Committing Conflicting Transactions in an STM

Hany E. Ramadan
University of Texas at Austin
ramadan@cs.utexas.edu

Indrajit Roy
University of Texas at Austin
indrajit@cs.utexas.edu

Maurice Herlihy
Brown University
mph@cs.brown.edu

Emmett Witchel
University of Texas at Austin
witchel@cs.utexas.edu

Abstract

Dependence-aware transactional memory (DATM) is a recently proposed model for increasing concurrency of memory transactions without complicating their interface. DATM manages dependences between conflicting, uncommitted transactions so that they commit safely.

The contributions of this paper are twofold. First, we provide a safety proof for the dependence-aware model. This proof also shows that the DATM model accepts all concurrent interleavings that are conflict-serializable.

Second, we describe the first application of dependence tracking to software transactional memory (STM) design and implementation. We compare our implementation with a state of the art STM, TL2 [5]. We use benchmarks from the STAMP [22] suite, quantifying how dependence tracking converts certain types of transactional conflicts into successful commits. On high contention workloads, DATM is able to take advantage of dependences to speed up execution by up to $4.8\times$.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architecture; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.4.1 [*Operating Systems*]: Process Management—Concurrency; Synchronization; Threads

General Terms Algorithms, Design, Performance

Keywords transactional memory, dependence-aware, serializability, concurrency control

1. Introduction

Interest in programming with transactions has experienced a renaissance with the advent of multi-core processors. Transactional memory [19], whether in software (STM) [31] or hardware (HTM) [16], replaces locks with transactions, promising an easier programming model without compromising performance.

Memory transactions provide *atomicity*: if a transaction fails for any reason its effects are discarded. Transactions provide *isolation*: no transaction sees the partial effects of any other. Transactions are also *linearizable* meaning that each committed transaction appears to take effect instantaneously at some point between when it starts and when it finishes.

A transactional *conflict* occurs when one transaction writes data that is read or written by another transaction. When the ordering of all conflicting memory accesses is identical to a serial execution order of all transactions, the execution is called *conflict-serializable* [8]. There are three kinds of memory conflicts, R→W, W→R, and W→W, where R is a read, W is a write, and R→W indicates that a memory location was first read by one transaction and subsequently written by another.

Most transactional memory systems detect conflicts between pairs of transactions and respond by delaying or restarting one of the transactions. The key insight behind DATM is that this policy restricts concurrency more than is necessary [2]. DATM manages conflicts by making transactions aware of dependences and, in the case of a W→R dependence, forwards data values from one uncommitted transaction to another. Dependence-awareness allows two conflicting but conflict-serializable transactions to both commit safely, increasing concurrency and making better use of concurrent hardware [27].

This paper presents a formal model for dependence-aware transactions, proving them to be *safe* because transactions remain atomic and isolated. Moreover, the paper shows that the model allows *every* conflict-serializable schedule to occur.

The paper demonstrates the first application of dependence-awareness to STMs. Dependence-aware software transactional memory (DASTM) uses techniques from TL2 [5] that are modified to support dependences and data forwarding. We implement DASTM in C and in Java. The C version is word-based and the Java implementation is object-based. We evaluate DASTM on high contention workloads—a shared counter micro-benchmark, three programs from the STAMP benchmark suite [23], and STMBench7 [9]. We show that some STAMP benchmarks benefit from managing dependences and forwarding data between uncommitted transactions.

This paper makes the following contributions:

1. It presents a formal model for dependence-aware transactions and then proves its safety and its ability to accept any interleaving that is conflict-serializable.
2. It presents the design of a dependence-aware STM system, which we have implemented in both C and Java.
3. It presents performance data and statistics for our dependence-aware STM system. DASTM shows up to a 4.8× performance improvement on high contention benchmarks from STAMP. The performance improvement is directly attributable to DASTM's ability to manage dependences.

We summarize the dependence-aware model and explain its benefits in Section 2. Section 3 presents our formal model and proof. Section 4 describes our dependence-aware STM implementation, and Section 5 presents performance numbers for the prototype implementation and compares it to existing systems. Section 6 discusses related work and Section 7 concludes.

2. Dependence-aware model

The dependence-aware model accepts more concurrent interleavings than current transactional memory safety mechanisms [27]. This section summarizes the most important parts of the model to keep the paper self-contained, and Section 3 formalizes the model. The details of the model are available elsewhere [27].

2.1 Safely committing conflicting transactions

In conventional TM systems, if two transactions access the same datum and at least one of the accesses is a write, then one transaction must either block or restart. Making transactions dependence-aware removes this restriction, allowing both transactions to commit safely under certain circumstances. Active (in-progress) transactions are coordinated by two mechanisms: ordering and forwarding speculative data.

If transaction A reads a datum and then transaction B writes it (R→W), both transactions can continue so long as transaction A commits or aborts first. An implementation needs a mechanism to delay B's commit. W→W dependences require the same mechanism.

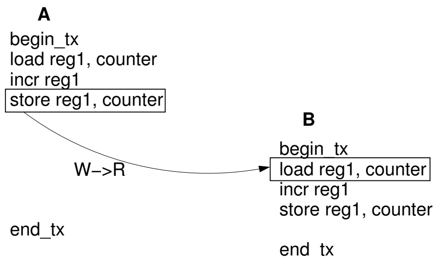

Figure 1. Two transactions increment the same counter, illustrating an interleaving where two conflicting transactions can both safely commit.

If transaction A writes a datum and then transaction B reads it (W→R), the system can forward the speculative data from A to B and make sure that A commits first. An implementation needs a mechanism to detect if A overwrites the data or restarts, in which case the runtime system restarts B.

2.2 Example: shared counter

Figure 1 shows two transactions, A and B, both of which increment a shared counter. A reads the counter value, which is initialized to 0, increments it to 1 and stores it back. B then reads the counter value. The system establishes the W→R dependence and forwards the value (1) from A. B increments the value to 2 and stores it. The system makes sure that B does not commit unless and until A commits. When both commit successfully, the final value of the counter is 2.

2.3 Dependence management

The system tracks all dependences at the level of memory bytes or objects. The dependences are tracked as they arise during transaction execution. R→W and W→W dependences restrict commit order, while W→R restrict commit order and forward data from the active transaction. When data is forwarded, the destination transaction must restart if the source restarts or overwrites the memory location whose value was previously forwarded.

Multiple dependences may arise if two transactions conflict on more than one memory cell. Dependences can form cycles among transactions, which must be broken by the system to avoid deadlock. Once detected, a cycle is resolved by restarting at least one transaction. A cyclic dependence means that the transactions have executed in a way that is not conflict serializable (this is formalized and proved in Section 3). The contention manager for dependence-aware transactions can break cycles using information from the dependence graph [27]. For example, it is probably advantageous to restart a transaction that has not forwarded any data, if possible.

2.4 Exceptions and stale data

Because the model forwards data between transactions, transactions that are doomed to restart (zombies) can read invalid or inconsistent data. Zombies will be restarted by the runtime when their source transaction either restarts or overwrites the incorrectly forwarded data. But before the zombie restarts, the runtime must contain the side effects of the zombie having read inconsistent data. The runtime effectively deals with zombie transactions by restarting them in a mode that prevents W→R dependences, thereby preventing them from reading forwarded data ("no-forward mode," details in Section 4.4).

2.5 Cascading aborts

Cascading aborts happen when one transaction's abort causes other transactions to abort. In DATM, cascading aborts arise only from W→R dependences, where the source aborts or overwrites forwarded data. This data sharing pattern, with one transaction updating a variable multiple times while other transactions read it, is not conflict serializable. Any safe transactional system will serialize such transactions, either by stalling or aborting.

3. Formal model

This section introduces our formal model. For the first time, it proves that the DATM model is safe and that it accepts any conflict-serializable schedule.

3.1 Intuitions

Informally, one can think of a concurrency control mechanism as an automaton that accepts concurrent schedules of events. We prove two properties of DATM. First, DATM is an automaton that accepts only those schedules of transactions that preserve serializability.

Second, DATM accepts *all* conflict-serializable schedules. Contrast this with other TM systems that accept only a subset of the schedules because they either explicitly use two-phase locking or effectively have the same behavior as two-phase locking. We prove that all schedules are accepted by showing that DATM tracks read/write dependences and aborts transactions only if there are cycles in the underlying serialization graph.

The formal model is adapted from Lynch et al. [20]. The model uses non-deterministic transitions to avoid constraining implementations. For example, when a transaction reads a variable, it can read either that variable's committed value or a value written by an uncommitted transaction.

3.2 Safety

A computation is modeled as a *history*, that is, a sequence of instantaneous *events* of the form:

- $\langle T,\ x.read(v)\ \rangle$: T reads v from variable x.
- $\langle T,\ x.write(v)\ \rangle$: T writes v to variable x.
- $\langle T\ commit\rangle$: T commits
- $\langle T\ abort\rangle$: T aborts

For example, here is a history in which transaction T_A reads 0 from x, writes 1 to y, and commits.

$$\langle T_A,\ x.read(0)\ \rangle \cdot \langle T_A,\ y.write(1)\ \rangle \cdot \langle T_A\ commit\rangle$$

A history is *well-formed* if no transaction both commits and aborts, and if no transaction takes any steps after it commits or aborts. Without loss of generality, values read to and written from variables are unique.

A history is *failure-free* if all transactions commit, and it is *serial* if steps of distinct transactions are not interleaved. A serial failure-free history is *legal* if each value read from a variable is the value most recently written.

A *subhistory* of a history h is a sub-sequence of the events of h. If h is a history and S a set of transactions, $h|S$ is the sub-sequence of events labeled with transactions in S. Two histories, h and h', are *equivalent* if for every transaction T_A, $h|T_A = h'|T_A$. If h is a history, $committed(h)$ is the sub-sequence of h consisting of all events of committed transactions, and $active(h)$ is the set of active (not committed or aborted) transactions. If x is a variable, $prior(x, h)$ is the set of active transactions that have read or written x in h, $writer(x, h)$ is the active or committed transaction that most recently wrote x in h, and $value(x, h)$ is the value written.

Definition 3.1. *A history h is* atomic *if $committed(h)$ is equivalent to a legal failure-free serial history.*

Let $serial(h)$ be the serial history equivalent to $committed(h)$, in which transactions appear in the order of their commit events in h.

A concurrency control mechanism can be thought of as an automaton that accepts concurrent histories. The mechanism is correct if those histories are atomic. We now define dependence-aware transactional memory as an automaton that accepts atomic histories. The automaton keeps the following state. The history h is the history accepted so far. For each T, we keep track of $earlier(T)$, the set of transactions that must *commit* before T can commit. We also keep track of $notLater(T)$, the set of transactions that must *commit* or *abort* before T can commit. Intuitively, $earlier(T)$ tracks the write-read dependence while $notLater(T)$ tracks all the remaining types of dependences. Transitions are given by pre- and post-conditions, where X' denotes the new state of component X.

An active transaction can always abort:
- Pre: $T \in active(h)$
- Post: $h' = h \cdot \langle T\ abort\rangle$

This transition captures the effects of deadlock detection, either exact or inexact (that is, timeouts).

When a transaction T reads, we track the write-read dependence on the transaction whose value it read.
- Pre: $T \in active(h)$
- Post:
  - $h' = h \cdot \langle T,\ x.read(value(x, h))\ \rangle$
  - $earlier'(T) = earlier(T) \cup \{writer(x, h)\}$.

When a transaction writes, we track its read-write and write-write dependences on the active transactions that read or wrote that variable.

- Pre: $T \in active(h)$
- Post:
  - $h' = h' \cdot \langle T,\ x.write(v)\ \rangle$
  - $notLater'(T) = notLater(T) \cup prior(x,h)$.

A transaction can commit only if every value it read was written by a now-committed transaction, and every value overwritten was previously read or written by a now-committed or now-aborted transaction.

- Pre:
  - $T \in active(h)$
  - $earlier(T) \subset committed(h) \cup \{T\}$
  - $notLater(T) \subset committed(h) \cup aborted(h) \cup \{T\}$
- Post: $h' = h \cdot \langle T\ commit\rangle$

The dependence-aware algorithm satisfies this simple invariant: for committed transactions, serialization does not reorder reads and writes to the same variable.

Lemma 3.1. *Let w_0 be a write event by T_0, and e_1 either a read or write event by T_1, both to a variable x in h. If w_0 precedes e_1 in $serial(h)$, then w_0 precedes e_1 in h.*

Proof. Suppose instead that e_1 precedes w_0 in h. Because they were reordered, T_0 and T_1 were both active when w_0 was appended to history h. It follows that $T_1 \in prior(x,h)$, so $T_1 \in notLater(T_0)$, implying that T_1 must commit before T_0, ordering e_1 before w_0 in $serial(h)$, a contradiction. □

Lemma 3.2. *Let h be a history containing w_0, a write event by T_0, r_1, a read event by T_1 that returns the value written by w_0, and w_2, a write event by T_2 that follows w_0 in h.*

We claim that $T_1 \in notLater(T_2)$.

Proof. Because r_1 returns the value written by w_0, there are no writes between w_0 and r_1. Therefore w_2 follows r_1 in h, and the claim is established when w_2 is appended to the history. □

If r is a read event in h, let $readsFrom(r,h)$ be the write event whose value r returns. That is, for every read event r in $serial(h)$,

$$readsFrom(r, serial(h)) = readsFrom(r,h). \quad (1)$$

The property holds vacuously in the original state. Suppose, by way of contradiction, the property becomes violated at some step. That step must be the commit of a transaction T_0, because the other steps leave $serial(h)$ unchanged. Let r_0 be a read step in $h|T_0$ that violates the property, let $w_1 = readsFrom(r,h')$, and let T_1 be the transaction that executed w_1.

First, w_1 must be in $serial(h)$ because $T_1 \in earlier(T_0)$, and a precondition for T_0 to commit is that T_1 be committed. Second, there can be no w_2 by T_2 between w_0 and r_1 in $serial(h)$. If w_2 comes after w_1 in $serial(h)$, then by Lemma 3.1, w_2 comes after w_1 in h, so by Lemma 3.2, $T_1 \in notLater(T_2)$, implying that T_1 could not have committed after T_2.

It follows that when a transaction commits, the new serial history is legal, because every read event returns the value written by the most recent write event in $serial(h')$.

3.3 Accepting all conflict-serializable histories

We have shown that our implementation is *safe*: all histories accepted are atomic. We now focus on a stronger claim, that this automaton accepts *all* conflict-serializable [8] histories. By contrast, most TM systems, whether hardware or software, accept only those histories admitted by two-phase locking, a strictly smaller set.

Some care is needed when interpreting this claim. As noted, an active transaction can be aborted at any time, for any reason. In practice, an implementation will abort a transaction only if it detects, or suspects, a deadlock resulting from a cyclic dependence. Our automaton accepts all conflict-serializable histories, but an actual implementation may reject some as a result of imprecise deadlock detection (for example, premature timeouts). Our implementation also turns off forwarding when it appears to be ineffective, for example, when restarting a transaction aborted by a cyclic dependence.

Any history has an associated *serialization graph*. Each node is labeled with a committed transaction, and there is a directed edge from T_0 to T_1, if first T_0 and then T_1 apply conflicting operations (at least one write) to the same object. A history is conflict-serializable if and only if the associated serialization graph is acyclic [8].

We define a directed *wait-graph* for a history h as $G(h) = (V,E)$ where $V = \{T : T \in active(h)\}$ and $E = \{(T_1,T_2) : T_1 \in earlier(T_2) \cup notLater(T_2) \wedge T_1, T_2 \in V\}$. Note that according to this wait-graph a transaction T commits iff it has no incoming edges.

To prove that the automaton accepts all conflict-serializable histories, we change the abort rule to reflect precise detection. A transaction is aborted if it has read a value from a transaction that later aborted or if it is part of a cycle in the *wait-graph*.

- Pre:
  - $T \in active(h)$
  - $(\exists T_1 \in earlier(T) \wedge T_1 \in aborted(h)) \bigvee (T \in$ cycle in $G(h))$
- Post: $h' = h \cdot \langle T\ abort\rangle$

Observe that according to this new abort rule the first transaction abort will occur because of a cycle in the wait-graph. Subsequent aborts may occur due to the dependences tracked by the set $earlier(T)$ or other cycles in the graph.

Lemma 3.3. *If an input history H is conflict-serializable then $G(H)$ is acyclic.*

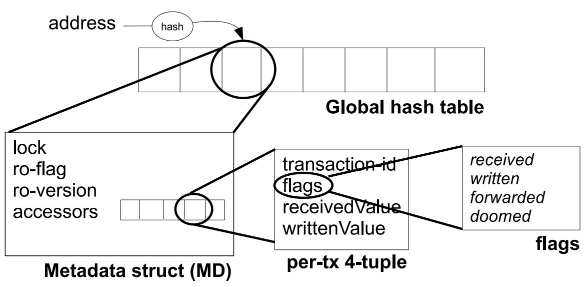

Figure 2. Key data structures in DASTM.

Proof. Let h be the earliest subhistory of H such that there is a cycle in $G(h)$. Let $\langle T_1, T_2, .., T_k\rangle$ denote the cycle. Consider the nodes T_1 and T_k. By definition the edge (T_k, T_1) exists because $T_k \in \mathit{earlier}(T_1) \cup \mathit{notLater}(T_1)$. Using the transitive property, the chain of edges from T_1 to T_k imply that $T_1 \in \mathit{earlier}(T_k) \cup \mathit{notLater}(T_k)$. Using the definitions of $\mathit{earlier}(T)$ and $\mathit{notLater}(T)$, there are 9 possible combinations of dependence between T_1 and T_k. Each of these combinations result in a cycle between T_1 and T_k in the *serializability* graph of h. This is a contradiction, because a cycle in the serializability graph of h implies H is not conflict-serializable. □

Lemma 3.4. *If an input history is conflict-serializable, then it is accepted by the automaton.*

Proof. By Lemma 3.3, the wait-free graph for the input history is acyclic. According to the abort rule, no transaction would have aborted as the first abort is triggered by a cycle. Hence, the given history is accepted as-is by the automaton. □

4. Design

This section introduces our prototype implementation of dependence-aware software transactional memory (DASTM). It presents the key data structures and the basic steps transactions follow. Finally, we discuss some of the more interesting optimizations.

4.1 Data structures

Each thread maintains transaction-specific information in thread-local storage. Each transaction has a read-set and a write-set, implemented as linked lists with bloom filters to reduce list searches (like TL2 [5]). There is a single shared *global-clock* vector Each transaction also has a *wait-vector* to manage dependences.

The primary shared data structure is a global hashtable that contains the system metadata, shown in Figure 2. Active transactions hash memory addresses to look up memory metadata structures (MDs) in the hashtable. Each address requires a unique MD, so hashtable collisions are resolved using a linked list of entries. DASTM uses the same addressing interface as the STAMP TL2 implementation, where load and store addresses are to 4-byte, aligned data units. Each MD contains the following fields.

- *lock*
- *ro-flag*
- *ro-version*
- *accessors*, a sequence of 4-tuples, each comprised of: **[transaction-id, flags, receivedValue, writtenValue]**

For efficiency, all addresses that hash to the same value share the same *lock ro-flag* and *ro-version*. The *lock* is a recursive spinlock that protects access to the MD structure. The *ro-flag* and the *ro-version*enable an optimization for memory locations that are only read during a transaction (see Section 4.3.2).

The core of the MD structure is the *accessors* list, an ordered sequence of 4-tuples. Each tuple has a transaction-id that identifies the transaction accessing this memory location. The flags field contains four bits, Received, Written, Forwarded, and Doomed. The first three bits indicate whether the tuple has received, written, or forwarded a value. The Doomed flag indicates that the transaction accessing the address will have to abort. The receivedValue field holds the memory value retrieved from memory or the forwarded value from another in-progress transaction. The writtenValue field records updates to the memory location made by the transaction.

4.2 Basic transaction execution

The following steps summarize transaction execution. Transactions end either in commit or abort (where aborted transactions restart).

1. **Transaction initialization.** Transactions begin by clearing the thread-local read and write sets. As described below, they obtain a transaction-id and initialize their *wait-vector* to all zeros.
2. **Transactional accesses.** Memory reads and writes add the address to the transaction's thread-local read or write set (respectively). They then look up and create, if necessary, the MD structure corresponding to the memory address in the global hashtable. The lock protecting the MD structure is held for the duration of servicing the memory operation. If this access is the first access to the address by the transaction, a new 4-tuple is appended to the accessors sequence in the MD.

   a **Reads.** If this is the first access to the memory location, the value is read either from an active transaction or from memory, and is then stored in the recieved-Value field. Forwarding happens by having the transaction scan the accessors list backwards from the end for previous tuples. If it finds a tuple that does not have the Doomed flag set and has its Written flag set, the transaction copies the writtenValue, sets the Forwarded flag, and sets the Received flag in the receiving transaction's accessor tuple. If no such tuple exists, then the transaction initializes the receivedValue directly from the memory. The value returned for the read operation is the value in the receivedValue field, or, if the written flag is set, the value in the writtenValue field.

b **Writes.** The Written flag in the MD accessor flags field is turned on and the writtenValue field is updated with the new value being stored. If the memory address is previously read but not written (Written flag is not set), then it turns on the Doomed flag of all tuples that are later in the sequence.

3. **Transaction commit.**

   a **Resolve dependences.** Wait until all dependences are resolved, i.e., all transactions this one depends on $(earlier(T) \cup notLater(T))$ must commit or abort (see Section 4.3.3 for details).

   b **Write-set locking.** Acquire and hold the MD structure locks for all addresses in the write-set. If any write-set tuples for this transaction have the Doomed flag set, release all the locks and abort.

   c **Read-set validation.** Validate the read-set by ensuring that none of the MD accessor tuples for this transaction have the Doomed flag set. The MD structure is locked only for the duration of the check. If the validation fails, the transaction releases all held locks and aborts. The read set does not need to be locked for the duration of commit because any subsequent writer will form a dependence and wait for this transaction to commit.

   d **Write-back.** Write back the value of each element in the write-set to main memory, and release the MD lock. If the Forwarded flag of the transaction's tuple is set, the transaction dooms any dependent transaction that has received a stale value for this memory address. The transaction scans the dependents, dooming any entry that has its Received flag set if the receivedValue is different from the committing transaction's writtenValue. This check terminates at the end of the sequence, or at a non-doomed tuple that does not have the Received flag set.

   The start of write-back is the transaction's linearization point [17]—any transaction that starts write-back will successfully commit, with any contending transaction serializing afterwards.

4. **Transaction abort.** A transaction that aborts must ensure that all transactions dependent on it also abort. For all addresses in the write-set with the Forwarded flag set, the transaction sets the Doomed flag for all subsequent accesses by transactions that have the Received flag set. The transaction stops at the first non-doomed tuple that has the Written flag set and the Received flag clear. Each MD structure is locked only for the time it takes to perform this check.

5. **Cleanup.** Both Commit and Abort complete by removing all tuples that correspond to the transaction's read and write set from the corresponding MD structures. The MD structures themselves (if dynamically allocated) may be freed if the tuple-sequence has become empty.

4.3 Design details

This section describes a few of the important optimizations and design choices made in our prototype.

4.3.1 Resolving dependences

A transaction must wait until all transactions on which it depends complete (commit or abort). After they commit, it can proceed (past step 3a) and continue its attempt to commit. A transaction's dependences are implicitly encoded by its tuple's position in the MD accessors sequence—the transactions of tuples preceding it in the sequence are the ones it may depend on. One strategy for resolving dependences is to iterate through each address in the working set, checking if all preceding transactions in the MD accessor list that have the Written flag set have completed. Recall that when a transaction completes, it removes its tuple from the MD accessor.

We implement a more efficient strategy for resolving dependences that uses vector clocks. The runtime has a *global-clock* (GC) that tracks the number of transactions completed by each processor. Each transaction has a *wait-vector* that is used to summarize the transactions it has to wait on. When a transaction starts on a processor p, it reads GC[p] (the pth entry in the vector clock) and keeps track of V = GC[p]+1. This scalar (V) represents the value which the transaction will write into the *global-clock* when it completes, and is communicated to other processors that wish to take a dependence on this transaction. This communication occurs when a transaction accesses any memory address, it updates its *wait-vector* with the V values of all transactions preceding it in the accessor tuple. The V value is present in each tuple, as the transaction-id field encodes both p and V. When a transaction completes (whether commit or abort), it writes V to GC[p], after dooming its write-set. Dependence resolution is thus reduced to each transaction waiting for *global-clock* to be greater-or-equal to its *wait-vector*.

4.3.2 Optimizing read data

Most transactions read more data than they write. Some addresses are only read during a transaction and never written. For such transactions, the basic algorithm can impose a heavy performance penalty in the steps required to process a read (2a) and to validate the read-set at commit time (3c). For data that is only read during a transaction, we would like to avoid any MD structure locking , vector-clock management, tuple management, and so on.

Our approach initially assumes that all data accessed by a transaction is read-only—as indicated by the *ro-flag* field in the MD. The transaction reads the desired data from main memory, and saves the *ro-version* value in its read-set. The validation phase (3c) for such memory locations consists of ensuring that for each address the *ro-version* has not changed in the MD structure, and that the *ro-flag* still indicates that the address is in read-only mode.

If a transaction stores to a memory location (i.e. uses the MD structure), the *ro-flag* is cleared. Any transaction that previously read the location while the *ro-flag* was set will abort during validation if it sees that the flag has been turned off. With the flag off, reads are processed as in 2a. The runtime can decide to transition an MD back to read-only mode by resetting the *ro-flag* and increasing the *ro-version*. Increasing the *ro-version* ensures that any outstanding transaction that reads the location in read-only mode will abort during validation. The runtime might turn on all *ro-flag*s if there are no active transactions, or might turn them on every N transactions.

4.3.3 Deadlock management

Deadlock can arise in the commit protocol, steps 3a-c, for a variety of reasons. First, cyclical dependences in the DASTM model result in two transactions waiting for each other to commit, and thus both stay in step 3a indefinitely. Second, we do not impose any specific ordering on lock acquires (exacerbated by the fact that we acquire write-set locks before read-set), so transactions can deadlock in steps 3b or 3c. Third, since our implementation uses a single *lock* to protect multiple MD structures that hash to the same bucket, on rare occasions false conflicts can cause deadlocks.

Deadlocks are handled using timeout, similar to TL2. Other approaches are possible, including implementations that avoid deadlocks (e.g. by restricting dependence creation to guarantee acyclic dependences or imposing lock order), or which use more sophisticated deadlock detection techniques like Dreadlocks [13].

4.3.4 Design tradeoffs

Our STM design does not implement every feature of the dependence-aware model. Transaction dependences are always created relative to the most recently written value of the object. With this policy, the dependence graph is always a chain, and new dependences are appended to the end. A given transaction will forward only a single value, even if the address is written multiple times. That single value can be forwarded to multiple transactions. Preliminary data indicated that these optimizations would generate little performance and add complexity.

4.4 Containing zombies

Because W→R dependences forward uncommitted data, a transaction can read invalid or inconsistent data (see Section 2.4). Zombie transactions (those that will never commit) can enter infinite loops, write to incorrect addresses, read from incorrect addresses, jump to incorrect addresses, and fail program assertions. Some STM systems allow zombie transactions and have mechanisms to deal with them [6]. DASTM's control over data forwarding makes containing zombies easy.

With dependence-aware transactions, infinite loops are resolved by runtime support. When transaction A enters an infinite loop (that is not present in the original program), it must have read inconsistent data from a transaction B that will not successfully commit. When transaction B restarts, the runtime restarts transaction A. If B is also in an infinite loop because of a dependence on A, the runtime system periodically polls for circular dependences and restarts both transactions. In Java, the VM can propagate the restart. In C, the runtime system sends a signal.

The runtime buffers data written during a transaction, which prevents zombie transactions from corrupting the program's data structures, and from causing spurious exceptions due to stores to incorrect addresses. Zombie transactions can load from incorrect or invalid addresses, causing incorrect control flow or spurious exceptions. When any transaction that has read forwarded data throws an exception, the transaction is restarted in no-forward mode. Otherwise, it will be restarted when the source of the inconsistent data restarts. A managed runtime can detect exceptions directly, while an unmanaged environment can use signal handlers.

Jumping to a loaded address in a transaction that has read forwarded data causes the runtime to restart the transaction in no-forward mode. Program assertions must be integrated with the STM runtime. Any failed assertion in a transaction that has read forwarded data is restarted in no-forward mode.

5. Evaluation

We conducted the experiments for DASTM on a Sun server using the UltraSparc T1 (Niagara) processor. This processor contains eight multi-threaded cores with four contexts per core, for a total of 32 total hardware contexts. The machine runs the 64-bit Linux 2.6.24-19 operating system. Our Java tests (DASTM-J) are on a machine with 4 quad-core Intel Xeon 2.93GHz processors, for a total of 16 hardware cores. The machine runs Linux kernel version 2.6.22-14. We use counter as a micro-benchmark to study how DASTM performs in the presence of hot-spots. We also report the performance results for three representative STAMP 0.9.8 benchmarks—vacation, labyrinth and ssca2. We compare DASTM with unmodified TL2 [6] on each of these benchmarks. We report TL2 statistics that differ from those reported by Minh et al. [22] because their results are from a simulator, while ours are from real hardware. We use the TL2 code distributed with the STAMP suite. We ensure each benchmark's threads are appropriately pinned to individual processors (using thread affinity) to avoid OS scheduling anomalies. We report the averages of three benchmark runs.

5.1 counter

Writing shared data within a transaction generally leads to hot-spots that result in poor performance of an STM. In

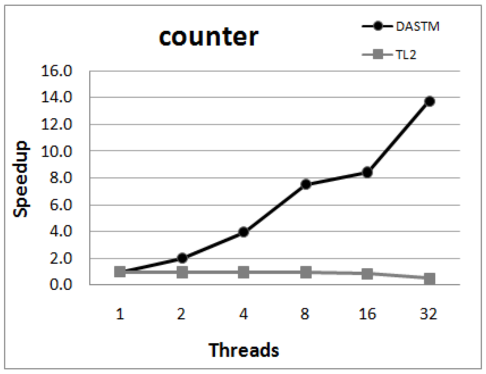

Figure 3. Speedup (higher is better) seen in DASTM and TL2 on the counter benchmark.

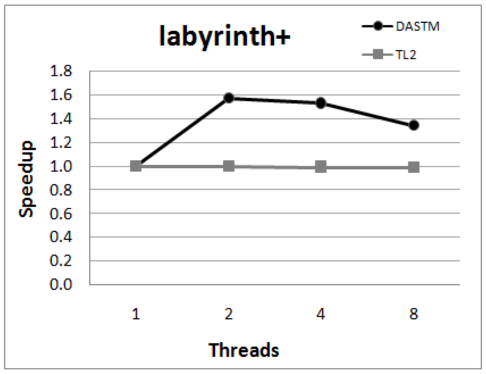

Figure 5. Speedup (higher is better) achieved by DASTM and TL2 compared to the single thread performance of TL2 on labyrinth+, a high contention variant of labyrinth

Figure 3, we study the effect of updating a shared counter for a total of 100,000 times using a variable number of threads. Each increment transaction also contains a fixed amount of think time (5,000 iterations of a local loop), to simulate work on private data. TL2 does not scale at all, revealing the inherent lack of concurrency in two-phase locking systems. This micro-benchmark demonstrates how DASTM, in ideal conditions, can increase concurrency by allowing conflicting transactions to safely commit.

5.2 STAMP

vacation Vacation models a travel reservation system. It uses red-black trees to store data. Client tasks are performed within transactions to provide safe access to this data. Our experiments execute 1,000 transactions and use the parameters "-t 1000 -n 100 -u 50". This particular configuration has very high contention and more than 86% of the benchmark time is spent within transactions. Figure 4 depicts how DASTM compares to TL2. DASTM outperforms TL2 by $4.86\times$ at 16 threads.

We see that vacation performance decreases on DASTM going from 16 to 32 threads. The abort rate doubles when going from 16 to 32 threads with almost all of vacation's aborts due to timeouts waiting for dependences to be satisfied at commit (see Table 1 in Section 5.3 below). We increased the timeout value which decreased the abort rate and improved performance at 32 processor threads, nearly matching 16 thread performance. With more cores, deadlock detection that is more precise than simple timeouts (for example, Dreadlocks [13]) are likely to become important to sustain good performance.

labyrinth The labyrinth benchmark uses Lee's algorithm to find the shortest path between pairs of nodes in a maze [22]. The program reads and updates memory locations within data structures, such as a worklist and a grid. Most of the updates occur in long transactions. Figure 4 shows the results for a maze of size $256 \times 256 \times 5$, using parameters "-i random-x256-y256-z5-n256.txt". The total number of transactions is between 514 to 576 (depending on the number of threads). With the default transaction boundaries, both TL2 and DASTM are able to scale well on this benchmark. We therefore decided to use different transaction boundaries: the benchmark's primary loop creates two transactions per iteration, which we merge into a single transaction. This is another way to transactionalize the benchmark (producing the same results), however contention is much higher. Figure 5 shows the results for this variant, which we call labyrinth+. DASTM is able to improve performance by up to approximately $1.6\times$ with additional cores, whereas TL2 is unable to improve beyond single thread performance. Neither system achieves any additional speedup as the number of hardware threads is increased above 2. Like vacation, aborts due to time out increase with more threads. In addition, overwrite aborts (shown as A_2 in Table 1) also increase with more threads. Experiments with increased timeout thresholds do reduce those aborts by up to 60%, but overwrite aborts remain unchanged and thus become the limiting factor for performance. Even with more sophisticated deadlock detection, labyrinth+ is inherently limited in the amount of concurrency that can automatically be achieved.

ssca2 The ssca2 benchmark uses a scientific computational kernel that operates on a multi-graph to produce an efficient graph structure representation using adjacency (and other auxiliary) arrays [22]. We run the benchmark using the parameters "-s17 -i1.0 -u1.0 -l3 -p3". It creates a large number of transactions: over 1.4 million, which individually are relatively small. The benchmark has very little contention, so it does not benefit much from dependence management. Figure 4 shows that on single-threaded runs, TL2 is roughly 20% faster than DASTM, due to overheads for managing metadata. Overheads for other benchmarks depend on a variety of factors including access to DASTM metadata, transaction contention, and the ratio of transaction computation to data accesses. TL2 achieves a peak speedup of approximately $2.4\times$ at 32 threads, while DASTM's overhead causes its peak speedup to be slightly lower at $2.25\times$.

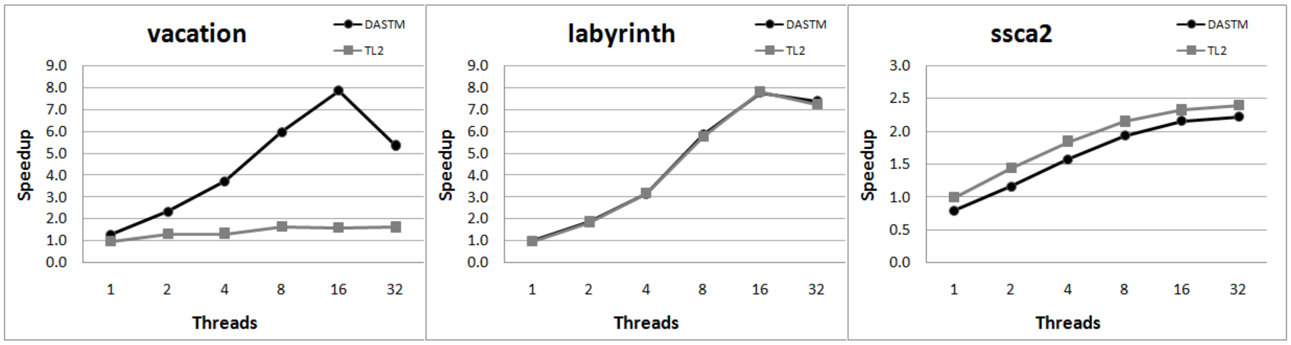

Figure 4. Speedup (higher is better) achieved by DASTM and TL2 compared to the single thread performance of TL2 on (A) vacation (B) labyrinth and (C) ssca2.

Parameter (in %)	**vacation**	**labyrinth+**	**counter**
Reduction in Exec-time	72.5	23.9	86.8
Reduction in Restarts	98.2	90.0	99.5
Abort Rate	44.3	88.8	3.3
A$_1$: Dep. Wait Aborts	79.5	62.6	20.7
A$_2$: Overwrite Aborts	16.2	34.0	79.3
A$_3$: Lock Timeout Aborts	4.2	0.0	0.0
D$_1$: Tx using R→W	3.6	3.8	0.0
D$_2$: Tx using W→W	1.3	76.6	99.6
D$_3$: Tx using W→R	34.0	77.4	99.6

Table 1. DASTM statistics for vacation, labyrinth+ and counter at 8 threads.

5.3 DASTM statistics

Table 1 shows the reduction in execution time and in transactional restarts moving from TL2 to DASTM at 8 threads for the three highest contention benchmarks—vacation, labyrinth+ and counter. We see that on all of them, the number of dynamic aborts is reduced by 90% or more. These findings validate our observation that current STMs abort more transactions than what is strictly necessary to remain safe. DASTM's reduction in aborts translates to increased performance.

Table 1 also gives the percentage of transactions that restart (abort rate) and a breakdown of the various types of aborts and dependences seen in these benchmarks. The *Abort rate* of the vacation benchmark shows that 44.3% of the total transaction attempts resulted in aborts. These aborts occur for three reasons: (1) timeouts while waiting for dependences to be satisfied (A_1), (2) aborts because a transaction overwrote a value that it had already forwarded (A_2), and (3) timeouts while trying to acquire locks on memory locations (A_3). The values of categories A_1–A_3 equals 100% of aborts (nearly 100% for labyrinth+, which has some explicit calls to abort). In a workload such as the counter benchmark aborts are mostly due to forwarding of values that are then overwritten, while labyrinth+ and vacation mostly abort due to timeout while waiting for dependences to resolve.

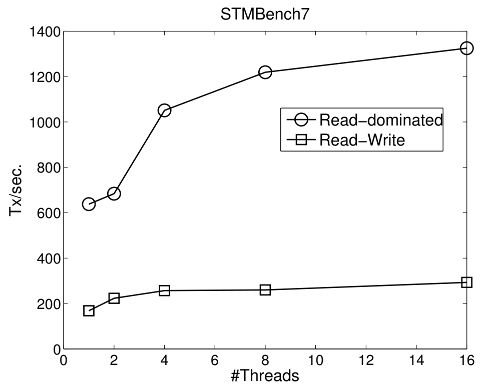

Figure 6. DASTM-J results for STMBench7 read dominated and read-write workload. The benchmark uses the default parameters with long traversals disabled.

D_1, D_2 and D_3 give the number of transactions involved in R→W, W→W and R→W dependences. For example, 99.6% of the transactions in counter read values forwarded by the (W→R) dependences. The numbers in these categories do not total to 100% because a single transaction can have multiple dependence types.

5.4 DASTM-J: an object based STM

We have also implemented DASTM-J, an object based dependence-aware STM. DASTM-J is written in Java and uses the same high level design as presented in Section 4. We present the performance of DASTM-J on STMBench7 to show that our prototype has scalable performance and that dependences are useful for large transactional workloads.

STMBench7 consists of traversals and modifications in a graph with a million objects [9]. The authors of STMBench7 note the difficulty that every public STM has in running their benchmark [7]. In Figure 6, we show the scalable performance of DASTM-J for two different inputs to STMBench7, creating a read dominated and a read-write workload. At

16 threads, DASTM-J achieves more than $1,300$ Tx/sec on the read-dominated workload. The read-dominated workload does not benefit from dependences, but 2.5% of transactions create dependences for the read-write workload. Dependences are more useful as contention increases.

6. Related work

In this section, we discuss the most relevant related work from the literature. There have been many recent advances in STM research [1, 10, 11, 15, 21]. We refer the reader to Larus and Rajwar for a comprehensive reference of transactional memory systems as of summer 2006 [19].

TM isolation Other transactional memory designs and implementations have also observed that modifying the safety conditions for transactions can allow a system to extract more concurrency from workloads. Along with DATM [26, 27], TSTM [2] identifies that using conflict serializability as a correctness criteria, rather than two-phase locking, benefits transactional memory systems by allowing more concurrency. However, their model is based on timestamp ordering, and does not accept every conflict serializable schedule (e.g. those that involve forwarding). The developers of CS-STM [30] (which utilizes a new consistency criterion that the authors call z-linearizable), also consider a variant of that algorithm which maintains full serializability. This variant (called S-STM) is only briefly described as using timestamps and vector clocks and having to maintains partial precedence graphs. The authors state that the runtime overhead of managing their intricate data structures can be prohibitive, especially for smaller transactions, though performance data is not reported.

SI-STM uses snapshot isolation, a weaker isolation level than conflict-serializability [29]. SI-STM shares some of the performance goals of DATM, trying to get conflicting transactions to commit. It also shares some implementation techniques with DATM, namely preserving multiple versions of the same memory byte. Snapshot isolation is more difficult for the programmer to reason about than conflict serializability and is applicable to fewer situations than DATM.

Database systems DATM can be viewed as an efficient implementation of a SGT-based (Serialization Graph Testing-based) certifying concurrency-control scheduler [3]. It is designed to permit *recoverable* schedules, a superset of ACA (Avoid Cascading Aborts) schedules. It does not build up the actual serialization graph, since the dependences on the shared objects provide sufficient information to provide the necessary constraints.

Transaction dependences also have roots in early database research. Spheres of control [4], in the context of a static hierarchy of abstract data types, introduced the notion of dynamic spheres created around actions accessing shared data. Transaction dependences are at one level a refinement and formalization of the general notion of spheres of control in a way that can be implemented in the context of transactional memory.

Time-domain addressing [28] (also called multi-version concurrency control (MVCC)) tracks multiple versions of objects modified concurrently, as does DATM, and thus both systems address similar issues. However, write-shared data are known to degrade MVCC performance, while DATM is designed to scale in their presence. Also, the techniques DATM employs to achieve conflict-serializability (notably, the different types of dependences and the forwarding of uncommitted data) are not found in MVCC systems.

Changing the STM programming model TM systems can change the programming model to increase performance. The programmer must deal with more complexity, but the runtime can be more efficient.

Privatization [33], and its complement (publication) are programming technique that allows programmers to carefully manage when data is shared and accessible by other transactions, versus being private. They bridge the conceptual gap between per-CPU data structures and shared data structures. Early release [32] allows a programmer to drop transactional isolation on given memory locations. The programmer must be correct in his judgment that isolation is not needed on those locations, or the program will no longer be correct. Having code in an escape action access the same data as a paused transaction can cause semantic anomalies [24]. Open nesting [25] trades physical isolation for logical isolation, with the programmer guaranteeing correctness. All of these techniques require more programmer effort than DASTM.

Galois classes [18] and transactional boosting [14] allow the programmer to provide inverse operations for the concurrent data structures. These techniques are orthogonal to dependence-awareness, and can be used to complement them. They have the potential to eliminate structural conflicts in many situations, though programmers may have varying success providing inverses for different data structures (e.g., k-d tree inserts are not straightforward to handle), and defining commutativity relationships between all operations. Similarly, abstract nested transactions (ANTs) [12] attempt to reduce the performance effect of benign conflicts, but require the programmer identify the regions of code that are likely to be victims of such conflicts. The system then ensures that ANTs are re-executed appropriately if they do experience a conflict. ANTs differ from closed nesting in when the re-executing can occur, specifically ANT re-execution can be delayed until the top-level transaction attempts to commit.

7. Conclusion

Dependence-aware transactions allow conflicting transactions to safely commit. This paper presents a formal model of DATM, proves its key properties, and presents the design of the first STM implementation to use the model.

Experimental results from our prototypes (in C and Java) confirm the potential performance benefits of dependence-aware transactional memory as compared to traditional STM implementations.

8. Acknowledgements

We extend thanks to Donald E. Porter and Christopher J. Rossbach for careful reading of drafts and suggestions. This research is supported by NSF CISE Research Infrastructure Grant EIA-0303609, NSF Career Award 0644205, and the DARPA computer science study group. Sun Microsystems provided the T2000 machine via the OpenSPARC program. Intel equipment grant 48395 provided the 4 quad-core Intel server machine.

References

[1] Ali-Reza Adl-Tabatabai, Brian Lewis, Vijay Menon, Brian Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, Jun 2006.

[2] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT*, Feb 2008.

[3] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[4] Charles T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2), 1978.

[5] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, Sep 2006.

[6] Dave Dice and Nir Shavit. What really makes transactions faster? In *TRANSACT*, Jun 2006.

[7] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Dividing Transactional Memories by Zero. In *TRANSACT*, Feb 2008.

[8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In *EuroSys*, Mar 2007.

[10] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, Oct 2003.

[11] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI*, Jun 2006.

[12] Tim Harris and Srdan Stipic. Abstract nested transactions. In *TRANSACT*, Aug 2007.

[13] Maurice Herlihy and Eric Koskinen. Dreadlocks: Efficient deadlock detection for stm. In *TRANSACT*, Feb 2008.

[14] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, Feb 2008.

[15] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, Oct 2006.

[16] Maurice Herlihy and J. Eliot Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[17] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, Jul 1990.

[18] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, Jun 2007.

[19] Jim Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[20] Nancy A. Lynch, Michael Merritt, William E. Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1993.

[21] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, Jun 2006.

[22] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, Sep 2008.

[23] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, Jun 2007.

[24] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS*, Oct 2006.

[25] J. Eliot Moss and Antony L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, Oct 2005.

[26] Hany E. Ramadan, Christopher J. Rossbach, Owen Hofmann, and Emmett Witchel. Dependence-aware transactional memory. Technical Report TR-07-58, University of Texas at Austin, Computer Sciences Department, 2007.

[27] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactions for increased concurrency. In *MICRO*, Nov 2008.

[28] David P. Reed. Implementing atomic actions on decentralized data. *ACM TOCS*, 1(1), 1981.

[29] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *TRANSACT*, Jun 2006.

[30] Torvald Riegel, Heiko Sturzrehm, Pascal Felber, and Christof Fetzer. From causal to z-linearizable transactional memory. Technical Report RR-I-07-02.1, Universite de Neuchatel, Institut d'Informatique, February 2007.

[31] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, Aug 1995.

[32] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*, Jun 2006.

[33] Michael Spear, Virendra Marathe, Luke Dalessandro, and Michael Scott. Privatization techniques for software transactional memory. In *PODC*, Aug 2007.