Integrating Caching and Prefetching Mechanisms in a Distributed Transactional Memory

Alokika Dash and Brian Demsky
A. Dash and B. Demsky are with the University of California, Irvine.

Abstract—We present a distributed transactional memory system that exploits a new opportunity to automatically hide network latency by speculatively prefetching and caching objects. The system includes an object caching framework, language extensions to support our approach, and symbolic prefetches. To our knowledge, this is the first prefetching approach that can prefetch objects whose addresses have not been computed or predicted.
Our approach makes aggressive use of both prefetching and caching of remote objects to hide network latency while relying on the transaction commit mechanism to preserve the simple transactional consistency model that we present to the developer. We have evaluated this approach on three distributed benchmarks, five scientific benchmarks, and several microbenchmarks. We have found that our approach enables our benchmark applications to effectively utilize multiple machines and benefit from prefetching and caching. We have observed a speedup of up to $7.26\times$ for distributed applications on our system using prefetching and caching and a speedup of up to $5.55\times$ for parallel applications on our system.

Index Terms—Distributed Shared Memory, Software Transactional Memory, Prefetching

1 Introduction

The growth of networking along with price decreases in hardware have led to the widespread adoption of distributed computing. Developing efficient software for distributed systems while simultaneously managing complexity can be challenging. For example, the underlying hardware often only supports communication between devices through network packets. As a result, developers of distributed applications must reason about communication patterns, write code to traverse and marshall possibly complex data structures into messages, write communication code to route these messages from producers to consumers, and write code to unmarshall these messages back into data structures.

Researchers have developed software distributed shared memories to provide developers with the illusion of a simple shared memory abstraction on distributed systems. A straightforward implementation of a distributed shared memory can provide developers with a simple memory model to program. However, accessing remote data in such implementations requires waiting for network communication and therefore is expensive. In response to this issue, researchers have developed distributed shared memory systems that achieve better performance by relaxing memory consistency guarantees. Developing software for relaxed memory consistency models can be challenging — the developer must read and understand complicated memory consistency properties to understand the possible behaviors of the program.

In recent years, researchers have explored transactional memory as a simpler concurrency primitive. Transactional constructs were researched to simplify software development by enabling developers to control concurrency without having to reason about potentially complex locking disciplines.

1.1 Basic Technical Approach

In this paper, we present a distributed transactional memory that presents a simple programming model to the developer. One of the primary challenges in designing distributed shared memory systems is hiding the latency of accessing remote objects. Previous work on distributed transactional memory primarily focused on providing transactional guarantees and largely overlooked a promising opportunity for utilizing the transaction commit mechanism to safely enable optimizations. Our approach caches remote objects and relies on the transaction commit checks to safely recover from mis-speculations.

Many traditional prefetching approaches have had limited success hiding the latency of remote object accesses in the distributed environment because they require the computation to first compute or accurately predict an object's address before prefetching that object. With our approach, a developer identifies objects that would benefit from prefetching and writes a prefetch hint that describes the paths through the heap that reach those objects even though their addresses are unknown. The runtime system can then often prefetch all of those objects in a single round-trip communication. This reduces the cost of remote object accesses. We use object versioning and unique object identifiers to track committed changes to objects and rely on the transaction commit checks to safely recover from mis-speculations.

1.2 Contributions

This paper makes the following contributions:

- **Distributed Transactional Memory:** It presents the first distributed transactional memory with both language and compiler support. In this model, the developer uses the `atomic` keyword to declare that a region of code should be executed with transactional semantics. In this context, transactional semantics means that the execution of the reads and writes in the transactions is consistent with some sequential ordering of the transactions.
- **Object Caching and Prefetching:** Caching and prefetching objects can potentially hide the latency of accessing remote objects. To address the possibility of accessing an old object version, our approach leverages transaction commit checks to ensure that a transaction only accessed the latest object versions.
- **Symbolic Prefetches:** Traditionally, prefetching a memory location requires that the program first computes or predicts the address of that memory location. Such prefetch strategies can perform poorly for traversals over remote, linked data structures, such as a linked list, as they require the program to incur the round-trip network latency when accessing each new element in the linked list. Our approach introduces symbolic prefetches: a symbolic prefetch specifies the object identifier of the first object to be prefetched followed by a list of field offsets or array indices that define a set of paths through the heap. These paths traverse the objects to be prefetched. The remote machine processes the symbolic expression and responds with a copy of the initial object and the objects along the paths specified by the symbolic expression that are in the remote machine's object store. The end result leverages data locality at the machine granularity to minimize communication rounds and thereby minimize delays due to network latency.
- **Optimized Cache Coherence:** Systems with multiple caches must ensure that data residing in the caches is consistent. This typically requires using an expensive cache coherency protocol. We instead use a set of techniques that trade rigorous consistency guarantees for performance. In this context, optimized coherency is safe because the commit process will detect and correct the occasional access to old object versions.

The remainder of this paper is structured as follows. Section 2 presents an example. Section 3 presents the programming model and locality analysis. Section 4 presents the runtime. Section 5 presents the prefetching mechanism. Section 6 presents an evaluation on several benchmarks. Section 7 discusses related work and we conclude in Section 8.

```
1  public class DistributedHashtable {
2    EntryList table[];
3
4    DistributedHashtable(int capacity) {
5      table=shared EntryList[capacity];
6    }
7
8    Object get(Object key) {
9      int index=hash(key, table.length);
10     prefetch(table[index].list.
11      next[0..2].key);
12    EntryList fl=table[index];
13    if (fl==null) return null;
14    DHashEntry ptr=fl.list;
15    for(;ptr!=null;ptr=ptr.next)
16      if (ptr.key.equals(key))
17        return ptr.value;
18    return null;
19   }
20  ...
21 }
22
23 class EntryList {
24   DEntry list;
25 }
26
27 class DEntry {
28   Object key;
29   Object value;
30   DEntry next;
31 }
```

Fig. 1. Distributed Hashtable

```
1 DistributedHashtable dht = ...;
2 atomic {
3   for(int i=0;i<a.length;i++)
4     b[i]=dht.get(a[i]);
5 }
```

Fig. 2. Example Transaction

2 Example

Figure 1 presents a distributed hash table example. The distributed hash table uses a table with an extra level of indirection to reduce conflicts on the array. The object initializer in lines 4 through 6 for the `DistributedHashtable` creates the top-level array. The allocation site contains the keyword `shared` to indicate that the array object is shared. Shared objects can be accessed by any thread on any machine. Our distributed system supports local objects that can only be accessed by a single thread. Our system assumes by default that allocation sites without the `shared` modifier allocate local objects. Each object has an *authoritative* copy that contains all committed changes and is stored permanently on the machine that allocated the object using the `shared new` allocation statement.

Lines 8 through 19 present the code for the `get` method, which looks up keys in the distributed hash table. Line 10 of the `get` method includes a symbolic prefetch annotation. A symbolic prefetch annotation consists of a starting object identifier followed by a symbolic expression that specifies a set of paths through the heap from the starting variable. For example, the symbolic prefetch expression `table[index].list.next[0..2].key` in line 10 specifies paths starting at the array referenced by `table` that traverse the `index` element of the array, the `list` field once, the `next` field between zero through two times, and finally the `key` field. The `[]` operator applied to an array specifies a range of indices to traverse. The `[]` operator applied to a field specifies how many times to traverse the field.

Figure 2 presents an example of a transaction that calls a `DistributedHashtable` instance's `get` method on the elements of the array `a`. The `atomic` keyword in line 2 declares that the enclosed block should be executed with transactional semantics. Note that our system statically imposes the constraint that shared objects may only be accessed inside transactions.

2.1 Program Execution

The `atomic` keyword in line 2 from Figure 2 causes the runtime system to execute the code block in lines 2 through 5 with transactional semantics. Our system maintains the invariant that if a variable is both used inside the current transaction and references a shared object, it points to the transaction's working copy during the duration of the transaction. To establish this invariant, the compiler generates code at the entrance of this atomic block that converts the object identifier stored in the `dht` variable into a reference to the transaction's working copy of the object. This code first checks to see if the transaction already contains a copy of the object, then checks to see if the authoritative copy resides on the local machine, next checks to see if the local machine has a cached copy of the object, and finally contacts the remote machine that holds the authoritative copy of the object to obtain a copy of the object. If the transaction has not already accessed the object, the runtime system makes a working copy of the object for the transaction and points the `dht` variable to this copy.

When the transaction completes, it calls the runtime to commit the transaction. The runtime sorts the objects into groups by the machine that holds the authoritative copy of the object. It then sends each group to the machine that holds the authoritative copies of the objects in that group. The current execution thread next serves as a coordinator in a two-phase commit protocol to commit the changes.

Each shared object contains a *version number*. The version number is incremented every time the committed (or authoritative) copy of the object is changed. In the first phase, each authoritative machine verifies that the transaction has only accessed the latest versions of the objects and votes to abort if the transaction accessed an old version of any object. If all authoritative machines vote to commit, the coordinator sends a commit command. If any machine votes to abort, the system must re-execute the transaction.

Our system also supports thread local objects. If a transaction modifies a local object, the compiled code makes a backup to enable restoration of the object in the event that the transaction aborts.

2.2 Object Prefetching

The example accesses objects that are not likely to be cached on the local machine. Our approach uses prefetching to hide the latency of these object accesses. Consider the path `table[index].list.next[0..2].key` that is represented by the symbolic prefetch expression in line 10 of Figure 1. The traditional prefetching approach would first prefetch `table`, next `table[index]`, and so on. This strategy requires six consecutive round-trip communications. Our approach sends a symbolic prefetch request for (1) the object identifier stored in `table` and (2) the paths defined by the symbolic expression `table[index].list.next[0..2].key`. If the remote machine contains all of the objects, all eight objects including the three key objects can be prefetched in a single round-trip communication.

3 Programming Model

We have developed several language extensions to a core subset of Java to support distributed transactional memory. These extensions let the developer declare objects as shared. They place the constraint on the developer that shared objects can only be accessed inside of transactions. The developer can then write transactional code for distributed nodes that accesses shared objects.

3.1 Java Extensions

Our extensions add the `atomic` keyword to declare that a block of code should have transactional semantics. This keyword can be applied to either (1) a method declaration to declare that the method should be executed inside a transaction or (2) a block of code enclosed by a pair of braces. We allow these constructs to be nested — the implementation simply ignores any transaction declaration that appears inside of another transaction declaration. The shared memory extensions are similar to those present in Titanium [1] although our implementation prohibits accessing shared objects outside of transactions.

Our extensions also add the `shared` keyword to the language. The `shared` keyword can be used as a modifier to the `new` allocation statement to declare that an object should be allocated in a shared memory region. Shared objects can only reference other shared objects. Our approach allows local objects to reference both shared and local objects. However, the developer must declare that a field in a local object references a shared object by using the `shared` keyword as a modifier to that field's declaration.

In general, methods are polymorphic in whether their parameter objects are shared. In some cases, the developer may desire a method to have different behavior depending on whether the parameter objects are shared objects. Our extensions support creating different method versions for shared and local objects — the developer designates the shared version with the `shared` keyword and the local version with the `local` keyword.

The extensions modify the `start` method to take a machine identifier that specifies which machine to start the thread on. The implementation contains a `join` method that waits for the completion of other threads.

3.2 Inference Algorithm

We use a flow-sensitive, data-flow–based inference algorithm to infer for each program point whether a variable references a shared object or a local object. We define $\mathcal{A} = \{$ *either*, *shared*, *local*, $\perp\}$ to be the set of abstract object states. We define $\mathcal{V}$ as the set of program variables. The data flow analysis computes the mapping $\mathcal{S} \subseteq \mathcal{V} \times \mathcal{A}$ from program variables to abstract object states. We use the notation $\mathtt{f_s}$ to denote a field `f` that has been declared as shared with the `shared` modifier. Figure 3 presents the lattice for the abstract object state domain.

Figure 4 presents the transfer functions for the data flow analysis. The analysis starts by analyzing the `main` function

$$\begin{tikzcd}
 & \text{either} \arrow[dl] \arrow[dr] & \\
\text{shared} \arrow[dr] & & \text{local} \arrow[dl] \\
 & \bot &
\end{tikzcd}$$

Fig. 3. Lattice for Analysis

st	kill	gen
$\mathtt{x} = \mathtt{shared\ new\ C}$	$\langle \mathtt{x}, * \rangle$	$\langle \mathtt{x}, shared \rangle$
$\mathtt{x} = \mathtt{new\ C}$	$\langle \mathtt{x}, * \rangle$	$\langle \mathtt{x}, local \rangle$
$\mathtt{x} = \mathtt{y}$	$\langle \mathtt{x}, * \rangle$	$\langle \mathtt{x}, \mathcal{S}(\mathtt{y}) \rangle$
$\mathtt{x} = \mathtt{null}$	$\langle \mathtt{x}, * \rangle$	$\langle \mathtt{x}, either \rangle$
$\mathtt{x} = \mathtt{y.f}$	$\langle \mathtt{x}, * \rangle$	$\langle \mathtt{x}, \mathcal{S}(\mathtt{y}) \rangle$
$\mathtt{x} = \mathtt{y.f}_{\mathtt{s}}$	$\langle \mathtt{x}, * \rangle$	$\langle \mathtt{x}, shared \rangle$
$\mathtt{x} = \mathtt{call\ m}(\mathtt{y}, ..., \mathtt{z})$	$\langle \mathtt{x}, * \rangle$	return value of m in the context $\mathcal{S}(\mathtt{y}), ...\mathcal{S}(\mathtt{z})$
other statements	—	—

Fig. 4. Transfer Function for Inference Analysis

in the non-atomic context with a local string array object as its parameter. The analysis initializes the parameter variables' abstract state from the method's calling context. The analysis proceeds using a standard forward-flow, fixed-point–based data-flow analysis.

When the analysis processes a call site for a method context that has not been analyzed, it enqueues that method context to be analyzed. The analysis then uses the *either* value for the abstract state of the return value until the analysis can determine the return value's actual abstract state. When the analysis updates the return value for a method context, it enqueues all callers of that context for re-analysis.

The inference algorithm uses the abstract object states to statically check several safety properties: (1) it ensures that the program does not attempt to store a reference to a local object in a shared object, (2) that the compiler can statically determine for each object access whether the object is shared, local, or null, (3) that the program does not attempt to store references to shared objects in a local field that has not been declared shared, (4) that native methods are not called inside of transactions[1], (5) that shared objects are not accessed outside of transactions, and (6) that shared objects are not passed into native methods.

It is of course possible to allow accesses to shared objects outside of transactions by sending messages for each object access. However, if code performs multiple accesses to shared objects, transactions are typically beneficial for performance. Transaction allow the system to speculatively access shared objects using locally cached objects, and then with a single communication round trip validate all of those object accesses. The system can then guarantee coherent accesses to all of the objects while incurring only a small fraction of the latency it takes to actually access those objects.

The compiler uses the analysis results to generate specialized versions of methods for each calling context. These specialized versions optimize field and array accesses depending on whether the object is local or shared and whether the method is invoked inside a transaction. Note that it is possible for a variable's abstract state to be *either* if the variable is always null in that context. In this case, the compiler simply generates code for local accesses to give the appropriate runtime error behavior. If the compiler cannot determine whether an operation is performed on a local or shared object, it generates a compilation error.

We note that there is the potential for the analysis to generate a large number of implementations for a method with many parameters. We expect that this would rarely occur. However, in the event that a large number are generated, a production compiler could simply generate a generic version of those methods that dynamically checks whether an object is shared, dynamically checks the safety properties, and selects the appropriate implementation at runtime.

4 Runtime Overview of Distributed Transactional Memory

Figure 5 presents an overview of the runtime. The runtime is object-based — data is accessed and committed at the granularity of objects. When a shared object is allocated, it is assigned a globally unique object identifier. The object identifier is then used to reference and access the object. We statically partition the object identifiers between nodes so that each node can assign unique identifiers. The runtime system determines the location of an object directly from its object identifier.

Our system implements optimistic concurrency control using a version-based strategy. Each shared object contains a version number — the version number is incremented when a transaction commits a write to the object. The implementation uses the version numbers to determine whether it is safe to commit a transaction. If a transaction accesses an old version of any object, the transaction must be aborted. The commit protocol has two phases: the first phase verifies that the transaction operated on the latest versions of all objects and the second phase commits the changes.

The implementation maintains the following types of object copies:

- **Authoritative Copy:** The authoritative copy contains all updates that have been committed to the object. Each object has exactly one authoritative copy. The machine in which the authoritative copy resides is fixed when the object is allocated. The location of an object's authoritative copy is encoded in its object identifier.
- **Cached Copy:** Cached copies are used to hide the latency of remote object accesses. When a transaction accesses a cached copy of an object, the runtime makes a transaction local copy of the object for that transaction. The cached copy can be stale — if a transaction accesses a stale object, the transaction will abort.

1. This constraint prohibits I/O calls inside transactions. We make an exception for a debugging print statement and known side-effect free native methods such as floating point operations.

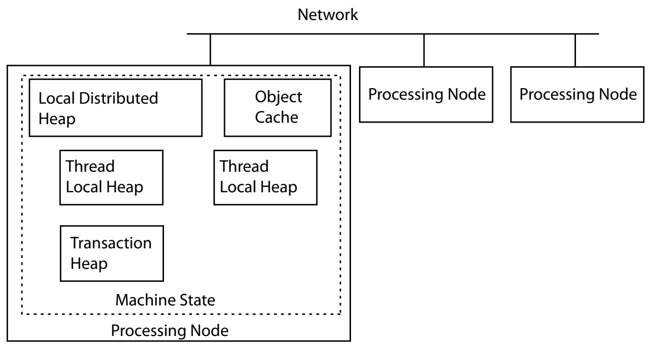

Fig. 5. Overview of System Architecture

- **Transaction Local Copy:** When a transaction accesses a shared object, a transaction local copy is made for that transaction. The transaction performs reads and writes on this local copy. When the transaction commits, any updates to the local copy are copied to the authoritative copy. It is possible for the local copy to be stale in which case the transaction will abort.

4.1 Memory Architecture

We next discuss the shared memory architecture of our system. The expanded processing node in Figure 5 presents the major components in our distributed transactional memory system. Each processing node contains the following state:

- **Local Distributed Heap:** The shared memory is partitioned across all processing nodes. Each node stores a disjoint subset of the authoritative copies of distributed objects in its local distributed heap. The local distributed heap stores the most recent committed state for each shared object whose authoritative copy resides on the local machine. Each local distributed heap contains a hash table that maps object identifiers to the object's location in the local distributed heap.
- **Thread Local Heap:** In addition to shared objects, objects can be allocated in thread local heaps. There is one thread local heap for each application thread. Thread local objects can be accessed at any time during the computation by the thread that owns the object.
- **Transaction Heap:** There is a transaction heap for each transaction. The transaction heap stores the transaction local copy of any shared object that it has accessed. Each transaction heap contains a hash table that maps the object identifiers that the transaction has accessed to the location of the transaction local copy in the transaction heap.
- **Object Cache:** Each processing node has an object cache that is used to cache objects and to store prefetched objects. Each object cache contains a hash table that maps the object identifiers of the objects in the cache to the object's location in the cache.

4.2 Accessing Objects

Our system uses a partitioned global address space (PGAS) programming model [1]–[3]. Recall that our system contains two classes of objects: local objects and shared objects. Accessing a local object outside of a transaction and reading a local object inside a transaction only require a simple pointer dereference. Writing to a local object inside a transaction requires a write barrier that ensures that a backup copy of the object exists. If the transaction is aborted, the object is restored from the backup copy.

Shared objects can only be accessed inside of a transaction. When code inside a transaction attempts to lookup an object identifier to obtain a pointer to a transaction local copy of the object, the runtime system attempts to locate the object in the following places:

1) The system first checks to see if the object is already in the transaction heap.
2) If the object is located on the local machine, the system looks up the object in the local distributed heap.
3) If the object is located on a remote machine, the system next checks the object cache on the local machine.
4) Otherwise, the system sends a request for the object to the remote machine.

Note that primitive field or primitive array element accesses do not incur these extra overheads as the code already has a reference to the transaction local copy of the object. We expect that for most applications, the majority of accesses to reference fields or reference array elements will access objects that the transaction has already read. This code is inlined and the common case of locating the transaction local copy of an object involves approximately ten x86 instructions: a bit mask and a shift operation to compute the hash, an address computation operation, a memory dereference to lookup the object identifier, a comparison to verify the identifier, and a memory dereference to obtain the transaction local copy's location.

The compiler generates write barriers that mark shared objects as dirty when they are written to[2]. The runtime uses a shared object's dirty status to determine whether the commit must update the authoritative copy of the object.

4.3 Commit Process

We next describe the operation of the transaction commit. When a transaction has completed execution, it calls the transaction commit method. The commit method begins by sorting shared objects in the transaction heap into groups based on the machine that holds the authoritative copy of the object. For each machine, the commit method groups the shared objects based upon whether they have been written to or simply read from. The commit operates in two phases: the first phase verifies that the transaction operated only on the latest versions of objects and the second phase commits the changes. We next describe how the algorithm processes each category of shared object:

2. Each object contains a dirty flag, and the write barrier marks the object as dirty by setting the object's dirty flag.

– **Clean Objects:** For clean objects, the transaction commit verifies that the transaction read the latest version. The transaction coordinator sends the object's version number to the machine with the authoritative copy. That machine acquires a read lock on the object and compares versions. If the versions do not match the machine releases the object locks and votes to abort the transaction. Otherwise, the locks are held until the transaction commits.
– **Dirty Objects:** The transaction commit must copy the updates made by the transaction from the dirty objects to the authoritative copies of those objects. The system transfers a copy of the dirty object along with its version number to the machine holding the authoritative copy. The remote machine then acquires a write lock on the authoritative copy and compares versions. If the versions do not match, it votes to abort the transaction. If the transaction coordinator responds with a commit command, the changes are copied from the dirty copy to the authoritative copy and the object lock is released. If the coordinator responds with an abort command, the lock is simply released without changing the authoritative copies.

If all authoritative machines respond that all version numbers match, the transaction coordinator will decide to commit the transaction and transmit commit commands to all participants. If any authoritative machine responds with an abort request, the transaction coordinator will decide to abort and transmit abort commands to all participants. If any authoritative machine cannot immediately lock an object, the coordinator will abort the commit process to avoid deadlock and retry the commit process.

Code inside a transaction can also modify thread local objects and local variables. When a transaction begins, it executes compiler-inserted code that makes a copy of all live local variables. Whenever a transaction writes to a local object, the compiler-inserted code first checks if there is a copy of the object's state and then makes a copy if necessary. If the transaction is aborted, the generated code restores the local variables and uses the local object copies to revert the thread local objects back to their states at the beginning of the transaction.

4.4 Sandboxing

During a transaction, the execution can potentially read inconsistent versions of objects. While such executions will eventually abort during the commit process, reading inconsistent values can cause even correct code to potentially loop, throw an error, or run out of memory before the transaction aborts. Therefore, if the execution of a transaction throws an exception, the runtime system verifies that the transaction read consistent versions of the objects before propagating the exception. If an exception occurs, our system checks that the transaction has only accessed the latest versions of objects. If the transaction has accessed stale objects, the transaction is aborted. If the transaction has only accessed the most recent versions of objects, the exception is propagated. Similarly, there is the potential for looping due to reading inconsistent values. To prevent looping, our system periodically validates the read sets. If the object versions are consistent, the execution will continue, otherwise the transaction is aborted. Our system also validates the read set after a transaction's allocations have exceeded a threshold. We use an adaptive strategy that lowers the validation limits if a given transaction has failed a previous validation.

4.5 Compilation

Inside a transaction, our compiler maintains the invariant that if a variable both references a shared object and can potentially be accessed inside the current transaction, the variable points to a transaction local copy of the shared object. This invariant makes subsequent reads from primitive fields of shared objects as inexpensive as reading a field of a local object. The compiler maintains the invariant that variables that reference shared objects outside of a transaction store object identifiers. Our approach uses a simple dataflow analysis to determine whether a variable that references a shared object is accessed inside a transaction. The compiler then inserts, as necessary, code to convert object identifiers into references to transaction local copies and code to convert references to transaction local copies back into object identifiers.

4.6 Optimized Cache Coherency

While object caching and prefetching have the potential to improve performance by hiding the latency of remote reads for shared objects, they can increase the likelihood that transactions may abort due to reading stale data from the object cache. The obvious approach, a cache coherency protocol, for addressing this issue introduces a number of overheads. We have introduced several new mechanisms that are collectively designed to provide optimized cache coherence. These mechanisms do not guarantee cache coherence, they merely attempt to minimize the likelihood that cache reads return old object versions.

We use a combination of two techniques to evict old versions of cached objects. The first technique is designed for small scale deployments on a LAN. This technique uses unreliable UDP broadcast to send a small invalidation message when a transaction commits. This invalidation message lists the objects that the transaction modified. The implementation does not guarantee that the invalidation messages will arrive and does not wait for the messages to be processed. The second technique evicts the oldest objects in the cache whenever the cache needs more space. We expect that larger scale deployments of our approach would require different techniques for optimized cache coherence. Techniques for very large deployments could include profiling to determine at what points objects of a given type should be evicted.

Our implementation uses information from local transactions to update the object cache. Whenever a local transaction commits, it updates the local cache with the latest versions of any remote objects the transaction modified.

If a transaction aborts, the implementation learns information about the objects it is likely to access. The implementation can use this information to minimize the number of remote object requests that must be made when retrying the transaction. When transactions abort, the remote machines in our implementation send the latest versions of any stale objects that the aborted transaction accessed along with their abort response. These objects are then placed in the object cache and the transaction is retried.

It is important to note that although our approach only maintains optimized cache coherence, we preserve the correct execution semantics by detecting and correcting any stale object accesses in the transaction commit process.

4.7 Scalability

Our core implementation does not introduce scalability limitations beyond those inherent in the application. The implementation does not update any global internal structures. If an application runs on m groups of n machines and the computation on each group only accesses data in the same group, our implementation will not send messages between groups and therefore should scale perfectly as m increases. Our extension for object invalidation does violate this property, but sends very little data (8 bytes per modified object) and therefore even it would likely scale to reasonably large clusters.

4.8 Design Rationale

The round-trip network latency on a gigabit LAN between commodity workstations is approximately 100 μS. On a modern 3 GHz processor, this corresponds to waiting 300,000 clock cycles. Therefore, the guiding principle for our design is to avoid waiting on network responses whenever possible.

The benefits of detecting and resolving conflicts early is an open question for transactional memory systems for single machines [4]. In the distributed context, eager conflict detection is significantly more expensive as it requires waiting for network communications to multiple systems for the first write to each object. We therefore detect conflicts when transactions commit. We do not implement explicit contention management, instead we simply rely on lazy validation. Other researchers have found that lazy validation serves as a form of contention management [4]. For workloads with low contention , the choice of contention management has little effect. For workloads with high contention, the contention manager is likely to abort a transaction that might commit in favor of a transaction that will later be aborted. Our simulations show that lazy validation has similar performance to the karma contention manager and significantly better performance than either the attack or polite contention manager.

One potential concern is the possibility that transactions may abort repeatedly. Our system implements an optimization that accelerates aborted transactions — when a transaction aborts at commit, the system gains knowledge of the objects that transaction is likely to access. The remote systems therefore send fresh copies of any stale objects that caused the previous transaction attempt to abort. The re-execution of the transaction is likely to only access cached objects and therefore should execute relatively quickly.

Our system implements lazy validation of reads. While eager validation can avoid wasted effort, in the distributed context it requires waiting on network communication. As mentioned in Section 4.4, we use sandboxing to avoid the correctness issues associated with lazy validation.

Lazy validation approaches are known to be less prone to livelock because transactions only hold object locks for a brief period of time during the commit process [4]. With the exception of livelock when acquiring object locks to commit a transaction, transactions can only be aborted if another transaction has committed. To ensure with high probability that livelock does not occur, we use an exponential randomized abort-time backoff algorithm when a transaction cannot acquire all object locks but the versions of all objects in the readset are current. While our system ensures the absence of live lock with high probability, starvation of a transaction can occur if a series of transactions repeatedly commit with it. It is possible to extend the system to prevent starvation by using an abort retry threshold that when a transaction reaches it causes the transaction to switch to a special safe mode. This safe mode would assign the transaction a sequence number and the transaction would acquire locks when the objects are first accessed. If two such transactions both attempt to acquire the same lock, the one with the earliest sequence number would win.

5 Symbolic Prefetching

Our approach to distributed transactional memory creates a new opportunity to safely speculatively prefetch and cache remote objects without concern for memory coherency — the commit process ensures that transactions only access the latest object versions. Many traditional address-based prefetching approaches were primarily designed to hide the latency of accessing local memory — such prefetching incurs large latencies when accessing linked data structures because the computation must wait to compute an object's address before prefetching the object. For linked data structures, this requires waiting for a round-trip communication for each object to be prefetched.

We introduce a new approach to prefetching objects in the distributed environment that leverages the computational capabilities of the remote processors. Our approach communicates symbolic expressions that describe paths through the heap that traverse the objects to be prefetched. We next describe the runtime mechanism that enables our implementation to efficiently prefetch complex linked-data structures from remote machines.

5.1 Runtime Mechanism

Symbolic prefetch expressions have the form: prefetch expression := base object identifier(.field[$i_0..i_1$] | .arrayexpr)* where arrayexpr := array[$j_0..j_1$, stride] | arrayexpr [$j_0..j_1$, stride]. The base object identifier component of the symbolic prefetch gives the object identifier of the first object in the path. The list of fields and arrays describe a path through heap from the first object. Each field contains two indices i_0 and i_1 that specify to traverse the field between i_0 and i_1 times. An n-dimensional array contain n triples consisting of indices j_0 and j_1 and a stride; a triple specifies that the path traverses the range of indices between j_0 through j_1 with the given stride.

We next consider the following example code segment:

```
LinkedList search(Object key) {
 for(LinkedList ptr=head;ptr!=null&&
    !ptr.key.equals(key))
    ptr=ptr.next;
 return ptr;
}
```

Without prefetching, completely searching a remote linked list of length n requires n consecutive round-trip message exchanges. If we add a prefetch for the expression `ptr.next[1..5,1].key` after line 1, the runtime will have prefetch requests in flight for the next linked list node and the subsequent four nodes that follow that node along with their associated key objects. The example symbolic prefetch enables the `search` method to potentially execute ten times faster by prefetching five `LinkedList` objects and their corresponding keys in a single round trip. Longer symbolic expressions can further increase the potential speedup. Note that while prefetching objects for five loop iterations ahead may not be sufficient to hide all of the latency of accessing remote objects, the latency of the single round-trip communication is now divided over the ten objects that have prefetch requests in flight. Therefore, accessing each object incurs an effective latency of only 10% of the actual network latency.

Our symbolic expression implementation contains the following key components:

1) **Prefetch Calls:** Prefetching begins with a prefetch call from the application. Our implementation supports issuing several symbolic expression prefetches with a single prefetch call. The prefetch takes as input the number of symbolic expression prefetches, the length of each prefetch, and an array of 16-bit unsigned integers that stores a sequence of the combination of field offsets and array indices. The runtime system differentiates between field offsets and array indices based on the type of the previous object in the path. The prefetch method places the prefetch request in the prefetch queue and returns immediately to the caller. A thread in the local runtime processes prefetch requests from the queue.
2) **Local Processing:** In many cases, the local distributed heap and object cache may already contain many of the objects in the prefetch request. The runtime system next processes as much of the prefetch request as possible locally before sending the request to the remote machines. The local processing starts by looking up the object identifier component of the prefetch request in both the local distributed heap and the object cache. If the object is found locally, the local runtime system uses the field offset (or array index) to look up the object identifier of the next object in the path and remove the first offset value from the symbolic expression. The runtime repeats this procedure to process the components of the prefetch request that are available locally. The runtime then prunes the local component from the prefetch request to generate a new prefetch request with the first non-locally available object as its base.
3) **Sorting and Combining:** The runtime finally groups the prefetch requests by the base object identifier's authoritative machine. We note that it may become apparent at runtime that a prefetch request is redundant. Consider the two prefetch requests `a.f.g` and `b.f.g.h`. If at runtime both the expressions `a` and `b` reference the same object, the set of objects described by the prefetch request `a.f.g` is a subset of the set of objects described by the prefetch request `b.f.g.h`. When the runtime adds a new request to a group, if a request is subsumed by a second request the runtime drops the subsumed request.
4) **Transmitting Request:** The local machine next sends the prefetch requests to the remote machines. Each request contains the machine identifier that should receive the response.
5) **Remote Processing:** When the remote machine receives a prefetch request it begins with the object identifier. It processes an object identifier by looking up the object identifier first in its local distributed heap and then (optionally) if necessary in its object cache. Once it locates the object, it looks up the next object identifier by using the field offset or array index from the prefetch expression. It repeats this process until either it has served the complete request or it cannot locate a local copy of the object. It sends copies of the objects to the machine that originally initiated the prefetch request.
   If the remote machine does not have a copy of an object specified by the symbolic prefetch, it then forwards any remaining part of the prefetch request to the next machine with the machine identifier for the machine that made the original request.[3]
6) **Receiving Response:** When the local machine receives a response message, it adds the copies of the objects from the response message to its local object cache.

6 Evaluation

We ran our benchmarks on a cluster of 8 identical 3.06 GHz Intel Xeon servers running Linux version 2.6.25 and connected with gigabit Ethernet. We have implemented the distributed transactional memory, symbolic prefetching, and the language extensions. We present results for three distributed benchmarks, five scientific benchmarks, and microbenchmarks. For all benchmarks we insert symbolic prefetches where needed. Our implementation contains over 72,000 lines of C and Java code and is available for download at `http://demsky.eecs.uci.edu/software.php`. We report results for: *base*, transactional versions without caching or prefetching; *caching*, transactional versions with caching enabled; and *prefetch*, transactional versions with both caching and prefetching enabled. For the scientific benchmarks, we report results for *1J*, single-threaded non-transactional Java implementations compiled into C code. For the distributed benchmarks, we report results for *Java* for 1, 2, 4 and 8 threads that are hand-developed, non-transactional distributed versions compiled into C code. We report numbers in seconds that are

3. We need to forward because after a machine processes the prefetch request, it could contain references to still more remote objects.

averaged over ten executions for 1, 2, 4, and 8 nodes with one thread running per node.

6.1 Distributed Spam Filter

The distributed spam filter benchmark is a collaborative spam filter that identifies spam using user feedback. It is based on the Spamato spam filter project and contains 2,639 lines of code [5]. In the original version, a collection of spam filters communicates information to a centralized server. Our implementation replaces the centralized server with distributed data structures.

Spam Filter	Base	Caching	Prefetch
1	1.52s	—	—
2	12.77s	4.16s	2.85s
4	16.88s	5.02s	3.90s
8	21.53s	6.57s	5.29s

Fig. 6. Spam Filter Results

When the spam filter receives an email, it calculates a set of MD5-hash based signatures for that message. It generates Ephemeral hash-based signatures for the text parts of a message and Whiplash URL-based signatures for the URLs in the message. It then looks up those signatures in a distributed hash table that maps a signature to the associated spam statistics. The spam statistics are generated from collaborative user feedback. The spam filter uses those statistics to estimate whether an email is spam. If the user corrects the spam filter's categorization of an email, it updates the spam statistics for all of the signatures in that email.

Our workload emails are automatically generated from a large corpus that includes spam words. Each email in the automatically generated set is pre-identified as spam or legitimate based on whether it includes text from the spam corpus. Figure 6 presents the results for the distributed spam filter benchmark. Our workload presents each spam filter client with 1,000 emails to simulate real deployments. In each iteration, the synthetic workload randomly picks an email and presents it to the spam filter. The workload then corrects the spam filter using the emails pre-identification. The correction is noisy to simulate user errors — with a small probability the workload will tell the spam filter the wrong identification.

Our workload holds the work constant per client machine. As a result the total amount of work increases as we add more clients resulting in an increase in execution time as shown in Figure 6. There are two primary causes of this increase: (1) the hash table is more likely to contain the hash signature and therefore lookups access more objects and (2) a larger percentage of the objects are remote. We show the results for caching here so readers may quantify the benefits of caching verses prefetching. We see 4.07× speedup for our 8-threaded prefetching version relative to the 8-threaded base version and 1.24× speedup for our 8-threaded prefetching version relative to the caching version. Caching hides 79% of the remote reads for the 8-threaded version. Prefetching then hides 75% of the remaining remote reads for the 8-threaded version. We observe a benefit of up to 307% due to prefetching and caching relative to our base version. Figure 9 presents the abort rate for transactions running on multiple clients for this benchmark. Up to 10% of transactions abort due to conflicts. We do not have a Java version of this benchmark to compare to.

6.2 Distributed Multiplayer Game

The multiplayer game benchmark is an interactive game where players play the roles of tree planters and lumberjacks. The base version contains 1,416 lines of code. Each client connects to the server hosting the game map. A player can either be a planter or a lumberjack. The planters are responsible for planting trees in a block of land while lumberjacks cut trees. Both the planters and lumberjack choose a location in the map to either plant a tree or cut one down while taking the shortest path to the destination. The clients use the A* graph search algorithm to plan routes. The game is played in rounds and in each round, the player either plants a tree, cuts a tree, or moves one step on the map. There is contention in this benchmark: players change the map by planting or removing trees. If a player accessed the part of the map updated by another player, the distributed transactional memory version aborts the transaction surrounding that move. The reference Java version only recomputes a player's move if a change made by a second player makes the first player's move illegal.

Figure 7 presents results for the multiplayer gaming benchmark. The game is played on a map of size 400×100 for 512 rounds. Like the previous benchmark we held the work constant per client machine and therefore the total amount of work increases as we add more clients. For this benchmark, perfect scaling occurs when the execution time holds constant as the number of machines increases.

Multiplayer Game	Java	Base	Caching	Prefetch
1	46.78s	7.74s	—	—
2	51.99s	9.59s	9.48s	9.12s
4	71.54s	11.79s	10.95s	10.60s
8	97.22s	16.09s	13.69s	13.39s

Fig. 7. Multiplayer Game Results

Our base version is faster than the Java version because of the way the A* algorithm accesses the map. In the Java version, the server sends the map at the beginning of each round to make the search algorithm code manageable. The distributed transactional memory version only transfers the small parts of the map that the A* algorithm actually needs. We see a 7.26× speedup for our 8-threaded prefetching version relative to the 8-threaded Java version. Caching hides 71% of the remote reads for the 8-threaded version. Prefetching then hides 22% of the remaining remote reads for the 8-threaded version. We observe a benefit of up to 20% relative to the base version from prefetching and caching relative to our base version. Prefetching provides little benefits beyond caching because caching eliminates most of the remote object accesses. Figure 9 presents the abort rate for transactions running on multiple clients. Up to 3% of transactions abort due to conflicts.

6.3 Distributed Lookup

The Distributed Lookup benchmark provides an object-based lookup service. The base version contains 311 lines of code. In this model the objects are stored in a shared hash map. Clients perform search on the shared hash map and occasionally update the key-value pairs in the hash table. A client randomly generates the keys for searching. We perform 1,000 transactions on each client where each transaction performs 10 lookup or update operations with a 4% probability of updating a key-value pair. The initial capacity of the shared hash table is 100 keys. The number of entries in the table increase as remote transactions begin to update the table.

Figure 8 presents results for the LookUp service benchmark. Note that the number of lookups per client machine is held constant and the total amount of work performed increases as we add more machines. Therefore, for this benchmark perfect scaling occurs when the times stay constant. We see a 1.77× or 77% speedup for our 8-threaded caching version relative to the 8-threaded Java version. Caching hides 94% of the remote reads for the 8-threaded version eliminating nearly all remote reads. Prefetching then hides 3% of the remaining remote reads for the 8-threaded version. Because caching eliminates nearly all of the remote reads, we observe no benefit from prefetching relative to caching. We observe a benefit of up to 435% due to caching relative to our base version. Figure 9 presents the abort rate for transactions running on multiple clients. Up to 11% of transactions abort due to conflicts.

LookUpService	Java	Base	Caching	Prefetch
1	2.89s	0.52s	—	—
2	3.75s	6.21s	1.16s	1.27s
4	3.85s	7.09s	1.45s	1.59s
8	4.18s	9.26s	2.36s	2.56s

Fig. 8. LookUpService Results

6.4 Scientific Benchmarks

Our scientific benchmarks are shared memory parallel benchmarks that have less data contention. Figure 10 presents the results for all of the scientific benchmarks. Note that caching provides no benefit for the scientific benchmarks as different transactions rarely access the same data and the transaction cache already caches multiple uses by the same transaction. We note that as these benchmarks were ported from scientific benchmark suites, they use barriers to separate memory accesses that could potentially cause transactions in different threads to conflict. The transactional memory mechanism is still valuable in this context as it allows our system to correctly speculate that cached copies of objects can be used instead of incurring the overhead to read remote objects. We note that even in the absence of contention that transactions do occasionally abort due to delays in receiving or processing cache invalidation messages.

2D Convolution The 2D Convolution benchmark computes the application of a mask to a 2D image. The base version contains 988 lines of code. The output and input matrices are shared objects in our experiment with size of 10,000×1,000. We use a Gaussian convolution mask of size 13×13 for our experiments. We inserted a manual prefetch for the first 32 input and output image objects before the beginning of convolution computation. A manual prefetch in the out loop prefetches batches of 32 objects every 32nd iteration. We find that our 8-threaded prefetching version provides a speedup of 5.55× over the single-threaded Java version. Prefetching hides 99.8% of the remote reads for the 8-threaded version while caching eliminates only 0.1% of the remote reads. We observe a benefit of up to 7% due to prefetching relative to our base version because this benchmark accesses a small number of remote objects.

Matrix Multiply The matrix multiplication benchmark implements the standard matrix multiplication algorithm for matrix A and matrix B to get the product matrix C. Our benchmark computes fifty 650 × 650 product matrices. The computation of the product matrix is partitioned over multiple threads. We insert manual prefetches to obtain the entire matrix B for the first 8 matrices followed by the first 16 one-dimensional arrays from the matrices A and C. Inside the main loop, we inserted manual prefetches that obtain the remaining one-dimensional arrays from the matrices A and C in batches of 16. We observe that our 8-threaded prefetching version provides a speedup of 5.14× over the single-threaded Java version. Prefetching hides 90% of the remote reads for the 8-threaded version while caching alone eliminates only 0.05% of the remote reads. We observe a benefit of up to 44% due to prefetching relative to our base version.

2DFFT The 2DFFT benchmark is a two-dimensional fast Fourier transformation. The base version contains 821 lines of code. The algorithm was taken from *Digital Signal Processing* by Lyon and Rao. We set the matrix dimensions to 1,500×1,500 and we compute the fast Fourier transformation for five matrices. The computation performs a set of 1D fast Fourier transforms in parallel, a serial transpose, and then a second set of 1D fast Fourier transforms in parallel. We inserted a prefetch for the first 16 real and imaginary objects in the complex number array. The loop contains a second prefetch for a batch of 16 objects every 16th iteration before the 1D fast Fourier transforms. We observe that our 8-threaded prefetching version provides a speedup of 2.10× over the single-threaded Java version. The speedup was limited as the transpose operation is performed serially and the benchmark requires moving a large amount of data across the network relative to the amount of computation that is performed. Prefetching hides 97% of the remote reads for the 8-threaded version while caching alone eliminates only 0.3% of the remote reads. We observe a benefit of up to 10% due to prefetching relative to our base version.

Molecular Dynamics Moldyn, a molecular dynamics benchmark, was taken from the Java Grande benchmark suite [6]. The base version contains 1,172 lines of code. It is a N-body simulation of particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. We used 8,788 particles and 50 iterations. This

	Spam Filter		Game		SOR		LookUp	
Thds	Base	Prefetch	Base	Prefetch	Base	Prefetch	Base	Prefetch
2	4.45%	7.05%	0.19%	0.19%	0.00%	0.00%	0.00%	0.00%
4	7.90%	8.54%	0.19%	0.19%	0.49%	0.49%	2.52%	4.66%
8	9.37%	9.48%	2.46%	0.96%	0.73%	0.73%	7.90%	11.40%

Fig. 9. Abort rate

	2D Conv			Matrix Multiply			2DFFT			Moldyn			SOR		
	Base	Cache	P	Base	Cache	P	Base	Cache	P	Base	Cache	P	Base	Cache	P
1J	34.20s	—	—	96.00s	—	—	16.29s	—	—	103.10s	—	—	239.05s	—	—
1	36.76s	—	—	96.88s	—	—	23.63s	—	—	113.91s	—	—	678.35	—	—
2	21.01s	21.15s	19.71s	58.37s	62.31s	51.65s	16.31s	16.65s	14.44s	69.71s	71.16s	69.04s	347.52s	346.50s	345.35s
4	11.32s	11.39s	10.65s	39.18s	35.85s	28.93s	10.91s	11.25s	9.54s	34.82s	33.05s	32.93s	186.44s	185.60s	185.28s
8	6.52s	6.53s	6.16s	26.81s	23.69s	18.66s	8.49s	8.37s	7.75s	19.99s	19.36s	19.23s	105.93s	104.39s	104.56s

Fig. 10. Scientific Benchmark Results (P = Prefetch)

benchmark contains very limited opportunities for prefetching. We only prefetch shared elements for particle generation at the beginning before the molecular dynamics simulation starts. We observe that our 8-threaded prefetching version provides a speedup of 5.36× over the single-threaded Java version. Caching hides 73% of the remote reads for the 8-threaded version while prefetching hides only 1.3% of the remaining remote reads.

SOR The SOR benchmark was taken from the Java Grande benchmark suite [6]. The base version contains 680 lines of code. It performs 200 iterations of an over-relaxation algorithm on a 8,000×8,000 grid. We observe that the 8-threaded prefetching version provides a speedup of 2.29× over the single-threaded Java version. The one machine distributed transactional version is slower than the single-threaded Java version because each node must locally copy many large array objects to implement transactions. This large overhead means that although the benchmark scales extremely well, the 8-threaded version is only a little over two times faster than the single-threaded Java version, Figure 9 presents the abort rate for transactions running on multiple nodes. Prefetching and caching yield no performance benefits because this benchmark accesses a small number of remote objects. Caching hides 78% of the remote reads for the 8-threaded version while prefetching does not hide any additional remote reads.
Figure 11 presents the speedups for all scientific benchmarks relative to single-threaded Java version.

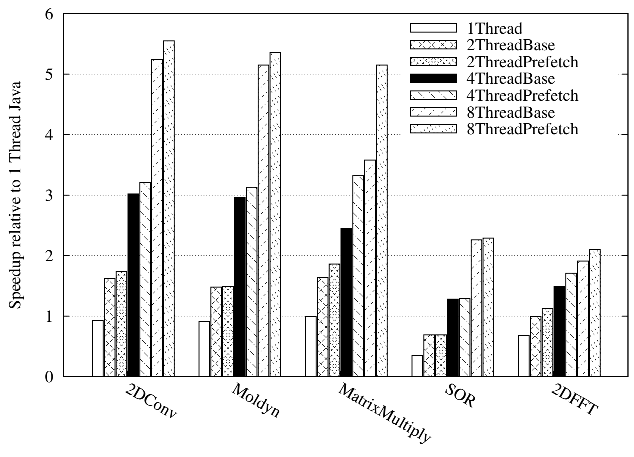

Fig. 11. Scientific Benchmark Speedups (Higher is Better)

6.5 Microbenchmarks

We present results from a three-dimensional array traversal microbenchmark to measure the performance gains from prefetching objects for regular access patterns over short runs. The array microbenchmark sums all of the elements in a 100x4,000x10 three dimensional array of integers that is located on a remote machine. Without prefetching the benchmark takes 43.40 seconds and with prefetching it takes 1.41 seconds. Prefetching improves the performance of the array microbenchmark by a factor of 30.78×.

The second microbenchmark traverses a remote linked list with 1,000,000 nodes. Prefetching improves the performance of this benchmark by 27.07×. The microbenchmarks were only intended to quantify the possible contributions of prefetching.

6.6 Commit Intensive Microbenchmarks

Figure 12 presents the result of four microbenchmarks that evaluate the overhead of committing 10,000 transactions in the absence of data contention. We executed each of the benchmarks with 1, 2, 4, and 8 nodes in the system. In the 1Read benchmark, one node commits 10,000 transactions and each transaction reads a shared object from each node in the system. MultiRead implements the same computation, but every node in the system executes the transactions. The microbenchmarks 1Write and MultiWrite perform the same basic computation, but write to the objects instead of reading from them. As the round trip network latency for our testbed is on the order of 100 microseconds, the microbenchmark results show that the overhead to commit transactions that involve a small number of machines is on the order of a single network round trip.

	1Read	MultiRead	1Write	MultiWrite
1	0.65s	0.66s	0.86s	0.86s
2	1.71s	2.46s	1.99s	2.74s
4	1.99s	3.08s	2.29s	3.73s
8	2.50s	4.68s	2.75s	6.22s

Fig. 12. Commit Microbenchmark Results

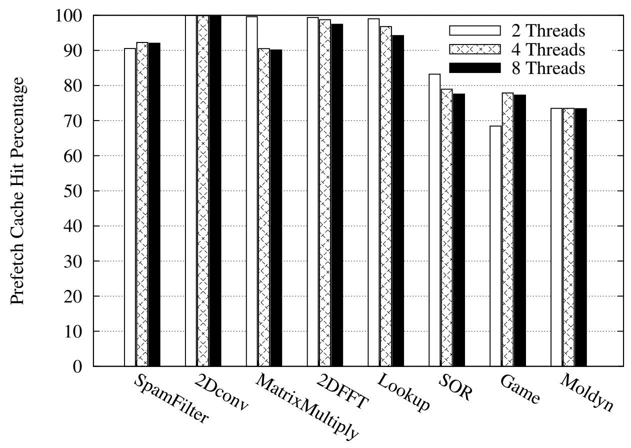

Fig. 13. Prefetching Hit Percentage

6.7 Discussion

The Java versions of the Lookup and Game benchmarks contain 355 and 1,873 lines of code, respectively. The Java versions of Lookup and Game contain 10% and 32% more lines of code, respectively, than the base version. If a developer wished to take advantage of prefetching and caching in Java, the developer would need to write complicated code to manage prefetching and caching of objects. Developers get these benefits for free in our system.

Figure 13 presents the hit percentage in the cache for the prefetching versions of the benchmarks. We hit in the cache more than 80% of the time for most benchmarks showing that prefetching and caching can hide the latency of most remote object accesses.

Figures 14 and 15 present the numbers for remote reads averaged over all machines participating in the running the transactions per benchmark. The base version shows the number of remote reads without prefetching or caching, The cache version shows the number of remote reads with only caching enabled and prefetch version shows the number of remote reads with prefetching and caching enabled. We can use the round trip latency of our system along with the number of eliminated remote reads to predict the speedups from caching and prefetching. In general, our results show that our techniques are able to eliminate most of the latency for our benchmarks. Several benchmarks perform very few remote reads, and therefore the absolute performance gains achieved by eliminating these remote reads remain small. We note that in some cases, we achieve larger speedups than predicted. Reading a large object requires waiting for both the network latency and the transfer time for the object. Prefetching in this case also hide the transfer time of the objects.

We found that writing manual prefetches is straightforward and works well for benchmarks that access data in a predictable order. A developer simply write prefetch annotations to obtain data in batches without having to reason about the details of implementing prefetching.

Our system is not without limitations. Benchmarks that perform relatively little computation compared to the amount of data exchanged over the network are poor choices for our system unless they must be distributed for some other reason. Small problem sizes can yield relatively poor performance as the communication overheads dominate the performance. We expect our approach to be primarily useful for long running computations that need the computational resources of many machines or applications that are inherently distributed in nature.

7 Related Work

We survey related work in distributed shared memory systems, software transaction memory systems, distributed transactional memory systems, and prefetching optimizations.

7.1 Distributed Shared Memory Systems

The IVY shared memory system allows multiple data structures copies to exist to decrease the overhead of reading remote data [7]. The complication with this approach is ensuring that all the copies are consistent after memory writes. IVY uses a write-invalidate protocol that invalidates all copies before writing to a page, and therefore the required round-trip communications make writes to shared memory potentially expensive. To address this issue, researchers have developed more sophisticated approaches including TreadMarks [8], Midway [9], and Munin [10] that achieve higher performance by weakening the memory consistency guarantees [11], [12]. Developing software for weaker memory models requires understanding complicated consistency properties to know which values a read from a memory location can return.

7.2 Transactional Memory

Knight proposed a hardware transactional memory that supported a single store operation [13]. Herlihy and Moss extended this work to support short transactions that write to multiple memory locations [14]. More recent approaches have relaxed the constraints on the transaction size [15], [16]. Shavit and Touitou first proposed a software approach to transactional memory for transactions whose data set can be statically determined [17]. Herlihy et al. extend the software approaches to handle dynamic transactions whose accesses are determined at runtime [18].

7.3 Distributed Transactional Memory

Researchers have explored distributed transactional memory as a mechanism to provide stronger consistency properties. Bodorik et al. developed a hardware-assisted lock-based approach, in which transactions must hold a lock on a memory location before accessing that location [19]. Hastings extended

Thds	Spam Filter			Lookup			Game		
	Base	Cache	Prefetch	Base	Cache	Prefetch	Base	Cache	Prefetch
2	88319	17481	1548	48870	682	482	10335	3312	3262
4	101556	20742	3450	53946	1869	1729	11838	3115	2617
8	107130	22533	5634	58391	3473	3359	13310	3860	3025

Fig. 14. Remote Read Results - Distributed Benchmarks

Thds	2DConv			MolDyn			Matrix Multiply			SOR			2DFFT		
	Base	Cache	Prefetch	Base	Cache	Prefetch	Base	Cache	Prefetch	Base	Cache	Prefetch	Base	Cache	Prefetch
2	10019	10016	4	1740	467	461	32736	32676	127	1205	202	202	15029	15014	97
4	5019	5016	4	1740	467	461	36936	36906	3030	1278	270	270	7529	7513	94
8	2519	2516	5	1740	467	461	35816	35798	3502	1301	289	289	3780	3765	95

Fig. 15. Remote Read Results - Scientific Benchmarks

the Camelot distributed shared memory system to support transactions through a lock-based approach [20]. Ahn et al. developed a lock-based distributed transactional memory system [21]. LOTEC is a lock-based distributed transactional memory [22]. All of these implementations incur network latencies when the application code accesses a remote object because the machine must first communicate to a remote node to acquire a lock.

DiSTM is a distributed transactional memory system [23]. Its commit process checks whether any running (remote) transactions conflict with the current transaction and therefore may incur scaling problems. DiSTM implements three algorithms. In the TCC algorithm, every transaction must send its read and write sets to every other machine. Therefore, the total amount of communications to commit transactions scales as $O(N^2)$, where N is the number of machines. In the serialized lease algorithm, transactions are serialized and must commit to a master in that order. In the multiple lease algorithm, a single master machine must evaluate simultaneous leases for conflicts. Even for computations that scale perfectly, the underlying DiSTM algorithms do not. DiSTM appears to have all machines cache all objects and to propagate all updates to all machines — for large clusters this would become a scalability limitation and limit the total memory available to a computation. Another downside of placing all objects on all machines is that it is not possible to write programs that carefully arrange shared objects to avoid consuming network bandwidth.

Anaconda is a distributed transactional memory system that uses a distributed commit algorithm [24]. It uses a three phase commit protocol in which locks are first acquired, the transaction is validated against running transactions on other nodes, and finally it updates the objects. While both approaches use caching, Anaconda ensures that all cached copies are coherent while our implementation avoids the overhead of updating cached copies and may allow cached objects to become stale.

D^2STM is a fault-tolerant distributed transactional memory implementation [25]. D^2STM replicates objects to provide fault tolerance. D^2STM is a non-voting based transactional memory approach that uses atomic broadcast to ensure that all nodes see the transaction commit requests in the same order. A transaction's read set is encoded as a bloom filter and is validated against transactions that have committed since the committing transaction began. There may be some scalability issues as all nodes process all transaction updates.

Manassiev et al. introduced a version-based distributed transactional memory that replicates all program state on all machines [26]. Their approach is likely to have problems scaling to a large number of machines even if the underlying computation is highly parallel because all writes must be sent to all nodes and all nodes must agree to all transaction commits. Sinfonia is a system that allows machines to share data in a fault-tolerant, scalable, and consistent manner. This service uses mini-transactions to manage distributed state [27]. Mini-transactions piggyback all transaction communications on the commit message. Mini-transactions trade off expressiveness for reduced communication overhead — for example, a single mini-transaction cannot read a value and then write that value to a different location. Our system provides a more general programming model — transactions can immediately use the values they read and can perform sequences of operations that require more than one round of communications. Sinfonia does not provide support for caching or prefetching, but is able to commit the restricted mini-transactions using only one round of communications.

Bocchino et al. have developed Cluster-STM, a word-based software transaction memory system [28]. Their work enumerates and explores a range of transactional memory implementation strategies. They mention but did not implement a distributed transactional memory that uses the read versioning, late acquire, and write buffering approach that our system uses. Relative to Cluster-STM, our system uses speculative caching and prefetching to hide the latency of remote data accesses. Herlihy and Sun proposed a distributed transaction memory for metric-space networks [29]. Their design requires moving objects to the local node before writing to the object. Because these approaches do not contain mechanisms to cache or prefetch remote objects, latency may be an issue. Zhang and Ravindran propose the Relay cache coherence protocol [30]. Their approach defines a cache coherence protocol that moves objects to support distributed transactional memory.

7.4 Prefetching and Caching

Researchers have developed techniques for prefetching recursive data structures in a single machine. Luk and Mowry proposed to greedily prefetch object fields, to automatically add prefetch pointers to objects that point to objects to prefetch, and to linearize recursive data structures when possible [31]. Greedy prefetches require first knowing the address of the object. Prefetch pointers do not help with the initial traversal of a data structure and may be difficult to maintain in a distributed environment. Linearizing is only applicable if the creation order is the same as the traversal order. Cahoon and McKinley proposed a dataflow analysis for software prefetching [32]. Roth et al. propose a hardware-based approach to prefetching linked data structures that hides the latency of accessing linked data structures in useful work [33]. However, in distributed shared memories the latency of accessing remote memory is likely to be much longer than the time that can be filled with useful work.

Researchers have explored communication optimizations for distributed computations. Zhu and Hendren implemented an approach to combine multiple reads into a single block [34]. Because their approach requires that the address of the memory locations to be read is known, it at least incurs the round-trip network latency for accessing each object in a linked data structure traversal. Rogers et al. propose thread migration to improve the performance of accessing remote data structures [35]. An issue with thread migration is that it is not efficient for code that simultaneously operates on data that spans multiple machines.

Gupta proposes a naming scheme for objects in data structures to enable fast traversals of remote data structures [36]. The approach places constraints on data structure updates — only a single node can be added to a data structure at a time. Moreover, many changes to data structures require renaming all of the objects in the data structure and propagating the names changes to all machines.

Speight uses a dynamic prediction-based prefetching algorithm for software distributed shared memory [37]. Joseph and Grunwald use Markov predictors to generate prefetches on a single machine environment [38]. Ferdman and Falsafi store access sequences and then stream the addresses from those access sequences [39]. The transaction component of our work is complementary to dynamic prefetching— our work relaxes constraints on coherency to enable prefetch algorithms to function better and could potentially benefit from dynamic prefetch predictors. The two prefetching approaches may be complementary — we expect that our approach will work better for deterministic access patterns and that dynamic predictors may work better for less deterministic patterns that are repeated.

Distributed databases have used a wide range of caching strategies to efficiently implement transactions [40]. We believe our implementation is the first distributed transactional memory to use optimized cache coherence without requiring that all updates be forwarded to all machines.

8 Conclusion

We have presented a new distributed transactional memory system with support for object caching and prefetching. We have presented a new symbolic expression-based prefetching algorithm that is the only prefetching algorithm to our knowledge that can prefetch objects before the object's address is computed or predicted. We have implemented the language extensions and the distributed shared memory system in our compiler. We observe both significant speedups for our benchmarks. Prefetching is able to hide most of latency of accessing remote objects for our benchmarks.

Acknowledgments. This research was supported by the National Science Foundation under grants CNS-0720854, CCF-0846195, and CCF-0725350. We would like to thank the anonymous reviewers and our editor for their helpful feedback. We would also like to thank Brad Chamberlain for feedback on our paper.

References

[1] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. G. ham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 10-13, September-November 1998.

[2] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, 2007.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Messen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt, *The Fortress Language Specification*, Sun Microsystems, Inc., 2006.

[4] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2009, pp. 141–150.

[5] K. Albrecht, N. Burri, and R. Wattenhofer, "Spamato - an extendable spam filter system," in *2nd Conference on Email and Anti-Spam (CEAS), Stanford University, Palo Alto, California, USA*, July 2005.

[6] L. A. Smith, J. M. Bull, and J. Obdrzalek, "A parallel Java Grande benchmark suite," in *Proceedings of SC2001*, 2001.

[7] K. Li, "Ivy: A shared virtual memory system for parallel computing," in *Proceedings of the 1998 International Conference on Parallel Processing*, 1988, pp. 94–101.

[8] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed shared memory on standard workstations and operating systems," in *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[9] B. N. Bershad and M. J. Zekauskas, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," in *Compcon 93*, 1993.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proceedings of the Second Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 168–176.

[11] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[12] P. B. Gibbons, "A more practical PRAM model," in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[13] T. Knight, "An architecture for mostly functional languages," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986, pp. 105–112.

[14] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[15] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency (TCC)," in *Proceedings of the 11th Intl. Symposium on Computer Architecture*, June 2004.

[16] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *11th International Symposium on High Performance Computer Architecture*, 2005.

[17] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, August 1995.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[19] P. Bodorik, F. I. Smith, and D. J-Lewis, "Transactions in distributed shared memory systems," in *Proceedings of the Eigthth International Conference on Data Engineering*, February 1992.

[20] A. B. Hastings, "Distributed lock management in a transaction processing environment," in *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, October 1990.

[21] J.-H. Ahn, K.-W. Lee, and H.-J. Kim, "Architectural issues in adopting distributed shared memory for distributed object management systems," in *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, August 1995.

[22] P. Graham and Y. Sui, "LOTEC: A simple DSM consistency protocol for Nested Object Transactions," in *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, 1999.

[23] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "DiSTM: A software transactional memory framework for clusters," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, Washington, DC, USA, 2008, pp. 51–58.

[24] C. Kotselidis, M. Luján, M. Ansari, K. Malakasis, B. Khan, C. Kirkham, and I. Watson, "Clustering JVMs with software transactional memory support," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[25] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009.

[26] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[27] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[28] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, 2008.

[29] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *Proceedings of the 19th International Symposium on Distributed Computing*, September 2005.

[30] B. Zhang and B. Ravindran, "Relay: A cache-coherence protocol for distributed transactional memory," in *Proceedings of the 2009 International Conference On Principles Of Distributed Systems*, December 2009.

[31] C.-K. Luk and T. C. Mowry, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 134–141, February 1999.

[32] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in Java," in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[33] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[34] Y. Zhu and L. J. Hendren, "Communication optimizations for parallel C programs," in *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, 1998.

[35] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, 1995.

[36] R. Gupta, "SPMD execution of programs with dynamic data structures on distributed memory machines," in *Proceedings of the 1992 International Conference on Computer Languages*, April 1992.

[37] E. Speight and M. Burtscher, "Delphi: Prediction-based page prefetching to improve the perfo rmance of shared virtual memory systems," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[38] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[39] M. Ferdman and B. Falsafi, "Last-touch correlated data streaming," in *IEEE International Symposium on Systems and Software*, April 2007.

[40] M. J. Franklin, M. J. Carey, and M. Livny, "Transactional client-server cache consistency: Alternatives and performance," *ACM Transactions on Database Systems*, vol. 22, pp. 315–363, 1995.