Snapshot Isolation for Software Transactional Memory

Torvald Riegel
Dresden University of Technology, Germany
torvald.riegel@tu-dresden.de

Christof Fetzer
Dresden University of Technology, Germany
christof.fetzer@tu-dresden.de

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

ABSTRACT

Software transactional memory (STM) has been proposed to simplify the development and to increase the scalability of concurrent programs. One problem of existing STMs is that of having long-running read transactions co-exist with shorter update transactions. This problem is of practical importance and has so far not been addressed by other papers in this domain. We approach this problem by investigating the performance of a STM using snapshot isolation and a novel lazy multi-version snapshot algorithm to decrease the validation costs - which can increase quadratically with the number of objects read in STMs with invisible reads. Our measurements demonstrate that snapshot isolation can increase throughput for workloads with long transactions. In comparison to other STMs with invisible reads, we can reduce the validation costs by using our lazy consistent snapshot algorithm.

1. INTRODUCTION

Software transactional memory (STM) [20] has been introduced as a means to support lightweight transactions in concurrent applications. It provides programmers with constructs to delimit transactional operations and implicitly takes care of the correctness of concurrent accesses to shared data. STM has been an active field of research over the last few years, e.g., [11, 13, 7, 12, 18, 17, 4, 10, 8].

In typical application workloads one cannot always expect that all transactions are short. One would expect that applications have a mix of long-running read transactions and short read or update transactions. One problem of existing STMs is that of having long-running read transactions efficiently co-exist with shorter update transactions. STMs typically perform best when contention is low. For transactions one should expect that the probability of conflicts increases with the length of a transaction. This problem is of practical importance but has so far not yet been addressed by the other papers in this domain. We address this problem by investigating the performance of a STM using *snapshot isolation* [1].

The key idea of snapshot isolation (a more precise description is given below) is to provide each transaction T with a consistent snapshot of all objects and all writes of T occur atomically but possibly at a later time than the time at which the snapshot is valid. This decoupling of the reads and the writes has the potential of increasing the transaction throughput but gives application developers possibly less ideal semantics than, say, STMs that guarantee serializability [2] or linearizability [14].

Snapshot isolation (SI) has been used in the database domain to address the analog problem of dealing with long read transactions in databases. STMs and databases are sufficiently different such that it is a priori not sure that (P1) SI will improve the throughput of a STM sufficiently and (P2) SI provides the right semantics for application programmers. In this paper we focus on problem P1 and will only briefly discuss P2. Note that engineering is about tradeoffs and typically application developers are willing to accept weaker (or, less ideal) semantics if the performance gain is sufficiently high over stronger (or, more ideal) alternatives. Hence, the answer to P2 will inherently depend on the answer of P1.

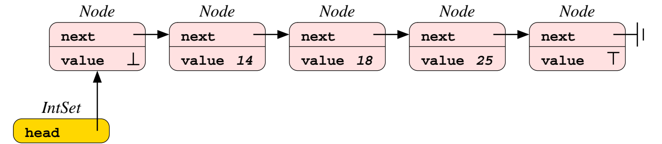

Figure 1: Integer set example.

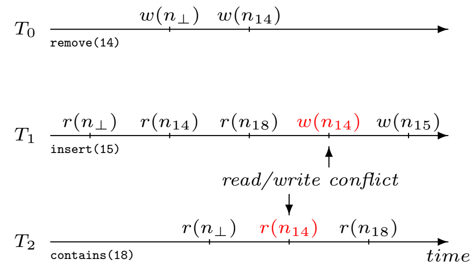

Figure 2: Two sample transactions.

Example 1. *We shall illustrate our work with the same example as in [13], i.e., an integer set implemented as a linked list. Specific values can be added to, removed from, or looked up in the set. Figure 1 shows an instance of an integer set with five nodes representing 3 integers (14, 18, and 25) and two special values ($\perp$ and $\top$) used to indicate the first and last elements of the linked list. We shall denote these nodes by n_{14}, n_{18}, n_{25}, $n_\perp$, and $n_\top$, respectively.*

Consider transactions T_1 inserting integer 15 in the set and T_2 looking up integer 18 (Figure 2). T_1 must traverse the first three nodes of the list to find the proper location for inserting the new node, create a new node, and link it

to the list. Three nodes ($n_\perp$, n_{14}, and n_{18}) are accessed but only one (n_{14}) is actually updated. T_2 also traverses the first three nodes, but none of them is updated.

STM systems typically distinguish read from write accesses to shared objects. Multiple threads can access the same object in read mode (e.g., node $n_\perp$ can be read simultaneously by T_1 and T_2) but only one thread can access an object in write mode (e.g., n_{14} by T_1). Furthermore, write accesses must be performed in isolation from any read access by another transaction. For instance, assuming that T_1 tries to write n_{14} after T_2 has read n_{14} but before T_2 completes (see Figure 2), a STM system that guarantees linearizability or serializability will detect a conflict and abort (or, in the most benign cases, delay) one of the transactions. Typically, transactions that fail to commit are restarted until they eventually succeed.

For a SI-based STM, the two transaction T_1 and T_2 will not conflict because T_2 is a read transaction that accesses a consistent snapshot that is not affected by potentially concurrent writes by T_1. Update transactions like T_1 will also read from a consistent snapshot that can become stale before the time at which T_1 writes to n_{14}. The price an application programmer has to pay - in comparison to a serializable STM - is that some read/write conflicts might have to be converted into write/write conflicts (see [16] for more details). For example, if an update transaction T_0 removes node n_{14}, we need to make sure that T_0 writes not only $n_\perp$ but also n_{14} to make sure that any concurrent transaction like T_1 that inserts a new node directly after n_{14} has a write/write conflict with T_0. □

Regarding problem P2, snapshot isolation avoids common isolation anomalies like dirty reads, dirty writes, lost updates, and fuzzy reads [1]. Because snapshot isolation circumvents read/write conflicts, application programmers might need to convert read/write conflicts into write/write conflicts if the detection of the former are needed to enforce consistency [16]. On a very high level of abstraction, this is similar to the inverse problem of deciding which objects can be released early [13]: in early release a programmer can remove the visibility of read objects while in SI a programmer might need to make certain objects in the read set "visible" by dummy writes. However, SI guarantees that the read snapshot always stays consistent which might simplify matters in comparison to using an early release mechanism.

In this paper, we propose a software transactional memory SI-STM that integrates several important features to ease the development of transactional applications and maximize their efficiency. We improve the throughput of workloads with both short transactions and long read transaction by eliminating/reducing read/write contention, by investigating a novel multi-version concurrency control algorithm that implements a variant of *snapshot isolation*. We use a variant because instead of letting always the first committer win, we let a contention manager decide which transaction wins a write/write conflict. We have developed an original algorithm to implement a multi-version isolation level based on snapshot isolation that can—if so requested—ensure linearizability of transactions. This algorithm is implemented without using any locks, which are known to severely limit scalability on multi-processor architectures and introduce the risk of deadlocks and software bugs.

Our experimental evaluation of a prototype implementation demonstrates the benefits of our architecture. The performance of our prototype is competitive with lock-based implementations and it scales well in our benchmarks.

The rest of the paper is organized as follows: Section 2 discusses related work and Section 3 introduces the principle of snapshot isolation more precisely and describes efficient algorithms to implement it, with or without additional linearizability of individual transactions. Section 4 presents our STM implementation and Section 5 describes its seamless integration in the Java language using only standard Java mechanisms. We evaluate the efficiency of our architecture and algorithms in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

2.1 Software Transactional Memory

Software Transaction Memory is not a new concept [20] but it recently attracted much attention because of the rise of multi-processor and multi-core systems. There are word-based [11] and object-based [13] STM implementations. The design of the latter, Herlihy's DSTM, is used by several current STM implementations. Our SI-STM is object-based and thus uses some of DSTM's concepts. However, SI-STM is a multi-version STM, whereas in DSTM objects have only a single version. Furthermore, existing STM implementations only provide strict transactional consistency, whereas SI-STM additionally provides support for snapshot isolation, which can increase the performance of suitable applications.

In the original STM implementations, reads by a transaction are invisible to other transactions: to ensure that consistent data is read, one must validate that all previously opened objects have not been updated in the meantime. If reads are to be visible, transactions must add themselves to a list of readers at every transactional object they read from. Reader lists enable update transactions to detect conflicts with read transactions. However, the respective checks can be costly because readers on other CPUs update the list, which in turn increases the contention of the memory interconnect. Scherer and Scott [19, 18] investigated the trade-off between invisible and visible reads. They showed that visible reads perform much better in several benchmarks but, ultimately, the decision remains application-specific. Marathe *et al.* [17] present an STM implementation that adapts between eager and lazy acquisition of objects (i.e., at access or commit time) based on the execution of previous transactions. However, they do not explore the trade-off between visible and invisible reads but suggest that adaptation in this dimension could increase performance. Cole and Herlihy propose a snapshot access mode [4] that can be roughly described as application-controlled invisible reads for selected transactional objects with explicit validation by the application. The only STM that we are aware of having a design similar to ours is [3]. However, in their STM design, every commit operation, including the upgrade of transaction-private data to data accessible by other threads, synchronizes on a single global lock. Thus, this design is not fault-tolerant because there is no roll-back mechanism for commits. Additionally, even in cases where write operations do not conflict, only a single thread can be used for updating memory. No performance benchmark results are provided.

Read accesses in our SI-STM are invisible to other trans-

actions but do not require revalidation of previously read objects on every new read access. The multi-version information available to each transactional objects provides inexpensive validation by inspection of the timestamps of each version (without having to access previously read objects). We thus get the benefits of invisible reads but at a much lower cost.

Most STM implementations support explicit transaction demarcation and read and write operations, whereas only a few provide more convenient language integration. Harris and Fraser propose adding guarded code blocks to the Java language [11], which are executed as transactions as soon as the guard condition becomes true. SXM [9] is an object-based STM implementation in C#, which uses attributes (similar to Java annotations) for the declaration of transaction boundaries but requires additional code to call a transaction (i.e., the call is different from a normal method call). They suggest extending the C# post-processor to implicitly start transactions. In contrast, our SI-STM employs widely used aspect weavers and Java's annotations to transparently add transaction support. It does not require any changes to the programming language.

Most STM implementations are obstruction-free and use contention managers [13] to ensure progress. Scherer and Scott presented several contention managers [19, 18] including the Karma manager used in Section 6. Guerraoui *et al.* investigated how to mix different managers [9] and presented the Greedy [10] and FTGreedy [8] managers, which respectively guarantee a bound on response time and achieve fault-tolerance.

2.2 Snapshot Isolation

Snapshot isolation was first proposed by Berenson *et al.* [1] and is used by several database systems. Elnikety *et al.* present a variant [5] of snapshot isolation in which transactions are allowed to read versions of data that are older than the start timestamp of the transaction. They use this weaker notion for database replication but require conventional snapshot isolation for transactions running on the same database node.

Conditions under which non-serializable executions can occur under snapshot isolation are analyzed by Fekete *et al.* [6]. They show how to modify applications to execute correctly under snapshot isolation and show that the TPC-C benchmark, an important database benchmark that is representative for real-world applications, runs correctly under snapshot isolation.

Lu *et al.* formalize in [16] the conditions under which transactions can be safely executed with snapshot isolation. They use a notion of semantic correctness instead of strict serializability. This way, the checks that have to be performed to ensure correctness are reduced to the combinations between the postcondition of the set of all read operations of a transaction and the write operations of other transactions. No further intermediate states have to be considered. We have used their conditions to construct SI-safe implementations of a linked list and a skip list.

3. SNAPSHOT ISOLATION

The idea of *snapshot isolation* [1] is to take a consistent snapshot S_T of the data at the time $start_T$ when a transaction T starts, and have T perform all read and write operations on S_T. When an update T tries to commit, it has to get a unique timestamp $commit_T$ that is larger than any existing *start* or *commit* timestamp. Snapshot isolation avoids write/write conflicts based on the *first-committer-wins* principle: if another transaction T_2 commits before T tries to commit and T_2's updates are not in T's snapshot S_T, i.e., $commit_{T_2} > start_T$, then T has to be aborted.

Snapshot isolation does not guarantee serializability but avoids common isolation anomalies like dirty reads, dirty writes, lost updates, and fuzzy reads [1]. Snapshot isolation is an optimistic approach that is expected to perform well for workloads with short update transactions that conflict minimally and long read-only transactions. This matches many important application domains and slight variations of snapshot isolation are used in common databases like Oracle and Microsoft SQL server [6]. Hence, we are investigating if snapshot isolation could be a good foundation for STMs too.

3.1 Design and Semantics

Our *SI-STM* provides the same properties as standard snapshot isolation except that we do not enforce the first-committer-wins principle. Instead, as in other obstruction free STM implementations, we use contention managers to arbitrate write/write conflicts. We also provide the option to enforce linearizability for transactions: at commit time, we check for read/write conflicts and only permit transactions to commit if they have neither write/write nor read/write conflicts.

Our major goal was to develop a lightweight snapshot algorithm that can both decrease the overhead of snapshot isolation and maximize the freshness of the objects used in a transaction. The motivation behind the freshness requirement is twofold. First, to address the often heard critique about snapshot isolation being difficult to use because it accesses old data. Second, to reduce the number of write/write conflicts and the memory footprint of the system (by facilitating that old versions be discarded earlier). Indeed, the fresher the data in the snapshot, the lower is the probability of having a write/write conflict because it might contain the newest data written by other transactions.

The main feature of our design is a *lazy interval snapshot* algorithm. Instead of taking a snapshot at the start of a transaction T, we lazily acquire a snapshot: we add a copy of an object o to the snapshot just before T accesses o for the first time. Preferably, we would like to add o's *latest version*, i.e., a copy taken after the most recent committed transaction that updated o. However, this might not guarantee that the snapshot remains *consistent*. We say that a snapshot S_T is *consistent* iff there exists a time t such that each copy c_i of object o_i in S_T corresponds to the most recent version of o_i at time t.

To keep a snapshot consistent, one could perform a validation of the snapshot whenever adding a new object to the read set. A naive validation would be quadratic in the size of the read set. This would be unacceptable for large transactions. To address this issue, we designed a new algorithm to determine the consistency more efficiently.

Each transaction T lazily acquires a consistent *interval snapshot* S_T that is *valid* within an non-empty *validity interval* $V_T = [min_T, max_T]$: each copy c_i of object o_i in S_T is the most recent version of o_i for any time in V_T and no other transaction can commit a newer version of o_i in interval $(min_T, max_T]$. The validity interval is computed on

the fly according to the objects read by the transaction and their available versions. Of course different transactions will share a copy c_i as long as these transactions only perform read accesses.

Let $first_T$ be the time when transaction T accesses its first object. Our algorithm constructs a snapshot S_T with validity interval $V_T = [min_T, max_T]$, where $max_T \geq min_T$. We guarantee that the snapshot is valid at some point in time that follows, or coincides with, the first access, i.e., $max_T \geq first_T$. The validity interval of the snapshot can be such that $min_T > first_T$. This means that, unlike other optimizations of snapshot isolation that use snapshots of the past, we can actually take a snapshot of the future, i.e., not yet valid at the time the transaction starts processing. To simplify matters, we define the effective start time of transaction T as $max(first_T, min_T)$. In that way, a snapshot is conceptually taken at the start of a transaction—just as expected by snapshot isolation.

3.2 Algorithm

Each update transaction T has a unique commit timestamp $commit_T$. The timestamps used in our implementation are all based on unique and monotonically increasing integer values for commit times. This allows us to associate each object o with a history of object versions $o^{v_1}, o^{v_2}, \ldots$ with $v_{i+1} > v_i$ and object version o^{v_i} being valid in the time range $[v_i, v_{i+1} - 1]$. We call this range the *validity interval* of object version o^{v_i}. It indicates that o was updated by a transaction that committed at time v_i and no other transaction has committed a new version of o within $(v_i, v_{i+1} - 1]$.

The validity interval of object versions allows us to associate the snapshot S_T, constructed lazily by a transaction T, with a validity interval $V_T = [min_T, max_T]$. V_T is the intersection of the validity intervals of all object versions in S_T. Hence, each object version in S_T was committed no later than min_T and no transaction committed another version within V_T.

Read access: When a transaction T reads an object o that is not yet in S_T, we look for the most recent version o^{v_i} with a validity interval V that overlaps V_T. We compute the new validity interval of the transaction as the intersection of V and V_T.

Write access: When a transaction T tries to update an object o for the first time, a private copy of this object is created. We only permit one transaction to acquire a private copy of an object. If a second transaction T_2 attempts to update o before T committed its changes, we have a write/write conflict. In this case, the contention manager is called to determine which of the two transactions needs to be aborted (or delayed). In that way, we perform a forward validation of update transactions.

Commit: A transaction can commit as long as its validity interval $V_T = [min_T, max_T]$ is non-empty, i.e., $max_T \geq min_T$. If we keep a sufficiently long history of objects, the validity interval will never become empty. When an update transaction commits, it receives a unique timestamp $commit_T$. Read-only transaction do not have a unique commit timestamp as they do not update objects.

Memory Overhead: In our measurements we keep a small number k of old variants for each object. In future we will change this and will use a fixed number of weak references to old variants of an object instead. In this way, the Java garbage collector will be able to automatically reclaim old variants in case more memory is needed. The memory overhead will then depend on the available memory, i.e., no additional copies are kept in case no memory is available and up to k variants if the Java virtual machine has sufficient memory available.

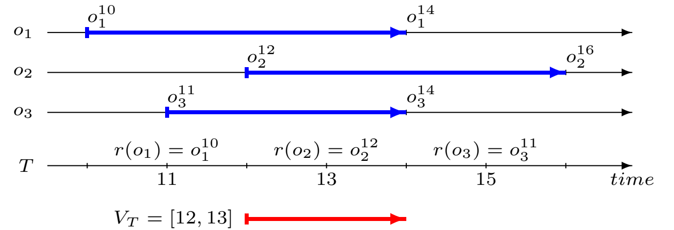

Figure 3: A transaction reading three objects.

Extension of validity intervals: When a transaction T adds the most recent object version o^{v_i} to its snapshot S_T, the time v_{i+1} at which o^{v_i} expires is not yet known (otherwise, o^{v_i} would not be the most recent version). Thus, we set the upper bound on o^{v_i}'s validity temporarily to the most recent commit time ($commit_{T_c}$, where T_c is the most recently committed transaction).

To extend the validity range of transaction T, we check if any temporary upper bound on the validity of the objects in S_T can be shifted to a later time. Our system tries to extend the validity interval V_T if V_T becomes empty. The goal of this extension is to decrease the abort frequency. Additional proactive extensions could be useful in some cases. However, deciding whether extension costs are justified by possible throughput gains is nontrivial and remains a task for future work.

EXAMPLE 2. *To illustrate the concepts of lazy snapshot isolation, consider a transaction T that reads objects o_1, o_2 and o_3 (see Figure 3). When T accesses o_2 for the first time at time* 13*, T reads the most recent version o_2^{12} of o_2 even though this version did not yet exist when T read o_1 at* 11*. When accessing o_3 at* 15*, T cannot use the most recent version o_3^{14} of o_3 because the validity intervals of o_1^{10} and o_3^{14} do not overlap. Therefore, the snapshot S of T consists of object versions o_1^{10}, o_2^{12}, and o_3^{11} with a validity interval* $V_T = [12, 13]$. □

3.3 Linearizability

We have implemented an optimistic approach that can enforce linearizability [2] of transactions. If a programmer requests linearizability, a transaction T can only commit at time $commit_T$ if its validity interval contains time $commit_T - 1$, i.e., all objects read by T are still valid at the time T commits. The intuition is that all object versions in T's snapshot are valid up to T's commit time and, hence, there are neither read/write nor write/write conflicts affecting T.

To minimize aborts, a transaction T will try to extend its validity interval before committing. If there are no read-/write conflicts, i.e., no objects of T's read-set have been updated, T will be able to extend the validity interval to the current time and consequently commit.

4. STM IMPLEMENTATION

We now describe the architecture developed to support lightweight transactions in Java. Our transactional mem-

ory is implemented as a software library. The main components exposed to the application developer are *transactions* and *transactional objects*. In addition, it features a modular architecture for dealing with contention and transaction management.

4.1 Transactions

Transactions are implemented as thread-local objects, i.e., the scope of a transaction is confined inside the current thread of control. The application developer can programmatically start a transaction, try to commit it, or force it to abort.

As in [13], transaction objects (see Figure 4) contain a status field, initially set to `ACTIVE`, that can be atomically changed to either `COMMITTED` or `ABORTED` using a *compare and swap* (CAS) operation*depending on whether the transaction successfully completes or not. A transaction object can additionally keep track of the objects being read and updated (read-set and write-set) and maintains timestamps indicating the transaction's start and commit times. Timestamps are discrete values generated by a global lock-free counter that can be atomically incremented and read.

4.2 Transactional Objects

Transactional objects are STM-specific wrappers that control accesses to application objects. They manage multiple version of the object's state on behalf of active transactions. Regular objects being wrapped must be able to duplicate their state, i.e., clone themselves, as transactional wrappers need to create new versions.

Before being used by the application, a transactional object must be "opened", i.e., a reference to the current state of the application object must be acquired. A transactional object can be opened for reading or for writing. If a transaction opens the same object multiple times, the same state is returned. An object opened for reading can be subsequently opened for writing (similar to *lock promotion* in databases). Opening a transactional object may fail and force the current transaction to abort.

4.3 Contention Management

Conflicts are handled in a modular way by the means of *contention managers*, as in [13]. Contention managers are invoked when a conflict occurs between two transactions and they must take actions to resolve the conflict, e.g., by aborting or delaying one of the conflicting transactions. Contention managers can take decisions based on information stored in transaction objects (read- and write-set, timestamps), as well as historical data maintained over time. In particular, contention managers can request to be notified of transactional events (start, commit, abort, read, write) and use this information to implement sophisticated conflict resolution strategies.

4.4 Transaction Management

Our STM implementation currently supports two transaction management models. The first one is very similar to the SXM of Herlihy *et al.* [9], which is in turn similar to DSTM [13] but uses visible reads. It allows multiple readers or a single writer—but not both—to access a given object. Updates to a shared object are performed on a transaction-local copy, which becomes the *current* version when the transaction commits. A single consistent version of each shared object is maintained at a given time. Support for SXM has been implemented essentially for comparison purposes and we shall not describe it further.

The second transaction management model, termed SI-STM, implements multi-version concurrency control and snapshot isolation as described in Section 3. Shared objects are accessed indirectly via *transactional wrappers* that can be invoked concurrently by multiple threads and effectively behave as transactional objects.

Transactional objects maintain a reference to a descriptor, called *locator* [13], that keeps track of several versions of the object's state (see Figure 4): a *tentative* version being written to by an update transaction (`tentative`); a *committed* version (`state`) together with its commit timestamp (`commit_ts`); and the n *previous* committed versions of the object (`old_versions`) together with their commit timestamp. n is a small value that is typically between 1 and 8. A locator additionally stores a reference to the *writer*, i.e., the transaction that updates the tentative version, if any (`transaction`). Note that the locator does not keep track of transactions that read the object.

References to a locator can be read atomically and updated using a CAS operation. Once a locator has been registered by a transactional object, it becomes immutable and is never modified. When a transactional object is created, its locator is initialized with the state of the object being wrapped as committed version, and 0 as commit timestamp; other fields are set to null.

We define the *current* version of the object as follows: if the `transaction` field of the locator is null, or if the last writer has aborted, then the current version corresponds to the committed version of the object (`state`) with its associated commit timestamp (`commit_ts`); if the last writer has committed, then the current version corresponds to the tentative version of the object (`tentative`) with a commit timestamp equal to that of the writer; finally, if the writer is still active, the current version is undefined.

When a transaction accesses an object in write mode for the first time, we check in the current locator whether there is already an *active* writer. If that is the case, there is a conflict and we ask the contention manager to arbitrate between both transactions before retrying. Otherwise, if a validity condition to be described shortly is met, we create a new locator and register the current transaction as writer. We store references to the current and previous versions in the new locator and we create a new tentative version by duplicating the state of the current version. Finally, we try to update the reference to the locator in the transactional object using a CAS operation. If this fails, then a concurrent transaction has updated the reference in the meantime and we retry the whole procedure. Otherwise, the current transaction continues its execution by accessing its local tentative version.

EXAMPLE 3. *Consider the example in Figure 5. Transaction T_1 is registered as writer in the locator of the transactional object. As T_1 has committed, the tentative version corresponds to the current state of the object, with a commit timestamp of* 53. *Transaction T_2 accesses the transactional*

*A CAS operation on a variable takes as argument a new value v and an expected value e. It atomically sets the value of the variable to v if the current value of v is equal to e. It returns the value of v that was read.

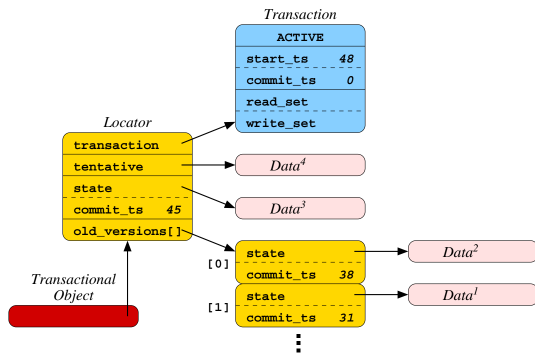

Figure 4: Sample locator for a transactional object with an active writer. The latest consistent version is $Data^3$ with validity starting at time 45.

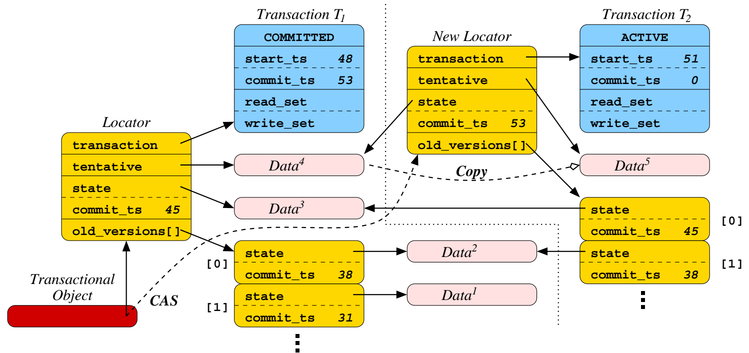

Figure 5: Sample locator for a transactional object with a committed writer T_1 (left). Another transaction T_2 opens the transactional object in write mode and creates a new locator (right).

object in write mode and creates a new locator, with versions shifted by one position with respect to the old locator (the old tentative version becomes the new committed version). Then, T_2 creates a copy of the current state as tentative version and uses a CAS operation to update the reference to the locator in the transactional object. □

One can note that the algorithm for accessing transactional objects in write mode follows the same general principle as in DSTM, with variations resulting principally from versioning and timestamp management. In contrast, read operations are handled in a very different manner. As a matter of fact, the key to the efficiency of our SI-STM model is that no modification to the locator nor validation of previously read objects is necessary when accessing a transactional object in read mode.

Each version has a *validity range*, i.e., an interval between two timestamps during which the version was representing the current state. This range starts with the commit timestamp of the version and ends one time unit before the commit timestamp of the next version. For instance, in Figure 4, $Data^1$ and $Data^2$ have validity ranges of $[31, 38)$ and $[38, 45)$, respectively; $Data^3$ has a validity range starting at 45 with an upper bound still unknown. For each transaction, we also maintain a validity range that corresponds to the intersection of the validity ranges of all the objects in its read-set. A necessary condition for the transaction to be able to commit is that this range remains non-empty.

When opening a transactional object in read mode, the transaction searches through the committed versions of the object starting by the most recent and selects the first that intersects with its validity range. If there is no such version, we try to extend the validity range of the transaction by recomputing the unknown upper bounds of the objects in the read set, as described in Section 3. If the intersection remains empty after the extend, the transaction needs to abort. In all other cases, we simply update the validity range of the transaction and return the selected version.

We can now describe the missing validity condition on write accesses. Tentative versions also have an open-ended validity range, which starts with the commit timestamp of the cloned state and must also intersect with the validity range of the transaction. Therefore, a write access will fail if the commit timestamp of the current version is posterior to the validity range of the transaction (even after an extend).

5. LANGUAGE INTEGRATION

Most of the STM implementations we know of provide explicit constructs for transaction demarcation and accesses to transactional objects. The programmer uses special operations to start, abort, or commit the transaction associated with the current thread, as well as retry transactions that fail to commit. Further, the programmer needs to explicitly instantiate transactional objects and provide support for creating copies of the wrapped objects.

Our STM implementation is no exception and features such a programmatic interface. It features a *declarative* approach for seamless integration of lightweight transaction in Java applications. To that end, we use a combination of standard techniques: the *annotation* feature of Java 1.5 together with *aspect-oriented programming* (AOP) [15]. Annotations are metadata that can be associated with types, methods, and fields and allow programmers to decorate Java code with their own attributes. Aspect-oriented programming is an approach to writing software, which allows developers to easily capture and integrate cross-cutting concerns, or *aspects*, in their applications.

5.1 Declarative STM Support

Our language integration mechanisms provide implicit transaction demarcation and transparent access to transactional objects. The programmer only needs to add annotations to relevant classes and methods. He is freed from the burden of dealing programmatically with the STM, which in turn limits the risk of introducing software bugs in complex transactional constructs.

5.1.1 Declaring transactional objects

Transactional objects to be accessed in the context of concurrent transactions must have the annotation `@Transactional`. All accesses to their methods and fields are managed by the transactional library so as to guarantee isolation. Specific methods can be additionally annotated by `@ReadOnly` to indicate that they do not modify the state of the target object; the transaction manager relies on this information to distinguish reads from writes.

As mentioned in Section 4, transactional objects should be able to clone their state. Support for object duplication is added transparently to transactional objects, provided that all their instance fields are either (1) of primitive type, or (2) immutable (e.g., strings), or (3) transactional. If that is

not the case, the transactional object should define a public method `duplicate()` that performs a deep copy of the object's state.

5.1.2 Specifying transaction demarcation

Our language integration mechanisms also feature implicit transaction demarcation: methods that have the annotation `@Atomic` will always execute in the context of a new transaction. Such atomic method are transparently reinvoked if the enclosing transaction fails to commit due to conflicting accesses to transactional objects. Transactions that span arbitrary blocks of code must using explicit demarcation.

Alternatively, a method can be declared with the `@Isolated` annotation. The difference between atomic and isolated is subtle: if an exception is raised by an atomic method, the enclosing transaction is aborted before propagating the exception to the caller; in contrast, isolated methods always commit the partial effects of the transaction before propagating the exception. The choice between atomic and isolated methods depends on the application semantics.

EXAMPLE 4. *Figure 6 presents an implementation of the integer set introduced in Example 1. Observe that the code makes no reference to STM, with the exception of the annotations. Transactional constructs are transparently weaved in the application by AOP. Compare this code with the explicit approach presented in [13].* □

5.2 AOP Implementation

Our STM implementation uses AOP to transparently add transactional support to the application based on the annotations inserted by the developer. Each object declared as transactional is extended with a reference to a transactional wrapper, methods to open the object in read and write mode, and support for state duplication.

We use AOP *around* advices to transparently create a new transaction for each call to an atomic or isolated method. Transactions that fail to commit are automatically retried. Similar advices are defined on transactional objects to intercept and redirect method calls and field accesses to the appropriate version.

The AOP weaver integrates the aspects in the application at compile-time or at load-time. In comparison with explicit transaction management, an application that uses declarative STM incurs a small performance penalty, mostly due to the additional runtime overhead of advices and the extra indirection for every access to a transactional object (instead of the first access only). Overall, the efficiency loss remains very small and is easily compensated by the many benefits of implicit transaction demarcation and transparent access to transactional objects. Note finally that declarative and programmatic constructs can be mixed within the same application.

6. PERFORMANCE EVALUATION

To evaluate the performance of our STM with snapshot isolation, we compared it with two other implementations. The first one follows the design of SXM by Herlihy *et al.* [9], an object-based STM with visible reads, with a few minor extensions. The second follows the design of Eager ASTM by Marathe *et al.* as described in [17]. Henceforth, we shall call these STM implementations *SI-STM*, *SXM*, and *ASTM*. Read operations in SXM are visible to other threads, whereas they are invisible in ASTM and SI-STM. Where appropriate, we show results for another variant of ASTM that only validates the read objects at the end of a transaction (*single-validate ASTM*). All other STM implementations guarantee that all objects read in a transaction always represent a consistent view. Note that we compare SI-STM with similarly designed STMs so as to determine the performance of snapshot isolation and SI-STM's inexpensive validation.

We use five micro-benchmarks: a simple *bank* application; two micro-benchmarks to investigate the CPU time required for the read and write operations of an STM; and an integer set implemented as a sorted *linked list*; and an integer set implemented as a *skip list*.

The bank micro-benchmark consists of two transaction types: (1) transfers, i.e., a withdrawal from one account followed by a deposit on another account, and (2) computation of the aggregate balance of all accounts. Whereas the former transaction is small and contains 2 read/write accesses, the latter is a long transaction consisting only of read accesses (one per account). To highlight the advantages of STMs, we additionally present results for fine-granular and coarse-granular lock-based implementations of these transactions, in which locks are explicitly acquired and released. The former uses one lock (standard monitor implementation) per account while the latter uses a single lock for all accounts. Note that the lock-based implementation has lower runtime overhead as it uses programmatic constructs instead of the declarative transactions of SI-STM; hence, comparison of absolute performance figures is not exactly fair.

We executed all benchmarks on a system with four Xeon CPUs, hyperthreading enabled (resulting in eight logical CPUs), 8GB of RAM, and Sun's Java Virtual Machine version 1.5.0. We used the virtual machine's default configuration for our system: a server-mode virtual machine, the Parallel garbage collector, and a maximum heap size of 1GB. We set the start size of the heap to its maximum size. Results were obtained by executing five runs of 10 seconds for every tested configuration and computing the 20% trimmed mean, i.e., the mean of the three median values. All STMs use the Karma [19] contention manager.

Figure 7 shows the throughput results for the bank application with 50 and 1024 accounts, and with 0% and 10% read transactions (other transactions are money transfers). Note that throughput is the total throughput of all threads and that the number of threads is shown with a logarithmic scale.

Under high write contention workloads (50 accounts) and without long read-only transactions, SI-STM has slightly higher overhead than SXM and ASTM. For larger numbers of accounts (not shown), throughput increases for the STMs and fine-grained locks because of less contention.

SI-STM also scales well when there are long read-all transactions, whereas SXM suffers from a high conflict rate because of visible reads and cannot take advantage of additional CPUs. Although both SI-STM and ASTM use invisible reads, the throughput of the ASTM version that always guarantees consistent reads is very low because of the validation overhead. When ASTM only performs validation at the end of a read-only transaction (single-validate), the throughput is significantly higher. However, the transactions might read inconsistent data. For example, if a transaction needs

```
@Transactional
public class Node {
  private int value;
  private Node next;

  public Node(int v) { value = v; }

  public void setValue(int v) { value = v; }
  public void setNext(Node n) { next = n; }

  @ReadOnly
  public int getValue() { return value; }
  @ReadOnly
  public Node getNext() { return next; }
}

public class IntSet {
  private Node head;

  public IntSetOOP() {
    Node min = new Node(Integer.MIN_VALUE);
    Node max = new Node(Integer.MAX_VALUE);
    min.setNext(max);
    head = min;
  }

  // Continued in next column...
```

```
  @Atomic
  public boolean add(int v) {
    Node prev = head;
    Node next = prev.getNext();
    while (next.getValue() < v) {
      prev = next;
      next = prev.getNext();
    }
    if (next.getValue() == v)
      return false;
    Node n = new Node(v);
    n.setNext(prev.getNext());
    prev.setNext(n);
    return true;
  }

  @Atomic
  public boolean contains(int v) {
    Node prev = head;
    Node next = prev.getNext();
    while (next.getValue() < v) {
      prev = next;
      next = prev.getNext();
    }
    return (next.getValue() == v);
  }
}
```

Figure 6: Integer set implementation using declarative STM support.

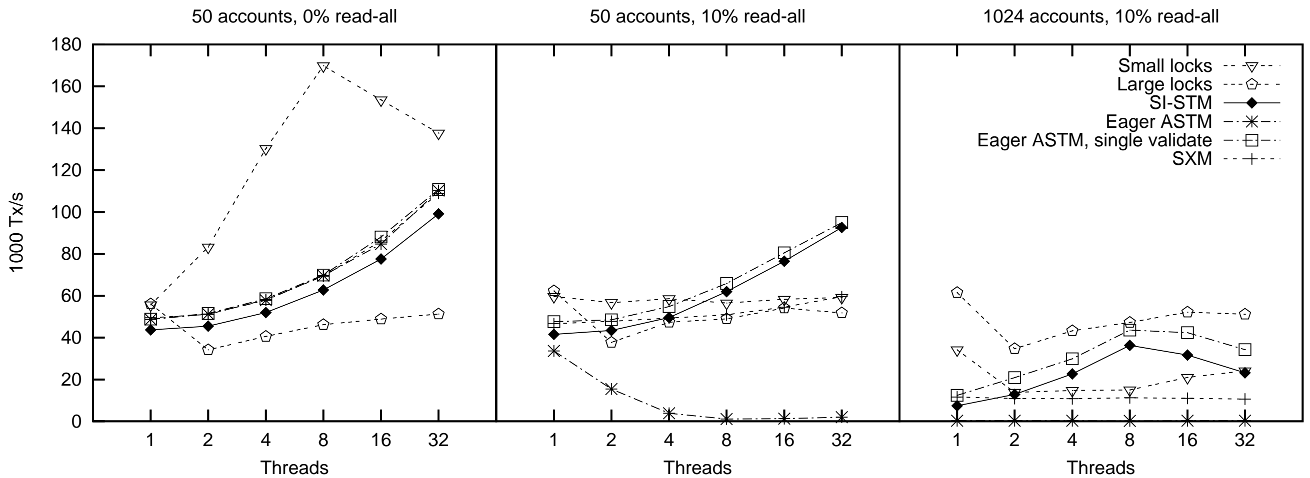

Figure 7: Throughput results for the bank benchmark.

to read all elements of a linked-list–based queue, it needs to validate its read set during the transaction to guarantee that it terminates even when the queue is being modified by other transactions.

If the number of accounts is large (1024) and, as a result, write contention and the chance that an object gets updated is low, SI-STM and single-validate ASTM outperform the other STM variants. However, if there is more than one thread per CPU, the throughput of the STMs using invisible reads decreases because preemption of threads decreases the chance of optimistically obtaining a consistent view.

To highlight the differences between STM designs that use visible and invisible reads, Figure 8 shows the CPU time required for one read operation for read-only transactions of different sizes. In this micro-benchmark, 8 threads read the given number of objects. All transactions read the same objects (with the exception of the SXM benchmark run with disjoint accesses) and there are no concurrent updates to these objects. The fixed overhead of a transaction gets negligible when the number of objects read during the trans-

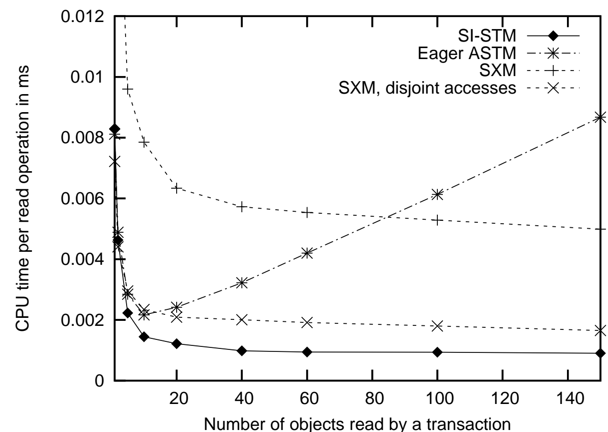

Figure 8: SI-STM read overhead

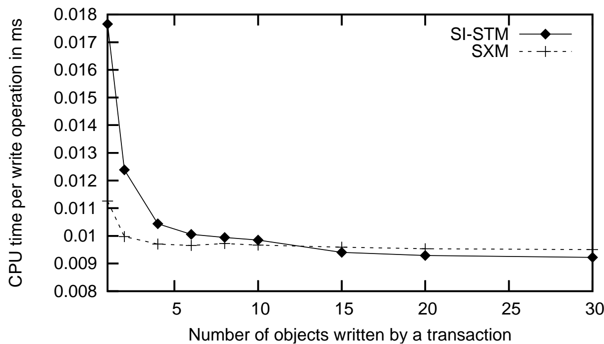

Figure 9: SI-STM write overhead

action is high. SXM's visible reads have a higher overhead than SI-STM's invisible reads. This overhead consists of the costs of the CAS operation and possible cache misses and CAS failures if transactions on different CPUs add themselves to the reader list of the same object. ASTM has to guarantee the consistency of reads by validating all objects previously read in the transaction, which increases the overhead of read operations when transactions get larger. Note that, although not shown here, ASTM transactions with only a single validate at the end of each transaction perform very similar to SI-STM.

SI-STM requires a central counter for the timestamps that it needs for update transactions. SXM and ASTM do not need such a counter, which is a source of contention if the rate of commits is high. Figure 9 shows the overhead of write operations in SI-STM by means of a micro-benchmark similar to the one used for Figure 8. However, now the 8 threads write to disjoint, thread-local objects. Acquiring timestamps induces a small overhead, which, however, gets negligible when at least 10 objects are written by a transaction. Furthermore, the overhead is smaller than the costs of a single write operation. However, the results in Figure 8 and Figure 9 are of course hardware-specific.

Figure 10 shows throughput results for two micro-benchmarks that are often used to evaluate STM implementations, namely integer sets implemented via sorted linked lists and skip lists. Each benchmark consists of read transactions, which determine whether an element is in the set, and update transactions, which either add or remove an element. For SI-STM, we present two results. First, modified implementations of the integer sets that operate correctly when the STM provides snapshot-isolation, labeled as *SI-safe*; these variants were obtained by adding some write accesses and using the correctness conditions given in [16]. Second, the original (sequential) implementations (see Figure 6) that require strict transactional consistency and for which SI-STM is configured to ensure linearizability. Distinguishing between these variants allows us to show the performance impact of snapshot isolation and inexpensive validation separately. We do not release objects early. Although early release decreases the possibility of conflicts, it can mainly be used in cases in which the access path to an object is known. We use the linked list to conveniently model transactions in which a modification takes place, which depends on a large amount of data that might be modified by other transactions. Note that, for this type of transactions, lazily acquiring updated objects makes not much of a difference because the update operations are near the end of the transaction. Thus, using Eager ASTM should give representative results.

For the skip list, STMs using invisible reads (ASTM and SI-STM) show good scalability and outperform SXM, which suffers from the contention on the reader lists. However, the transactions in the linked list benchmark are quite large (the integer sets contain 250 elements) and ASTMs validation is expensive. SI-STM, on the contrary, uses version information to compute the validity range much faster and scales well up to the number of available CPUs.

The SI-safe variants perform better than the original implementations if the number of objects read by a transaction is large, as in the linked list benchmark. On the other hand, the overhead of the validation phase required to ensure linearizability is negligible in the skip list benchmark, where the number of read objects is smaller. Furthermore, transactions are shorter, which decreases the probability of concurrent updates resulting in a failed validation. SI-STM enables the user to choose between both alternatives depending on application specifics and performance requirements. Note that SI-STM with linearizability still outperforms SXM and ASTM in most cases: applications can benefit from SI-STM even without using snapshot isolation and its additional engineering costs.

For all benchmark results for SI-STM shown here, the maximum number of versions kept per object was 8. During several tests with these benchmarks, we have noticed that the maximum number of versions often had only a small influence on the throughput. Keeping one or two versions was sufficient to achieve similar and sometimes even better results than with 8 versions. We also found that, in our benchmarks, single-version STMs and SI-STM are throughput-wise similarly affected by garbage collection overheads when the heap size is small. We are currently investigating how weak references and proactively extending the validity range affect the properties of SI-STM.

7. CONCLUSION

We have designed, implemented, and evaluated a software transaction memory architecture (SI-STM) based on a variant of snapshot isolation. In this variant we use a contention manager to support the first-committer-wins principle. The performance of SI-STM is competitive even with manual lock-based implementations that do not have the overhead of AOP. Our benchmarks point out that SI-STM shows good performance in particular for transaction workloads with long transactions. Our novel lazy snapshot algorithm can reduce the validation cost in comparison to other STMs with invisible reads like ASTM.

8. REFERENCES

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD*, pages 1–10, 1995.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In *Proceedings of SCOOL*, 2005.

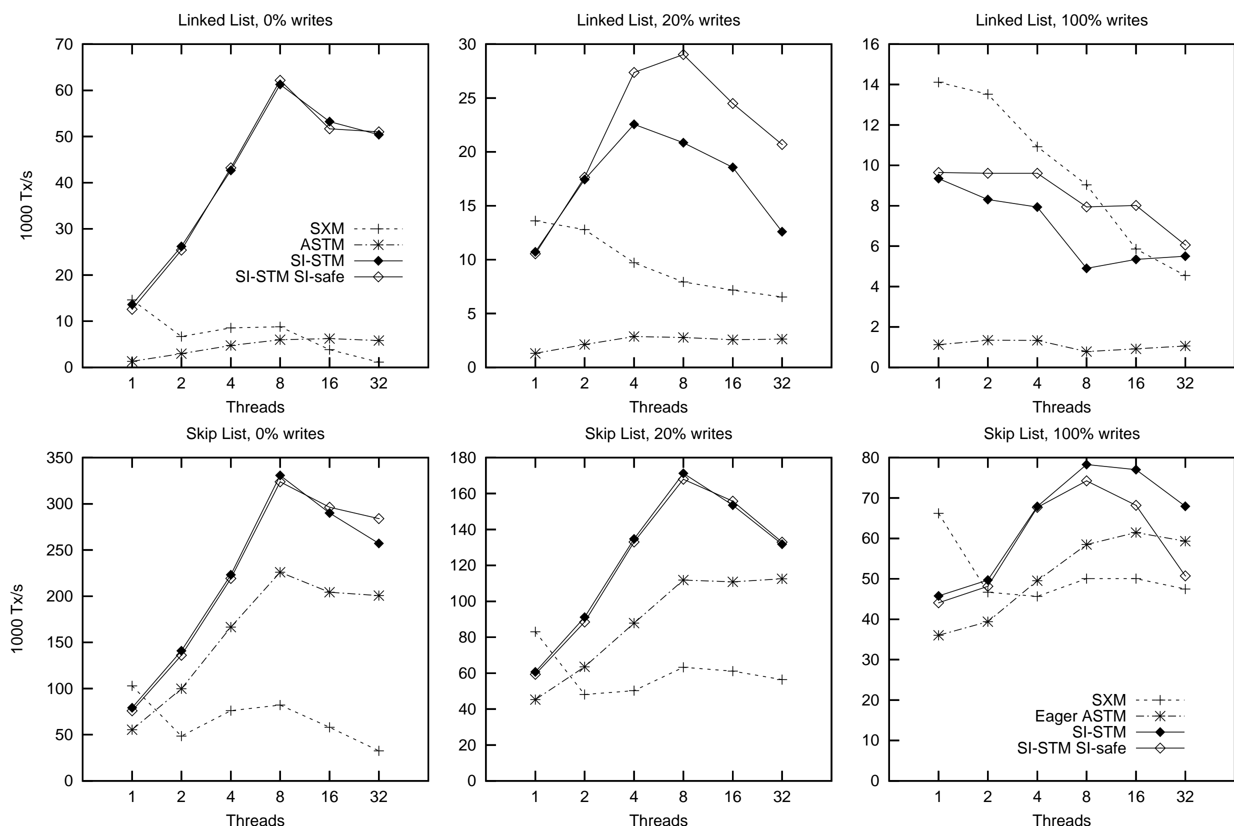

Figure 10: Throughput results for the linked list (top) and skip list (bottom) benchmarks.

[4] C. Cole and M. Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 2005. To appear.

[5] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proceedings of SRDS*, pages 73–84, Oct 2005.

[6] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2), 2005.

[7] P. Felber and M. Reiter. Advanced concurrency control in Java. *Concurrency and Computation: Practice & Experience*, 14(4):261–285, 2002.

[8] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *Proceedings of SCOOL*, 2005.

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proceedings of DISC*, Sep 2005.

[10] R. Guerraoui, M. Herlihy, and S. Pochon. Toward a theory of transactional contention managers. In *Proceedings of PODC*, Jul 2005.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[12] M. Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *Proceedings of PLDI*, 2005.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC*, pages 92–101, Jul 2003.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP*, 1997.

[16] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[17] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of DISC*, pages 354–368, 2005.

[18] W. Scherer III and M. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC*, pages 240–248, Jul 2005.

[19] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[20] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of PODC*, Aug 1995.