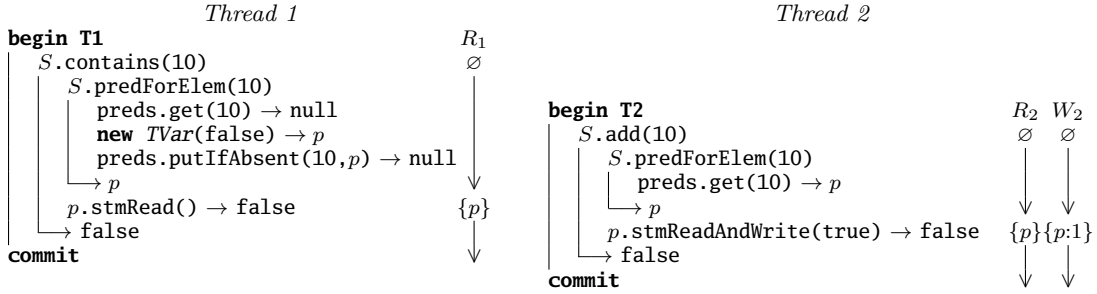


Figure 2: A simultaneous execution of `contains(10)` and `add(10)` using the code from Figure 1. R_i and W_i are the read and write sets. Thread 1 lazily initializes the predicate for element 10.

the lazy initialization of the $10 \rightarrow p$ mapping. An association is present by the time that thread 2 queries the mapping, so it doesn't need to call `putIfAbsent`. At commit time, T_1's read set contains only the element p. This means that there is no conflict with a transaction that accesses any other key of S, and optimistic concurrency control can be used to improve the scalability of parallel reads.

The abstract state of the set is completely encoded in STM-managed memory locations, so the STM provides atomicity and isolation for the data structure. Unlike previous approaches to semantic conflict detection, no write buffer or undo log separate from the STM's are required, and no additional data structures are required to detect conflicts. This has efficiency benefits, because the STM's version management and conflict detection are highly optimized. It also has semantic benefits, because opacity, closed nesting, modular blocking, and sophisticated conflict management schemes continue to work unaffected.

There are two subtleties that deserve emphasis: 1) A predicate must be inserted into the underlying map even if the key is absent. This guarantees that a semantic conflict will be generated if another transaction adds a key and commits. 2) When inserting a new predicate during `add`, the initial state of the predicate must be `false` and a transactional write must be used to set it to `true`. This guarantees that contexts that observe the predicate before the adding transaction's commit will not see the speculative add.

3.2 Direct STM vs. Transactional Predication

Figure 3 shows how two transactional set implementations might execute `contains(10)`. In part (a) the set is presented by a hash table with chaining. To locate the element, a transactional read must be performed to locate the current hash array, then a transactional read of the array is used to begin a search through the bucket chain. Each access through the STM incurs a performance penalty, because it must be recorded in the read set and validated during commit. In addition, reads that occur to portions of the data structure that are not specific to a particular key may lead to false conflicts. In this example, `remove(27)` will conflict with `contains(10)`, even though at a semantic level those operations are independent.

Figure 3b shows a predicated set executing `contains(10)`. A concurrent hash map lookup is performed outside the STM to locate the predicate. A single transactional read of the predicate is then used to answer the query. The abstract state is encoded entirely in these STM-managed memory locations; the mapping from key to predicate has no

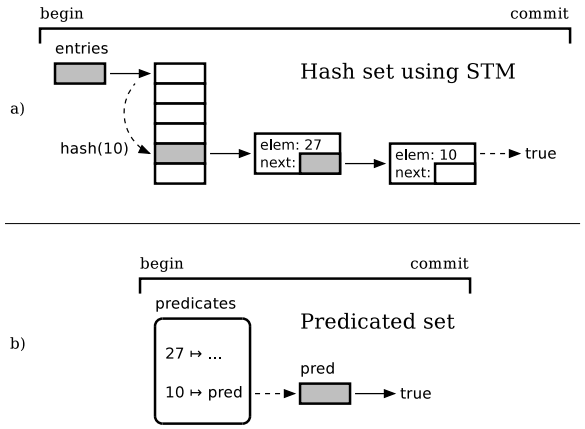


Figure 3: Execution of `contains(10)` in: a) a hash table performing all accesses to shared mutable state via STM; and b) a transactionally predicated set. ▭ are STM-managed reads.

side effects and requires no atomicity or isolation. Thus no scheduling constraints are placed on the STM, and no separate undo log, write buffer or conflict information is needed.

3.3 Extending Predication to Maps

The predicate stores the abstract state for its associated element. The per-element state for a set consists only of presence or absence, but for a map we must also store a value. We encode this using Scala's *`Option`* algebraic data type, which is `Some(`v`)` for the presence of value v, or `None` for absence. *`TMap`*`[`*`K`*`,`*`V`*`]` predicates have type *`TVar`*`[`*`Option`*`[`*`V`*`]]`:

```
class TMap[K,V] {
  def get(key: K): Option[V] =
      predForKey(key).stmRead()
  def put(key: K, value: V): Option[V] =
      predForKey(key).stmReadAndWrite(Some(value))
  def remove(key: K): Option[V] =
      predForKey(key).stmReadAndWrite(None)
  private def predForKey(k: K): TVar[Option[V]] = ...
}
```

3.4 Sharing of Uncommitted Data

Like other forms of semantic conflict detection, transactional predication must make the keys of the predicate map public before the calling atomic block has committed. Carlstrom et al. [4] propose addressing this problem by using Java's *`Serializable`* interface to reduce keys to byte arrays

before passing them across an isolation boundary in their hardware transactional memory (HTM). The version management and conflict detection in most STMs does not span multiple objects; for these systems Moss [21] shows that immutable keys can be shared across isolation boundaries. In our presentation and in our experimental evaluation we assume that keys are immutable.

4. GARBAGE COLLECTION

The minimal implementation presented in the previous section never garbage collects its *TVar* predicates. The underlying concurrent map will contain entries for keys that were removed or that were queried but found absent. While information about absence must be kept for the duration of the accessing transaction to guarantee serializability, for a general purpose data structure it should not be retained indefinitely. Some sort of garbage collection is needed.

Predicates serve two purposes: they encode the abstract state of the set or map, and they guarantee that semantically conflicting operations will have a structural conflict. The abstract state will be unaffected if we remove a predicate that records absence, so to determine if such a predicate can be reclaimed we only need to reason about conflict detection. Semantic conflict detection requires that any two active transactions that perform a conflicting operation on the predicated collection must agree on the predicate, because the predicate's structural conflict stands in for the semantic conflict. If transaction T_1 calls `get(`k`)` and a simultaneous T_2 calls `put(`k`,`v`)`, then they must agree on k's predicate so that T_1 will be rolled back if T_2 commits.

For STMs that linearize during commit, it is sufficient that transactions agree on the predicate for k during the interval between the transactional map operation that uses k and their commit. To see that this is sufficient, let (a_i, e_i) be the interval that includes T_i's access to k and T_i's commit. If the intervals overlap, then T_1 and T_2 agree on k's predicate. If the intervals don't overlap, then assume WLOG that $e_1 < a_2$ and that there is no intervening transaction. The predicate could not have been garbage collected unless T_1's committed state implies k is absent, so at a_2 a new empty predicate will be created. T_2's commit occurs after a_2, so T_2 linearizes after T_1. T_1's final state for k is equal to the initial abstract state for T_2, so the execution is serializable.

Algorithms that guarantee opacity can optimize read-only transactions by linearizing them before their commit, because consistency was guaranteed at the last transactional read (and all earlier ones). The TL2 [6] algorithm, for example, performs this optimization. We can provide correct execution for TL2 despite using the weaker agreement property by arranging for newly created predicates to have a larger timestamp than any reclaimed predicate. This guarantees that if a predicate modified by T_1 has been reclaimed, any successful transaction that installs a new predicate must linearize after T_1. We expect that this technique will generalize to other timestamp-based STMs. We leave for future work a formal treatment of object creation by escape actions in a timestamp-based STM. The code used in the experimental evaluation (Section 6) includes the TL2-specific mechanism for handling this issue.

Predicate reclamation can be easily extended to include **`retry`** and **`orElse`**, Harris et al.'s modular blocking operators [11]. All predicates accessed by a transaction awaiting **`retry`** are considered live.

4.1 Reference Counting

One option for reclaiming predicates once they are no longer in use is reference counting. There is only a single level of indirection, so there are no cycles that would require a backup collector. Reference counts are incremented on access to a predicate, and decremented when the enclosing transaction commits or rolls back. Whenever the reference count drops to zero and the committed state of a predicate records absence, it may be reclaimed.

Reference counting is slightly complicated by an inability to decrement the reference count and check the value of the predicate in a single step. We solve this problem by giving present predicates a reference count bonus, so that a zero reference count guarantees that the predicate's committed state records absence. Transactions that perform a `put` that results in an insertion add the bonus during commit (actually, they just skip the normal end-of-transaction decrement), and transactions that perform a `remove` of a present key subtract the bonus during commit. To prevent a race between a subsequent increment of the reference count and the predicate's removal from the underlying map, we never reuse a predicate after its reference count has become 0. This mechanism is appealing because it keeps the predicate map as small as possible. It requires writes to a shared memory location, however, which can limit scalability if many transactions read the same key. Reference counting can is suitable for use in an unmanaged environment if coupled with a memory reclamation strategy such as hazard pointers [20].

4.2 Soft References

When running in a managed environment, we can take advantage of weak references to reclaim unused predicates. Weak references can be traversed to retrieve their referent if it is still available, but do not prevent the language's GC from reclaiming the referenced object. Some platforms have multiple types of weak references, giving the programmer an opportunity to provide a hint to the GC about expected reuse. On the JVM there is a distinction between *WeakReference*s, which are garbage collected at the first opportunity, and *SoftReference*s, which survive collection if there is no memory pressure. Reclamations require mutation of the underlying predicate map, so to maximize scalability we use soft references.

Soft references to the predicates themselves are not correct, because the underlying map may hold the only reference (via the predicate) to a valid key-value association. Instead, we use a soft reference from the predicate to a discardable token object. Collection of the token triggers cleanup, so we include a strong reference to the token inside the predicate's *TVar* if the predicate indicates presence. For sets, we replace the *TVar*`[`*Boolean*`]` with *TVar*`[`*Token*`]`, representing a present entry as a *Token* and an absent entry as `null`. For maps, we replace the *TVar*`[`*Option*`[`*V*`]]` with a *TVar*`[(`*Token*`,`*V*`)]`, encoding a present key-value association as `(`*Token*`,`v`)` and an absent association as `(null,*)`.

If an element or association is present in any transaction context then a strong reference to the token exists. If a transactional read indicates absence, then a strong reference to the token is added to the transaction object itself, to guarantee that the token will survive at least until the end of the transaction. A predicate whose token has been garbage collected is stale and no longer usable, the same as a predicate with a zero reference count. If a predicate is

not stale, contexts may disagree about whether the entry is present or absent, but they will all agree on the transition into the stale state.

4.3 Optimizing Non-Transactional Access

Ideally, transactional sets and maps would be as efficient as best-of-breed linearizable collections when used outside a transaction. If code that doesn't need STM integration doesn't pay a penalty for the existence of those features, then each portion of the program can locally make the best feature/performance tradeoff. Transactionally predicated maps do not completely match the performance of non-composable concurrent maps, but we can keep the gap small by carefully optimizing non-transactional operations.

Avoiding the overhead of a transaction: The transactionally predicated sets and maps presented so far perform exactly one access to an STM-managed memory location per operation. If an operation is called outside an atomic block, we can use an isolation barrier, an optimized code sequence that has the effect of performing a single-access transaction [17]. Our scheme for unordered enumeration (Section 5.1) requires two accesses for operations that change the size of the collection, but both locations are known ahead of time. Saha et al. [23] show that STMs can support a multi-word compare-and-swap with lower overheads than the equivalent dynamic transaction.

Reading without creating a predicate: While non-transactional accesses to the predicated set or map must be linearizable, the implementation is free to choose its own linearization point independent of the STM. This means that `get(`k`)` and `remove(`k`)` do not need to create a predicate for k if one does not already exist. A predicate is present whenever a key is in the committed state of the map, so if no predicate is found then `get` and `remove` can linearize at the read of the underlying map, reporting absence to the caller.

Reading from a stale predicate: `get` and `remove` can skip removal and replacement if they discover a stale predicate, by linearizing at the later of the lookup time and the time at which the predicate became stale.

Inserting a pre-populated predicate: We can linearize a `put(`k, v`)` that must insert a new predicate at the moment of insertion. Therefore we can place v in the predicate during creation, rather than via an isolation barrier.

5. ITERATION AND RANGE SEARCHES

So far we have considered only transactional operations on entries identified by a user-specified key. Maps also support useful operations over multiple elements, such as iteration, or that locate their entry without an exact key match, such as finding the smallest entry larger than a particular key in an ordered map. Transactionally predicated maps can implement these operations using the iteration or search functionality of the underlying predicate map.

5.1 Transactional Iteration

For a transactionally predicated map M, every key present in the committed state or a speculative state is part of the underlying predicate map P. If a transaction T visits all of the keys of P, it will visit all of the keys of its transactional perspective of M, except keys added to M by an operation that starts after the iteration. If T can guarantee that no `put`s that commit before T were executed after the iteration started, it can be certain that it has visited every key that might be in M. The exact set of keys in M (and their values) can be determined by `get(`k`)` for keys k in P.

In Section 3 we perform semantic conflict detection for per-element operations by arranging for those operations to make conflicting accesses to STM-managed memory. We use the same strategy for detecting conflicts between insertions and iterations, by adding an insertion counter. Iterations of M read this STM-managed `TVar[Int]`, and insertions that create a new predicate increment it. Iterations that miss a key will therefore be invalidated.

Unfortunately, a shared insertion counter introduces a false conflict between `put(`k_1, v_1`)` and `put(`k_2, v_2`)`. Rollbacks from this conflict could be avoided by Harris et al.'s abstract nested transactions [12], but we use a simpler scheme that stripes the counter across multiple transactionally-managed memory locations. Insertions increment only the value of their particular stripe, and iterations perform a read of all stripes. By fixing a pseudo-random binding from thread to stripe, the probability of a false conflict is kept independent of transaction size.

There is some flexibility as to when changes to the insertion count are committed. Let t_{P+} be the time at which the key was inserted into the predicate map P and t_{M+} be the linearization time of k's insertion into M. No conflict is required for iterations that linearize before t_{M+}, because k is not part of M in their context. No conflict is required for iterations that start after t_{P+}, because they will include k. This means that any iteration that conflicts with the insertion must have read the insertion counter before t_{P+} and linearized after t_{M+}. The increment can be performed either in a transaction or via an isolation barrier, so long as it linearizes in the interval $(t_{P+}, t_{M+}]$. Incrementing the insertion counter at t_{M+}, as part of the transaction that adds k to M, allows a transactionally-consistent insertion count to be computed by summing the stripes. If a removal counter is also maintained, then we can provide a transactional `size()` as the difference.

Note that optimistic iteration is likely to produce the starving elder pathology for large maps with concurrent mutating transactions [1]. We assume that the STM's contention manager guarantees eventual completion for the iterating transaction.

5.2 Iteration and Search in an Ordered Map

In an ordered map, it is more likely that an iterator will be used to access only a fraction of the elements, for example to retrieve the m smallest keys. For this use case, the insertion counter strategy is too conservative, detecting conflict even when an insertion is performed that does not conflict with the partially-consumed iterator. Ordered maps and sets also often provide operations that return the smallest or largest entry whose key falls in a range. Tracking insertions only at the collection level will lead to many false conflicts.

We solve this problem by storing an insertion count in each entry of P, as well as one additional per-collection count. An entry's counter is incremented when a predicate is added to P that becomes that entry's successor, and the per-collection counter is incremented when a new minimal entry is inserted. (Alternately a sentinal entry with a key of $-\infty$ could be used.) Because these counters only protect forward traversals, a search for the smallest key $> k$ first finds the largest key $\leq k$ and then performs a protected

traversal. The successor-insertion counters for the ordered map are not useful for computing `size()`, so we increment them using a non-transactional isolation barrier.

Despite the navigation of the underlying map required when inserting a new predicate, our scheme results in no false conflicts for `get`, `put` and `remove`, since these operations neither read nor write any of the insertion counters. Transactional iteration and range queries may experience false conflicts if a concurrent operation inserts a predicate into an interval already traversed by the transaction, but unlike false conflicts in an STM-based tree or skip list at most one interval is affected by each insertion.

6. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of an unordered and ordered map implemented using transactional predication. We first evaluate the predicate reclamation schemes from Section 4, concluding that soft references are the best all-around choice. We then show that predicated hash maps have better performance and scalability than either an STM-based hash table or a boosted concurrent hash table. Finally, we evaluate a predicated concurrent skip list.

Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors, and 24GB of RAM. We used the Linux kernel version `2.6.28-16-server`. Hyper-Threading was enabled, yielding a total of 16 hardware thread contexts. Code was compiled with Scala version 2.7.7. We ran our experiments in Sun's Java SE Runtime Environment, build 1.6.0_16-b01, using the HotSpot 64-Bit Server VM with compressed object pointers. We use CCSTM, a reference-based STM for Scala [3]. CCSTM uses the SwissTM algorithm [7], which is a variant of TL2 [6] that detects write-write conflicts eagerly.

Our experiments emulate the methodology used by Herlihy et al. [14]. Each pass consists of each thread performing 10^6 randomly chosen operations on a shared map; a new map is used for each pass. To simulate a variety of workloads, two parameters are varied: the proportion of `get`, `put` and `remove` operations, and the range from which the keys are uniformly selected. A smaller fraction of `get`s and a smaller key range both increase contention. Because `put` and `remove` are equally likely in our tests, the map size converges to half the key range. To allow for HotSpot's dynamic compilation, each experiment consists of twenty passes; the first ten warm up the VM and the second ten are timed. Each experiment was run five times and the arithmetic average is reported as the final result.

6.1 Garbage Collection Strategy

To evaluate predicate reclamation strategies, Figure 4 shows experiments using the following map implementations:

- **conc-hash** – Lea's *`ConcurrentHashMap`* [19], as included in the JRE's standard library;
- **txn-pred-none** – a transactionally predicated *`ConcurrentHashMap`*, with no reclamation of stale predicates;
- **txn-pred-rc** – a predicated hash map that uses the reference counting scheme of Section 4.1; and
- **txn-pred-soft** – a predicated hash map that uses the the soft reference mechanism of Section 4.2.

Txn-pred-rc performs the most foreground work, but its aggressive reclamation yields the smallest memory footprint.

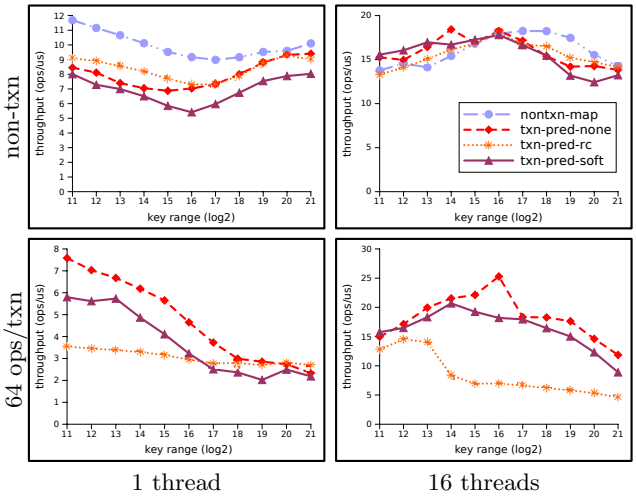


Figure 4: Throughput for three predicate reclamation strategies (none, reference counting and soft references), with 80% reads. Lea's non-composable *`ConcurrentHashMap`* is included for reference.

Txn-pred-soft delays predicate cleanup and has larger predicate objects than txn-pred-rc, reducing locality. Because the performance effect of locality depends on the working set size, we show a sweep of the key range for a fixed instruction mix (80% get, 10% put and 10% remove), at minimum and maximum thread counts. The optimizations from Section 4.3 also have a large impact, so we show both non-transactional access and access in transactions that perform 64 operations (txn2's curves are similar to txn64's).

Except for conc-hash, the non-txn experiments represent the performance of a map that *supports* transactional access, but is currently being accessed outside an atomic block. Conc-hash is faster than any of the transactional maps, at least for 1 thread. For some multi-thread non-txn experiments, however, txn-pred-none and txn-pred-soft perform better than conc-hash, despite using a *`ConcurrentHashMap`* in their implementation. This is because they allow a predicate to remain after its key is removed from the abstract state of the map, replacing a use of conc-hash's contended segment lock with an uncontended write to a `TVar`. If the key is re-added, the savings are doubled. This effect appears even more prominently in Figures 5 and **??**, discussed below.

For most transactional configurations (that cannot use the non-transactional optimizations) txn-pred-soft is both faster and more scalable than txn-pred-rc. The exception is uncontended (1 thread) access to a large map, where reference counting's smaller memory footprint has a locality advantage that eventually compensates for its extra work. The largest difference in memory usage between txn-pred-soft and txn-pred-rc occurs for workloads that perform transactional `get` on an empty map for many different keys. In this case txn-pred-rc's predicate map will contain only a few entries, while txn-pred-soft's may grow quite large. For single-threaded access and 2^{21} key range, this 0% hit rate scenario yields 73% higher throughput for reference counting. Once multiple threads are involved, however, reference counting's locality advantage is negated by shared writes to the underlying predicate map, and txn-pred-soft performs better across all key ranges and hit rates.

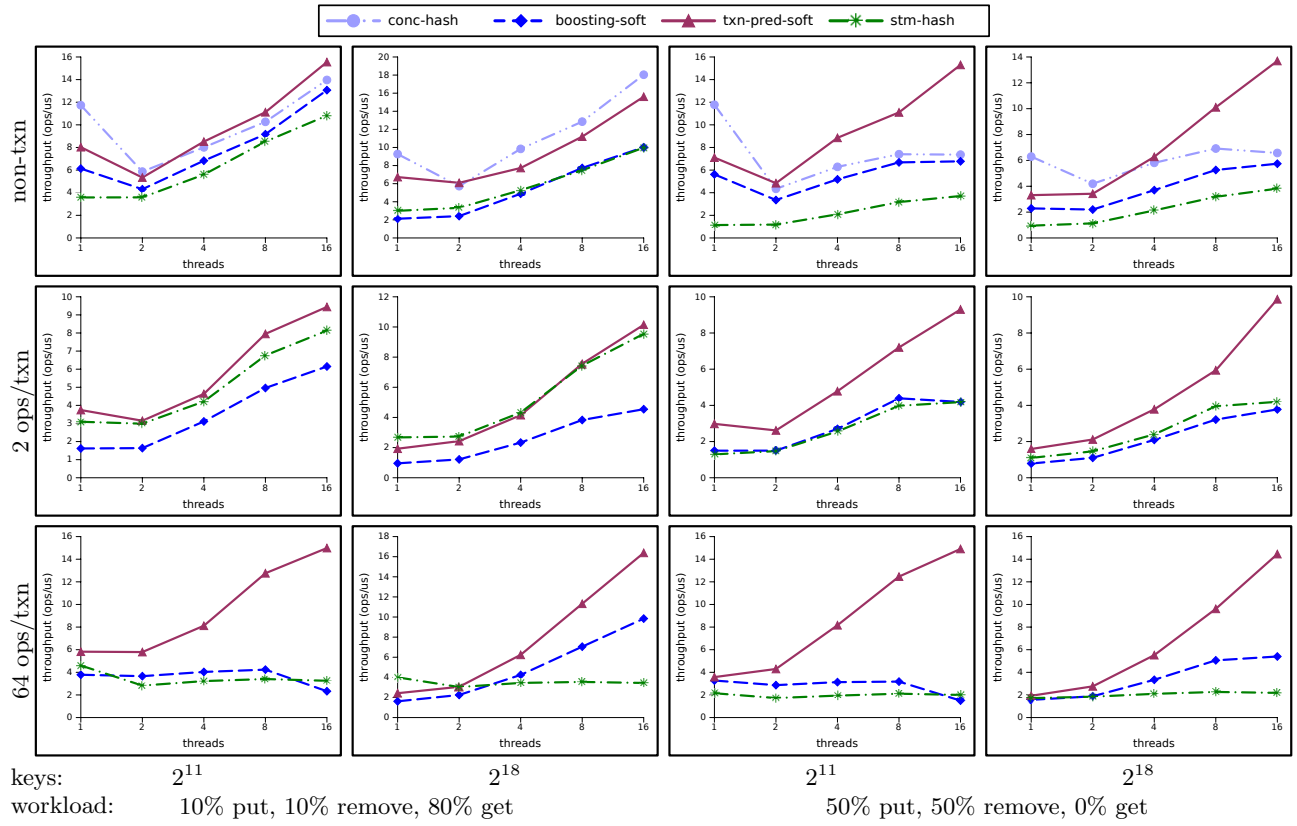


Figure 5: Throughput of transactional map implementations across a range of configurations. Each graph plots operations per microsecond, for thread counts from 1 to 16.

Txn-pred-soft shows better overall performance than txn-pred-rc, so it is our default choice. For the rest of the performance evaluation we focus only on txn-pred-soft.

6.2 Comparison To Other Transactional Maps

Figure 5 compares the performance of txn-pred-soft to a hash table implemented via STM, and to a transactionally boosted map. Conc-hash is included in the non-txn configurations for reference:

- **stm-hash** – a hash map with 16 segments, each of which is a resizeable transactional hash table.
- **boosting-soft** – a transactionally boosted *`ConcurrentHashMap`*. Soft references are used to reclaim the locks.

An obvious feature of most of the graphs is decreasing or constant throughput when moving from 1 to 2 threads. This is a consequence of the Linux scheduling policy, which prefers to spread threads across chips. This policy maximizes the cache and memory bandwidth available to an application, but it increases coherence costs for writes to shared memory locations. We verified this by repeating experiments from Figure 5 using a single processor. For very high-contention experiments such as $\langle$non-txn, 2^{11}, 0% get$\rangle$, off-chip coherence costs outweigh the benefits of additional threads, yielding higher throughput for 8 threads on 1 chip than 16 threads on 2 chips.

Stm-hash includes several optimizations over the hash table example used in Section 3.2. To reduce conflicts from maintaining load factor information, stm-hash distributes its entries over 16 segments. Each segment is an independently resizeable hash table. In addition, segments avoid unnecessary rollbacks by updating their load factor information in an abstract nested transaction (ANT) [12]. To reduce the number of transactional reads and writes, bucket chains are immutable. This requires extra object allocation during `put` and `remove`, but improves both performance and scalability. Finally, we optimized non-transactional `get` by performing its reads in a hand-rolled optimistic retry loop, avoiding the overheads of transaction setup and commit.

The optimizations applied to stm-hash help it to achieve good performance and scalability for read-dominated non-txn workloads, and read- or write-dominated workloads with few accesses. Non-txn writes have good scalability, but their single-thread performance is poor; the constant overheads of the required transaction can't be amortized across multiple operations. Stm-hash has good single-thread performance when used by transactions that perform many accesses, but does not scale well in that situation. Each transaction updates several segments' load factors, making conflicts likely. Although the ANTs avoid rollback when this occurs, conflicting transactions cannot commit in parallel.

Txn-pred-soft is faster than boosting-soft for every configuration we tested. For non-txn workloads, predication has two advantages over boosting: 1) The optimizations of Section 4.3 mean that txn-pred's non-transactional `get(`k`)` never needs to insert a predicate, while boosting must insert a lock for k even if k is not in the map. This effect is visible in the non-txn 80% get configurations across all thread counts. 2) Boosting's scalability is bounded by the underlying *`ConcurrentHashMap`*. For write-heavy workloads

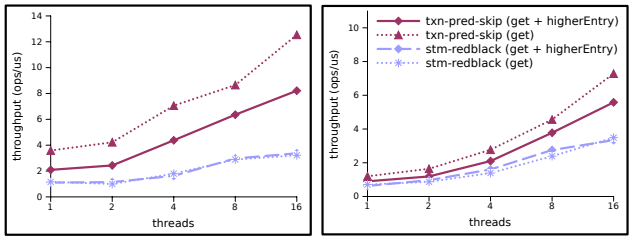
conc-hash's 16 segment locks ping-pong from core to core during each insertion or removal. Txn-pred-soft's predicates are often retained (and possibly reused) after a key is removed from the map, moving writes to lightly-contended *TVar*s. Conc-hash's bound on boosting can be clearly seen in $\langle$non-txn, 2^{11}, 0% get$\rangle$, but applies to all workloads.

Some of predication's performance advantage across all thread counts comes from a reduction in single-thread overheads. Boosting's implementation incurs a per-transaction cost because each transaction that accesses a boosted map must allocate a side data structure to record locks and undo information, and a commit and rollback handler must be registered and invoked. Txn-pred-soft uses neither per-transaction data or transaction life-cycle callbacks. Small transactions have less opportunity to amortize boosting's overhead, so the single-thread performance advantage of predication is higher for txn2 configurations than for txn64.

The remainder of predication's performance advantage is from better scaling, a result of its use of optimistic reads and its lack of interference with the STM's contention management strategy. The scaling advantage of optimistic reads is largest for small key ranges and long transactions, both of which increase the chance that multiple transactions will be accessing a map entry; see $\langle$64 ops/txn, 2^{11} keys, 80% get$\rangle$. Boosting's locks are not visible to or revocable by the STM's contention manager, so they negate its ability to prioritize transactions. This is most detrimental under high contention, such as $\langle$64 ops/txn, 2^{11} keys, 0% get$\rangle$. In this experiment boosting achieves its best throughput at 1 thread, while CCSTM's contention manager is able to provide some scalability for predication despite a large number of conflicts.

6.3 Ordered Maps

Finally, we evaluate the performance of a transactionally predicated ordered map, which adds optimistic ordered iteration and range searches to the key-equality operations. The underlying predicate map is Lea's *ConcurrentSkipListMap*. Figure 6 compares the performance of the predicated skip list to a red-black tree implemented using STM. (We also evaluated an STM skip list, but it was slower than the red-black tree.) The predicated ordered map outperforms the STM-based ordered map for both configurations and across all thread counts.



2^{11} keys 2^{18} keys

Figure 6: Throughput for ordered transactional maps performing 80% reads, either all `get` or half `get` and `higherEntry`. *SortedMap*.`higherEntry`(k) returns the entry with the smallest key $> k$.

7. RELATED WORK

7.1 Avoiding Structural Conflicts

Herlihy et al. introduced early release as a method to reduce the chance of structural conflict during tree searches in their seminal paper on dynamically-sized STM [15]. Early release allows the programmer to remove elements from a transaction's read set if it can be proved that the results of the transaction will be correct regardless of whether that read was consistent. This reasoning is subtle, especially when reasoning about STM as a means for composing operations, rather than an internal data structure mechanism for implementing linearizability. Felber et al.'s elastic transactions provide the conflict reduction benefits of early release with a more disciplined model [8]. Neither of these techniques reduces the number of transactional barriers.

Harris et al.'s abstract nested transactions (ANT) [12] allow portions of a transaction to be retried, increasing the number of transactions that can commit. ANTs could be used to insulate the caller's transaction from false conflicts that occur inside data structure operations. However, they do not avoid the need to roll back and retry the nested transaction, and add extra overheads to the base sequential case.

7.2 Semantic Conflict Detection

Semantic conflict detection using open nested transactions was described concurrently by Ni et al. [22] and Carlstrom et al. [4]. Ni et al. use open nested transactions in an STM to commit updates to transactionally managed data structures before their enclosing transaction commits, by tracking semantic conflicts with pessimistic locks. Their locks support shared, exclusive, and intension exclusive access, which enables them to support concurrent iteration or concurrent mutation, while correctly preventing simultaneous mutation and iteration. Carlstrom et al. use open nested transactions in a hardware transactional memory (HTM) to manage both the shared collection class and information about the operations performed by active transactions. This side information allows optimistic conflict detection. It is more general in form than abstract locks, and provides better fidelity than locks for range queries and partial iteration. Approaches that use open nesting still use transactions to perform all accesses to the underlying data structure, and incur additional overhead due to the side data structures and deeper nesting. This means that although they reduce false conflicts, they don't reduce STM's constant factors.

Kulkarni et al. associate lists of uncommitted operations with the objects in their Galois system [18], allowing additional operations to be added to a list only if there is no semantic conflict with the previous ones. Several application-specific optimizations are used to reduce overheads.

Herlihy et al. described transactional boosting [13], which addresses both false conflicts and STM constant factors. Boosting uses two-phase locking to prohibit conflicting accesses to an underlying linearizable data structure. These locks essentially implement a pessimistic visible-reader STM on top of the base STM, requiring a separate undo log and deadlock avoidance strategy. The resulting hybrid provides atomicity and isolation, but loses useful properties and features of the underlying STM, including starvation freedom for individual transactions, obstruction- or lock-freedom, modular blocking, and timestamp-based opacity. In addition, boosting requires that the STM linearize during commit, which eliminates the read-only transaction optimization possible in STMs such as TL2 [6] and SwissTM [7].

8. CONCLUSION

This paper has introduced transactional predication, a technique for implementing high performance concurrent collections whose operations may be composed using STM. We have shown that for sets and maps we can choose a representation that allows a portion of the transactional work to safely bypass the STM. The resulting data structures approximate semantic conflict detection using the STM's structural conflict detection mechanism, while leaving the STM completely responsible for atomicity and isolation. Predication is applicable to unordered and ordered sets and maps, and can support optimistic iteration and range queries.

Users currently face a tradeoff between the performance of non-composable concurrent collections and the programmability of STM's atomic blocks; transactional predication can provide both. Predicated collections are faster than existing transactional implementations across a wide range of workloads, offer good performance when used outside a transaction, and do not interfere with the underlying STM's opacity, modular blocking or contention management.

9. ACKNOWLEDGEMENTS

This work was supported by the Stanford Pervasive Parallelism Lab, by Dept. of the Army, AHPCRC W911NF-07-2-0027-1, and by the National Science Foundation under grant CNS–0720905.

10. REFERENCES

[1] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 81–91, New York, NY, USA, 2007. ACM.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268, New York, NY, USA, January 2010. ACM.

[3] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days 2010*, April 2010.

[4] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 56–67, New York, NY, USA, 2007. ACM.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208. Springer, March 2006.

[7] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM.

[8] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC '09: Proceedings of the 23rd International Symposium on Distributed Computing*, pages 93–107. Springer, 2009.

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 258–264, New York, NY, USA, 2005. ACM.

[10] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[11] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[12] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, August 2007.

[13] M. Herlihy and E. Koskinen. Transactional Boosting: A methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[14] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems*, December 2006.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, San Francisco, CA, USA, 2008.

[17] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 82–91, New York, NY, USA, 2006. ACM.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[19] D. Lea. Concurrent hash map in JSR 166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`, December 2007.

[20] M. M. Michael. Hazard Pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel Distributed Systems*, 15(6):491–504, 2004.

[21] J. E. B. Moss. Open nested transactions: Semantics and support. In *Poster at the 4th Workshop on Memory Performance Issues (WMPI-2006)*. February 2006.

[22] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–78, New York, NY, USA, 2007. ACM Press.

[23] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT–STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, March 2006. ACM Press.

[24] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM.