Automatically Generating Symbolic Prefetches for Distributed Transactional Memories

Alokika Dash and Brian Demsky

University of California, Irvine

Abstract. Developing efficient distributed applications while managing complexity can be challenging. Managing network latency is a key challenge for distributed applications. We propose a new approach to prefetching, symbolic prefetching, that can prefetch remote objects before their addresses are known. Our approach was designed to hide the latency of accessing remote objects in distributed transactional memory and a wide range of distributed object middleware frameworks. We present a static compiler analysis for the automatic generation of symbolic prefetches — symbolic prefetches allow objects whose addresses are unknown to be prefetched.

We evaluate this prefetching mechanism in the context of a middleware framework for distributed transactional memory. Our evaluation includes microbenchmarks, scientific benchmarks, and distributed benchmarks. Our results show that symbolic prefetching combined with caching can eliminate an average of 87% of remote reads. We measured speedups due to prefetching of up to 13.31× for accessing arrays and 4.54× for accessing linked lists.

1 Introduction

Developing efficient distributed applications while managing complexity can be challenging. Recently, several researchers have developed distributed transactional memory systems that adapt ideas from work on software transactional memory for use in distributed systems [1,2,3,4,5,6,7,8]. These systems use the transaction mechanism to simplify managing concurrent access to data structures in distributed systems while eliminating possibility of distributed deadlocks. Moreover, these techniques can be adapted to provide memory transactions that guarantee durability, atomicity, and isolation even in the presence of failures [1]. An additional benefit of distributed transactional memory is that it batches communications to achieve consistency and allows safe speculation. This approach amortizes communication overhead and therefore reduces the overhead of coherency. Distributed transactional memory middleware frameworks provide powerful constructs and sophisticated optimizations to greatly simplify the process of developing distributed applications.

A key challenge in designing distributed software applications is managing the latency of accessing remote data. Traditional prefetching approaches have had limited success in hiding the latency of remote object accesses because they require programs to compute or predict objects' addresses before issuing prefetches. We propose a novel prefetching mechanism, *symbolic prefetches*, as an optimization for distributed object middleware frameworks including distributed transactional memory.

I. Gupta and C. Mascolo (Eds.): Middleware 2010, LNCS 6452, pp. 355–375, 2010.

Symbolic prefetches specify a start point in a program's heap and a set of fields or array indices that define a path through the heap. Symbolic prefetches allow an application to prefetch a chain of objects (some of whose addresses are unknown) in a single round trip message exchange instead of the multiple round trips that are currently necessary. Therefore, symbolic prefetches have the potential to hide much of latency of accessing remote objects. To our knowledge, this is the first prefetching approach to specify objects in terms of paths through the heap. While we have only evaluated symbolic prefetching in the context of a distributed transactional memory middleware framework, we expect the technique to be applicable to a wide range of distributed object middleware frameworks ranging from distributed object stores [9,10,11] to mobile object frameworks.

This paper presents a novel static compiler analysis that can automatically generate symbolic prefetches for distributed applications. It combines this static analysis with a dynamic optimization that turns off symbolic prefetching during execution phases when it does not yield performance benefits. This paper makes the following contributions:

- **Prefetch Analysis:** It presents a prefetch analysis that automatically generates symbolic prefetches: a symbolic prefetch specifies the first object to be prefetched followed by a list of fields or array indices that define a path through the heap.
- **Dynamic Optimizations:** It presents several novel dynamic optimizations that lower runtime overheads.
- **Evaluation:** We have evaluated our implementation on several parallel and distributed benchmarks and found that our system works well and that prefetching improves the performance of our benchmarks and can hide a significant percentage of the network latency.

The remainder of the paper is structured as follows. Section 2 presents an example that we use to illustrate our approach. Section 3 presents an overview of the system. Section 4 presents programming model. Section 5 presents the prefetching analysis. Section 6 presents dynamic optimizations. Section 7 presents our evaluation of the approach on several benchmark applications. Section 8 discusses related work; we conclude in Section 9.

2 Example

Figure 1 presents a distributed matrix multiplication example. The example takes as input the matrices `a` and `btrans` and computes the product matrix `c`. The `shared` keyword that appears in the allocation statement in the `Matrix` constructor indicates that the allocated array is shared and can be accessed by remote machines. The `parallelmult` method partitions the matrix multiplication into several subcomputations. Each `MatrixMultiply` object represents one such subcomputation and its `x0` and `x1` fields define the block of the product matrix that it computes.

2.1 Program Execution

We next describe the execution of a `MatrixMultiply` thread. A `MatrixMultiply` thread starts when the program invokes the `start` method on the `MatrixMultiply`

object's *object identifier*. An object identifier uniquely identifies a shared object. The `start` method takes as input the *machine identifier* for the machine that should execute the thread. The `start` method causes the runtime to start a thread on the given machine to execute the object's `run` method.

The `atomic` keyword in line 17 causes the code from lines 18 through 26 to execute with transactional semantics. Transactional semantics means that the reads and writes that a transaction performs are consistent with some total ordering of the transactions. Upon entering this transaction, the thread executes compiler-inserted code that converts the object identifier stored in the `this` variable into a reference to a *transaction local copy* of the object. A transaction local copy of an object is made the first time the transaction accesses an object and it contains any changes the transaction has made to the object. This code first checks if the transaction has already accessed the object and therefore contains a transaction local copy of the object. If not the code checks for a

```
1  public class Matrix {
2    double[][] m;
3    public Matrix(int M, int N) {
4      m = shared new double[M][N];
5    }
6  }
7  public class MatrixMultiply extends Thread {
8    public MatrixMultiply(int x0, int x1, Matrix a, Matrix btrans, Matrix c) {
9      this.a=a;
10     this.btrans=btrans;
11     this.c=c;
12     this.x0=x0;
13     this.x1=x1;
14   }
15   Matrix a, btrans, c; int x0, x1;
16   public void run() {
17     atomic {
18       for(int i=x0; i<x1; i++) {
19         for(int j=0; j<c.m[i].length; j++) {
20           double prod=0;
21           for(int k=0; k< a.m[i].length; k++)
22             prod+= a.m[i][k]*btrans.m[j][k];
23           c.m[i][j]=prod;
24         }
25       }
26     }
27   }
28   public static void parallelmult(int numthreads, ...) {
29     Wrapper thread[]=new Wrapper[numthreads];
30     atomic {
31       Matrix a=shared new Matrix(L,M);
32       Matrix btrans=shared new Matrix(N,M);
33       Matrix c=shared new Matrix(L,N);
34       for(int i=0;i<numthreads;i++) {
35        int low=i*(L/numthreads);
36        int high=(i==numthreads-1)?L:(i+1)*(L/numthreads);
37        thread[i]=new Wrapper(shared new MatrixMultiply(low, high, a, btrans, c));
38       }
39     }
40     for(int i=0;i<numthreads;i++)
41       thread[i].wrap.start(machine[i]);
42     ...
43   }
44 }
```

Fig. 1. `MatrixMultiply` Example

cached copy on the local machine. Our implementation caches objects to avoid communication overhead. If the cache does not contain the object, then the code contacts the machine that holds the *authoritative copy* of the object. The authoritative copy contains all the committed changes and is stored on the machine that allocated the object.

In line 22, the `run` method accesses `Matrix` objects through the `a` and `btrans` fields. To implement these field accesses, the generated code reads the object identifier from the `m` field, locates the corresponding object, makes a transaction local copy, and points a temporary at the copy. Our compiler maintains the invariant that if a transaction uses a variable and that variable references a shared object, the variable points to the transaction local copy for the duration of the transaction.

When the transaction completes, the `run` method calls the runtime to commit the transaction. A secondary benefit of transactions in our system is that accessing remote objects inside of transactions is cheaper than accessing them outside of transactions because the transactions enable our system to safely speculatively prefetch and cache objects without violating memory coherency.

2.2 Object Prefetching

The example accesses array objects that are unlikely to be available on the local machine. Our system uses symbolic prefetching to hide the latency of such accesses. Consider the expression `a.m[i][k]` in line 22 of Figure 1. The traditional approach to prefetching this expression would use three consecutive round trips over the network to prefetch `a`, then `a.m`, and finally `a.m[i]`. Symbolic prefetches instead bundle a starting object identifier along with a symbolic expression for a path through the heap into a single message to the remote machine. For the example expression, our approach would generate a single symbolic prefetch expression `a.m[i]` that begins with the object identifier stored in `a` and then specifies the path defined by the offset of field `m` and the $\texttt{i}^{th}$ array element. When the symbolic prefetch `a.m[i]` is executed, the system sends a message containing the object identifier stored in `a` along with the symbolic expression `.m[i]` to the remote machine. If the remote machine contains all three objects it sends all of them at once, and therefore all three objects can be prefetched in a single round trip communication.

Prefetch Analysis. The prefetch analysis computes at each program point a set of symbolic prefetch expressions that contain (1) a heap path to specify the objects to prefetch and (2) an estimation of the probability that the program will access these objects. The prefetch compiler analysis is structured as a standard backwards fixed-point computation over the control flow graph. At program statements that access a field, the analysis creates a prefetch expression for the field access and associates an initial probability of 100% with it. For example, in line 22 the analysis would generate the prefetch expression `btrans.m[j]` (and `a.m[i]`) with a 100% probability because the statement reads those expressions. Note that `k` does not appear in the prefetch expression because `b.m[j][k]` does not refer to an object.

When the prefetch analysis propagates this prefetch expression backwards it hits the `for` loop on line 21. To propagate the prefetch expression beyond this loop the analysis uses a 90% loop conditional branch probability. As the expression propagates the

analysis calculates new probabilities by multiplying the old probabilities with the loop condition probability. As a result, the analysis computes the prefetch expression `b.m[j]` with a 90% probability after line 20. When the analysis propagates the `btrans.m[j]` expression to line 19, the variable increment `j++` along with the conditional branch for the loop causes the analysis to rewrite the expression `btrans.m[j]` into the expression `btrans.m[j+1]` with an 81% probability at line 24. The analysis computes an 81% probability for the prefetch expression `btrans.m[j+1]` because the analysis propagates it through the two loop conditions at lines 19 and 21 (each with a 90% probability). The fixed-point computation continues until it satisfies the convergence criteria described in Section 5.2.

Prefetch Placement. After the prefetch analysis computes prefetch expressions for all program points, the compiler computes where to generate code for the symbolic 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 control flow edges where the prefetch probabilities cross a threshold. By default we set this threshold at 30%. For example, the compiler might place a prefetch for the analysis-generated expression `btrans.m[j+10]` after line 19 as its probability will drop below 30% if it propagates across the loop body again; and would place prefetches for analysis-generated expressions `btrans.m[0]`, `btrans.m[1]`, `btrans.m[2]`, `btrans.m[3]`, ..., and `btrans.m[9]` before line 18 as this is the first statement of the transaction.

3 System Overview

Our distributed transactional memory is object-based — data is accessed and committed at the granularity of objects. Our system uses a partitioned global address space (PGAS) programming model [12] with two classes of objects: local and shared objects. Shared objects are assigned a globally unique object identifier when they are allocated. The object identifier is then used to reference the object.

Each shared object has an *authoritative copy* that contains the most recent committed version and resides on the machine that allocated the object. When a transaction accesses a shared object, it makes a transaction local copy of the object. The transaction then performs all updates to the transaction local copy.

Each shared object has a version number, which is incremented when a transaction commits an update to the object. Our system uses the version numbers to check if the transaction local copies of objects are up to date when committing. We use a standard two phase transaction commit protocol [13] with commit-time locking and validation.

Each machine contains an object cache that can cache recently accessed objects. There is no guarantee that these objects are up to date. However, a best effort invalidation approach ensures with high probability that stale object copies are removed from the cache. The commit procedure ensures that committed transaction always access the latest versions of objects.

We use an optimistic approach to commit transactions. It is possible for some transactions to end up aborting repeatedly with prefetched objects that could be more likely

to be stale with eager prefetching. Our system can admit such zombie transactions (transactions that have accessed stale objects) . As our system is type safe and its distributed nature increases the cost of guaranteeing that a transaction always operates on consistent snapshots, we use a sandboxing approach that validates a transaction's read set periodically or upon runtime errors.

4 Programming Model

Our system extends Java with several language constructs designed to support distributed transactional memory. We add the `atomic` keyword for declaring 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. The shared memory extensions are similar to those present in Titanium [12].

The `shared` keyword can be used as a modifier to the `new` allocation statement to declare that an object should be allocated in the shared heap. Object fields in shared objects can only reference other shared objects. Local objects can 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 in that field's declaration.

In general, methods are polymorphic in whether their parameters are shared. The developer may desire that method has different behavior depending on whether its parameters are shared. Our compiler supports creating different method versions for shared and local objects — the shared version is designed by the `shared` keyword and the local version by the `local` keyword. The compiler uses a flow-sensitive, data-flow analysis to infer for each program point whether a variable references a shared object or a local object. The compiler uses the analysis results to generate specialized versions of methods for each calling context.

5 Prefetching

Traditional address-based prefetching approaches were largely designed to hide the latency of the local memory system — addressed-based prefetching incurs large latencies when accessing remote linked data structures because the computation must compute an object's address before prefetching it. In effect this requires waiting for a round trip over the network for each object to be accessed in the linked data structure.

Our prefetching approach eliminates the need to know an object's address prior to prefetching it. Symbolic prefetches describe paths through the heap that traverse the objects to be prefetched. Symbolic prefetches have the form:

$$\textit{symbolic prefetch} := \textit{base object identifier}(.\textit{field} \mid [\textit{integer}])^*$$

The base object identifier component of the symbolic prefetch holds the object identifier of the first object to be prefetched. The list of fields and array indices define a path through the heap from the first object. We combine the runtime technique with a compiler analysis that automatically generates prefetches for arbitrary structures and arrays. Symbolic prefetches allow our system to prefetch multiple objects along a chain of references with a single round-trip network communication. Consider the following code:

```
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, traversing a remote linked list of length n requires n consecutive round-trip communications. If we add a prefetch for `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[1]. The example prefetch enables the `search` method to potentially execute five times faster. Longer symbolic prefetches 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. In this example the symbolic prefetch effectively decreases the latency of accessing the remote objects by 80%.

5.1 Prefetch Analysis

We have developed an unsound, intraprocedural static analysis in our compiler that uses a simple probabilistic model to generate prefetches for the objects that the program may access and to estimate the probabilities that the objects represented by the prefetch expressions will be accessed. The probabilistic model is naïve and makes assumptions of independence that are not true in general. However, the object access frequencies need not be precise and simply provide an approximation of the program's data access patterns. It is safe for the analysis to be unsound because prefetches do not affect the program's correctness.

The analysis is a backwards program analysis that computes a set of prefetch tuples $\mathcal{P} \subseteq \Phi \times \mathbb{R}$ containing a symbolic prefetch expression $\phi \in \Phi$ and a corresponding probability $d \in \mathbb{R}$ for each program point. Each symbolic prefetch 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 $\mathcal{I} = .\mathit{offset} \mid [\mathit{index}]$. Each array index $\mathit{index} = \mathit{tmp}_0 + ... + \mathit{tmp}_{m-1} + c$ is a sum of m temporary variables represented by the terms tmp_i and a constant offset c.

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

Figure 2 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 symbolic prefetches with an associated probability of 100% and rewrite any symbolic prefetches that contain the destination variable. The transfer functions for statements that make assignments, write to fields, or write to array elements rewrite symbolic prefetches that begin with the same variable and field or array index. Figure 3 presents the `REPLACE` function that rewrites the symbolic prefetch. The $\texttt{REPLACE}(\phi_1, \phi_2, \mathcal{P})$ function takes all prefetch tuples in $\mathcal{P}$ that contain a symbolic prefetch expression with the prefix ϕ_1 and replace that prefix with ϕ_2. The `REPLACE` function simply copies the

[1] 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.

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

Fig. 2. Transfer Functions

$$
\begin{aligned}
\mathtt{REPLACE}(\phi_1, \phi_2, \mathcal{P}) &= \mathtt{COMBINE}(\mathtt{REWRITE}(\phi_1, \phi_2, \mathcal{P})) \\
\mathtt{REWRITE}(\phi_1, \phi_2, \mathcal{P}) &= \{\langle \pi(\phi, \phi_1, \phi_2), d \rangle \mid \langle \phi, d \rangle \in \mathcal{P}\} \\
\mathtt{COMBINE}(\mathcal{P}) &= \{\langle \phi, d \rangle \mid \{d_0, d_1, ..., d_{n-1}\} = \mathcal{P}(\phi), \\
&\qquad d = 1 - (1 - d_0)(1 - d_1)...(1 - d_{n-1})\} \\
\pi(\phi, \phi_1, \phi_2) &= \begin{cases} \phi_2 \mathcal{I}_0 ... \mathcal{I}_n & \text{if } \phi = \phi_1 \mathcal{I}_0 ... \mathcal{I}_n \\ \phi & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 3. Equation for the REPLACE Function

prefetch tuples with symbolic prefetch expressions whose prefixes do not match. One potential issue is that a rewritten symbolic prefetch may match an existing symbolic prefetch. The `COMBINE` function computes the new probability making the assumption that the probabilities for the symbolic prefetches are independent. We have omitted the `REPLACE` functions for index variables for space reasons.

Our analysis associates a probability with each conditional branch. By default, we assume that loops continue with a 90% probability and other conditional branches take the true branch with a 50% probability. The prefetch tuples for a given exit edge of a conditional branch are weighted by the branch probability for that exit. The two sets of symbolic tuples are merged and if two symbolic tuples have identical symbolic prefetch expressions they are replaced with a new symbolic tuple with a probability equal to the sum of their probabilities. We note it is straightforward to extend the analysis to use branch statistics collected from profiling.

5.2 Termination of Prefetch Analysis

While the transfer functions are monotonic, the partial order on $\mathcal{P}$ is not a lattice because there is no top element. Therefore, the standard termination arguments for dataflow analysis do not apply. We extend our prefetch analysis to ensure termination. One issue is that the analysis can generate symbolic prefetch expressions of unbounded length. We address this issue by introducing a minimum symbolic prefetch probability μ. If a prefetch tuple has a probability less than μ at a program point, the analysis drops that

prefetch tuple. A second issue is that the analysis can converge slowly as the analysis makes increasingly smaller increments to the symbolic prefetch 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 that 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. Our implementation, therefore, allows the developer to specify a probability threshold σ. Our compiler places symbolic prefetches at the program point that the probability for a prefetch expression crosses this threshold. By default we selected σ to be 30%.

We instrument the prefetch analysis to record the mapping $\gamma(\phi, E) \rightarrow \phi'$ which maps the symbolic prefetch ϕ at the source of the edge E to the corresponding symbolic prefetch ϕ' at the target of the edge E. Prefetches are placed on edges where the probability of using the objects specified by a symbolic prefetch crosses the developer specified threshold. Formally, we state this criterion by defining the function τ below to check if a prefetch crosses the probability threshold at the an edge E:

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

Simply using this threshold-crossing criteria to generate prefetch calls can place redundant prefetch calls. We therefore extend our static analysis to check whether the symbolic prefetch is redundant. We define the set S_N at each program point to be the set of symbolic prefetches that have been prefetched when the program executes the statement at node N. This set is the intersection of the set of prefetched symbolic prefetches along each incoming edge E to node N. We split the prefetched symbolic prefetches into two components: $\mathcal{S}_E$ is the set of symbolic prefetches 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 symbolic prefetches in $\delta_E - \mathcal{S}_E$, the set of symbolic prefetches that cross the threshold but have not already been prefetched.

5.4 Prefetch Runtime Mechanism

Our analysis generates prefetch calls at each prefetch site. A prefetch call takes as input the site identifier of the prefetch call, the number of prefetches, an array of base

object identifiers for each prefetch, an array of lengths of each prefetch, and an array of unsigned shorts that stores the sequence of field offsets and array indices for every prefetch at that site. The runtime system differentiates between fields and array indices based on the type of the previous object in the path.

Processing a prefetch starts by locally looking up the base object identifier component of the prefetch request in both the local distributed heap and the object cache — in many cases some of the objects in the request are available locally. If the object is found locally, the local runtime uses the field offset (or array index) to look up the object identifier of the next object in the path and removes the first offset value from the symbolic prefetch. 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 object identifier.

The runtime groups the prefetch requests by the machine that is authoritative for the base object identifier. The local machine next sends the prefetch requests to the remote machines. Each request contains the machine identifier that should receive the response. 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`. If one request subsumes another request, the runtime drops the subsumed request.

When the remote machine receives a prefetch request it begins by looking up the base object identifier in its local distributed heap and then (optionally) if necessary in its object cache. Once it finds the object, it looks up the next object identifier by using the field offset or array index from the symbolic prefetch. It repeats this process until it has served the complete request. As it serves the request it sends the copies of the objects to the machine that initiated the prefetch request. When the local machine receives objects it adds the objects to its object cache.

If the remote machine does not have an object specified by the symbolic prefetch, it forwards the remainder of the prefetch request along with the machine identifier of the machine that originated the request to the machine that holds the authoritative copy of the object. Forwarding is necessary because a single machine may not have all the objects for the symbolic prefetch.

Our prefetching implementation includes a heuristic optimization for frequently updated objects. Consider for example a shared hash table data structure. The hash table's array may be frequently updated and reference objects on different machines, and therefore will likely introduce an extra forwarding step in processing the prefetch. If the local machine does not contain a valid copy of the object, our implementation checks whether the local machine contains an invalid copy of an object. The intuition is that while invalid objects cannot safely be used to execute transactions, their fields or elements often contain the correct reference. If an invalid object is present locally, the runtime uses the invalid object copy to skip forwarding while processing the prefetch. This heuristic avoids the network latency that would be incurred if the request has to be forwarded and is safe because the actual transaction never accesses the invalidated objects.

6 Dynamic Prefetching Optimizations

In some applications, transactions repeatedly access the same objects. For example, matrix multiplication accesses some of the same objects in different iterations of the outermost loop. In this case, a prefetch call will repeatedly prefetch the same objects. Processing repeated prefetches 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 accesses new objects and phases in which it accesses the same objects. We therefore support a mechanism that dynamically shuts down prefetch sites when they stop providing benefits. This mechanism allows the application to benefit from prefetching while minimizing the overhead.

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 eight 3.06 GHz Intel Xeon servers running Linux and connected through gigabit Ethernet. Our implementation contains over 120,000 lines of code and is available for download along with all benchmarks at `http://demsky.eecs.uci.edu/compiler.php`. To evaluate our Java compilation, we have performed experiments to compare the code outputted by our compiler with hand-developed C code and have found the performance to be similar. We present results for several microbenchmarks, five scientific benchmarks, and two distributed benchmarks. We report results for: `Base`, versions without caching or prefetching, and `Prefetch`, versions with both caching and prefetching. For the scientific benchmarks, we report results for `1J`, single-threaded reference Java implementations compiled into C code. For the distributed benchmarks, we also report results for `Caching`, versions with only caching enabled. We average execution times over ten executions.

We performed all of the experiments on a LAN-based test bed system. The relative benefits of eliminating message exchanges through symbolic prefetches increase as the round trip message latency between machines increases. Therefore, we expect that symbolic prefetches will yield much larger benefits for wide area networks because they have significantly larger latencies than our LAN-based test bed system.

7.1 Microbenchmarks

The microbenchmarks are intended to quantify the benefits of prefetching. We present results from a three-dimensional array traversal microbenchmark to measure the performance gains from prefetching objects for regular access patterns. The array

microbenchmark sums all of the elements in a $10 \times 32{,}000 \times 4$ array of integers that is located on a remote machine. Prefetching improves the performance of this microbenchmark by a factor of $13.31\times$. The linked list microbenchmark traverses a remote linked list with 1,000,000 nodes. Prefetching improves the performance of this benchmark by $4.54\times$. The speedup of the linked list microbenchmarks is limited because each prefetch must traverse many linked-list nodes to prefetch one new linked list node.

7.2 Distributed SpamFilter

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 2,457 lines of code [14]. 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.

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.

Figure 4 presents the results for the distributed spam filter benchmark. Our workload presents each spam filter client with 1,000 emails. Note that our workload holds the work constant per client machine. Therefore, the total amount of work increases as we add more clients. The observed increase in execution time has two primary causes: (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 to quantify the benefits of caching versus prefetching. Prefetching provides speedups up to $4.54\times$ relative to the base version and $1.49\times$ relative to the caching version. Prefetching and caching hide up to 98% of remote reads relative to the base version and prefetching hides up to 87% of remote reads relative to the caching version. Figure 5 presents the abort rate for transactions running on multiple clients for this benchmark. Up to 9% of transactions abort due to conflicts. We omit Java comparisons because we do not have a Java version of this benchmark.

7.3 Distributed Multiplayer Game

In the multiplayer game benchmark clients play the roles of tree planters and lumberjacks. The benchmark contains 1,420 lines of code. The game is played on a map by planters and lumberjacks — planters plant trees while lumberjacks cut trees. Both the planters and lumberjacks choose a location in the map to either plant a tree or cut one down and take the shortest path to the destination. The clients use the A* graph search algorithm to plan routes. The clients introduce contention in this benchmark when they attempt to plant or remove trees in the same region of the map. If a client accessed the

part of the map updated by another client, the transactional version aborts the transaction surrounding that move. The Java version is optimized to only recompute a client's move if another client's move makes client's original move illegal.

SpamFilter	Base	Caching	Prefetch
1	1.50s	—	—
2	13.41s	4.39s	2.95s
4	17.12s	5.17s	3.83s
8	20.97s	6.40s	5.32s

Fig. 4. SpamFilter Results

	SpamFilter		Game		SOR	
Thds	Base	Prefetch	Base	Prefetch	Base	Prefetch
2	4.2%	5.8%	0.2%	0.2%	0.0%	0.2%
4	6.1%	8.3%	1.0%	0.2%	0.2%	0.5%
8	8.1%	9.0%	2.5%	1.0%	0.7%	1.2%

Fig. 5. Abort rate

Figure 6 presents results for the multiplayer gaming benchmark. The game is played on a map of size 400×100 for 512 rounds. 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. This benchmark accesses the same data in different transactions and therefore caching provides a benefit. Prefetching and caching provided speedups of up to 26%. Prefetching and caching hide up to 77% of remote reads relative to the base version and prefetching hides up to 26% of remote reads relative to the caching version.

Game	Java	Base	Caching	Prefetch
1	46.78s	8.06s	—	—
2	51.99s	10.22s	9.99s	9.76s
4	71.54s	12.75s	11.35s	10.93s
8	97.22s	16.73s	13.69s	12.34s

Fig. 6. Multiplayer Game Results

The base version is faster than the Java version because of the way the A* algorithm accesses the map. In the Java version, the server transfers the map at the beginning of each round to make the code manageable. The transactional version only transfers the parts of the map that are needed. We see a $7.87\times$ speedup for the 8 threaded prefetching version relative to the 8 threaded Java version.

Figure 5 presents the abort rate for transactions in this benchmark.

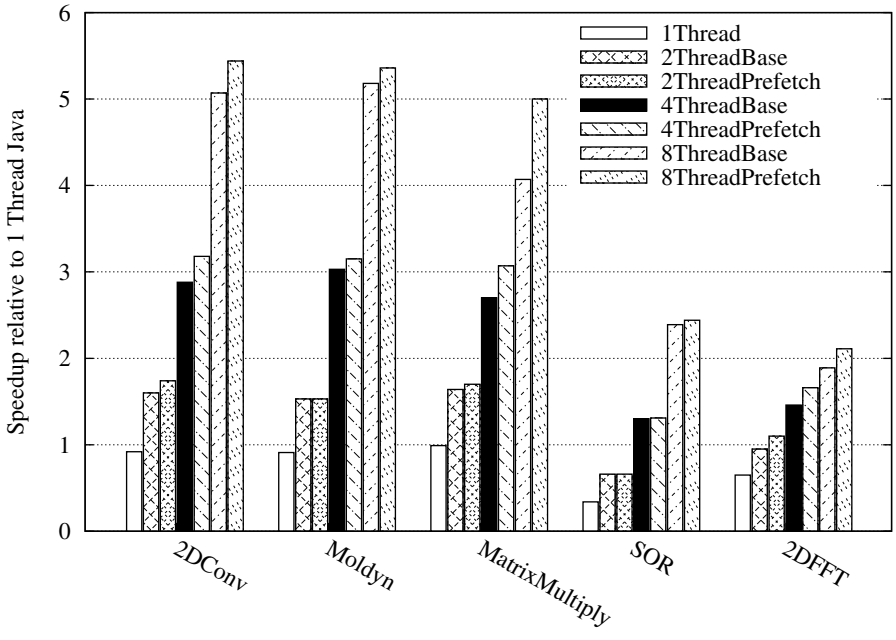

Fig. 7. Scientific Benchmark Speedups (Higher is Better)

7.4 Scientific Benchmarks

We next present results from five scientific benchmarks: 2DConv, Moldyn, MatrixMultiply, SOR, and 2DFFT. These benchmarks do not access the same versions of objects in multiple transactions and therefore caching adds no benefit beyond the base version. We therefore omit caching results for the scientific benchmarks and only report results for the base and the prefetching versions. Figure 7 presents the speedups for all scientific benchmarks relative to single-threaded Java version and Figure 8 presents the absolute execution times.

2DConv. The 2D Convolution benchmark computes the application of a mask to a 2D image. Each machine computes a region of the output image in parallel. The base version contains 1,015 lines of code. The output and input matrices are shared objects in our experiment with dimensions 10,000×1,000 with a convolution mask of 13×13. Figure 8 presents results for the 2DConv benchmark. The 8 threaded prefetching version provides a speedup of 5.44× over the single-threaded Java version. Prefetching hides nearly 100% of remote reads and provides speedups of up to 9%.

Molecular Dynamics. Moldyn is from the Java Grande benchmark suite [15]. The base version contains 1,172 lines of code. Moldyn models the interaction of molecular particles. We used 8,788 particles and 50 iterations. The 8 threaded prefetching version

provides a speedup of 5.35× over the single-threaded Java version. This benchmark accesses a small number of remote objects and therefore we observe speedups of up to 4% from prefetching even though prefetching hides up to 79% of remote reads.

	2DConv		MolDyn		Matrix Multiply		SOR		2DFFT	
	Base	Prefetch	Base	Prefetch	Base	Prefetch	Base	Prefetch	Base	Prefetch
1J	34.20s	—	103.10s	—	96.00s	—	239.05s	—	16.29s	—
1	37.11s	—	113.92s	—	96.48s	—	646.86s	—	25.09s	—
2	21.32s	19.71s	67.57s	67.46s	58.62s	56.31s	362.29s	360.98s	17.08s	14.81s
4	11.86s	10.75s	34.02s	32.72s	35.53s	31.28s	183.66s	183.13s	11.15s	9.79s
8	6.74s	6.29s	19.91s	19.26s	23.61s	19.19s	100.17s	98.00s	8.61s	7.69s

Fig. 8. Scientific Benchmark Results

Matrix Multiply. The matrix multiplication benchmark implements the standard matrix multiplication algorithm for matrix A and matrix B to get the product matrix C. This version computes fifty 650×650 product matrices. All matrices are shared objects. The computation of each product matrix is partitioned over multiple machines. The 8 threaded prefetching version provides a speedup of 5.00× over the single-threaded Java version and prefetching improves performance by 19%. Prefetching hides up to 88% of remote reads for this benchmark.

SOR. The SOR benchmark is from the Java Grande benchmark suite [15]. SOR contains 737 lines of code. It performs 200 iterations of an over-relaxation algorithm on a 8,000×8,000 grid. The 8 threaded prefetching version provides a speedup of 2.43× over the single-threaded Java version. The one machine distributed transactional version is slower than the single-threaded Java version because each machine must locally copy many large array objects to implement transactions. This 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 5 presents the abort rate for transactions running on multiple machines. Prefetching only improves the performance up to 2% even though prefetching hides up to 83% of remote reads because this benchmark accesses a small number of remote objects.

2DFFT. The 2DFFT benchmark is a two-dimensional fast Fourier transform. The base version contains 889 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 FFT for five matrices. The FFT of each matrix is parallalized across all machines. The computation performs 1D FFT in parallel, a serial transpose, and then 1D FFT in parallel. The 8 threaded prefetching version provides a speedup of 2.11× 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 improves performance up to 13% and hides up to 99% of remote reads.

7.5 Prefetching Effectiveness

Figure 9 presents the hit percentage in the cache for the prefetching versions. We hit in the cache 87% of the time on average eliminating the majority of requests over the network. Figure 10 presents the total numbers of 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, the `Cache` version shows the number of remote reads with caching alone, and the `Prefetch` version shows the number of remote reads with prefetching. We omit results for the `Cache` version of the scientific benchmarks as they never access the same object in two transactions and therefore the `Cache` versions perform exactly the same number of remote reads as the `Base` versions. Note that reducing remote reads eliminates the latency from an equivalent number of round trips.

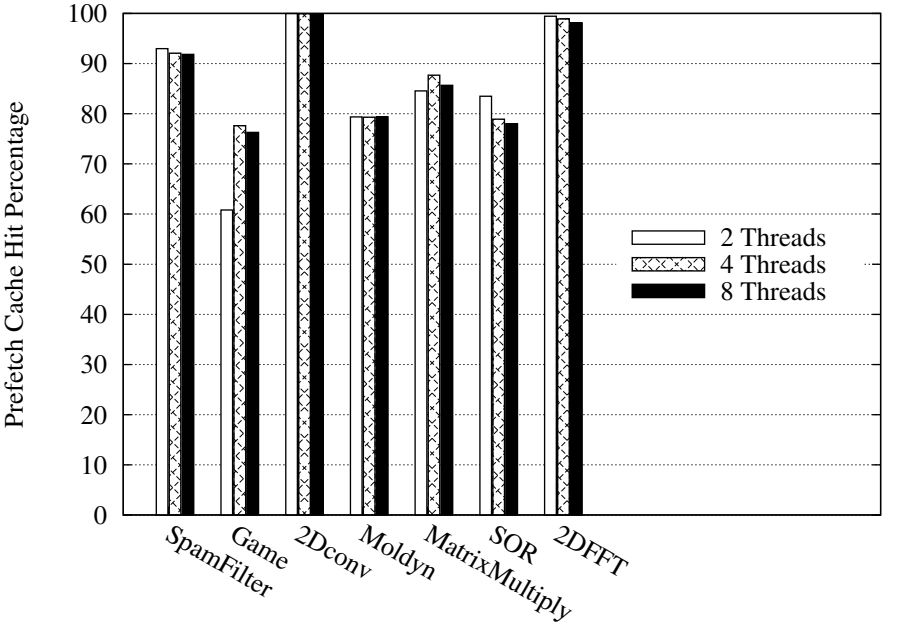

Fig. 9. Prefetching Hit Percentage

Thds	SpamFilter			Game			2DConv		MolDyn		Matrix Multiply		SOR		2DFFT	
	Base	Cache	P	Base	Cache	P	Base	P	Base	P	Base	P	Base	P	Base	P
2	94037	18059	2197	10335	4312	4049	10018	2	1740	359	32736	5059	1205	201	15029	85
4	104303	21118	3907	11791	3632	2699	5018	1	1740	359	36936	4541	1276	269	7529	82
8	109220	22929	5525	13115	4398	3194	2518	2	1740	359	35816	5136	1301	289	3780	71

Fig. 10. Remote Read Results - Scientific Benchmarks (P = Prefetch)

7.6 Tunable Parameters

We studied the sensitivity of the benchmarks' performance to the tunable parameters for the loop conditional branch probability and the probability threshold σ. For the first experiment we executed all of the benchmarks for several different branch probabilities in the range between 50% and 96%. We found that all benchmarks achieved maximum performance for probabilities larger than 90% and that in this range performance was relatively insensitive to the exact value. We conducted a similar experiment for the user defined probability threshold σ. We varied the threshold between 50% for less aggressive prefetching and 5% for more aggressive prefetching. We found that we achieved maximum performance for thresholds near 30% and that near this value performance was relatively insensitive to the exact value.

8 Related Work

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

8.1 Distributed Transactional Memory Systems

We next survey the large body of existing research on distributed transactional memory. Symbolic prefetching is applicable to most of these systems and would greatly improve their performance by hiding much of their latency to access remote objects.

Researchers have explored distributed transactional memory [16,17] systems as a mechanism for providing 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 location [18]. Hastings extended Camelot distributed system to support transactions through a lock-based approach [19]. Ahn et al. developed a lock-based distributed shared memory with support for transactions [20]. LOTEC is another lock-based transactional distributed shared memory [21]. All of these implementations incur round trip network latencies whenever the application accesses a remote object because the machine must first communicate to acquire a lock.

DiSTM is a distributed transactional memory system [5]. Its commit process checks whether any running transactions conflict with the current transaction and therefore may not scale well. Anaconda is a distributed transactional memory system that uses a distributed commit algorithm [3]. 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.

D^2STM is a fault-tolerant distributed transactional memory system [1]. 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 beginning of the committing transaction.

Manassiev et al. introduced a version-based transactional distributed shared memory that replicates all program state on all machines [8]. Sinfonia allows machines to share data in a fault-tolerant and scalable manner using mini-transactions [22].

Bocchino et al. developed a word-based software transaction memory [7]. Herlihy and Sun proposed a distributed transaction memory for metric-space networks [6]. Their design moves objects to the local machine before writing to the object.

8.2 Prefetching

Researchers have developed several techniques for prefetching recursive data structures. Luk and Mowry propose to greedily prefetch object fields, automatically add prefetch pointers to objects that point to objects to prefetch, and linearize recursive data structures when possible [23]. Greedy prefetches require knowing the address of an object. Prefetch pointers do not help with the initial traversal of data structures. Linearizing is only applicable if the creation order is the same as the traversal order. Cahoon and McKinley proposed an analysis for software prefetching [24]. 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 [25]. Wu et al. [26] and Inagaki et al. [27] propose stride prefetching for irregular references. However, in distributed systems the latency of accessing remote memory is likely to be longer than the time that can be filled with useful work.

Researchers have explored communication optimizations for distributed computations. Zhu and Hendren combine multiple reads into a single block [28]. Because their approach requires that the address of objects to be read is known, it incurs a 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 [29]. An issue with thread migration is that it cannot provide efficient simultaneous access to data that spans multiple machines.

Gupta proposes a naming scheme for objects in data structures to enable fast traversals of remote data structures [30]. In this approach many data structure updates require renaming all the objects in a data structure and propagating changes to all machines.

Speight uses a dynamic prediction-based prefetching approach [31]. Joseph and Grunwald use Markov predictors to generate prefetches on a single machine [32]. Ferdman and Falsafi store access sequences and then stream the addresses from these access sequences into a chip's cache [33]. These prefetching approaches typically require object accesses that repeat during execution — as our caches for remote objects can be much larger than CPU caches it is likely that such objects are already in the cache.

8.3 Distributed Systems

Researchers have developed distributed object stores that present a transparent object-oriented view of storage. Such systems are designed to simplify scalable service in cluster environments. TODS [11] is a cluster object storage system. Thor [10] provides a distributed object-oriented database system that supports object navigation and therefore incurs additional message round trips for each commit. Ceph [9] is a scalable object based storage system that uses metadata server (MDS) clusters for managing a distributed file system. We expect that symbolic prefetching along with our prefetch analysis would improve the performance of these distributed object stores.

Distributed shared memories were intended to provide developers with a simple shared memory abstraction on message-passing machines. While earlier systems

provide a strict consistent memory model, other sophisticated approaches [34,35,36] achieve higher performance by weakening the memory consistency guarantees and reduce coherence overheads. Developing software for weaker memory models requires the developer to understand complicated consistency properties.

9 Conclusion

Transactional memory provides a powerful new approach to developing distributed applications — it provides information about the granularity of a thread's accesses to distributed data structures and a mechanism to enable speculative optimizations. We have presented an analysis that generates symbolic prefetches for objects. Our prefetch analysis and runtime provides developers with a simple programming model for writing applications in a distributed system. Our benchmark results show that our system provides excellent performance and that prefetching can hide the latency of the majority of remote object accesses.

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

References

1. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (2009)
2. Carvalho, N., Cachopo, J., Rodrigues, L., Silva, A.R.: Versioned transactional shared memory for the FénixEDU web application. In: Proceedings of the 2nd Workshop on Dependable Distributed Data Management (2008)
3. Kotselidis, C., Luján, M., Ansari, M., Malakasis, K., Khan, B., Kirkham, C., Watson, I.: Clustering JVMs with software transactional memory support. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (2010)
4. Zhang, B., Ravindran, B.: Relay: A cache-coherence protocol for distributed transactional memory. In: Proceedings of the 2009 International Conference on Principles of Distributed Systems (December 2009)
5. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: DiSTM: A software transactional memory framework for clusters. In: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington, DC, USA, pp. 51–58 (2008)
6. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 324–338. Springer, Heidelberg (2005)
7. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (2008)
8. Manassiev, K., Mihailescu, M., Amza, C.: 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)
9. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 307–320 (2006)

10. Liskov, B., Castro, M., Shrira, L., Adya, A.: Providing persistent objects in distributed systems. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, p. 230. Springer, Heidelberg (1999)
11. Jin, C., Zheng, W., Zhou, F., Wu, Y.: A distributed persistent object store for scalable service. SIGOPS Operating Systems Review 36(4), 36–49 (2002)
12. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Ham, S.G., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. Concurrency: Practice and Experience 10(10-13) (September-November 1998)
13. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1993)
14. Albrecht, K., Burri, N., Wattenhofer, R.: Spamato - an extendable spam filter system. In: 2nd Conference on Email and Anti-Spam (CEAS) (July 2005)
15. Smith, L.A., Bull, J.M., Obdrzalek, J.: A parallel Java Grande benchmark suite. In: Proceedings of SC 2001 (2001)
16. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (August 1995)
17. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.: 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)
18. Bodorik, P., Smith, F.I., J-Lewis, D.: Transactions in distributed shared memory systems. In: Proceedings of the Eigthth International Conference on Data Engineering (February 1992)
19. Hastings, A.B.: Distributed lock management in a transaction processing environment. In: Proceedings of the Ninth Symposium on Reliable Distributed Systems (October 1990)
20. Ahn, J.H., Lee, K.W., Kim, H.J.: 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)
21. Graham, P., Sui, Y.: LOTEC: A simple DSM consistency protocol for Nested Object Transactions. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (1999)
22. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: A new paradigm for building scalable distributed systems. In: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (2007)
23. Luk, C.K., Mowry, T.C.: Automatic compiler-inserted prefetching for pointer-based applications. IEEE Transactions on Computers 48(2), 134–141 (1999)
24. Cahoon, B., McKinley, K.S.: 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)
25. Roth, A., Moshovos, A., Sohi, G.S.: 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)
26. Wu, Y., Serrano, M.J., Krishnaiyer, R., Li, W., Fang, J.: Value-profile guided stride prefetching for irregular code. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 307–324. Springer, Heidelberg (2002)
27. Inagaki, T., Onodera, T., Komatsu, H., Nakatani, T.: Stride prefetching by dynamically inspecting objects. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 269–277 (2003)
28. Zhu, Y., Hendren, L.J.: Communication optimizations for parallel C programs. In: Proceedings of the 1998 Conference on Programming Language Design and Implementation (1998)
29. Rogers, A., Carlisle, M.C., Reppy, J.H., Hendren, L.J.: Supporting dynamic data structures on distributed-memory machines. ACM Transactions on Programming Languages and Systems 17(2), 233–263 (1995)

30. Gupta, R.: SPMD execution of programs with dynamic data structures on distributed memory machines. In: Proceedings of the 1992 International Conference on Computer Languages (April 1992)
31. Speight, E., Burtscher, M.: 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)
32. Joseph, D., Grunwald, D.: Prefetching using markov predictors. In: Proceedings of the 24th International Symposium on Computer Architecture (1997)
33. Ferdman, M., Falsafi, B.: Last-touch correlated data streaming. In: IEEE International Symposium on Systems and Software (April 2007)
34. Keleher, P., Cox, A.L., Dwarkadas, S., Zwaenepoel, W.: TreadMarks: Distributed shared memory on standard workstations and operating systems. In: Proceedings of the USENIX Winter 1994 Technical Conference (1994)
35. Bershad, B.N., Zekauskas, M.J.: Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. In: Compcon 1993 (1993)
36. Bennett, J.K., Carter, J.B., Zwaenepoel, W.: Munin: Distributed shared memory based on type-specific memory coherence. In: Proceedings of the Second Symposium on Principles and Practice of Parallel Programming, pp. 168–176 (1990)