Institute for Software Research

University of California, Irvine

Software Transactional Distributed Shared Memory

Alokika Dash
University of California, Irvine
adash@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

February 2009

ISR Technical Report # UCI-ISR-09-2

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

www.isr.uci.edu/tech-reports.html

Software Transactional Distributed Shared Memory

Alokika Dash, Brian Demsky
University of California, Irvine

Institute for Software Research
UCI-ISR-09-2

February 2009

We present a new transaction-based approach to distributed shared memory, an object caching framework, language extensions to support our approach, path-expression-based prefetches, and an analysis to generate path expression 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 microbenchmarks and four shared memory parallel benchmarks. We have found that our approach enables our benchmark applications to effectively utilize multiple machines and benefit from prefetching and caching of objects.

Software Transactional Distributed Shared Memory

Alokika Dash and Brian Demsky

University of California, Irvine
Institute for Software Research
UCI-ISR-09-2
February 2009

Abstract

We present a new transaction-based approach to distributed shared memory, an object caching framework, language extensions to support our approach, path-expression-based prefetches, and an analysis to generate path expression 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 microbenchmarks and four shared memory parallel benchmarks. We have found that our approach enables our benchmark applications to effectively utilize multiple machines and benefit from prefetching and caching of objects.

1. Introduction

Price decreases in commodity hardware have led to the widespread adoption of cluster computing. Developing software for these clusters can be challenging. While previous generations of high-performance computers commonly provided developers with a shared memory, modern clusters typically do not provide the developer with a shared memory. Instead, the underlying hardware supports communication between processing nodes through message passing primitives. As a consequence, the already challenging task of developing parallel software has become even more difficult. Developers must now reason about communication patterns, write code to traverse and marshall possibly complex data structures into messages, write communication code to interface with MPI or PVM to route these messages from producers to consumers [17, 13], and write code to unmarshall these message back into data structures.

In response to this trend, researchers have developed software distributed shared memories to provide developers with the illusion of a simple shared memory abstraction on message passing machines. A straightforward implementation of a distributed shared memory can provide developers with a simple memory model to program. However, accessing remote objects in such implementations requires waiting for network communication and therefore is expensive. In response to this issue, researchers have developed several distributed shared memory systems that achieve better performance through relaxing memory consistency guarantees. However, developing software for these relaxed memory consistency models can be challenging — the developer must often read and understand sometimes complicated memory consistency properties to understand the possible behaviors of the program. Developers can also use programming models like Map-Reduce or Dryad [25, 11] to write applications that work in a distributed shared memory but such models can be restrictive to program applications that have a staged structure or have irregular access patterns.

In recent years, a general recognition of the importance of programmer productivity has shifted the focus in computing research from solely performance to the more holistic focus of high productivity computing which encompasses programmer productivity. Both the Chapel [10] and Fortress [3] high performance computing languages include language constructs that specify that code should be executed with transactional semantics. These transactional constructs were included to potentially simplify software development by enabling developers to control concurrency without having to reason about potentially complex locking disciplines. In this paper we have focused on small clusters of servers interconnected with ethernet networks. We expect, but do not have evidence, that our approach would work well to hide latencies over larger networks.

1.1 Basic Technical Approach

In this paper, we present a new transaction-based approach to distributed shared memory that presents a simple programming model to the developer. We use object versioning to track committed changes to objects. A transaction is safe to commit if it only accessed the latest versions of objects.

One of the primary challenges in designing distributed shared memory systems is hiding the latency of accessing remote objects. Previous work on transactional distributed shared 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 prefetches and caches remote objects and relies on the transaction commit checks to safely recover from mis-speculations.

Many traditional approaches to prefetching 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 issuing a prefetch for that object. Our approach describes prefetches in terms of paths through the heap enabling it to prefetch objects whose addresses are not yet known.

1.2 Contributions

This paper makes the following contributions:

- **Transaction-based Distributed Shared Memory:** Our approach is based on the transactional memory model [35, 22, 20, 37]. In this model, the developer uses the `atomic` keyword to declare that a region of code should be executed with transactional semantics[1].
- **Object Prefetching and Caching:** Caching and prefetching objects can potentially hide the latency of reading from or writing to remote objects. To address the possibility of accessing an old version of an object, our approach leverages the transaction commit checks to ensure that a transaction commits only for the latest versions of the objects.
- **Approximate Cache Coherence:** Typically, systems with multiple caches must ensure that data residing in multiple caches is consistent. This typically requires implementing an often expensive cache coherency protocol. We instead use a set of techniques that trade rigorous consistency guarantees for perfor-

[1] In this context, transactional semantics means that the set of reads and writes that the transactional region of code performs is consistent with some sequential ordering of the transactions.

mance. In this context, approximate coherency is safe because the commit process will catch and correct any reads from or writes to old object versions.

- **Path Expression Prefetches:** Traditionally, prefetching a memory location requires that the program first computes or predicts the address of that memory location. Traditional 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 path expression prefetches: a path expression 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 path through the heap. This path traverses the objects to be prefetched. The remote machine processes the path expression and responds with a copy of the initial object and the objects along the path 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 communications.
- **Prefetch Analysis:** Our approach uses a static prefetch analysis to generate path expression prefetches. The analysis uses a statistical approach to estimate at each program point how likely the objects specified by a path expression are to be accessed in the future. The prefetch placement algorithm uses the results of the prefetch analysis to place prefetch instructions.

The remainder of this paper is structured as follows. Section 2 presents an example to illustrate our approach. Section 3 presents the runtime system. Section 4 presents the programming model and locality analysis. Section 5 presents the prefetching mechanism and analysis. Section 6 presents the runtime and prefetch optimizations. Section 7 presents an evaluation on several benchmarks. Section 8 discusses related work and we conclude in Section 9.

2. Example

Figure 1 presents a parallel matrix multiplication example. The example takes as inputs the matrix `a` and the matrix `btrans` and produces as output the product matrix `c`. The `Matrix` class stores the input and output matrices. Lines 6 through 10 initialize the `Matrix` class. The array allocations in lines 7 through 9 each contain the `shared` modifier to indicate that the object is a shared object. Shared objects can be accessed by any thread. 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.

The example partitions the matrix multiplication into several subcomputations that each compute a sub-block of the final product matrix. Figure 2 presents the code for the `MatrixMultiply` class. Each instance of the `MatrixMultiply` class computes one block of the product matrix. Each instance of the `MatrixMultiply` class contains the field `m` that references the shared `Matrix` object and the fields `x0`, `x1`, `y0`, and `y1` that define the block of the product matrix that the `MatrixMultiply` instance computes. Lines 7 through 17 present the code for the `run` method. The `run` method computes the sub-block of the product matrix. Line 8 uses the `atomic` keyword to declare that the enclosed block should be executed with transactional semantics. Note that our system imposes the constraint that shared objects may only be accessed inside transactions.

2.1 Program Execution

We next describe the execution of a `MatrixMultiply` thread. A `MatrixMultiply` thread starts when another thread invokes the `start` method on the `MatrixMultiply` object's *object identifier*. An object identifier uniquely identifies an object. The `start` method takes as a parameter the *machine identifier* for the machine that the thread should be executed on. A machine identifier uniquely identifies a machine participating in the computation. The `start` method causes the runtime system to start a new thread on the specified machine, and then the newly created thread invokes the `run` method on the `MatrixMultiply` object.

```
1  public class Matrix {
2   double[][] a;
3   double[][] btrans;
4   double[][] c;
5
6   public Matrix(int L, int M, int N) {
7    a=shared new double[L][M];
8    btrans=shared new double[N][M];
9    c=shared new double[L][N];
10  }
11   ...
12 }
```

Figure 1. `Matrix` Class

```
1  public class MatrixMultiply
2      extends Thread {
3    Matrix m;
4    int x0, x1, y0, y1;
5    ...
6
7    public void run(){
8     atomic {
9      for(int i=x0; i<x1; i++)
10      for(int j=y0; j<y1; j++) {
11       double prod=0;
12       for(int k=0; k<m.a[i].length; k++)
13        prod+=m.a[i][k]*m.btrans[j][k];
14       m.c[i][j]=prod;
15      }
16    }
17   }
18 }
```

Figure 2. `MatrixMultiply` Class

The `atomic` keyword in line 8 of the `run` method causes the runtime system to execute the code block in lines 9 through 15 with transaction semantics. Upon entering this atomic block, the thread executes compiler inserted code that converts the object identifier stored in the `this` 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. The system then points the `this` variable at the working copy of the object.

In line 12, the `run` method accesses the `Matrix` object through the `m` field of the `this` object. When the example dereferences the `m` field, the generated code reads the object identifier out of the `m` field, searches for the corresponding object, makes a working copy, and points a temporary variable at the working copy. 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. Our system also supports local objects. If a local object is modified inside a transaction, the compiled code makes a backup to enable restoration of the object in the event that the transaction aborts.

When the transaction completes, the `run` method calls the runtime system to commit the transaction. The runtime system sorts the objects into groups by the machine that holds the authoritative copy of the object. It then sends each group of objects 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 re-executes the transaction.

2.2 Object Prefetching

The matrix multiplication example accesses array 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 expression `m.a[i][k]` in line 13 of Figure 2. The traditional approach to prefetching this expression would first prefetch `m`, next `m.a`, and finally `m.a[i]`. This strategy requires three consecutive round trip communications. Our approach sends a path expression prefetch request for (1) the object identifier stored in `m` and (2) the path defined by the offset of field `a` and the $\texttt{i}^{th}$ array element. If the remote machine contains all of the objects, all three objects can be prefetched in a single round trip communication.

2.2.1 Prefetch Analysis

We next describe the operation of our compiler's path expression prefetch analysis on the example. For each program point, the analysis computes a set of path expressions that describe which objects to prefetch along with estimated probabilities that describe how likely the program is to access the objects described by the prefetch expression.

The analysis begins with a backwards fixed-point computation over the method's control flow graph. At each field access node, the analysis creates a path expression for the object(s) to be accessed and it associates a probability of 100% with it. The probability describes how likely the program is to access the object described by the prefetch expression at the current program point. For example, in line 14 the analysis would generate the path expression $\texttt{m.c}[\texttt{i}]$ with a probability of 100%. Note that `j` does not appear in the path expression because the value $\texttt{m.c}[\texttt{i}][\texttt{j}]$ is not read (nor does it refer to an object). When this path expression propagates backwards it hits the `for` loop on line 12. To propagate beyond this loop we use loop exit probability of 20% (when the loop condition is false) and a backedge probability of 80% (when the loop condition is true). As the expression propagates we calculate a new probability by multiplying the old probability with the loop condition probability. As a result, after line 13 we see the path expression $\texttt{m.c}[\texttt{i}]$ with a 20% probability (the probability of the false loop condition). Line 13 then generates two new path expressions $\texttt{m.a}[\texttt{i}]$ and $\texttt{m.btrans}[\texttt{j}]$ with a 100% probability. Path expressions change as they propagate beyond program variables and field names. For example, when the analysis propagates the $\texttt{m.btrans}[\texttt{j}]$ expression to line 10, the loop variable increment causes the analysis to rewrite the expression $\texttt{m.btrans}[\texttt{j}]$ into the expression $\texttt{m.btrans}[\texttt{j} + \texttt{1}]$ with an 64% probability at line 15. We compute a 64% probability for the expression $\texttt{m.btrans}[\texttt{j} + \texttt{1}]$ because it propagates through the two loop conditions at lines 12 and 10 (each with a probability of 80%). The algorithm continues the fixed-point computation until a convergence threshold criteria is satisfied.

2.2.2 Prefetch Placement

After the analysis generates path expressions for all program points, the compiler places the prefetches. In general, we want to place prefetches as early as possible while ensuring that there is a high probability that the prefetches will fetch useful objects. The analysis places prefetches on the edges where the path expression probabilities cross a specified probability threshold. In the example, the compiler places a prefetch for $\texttt{m.btrans}[\texttt{j} + \texttt{4}]$ after line 10; and prefetches for $\texttt{m.btrans}[\texttt{y0}]$, $\texttt{m.btrans}[\texttt{y0} + \texttt{1}]$, $\texttt{m.btrans}[\texttt{y0} + \texttt{2}]$, and $\texttt{m.btrans}[\texttt{y0} + \texttt{3}]$ before line 10.

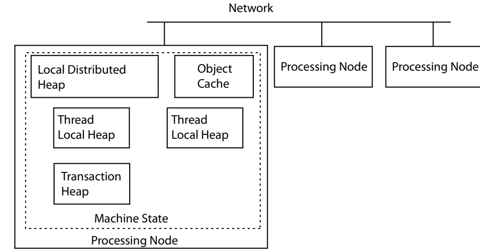

Figure 3. Overview of Transactional Distributed Shared Memory Architecture

3. Transactional Distributed Shared Memory

Figure 3 presents an overview of the runtime system components. The runtime system provides the primitives that the compiler uses to create the illusion of a single address space, while the underlying hardware actually consists of a number of network-connected processing nodes. The runtime system 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. The object identifiers are statically partitioned between the processing nodes. The runtime system can determine the location of an object directly from its object identifier.

We use a version-based strategy to implement transactions. 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. Our runtime uses the standard two-phase commit protocol to determine whether a transaction is safe to commit [16].

The implementation maintains the following types of object copies:

- **Authoritative Copy:** The authoritative copy contains all of the 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 any object, the runtime makes a working copy of the object for that transaction. The cached copy can be stale — if a transaction accesses a stale object copy, the transaction commit process will abort.
- **Working Copy:** When a transaction accesses a shared object, a working copy is made for that transaction. The transaction performs writes on the working copy. When the transaction commits, any updates to the object are copied from the working copy to the authoritative copy. It is possible for the working copy to be stale. If the working copy is stale, the transaction commit process will abort.

3.1 Basic Architecture

We next discuss the basic architecture of our approach. The expanded processing node in Figure 3 presents the major components in our software distributed shared memory system. Each processing node contains the following state:

- **Local Distributed Heap:** The shared memory is partitioned across all processing nodes. Each node will store a disjoint sub-

set 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 *local distributed hashtable* 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 thread local object heap.
- **Transaction Heap:** There is a transaction heap for each ongoing transaction. The transaction heap stores the working copy of any shared object that the current transaction has accessed. Each transaction heap contains a *transaction hashtable* that maps the object identifiers that the transaction has accessed to the object's location 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 an *object cache hashtable* that maps the object identifiers of the objects in the cache to the object's location in the cache.

3.2 Accessing Objects

Our system uses a partitioned global address space (PGAS) programming model [39, 10, 3]. Our system contains two classes of objects: local objects and shared objects. Local objects are local to a single thread and can only be accessed by that thread. They can be accessed both inside and outside of a transaction. Accessing a local object outside of a transaction requires a simple pointer dereference. Reading a local object inside a transaction also requires 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 working 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 working 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. In this case, locating the working copy of an object involves only a few instructions: a bit mask and a shift operation to compute a hash value, an address computation operation, a memory dereference to lookup the object identifier, a comparison to verify object identifier, and a memory dereference to obtain the working 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 transaction commit must update the authoritative copy of the object.

[2] Each object contains a status word, and the write barrier marks the object as dirty by setting the dirty flag in the object's status word.

3.3 Commit Process

We next describe the operation of the transaction commit process. 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 divides the shared objects based upon whether they have been written to or simply read from. The commit process uses the standard two-phase commit. We next describe how the algorithm processes each category of shared object:

- **Clean Objects:** For clean objects, the transaction commit process 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 locks the object and compares version numbers. If the version numbers do not match the machine releases the lock on the objects and votes to abort the transaction.
- **Dirty Objects:** The transaction commit process must copy the updates that the transaction made to 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 locks the authoritative copy and compares version numbers. If the version numbers 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 of the authoritative machines respond that all version numbers match, the transaction coordinator will decide to commit the transaction and transmit commit commands to all other machines. If any authoritative machine responds with an abort request, the transaction coordinator will decide to abort and transmit abort commands to all other machines. If any authoritative machine cannot acquire a lock on an object, the coordinator will abort the commit process and retry.

Code inside a transaction can also modify thread local objects and local variables. When a transaction begins, the compiler generates code that makes a copy of all live local variables. Whenever a transaction writes to a local object, the compiled 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.

3.4 Error Handling

During a transaction, the execution can potentially read inconsistent versions of objects. While such executions will abort during the commit process, reading inconsistent values can cause even correct code to potentially throw an error before the transaction commits. Therefore, if the execution of a transaction throws an exception, the runtime system must verify that the transaction read consistent versions of the objects before propagating the error. The system simply performs the transaction commit checks to verify that the error is real before propagating the error. Similarly, there is the potential for infinite looping due to reading inconsistent versions of objects. We plan to include a timeout after which the runtime will verify that the transaction has read consistent versions of the objects. If the object versions are consistent, the execution of code will continue, otherwise the transaction will be restarted.

3.5 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 work-

ing copy of the shared object. This invariant makes subsequent reads of 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 the shared object's object identifier. 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 working copies and code to convert references to working copies back into object identifiers.

4. Programming Model

Our approach uses a set of language extensions to a subset of Java to support transactions. We describe these extensions below.

4.1 Language 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 [39] though our use of transactions introduces additional constraints on when the application may access shared objects.

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 that a method has different behavior depending on whether the parameter objects are shared objects. Our extensions support different creating method versions for local and shared objects — the developer designates the shared version with the `shared` 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.

4.2 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 4 presents the lattice for the abstract object state domain.

Figure 5 presents the transfer functions for the data flow analysis. The analysis starts by analyzing the `main` function 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 encounters a call site for a method context that it has not yet 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 actual abstract state of the return value. Whenever the analysis updates the return value for a method context, it enqueues all callers of that context for re-analysis.

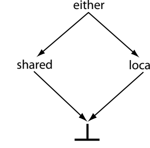

Figure 4. Lattice for Analysis

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

Figure 5. Transfer Function for Inference 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[3], (5) that shared objects are not accessed outside of transactions, and (6) that shared objects are not passed into native methods.

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 also note that there is the potential for the analysis to generate a large number of versions for a single method. If this occurs, the compiler could simply generate a generic version of the method.

5. Path Expression Prefetching

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

We introduce a new approach to prefetching objects in the distributed environment that leverages the computational capabilities of the remote processors. Our approach communicates path expressions that describe a path through the heap that traverses the objects to be prefetched. We next describe the path expression runtime

[3] This constraint prohibits I/O calls inside of transactions. We make an exception for a debugging print statement and known side-effect free native methods including the standard floating point calls.

mechanism and a corresponding compiler analysis that enables our implementation to efficiently prefetch complex linked data structures from remote machines.

5.1 Runtime Mechanism

We have developed path-expression-based prefetching, a new prefetching mechanism that enables prefetching multiple objects even multiple references away with a single round-trip network communication. Path expressions have the form: path expression := base object identifier(.field | [integer])*. The base object identifier component of the path expression gives the object identifier of the first object in the path expression. The list of field offsets and array indices describe a path through heap from the first object.

We next consider the following example code segment:

```
1 LinkedList search(int key) {
2  for(LinkedList ptr=head;ptr!=null&&ptr.key!=key)
3    ptr=ptr.next;
4  return ptr;
5 }
```

Without prefetching, completely searching a remote linked list of length n requires making n consecutive round-trip message exchanges. If we add a prefetch for the expression `ptr.next.next.next.next.next` between lines 2 and 3, the runtime will have prefetch requests in flight for the next linked list node and the subsequent four nodes that follow that node[4]. The example path expression prefetch enables the `search` method to potentially execute five times faster. Longer path 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 five objects that have prefetch requests in flight.

Our path expression implementation contains the following key components:

1. **Prefetch Calls:** Our prefetching approach begins with a prefetch call from the application. Our implementation supports issuing several path expression prefetches with a single prefetch call. The prefetch takes as input the number of path 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 path 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 machine that is authoritative for the base object identifier. 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 path 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 the prefetch response to the original machine with copies of the objects. 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.[5]
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.

5.2 Prefetch Analysis

We have developed an unsound, intraprocedural static analysis that uses a simple probabilistic model to generate both a set of path expressions that the program may access and the corresponding estimated probabilities that the objects represented by that path expressions will be accessed. The probabilistic model is naive — it makes assumptions of independence that are likely not true. However, the results need not be precise, but simply provide a rough approximation of the real program's data access patterns. It is acceptable for the analysis to be unsound because prefetches do not affect the program's correctness.

The analysis is a backward flow analysis that computes set of tuples $\mathcal{P} \subseteq \Phi \times \mathbb{R}$ containing a path expression ϕ and a corresponding probability $\mathbb{R}$ for each program point. Each path expression $\phi = \mathcal{V}\mathcal{I}_0\mathcal{I}_1...\mathcal{I}_{n-1} \in \Phi$ is comprised of a variable $\mathcal{V}$ and a sequence of field offsets or array indices $\mathcal{I} =$ .offset | [index]. Each array index index $= \text{tmp}_0 + ... + \text{tmp}_{m-1} + c$ is a summation of temporary variables tmp and a constant offset c.

The analysis initializes the set of tuples for each program point to the empty set. The ordering relation is $\mathcal{P}_1 \sqsubseteq \mathcal{P}_2$ iff $\forall \langle \phi, d_1 \rangle \in \mathcal{P}_1$ there $\exists d_2 > d_1$ such that $\langle \phi, d_2 \rangle \in \mathcal{P}_2$.

Figure 6 presents the transfer functions for the analysis. The transfer functions for statements that read an object reference from a field or an array element generate new path expressions with an associated probability of 100% and rewrite any path expressions that contained the destination variable. The transfer functions for statements that make assignments, write to fields, or write to array elements rewrite path expressions that begin with the same variable and field or array index. Figure 7 present the `REPLACE` function that rewrites the path expressions. One possible issue is that a rewritten path expression may match an existing path expression. The `COMBINE` function computes the new probability making the assumption that the probabilities for the path expressions were independent. We have omitted the `REPLACE` functions for index variables for space reasons.

[4] The prefetch look-ahead distance is not fixed. Instead it depends on the analysis's estimation of how likely the prefetched values are to be used.

[5] We need to forward because after the original machine processes the prefetch request, it could contain references to still more remote objects.

st	$[\![st]\!](\mathcal{P})$
$\texttt{x} = \texttt{y.f}$	$(\texttt{REPLACE}(\texttt{x}, \texttt{y.f}, \mathcal{P}) - \langle \texttt{y.f}, * \rangle) \cup \langle \texttt{y.f}, 1 \rangle$
$\texttt{x} = \texttt{y[t]}$	$(\texttt{REPLACE}(\texttt{x}, \texttt{y[t]}, \mathcal{P}) - \langle \texttt{y[t]}, * \rangle) \cup \langle \texttt{y[t]}, 1 \rangle$
$\texttt{x} = \texttt{y}$	$\texttt{REPLACE}(\texttt{x}, \texttt{y}, \mathcal{P})$
$\texttt{x.f} = \texttt{y}$	$\texttt{REPLACE}(\texttt{x.f}, \texttt{y}, \mathcal{P})$
$\texttt{x[t]} = \texttt{y}$	$\texttt{REPLACE}(\texttt{x[t]}, \texttt{y}, \mathcal{P})$
$\texttt{t} = \texttt{t}_1 + \texttt{t}_2$	$\texttt{REPLACE}(\texttt{t}, \texttt{t}_1 + \texttt{t}_2, \mathcal{P})$
$\texttt{t} = \texttt{c}$	$\texttt{REPLACE}(\texttt{t}, \texttt{c}, \mathcal{P})$
other assignments to x	$\mathcal{P} - \langle \texttt{x}, * \rangle$

Figure 6. Transfer Functions

$\texttt{REPLACE}(\phi_1, \phi_2, \mathcal{P}) = \texttt{COMBINE}(\texttt{REWRITE}(\phi_1, \phi_2, \mathcal{P}))$
$\texttt{REWRITE}(\phi_1, \phi_2, \mathcal{P}) = \{\langle \pi(\phi, \phi_1, \phi_2), d \rangle \mid \langle \phi, d \rangle \in \mathcal{P}\}$
$\texttt{COMBINE}(\mathcal{P}) = \{\langle \phi, d \rangle \mid \{d_0, d_1, ..., d_{n-1}\} = \mathcal{P}(\phi), d = 1 - (1 - d_0)(1 - d_1)...(1 - d_{n-1})\}$
$\pi(\phi, \phi_1, \phi_2) = \phi_2 \mathcal{I}_0 ... \mathcal{I}_n$ if $\phi = \phi_1 \mathcal{I}_0 ... \mathcal{I}_n$, ϕ otherwise

Figure 7. Equation for the REPLACE Function

Our analysis associates a probability with each conditional branch. By default, we assume that loop branches take the true branch with an 80% probability and other branches take the true branch with a 50% probability. Our meet operation merges the path expressions from two branches by weighting the probabilities in prefetch set for the true branch with the weight p and the probabilities in the prefetch set for the false branch with the weight $1 - p$. The analysis can handle loops terminated by exception by explicitly implementing exceptions in the control flow graph.

Note that the partial order on $\mathcal{P}$ is not a lattice. The analysis as stated does not terminate. We next present extensions that ensure termination. One issue is that the analysis can generate path expressions of unbounded length. We address this issue by introducing a minimum path expression probability μ. If a path expression has a probability less that μ at a program point, the analysis drops that path expression. A second issue is that the analysis can converge slowly as the analysis makes increasingly smaller increments to the path expression probabilities. We introduce a minimum change threshold δ. If the probability changes by less than δ, the fixed-point algorithm considers the probability to be the same.

5.3 Prefetch Placement

There is a trade off between placing prefetches early to minimize the time that the application waits for data and waiting long enough to make sure the program is likely to use the prefetched data. This trade off can depend on the specific architecture of the machine and the application — bandwidth constraints can be satisfied by delaying prefetches while latency constraints can be satisfied by moving prefetches earlier in the execution. Our implementation, therefore, allows the developer to specify a probability threshold σ. We selected σ to be 30% for all of our benchmarks.

We instrument the analysis in the previous section to record the mapping $\gamma(\phi, E) \rightarrow \phi'$ which maps the path expression ϕ at the source of the edge E to the corresponding path expression ϕ' at the target of the edge E. Prefetches are placed on edges where the probability of using the objects specified by a path expression crosses the developer specified threshold. We define the function τ below to check if an edge crosses the probability threshold:

$\tau(\phi, E) = (\mathcal{P}_{dst(E)}(\phi) > \sigma) \wedge (\mathcal{P}_{src(E)}(\gamma(\phi, E)) < \sigma)$

Simply using a threshold crossing criteria to place prefetches can result in redundant prefetches. We therefore extend our approach to check whether the path expression has already been prefetched. We define the set S_N at each program point to be the set of path expressions that have been prefetched when the program executes the statement at node N. This set is the intersection of the set of prefetched path expressions along each incoming edge E to node N. We split the prefetched path expressions into two components: $\mathcal{S}_E$ is the set of path expressions that have been prefetched before the source node of E has been executed and δ_E is the set of prefetches inserted at E. The equations for each set follow:

$S_N = \bigcap_{E=\texttt{incoming edges to } N} (\mathcal{S}_E \cup \delta_E)$
$\mathcal{S}_E = \{\gamma(\phi, E) \mid \phi \in S_{src(E)}\}$
$\delta_E = \{\phi \mid \exists d, \langle \phi, d \rangle \in \mathcal{P}_{src(E)}, \tau(\phi, E)\}$

We use a fixed point algorithm to compute these sets for all program points. At each edge E, our prefetch placement algorithm places prefetches for the path expressions in $\delta_E - \mathcal{S}_E$ (the set of path expressions that cross the threshold but have not already been prefetched).

6. Runtime and Prefetch Optimizations

In this section, we describe a number of runtime optimizations that our system implements.

6.1 Approximate 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 prefetch cache. The obvious approach, a cache coherence protocol, for addressing this issue introduces a number of overheads. We have introduced several new mechanisms that are collectively designed to provide approximate 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 list 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 approximate cache coherence. Techniques for very large deployments could include profiling to determine at what points objects of a given type should be invalidated from caches.

Our implementation uses information from local transactions to update the prefetch 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 prefetch cache and the transaction is retried.

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

6.2 Dynamic prefetching optimizations

Transactions often access an object multiple times. For example, the loops in matrix multiply will walk over the same array objects repeatedly. In this case, our prefetch analysis can generate prefetch instructions that prefetch the same object repeatedly. Processing these repeated prefetch instructions introduces overhead and yields no performance benefits as the objects are already cached.

We observe that in many benchmarks, the execution transitions between phases in which it mostly accesses new objects and phases

in which it accesses the same objects. In the matrix example, during the first iteration of the outer loop the code accesses new arrays in the `btrans` matrix. For the remaining iterations, the code accesses the same arrays repeatedly. We therefore introduce a mechanism that dynamically shuts down prefetch sites when they stop providing benefits. This mechanism allows the application to receive the benefit of prefetches while minimizing the overhead.

Our approach assigns a unique identifier to each prefetch site. The prefetch requests are labeled with this unique identifier. Each time a prefetch is generated for objects that are already in the local cache, the runtime increments a count associated with the prefetch site. When the prefetch site generates a prefetch request that is not locally available, the runtime resets this count. Once this count hits a threshold, the runtime sets a flag that shuts down this prefetch site. Our implementation continues to monitor the prefetch site by occasionally retrying prefetches after a shutdown. If the prefetch retry request prefetches a non-cached object, the runtime turns the prefetch site back on.

7. 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 through a gigabit switch. We have implemented the DSM system, path expression prefetching, the language extensions, and the analysis. We present results for microbenchmarks and four shared memory parallel benchmarks. We report results for 1Threaded Java for a single-threaded non-transactional Java version compiled into C code; the base column presents results without caching or prefetching; and the prefetch column presents results with both caching and prefetching enabled. The prefetching versions are generated automatically using our prefetch analysis. We have reported numbers in seconds that are averaged over ten executions for 1, 2, 4, and 8 nodes with one thread running per node.

7.1 2DConv

The 2D convolution benchmark computes the application of a mask to a 2D image. The output image, C, is computed from the input image, A, and the convolution mask, H. Each machine computes a region of the output image in parallel. Table 1 presents results for the 2DConv benchmark. We observe speedups as we increase the number of nodes but prefetching ceases to provide benefits as we hit 8 nodes. A measurement of the 4096x4096 version reveals the problem: the 8 node benchmark transmits 1.9 gigabits of data (the output and input matrices) from the master to the nodes and then the commits transmit 0.9 gigabits of data from the nodes to the master over a network with a peak capacity of 1 gigabit per second. The problem is that the benchmark becomes bandwidth limited and data prefetched for future operations competes with immediate remote requests for bandwidth. The 2048x2048 version suffers from the same bandwidth limit.

2DConv	$M = N = 2048$		$M = N = 4096$	
	Base	Prefetch	Base	Prefetch
1Threaded Java	2.54s	—	11.57s	—
1	2.80s	—	11.54s	—
2	2.49s	2.21s	9.21s	8.19s
4	1.65s	1.47s	6.14s	5.66s
8	1.28s	1.28s	4.84s	4.98s

Table 1. 2DConv Results

7.2 Matrix Multiply

The matrix multiplication benchmark implements the standard matrix multiplication algorithm. The computation of the product matrix is partitioned over multiple threads. Matrix Multiply places all of the arrays on the machine that started the computation.

Table 2 presents the results for a 600x600 matrix multiplication benchmark. We observe significant speedup for small matrix size as we increase the number of nodes and we gain 35.5% speedup with prefetching for 8 nodes.

Matrix Multiply	Base	Prefetch
1Threaded Java	2.30s	—
1	4.30s	—
2	2.45s	2.37s
4	1.70s	1.35s
8	1.26s	0.93s

Table 2. Matrix Multiplication Results

7.3 Moldyn

Moldyn, a molecular dynamics benchmark, was taken from the Java Grande benchmark suite [36]. It is a N-body simulation of particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions.

Table 3 presents results for the Moldyn benchmark for $N = 2048$ particles and $i = 100$ iterations. We observe speedups as we increase the number of nodes. We gain 14.7% speedup with prefetching for 8 nodes.

Moldyn	Base	Prefetching
1Threaded Java	3.47s	—
1	7.79s	—
2	4.88s	4.16s
4	3.23s	2.95s
8	3.35s	2.92s

Table 3. Moldyn Results

7.4 2DFFT

The 2DFFT benchmark is an example of two-dimensional fast Fourier transformation. The algorithm was taken from *Digital Signal Processing* by Lyon and Rao and parallelized. The main matrix is placed on the machine that started the computation. This 1152×1152 matrix is accessed remotely by other nodes that perform one dimensional Fourier transforms on the rows in parallel followed by one dimensional Fourier transforms on the columns in parallel. Table 4 presents results for the 2DFFT benchmark. We observe significant speedup as we increase the number of nodes and gains from prefetches.

2DFFT	Base	Prefetch
1Threaded Java	3.64s	—
1	5.35s	—
2	3.67s	3.65s
4	2.63s	2.58s
8	2.09s	1.97s

Table 4. 2DFFT Results

7.5 Array Microbenchmark

We present results from a two-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 10000x10 two dimensional array of integers that is located on a remote machine. Without prefetching the benchmark takes 1.03 seconds and with prefetching it takes 0.15 seconds. Prefetching improves the performance of the array microbenchmark by a factor of 6.9. The microbenchmark is only intended to quantify the contributions of prefetching. Therefore, results for even short runs demonstrate that some access patterns can benefit significantly from prefetching.

7.6 Overhead From Transactions

Table 5 presents the result of four micro-benchmarks that evaluate the overhead of committing 10,000 transactions in absence of data contention for 1, 2, 4, and 8 nodes. Note that the roundtrip network latency is on the order of 100 microseconds — 10,000 network roundtrips take 1 second. In OneMCRead benchmark, one node commits 10,000 transactions when reading a shared array. In MultiMCRead, multiple nodes commit transactions when reading shared arrays that are located remotely. In OneMCWrite benchmark, one node commits 10,000 transactions on a shared array located remotely while in MultiMCWrite, multiple nodes write to different parts of shared arrays located remotely. From the results we

see that the overhead of transactions scale linearly with the number of nodes and are in fact dependent on the round trip communication involved in accessing remote data. Note that the overhead to commit transactions that involve a small number of machines is on the order of a single network round trip or the time to read a single remote object.

	OneMCRead	MultiMCRead	OneMCWrite	MultiMCWrite
1	0.25s	0.25s	0.52s	0.51s
2	1.47s	1.70s	1.96s	2.36s
4	1.79s	3.44s	2.72s	4.03s
8	2.53s	5.50s	4.43s	7.49s

Table 5. Commit Benchmark Results

8. Related Work

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

8.1 Distributed Shared Memory Systems

The IVY shared memory system allows multiple data structures copies to exist to decrease the overhead of reading remote data [30]. The complication with this approach is ensuring that all the copies are consistent after memory writes. IVY uses a write-invalidate protocol to invalidate all copies before writing to a page, and therefore the required round trip communications makes writes to shared memory potentially expensive. We note that there is the potential to ping pages back and forth between machines.

To address this issue, researchers have developed more sophisticated approaches including TreadMarks [27], Midway [6], and Munin [5] that achieve higher performance by weakening the memory consistency guarantees [28, 14]. Developing software for weaker memory models requires the developer to understand complicated consistency properties to understand which values reads from memory locations can return.

8.2 Transactional Memory

Knight proposed a limited form of hardware transactional memory that supported a single store operation [29]. Herlihy and Moss extended this work to support short transactions that write to multiple memory locations in hardware [23]. More recent approaches have relaxed the constraints on the transaction size [19, 4]. Shavit and Touitou first proposed a software approach to transactional memory for transactions whose data set can be statically determined [35]. Herlihy et al. extend the software approaches to handle dynamic transactions whose accesses are determined at runtime [22].

8.3 Transactional Distributed Shared Memory

Researchers have explored transactional distributed shared memory systems 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 [8]. Hastings extended the Camelot distributed shared memory system to support transactions though a lock-based approach [21]. Ahn et al. developed a lock-based distributed shared memory system with support for transactions [2]. LOTEC is another lock-based transactional distributed shared memory [15]. All of these implementations incur round trip network latencies whenever the application code accesses a remote object because the machine must first communicate to a remote node to acquire a lock.

Manassiev et al. introduced a version-based transactional distributed shared memory that replicates all program state on all machines [32]. 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. Marcos et al. have developed 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 [1].

Two of the DARPA high productivity computer systems languages, Chapel [10] and Fortress [3], provide transaction constructs that guarantee that code is executed with transactional semantics. As part of this effort Boechino et al. have developed a word-based software transaction memory system [7]. Herlihy and Sun proposed a distributed transaction memory for metric-space networks [24]. Their design requires moving objects to the local node before writing to the object. Because neither of these approaches contain mechanisms to cache or prefetch remote objects, the latency of accessing remote objects may be an issue.

8.4 Prefetching

Researchers have developed several techniques for prefetching recursive data structures in a single machine environment. Luk and Mowry propose 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 in Java [9]. 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 [34]. 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 [40]. 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 [33]. 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 [18]. 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 [38]. Joseph and Grunwald use Markov predictors to generate prefetches on a single machine environment [26]. Ferdman and Falsafi store access sequences and then stream the addresses from these access sequences onto a chip's cache [12]. The transaction component of the 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 static approach will work better for deterministic object access patterns and that dynamic predictors may work better for less deterministic access patterns that are repeated many times.

9. Conclusion

We have presented a new transaction-based distributed shared memory system with support for object caching. We have presented a new path 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 prefetching analysis, the language extensions, and the distributed shared memory system in our compiler. We have observed speedups for our benchmarks as the number of machines increases and also observe benefits from prefetching objects.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[2] 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.

[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., September 2006.

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

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

[13] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 258–261, 1991.

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

[15] 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.

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

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[18] 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.

[19] 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.

[20] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation*, June.

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

[22] 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.

[23] 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.

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

[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

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

[27] 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.

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

[29] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

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

[31] C.-K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2):134–141, February 1999.

[32] 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.

[33] 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*, 17(2):233–263, 1995.

[34] 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.

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

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

[37] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict detection and validation strategies for software transactional

memory. In *Proceedings of the Twentieth International Symposium on Distributed Computing*.

[38] 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.

[39] K. Yelick, L. Semenzato, G. Pike, C. M. iyamoto, 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*, 10(10-13), September-November 1998.

[40] Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998.