Dependence-Aware Transactional Memory

Hany E. Ramadan, Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel
Department of Computer Sciences, University of Texas at Austin
{ramadan,rossbach,osh,witchel}@cs.utexas.edu

ABSTRACT

Transactional memory is a promising programming model to enable high performance programs with reasonable programmer effort on the parallel architectures favored by modern processor manufacturers. This paper introduces dependence-aware transactions, a new method for maintaining the conflict serializability safety property of memory transactions while allowing significant freedom for an implementation's version management and conflict detection.

The paper evaluates two implementations of dependence-aware transactional memory: one all-software implementation and one mostly-hardware implementation. The results of micro-benchmarks indicate the promise of dependence-aware transactions to increase the performance of a transactional memory system, with no change in the transactional programming model.

1. INTRODUCTION

Transactions are a promising programming model to deal with the parallel architectures favored by modern processor manufacturers. Power and physical limitations are preventing manufacturers from scaling the performance of individual processor cores, and forcing them to put more cores on a die. Multicore hardware creates a challenge for software performance to scale with additional cores, and transactions are proving to be a powerful model for parallel programming.

Transactional memory brings transactions out of the durable world of databases and into the volatile world of program synchronization. Memory transactions provide atomicity, which assures the programmer that if a transaction fails for any reason the system will revert to the pre-transaction state. Memory transactions also provide isolation (also called serializability) which provides a consistent, global order of all transactions. Strong isolation provides a consistent order for transactions and non-transactional operations.

Most transactional memory system provide *conflict serializability*, which order transactions according to their conflicting operations. A memory write from one transaction conflicts with a read or write from another transaction to the same memory. The most common way in current systems to order conflicting memory accesses is to restart one of the conflicting transactions.

To implement conflict serializability, transactional memory systems provide version management and conflict detection. Current version management and conflict detection schemes can be classified as *eager* or *lazy* [18]. Eager version management updates memory in place, using an auxiliary data structure to save previous values. Lazy version management updates memory in an auxiliary data structure and makes updates visible on transaction commit. Eager conflict detection checks every write to make sure one transaction does not write memory read or written by another transaction, and restarts one of the transactions if there is a conflict. Coherence hardware naturally enforces eager conflict detection. Lazy conflict detection verifies a lack of conflict at transaction commit time, and is common in software transactional memory systems.

This paper introduces dependence tracking as a new method for providing serializable transactional memory. Tracking explicit dependences among transactions allows version management and conflict detection that is neither eager nor lazy, but combines the best of both to minimize transaction restarts. Dependence tracking provides greater concurrency for transactions without change to the transactional programming model. The cost is some system complexity to track dependences, restricting commit order to respect dependences, the possibility that a transaction can read inconsistent data, the possibility of transaction restart because forwarded data is updated, and the need to detect circular dependences. We believe that this system complexity is better than the programmer complexity that comes with open nesting and other performance-oriented changes to the transactional programming model.

For transactions to deliver the promise of the simplicity of coarse-grained locking while providing the high performance of fine-grained locking, a programmer should be able to use the most straightforward algorithms and data structures for his task. A plethora of techniques have already emerged (e.g., privatization, early release) which increase performance in transactional code at the cost of a more complicated programming model. Dependence-aware transactions attempt to bring transactions back to their original promise: good system performance for programmers who write simple linked list code without worrying about early release, or who add a shared counter to all his transactions without worrying about making it CPU-local. Our philosophy is to push the basic transactional programming model towards increased performance without burdening the programmer with additional concepts and techniques.

With dependence tracking, transaction T_0 can write a piece of memory and T_1 can read it, receiving the value written by T_0. Data forwarding violates eager conflict detection, but maintains conflict serializability so long as T_0 commits before T_1 and T_0 does not write the same memory with a different value. With lazy conflict detection, if T_0 tries to commit first it would be allowed to commit, causing T_1 to retry when it tries to commit. With dependence tracking, T_0 can commit, and then T_1 can commit. If T_1 tries to commit first, the system detects the dependence on T_0 and delays T_1's commit until T_0 either commits or retries. Dependence tracking allows concurrent execution of transactions, and then tries to preserve the work done by all transactions by ordering dependences rather than wasting the concurrency by causing a transaction restart.

We present a full example in Section 2, but explicit dependence tracking among transactions provides serializability while providing better performance than either an eager or lazy system. Dependence tracking is applicable to any transactional memory system, and this paper provides data about prototype implementations of both a software and a hardware transactional memory system that use dependence tracking.

The paper starts with a detailed example in Section 2, which leads to a discussion of the dependence-aware model in Section 3. Then we describe the design of a software transactional memory

```
1   first = NULL;
2   lprev = NULL;
3   xbegin;
4   for(lptr = lhead; lptr;
5         lptr = lptr->next) {
6     if(lptr->val == target){
7       // Already head?, break
8       if(lprev == NULL) break;
9       // Move cell to head
10      lprev->next = lptr->next;
11      lptr->next = lhead;
12      lhead = lptr;
13      break;
14    }
15    lprev = lptr;
16  }
17  first = lhead;
18  xend;
19  return first;
```

Figure 1: Code that uses transactions to search a linked list and move the target entry to the front of the list. The head of the list is returned.

system (Section 4) and a hardware transactional memory system (Section 5) based on dependence tracking. Section 6 presents performance numbers for the prototype implementations and compares them to existing eager and lazy systems. Section 7 discusses related work and Section 8 concludes.

2. EXAMPLE

This section presents an extended code example to introduce many of the concepts and terminology of dependence-aware transactions. We will assume a generic transactional memory system, with a granularity equal to a single variable. The purpose of these examples is to build intuition and terminology for the model described in the next section.

2.1 Linked list pattern

Figure 1 shows code that starts a transaction to search a linked list and move the target (if found) to the start of the list (to cut down search time assuming temporal locality in accessing the list). The code returns a pointer to the head of the list. To analyze this code, assume two different threads on two different processors (P_0 and P_1) execute this code in two different transactions (T_0 and T_1) such that the executions overlap in time. To keep things a bit simpler, assume that T_0 searches for a target value that is not found—T_0 reads through the list searching for an element without finding it. T_1 searches for an element, finds it, and moves the element to the front of list. Any serialized schedule of the transactions will enforce the standard linked list invariants, in this case, the number of entries in the list never changes, all entries are accessible, and the list is NULL terminated.

We use standard notation for data dependences, e.g., W→R means a memory cell was written by one transaction and then the same cell was read by a different transaction. We use the generic term “memory cell” to indicate that the discussion applies to many different kinds of systems, e.g., hardware systems where the cell is usually a cache line and software systems where the cell is usually a language-level object. We subscript dependences with transaction numbers where that helps clarify the situation. We analyze several interesting interleaved executions.

2.1.1 Linked list dependences

```
T1
// Move cell to head
lprev->next = lptr->next;
                                        T0
                                        // Reads list
                    // including lprev->next from T1
lptr->next = lhead;
lhead = lptr;
break;
```

Figure 2: An interleaving of the linked list search that produces a cyclic dependence.

```
T1
// Move cell to head
lptr->next = lhead;
                                        T0
                                        // Reads list
                     // including lptr->next from T1
                            // and never terminates
lprev->next = lptr->next;
lhead = lptr;
break;
```

Figure 3: An interleaving of the linked list search that produces a cyclic dependence and causes T_0 to enter an infinite loop.

R_0 →W_1. Assume that the reading transaction executes first, but has not yet committed. Then T_1 executes, finds its target entry, and swaps it to the head of the list. T_0 will have read `lprev->next`, `lptr->next` and `lhead` which are all pointers written by T_1. As reflected by the dependency, T_0 must commit first, because it will return the value `lhead` held before either transaction executed. T_1 can then commit and return the pointer to the new `lhead` element.

W_1 →R_0. Assume that the writing transaction executes first, finds its target entry and swaps it to the head of the list. Then T_0 scans the list. The same pointers as above will have been written by T_1, and their values will be forwarded to T_0. In this case T_1 must commit first because T_0 will return the `lhead` value set by T_1.

W_1 →R_0 and R_0 →W_1. As shown in Figure 2, assume that T_1 writes `lprev->next` on line 10. Then T_0 reads the entire list, including the value of `lprev->next` forwarded from T_1, skipping the element pointed to by `lptr` because that element is temporarily off the list. The W_1 →R_0 dependence from `lprev->next` constrains T_0 from committing until T_1 commits. In order to complete, T_1 must write `lptr->next`, which does not cause a dependence because T_0 did not read that pointer. However, T_1 does write `lhead`, which T_0 read. The combination of W_1 →R_0 and R_0 →W_1 causes a cycle in the transaction dependence graph. The transactions will wait for each other to commit forever, so the system breaks the deadlock by restarting one of the transactions. If T_1 is restarted, then T_0 must restart, because it read data generated by T_1.

W_1 →R_0 and R_0 →W_1. As shown in Figure 3, assume that lines 10 and 11 are switched, so `lptr->next` is assigned `lhead` before the entry is unlinked from the list. T_0 reads the entire list after the assignment. If T_0 follows the `lptr->next` pointer after it has been updated to point to `lhead`, its search will never terminate. However, reading that pointer creates a W_1 →R_0 that constrains T_0 to commit after T_1 anyway. T_1 will eventually write `lhead` which T_0 read, causing a cycle and a transaction restart.

This example should provide a basis for understanding the mechanics of dependence-aware transactional memory, and an intu-

```
    // a == 5 and b == 10
T0
xbegin;                 T1
if(a==5) {              xbegin;
                        a = 6;
                        b = 12;
                        xend;
   b = 10;
}
xend;
```

Figure 4: An interleaving of code that demonstrates the need to order R→W dependences.

Dependence	Forward	Ordered	Restart
$W_0 \leftrightarrow W_1$	No	No	–
$R_0 \rightarrow W_1$	No	Yes	If in cycle
$W_0 \rightarrow R_1$	Yes	Yes	If in cycle, and T_1 must if either: **a)** T_0 does. **b)** T_0 overwrites forwarded data with new value.

Table 1: Summary of dependence types and their properties.

ition for its ability to provide serializability. However a formal argument by which dependence-aware transactional memory can achieve serializability is the subject of Section 3.

2.1.2 Ordering R→W

The need to order R→W dependences is best explained with an example. Figure 4 shows an interleaving (time flowing down) of two transactions that maintain the invariant: $b = 2a$. For the interleaving shown, there is an $R_0 \rightarrow W_1$ dependence on a. If T_1 is allowed to commit, then the dependence goes away, T_0 can commit and the values for a and b are inconsistent (6 and 10). When T_1 is delayed for T_0 to commit first, there is a $W_0 \leftrightarrow W_1$ dependence on b that does not prevent either transaction from committing.

2.2 Best of eager and lazy

One thing the example should make clear is that dependence-aware transactions are not eager or lazy when it comes to version management or conflict detection. Version management can be mostly lazy or mostly eager, depending on the needs of an implementation. The forwarding of data values has an eager flavor, because the the most recent value of the memory cell is forwarded.

Conflict detection is replaced by dependence tracking, though the dependence creation has an eager bent. For dependences to do the most good in avoiding transactional restarts, they should be discovered before a transaction attempts to commit, but this is not required by the model. The constraints on commit order imposed by dependences have a lazy flavor, though most lazy version management systems have a first-to-commit arbitration policy.

3. DEPENDENCE-AWARE MODEL

This section introduces the theoretical model of dependence-aware transactional memory. We present the model in full generality—any particular implementation of the model can make simplifications that are appropriate for the implementation technology (e.g., software or hardware).

3.1 Dependences

Dependences arise between two transactions if at least one of them is writing to a value that they have both accessed. In other words, dependences only exist in the same cases where conflict serializability would have detected a conflict.

3.1.1 Dependence types

Table 1 shows a summary of the dependence types and their properties. We use the dependence notation W→R meaning a read after write (RAW) dependence—one thread read a memory cell that was written by another thread. Dependences are subscripted with transaction numbers, so $W_0 \rightarrow R_1$ means a write from transaction T_0 was read by transaction T_1. W↔W dependences have a double arrow because they are symmetric and never directional. The dependences are listed in order of increasing restrictions, with W→R dependences imposing more restrictions than W↔W dependences.

The system tracks all dependences at the level of memory cells (which are implementation=dependent), and are tracked as they arise during transaction execution. If a dependence has a "Yes" in the **Forward** column in Table 1, then the system forwards the data in the memory cell when the dependence is created. For example, a $W_0 \rightarrow R_1$ dependence requires the value of the memory cell be forwarded from T_0 to T_1. The system records that the cell has been forwarded.

Commit ordering restrictions are noted in the **Ordered** column of Table 1. $W_0 \rightarrow R_1$ and $R_0 \rightarrow W_1$ dependences always constrain commit order, while $W_0 \leftrightarrow W_1$ do not. The W→R dependence is a producer/consumer relationship, which implies ordering (producer before consumer). The need to order R→W dependences is explained in Section 2.1.2. W↔W dependences do not restrict commit order because the dependence is non-directional. A non-directional dependence does not participate in the computation of cyclic dependences (Section 3.2.3).

Finally, the conditions by which a dependence necessitates a restart is also noted in the same Table. Cyclic dependences require restarts, as explained fully in Section 3.2.3, and only R→W and W→R dependences participate in cycles. W→R dependences can require additional restarts. For a $W_0 \rightarrow R_1$ dependence we call T_0 the *source* transaction and T_1 the *destination*. The destination transaction must restart if the source restarts, because it has read data forwarded by the source. The destination transaction must also restart if the source overwrites the data it forwarded to the destination with a new value. The forwarded data value is now stale, invalidating the destination transaction.

3.1.2 Dependence discussion

Dependences are created per memory cell on first access to the cell. Subsequent accesses to the same object do not affect dependence structure, with the single exception, as noted in Table 1 that an overwrite of a forwarded cell does cause a restart. For example, if T_0 writes a cell that T_1 then writes, and then T_1 reads the object, the resultant dependence is simply $W_0 \leftrightarrow W_1$.

Dependences do not survive transaction restarts. A restart eliminates all dependences on the restarting transaction. Of course the restarted transaction may create new dependences. Dependences do not survive transaction commit.

Note that the actions and constraints noted in the table are minimal. A particular implementation can add constraints (but cannot relax them). For instance, a particular implementation may restart a destination transaction if forwarded data is overwritten with the same value. An implementation may detect an overwrite eagerly or lazily. Another example would be relaxing the restart rule, e.g., an implementation can restart a secondary transaction for a R→W dependence when the source transaction restarts.

One attractive property of dependence-aware transactions is that they are not all-or-nothing, they can exist with restart-based techniques for ensuring conflict serializability. For instance, a system can track R→W dependences, but simply restart one of the transactions involved in W→R dependences.

3.2 Multiple dependences

The discussion so far has focused on a single dependence between two transactions. In any implementation of the model, however, multiple dependences will be created, and bring with them their own set of issues.

3.2.1 Multiple dependence between two transactions

Multiple dependences may arise if two transactions conflict on more than one memory cell. Conceptually, each memory cell on which two transactions conflict will lead to a separate dependence, which may or may not be of the same type or in the same direction. To manage multiple dependences between two transactions, the model has the following rule.

Restrictive dependence rule: The relationship between two transactions, in each direction, is governed by the most restrictive dependence going in that direction.

If a transaction is the source for a R→W dependence, and later on it writes and forwards a different memory cell to the same destination transaction (thereby creating a W→R dependence), the transactions are constrained by the more restrictive W→R dependence. Both dependences are of course still tracked by the system.

3.2.2 Multiple transactions

More than two transactions can concurrently access the same memory cell. The first two will create a dependence as described above. The third transaction will create its dependence with the last writer of the memory cell (with the system making an arbitrary choice for W↔W dependences). Every dependence involves at least one writer.

Because dependences are binary, a single memory cell may have a tree of dependences, the depth of which is one level per writing transaction. Readers will attach in the tree to the latest writer (i.e. lowest in the tree). If all transactions are writing, the tree is simply a chain. Actions and ordering constraints are pairwise and independent from the larger structure of the dependence structure. The overall structure of dependences can create cyclic dependences, which is the subject of the next section.

3.2.3 Cyclic Dependences

Dependences can restrict commit order and dependences can form a cycle. If the system simply waits for dependences to resolve, then some cyclic dependence chains will cause deadlock. The system must prevent or resolve these deadlocks.

While dependences arise from reads and writes of memory cells, deadlock occurs from a cycle in the transaction dependence graph. The transaction dependence graph contains a directed link between two transactions if there is a directed dependence between them on any memory cell. Note that W↔W dependences are not directional, so they do not participate in the dependence graph (conceptually, write dependences do not create an ordering the way R→W and W→R dependences do). A cycle in the transaction dependence graph can be of any length up to the total number of active transactions in the system.

Cyclic dependences cause transactional restarts in dependence-aware TM, so the system should detect them accurately and resolve them efficiently. An implementation has wide latitude to deal with cyclic dependences. Cycle avoidance is usually difficult to make efficient, so we expect most implementations to resolve cyclic dependences by restarting a transaction within the dependence chain (restarting one transaction is all that is needed to break the chain). Indeed, computing the dependence chain itself might be onerous, so using a simple timeout will avoid deadlock. Implementations may add restrictions to the model to minimize the occurrence of deadlocks. For example, imposing restrictions on the size of any memory cell's dependence tree (chain), may help reduce the occurrence of cycles, although perhaps at a performance cost. If the system computes the dependence chain (or part of it), it can pick the victim transaction based on its position in the dependence graph (i.e., it is probably advantageous to restart a transaction that has not forwarded any data), transaction size, age, etc. Dependence-aware transactions do not have a traditional "contention manager", but the cycle-resolution strategy plays a similar role in these systems.

3.3 Stale data and exceptions

Because the model forwards data between transactions, it is possible that a transaction can read invalid data. For instance, Figure 3 depicts an interleaving where one transaction reads a forwarded pointer value causing it to infinite loop. The loop is benign, because it serves to delay the transaction which is what the system would do if the transaction proceeded. The loop will be terminated by the system once the producing transactions completes. On reading inconsistent data the destination transaction can throw an exception that would not occur in a serial transactional schedule.

There are two ways to deal with non-serializable exceptions, delay them or restart the transaction. A transaction that raises an exception can wait for the dependence involving that memory location (or for all its dependences) to resolve, before raising an exception. This guarantees that all memory read by the transaction has indeed been committed by previous transactions, so the exception can by ordered after the committed transactions (and thus any exception is a "real" exception, not an artifact of the concurrency in the system).

Another approach would be to attempt to immediately restart the transaction when encountering an exception. Because programmers naturally do not generally leave data structures in inconsistent states for long, it is unlikely for a transaction to read the same inconsistent data after a restart. The choice of whether to delay exceptions or cause a restart depends on the details of a particular implementation. Hybrids are possible that allow some number of restarts before waiting for dependences to resolve, or that wait for dependences for a limited time duration.

3.4 Model safety

This section demonstrates how the presented model guarantees conflict serializability. Transaction processing theory defines two operations as *in conflict* if they access the same memory location and at least one of them is a write. In the *conflict relation* of a schedule, $conf(S)$, $p < q$ if operations p and q are in conflict, and p occurs before q in S. A schedule S is *conflict serializable* if it has the same set of operations and the same conflict relation as a serial schedule S' [24].

To efficiently test for conflict serializability, the set of conflicts between committed transactions is abstracted as the *conflict graph* for a schedule S, or $G(S)$. Each node in the conflict graph is a committed transaction t. $G(S)$ contains an edge from transaction t to t' if there are operations $p \in t$ and $q \in t'$ and $\langle p, q \rangle \in conf(S)$ [24]. In our model, dependences between active transactions represent potential edges in the conflict graph.

A schedule S is conflict serializable if and only if $G(S)$ is acyclic.

To ensure that $G(S)$ is acyclic, a transaction processing system may enforce a more restrictive *commitment ordering* rule, that transactions commit in conflict order [24]. In order to account for the effects of aborted transactions, TM systems must also enforce *recoverability*, the property that a transaction may commit only after transactions from which it has read data have committed [24].

Our model differs from traditional TM systems in how the acyclic property of $G(S)$ and recoverability is enforced. Eager and lazy TM systems enforce commitment ordering by disallowing conflicts between active transactions, either at commit time or when the conflicting operation takes place. Transactions do not read updates of other active transactions. A transaction that commits will only have conflicts with transactions that have already committed, and all of these conflicts will represent incoming edges in the conflict graph. Thus transactions always commit in conflict order, and a transaction that commits will only have read updates from already committed transactions.

Our model allows conflicts between active transactions. To enforce commitment ordering, a transaction will not commit until all transactions on which it depends have committed, i.e. there are no incoming edges in the conflict graph except for those from committed transactions. Cycles prevent transactions from committing in conflict order, and so require at least one transaction to be restarted. Recoverability is ensured by aborting readers in W→R dependences when the writer aborts.

In our model W→W dependences are not tracked. We assume that writes are speculatively buffered, so W→W dependences may be inserted in transaction commit order. Our model enforces that all other dependences occur in transaction commit order by ordering transaction commits, thus a W→W dependence will not form a cycle.

4. DEPENDENCE-AWARE STM

This section describes the implementation of *DASTM*, a software-based transactional memory system designed to use transaction dependences. *DASTM* is a C++ object-based STM library. *DASTM* provides an entirely new implementation of the Rochester STM [10] API, a well-known, publicly available STM, allowing us to leverage the same benchmarks across both systems. We present a brief description of the API, followed by discussion of version management, object access, transaction completion, cycle-handling, how dependences are observed in *DASTM*and some simplifications made to ease implementation of *DASTM*.

4.1 Public Object Model

The public object model of RSTM v2 is retained with only minor modifications for *DASTM*. The main public classes are *Object*, which is the superclass of all transactional classes, and *SharedObject*, used to declare and use transactional object instances.

The primary change to the *Object* class is the addition of a virtual `equals` method, to check equality of object instances, which *DASTM* uses to verify that a forwarded object is indeed valid (as described below). A default implementation of `equals` performs a byte-wise comparison of the object data, although transactional classes can provide custom implementations. Managed environments, such as .NET and Java, already provide virtual `equals` method at the top of their class hierarchy.

The primary change to the *SharedObject* class is the addition of a open_WO method which complements the existing open_RO and open_RW methods. These methods are used to access instances of a shared object within a transaction. The open_WO method allows transactions which do not intend to read a value to benefit from the less restrictive W↔W dependences. While this method is not used

```
class ObjectWrapper {
  bool    Committed_Flag
  bool    Aborted_Flag
  bool    Snapshot_OK_Flag
  int     Readers_Count
  Object* Active_or_Committed_Object
  Object* Forwarded_Snapshot_Object
  SharedObject*  Owner_Shared_Object
  ObjectWrapper* Next_Node
}
```

Figure 5: State in an ObjectWrapper node

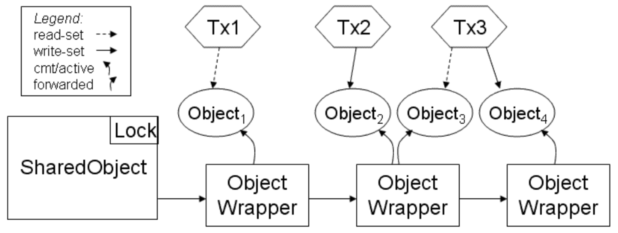

Figure 6: Sample DASTM execution showing a single shared object. Tx1 is reading the committed value of the object (Obj1), Tx2 is actively writing a new value (Obj2), and Tx3 has issued a read (serviced by the forwarded Obj3) and is writing a new value (Obj4). The transactions must commit in the order Tx1, Tx2, Tx3. If Tx2 restarts, Tx3 will restart due to the forward property of its dependence on Tx2. If Obj3 is not equal to Obj2 when Tx2 commits, then Tx3 must abort as well. Any new transactions that read would read a forwarded snapshot created by cloning Obj4, attached to the most recent ObjectWrapper.

in existing benchmarks, we include it for future use.

4.2 Version management

In *DASTM*multiple transactions may concurrently access (read or write) the same shared object. Dependences allow the system to preserve consistency in the presence of multiple active copies of the object. Transactions may thus access an object modified by another transaction which has yet to commit.

DASTM manages dependences by associating a set of objects with each shared object (instance of SharedObject), maintaining at least one committed instance in addition to one instance for each active transaction which has opened that object for writing, as seen in Figure 6. The different instances of the shared object are linked together in a list. This restricts possible re-orderings but simplifies implementation. Each shared-object holds a pointer to the head of the list. The list is modified primarily when the object is initially opened by a transaction, and when transactions complete (commit/abort). To protect the list data structure from itself being corrupted by concurrent access a single lock is associated with each shared-object. The lock is held *only for the duration* of the list-modification operations - reading and writing to objects themselves does not lock the shared-object list. Because list modifications are more complex than simple insertion and deletion of nodes (i.e. searching for specific nodes, cleaning up garbage nodes), it was not deemed worth the extra complexity of trying to do away with the lock; for large enough transactions we do not expect this to limit overall concurrency.

Each node in the linked list is an ObjectWrapperrather than being the actual object instances. There are two key reasons for this. First, *DASTM* maintains for each node up to *two* object instances associated with that node (a "live" instance and a "forwarded" snapshot instance, described below). Second, various kinds of state associated with the ObjectWrapper must be maintained. Such state is modified, read, and used for notification (see below for polling) by all the transactions accessing the shared-object. By keeping this state separate from the actual object, we minimize interference with other transactions using the object instances.

Figure 5 shows the fields contained in each ObjectWrapper node. Figure 6 shows a snapshot of an execution of *DASTM*, showing the runtime relationships between the various objects. The following sections elaborate how such runtime state is created and maintained.

4.3 Accessing Objects

In *DASTM* a transaction may access a shared-object in read-only, read-write or write-only mode, with the mode specified by the programmer. The object may previously have been opened (in the same or a different mode) or it may never have been opened before. This section outlines the operation of the system in each of these scenarios.

4.3.1 Previously accessed object

If the object has been previously opened in an appropriate mode in the transaction, the existing reference to that object is simply returned. The Transaction c;ass tracks the read and write sets, which consist of pointers to the ObjectWrappers of the objects accessed during the transaction. These sets are inspected on an object-open request. An object previously opened for RW mode will be returned if subsequently opened in RO mode. An object previously opened in RO mode will be "upgraded" if subsequently accessed in RW mode (see below). Transaction objects are stack-allocated—they contain only a status field and the read and write sets.

4.3.2 Initial read-only access

If the object was not previously opened as part of the same transaction then the shared-object is locked (using the mutex). For RO access, the list of ObjectWrappers is traversed to find the last non-aborted ObjectWrapper(which is either active or committed). If the ObjectWrapper is committed (indicated by the Committed_Flag field), a reference to the committed object is returned (held in the Active_or_Committed_Object field).

If the ObjectWrapper is still active, this indicates an RO mode access to a value that was (and still perhaps is) being written by an active transaction. This "forwarded" value will be stored in the Forwarded_Snapshot_Object field of the ObjectWrapper. If a snapshot was previously made (the field is non-NULL) by a previous transaction, it will be used - sharing the snapshot is possible since all the transactions are simply reading it. If no previous snapshot exists, a snapshot will be made by use the clone() method to make a copy of the active object (held in the Active_or_Committed_Object field). It is important to note that the object may be concurrently accessed by the active transaction while the clone is occurring; the clone implementation must handle such situations to produce a consistent copy. Managed environments (Java, C#), can ameliorate this issue, as the VM may be able to provide a consistent clone automatically. Default implementations of clone in both environments are also possible (e.g. byte-copying the object state), and would suffice for simple objects.

Finally, the Readers_Count field is incremented, the ObjectWrapper is added to the Transaction's read-set, and the shared-object is unlocked. The actual object (either the Committed or Snapshot) is returned as the result of the RO access request.

4.3.3 Initial read-write or write-only access

When a request is made for read-write access to an object not previously opened, the procedure is a superset of the steps for an initial read-only request (see section 4.3.2), including locking the shared-object, finding the last non-aborted ObjectWrapper, determining which copy to read (committed or snapshot of active), incrementing the reader count, and adding the ObjectWrapper to the read-set. For a read-write request a few more steps occur before unlocking the shared-object, and a different object is returned as the result of the access request.

The main difference is that a new ObjectWrapper node is allocated, and inserted at the tail end of the shared-object's list. All the flags are initially false (indicating an Active wrapper), the readers-count is 0, and the Snapshot pointer is NULL. The Active_or_Committed_Object however, is initialized with a pointer to a new object, created by cloning the object obtained in the RO access request (whether it is a committed or a snapshot object). The new ObjectWrapper is inserted into the Transaction's write-set, and the Active object of this newly allocated node is the returned result of the RW access request.

Access requests for write-only mode perform the same steps as read-write access, without the initial work reading an existing ObjectWrapper.

4.4 Dependence management

The W$\leftrightarrow$W dependence is strengthened by enforcing ordering. Commit processing waits for each written object until all previously written objects (ObjectWrappers earlier on the SharedObject's list) are non-active before allowing the transaction to commit.

The R$\rightarrow$W dependence is maintained during the validation phase, by ensuring that for each object in the write-set, there are no readers for any previous ObjectWrappers.

The W$\rightarrow$R dependence is maintained during the validation phase, by ensuring that for each object in the read-set, the version read must have committed, and if the value read was forwarded, it is identical to the later committed value.

Cycles are detected using a timeout mechanism in the commit validation phase, when waiting for objects in the read-set to be complete or for the write-set to be ready for commit. The implementation simply retries these operations a fixed number of times: if the object is still not in the desired state, a cycle is assumed and the currently waiting transaction restarts. This approach is simple, and requires no cross-processor or cross-transaction communication. More sophisticated approaches, including using timestamp information, or deadlock avoidance are also possible, but are left for future work.

4.5 Transaction Completion

Transactions may abort for three reasons (excluding explicit calls to Abort, exceptions, etc.): first if a cycle in the dependence graph arises, second if a source transaction of a W$\rightarrow$R dependence aborts (requiring the destination to abort as well), and third if a forwarded value in W$\rightarrow$R dependence is subsequently over-written by a different value. *DASTM* detects these situations at *commit time*. This simplifies the implementation and respects the limitations of our unmanaged STM environment (e.g. reads and writes to the object in an active transaction are not intercepted). More sophisticated STM environments (e.g. managed environments, or those with APIs to read/write fields, such as RSTM v3), and certainly HTM systems, admit designs in which such events are detected as soon as they happen, potentially avoiding wasted work compared

```
For-each OW in Tx.Read-Set
  Wait until OW is Committed or Aborted
    If cycle detected, handle cycle
    If OW is Aborted
      Tx must Abort.
    If OW.Snapshot_OK_Flag is not set
      Tx must Abort

For-each OW in Tx.Write-Set
   Wait until all previous OWs to become
      non-active and without remaining
      readers
   If cycle detected, handle cycle

// At this point; guaranteed to commit
Tx.Status = Committed

For-each OW in Tx.Write-Set
  OW.OwnerSO.Lock
    If OW.Readers_Count == 0
      OW.Snapshot_OK_Flag = true
    Else
      OW.Snapshot_OK_Flag =
          Equals(A_Or_Cmt_Obj,
                 Fwd_Sshot_Obj)
    OW.Committed_Flag = true
  OW.OwnerSO.Unlock

For-each OW in Tx.Read-Set
  OW.OwnerSO.Lock
    OW.Reader_Count--
  OW.OwnerSO.Unlock
```

Figure 7: Transaction commit pseudo-code.

```
Tx.Status = Aborted

For-each OW in Tx.Write-Set
  OW.OwnerSO.Lock
    OW.Aborted_Flag = true
  OW.OwnerSO.Unlock

For-each OW in Tx.Read-Set
  OW.OwnerSO.Lock
    OW.Reader_Count--
  OW.OwnerSO.Unlock
```

Figure 8: Transaction abort pseudo-code.

to *DASTM*.

Figure 7 presents pseudo-code for transaction commit processing, which happens in two phases. Note that when waiting for the read-set to commit the shared-object lock does *not* need to be taken (it is only taken by the transaction that will commit/abort the object), and thus the flags are used as notification mechanisms (and would be quite efficient with common cache coherence protocols).

Transaction abort processing may occur in reaction to an explicit call to Abort or an exception thrown during transaction processing, or in the validation phase of Commit processing, where one of several events may cause a transaction to abort. The pseudo-code for Abort processing is shown in 8. For clarity, cleanup of ObjectWrappers from the list is omitted from the pseudo-code. Nodes are removed from the shared-object list when they are no longer needed by any transactions.

4.6 Implementation simplifications

DASTM is not highly optimized, does not leverage standard optimizations to speed up searches for previously opened objects. As our evaluation focuses on increased concurrency, less attention was paid to improving performance of each individual transaction. Moreover, several design simplifications were undertaken to ease the implementation.

The first simplification is that dependences created for any transaction are always created with respect to the most-recently written (committed or active) value of the object, i.e. the dependences are position at the “tail” of the dependence chain. By ruling out more flexible placement of dependences, certain design simplifications are enabled in *DASTM*. This simplification also entails that W$\leftrightarrow$W dependences, which need not be ordered, are actually ordered in *DASTM*.

The second simplification is that a given transaction will have only a single value which is forwarded to other transactions, which is the value at the time of the first forwarding operation. Even if the writer is continuously changing the object only a single snapshot will be used by the various readers. This may cause some readers to abort in cases where a more flexible policy would have allowed them to commit (e.g. a policy that always forwards the most recent write reduces the probability that updates of the forwarded value require readers to restart). However, since we do not yet know how frequent these cases are, this restriction is not unreasonable. A final simplification relates to infinite loops which may arise when a transaction reads inconsistent forwarded data. The *DASTM* implementation delays the check of whether read data was consistent to when the consuming transaction completes, instead of when the producing transaction overwrites the data. Since our benchmarks do not incur this problem, the current implementation does not handle it, although it should not be difficult to address in this design, or in alternative dependence-tracking designs.

5. DEPENDENCE-AWARE HTM

This section sketches the hardware support needed to implement dependence-aware HTM: W$\rightarrow$R data forwarding, restart when forwarded data is invalidated, commit ordering, and deadlock prevention/detection. Dependence tracking is done at the level of words, not cache lines. If T_0 writes a variable a and T_1 reads it, and then T_1 writes variable b and T_0 reads it, then the system should record both dependences $W_0 \rightarrow R_1$ and $W_1 \rightarrow R_0$, even if a and b are on the same cache line. The proposed hardware uses signatures [4], which can efficiently track read and write sets at the level of bytes, and the TM system is based on LogTM-SE [25]. We assume a CMP with a shared bus, though all of our techniques would work with a directory, albeit less efficiently.

5.1 W$\rightarrow$R data forwarding

LogTM-SE forbids other L1 caches from holding data that is in the write-set of a transaction on another core. It also forbids the exclusive caching (no M or E) of any line in the read-set of a transaction on another core. Dependence-aware HTM maintains this invariant, though it tracks read and write sets at the level of bytes. Therefore, other cores generate requests for writes to data read-cached by another (source) transaction and for reads to data write-cached by another (source) transaction.

W$\rightarrow$R data forwarding happens naturally—write requests are seen by the source transaction as coherence messages, and the source node responds with data. However, the source node must restart the destination node if it overwrites forwarded data. Therefore the source node maintains a *forwarded signature* in addition to the read and write signatures of LogTM-SE. Before retiring a write the local node must determine if the address written is contained in the local forwarding signature. If it is, the write value is broadcast on the bus. All transactions test the broadcast write address for membership in their their read set and if there is a match, they restart. Overwrites of the same value will cause needless restarts. False positive matches on the forwarded signature increase bus traffic, but they do not cause transaction restarts.

5.2 Ordering and deadlock detection

Each processor maintains a list that is a topological sort of the global transactional dependence graph. There are two types of bus messages: the first indicates a dependence a→b, and the second clears all dependences involving transaction a. All CPUs see all messages in order on the CMP bus, so all of them build the same dependence list. The cyclic dependency checking algorithm must require constant time to check each new dependence broadcast on the bus. Transaction commits and restarts broadcast the clear message. These are rarer than transactional memory accesses, so they can require a bit more work.

Each processor keeps an ordered list of processor identifiers that is a topological sort of the dependence graph—each identifier appears before identifiers that depend on it (sources before destinations). For every dependence the processor sees on the bus, it checks to see if the dependence is incompatible with its current topological sort. An incompatibility indicates a circular dependence. If either or both of the dependents is not present in the current sort they are added in the proper order. Each core tracks the dependences of the current transaction in enough detail to delay their own commit and to (conservatively) detect deadlock. At commit time, the commit blocks until the dependence list is empty. W→W dependences are not broadcast.

For instance, consider processor 3. It sees the dependence 0→5, so it adds 0,5 to its topological sort. It then forwards data to 4, making the sort 0,5,3,4. If it reads data from processor 5, that does not change the sort. If it sees a dependence 4→5 on the bus, it has detected a circular dependence. Because the sort is conservative, if there is a circular dependence in the transaction graph, at least one processor's topological sort will detect it.

The topological sort can give false positives. With dependences 1→2 and 3→4, the sort will be 1,2,3,4. A 3→2 dependence will be incorrectly flagged as circular.

The cache controller can implement the sorted list using $N+1$ counters, each with $log(N)+1$ bits, where N is the number of CPUs. The $N+1$th counter is the `nextID` counter, and it is initialized to 1, while every other counter is initialized to 0. When a dependence message is seen on the bus, if the source's counter is 0, set it to `nextID++`. If the destination counter is 0, set it to `nextID++`. If the destination counter is smaller than the source counter, signal a cycle. On a clear for processor c, the controller gets the value of c's counter, call it C. It sets c's counter to 0, then it decrements all counters from $C+1\ldots\ N$, and `nextID`.

Once a circular dependence is detected there are many ways it can be broken. Restart algorithms that require more communication can minimize the amount of lost work by choosing to restart a particular transaction (or transactions).

5.3 Virtualization

Dependence-aware transactions do not add much more state than what is contained in LogTM-SE. Each transaction requires a forwarded signature, though it is likely that the forwarded signature can be shorter than the read and write signatures because data forwarding is rare relative to reads and writes. Collisions in the forwarded signature create more bus traffic, but do not necessarily falsely restart transactions. A transaction that is inactive needs to have its forwarded signature stored, but the processor does not need to access the signature until the transaction becomes active again.

Processor identifiers are insufficient for dependence and deadlock detection when transactions can become inactive. Instead of processor identifiers, the system can use longer transaction identifiers. To support e.g., 3 inactive transactions per-processor, the transaction identifiers can have 2 extra bits. The low bits of the transaction identifier equal the processor identifier. The dependence list of transaction identifiers is per-transaction, and must be stored but not accessed for inactive transactions. The topological sort of transaction identifiers refers to active and inactive transactions.

Benchmark	*Description*
counter	Each thread repeatedly increments a shared global counter within a transaction. The RSTM code was modified to add think time to each thread.
llist	Each thread manipulates a shared linked list within a transaction. The linked list contains 512 nodes, and each transaction searches for a random node. Once found, with a certain probability the transaction will modify the list by moving the node to the head of the list. Two variants of this benchmark are llist8020, where the probability of update is 20%. llist5050 increase the probability of updates to 50%.
LFUcache	Web cache simulation benchmark, adapted unmodified from RSTM suite. Uses an array-based index and a priority queue to track the most frequently accessed pages.

Table 2: Benchmarks used to evaluate DASTM.

6. EVALUATION

This section presents experimental results of our software-based implementation of dependence-aware transactional memory. The results are compared with traditional STM approaches, represented by RSTMv2 [10][1], a publicly available STM implementation.

6.1 Benchmarks

We selected three benchmarks to use in our evaluation. As both TM implementations (dependence-aware and RSTM) expose the same API, the benchmark code was identical. Two of the benchmarks (counter, LFUcache) were adapted from the suite of benchmarks provided with the RSTM distribution, while the third benchmark (llist) was developed by us. Table 2 provides a brief description of each benchmarks.

6.2 Experimental setup

RSTM can be configured to use eager or lazy object acquisition, and visible or invisible readers. We provide results for all four configurations. All experiments were run on a Sun Fire T200 (Niagara) with 8 cores (32 thread contexts), running SunOS 5.10. Experiments were run from 1 to 31 threads because RSTM's visible readers implementation doesn't work with 32 threads.

6.3 Results

Figure 9 presents the performance of the counter benchmark. Traditional STM techniques as represented by all four configurations of RSTM do not scale at all due to the high-contention nature of the benchmark. The two lazy-acquisition configurations of RSTM before better than the eager-acquisition. The dependence-aware STM implementation is able to scale with the increased number of processors, with performance at 31 threads being 25x that of the lazy-acquisition configurations. Its performance also remains superior or equal, at low thread counts. The results show that with dependence-aware TM, programmers do not need to increase the complexity of their code (e.g. with privatization, escape actions, weaker semantics) to get higher performance.

[1]We began our work before the recent release of RSTMv3. While RSTMv3 may perform better than RSTMv2, it has a different interface, so we were not able to use it for this study.

Figures 10 and 11 present the linked list benchmark with different mixes of read and update transactions. For RSTM, the two configurations with visible readers provide higher performance than the invisible readers. Visible readers, while providing better absolute performance than dependence-aware TM, exhibit scaling only up to 8 threads, after which overall performance actually degrades. Dependence-aware TM continues to scale to 31 threads (the the maximum used in the experiment). The actual cross-over point where Dependence-aware TM provides higher absolute performance than RSTM's visible readers is around 16–20 threads, for the 20% update mix. When updates are more frequent, as demonstrated by the 50% update mix, RSTM performance is cut by half or more, while Dependence-aware TM performance remains unchanged, even in the presence of greater updates. The cross-over point at which Dependence-aware provides higher overall performance, is reduced to 12 threads. These results show that while at low parallelism count, other TM designs may achieve higher absolute performance, once the amount of concurrent work (threads) increases, dependence-aware TM allows traditional data structures to scale their performance, without adding extra complexity. (e.g. without having to add early release, as is done in one of the linked list programs in the RSTM benchmark suite)

Figure 12 presents the results for LFUcache, a benchmark where conflicts are common but transaction sizes are small. Unlike the counter benchmark, no think time is added, and this reduces the ability of dependences to increase concurrency by interleaving what would otherwise be conflicting transactions. The trend in transactional memory has been towards longer and larger transactions, which bodes well for the dependence-aware model. However, it is important to also understand and evaluate the performance of DASTM with shorter transactions, such as with LFUcache. In this case, both RSTM configurations as well as dependence-aware TM do not scale, and performance degrades with more threads. However, RSTM lazy-acquisition configurations as well as dependence-aware TM perform 1.5x to 2x better than eager-acquisition. These results show that for some workloads where transactions have not been able to provide scalability, our current implementation of dependence-aware TM has commensurate performance with the best current STM models.

Even across this limited set of micro-benchmarks, it is clear that as concerns existing models, the best performing one depends on the application. In some cases visible readers is an important factor, in others it is lazy-acquisition that is the key to better performance. Our prototype of dependence-aware STM manages to perform surprisingly well across all the benchmarks, and enables new important. We speculate that several more sophisticated optimizations of dependences (e.g. cycle detection not based on timeouts, more sophisticated management of dependences, not ordering W↔W dependences) may enable further performance improvements for dependence-aware TM.

6.4 Dependence Aware HTM Results

We evaluate Dependence-aware HTM, or *DAHTM* using a single micro-benchmark: **pipebm**. For comparison, we obtained a copy of **MetaTM** [22] which is an HTM system using eager versioning and eager conflict detection. The **pipebm** micro-benchmark simulates a multi-threaded application that has long transactions and high contention, and consists of multiple threads (4× the number of processors) working through a set of 8 phases. If all threads are working in the same phase, contention is very high, while execution can be overlapped with minimal data dependence for threads in different phases.

For 2 CPUs, *DAHTM* was able to resolve 27,500 memory operations that would have caused restarts in a traditional HTM system by forwarding values and managing dependencies instead. Only 3 unique transactions restart (due to cycles in the dependence graph), and the total number of restarts for the entire benchmark is 7 (several transactions restart more than once). For the same benchmark on MetaTM, all transactions restart: the total number of restarts is on the order of 9 million: *DAHTM* reduces the number of restarts by six orders of magnitude.

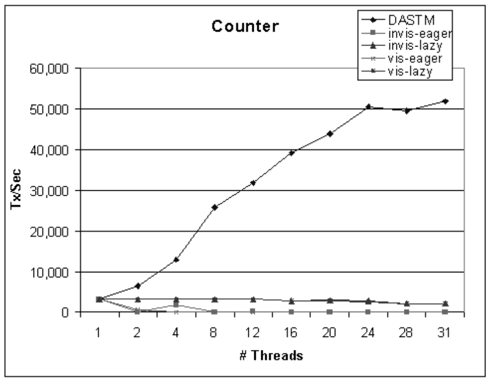

Figure 9: Comparative Performance for Counter benchmark.

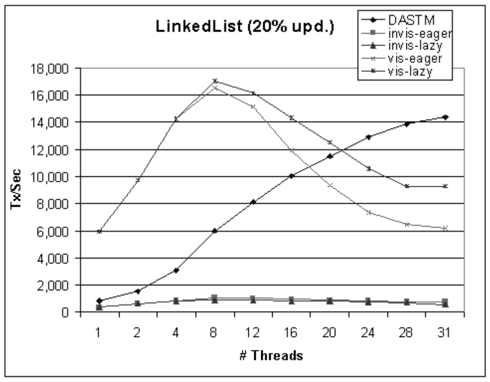

Figure 10: Comparative Performance for Linked List benchmark, with 20% probability of updates.

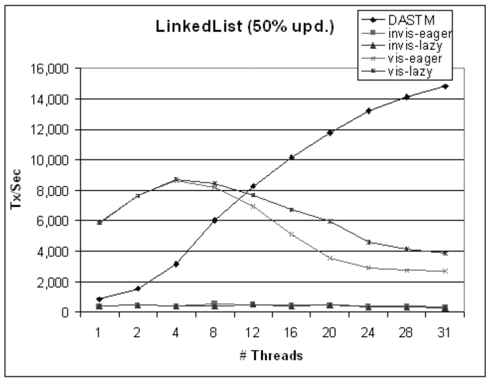

Figure 11: Comparative Performance for Linked List benchmark, with 50% probability of updates.

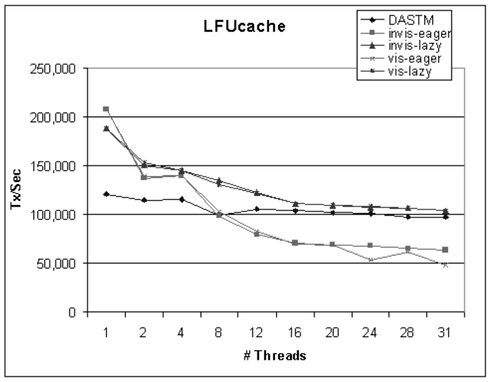

Figure 12: Comparative Performance for LFUcache benchmark.

7. RELATED WORK

Transactional memory has its roots in optimistic synchronization [12] and optimistic database concurrency control [15]. Herlihy and Moss [13] gave one of the earliest designs for hardware transactional memory. Rajwar and Goodman explored transactional [20] execution of critical sections, sparking a renewal of interest in HTM. Current work on HTMs has focused on the architectural mechanisms that provide transactional memory [1, 5, 11, 16, 18, 25], language-level support for HTM [3, 9], and transactional resource virtualization [2, 6, 21, 26].

Software transactional memory (STM) does not require hardware support, and usually works at the language level. There has been much recent work on making STM efficient [9], and integrated with language features like garbage collection. There is also work on hybrids [7, 14] or hardware-accelerated STM [**?**] that attempt to get the performance of hardware systems without the limitations of hardware or the need to virtualize.

Transaction dependences, as with transactional memory, also has roots in the field of databases in the concept of spheres of control [8]. Spheres of control are a powerful and general notion, presented in the context of a static hierarchy of abstract data types as well as dynamic spheres created around actions accessing shared data. Time-domain addressing systems [23] keep track of multiple versions of each object, indexed by time. One of the recurring problems in such systems is that transactions which only read an object require updating its history, adding to the problem of hotspots in such systems. Transaction dependences may be viewed as refining and formalizing the notion of dependences, and time-domain addressing, to an implementable realization, in the context of transactional memory.

Much attention has been paid to enabling concurrency for shared data structures, especially the canonical shared counter. Previous transactional memory proposals have proposed a transactional pause (*xact_pause*) to eliminate contention on the shared counter. Not only is programmer complexity significant for these techniques (must register compensation actions, increment must be performed with interlocked instructions), but the semantics are weaker (i.e. the counter is no longer monotonically increasing). It is notable that databases have had to deal with the high-contention counter problem as well, in the context of generating sequential, unique identifiers. The concurrency control mechanisms (locking) of databases, make it extremely difficult to have a scalable solution without special support. Such support includes support in the data definition language (e.g. marking a field as AUTOINCREMENT, so new rows are assigned unique values), as well as extensions (such as Sequence objects [19]) that operate outside the scope of the transaction, and thus have weaker semantics as well (for instance, gaps). In contrast, transaction dependences enable concurrency when shared data is being modified by multiple transactions, in a way invisible to the programmer, and is not tied to any specific data structure, able to benefit a range of common programming patterns.

Dependence-aware transactions do for transaction restarts what soft updates did for synchronous disk writes [17]. Dependence-aware transactions keep ordering information that allow it to reduce the number of restarts while maintaining the safety property of conflict serializability. Soft updates keep ordering information that allows it to reduce the number of synchronous disk writes while maintaining the safety property of the on-disk file system invariants.

8. CONCLUSION

This paper presents the concept of dependence-aware transactional memory. The different types of dependences and their properties are discussed, and issues such as multiple dependences, dependences among multiple transactions, and cycles are explored. We also present dependence-aware designs for both software and hardware transactional memory systems. Experimental results from our prototypes confirm the potential performance benefits of dependence-aware transactional memory, as compared with traditional TM implementations.

The increased throughput is due to increased concurrent execution by transactions that would otherwise conflict due to shared data structures. This allows programmers to write straightforward code and use familiar data structures, yet achieve good performance without having to resort to esoteric programming patterns or extensions to the TM programming model. Dependence-aware TM designs are thus an important step in the direction of fulfilling the alluring promise of transactional memory.

9. REFERENCES

[1] C. Anaian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[2] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[3] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *PLDI*, Jun 2006.

[4] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, 2006.

[5] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII*, 2006.

[6] J. Chung, C. Cao Minh, A. McDonald, H. Chafi, B. D. Carlstrom, T. Skare, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS*. ACM Press, Oct 2006.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XI*, 2006.

[8] C. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2), 1978.

[9] A.-R. A.-T. et al. Compiler and runtime support for efficient software transactional memory. In *PLDI*, Jun 2006.

[10] M. S. et al. *Rochester Software Transactional Memory*. `http://www.cs.rochester.edu/research/synchronization/rstm/`.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, page 102. IEEE Computer Society, Jun 2004.

[12] M. Herlihy. Wait-free synchronization. In *TOPLAS*, January 1991.

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

[14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP*, 2006.

[15] H. Kung and J. T. Robinson. On optimistic methods of concurrency control. In *ACM Transactions on Database Systems 6(2)*, June 1981.

[16] A. McDonald, J. Chung, B. Carlstrom, C. C.M., H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA*, Jun 2006.

[17] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. pages 1–17, 1999.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, , and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[19] Oracle Corp. *Guide to using SQL Sequence Number Generator*, 2003. `http://www.oracle.com/technology/products/rdb/pdf/0307_sequences.pdf`.

[20] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.

[21] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*. Jun 2005.

[22] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. Evaluating transactional memory tradeoffs with TxLinux. In *ISCA*, 2007.

[23] D. Reed. Implementing atomic actions on decentralized data. *ACM TOCS*, 1(1), 1981.

[24] G. Weikum and G. Vossum. *Transactional Information Systems*. Morgan Kaufmann, 1st edition, 2002.

[25] L. Yen, J. Bobba, , M. Marty, K. E. Moore, H. Volos, M. D. Hill, , M. M. Swift, and D. A. Wood. Logtm-SE: Decoupling hardware transactional memory from caches. In *HPCA*. 2007.

[26] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT*, Jun 2006.