Dependence-Aware Transactional Memory for Increased Concurrency

Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel
Department of Computer Sciences
University of Texas at Austin
Austin, TX, USA
Email: {ramadan, rossbach, witchel}@cs.utexas.edu

***Abstract*—Transactional memory (TM) is a promising paradigm for helping programmers take advantage of emerging multi-core platforms. Though they perform well under low contention, hardware TM systems have a reputation of not performing well under high contention, as compared to locks. This paper presents a model and implementation of dependence-aware transactional memory (DATM), a novel solution to the problem of scaling under contention. Unlike many proposals to deal with write-shared data (which arise in common data structures like counters and linked lists), DATM operates transparently to the programmer.**

The main idea in DATM is to accept any transaction execution interleaving that is *conflict serializable*, including interleavings that contain simple conflicts. Current TM systems reduce useful concurrency by restarting conflicting transactions, even if the execution interleaving is conflict serializable. DATM manages dependences between uncommitted transactions, sometimes forwarding data between them to safely commit conflicting transactions. The evaluation of our prototype shows that DATM increases concurrency, for example by reducing the runtime of STAMP benchmarks by up to 39% and reducing transaction restarts by up to 94%.

I. Introduction

Exploiting parallelism is the major software challenge for the coming decade. Power and frequency scaling limitations have caused manufacturers to shift their efforts away from scaling the performance of individual processor cores toward providing more cores on a chip. Memory transactions are a promising abstraction that can simplify concurrent programming thereby helping programmers harness the power of modern parallel architectures.

Transactional memory provides the abstraction of atomic, isolated execution of critical regions. By *atomic*, we mean that if a memory transaction fails for any reason its effects are discarded: either all of its updates become globally visible, or none of them do. By *isolated*, we mean that no memory transaction sees the partial effects of any other transaction: uncommitted or speculative state is private to a transaction. Transactions are also *linearizable*: each transaction appears to take effect instantaneously at some point between when it starts and when it finishes [12].

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].

Most transactional memory systems detect conflicts between two transactions and respond by forcing one of the transactions to restart or block. By restarting or blocking on conflict, TM implementations provide a level of concurrency that is equivalent to that of *two-phase locking* [8]. Even TM implementations that do not use locks [9], [18], including both eager and lazy systems, only provide concurrency equivalent to two-phase locking. Any data read or written by one transaction has an implicit lock on it that conflicts with any attempt to write the same data. The key insight of DATM is that using conflict serializability as the system's safety property increases concurrency relative to using two-phase locking.

We propose *dependence-awareness*, a transactional memory implementation technique that ensures conflict serializability. Dependence-aware transactional memory (DATM) manages conflicts by making transactions aware of dependences, and in some cases, by forwarding data values between uncommitted transactions. Dependence-awareness allows two conflicting transactions that are conflict-serializable to both commit safely, thereby increasing concurrency and making better use of parallel hardware than current TM systems. Dependence-awareness is *safe*—transactions remain atomic and isolated in the same way as current TM systems.

Most previous proposals to help TM deal with write-shared data involve mechanisms that complicate the programming model and require the attention of skilled programmers to be safe and effective. Dependence-awareness, by contrast, is completely transparent to the programmer. Transparency is particularly important because many common data structures, like shared counters and linked lists, have write-shared data that cause performance problems in conventional TM systems. Because dependence-awareness admits concurrency where current designs cannot, it provides good system performance without burdening programmers with exotic new programming issues.

This paper makes the following contributions:

1) We introduce dependence-aware transactions, a new TM model that permits substantially more concurrency than two-phase locking.
2) We present a design for a dependence-aware HTM sys-

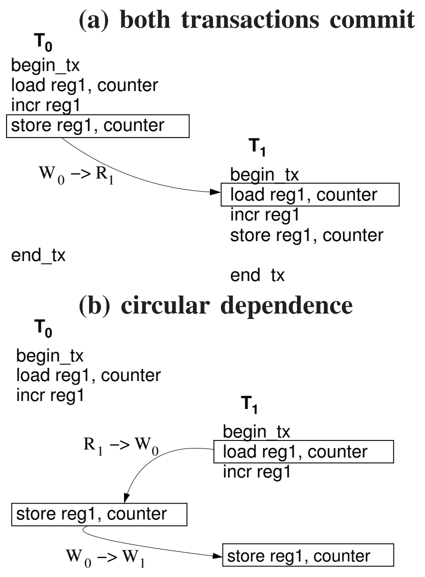


Fig. 1. Two transactions increment the same counter, illustrating (a) a successful commit using dependences with data forwarding, and (b) an abort due to circular dependences.

tem, including a cache coherence protocol called FRMSI (Forward/Receive MSI).

3) We evaluate our prototype implementation on a set of benchmarks that include representatives from STAMP [17] and TxLinux [23].

We illustrate dependence-awareness by example in Section II, and describe the full model in Section III. Section IV describes our dependence-aware HTM implementation, and Section V evaluates its performance. Section VI discusses related work and Section VII concludes.

II. Increasing concurrency with DATM

The dependence-aware model creates and tracks dependences between transactions that access the same datum, possibly allowing data to be forwarded speculatively from one transaction to another. Dependences let DATM commit transactions that a conventional TM would restart or block, making better use of concurrent work.

A. Shared counter example

We use standard notation for data dependences, for example, W$\rightarrow$R means a memory cell was written by one transaction and then the same cell was read by a different transaction. Dependences are subscripted with transaction numbers to indicate which transactions are involved. While the generic term "memory cell" indicates that the granularity of the datum is not intrinsic to the model, in this paper a "memory cell" is a cache line unless otherwise stated.

Consider the shared counter shown in Figure 1(a). Assume that two different threads on two different processors (P_0 and P_1) execute this code in two different transactions (T_0 and T_1). The executions overlap in time as shown in the figure, with time flowing down. If the counter value starts at 0, the figure shows T_0 forwarding its counter value (1) to T_1. DATM establishes a $W_0 \rightarrow R_1$ dependence for the counter, and ensures that T_1 commits after T_0. The transactions are allowed to proceed concurrently even though they both write the same memory location. The counter's final value is two, which corresponds to the serialization order T_0, T_1.

The interleaving in Figure 1(a) is conflict-serializable, but would not be allowed by the two-phase locking style of conflict detection done by current TM systems. In most current TM systems, after T_0 reads and writes the counter, any subsequent access to the counter by T_1 is considered a conflict, either forcing T_1 to block or one transaction to abort.

The interleaving in Figure 1(b) is not conflict-serializable, so both transactions cannot successfully commit. Here, T_0 writes the counter after it is read by T_1, creating a $R_1 \rightarrow W_0$ dependence, which constrains T_0 to commit after T_1. However, when T_1 writes the counter, it creates a $W_0 \rightarrow W_1$ dependence, which constrains T_1 to commit after T_0. The dependence graph contains a cycle, and if both transactions were to commit, the counter would have the wrong value. DATM handles this potential conflict by detecting the cycle—T_0 is dependent on T_1 and T_1 is dependent on T_0. It aborts one of the transactions to break the cycle.

B. Accepting more interleavings

Figure 2 shows three different interleavings (called schedules in the database literature) for the memory references of transactions that increment a shared counter. Interleavings (a) and (c) are conflict serializable. In (a), T_0 can be serialized before T_1, and in (c), T_1 can be serialized before T_0. Interleaving (b) is not conflict serializable. DATM accepts interleavings (a) and (c), while conventional TM implementations do not.

Of course, accepting more interleavings does not by itself imply that DATM will outperform conventional approaches, since many other factors impact actual performance. However, by accepting more interleavings DATM increases the likelihood that parallel resources are utilized when transactions execute concurrently—instead of conflicting, concurrent transactions can coordinate and both commit.

C. Comparison with other conflict resolution strategies

Figure 3 compares how DATM and existing systems execute a pair of transactions that conflict on a single shared datum. DATM creates a dependence from $T1$ to $T2$. Neither $T1$ nor $T2$ is forced to block or restart. DATM commits $T2$ earlier than the other conflict resolution strategies because it can accept memory access interleavings that require the other systems to block or restart.

Figure 3 shows eager conflict detection (done at the time of the memory reference) [18] and lazy conflict detection (done at commit time) [9]. Eager conflict detection with restart (Figure 3b) causes $T2$ to restart on the conflict, and $T2$ conflicts again. Eager conflict detection with stall-on-conflict (Figure 3c) causes $T2$ to stall until $T1$ commits. Finally, with lazy conflict

```
 T0                  T1                 | T0                  T1                 | T0                  T1
begin_tx            begin_tx            |begin_tx            begin_tx            | begin_tx           begin_tx
store reg1, counter                     |                    load reg1, counter  |                    load reg1, counter
                    load reg1, counter  |store reg1, counter                     |                    store reg1, counter
                    store reg1, counter |                    store reg1, counter | store reg1, counter
end_tx              end_tx              |                                        | end_tx              end_tx

               (a)                                     (b)                                     (c)
```

Fig. 2. Three execution interleavings of two simple transactions. Time flows down. All memory references are to the same shared counter. DATM can accept interleavings (a) and (c), indicated by the presence of the end_tx instruction.

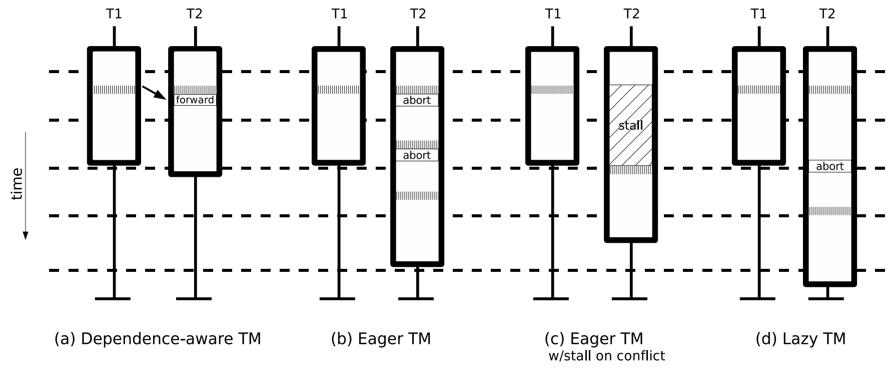


Fig. 3. Two transactions that conflict while incrementing a shared counter. Part (a) shows the dependence-aware implementation, while Parts(b-d) show conventional HTM techniques. Assume that transaction $T1$ always commits first. The shared counter accesses are the shaded regions within each transaction.

detection, $T2$ must restart when it tries to commit. Execution interleavings that cause stalls or restarts with current conflict resolution strategies are committed safely by DATM.

III. Dependence-aware model

This section presents the dependence-aware model, describing how the system maintains dependences and how those dependences affect transactions. The dependence aware model admits all conflict serializable schedules.

A. Dependence types

Table I shows a summary of dependence types and their properties. The notation W→R denotes a read after write (RAW) dependence—one transaction reads a cache line that was written by another transaction. Dependences are subscripted with transaction numbers, so that $W_0 \rightarrow R_1$ means a write from transaction T_0 was read by transaction T_1. All dependences restrict commit order. If there is a $X_A \rightarrow X_B$ dependence, then transaction A must commit before B.

The system tracks all dependences at the level of cache lines creating new dependences between transactions in response to memory accesses at runtime. The ordering of transactions depends on their dynamic behavior. The “Yes” in the **Forward** column for W→R dependences means the system forwards the data in the cache line when the dependence is created. The system records that the cache line has been forwarded.

For a $W_0 \rightarrow R_1$ dependence, we call T_0 the *source* transaction and T_1 the *destination*, or the dependent. The destination transaction must restart if the source restarts, because the destination has read data forwarded by the source. To maintain serializability, a dependent transaction can read a value from a source transaction only if that value will be the final value of the cache line for the source transaction. So the destination transaction must restart if the source transaction overwrites the data it forwarded. Table I lists the cases when restarts are necessary.

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

TABLE I
Summary of dependence types and their properties.

Dependences are created per cache line on first access to the cell. Subsequent accesses to the same object do not affect dependence structure For example, if T_0 writes a cache line that T_1 then writes, and then T_1 reads the cache line the resultant dependence is formed on the basis of the initial write and is $W_0 \rightarrow W_1$. When a transaction commits or aborts, all of its dependences disappear. The next section discusses how dependences between transactions form when they both access multiple memory cells.

B. Multiple dependences

Multiple dependences arise when two transactions conflict on more than one cache line. Each cache line on which two transactions conflict creates a separate dependence. To manage multiple dependences between two transactions, the model has the restrictive dependence rule: The relationship between transactions is governed by the most restrictive dependence in each direction. W→R is more restrictive than W→W and R→W dependences, and the latter two are not ordered relative to each other.

If a transaction is the source for a R→W dependence, and later it writes and forwards a different cache line 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 still tracked in the model.

If more than two transactions concurrently access the same cache line, then the first two will create a dependence as described above. The third transaction will create its dependence with the most recent writer of the cache line. The latest writer provides the most up to date version of the cache line. Conceptually, the dependences among transactions form a transaction dependence graph with a directed link between two transactions if there is a dependence between them on any memory cell.

C. Cyclic dependences

All dependences restrict commit order: a transaction must wait at commit time for any transaction that it depends on to commit. If cycles arise in the transaction dependence graph, the cyclic chain of dependences may cause deadlock. Dependences arise from reads and writes of memory cells, so a cycle indicates that the transactions have interleaved in a way that is not conflict serializable.

While there are several ways to handle cycles, our model avoids them. If a memory access would cause a cycle in the dependence graph, the system restarts at least one transaction in the cycle. The system does not allow cycles to form.

Another way to avoid cycles is to allow dependences only from older transactions to younger transactions. *Timestamp-ordered dependences* go in a single direction only, so they cannot form cycles. However, timestamp-ordered dependences do restrict concurrency more than a policy that allows dependences between any two transactions.

Contention management is important for dependence-aware transactions, just as it is for conventional TM systems [23], [27]. When the system detects a cycle in the dependence graph, it must restart at least one transaction in the cycle to break it. The contention management task is to preserve as much concurrent work as possible, such as by restarting transactions that do not have dependents.

D. Disabling dependence tracking: no-dep mode

One attractive property of dependence-aware transactions is that they co-exist with other conflict resolution strategies for ensuring safety. Restarting a transaction in *no-dep mode* disables dependence tracking for a particular transaction. Sections III-E and III-F explain uses of the no-dep mode.

E. Exceptions and inconsistent data

Because the model forwards data between transactions, it is possible that a transaction can read invalid data, which in turn can lead to exceptions or infinite loops. Inconsistent state seen by destination transactions are eventually made consistent at the completion of the source transactions. The writes that bring the source transactions into a consistent state cause a restart of the destination due to overwrites of forwarded data. The restart of the destination eliminates infinite loops that are not part of the application's serial behavior.

A transaction that has read inconsistent data can throw an exception before subsequent execution of the source transaction causes the destination to restart. The hardware informs the OS through the transaction status register if the currently running transaction has read forwarded data. The OS exception handlers suppress these exceptions and, according to its policy, can restart a transaction in no-dep mode to avoid further spurious exceptions. Section V quantifies the small number of times transactions execute in no-dep mode for our prototype.

Program asserts must also be made dependence-aware. Assert failures in transactions that have read forwarded data can be restarted or the failure is delayed until the source transaction commits.

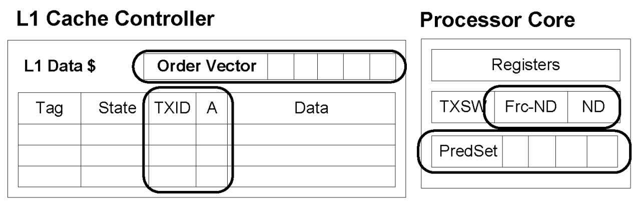
F. Cascading aborts

Cascaded aborts occur when one transaction's abort causes other transactions to abort. For example, a cascaded abort happens when a source transaction forwards a value to a destination transaction and the source aborts—the destination must abort as well. In DATM, cascaded 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. Section V quantifies the small effect of cascaded aborts on the performance of our prototype.

IV. Hardware design

This section discusses the hardware design for dependence-aware transactional memory. Key elements in the design are shown in Figure 4. The design must implement commit ordering, version management, W→R data forwarding, restart when forwarded data is overwritten (called a *forward restart*), and cyclic dependence prevention. To understand how these pieces interrelate, we first describe solving them with a minimum of new hardware. We then refine the design to improve performance while still keeping hardware state and hardware complexity low (Section IV-D).

DATM can be implemented with a novel cache coherence protocol called FRMSI (**F**orward **R**eceive **MSI**: pronounced like pharmacy), along with either an an ordered vector of



State Encoding

state	m	v		mt
M	1	1	0	0
S	0	1	0	0
I	0	0	0	0
TM	1	0	0	0
TS	0	1	1	0
TR	0	0	1	0
TMM	1	0	1	1
TMF	1	0	0	1
TMR	1	0	1	1
TMRF	1	1	1	1
CTM	1	1	0	1

Response to Bus Messages and Commit/Abort Events

msg	TM*	TR/TMR	CTM
GETX	asym. conflict	asym. conflict	reply(line,txid)
GETS	asym. conflict	asym. conflict	reply(line,txid)
TGETX	newdep:R→W:(txid)	newdep(txid)	reply(line,txid)
TGETS	newdep:W→R:fwd(line,txid)	newdep(txid)	reply(line,txid)
TXOVW	abort:TMR*->I	abort: TM*→I	--
xABT(local)	TM*→I	T*R→I	--
xCMT(local)	TM*→CTM(order)	TMR→CTM, TR→I	--
xABT(remote)	All: update order vector		
xCMT(remote)	All: update order vector		

Fig. 4. DATM architecture overview. DATM-specific state and structures are highlighted with dark lines.

transaction IDs maintained at each cache, or a timestamp table. The cache coherence protocol supports version management, helps order write backs of committed state, and handles data forwarding and forward restarts. DATM relies on global ordering for transaction commits and write backs of data modified in committed transactions, as well as for prevention of deadlocks and cycle dependences. Support for such ordering decisions can be implemented either using timestamps generated at transaction begin for contention management [21], or using an ordered vector of transaction IDs. The FRMSI protocol relies on the augmentation of cache lines with a transaction identifier [16], [23], shown as **TXID** in Figure 4.

An important design principle in DATM is that while dependences enable concurrency not currently accessible in TM designs, dependences are not a requirement for transactions to proceed. If any hardware resources or structures in the DATM design reach a limit, dependences for that transaction are dynamically disabled by restarting in a force-no-dependence mode (Section III-D) that resembles a current TM design. DATM is a *best effort* design, and contains no explicit overflow (sometimes called virtualization) strategy for when transactional state overflows hardware buffers—it can use any of the many current proposals [2], [5], [6], [22], [28].

A. Transaction status word

DATM adds two bits to the transaction status word, a register that holds the current state of the running transaction. One bit is a no-dependence bit (shown as **ND** in Figure 4), which indicates that the current transaction has not entered into any dependences. Transactions that have no dependences can be created and can commit without support from dependence-aware mechanisms: an explicit bit makes this check efficient. Decoupling dependence- and non-dependence-aware transactions ensures that regardless of the state of the dependence-aware hardware, transactions in the system can still commit, and forward progress can always be made.

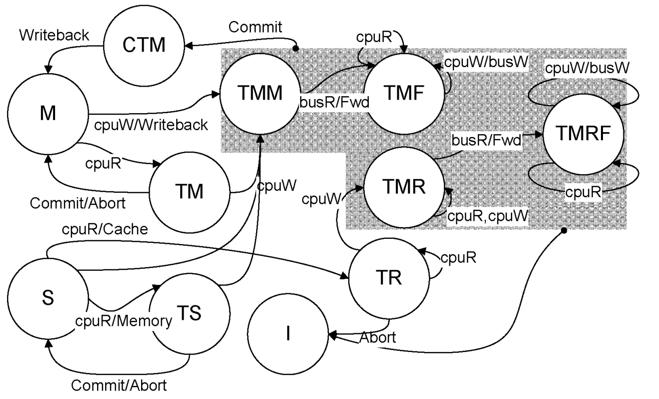


Fig. 5. State diagram for the FRMSI cache coherence protocol. Standard transitions between MSI are omitted for clarity. Transitions out of I are omitted as they are the same as those out of S.

DATM also adds a force-no-dependence bit (**Frc-ND**), which disallows the current transaction from entering transactional dependences. This state allows DATM to fall back into a traditional eager conflict-management HTM mode [18].

B. FRMSI coherence protocol

DATM is implemented with support from the FRMSI cache coherence protocol, which extends the MESI protocol, and has 11 stable states. The state diagram is shown in Figure 5. The E state is omitted for simplicity; it can be added as an optimization, but is not necessary. We could reduce the number of states if we use signatures [3] to track forwarded and received bytes. First we review the mechanics of the protocol and then show how the protocol achieves the goals of DATM.

Version management in DATM is complicated by data forwarding, which results in the ability of multiple caches to modify the same cache line. FRMSI contains states for forwarding and receiving data, allowing this kind of data sharing.

The TM and TS states are entered by lines in M or S that are read during a transaction. These lines can transition back to M or S when a transaction commits or aborts, because they are not modified during the transaction.

All TM* states (shaded) and TR (transaction received) states transition to invalid if the transaction aborts. All TM* states (note that TM* does not include TM) indicate a line is written during a transaction. These writes are buffered in the cache, but are discarded if the transaction is not successful, by a transition to I. A TR line must revert to I on abort because it contains speculative data received from an active transaction.

TMR and TR are states for cache lines that receive forwarded data, while TMF and TMRF are states for cache lines that have forwarded their data. These states are explained in detail in Section IV-B2. Commit of lines that are modified in a transaction is the subject of the next Section.

1) Committing: All TM* states (shaded) transition to CTM (committing transactional modified) on a commit. The CTM state is much like the M state in that it indicates a line that has been modified with respect to main memory and requires writeback. However, a line in CTM state must obey the ordering restrictions associated with the transaction that wrote the line. To understand the need for a CTM state, consider that FRMSI allows lines to exist in TM* states in multiple caches, but cannot allow lines to exist in the M state in multiple caches. This could be addressed without an additional state if, on completing a transaction, all lines in TM* state were to atomically write back to memory, stalling the transaction commit until all write backs complete and transitioning those lines to state I. Such a solution is unattractive because waiting for write backs increases the latency of transactional commit, which must be fast for TM to provide good performance [23].

Instead, the CTM state allows write backs to take place after transaction commit while still preserving ordering with respect to other transactions in the system. Transaction commit causes all updates made by a transaction to linearize [12] to that commit point. Consider a line that is written by transaction A, and then forwarded to B which also updates it. A is constrained to commit before B. If commit includes all write backs, then after both transactions commit, B's line is in memory and in its cache in state M, which is correct. With delayed write back for transactionally modified state, the lines enter CTM, where A's line is constrained to never overwrite B's. Any access to the line gets B's version, which is the latest one. The CTM state uses the order vector to order accesses and write backs to committed data, as explained in Section IV-C.

2) Forwarding and receiving: One of the chief goals of FRMSI is to enable cache line forwarding among transactions. When a cache controller sees a transactional bus read (**TGETS**) for a line that it has in state TMM or TMR (the line has been locally modified in a transaction), then it responds with the line and the identifier of the transaction that wrote the line, and moves the state into TMF or TMRF. The receiving cache can be transitioning from I or S into TR, the transactional received state.

Forwarded lines (states TMF and TMRF) publicize writes, in effect using an update protocol by sending a **TXOVW** message on the bus to indicate that previously forwarded speculative values are now stale. Any cache that has the line in a received state must abort its transaction if it sees a write to the line, because speculative data it received has been overwritten. The transaction only aborts in TMRF if the overwrite is from an earlier transaction where ordering is defined in Section IV-C. We later describe additions to the protocol, which reduce the granularity of detecting the overwriting of forwarded data (Section IV-D1).

3) Suspending transactions: DATM allows suspended transactions [23], [32], and it allows transactions with dependences to suspend and resume. Cache lines store the transaction ID to enable suspend and resume [23]. However, any attempt to create a dependence with a suspended transaction will fail and the operation will be handled as a transactional conflict, requiring the restart of one of the transactions involved.

Processor identifiers are insufficient for dependence management and cycle detection when transactions can suspend. For example, to support 3 inactive transactions per processor, the transaction identifiers have 2 bits more than the number of bits in the processor identifier.

C. DATM ordering requirements

DATM provides conflict serializability by ordering dependent transactions with respect to each other, and by linearizing their updates to transaction commits. We first present the ordering requirements of DATM and then discuss two implementation strategies: an order vector and a timestamp table.

These are DATM's ordering requirements.

1) Dependent transactions must commit in order.
2) Transactions that form dependences by receiving forwarded speculative data must become dependent on the most recent writer of that data.
3) Cyclic dependences must be detected in advance, and avoided by restarting one or more transactions.
4) Dependences are transitive: when transactions abort, dependence ordering must be preserved for transactions that remain active.
5) Caches with the same line in CTM state must write back the lines in the order dictated by their commit order, and subsequent requests for the line must be serviced from the last cache to commit.

The succeeding text has numbers in bold parenthesis to indicate how the design enforces the given requirement (e.g., **(1)** marks the explanation of how dependent transactions commit in order).

1) The order vector: A DATM implementation can support ordering by maintaining an **order vector** of transaction IDs in each cache. Each cache that contains a transaction with dependences maintains a copy of the order vector, and all copies have identical data. Each entry in the list has a transaction identifier, a valid bit and an active bit. The active entries in the vector topologically sort the dependence graph of the currently active transactions. The order vector provides the serialization order for active transactions and for writebacks of committed transactions whose results are still cache resident.

The vector reflects the superset of all dependences between transactions. Dependences are created when a cache snoops a memory access on the bus that is responded to by a cache rather than memory (using a mechanism analogous the *shared* response to **GETS** request for a line in **S** or **E** in MESI). New dependences are appended to the list, after all valid transaction identifiers. If transaction A forwards a cache line to transaction B, then A,B is appended to the list (with the rightmost position being the newest transaction). Each cache must see these dependences in the same order if the vector is to be identical at every cache. In our bus-based design, the bus ensures all dependences are seen in the same order: FRMSI would require extra messages to be extended to support a directory protocol.

2) Timestamp ordering: Timestamp-ordered dependences are implemented with a *timestamp table*. Each cache that contains a transaction with dependences (active, or with pending write backs) maintains a copy of the timestamp table, and all copies have identical data. Each entry in the table has a transaction ID, a timestamp, a valid bit and an active bit.

Using timestamps is similar to using the order vector, any difference are noted below in the discussion of the order vector. The main simplification of timestamp ordering is that dependences only go from older to newer transactions, so cyclic dependences cannot arise **(3)**.

3) Meeting ordering requirements: A transaction can only commit if it is the first (leftmost) active transaction in the order vector **(1)**. Being first ensures that this transaction does not depend on any others. When using timestamps, it can only commit if it is has the smallest timestamp in the timestamp table **(1)**.

If a transaction receives a cache line, its ID gets appended (on the right) to the order vector. The receiver uses the order vector to determine the last dependent transaction that provides the data for that line **(2)**. Suppose A has forwarded a line to B. If C reads that line, then C should receive the line from B, not A. If both A and B attempt to forward the line to C, the order vector is used to determine that B's data should be received and A's should be discarded. Using the received cache line from the transaction with the highest timestamp also creates a dependence with the latest writer **(2)**.

When a new dependence is added, the hardware checks if the transaction ID already appears in the vector of an active entry *to the left of the current transaction*. If it does, there is a cycle in the dependence graph and some transaction in the graph must be restarted **(3)**. The order vector thus provides a simple mechanism to detect cyclic dependences. Cyclic dependences cannot form when using timestamps **(3)**.

When a transaction aborts, it must publish this event to the coherence protocol by placing the **xABT** message on the bus along with its transaction ID. An abort message notifies other processors to turn off active and valid bits for that transaction ID in the order vector, while leaving its predecessors and successors in place **(4)**. Therefore, if a dependence chain of transactions A $\rightarrow$ B $\rightarrow$ C arises, B can abort without affecting A or C (provided the dependences are not forwarding dependences) C remains serialized behind A. The **xABT** message is not needed with timestamps, but an aborted transaction maintains its timestamp and hence its place in the serialization order **(4)**.

Commit must also be made visible to the coherence protocol, placing the **xCMT** message on the bus along with its transaction ID. The commit message notifies other processors to turn off the active flag but leave the valid bit. This way, the commit can retain its position in the order vector to order write backs for lines moving to the CTM state. This is explained in the next Section, and is handled identically when using timestamps.

4) Write backs: Position in the order vector is used both to order the commit of active transactions and to order writebacks for lines modified in committed transactions. Entries in the order vector for committed transactions with pending writebacks are valid but not active. Transactions remain valid in the order vector until all lines from the transaction have exited the CTM state. The ordering remains valid until all of the data updated during the transaction leaves the CTM state. A cache can detect when it has written back the last CTM line for a given transaction ID and at that point it sends a message to make the ID invalid in the order vector. Detection can be implemented using simple logic on the state bits, or if the CTM state were recoded as a single bit, as a wired OR.

Note that non-dependent transactions that include the TM and TS states can execute and commit while lines are in the CTM state. If the processor accesses any line in a CTM state, the line is written back and then the processor processes the operation as if the line were in M.

All cache lines in CTM are marked with their transaction identifier (**TXID**), and the serialization order is determined by looking up the TXID in the order vector or timestamp table **(5)**. The order vector or the timestamp table enforce a single, global order for write backs in addition to active transactions.

Write backs are ordered and can be squashed. Assume A forwards a line to B, B overwrites the line, and then both transactions commit. If A sees B write back the line that A forwarded, A can transition the line from CTM to I without writing back. B's version is serialized after A.

The situation is similar for bus reads. If another processor issues a bus read for the line that both A and B have in CTM, then both transactions can respond to the request, write back the value, and transition the line to state S. If B responds first and A observes B's response, then A can squash its own response and transition the line to I.

Timestamps are used to order write backs in the same way the order vector is used **(5)**.

5) Capacity of the order vector or timestamp table: Any transaction that needs a dependence can claim the newest index after the last valid index. Once claimed, the index serializes the transaction after any that has results that might be written back. Making the order vector large will minimize the probability that the vector will fill. The order vector can be large because it is not communicated and it is mostly consulted during a cache miss, when its access latency can be overlapped with data fetch.

No matter how long the order vector is, it can fill because with pathological line replacement, lines can remain in the CTM state indefinitely. A new transaction cannot get a dependence if the order vector is full or if it has the same identifier as a valid entry in the order vector. In these cases, transactions simply restart with the force-no-dependence flag and the computation continues.

When the order vector fills, the last entry must write back its CTM lines. Each cache can monitor if it has a transaction that occupies the last entry and initiate write backs for the lines that transaction has in the CTM state.

The timestamp table can also fill, which would also require a cache to write back lines in the CTM state in order to free an entry in the table.

D. Performance optimization

This section describes additions to the basic protocol to optimize performance. These changes allow the hardware to manage dependences at the word level while keeping write-back and most cache coherence operations at the cache-line level. It also prevents short transactions from convoying behind unrelated long-running transactions.

As a motivating example, consider a line that starts with its data equal to all zeroes and that is not present in any cache. Transaction A writes word 0 with value A and transaction B writes word 1 with value B. The write from transaction A causes the line to enter A's cache in the TMM state. B's write of word 1 results in a bus read for the cache line that is forwarded from A's cache. A's cache moves the line into TMF and B's has it in TMR (not TR because it wrote the word after receiving the line).

In the cache-line-based design, if A writes word 2, it generates a bus write that causes transaction B to abort, due to a circular dependence $W_A \rightarrow R_B$ and $W_B \rightarrow R_A$. However, there is no circular dependence at the word granularity because B does not read the word A writes. Eliminating these false cycles will improve DATM's performance.

1) Per-word accessed bits for received states: We augment the cache with per word access bits (labeled **A** in Figure 4). On receiving a line (TR, TMR or TMRF), the processor resets accessed bits, one per word in the line. Every time the processor reads or writes a word, it sets the access bit for the word. The access bits play an important role in interpreting bus writes to forwarded lines. Such writes either cause restarts, update the value of the word, or are ignored.

The rules for dealing with overwrites to forwarded lines enforce the obvious causality: previous transactions cannot overwrite data the current transaction has read or written, but a future transaction may. When the processor sees a bus write for a line that it has received, it compares the order vector entry of the transaction writing to the bus with the transaction identifier of the line. If the bus write index is earlier and the access bit is clear, the word is updated (a previous transaction is updating a word untouched by the current transaction). If it is later, the word is not updated (the word belongs to a transaction that is serialized after this one). If the access bit is set and the index is earlier, the transaction aborts (forward restart). Otherwise, the transaction ignores the message (a future transaction will change the same word this transaction changed).

Publishing writes to the bus for forwarded lines make these states act like an update protocol. All receivers have the same value as earlier transactions for words the receivers do not touch. This value agreement allows cache lines to be written back without lost updates. In the above example, if A writes word 2 and that write is not propagated to B, then the line that B commits will have a zero for word 2, not an A, which is a lost update.

2) Predecessor transaction set: The main problem with having a single ordered vector for all transactions in the system is that short transactions may have to wait for long transactions. For instance, if transaction A forwards data to B, and then transaction Y forwards data to Z, the order vector will read A,B,Y,Z. If Y and Z are very fast and A and B are slow, then throughput will suffer, as Y and Z must be at the head of the order vector in order to commit (condition **(1)** above).

We add a set of predecessor transactions to each processor (depicted as **PredSet** in Figure 4), to prevent transactions having to wait for unrelated transactions. A transaction can commit when its predecessor set is empty, it does not need to be at the head of the order vector. The transaction builds the predecessor set with the identifiers of any transaction from which it receives data. The set requires a maximum of only P entries, where P is the number of processors. An active transaction must restart if it wants to commit but has a suspended predecessor. The set can be smaller than P and, if it fills, the transaction restarts in force-no-dependence mode. Timestamp-ordered dependences benefit in the same way from the predecessor transaction set.

V. Evaluation

In this section we provide details of our simulation model, benchmarks and experimental results.

A. Prototype model

We implement a dependence-aware HTM model by modifying a publicly available HTM simulator (MetaTM [23]).

	Configuration
Processor	Pentium-4-like x86 instruction set, 1 GHz, 1 IPC
L1	Each core has separate data and instruction caches. 32 KB capacities, with 8-way associativity, 64-byte cache lines, lru-replacement policy, 1-cycle cache hit.
L2	4 MB capacity, 8-way associative, with 64-byte cache lines, 16-cycle access time.
Memory	1GB capacity, 350 cycle access time.
MetaTM	Timestamp contention management, linear backoff policy, word granularity conflict detection.

Fig. 6. Architectural parameters of simulated machines.

The model is implemented as a module in the Simics 3.0.27 machine simulator [15]. The core architectural parameters are shown in Figure 6. We evaluate the model with a 16-way SMP configuration (except for TxLinux benchmarks, where an 8-way SMP is used). Each processor has a private L2, and the L1 data caches contain both transactional and non-transactional data.

We modify the MetaTM cache coherence protocol, which is based on a MESI snooping protocol, to support transactional dependences using FRMSI. The latency of forwarding data between processors is conservatively modeled as a write back and a read from memory. Bus arbitration and bandwidth constraints are not modeled. The L2 cache also requires transactional state so that transactional state is visible to the coherence protocol (how to support transactional variants of MESI in the presence of multi-level private hierarchies is an open research question, and previous proposals would also be forced to make similar tradeoffs [29].)

The workloads are described in Table 7. They include a transactional operating system, several STAMP [17] benchmarks, and two micro-benchmarks to focus on specific data structures.

B. Experimental results

Figure 9 shows the basic runtime characteristics of the workloads on both MetaTM and DATM. Figure 8 shows graphically how DATM, normalized to MetaTM, reduces the execution time and number of restarts. The results show that, in almost all cases, DATM improves or does not harm the performance of realistic workloads. DATM increases concurrency by reducing the average number of restarts per transaction and average backoff cycles per transactions. Reducing restarts does not necessarily improve performance, but it does when the reduction is an indicator of increased concurrency.

In particular, the bayes and vacation STAMP workloads show a dramatic reduction in restarts, and a 39% improvement in execution time. The remaining STAMP benchmarks have little contention, so DATM does not change their performance. For instance, while DATM reduces ssca2 restarts by two orders of magnitude, it has only 0.1 restart per transaction on average under MetaTM.

Performance for the TxLinux workloads (pmake, config) is mostly flat because they spend a small amount of time executing transactions [23]. DATM reduces restarts substantially, but these

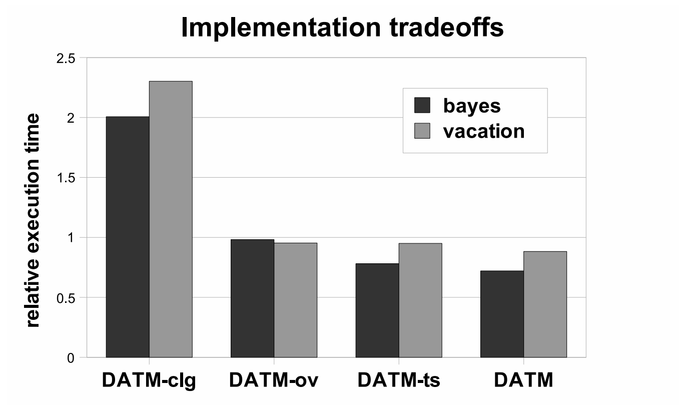


Fig. 11. Relative execution time for various DATM hardware designs. DATM-clg uses cache-line granularity to manage dependences, DATM-ov constrains dependences according to the order vector, and DATM-ts constrains dependences with timestamp-ordered dependences. All relative execution times are normalized to MetaTM. Lower is better.

restarts are not a performance problem. The counter micro-benchmark (especially with think-time) is able to dramatically benefit from dependences, with up to an order of magnitude improvement in performance. DATM effectively forwards the counter values between uncommitted transactions.

Dependence-related statistics are shown in Figure 10. Benchmarks that spend significant time in transactions commonly form dependences. For vacation, 35% of transactions form a forwarding dependence (W→R), and for labyrinth 32% of transactions form R→W dependences. The formation of dependences increases concurrency, reduces restarts and often improves performance.

DATM greatly reduces restarts relative to MetaTM, and the remaining restarts are classified in Figure 10. Restarts due to transactions overwriting forwarded data (forward restarts are rare (less than 1%) in the STAMP programs. They are a high percentage of aborts in TxLinux, but that is mostly due to there being few aborts in TxLinux. Cascaded aborts are generally responsible for single-digit percentages of restarts, which is low considering that about 40% of conflicts in both vacation and labyrinth involve more than two transactions (complex conflicts). While counter-tt has 100% cascaded aborts, the abort rate is 0.39% (from Figure 9). All of these aborts involve more than two transactions.

Finally, the number of inconsistent reads and transitions into no-dep mode are very low. While these mechanisms are necessary for correct operation, they are rarely needed. Also, the number of broadcast writes is less than $1/1000$-th of one percent of transactional writes for all benchmarks. While broadcasting writes is necessary to preserve the ability to write back entire cache lines, it does not create excessive interconnect traffic.

C. Hardware constraints

Figure 11 shows the performance impact of various hardware constraints: cache line granularity, the ordering vector, and timestamp-ordered dependences (shown for bayes and

Name	*Description*
bayes	From STAMP [17], learns the structure of a Bayesian network, "-v32 -r384 -n2 -p20 -s1"
config, pmake	These benchmarks report transactions created in TxLinux, a Linux-variant operating system with several subsystems converted to use transactions for synchronization, instead of spin-locks [23], [26]. The workload involves several user-mode applications (configure, make) which are running on the transactional OS.
counter, counter-tt	A micro-benchmark where threads increment a single shared counter. counter-tt adds think-time, to simulate longer transactions.
genome	From STAMP, a gene-sequencing bioinformatics application, "-g 1024 -s16 -n 4000000"
kmeans	From STAMP, implements a K-means clustering algorithm, "-m40 -n40 -t0.05 -i random-n65536-d32-c16.txt"
labyrinth	From STAMP, models an engineering program which performs path-routing in a maze, "-i random-x48-y48-z3-n48.txt"
list	A micro-benchmark which manipulates a linked-list. On a traversal, a thread may search for a random node (60%), insert a node (20%) or delete a node (20%). The number of nodes is 8192, node traversals per thread is 512, and the number of threads is set to four times the processor count.
ssca2	From STAMP, a scientific application with different kernels operating on a multi-graph, " -s13 -i1.0 -u1.0 -l3 -p3"
vacation	From STAMP, models a multi-user database, "-t 20000 -n 10"

Fig. 7. Workloads used in DATM evaluation. TxLinux and list are kernel-mode transactions, while the other benchmarks run in user-mode. All benchmarks use a number of threads equal to the number of processors, unless noted otherwise.

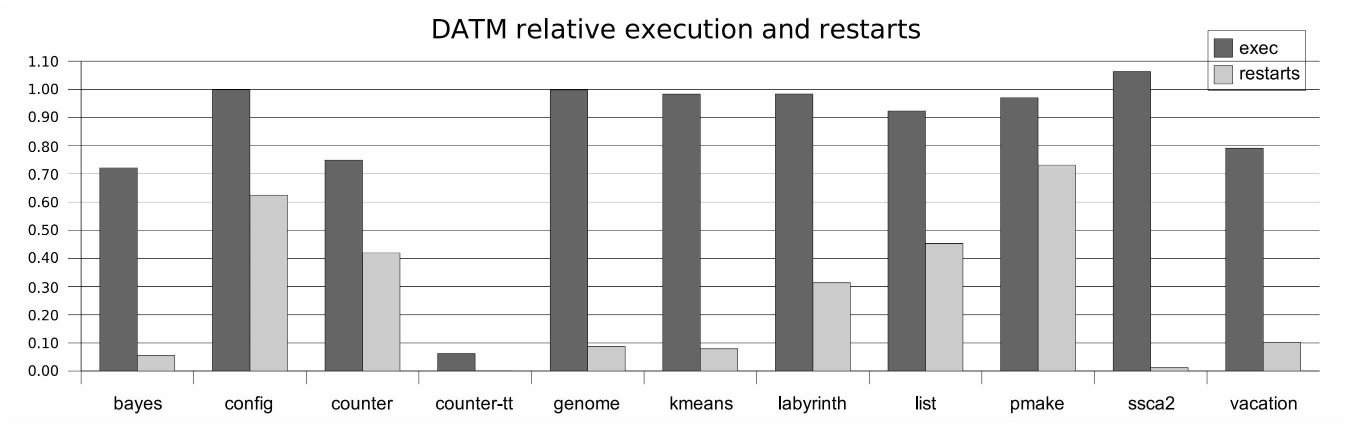


Fig. 8. Relative execution time and restarts per transaction in DATM, normalized to MetaTM. Lower is better.

vacation). By comparison to word-granularity implementations, managing dependences at cache line granularity reduces performance. Word-granularity requires extra state bits in the cache, but does not significantly increase bus traffic due to broadcast writes.

False cycles in the order vector reduce the performance of DATM. The average length of the order vector during transactions in bayes and vacation is approximately 6 (sampled at every dependence-causing memory operation) with maxima very close to the number of CPUs. Using timestamps to order dependences also reduces performance, but not as much as the order vector (e.g., bayes speedup goes from 39% to 14%).

D. Contention management

An attempt to create a dependence that would result in a cycle will cause DATM to invoke a contention management policy to resolve the conflict. DATM uses a novel dependence-aware contention management policy, which minimizes cascaded aborts by restarting the transaction with the fewest dependent transactions, resorting to timestamp when the number of dependents are equal. Figure 12 shows relative execution times for bayes and vacation using eruption, polka, and the dependence-aware contention management policies. It outperforms non-dependence-aware contention managers (including timestamp, which is not shown).

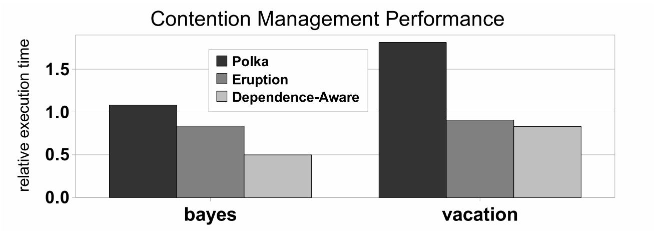


Fig. 12. Impact of contention management policies in the presence of DATM. Performance is normalized to the MetaTM performance.

VI. Related work

Larus and Rajwar provide a thorough reference on TM research through the beginning of summer 2006 [14].

Dependence-aware transactions detect conflicts in a way that is neither eager nor lazy [18], but rather combine strengths of both approaches. 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, which is absent with dependences. Since multiple transactions

benchmark	exec		tx	avg rst/tx		pct rst		avg bkcyc/tx	
bayes	0.0082	0.0059	762	13.9	0.8	9.38	5.45	27,283.48	759.82
config(8p)	3.5433	3.5369	4698136	0.1	0.1	1.91	1.86	1.31	0.45
counter	0.0948	0.0710	160000	9.4	4.0	59.59	76.83	519.17	119.42
counter-tt	3.0668	0.1892	16000	1056.3	0.1	99.99	0.39	264,060.82	0.12
genome	0.2122	0.2117	352376	0.1	0.1	0.21	0.15	0.99	0.06
kmeans	0.3005	0.2954	436986	1.1	0.1	11.11	6.33	58.39	0.91
labyrinth	0.0657	0.0646	128	88.5	27.8	36.77	29.74	140,085.71	41,521.21
list(8p)	0.3858	0.3552	78586	0.2	0.1	10.89	5.32	1.47	0.19
pmake(8p)	0.2604	0.2526	251844	0.2	0.1	3.93	3.80	4.46	3.07
ssca2	0.0075	0.0079	47304	0.1	0.001	0.15	0.10	28.42	0.05
vacation	0.0304	0.0248	20000	8.0	0.9	36.71	29.93	1,123.27	40.08

Fig. 9. Basic transactional characteristics of benchmarks running on on DATM and MetaTM. In cases where two numbers are present, MetaTM is the leftmost number, while DATM is the rightmost number. The "exec" metric is execution time in seconds (user time for STAMP and micro-benchmarks, and kernel time for TxLinux benchmarks), and the "tx" metric is the total number of transactions. The "avg rst/tx" metric is the average number of restarts per transaction, and the "pct rst" metric is the percentage of transactions that restart at least once. The "avg bkcyc/tx" metric is the average number of cycles spent backing off before restart per transaction. All data is for 16 CPUs, except TxLinux benchmarks `config` and `pmake`, which were run using 8 CPUs.

	WR deps	RW deps	forward restarts	cascade aborts	complex confl.	no-dep mode	incons. reads	broadcast writes
bayes	3.8%	8.5%	0.4%	0%	7.0%	1	3	1
config (8p)	0.3%	0.2%	19.7%	2.3%	0.3%	2	1	10,132
counter	90.0%	80.7%	0%	0%	90.8%	0	0	0
counter-tt	99.9%	0.3%	0%	100.0%	99.9%	0	0	0
genome	0.1%	0.1%	0%	14.2%	0.1%	0	1	104
kmeans	8.9%	6.0%	0%	3.1%	7.0%	0	0	40,723
labyrinth	32.0%	32.0%	0%	0.1%	39.0%	6	1	4
list	14.3%	3.8%	3.3%	0.2%	0%	3	0	86
pmake (8p)	0.5%	0.5%	12.9%	6.5%	0.6%	0	10	10,009
ssca2	0.1%	0.1%	0%	0%	0.1%	0	0	0
vacation	35.2%	7.9%	0.4%	3.5%	44.7%	6	34	143

Fig. 10. Basic dependence-related statistics. The first two columns show the percentage of transactions that were involved in a dependence of that type (W$\longrightarrow$W dependences formed between less than 0.2% of transactions for all workloads). The next three show the percentage of restarts that were due to forward restarts, cascading aborts, or complex conflicts. The next two columns provide an actual count of transactions that entered no-dep mode and experienced inconsistent reads. The last column shows the total broadcast writes for the workload.

that write the same memory cell cannot update the cell in place, DATM version management is lazy.

Aydonat and Abdelrahman [1] have (simultaneously with us) identified that current transactional memory implementations apply a stronger form of serializability than conflict serializability, thus reducing the amount of useful concurrency in the system. They have a software system which does not accept all conflict serializable schedules as DATM does. In particular, their implementation would not allow concurrent updates to a shared counter.

Write-shared data. One approach to the problem of write-shared data is to make it the responsibility of the programmer not to write-share data in the first place. Such systems usually provide at least some performance analysis tools [4] to help programmers identify data hotspots, leaving them with these alternatives: (1) Use a more sophisticated data structure (or adapt the existing one), to avoid the problem (e.g., use a vector of counters instead of a global single counter), (2) Eliminate a feature or reduce functionality (e.g., remove the counter feature), (3) Weaken the specifications (e.g., use cached, private copies of data at the cost of being stale), (4) Do nothing and live with the performance problem.

All these approaches are undesirable, and ultimately lead to some combination of greater programming effort, decreased maintainability, reduced functionality, and more bugs. These costs are a significant price to pay for higher concurrency.

TM programming model extensions. Several proposed extensions to the TM programming model can be used to achieve higher performance, including privatization [31], early release [30], escape actions [32], open and closed nesting [19], [20], Galois classes [13], transactional boosting [11] and abstract nested transactions [10].

These techniques all fundamentally affect the programming model, increase programmer effort, and increase program complexity as the price for better performance. They differ in their degree of applicability and the difficulty of reasoning involved, as well as the amount of additional compromises they force on their users. For example, using escape actions to implement a counter requires the programmer to also write a compensation block, which is a significant programmer burden. Moreover, semantics may be weakened when using this approach (e.g. a counter implemented this way is no longer monotonically increasing). In Galois and transactional boosting, the programmer needs to provide inverse operations for the concurrent data structures, which might be difficult (e.g., k-d tree), as well as define commutativity relationships between the various operations.

Thread-level speculation Designs for thread-level speculation [7], [25] are similar to DATM in their support for multiple versions of speculative data. In the TLS taxonomy, DATM merges its results with main memory lazily (via the CTM cache state). However, state management in DATM is much simpler than TLS. For example, of the five challenges to buffering state in TLS (including multiple speculative tasks per processor and multiple versions of a variable in a processor), DATM needs to deal with three of them (buffering and merging speculative state and multiple versions of the same variable at different processors). Current TM systems must deal with two of the challenges, buffering and merging speculative state.

TLS must squash speculation on dependence violation, and current designs tolerate some memory access conflicts. TLS tolerates a subset of the conflicts tolerated by DATM [25].

Databases. The dependence-aware model has some relation to multi-version concurrency control (MVCC). The DATM implementation keeps track of multiple versions of an object when it is being modified concurrently by many transactions. MVCC (also called time-domain addressing [24]) also tracks multiple versions of each object, so there are some common issues that both systems must deal with. However, a key difference between MVCC and DATM is that DATM is designed to specifically deal with hotspots, whereas hotspots are known to degrade performance of MVCC database systems. The techniques used by DATM (transactions dependences and forwarding data between transactions) are unique to DATM and not found in MVCC systems.

VII. Conclusion

Dependence-aware transactions increase throughput by enabling concurrent execution of transactions that would otherwise conflict due to updating shared data structures. This paper presents the design, and a prototype implementation of the first dependence-aware hardware transactional memory system. Experimental results from our prototype confirms the potential performance benefits of dependence-aware transactional memory as compared to traditional HTM implementations. DATM eliminates the need for programmers to resort to esoteric programming patterns or to extend the TM programming model. This performance improvement is achieved through mechanisms that are completely transparent to the programmer.

VIII. Acknowledgements

This research is supported by NSF CISE Research Infrastructure Grant EIA-0303609 and NSF Career Award 0644205. We thank Virtutech AB for their academic site license program.

References

[1] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT*, 2008.

[2] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the fast case common and the uncommon case simple in unbounded tm. In *ISCA*, 2007.

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

[4] H. Chafi, C. Minh, A. McDonald, B. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. Tape: A transactional application profiling environment. In *International Conference on Supercomputing*, 2005.

[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*, 2006.

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

[7] M. Garzarán, M. Prvulovic, J. Llabería, nals V. Vi L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM TACO*, 2(3), 2005.

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

[9] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[10] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT*, 2007.

[11] M. Herlihy and E. Koskinen. Transactional boosting. In *PPoPP*, 2008.

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

[13] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[14] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[15] P.S. Magnusson, M. Christianson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

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

[17] C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, 2007.

[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] J. E.B. Moss. *Nested transactions*. MIT, 1985.

[20] Y. Ni, V. Menon, A. Tabatabai, A. Hosking, R. Hudson, J. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

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

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

[23] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: Transactional memory for an operating system. In *ISCA*, 2007.

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

[25] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-level speculation on a CMP can be energy efficient. In *ICS*, 2005.

[26] C. Rossbach, O. Hofmann, D. Porter, H. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing transactional memory in an operating system. In *SOSP*, 2007.

[27] W. Scherer III and M. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.

[28] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA*, 2008.

[29] A. Shriraman, M. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA*, 2007.

[30] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *WTW*, 2006.

[31] M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization techniques for software transactional memory. In *PODC*, 2007.

[32] C. Zilles and L. Baugh. Extending hardware transactional memory. In *TRANSACT*, Jun 2006.