Efficient Global Object Space Support for Distributed JVM on Cluster*

Weijian Fang, Cho-Li Wang and Francis C.M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
{wjfang+clwang+fcmlau}@csis.hku.hk

Abstract

We present the design of a global object space in a distributed Java Virtual Machine that supports parallel execution of a multi-threaded Java program on a cluster of computers. The global object space virtualizes a single Java object heap across machine boundaries to facilitate transparent object accesses. Based on the object connectivity information that is available at runtime, the object reachable from threads at different nodes, named as distributed-shared object, are detected. With the detection of distributed-shared objects, we can alleviate overheads in maintaining the memory consistency within the global object space. Several runtime optimization methods have been incorporated in the global object space design, including an object home migration method that reallocates the home of a distributed-shared object, synchronized method migration that allows the remote execution of a synchronized method at the home node of its synchronized object, and object pushing that uses the object connectivity information to improve access locality.

1. Introduction

A *distributed Java Virtual Machine* (JVM) supports parallel execution of a multi-threaded Java application on a distributed-memory platform like cluster without any modification on the Java program. Java threads created within the program can be transparently distributed among the cluster nodes to achieve a higher degree of execution parallelism and leverage cluster resources to solve large-scale problems.

Due to the popularity of Java [3], distributed JVM has recently become an attractive research problem and several experimental prototypes have emerged. Java/DSM [18], cJVM [2], Hyperion [14], Jackal [17] and JESSICA [13], are some of the well-known examples. The distributed JVM presents a *single system image* (SSI) [8] to Java applications through the creation of a *global object space* (GOS) that "virtualizes" a single Java object heap across multiple cluster nodes to facilitate transparent object access in a distributed environment. For example, the JESSICA system [13] which uses a page-based DSM systems, JUMP [4], to build the GOS. This approach greatly alleviates the burden of the construction of GOS because all the memory consistency issues, such as object faulting, addressing, replication policy, and transmission mechanism, are all managed by the DSM's cache coherence protocol. Such a design, however, suffered from a mismatch between object-based memory model of Java and the underlying page-based DSM implementation. For example, the false sharing problem occurs because of inconsistent sharing granularity between the variable-sized Java objects and the fix-sized memory pages. As thus, the performance of JESSICA was not satisfactory [5]. More efficient solutions to support object sharing among distributed Java threads is demanded.

In this paper, a new *global object space* support for distributed JVM is proposed. We define two types of Java objects: *node-local object* that is reachable from the threads that are at the same cluster node, and *distributed-shared object* (DSO) that is reachable from at least two threads that are located at different cluster nodes.

We argue that the separation of distributed-shared objects and node-local objects can alleviate overheads in maintaining the memory consistency within the global object space and achieve better performance of distributed JVM. Firstly, only distributed-shared objects suffer from heavy overheads in maintaining the memory consistency since they may have multiple duplicated copies on different nodes. Detection of DSOs could make consistency protocol be more lightweight. Secondly, in Java program, synchronization primitives are not only used to protect critical section but also to maintain memory consistency. Synchronization operations on a node-local object do not need to trigger the distributed operations to maintain consistency, because node-local objects are only reachable from some local threads. Detection of DSO makes consistency maintaining less frequently. Thirdly, it is not necessary to apply the

*This research is supported by Hong Kong RGC grant HKU-7030/01E.

distributed garbage collection operations on node-local objects since it is safe to garbage collect node-local objects locally.

We proposed a lightweight solution for detecting the distributed-shared objects. Distributed-shared objects can be detected using an object connectivity graph derived from object reference information that is available at runtime. Our GOS design further leverages the identification of distributed-shared objects and the availability of connectivity information for realizing the Java memory model in a distributed JVM. Such connectivity information was not exploited in most of the previous object-based or page-based DSM systems.

Several areas of optimizations have been proposed in our GOS design: (1) the *object home migration* that reduces communication traffic by migrating the home of a distributed-shared object to a node that need to access the object more frequently; (2) *synchronized method migration* that optimizes critical section execution by shipping a synchronized method to the home node of its synchronized object; (3) *object pushing* that uses connectivity information to prefetch objects for achieving better access locality.

We have tested our design in our JESSICA distributed JVM. The preliminary results show that our approach is promising. With all the optimizations enabled, all four benchmark programs achieved an efficiency of over 84% on four nodes, and all achieved an efficiency of over 75% on eight nodes except one program.

The next section discusses the detection of distributed-shared objects in detail. Section 3 describes our home-based multiple-writer cache coherence protocol that implements the Java memory model. Section 4 discusses various optimizations implemented in GOS. Performance results are reported in section 5. In section 6, several related works are discussed and compared with our GOS. Conclusions are given in Section 7.

2. Distributed-shared Object

In the JVM, each variable, including not only object field that resides in the heap but also thread-local variable that resides in the Java thread stacks, has a type, either a reference type or a primitive type, such as integer, char, or float. This type information is known at compile time and written into class files generated by the compiler. At runtime, the class loader builds up type information from class files. Thus, by looking up runtime type information, we can identify those variables that are of reference type.

2.1 Connectivity Graph and Reachability

If an object's field contains a reference to another object, connectivity exists between these two objects. Instance objects created during runtime will strictly conform to the type information of the class. Therefore, a *connectivity graph* can be built to describe the referential relationship among all objects. The graph is dynamic since connectivity between objects may change from time to time through the reassignment of objects fields.

Reachability describes the relationship between thread and its reachable objects based on the connectivity graph. A thread can reach a subset of objects in the connectivity graph, which include the root objects whose references reside at the thread stack, and all other objects reachable from the root objects via some paths in connectivity graph. Based on the reachability, we can distinguish between *thread-local* objects that can only be reachable from one single thread, and *thread-escaping* objects that can be reachable from multiple threads.

In the context of distributed JVM, Java threads and objects are distributed among different nodes. With the consideration of the relative location between the thread and its reachable objects in a cluster environment, we extend the concepts of thread-local and thread-escaping object and define *node-local* object and *distributed-shared* object: (1) *Node-local object* is an object that is reachable from thread(s) located at the same cluster node. It is either a *thread-local* object or a *thread-escaping* object. (2) *Distributed-shared object* (DSO) is an object that is reachable from at least two threads located at different cluster nodes.

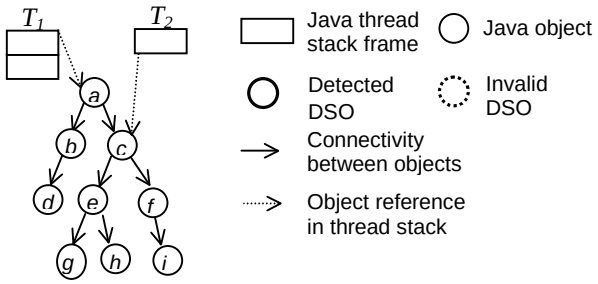
2.2 Detection of DSO

A mechanism to identify distributed-shared objects is essential in the GOS because the accesses on the DSO will initiate a series of thread synchronization and object consistency operations, which involve multiple cluster nodes' collaboration. To minimize the detection overheads, a lightweight DSO detection mechanism is proposed. The detection of DSO in GOS is postponed to the time when a thread is to be migrated or a remote object request is initiated, because not all reachable objects are necessarily accessed during the whole lifetime of the execution.

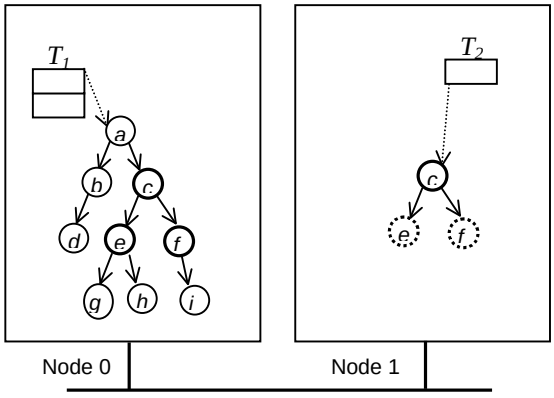
During the thread migration, we examine the thread context to be transmitted across node boundary. We also examine the object content sent to a remote node. The objective is to identify object references stored in them. The transmitted object reference implies the object is a DSO since it is reachable from the threads located at different nodes. If an object reference is identified, and the object has not been marked as a DSO, it is marked at this moment. On the first appearance of a received remote reference, an empty object of its exact class will be created and associated with it. The object's access state will be set to invalid. When it is accessed later, its up-to-date content will be faulted in. In this scheme, only those

objects whose references appear on multiple nodes will be detected as DSOs.

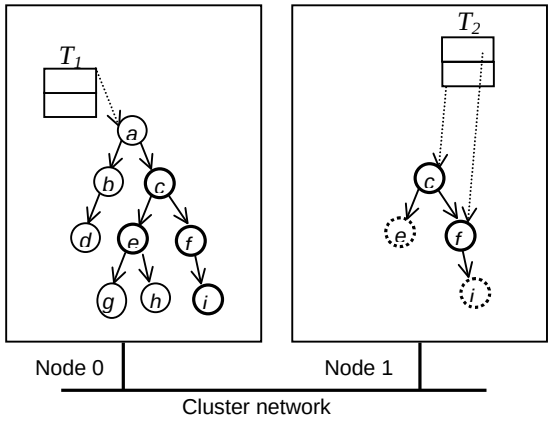
2.3 An Example



(a) Reachability graph



(b) After thread T_2 is distributed to Node 1



(c) Access on *f* by T_2 triggers detection of *i*

Figure 1. Detection of distributed-shared object

Examining the case in Figure 1, a thread T_1 prepares an object tree then passes the reference of object *c* to another thread T_2 as shown in the reachability graph (Figure 1.a). When T_2 is distributed to another cluster node, i.e. node 1, all the objects reachable from object *c* become DSOs. Object *a*, *b*, and *d* are not DSOs since they are thread-local to T_1. Instead of detecting all these objects as DSOs at one blow, we detect object *c* as a DSO and send object *c* to node 1. Because object *e* and *f* are directly connected with object *a*, we also detect object *e* and *f* as DSOs but do not send them to node 1 (Figure 1.b). On node 1, we create two objects whose type are exactly the same as the types of object *e* and *f*. Since the contents of object *e* and *f* are not available, we set their access state to *invalid*. Next time when object *f* is accessed by T_2 on node 1 (Figure 1.c), an object fault will occur. An object request message will be sent to node 0. This event will trigger the detection of object *i* as a DSO. The up-to-date content of object *f* is copied from node 0 to node 1. Details of how to maintain the coherence of objects located among multiple nodes are discussed in next section. If object *e* is not accessed by T_2, object *e* is always invalid on Node 1 and object *g* and *h* will never be detected as DSOs.

3. Cache Coherence Protocol

Java memory model (JMM) mainly defines the memory consistency [1] semantics of multi-threaded Java applications. Any implementation of GOS support for distributed JVM must conform to JMM. We follow the JMM proposed in [15], which is very similar to lazy release consistency [10].

In Java, there is a lock associated with each Java object. Java language provides synchronized block facility, either a synchronized method or a synchronized statement, for achieving exclusive access in a critical section. Enter and exit of a synchronized block correspond to acquiring and releasing the lock associated with the synchronized object. The JMM requires that when a thread acquires a lock, all object values modified by threads previously release the same lock, should be visible to the thread acquiring the lock.

Our GOS implements the JMM with a home-based multiple-writer cache coherence protocol. The object is the unit of coherence. Each DSO is associated with a home node, which is the node that creates the object. Since DSOs can be detected at runtime, accesses on invalid copies of DSOs will fault in their contents on demand. Upon releasing a lock, all updated values to non-home copies of DSOs should be forwarded to corresponding home nodes. Upon acquiring a lock, a flush action is required to set the access states of the non-home copies of DSOs invalid, which guarantees the up-to-date contents will be faulted in from the home nodes when they are accessed later. Before the flush, all updated values to non-home copies of DSOs should be forwarded to the corresponding home nodes. Therefore, in such a way, a thread is guaranteed to see the up-to-date content of DSOs after it acquires the proper lock. Since a lock can be regarded as a special field of an object, all the operations on a lock are also executed at the

corresponding home node. Thus the home node of the object being locked acts as the lock manager.

The concurrent writes to DSO are permitted by using twin and diff techniques [11]. On the first write to a non-home copy of DSO, a twin of object will be created, which is the exact copy of the object. On lock acquiring and releasing, the diff is created by comparing twin with current object content word by word and sent to the home node.

In addition, we can impose some special coherence protocols on some types of objects. For example, since String objects are read-only, the cached copy of a distributed shared String object can be simply treated as a node-local object. Some objects are considered as node-dependent resources, such as file etc. When these node-dependent objects are detected as DSOs, object replication should be prohibited. Instead, the accesses to them should be transparently redirected to their home nodes. This is an important issue to guarantee complete single system image to Java applications.

4. Optimizations

In this section, we study three optimization techniques coupled with the distributed-shared objects. The first two techniques, *object home migration* and *synchronized method migration*, are the refinements to our memory coherence protocol that implements JMM. The third one, *object pushing*, makes use of object connectivity information to improve access locality and achieve the effect of communication aggregation.

4.1 Object Home Migration

In our home-based protocol, a Java thread can access a DSO with less overhead if the thread is located at the home node of the DSO. Thus, it will be more efficient if we can set the home of a DSO according to thread's runtime object access pattern. In GOS, a mechanism is applied to determine the home of a DSO at runtime. Subsequent object home migration is allowed to adapt to thread's object access pattern.

We take a conservative solution that only those objects written from a single remote node will be applied the home migration. In other words, we only apply this optimization to objects exhibiting single writer access pattern. This scheme was adopted because migrating object home may have negative impacts on performance. For example, to notify a thread that doesn't know the object home has already been migrated, an additional redirection message should be sent.

Under our coherence protocol, non-home object writes are reflected to home node on synchronization points. On home node, object request can be considered as a remote read and the diff received on synchronization point as remote write. Object accesses on the home node are also recorded.

To minimize the overhead in detecting single writer pattern at runtime, we record only the consecutive writes on an object, which are from the same remote node. Table 1 shows the events and the corresponding actions on the object's current home node when object home migration is enabled. In the table, *C* denotes the count of consecutive writes from a specific remote node *N*. The counter *C* will be reset to 1 if a different remote node issues an object write.

The number of consecutive writes roughly records the number of synchronization iterations during which the object is only updated by that node. We follow a heuristic that an object presents single writer pattern if the count of consecutive writes exceeds a predefined threshold. If single writer pattern is detected, the object home is migrated to the writing node.

Table 1. Events and actions in object home migration

Event	Action
Local read	No action
Local write	*C* = 0
Remote read from a different node from *N*	No action
Remote write from a different node from *N*	*C* = 1; *N* = the writing node
Remote read from *N*	If *C* > threshold, migrate home to *N*
Remote write from *N*	*C*++

4.2 Synchronized Method Migration

```
1  class Counter {
2      private int i; // internal counter
3
4      public Counter() {
5          i = 0;
6      }
7
8      public synchronized void inc() {
9          i++;
10     }
11 }
```

Figure 2. Synchronized method migration example

Java's synchronization primitives (e.g., synchronized block, wait and notify methods of Object class) are originally designed for thread synchronization in a shared memory environment. The synchronization constructs built from them may be inefficient in the distributed JVM that is implemented in a distributed memory architecture like cluster.

Considering the Counter class source code in figure 2, we suppose the instance object is a DSO and its home is not the node that invokes inc(). Upon entering and exiting

the synchronized inc() method, the invoking node will acquire and release the lock of the instance object. In line 9, the object will be faulted in. In this case, we observe 3 message roundtrips.

It is a common behavior that synchronized object's fields will be accessed in the synchronized method. Thus, all the synchronization requests or object requests will be sent to the home node of the DSO. This will lead to multiple short messages floating between the nodes involving in this synchronization operation.

Migrating synchronized method of DSO to its home node for execution will effectively reduce the number of messages and reduce consistency maintaining overhead incurred in synchronization operations.

4.3 Object Pushing

In Java program execution, after an object is accessed, its reachable objects in connectivity graph are very likely to be accessed afterward. Since object connectivity information is available at runtime, it is possible to prefetch multiple related objects in connectivity graph to improve this kind of access locality.

We use object pushing to improve the prefetching accuracy. While requesting a DSO, the home node will push the requested object together with multiple objects reachable from it to the requesting node. This mechanism provides accurate prefetching since the home node has the up-to-date copies of the objects and the connectivity information maintained in the home node is always valid.

This solution is better than the pull-based one, which relies on the requesting node to fault in the requested objects. In this scenario, the faulting node issues explicit instructions to specify which objects to be pulled. A fatal drawback of this solution is that the connectivity information contained in the invalid object may be obsolete. Therefore, the prefetching accuracy is not guaranteed. Some unneeded objects, even garbage objects, may be prefetched. This will result in the waste of communication bandwidth.

In our implementation, we set an optimal message length, which is the preferred aggregation size of objects to be carried to the requesting node. Reachable objects rooted from the requested object will be selected to copy to the message buffer until the current message length is larger than the optimal message length. Some selection mechanism, either depth-first or bread-first algorithm, can be applied.

To reduce negative impact of pushing unneeded objects, we will not push large objects. For example, the arrays of reference type, e.g., multi-dimension arrays, are usually shared among multiple threads. Object pushing is not performed on the request of an array of reference type.

Overall, the object pushing improves the access locality since objects to be accessed in the future have been moved to the executing thread's local memory. Object pushing can also improve performance by achieving aggregation effect on communication because it can effectively reduce the number of object requests during the execution cycles.

5. Performance Evaluation

In this section, we study the performance of GOS. The GOS is embedded in our JESSICA distributed JVM for supporting object sharing in a cluster environment. All the tests are performed on a cluster of 300MHz Pentium II PCs, running Linux 2.2.14, and connected by a fast Ethernet. The JESSICA is executed under the interpreter mode. In our tests, when the Java applications are started, Java threads are automatically distributed among cluster nodes to achieve maximal parallelism.

5.1 Application Suite

Our application suite consists of four multi-threaded Java programs: All-pair Shortest Path (ASP), Successive Over-Relaxation (SOR), Traveling Salesman Problem (TSP), and Nbody.

ASP calculates the shortest path between any pair of nodes in a graph using a parallel version of Floyd's algorithm. It requires n iterations to solve an n-nodes problem. At iteration k, all threads need the value of the kth row of the distance matrix. There is a barrier at the end of each iteration. The workload is distributed equally among worker threads in row wise.

SOR does red-black successive over-relaxation on a 2-D matrix for a number of iterations. There are two barriers in each iteration. The workload is distributed equally among worker threads in row wise.

Nbody simulates the motion of particles due to gravitational forces over a number of simulation steps. The program follows the algorithm of Barnes & Hut. Each worker thread is computing the motion simulation of a part of particles. A quadtree is constructed at the beginning of each step, which will be accessed by all worker thread.

TSP finds the shortest route among a number of cities using parallel branch-and-bound algorithm, which prunes large parts of the search space by ignoring partial routes already longer than current best solution. We divide the whole search trees to many small ones to form a job queue. Every worker thread will get jobs from this queue until the queue is empty.

5.2 Application Performance

Figure 3 shows the efficiency for each application after all optimizations are enabled. Sequential performance

data is measured on the original Kaffe JVM when calculating efficiency.

All 4 benchmark programs have achieved efficiency larger than 84% on 4 nodes and all have achieved efficiency larger than 75% on 8 nodes except Nbody. ASP even achieves an efficiency of 98% on 4 nodes. In ASP, while the cluster size is scaled to 8 nodes, the global synchronization among all threads becomes a primary factor to pull down the efficiency. SOR's situation is similar. In Nbody, there is a construction of quadtree in each simulation step, which cannot be parallelized. When the main thread performs construction of quadtree, all other threads are waiting. The efficiency decreases while the cluster size increases. TSP is a computation intensive program comparing with other benchmark programs. Load imbalance among worker threads is a major factor affecting efficiency.

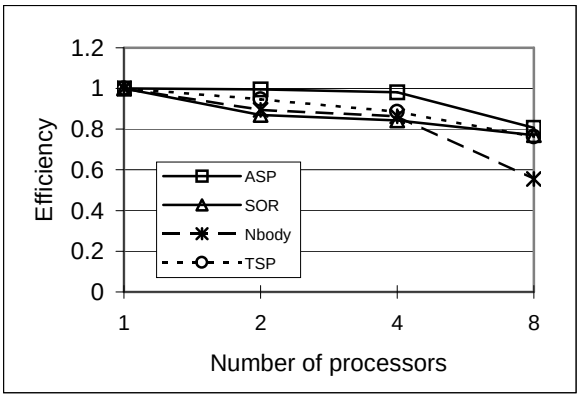


Figure 3. Efficiency

Table 2 shows their communication effort for some given parameters on a 4-node cluster after all optimizations are enabled. Msg column shows the number of messages and the Data column shows the network data volume involved. All the four programs need to access the object heap intensively and involve considerable communication effort except TSP.

Table 2. Communication effort

	Parameters	Msg (K)	Data (MB)
ASP	A graph of 512 vertices	21.5	24.98
SOR	1024 by 1024 matrix for 30 iterations	22.9	42.01
Nbody	400 particles for 10 simulation steps	10.6	4.74
TSP	12 cities	2.9	0.24

Figure 4 shows the normalized execution time break down against number of processors for the four benchmark programs. Obj denotes the time to request an up-to-date copy of a faulting object. Syn denotes the time spent on synchronization operations, such as lock, unlock, and wait. Comp denotes the computation time.

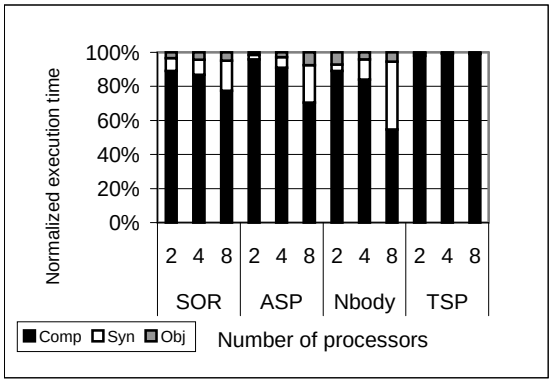


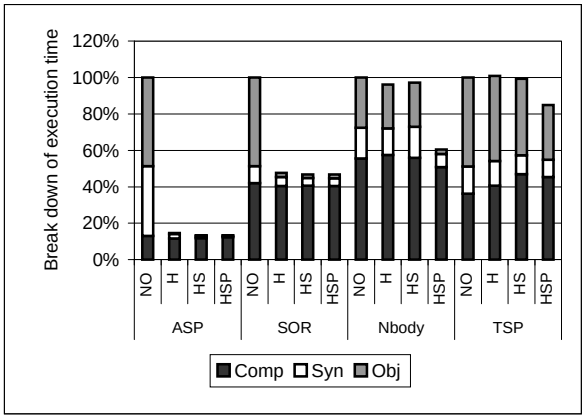
Figure 4. Percentage of execution time break down against no. of processors

Since we insert software checks before object accesses to test object access states, an additional test was conducted to evaluate the overhead of the access checks in our GOS. Comparing the sequential performance on JESSICA with that on Kaffe, the cost of checks can be derived. Since our implementation is based on interpreter model, check cost doesn't contribute significant overhead. In all four benchmarks, check cost is less than 3.5% against execution time on Kaffe.

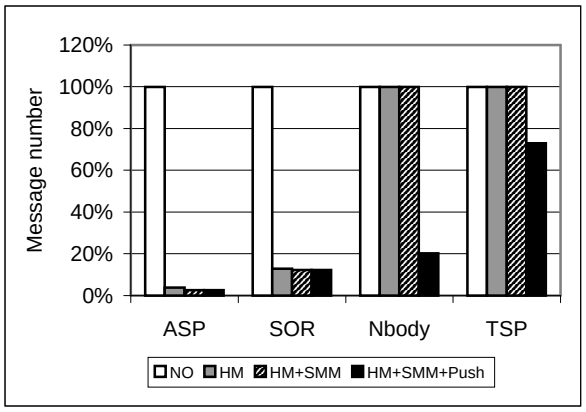
5.3 Effect of Optimizations

In this subsection, the effect of individual optimizations is studied. Figure 5 shows the effects of optimizations on execution time, message number, and communication data volume when running the benchmark suite on a 4-node cluster. In the below figures, NO means no optimization, HM means object home migration, SMM means synchronized method migration, Push means object pushing. In this test, TSP solves a problem of 8 cities.

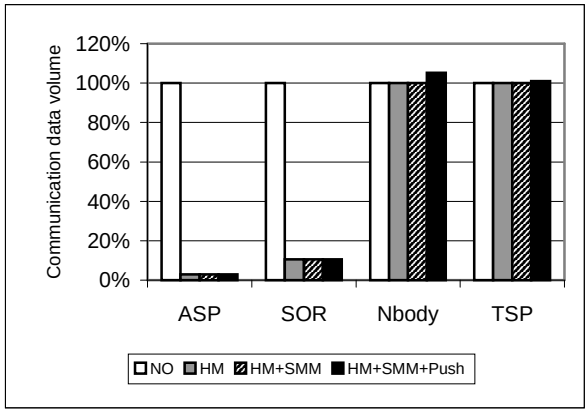
As seen from the figures, object home migration greatly improves the performance of ASP and SOR. This is because some DSOs are only written by one thread in some duration of execution in SOR and ASP. The use of synchronized method migration decreases the number of messages by 29.96% for ASP and 4.58% for SOR. Synchronized method migration also results in less synchronization operations. As a result, the execution time decreases by 8.82% for ASP and 1.84% for SOR. Object pushing aggregates small object messages into a larger message. Nbody is a typical application involved with lots of small-sized DSOs. The number of messages is remarkably reduced by 79.83% with object pushing enabled. Since object pushing may push unneeded objects as well, communication data volume increases by 5.23%. Nevertheless, Nbody's execution time decreases by



(a) Breakdown of execution time
H: HM, HS: HM+SMM, HSP: HM+SMM+Push



(b) Message number



(c) Communication data volume
Figure 5. The effects of optimizations

37.81% as a final result. Object pushing also reduces TSP's message number by 27% and incurs a little more communication data. As a result, TSP's execution time decreases by 14%. Compared with Nbody and TSP, most DSOs used in ASP and SOR are large 2-dimension arrays. Object pushing has little effect on them. Synchronized method migration increases Nbody's execution time by within 2%. Object pushing increases ASP's execution time by 1%. Overall, the negative impact incurred by these optimizations in our benchmark programs is very limited.

6. Related Work

As a distributed JVM, cJVM [2] uses a proxy object model to implement global object space. Method invocation and fields accessing on the *proxy object* are shipped to its master object in general. Several optimizing techniques were applied to reduce such shipping. This approach is more suitable for the sequential consistency memory model. However, under the proposed Java memory model, i.e., the lazy release consistency, this approach may not be very effective since a more aggressive object caching mechanism, like our global object space, seems more appropriate. In addition, the load distribution in cJVM is determined by object distribution in method shipping approach. Load balance might be difficult to achieve without programmer's effort.

JESSICA [13] leveraged a page-based DSM to build GOS. All objects are allocated into distributed shared memory. Such an approach suffers from false sharing problem that is inherited from the page-based DSM. Since page-based DSM isn't aware of Java runtime connectivity information, it is difficult to detect distributed-shared objects and do further optimizations. The detail analysis of various factors contributing to the efficiency in using page-based DSM for supporting distributed object sharing can be found in [5]. Java/DSM [18] is another similar example that builds global object space on top of page-based DSM.

Some other approaches reply on compiler techniques to transparently run multi-threaded Java applications on a cluster. They directly compile multi-threaded Java program to native code that is able to execute in a distributed platform. In these systems, JVM is not involved in the execution and a software DSM is employed to support global object accesses. Hyperion [14] compiles Java bytecode to C source code, then compiles to native code further. Jackal [17] compiles Java source code to native code. In both cases, most efforts to improve performance are done at compile time. Jackal's compiler enables two optimizations: *object-graph aggregation* and *automatic computation migration*, which are similar to our object pushing and synchronized method migration. Object-graph aggregation uses heap approximation algorithm [6] to identify those related objects. However, heap approximation algorithm cannot distinguish between different runtime objects that are created at the same allocation site. Hence this approach is effective only at the situation when the related objects are from different allocation sites. Comparatively, our object pushing is a runtime approach and has no such drawback.

Both Jackal and Hyperion do not intend to detect distributed-shared objects.

In the DSM field, DOSA [7] implements a fine-grained DSM support for typed language such as Java. Its aim is to keep sharing granularity at object level but still rely on the virtual memory support to do the access state check as in the page-based DSM. It introduces a level of indirection on object accessing. Access to objects will go through a handle table to locate object's actual address. The indirection adds overhead on object accesses and impairs cache locality.

7. Conclusions

This paper presents a global object space support for distributed JVM. Distributed-shared objects are detected with the help of runtime object connectivity information to improve the performance. Only distributed-shared objects are taken care of to maintain consistency in global object space. Several optimizations can be incorporated into the global object space. Among them, home migration and object pushing can effectively improve the performance of applications presenting certain access behaviors. Synchronized method migration can optimize the execution of Java synchronized method in the context of distributed JVM. After all optimizations are enabled, considerable performance is obtainable.

In our future work, we will incorporate the detection of distributed-shared object with our distributed garbage collection algorithm in global object space. To further improve the performance of global object space, an adaptive cache coherence protocol will be implemented, which will automatically adjust to the various access patterns of distributed-shared objects. As object access pattern may change dynamically during the execution lifetime, we believe a runtime solution is more effective to adapt to the access patterns.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: A Tutorial. *IEEE Computer*, 29(12): 66-76, December 1996.

[2] Y. Aridor, M. Factor, and A. Teperman. cjvm: a single system image of a jvm on a cluster. In *Proc. of International Conference on Parallel Processing*, 1999.

[3] Gilad Bracha, James Gosling, Bill Joy, and Guy Steele. The Java Language Specification, Second Edition. Addison Wesley, 2000.

[4] B. Cheung, C.L. Wang, Kai Hwang. A Migrating-Home Protocol for Implementing Scope Consistency model on a Cluster of Workstations. International Conference on *Parallel and Distributed Processing Techniques and Applications* (PDPTA' 99), **p**. 821-827, 1999, Las Vegas.

[5] W.L. Cheung, C.L. Wang, and F.C.M. Lau. Building a Global Object Space for Supporting Single System image on a Cluster. To appear in *Annual Review of Scalable Computing*, Volume 4, World Scientific, 2002.

[6] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 121--133, January 1998.

[7] Y. Charlie Hu, Weimin Yu, Dan Wallach, Alan Cox, and Willy Zwaenepoel. Runtime support for distributed sharing in typed languages. In *Proceedings of the Fifth ACM Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.

[8] K. Hwang, E. Chow, C.L. Wang, H. Jin, and Z. Xu, Desinging SSI Cluster with Hierarchical Checkpointing and Single I/O Space. In *IEEE Concurency*, 1999.

[9] P. Keleher. Distributed Shared Memory Home Pages. http://www.cs.umd.edu/~keleher/dsm.html.

[10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In Proceedings of the 19th *Annual International Symposium on Computer Architecture*, pages 13--21, May 1992.

[11] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Proceedings of *the Winter 94 Usenix Conference*, pp. 115-131, January 1994.

[12] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification, Second Edition. Addison Wesley, 1999.

[13] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. Jessica: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 60, Oct. 2000. (JESSICA source code is available at: http//www.srg.csis.hku.hk/Jessica-src/.)

[14] M. MacBeth, K. McGuigan, and P. Hatcher. Executing java threads in parallel in a distributed-memory environment. In *Proc. of IBM Center for Advanced Studies Conference*, 1998.

[15] Jeremy Manson and William Pugh. Core Semantics of Multithreaded Java. In *Proc. of Joint ACM Java Grande - ISCOPE 2001 Conference*, June 2001.

[16] Transvirtual Technologies Inc. Kaffe JVM. http://www.kaffe.org.

[17] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Proc. Joint ACM JavaGrande-ISCOPE 2001*, Stanford, 2001.

[18] W. Yu and A. Cox. Java/dsm: A platform for heterogeneous computing. In *Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation*, 1997.