2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks

Path-Analytic Distributed Object Prefetching

Yang Luo, King Tin Lam, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong
{yluo, ktlam, clwang}@cs.hku.hk

***Abstract*—This paper presents our studies on the connectivity between objects and traversal behavior over the access paths among objects in order to devise profitable prefetching policies for object-based distributed systems. We propose a profiling strategy that can classify classes and fields into a handful of generic types exhibiting distinctive and exploitable access patterns during the runtime. Based on the classifications, we propose an improved algorithm of object prefetching to select best candidates to prefetch under practical message size limits. We implement the methodology into our JESSICA2 distributed Java virtual machine and evaluate its effectiveness. Our experimental results show that our prefetching policies are able to eliminate over 93% cache coherence protocol messages and halve the execution time for fine-grained applications.**

Keywords-object sharing; prefetching; object access patterns; distributed Java virtual machine; distributed shared memory

I. INTRODUCTION

In most distributed systems, prefetching is a canonically useful technique to reduce data access latency by aggregating more data units into a single round-trip. If the amount of prefetched data sustains good spatial or temporal locality, the speculation will be positive and give lower amortized access cost. The usual data unit is a memory page or an object. Page-based prefetching has been well-studied in previous work including [1, 2, 3, 4]. In this paper, we focus on object prefetching techniques. Generally, there are two approaches to determining which objects to prefetch: *static* (by compiler-based analysis [5]) or *dynamic* (by exploiting *connectivity* between objects online [6]). With runtime information accessible, the latter is usually more flexible and accurate.

For object-based distributed shared memory (DSM) systems, prefetching shared objects for reducing data misses is critical to performance since the network latency involved in fetching an object from remote memory is much more costly than local heap access. As future accesses on prefetched objects are not all guaranteed to happen, we need to strike a balance on the prefetching amount to reduce the negative impact of sending unneeded objects. Prefetching accuracy can be calculated by accessed bytes over the total bytes of data prefetched (we consider all bytes of an object accessed if a single read/write on any of its fields is performed).

Significant speed improvement in recent years [12, 13] has enabled the unlocking of Java for high-performance computing over heterogeneous platforms. The distributed Java virtual machine (JVM) is a design of middleware platform that aims to provide multithreaded Java applications with transparent clustering support. Our previous work [6, 7] has built a virtual shared heap, the *global object space (GOS)*, to hide JVM boundaries from object access. Connectivity-based prefetching is known to be good in accuracy for the distributed JVM since objects in Java cannot be accessed without following reference fields. On top of a home-based cache coherence protocol, we proposed *object pushing* to prefetch objects in close proximity to the current or *root* object being accessed. In one implementation [6], upon receiving an object access request, the home node uses a breadth-first search to collect the reachable objects to push into the message buffer until its size reaches an optimal length. Another implementation [8] is to fetch only objects at the next immediate level under the current object graph without message length limit but carry out compression on the outgoing message to save network bandwidth.

However, we find that these rather conservative policies are still far from ideal for fine-grained applications that need more careful design on prefetching to achieve good aggregation effect while not worsening accuracy. In a heap comprising home and cache objects, problems could arise from stale cache or unstable connectivity among shared objects. Capturing correct interrelations between objects in such an environment to aid prefetching has to consider various issues including access locality, message size limit, search orientations (depth or breadth of traversal) and other system dynamics like home migrations [6].

This paper studies more advanced techniques to do object pushing based on *path analysis*. To be specific, an object is prefetched via following a reference field of another object, and we recognize the field a specific *path* of access. By recording access counts along different paths, we classify the relevant fields and classes into a few generic types with tailor-made policies. Our new prefetching can adapt in traversal order, breadth and depth of reachability for the best candidates to prefetch under practical message size limits. We implement the proposed methodology in an upgraded version of our JESSICA2 distributed JVM with a vastly revamped GOS that runs a more efficient protocol enforcing home-based lazy release consistency (HLRC) [11]. In our protocol, only home objects holding the latest copy can be prefetched along to the requester. Our system adopts a per-thread cache storage model that avoids the arrival of prefetched objects from overwriting dirty cache objects. By proper invalidation and flushing updates to homes across lock/unlock bounda-

978-0-7695-3908-9/09 $26.00 © 2009 IEEE
DOI 10.1109/I-SPAN.2009.131

IEEE computer society

ries, cache coherence is maintained regardless of our prefetching mechanism introduced. The system is tested with medium- to fine-grained applications, seeing promising performance gain after applying the *path-analytic* policies.

For the rest of this paper, Section 2 explains the difficult issues involved in object prefetching. Section 3 and 4 present our observed access patterns and the path-analytic prefetching algorithm respectively. Section 5 evaluates our implementation experimentally. We review the related work in Section 6 and conclude this paper in Section 7.

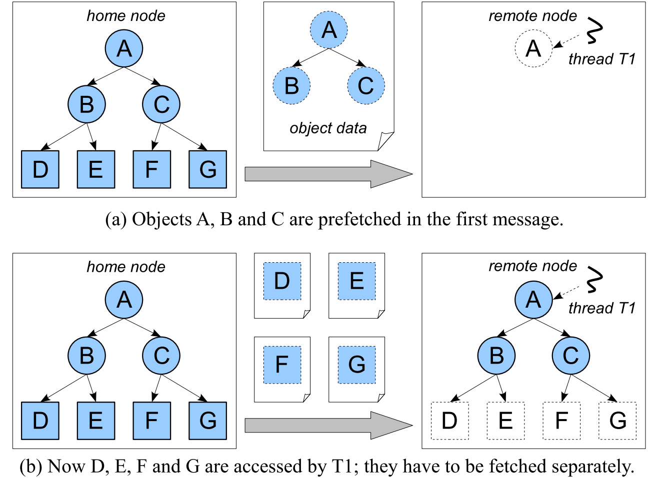
II. Challenges and Problems

In this section, we will explain several interrelated issues posing stern difficulty to distributed object prefetching.

1) Excessively fine granularity: fine-grained and object-intensive applications generally make prefetching harsher to work effectively. Taking Barnes-Hut, one of our tested benchmark programs, as an example: its basic data unit, the `Body` object, is composed of a few small-sized objects with a total size less than 100 bytes only. This implies at least 10-100 objects need to be packed in a single message if we are to achieve good aggregation effect on communication. For this application, our earlier scheme in [8] which prefetches only a single level deeper into the current object graph seems too conservative although it can maintain quite good accuracy. However, the dilemma is aggressive prefetching can benefit this case but hurt performance in another due to the negative impact of prefetching unneeded objects, which is equivalent to paying multiple times of access cost for a single memory location as well as network bandwidth. If the thread in execution will not access the prefetched objects eventually, this will on the contrary slow down the system. So we need more intelligent prefetching that can guarantee accuracy while going beyond conservative policies.

2) Biased paths: Different paths of access often indicate different usage in application-specific logics. It is a natural phenomenon that some paths are followed less frequently than others. A good example is that in many recursive data structures like trees and linked lists, some paths acting as "backlinks" (such as fields regularly named `node.parent`, `item.prev`, etc) are seldom accessed. Without analysis on path usage, we would hurt performance by following and prefetching along these rarely used paths. Apparently, we cannot make a guess at path usage based on field names and require a profiling method to discover such biasness to give more weight to those more frequently used objects.

3) Fragmentation effect: As mentioned, using a breadth-first search (BFS) to pack objects until hitting the message length limit is a rational policy to do prefetching. However, if the BFS stops immediately at a certain buffer threshold, this can leave many objects *isolated*, which have not been prefetched but their parents have been. These *orphans* have to be fetched individually later, in short messages, making the communication largely fragmented. Fig. 1 illustrates this problem. At the beginning (see (a)), thread $T1$ only has a



(a) Objects A, B and C are prefetched in the first message.

(b) Now D, E, F and G are accessed by T1; they have to be fetched separately.

Figure 1. Orphaned objects causing fragmented communication

reference to object A. When it accesses A, it asks the home of A for its latest copy. With simple buffer-limit prefetching, the home packs objects A, B and C altogether for $T1$, with references of D, E, F and G left behind. Later (see (b)), $T1$ accesses D, E, F and G in turns and sees they have to be fetched separately from the home because there is no more connectivity between them. Missing to include D to G in the first prefetch message creates four short-length but long-latency fetching round-trips over the cluster network.

III. Object Path Traversal Patterns

This section describes our classification scheme over classes and reference fields to characterize access behavior.

A. Class-Level Types (Object Types)

An object type or class can be broadly classified into the below generic types to characterize their access behavior.

1) Recursive class: a class is recursive if and only if it contains at least one reference field forming a *recursive path*. In other words, a recursive class would have *self-referential* pointers and most likely corresponds to common recursive data structures like trees and linked lists.

2) Terminal class: a class is terminal if and only if it has no outgoing paths of reference fields or all outgoing paths are either terminal or backward (to be explained).

3) Normal class: refer to simply all other object types.

B. Field-Level Types (Path Types)

A prefetching path, i.e. reference field, can be characterized into a few different types as follows.

1) Recursive field: a field is recursive if and only if its enclosing class and the target class it points to are the same or share a non-trivial common base (parent or super-) class, and it is not a backward field. Here a non-trivial common base class means classes other than `java.lang.Object`.

2) Backward field: a field is identified (by the system) as backward if most prefetched objects through that path are found duplicate. As mentioned in Section 2, some fields like

node.parent and item.prev acting as "backlinks" are rarely followed by threads. A simple unbiased scheme prefetching these paths as well within the same HLRC interval will see many duplicate objects prefetched from these "backward" paths. Our definition of backward fields is not given by a path's function which is difficult to guess. Instead, they are discovered by the system which observes paths' prefetching behavior. Backward paths receive the lowest priority during prefetching unless there are no other choices.

3) Terminal field: a field is terminal if and only if it points to a terminal class. Since terminal class cannot be fetched except via a single specific field, the default policy is to always follow it unless we encounter some very large objects (normally arrays in Java) as aggressively prefetching them usually impacts performance when there is no absolute guarantee on accuracy. Terminal paths are usually reached near the end of the prefetch traversal or search process.

4) Normal field: simply refer to all other fields.

TABLE I. EXAMPLE OF OBJECT AND PATH TYPES

```
1   class Link{    // RECURSIVE class
2      Molecule mol; // TERMINAL path (was NORMAL path)
3      Link prev;      // BACKWARD path (was RECURSIVE path)
4      Link next;      // RECURSIVE path
5   }
6   class Molecule{   // TERMINAL class (was NORMAL class)
7      Link parent;              // BACKWARD path (was NORMAL path)
8      Vector3D position; // TERMINAL path
9      Vector3D velocity; // TERMINAL path
10     double mass;
11  }
12  class Vector3D{   // TERMINAL class
13     double x, y, z;
14  }
```

Table I shows a real example of object and path types from a typical molecule dynamics application. The use of terminal classes like Vector3D is very common in Java as composite value type is not supported by the language itself. Notice that at the very beginning there is no backward path. All backward paths are identified at runtime. Since the definition of terminal class depends on backward paths, once we detect a backward path, we need to search over the JVM's internal class graph to mark all class and path type changes. In this example, detecting Molecule.parent to be backward will change Molecule to be a terminal class (since outgoing paths no longer exist) and Link.mol to be a terminal path.

IV. PREFETCHING STRATEGIES

In this section, we present our new object prefetching scheme that raises the overall prefetching effectiveness by giving more weight to those important paths.

Table II shows the pseudo-code of our object prefetching algorithm. A queue is used to maintain candidate objects for path analysis. Prefetching begins with putting the root object into the queue (START_PREFETCH(root)). A BSF loop keeps consuming items from the queue (DEQUEUE()) and calls the PREFETCH_OBJ routine on the dequeued object. The routine will find for every reference field of the object the path data structure (GET_PATH) maintaining metadata like the field's class, path type, and profiling counts for backward path discovery. By checking like lines 18, 21, the algorithm can tell if the current field would contribute a good path to continue the prefetching (ENQUEUE(cand) if yes). The whole process is aided by the following designs of policy.

TABLE II. PATH-ANALYTIC PREFETCHING ALGORITHM

```
1   START_PREFETCH(root)
2      ENQUEUE(root)   // put the root object into queue to start the search
3      while (not QUEUE_EMPTY()) do   // BFS loop
4         obj = DEQUEUE()
5         PREFETCH_OBJ(obj)
6      end while
7   PREFETCH_OBJ(obj)
8      PACK_OBJ_WITH_DFS(obj)    // fetch it first with its terminal children
9      obj_fields = TERMINAL_FIELDS(obj)  +
10                  RECURSIVE_FIELDS(obj) +
11                  NORMAL_FIELDS(obj)    // priority of paths
12     if obj_fields is empty then
13        obj_fields = BACKWARD_FIELDS(obj)
14     end if
15     for each field in obj_fields do
16        path = GET_PATH(obj, field)    // path data structure
17        cand = GETFIELD(obj, field)    // candidate object
18        if PREFETCHED(path, cand) then
19           continue
20        end if
21        if IS_TERMINAL(path) or not TERM_COND() then
23           ENQUEUE(cand)    // put candidate object to search queue
23        end if
24     end for
25  PACK_OBJ_WITH_DFS(obj)
26     PACK_OBJ(buf, obj)    // fetch itself first into message buffer
27     term_fields = TERMINAL_FIELDS(obj)
28     for each field in term_fields do
29        path = GET_PATH(obj, field)
30        cand = GETFIELD(obj, field)
31        if PREFETCHED(path, cand) then
32           continue
33        end if
34        PACK_OBJ_WITH_DFS(cand)    // recursive call for DFS
35     end for
36  PREFETCHED(path, obj)
37     if DUP_LOOKUP(obj) then    // check for any duplicate
38        path.fail_count++
39     endif
40     if BECOME_BACKWARD(path) then    // backward path discovery
41        CHANGE_TO_BACKWARD(path)
42        return false
43     else
44        path.success_count++
45        return true
46     end if
47  TERM_COND()    // don't push any more if buffer exceeds predefined limit
48     return (buf.length > pre_limit) ? true : false
```

1) Priorities of paths: Priority is given to terminal paths, recursive paths and then normal paths. Backward paths will only be used as a last resort. The traversal order on fields in PREFETCH_OBJ() is adjusted according to their path types.

2) Forcing terminal: If we find a terminal path during prefetching, we will always prefetch the target object if it is not too big even if the termination condition (TERM_COND() at line 21) has been reached. This is because if we do not prefetch it at this moment, we would have no chances and they will become orphans to be separately fetched later. Since the cost of creating one extra message is much more

than just including some more bytes in the current message, this speculation is usually profitable.

3) Discovering backward paths: For an HLRC protocol, each object only needs to be fetched at most once within the same interval. So we can maintain a simple hashed lookup-set on which objects have been prefetched for which thread during its current interval, so that objects will never be fetched twice. This doesn't take much space overhead since hash entries for past intervals can simply be discarded, and a thread only has to append its current interval number to an object request. Based on this mechanism, we can discover backward paths dynamically by computing their *duplicate rates* online (obtained by comparing counts of successful prefetching and attempted but duplicate prefetching). We detect backward paths in `PREFETCHED()`, which checks if the object has been prefetched before (from the same thread's same HLRC interval), and updates our per-path statistics of successful and failed (duplicate) prefetch attempts. When the duplicate rate is exceedingly high, the path is changed to backward (i.e. calling `CHANGE_TO_BACKWARD(path)`).

4) Traversal order of terminal objects: Even with limit on message length, a straightout BFS traversal could still make the outgoing buffer greatly overflow because of our terminal-forcing policy. For example, Fig. 2 shows a tree-like object graph composed of terminal and non-terminal objects. If we use a BFS-only policy, we first prefetch non-terminal objects *A*, *B* and *C*, then terminal objects *D*, *E*, *F* and *G* which are found near the end of traversal. When we have prefetched *D*, the size limit is already hit, but we have to force *E*, *F*, and *G* into the buffer to avoid orphans; such overflow is too much to be acceptable. To make our limit meaningful, we alter the original BFS policy to be a global BFS with a local DFS (depth-first search). In this policy, if we discover a subgraph with terminal paths, we use a DFS to prefetch all these objects immediately (by the recursive call `PACK_OBJ_WITH_DFS()`). As prefetching goes on, if we hit the message size limit, the worst-case overflow would be the extra bytes for finishing our current terminal subgraph. In realistic fine-grain applications, such terminal subgraph is usually quite small (within several hundreds bytes). We will evaluate in the next section whether it is true that message size overflow is reasonably close to the predefined limit. Fig. 2 also shows how our revised policy works. We first prefetch object *A*, then *B*. When we are prefetching *B*, we find its terminal paths towards *D* and *E*, so we change to use a local DFS to prefetch them immediately. Now the size limit has been hit, so we stop and leave *C*, *F*, *G* to the next message. The rationale behind this "global BFS; local DFS" policy is that a subgraph with terminal paths often indicates a logical aggregation of objects that should be fetched together. This policy achieves a balanced situation depicted by Fig. 3 (b) between serious fragmentation (Fig. 3 (a)) and poor accuracy with excessive bytes prefetched (Fig. 3 (c)).

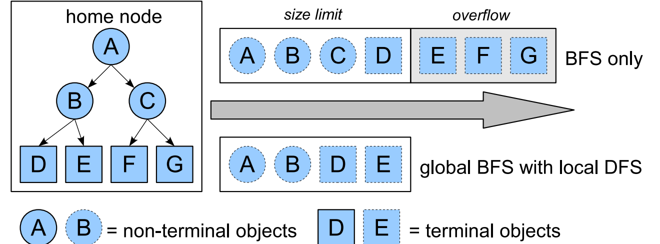


Figure 2. Overflow of prefetch message buffer

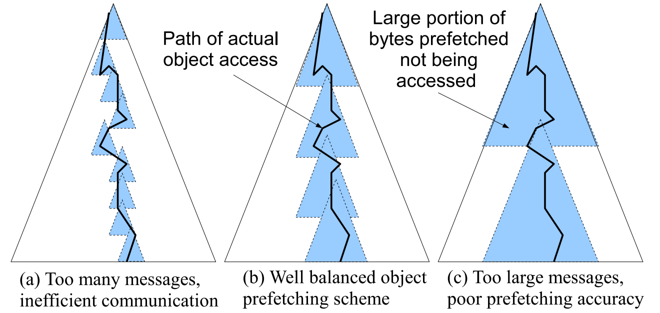


Figure 3. Effect of different message length limits

V. Performance Evaluation

A. Experimental Platform and Benchmarks

Our experiments are conducted in the HKU Gideon 300 Cluster [9] which consists of 300 PCs interconnected by a 312-port 100BASE-T Fast Ethernet switch. Each PC is of the following hardware configuration: an Intel Pentium 4 2GHz processor, 512MB DDR RAM and a 40GB-IDE hard disk. A segment of 8 nodes is used for all the experiments.

We evaluate our prefetching policies with two benchmark programs ported from the SPLASH-2 suite [10] to Java, namely Barnes-Hut and Water-Spatial, which belong to fine- and medium-grained applications respectively. Barnes-Hut is an N-Body simulation using hierarchical methods in which basic elements (i.e. bodies) are organized into a hierarchy, in our case an octree, to speed up computations. Such a hierarchy often involves a fine-grained recursive data structure that is difficult to support efficiently in early DSM systems without a careful prefetching scheme. Barnes-Hut also shows an irregular fine-grained object sharing pattern (each terminal body is of size less than 100 bytes) with some locality and moderate compute-intensiveness. Water-Spatial is a molecule dynamics application, simulating interactions between groups of water molecules. Each molecule is of at most a few hundreds bytes. Molecules within the same 3D box are put into a linked list to allow fast adding or removal of molecules. To sum up, runtime properties of Water-Spatial include near-neighbor 3D-box sharing patterns with medium granularity and intensive computations. For large problem sizes (sufficient room for JIT optimization), the raw speed of our ported Java version when running in server VM mode (tested with Sun JDK 1.6) is highly comparable to the C version (compiled with -O3 option). Barnes-Hut in Java runs even faster than the C code by 8-10% while Water-Spatial in Java is about 10% slower than its C counterpart.

TABLE III. EFFECT OF PREFETCHING POLCIES ON BARNES-HUT

Prefetching Policy	Execution Time		Message Count		Message Volume	
	Raw (sec.)	*Change (%)*	*Raw*	*Change (%)*	*Raw (KB)*	*Change (%)*
None	51.19	N/A	384,155	N/A	37,833	N/A
Simple	41.90	-18.15	187,010	-51.32	60,193	+59.10
Backward	26.62	-48.00	48,455	-87.39	57,175	+51.12
Backward + Terminal	24.47	-52.20	23,508	-93.88	59,563	+57.44

TABLE IV. EFFECT OF PREFETCHING POLCIES ON WATER-SPATIAL

Prefetching Policy	Execution Time		Message Count		Message Volume	
	Raw (sec.)	*Change (%)*	*Raw*	*Change (%)*	*Raw (KB)*	*Change (%)*
None	11.59	N/A	144,059	N/A	31,087	N/A
Simple	9.29	-19.89	72,337	-49.79	32,058	+3.12
Backward	7.96	-31.35	18,578	-87.10	31,224	+0.44
Backward + Terminal	7.91	-31.81	18,579	-87.10	31,231	+0.46

B. Testing Methodology

We evaluate our implementation in the following aspects:

1) Effect of policies: To testify the effectiveness of each of our policies, they are enabled incrementally and compared to no prefetching and the simple policy (used in our earlier work [6]). In each configuration, we measure the execution time, message count and volume in bytes for each program.

2) Accuracy of different message size limits: As we've discussed, choosing an optimal message size limit is vital to strike a balance between accuracy and aggreation effect. To check this out, we do the benchmarking with different trials of message size limits and measure each configuration's accuracy and real performance (total execution time). To measure prefetching accuracy, we compare message volume for different configurations with and without prefetching. If we do not enable prefetching and fetch a single object every time, it is guaranteed that each fetched object will be accessed. Therefore, we can use the measured message volume without prefetching as a reference of how many bytes a perfectly accurate prefetching scheme should prefetch, thus deducing prefetching accuracy by comparing with our prefetching schemes. This measurement is also more meaningful than calculating accuracy by object count, because object size can vary a lot in real-life applications.

3) Message length distribution: From the viewpoint of load balancing, we need to ensure short messages are as few as possible and most messages are uniform in size. To test for this, we log each protocol message's size for different policies, and draw a distribution graph. We also check whether the "global BFS; local DFS" policy is effective to avoid too much overflow of our buffer size limit.

C. Experimental Results

Table III-IV show the effect of our object prefetching policies for each application. Our scheme does prefetch more bytes but the reduction on total execution time and message count is phenomenal. Skipping backward paths is useful to both applications. Forcing terminal objects by a local DFS is very effective for finer granularity. We can see Barnes-Hut is more sensitive to the policies than Water-Spatial since it is not only more fine-grained but also having a recursive data structure (octree) with a much larger fan-out (8 vs. 1).

Table V-VI show the effect of different size limits on object prefetching (Note: raw execution times in Table V-VI do not reconcile with those in Table III-IV which were taken with some other profiling options, out of the scope of this work, enabled). Configurations of small limits (256B-1KB) are inferior to larger limits (4KB-16KB) in terms of message count and execution time. Lifting up the limit doesn't affect accuracy and generally improves speedup due to less fragmentation. Barnes-Hut shows an abnormal increase of message count with 64KB size limit, slightly extending the execution time. After close inspection, we attribute this to the application's inherent fragmentation so that prefetching a too large object graph leaves many smaller object graphs behind that need be fetched individually. Thus, we consider 4K a fairly appropriate size limit parameter although other sizes close to it may behave equally well. Again, Barnes-Hut reacts more readily towards the effect than Water-Spatial.

Fig. 4 displays the size distribution of some real message samples collected using 2KB size limit. Water-Spatial undergoes regular behavior with high accuracy under all policies. By application nature, the objects are larger and there is no complex data structure. Therefore, incrementally enabling more advanced policies won't help much in further regulat-

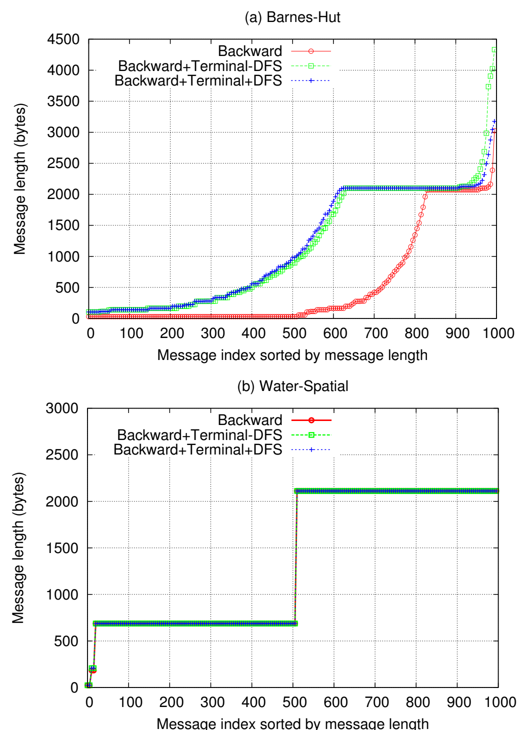


Figure 4. Message length distribution under different policies

TABLE V. EFFECT OF MESSAGE SIZE LIMIT ON BARNES-HUT

Size Limit	Execution Time		Message Count		Message Volume (KB)	Accuracy (%)
	Raw (sec.)	*Change (%)*	*Raw*	*Change (%)*		
No Pre.	40.00	N/A	384,155	N/A	37,833	N/A
256B	24.72	-38.20	91,000	-76.31	59,567	63.51
1KB	20.83	-47.91	33,690	-91.23	59,692	63.38
4KB	19.47	-51.31	16,923	-95.59	59,745	63.32
16KB	19.76	-50.60	10,666	-97.22	59,617	63.46
64KB	21.41	-46.48	15,487	-95.97	59,282	63.82

TABLE VI. EFFECT OF MESSAGE SIZE LIMIT ON WATER-SPATIAL

Size Limit	Execution Time		Message Count		Message Volume (KB)	Accuracy (%)
	Raw (sec.)	*Change (%)*	*Raw*	*Change (%)*		
No Pre.	12.30	N/A	144,061	N/A	31,087	N/A
256B	8.79	-28.53%	36,499	-74.66%	32,098	96.85%
1KB	8.41	-31.62%	23,060	-83.99%	31,988	97.18%
4KB	7.61	-38.11%	14,098	-90.21%	31,227	99.55%
16KB	7.44	-39.46%	5,138	-96.43%	31,218	99.58%
64KB	7.41	-39.72%	5,116	-96.45%	31,204	99.63%

ing communications and shortening execution since the simpler policy stack is already good enough, though enabling all policies won't make things worse too. The case of Barnes-Hut is more complex, but we can observe two things. First, the two policies with terminal forcing induce much fewer small messages, i.e. less fragmentation. Second, if we do not enforce the local DFS policy (i.e. Backward + Terminal - DFS), there could be significantly more messages that exceed much our size limit. To sum up, our best policy (Backward + Terminal + local DFS) works pretty well to regulate message sizes, eliminating most small messages while ensuring message overflow is bounded.

VI. RELATED WORK

Remote page prefetching has been studied in a number of page-based DSMs. In [2, 3], history prefetching and aggregate prefetching, which exploit temporal and spatial locality respectively, are studied and compared. Their work also did evaluation on Barnes and Water but showed no practical speedup. Page-based prefetching is found not effective for fine-grained applications. Our work belongs to dynamic or runtime prefetching at per-object granularity. Previously, we have studied connectivity-based prefetching at a basic level [6]. The work of this paper discusses a more advanced prefetching strategy that analyzes access paths with their behavior learnt. Compared to [6], this work shows an extra 34% improvement in execution time in the best case.

Jackal [5] employs *object-graph aggregation* techniques to reduce access checks and network roundtrips needed to fault in relevant objects. Their compiler detects regions that contain susceptible cache misses, and replaces access checks on the individual objects in the object graph with a single access check on the graph's root object. Object fault will bring in the entire graph. This improves both sequential and parallel performance. While Jackal uses the compiler to recognize a local aggregation of objects, our system detects it by more general and less conservative runtime methods of access tracking and path analysis.

VII. CONCLUSION AND FUTURE WORK

This paper has described a new method to prefetch objects more effectively in a distributed object sharing system. By means of access path tracking and analysis, we classify fields and classes into a handful of types providing clues on how likely they will be needed by the requester. Our prefetching scheme can learn about the weights of paths and balance between accuracy and aggregation effect. It would be interesting to see if any other generic access patterns are worth being analyzed during the prefetch process.

ACKNOWLEDGMENT

This research is supported by Hong Kong RGC grant HKU7176/06E and China 863 grant 2006AA01A111.

REFERENCES

[1] C. Amza, A. Cox, S. Dwarkadas, L. Jin, K.Rajamani and W. Zwaenepoel, "Adaptive Protocols for Software Distributed Shared Memory", In *Proc. of IEEE Special Issue on Distributed Shared Memory*, pp.467-475, Mar 1999.

[2] Hu Weiwu , Zhang Fuxin , Liu Haiming, "Dynamic data prefetching in home-based software DSMs", *Journal of Computer Science and Technology*, v.16 n.3, p.231-241, May 2001.

[3] H. Liu, W. Hu, "A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM," pp.10062a, *15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, 2001.

[4] H.-H. Wang , K.-C. Li , K.-J. Wang , S.-H. Lu, "On the Design and Implementation of an Effective Prefetch Strategy for DSM Systems", *The Journal of Supercomputing*, v.37 n.1, p.91-112, Jul 2006.

[5] R. Veldema , R. F. H. Hofman , R. A. F. Bhoedjang , C. J. H. Jacobs , H. E. Bal, "Source-level global optimizations for fine-grain distributed shared memory systems", *Proceedings of the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming*, p.83-92, June 2001, Snowbird, Utah, United States.

[6] W. Fang, C. L. Wang, and F. C. M. Lau. "Efficient global object space support for distributed JVM on cluster." In *Proc. Int. Conf. on Parallel Processing*, British Columbia, Canada, 2002, pp. 371-378.

[7] W. Zhu, C. L. Wang, and F. C. M. Lau. "JESSICA2: A distributed Java virtual machine with transparent thread migration support". In *Proc. IEEE 4th Int. Conf. Cluster Comput.*, Chicago, USA, 2002, pp. 381-388.

[8] W. Zhu. "Distributed Java Virtual Machine with Thread Migration". PhD thesis, The University of Hong Kong, Aug. 2004.

[9] The HKU Gideon 300 Cluster. http://www.srg.cs.hku.hk/gideon/.

[10] The modified SPLASH-2 benchmark suite. CAPSL, University of Deiaware. http://www.capsl.udel.edu/splash/index.html.

[11] Zhou, Y., Iftode, L., and Li, K. "Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems". SIGOPS Oper. Syst., Rev. 30, SI (Oct. 1996), 75-88.

[12] J. M. Bull , L. A. Smith , L. Pottage, and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications", *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, p.97-105, June 2001, Palo Alto, California, United States.

[13] J.P.Lewis and U. Neumann, "Performance of Java versus C++". http://www.idiom.com/~zilla/Computer/javaCbenchmark.html.