On the Design of Global Object Space for Efficient Multi-threading Java Computing on Clusters [1]

Weijian Fang, Cho-Li Wang, Francis C.M. Lau

System Research Group
Department of Computer Science and Information Systems
The University of Hong Kong

Abstract

The popularity of Java and recent advances in compilation and execution technology for Java are making the language one of the preferred ones in the field of high-performance scientific and engineering computing. A *distributed Java Virtual Machine* supports transparent parallel execution of multi-threaded Java programs on a cluster of computers. It provides an alternative platform for high-performance scientific computations. In this paper, we present the design of a *global object space* for a distributed JVM. It virtualizes a single Java object heap across machine boundaries to facilitate transparent object accesses. We leverage runtime object connectivity information to detect *distributed-shared objects* (DSOs) that are reachable from threads at different nodes to facilitate efficient memory management in the distributed JVM. Based on the concept of DSO, we propose a framework to characterize object access patterns, along three orthogonal dimensions. With this framework, we are able to effectively calibrate the runtime memory access patterns and dynamically apply optimized cache coherence protocols to minimize consistency maintenance overhead. The optimization devices include an *object home migration* method that optimizes the single-writer access pattern, *synchronized method migration* that allows the execution of a synchronized method to take place remotely at the home node of its locked object, and *connectivity-based object pushing* that uses object connectivity information to optimize the producer-consumer access pattern. Several benchmark applications in scientific computing have been tested on our distributed JVM. We report the performance results and give an in-depth analysis of the effects of the proposed adaptive solutions.

Key words: Java, Cluster Computing, Distributed Java Virtual Machine, Distributed Shared Memory, Adaptive Cache Coherence Protocol

[1] This research was supported in part by the Hong Kong RGC under Grant HKU-

1 Introduction

The Java programming language [1] supports concurrent programming with multiple threads, which makes it a potential language for parallel computing without the need to learn a new parallel language. Recent advances in Java compilation and execution technology, such as just-in-time compiler and the hotspot technology [2], add to the attractiveness of Java as a language for high performance scientific and engineering computing [3]. Some performance benchmark results even indicate that Java can outperform the C programming language in some numerical computations [4].

On the other hand, cluster [5,6] has gradually been accepted as a scalable and affordable parallel computing platform by both academia and industry in recent years. Several research projects have been conducted to support transparent and parallel execution of multi-threaded Java programs on clusters [7–11]. Among them, Java/DSM [9], cJVM [10], and JESSICA [11] introduced the idea of a *distributed JVM* that runs on a cluster of computers. A distributed JVM appears as a middleware that presents a *single system image* (SSI) [12] of the cluster to Java applications. With a distributed JVM, the Java threads created within one program can be run on different cluster nodes to achieve a higher degree of execution parallelism. In addition, cluster-wide resources such as memory, I/O, and network bandwidth can be unified and used as a whole to solve large-sized problems.

The adoption of the distributed JVM for parallel Java computing can also boost programming productivity. Given that the distributed JVM conforms to the JVM specification, any Java program can run on the distributed JVM without any modification. The steep learning curve can thus be avoided since the programmers do not need to learn a new parallel language, a new message passing library, or a new tool in order to develop parallel programs. It is also convenient for program development as the parallel algorithms can be implemented and tested in a single machine before it is submitted to a parallel computer for execution. Finally, many existing multi-threaded Java applications, especially server applications, can be ported to clusters when a cost-effective parallel platform is sought for.

In a distributed JVM, the shared memory nature of Java threads call for a *global object space* (GOS) that "virtualizes" a single Java object heap spanning multiple nodes or the entire cluster to facilitate transparent object access [11]. The GOS is indeed a *distributed shared memory* (DSM) service in an object-oriented system. The memory consistency semantics of the GOS are defined based on the Java memory model (Chapter 8 of the JVM specification [13]). The performance of the distributed JVM hinges on the GOS's ability to mini-

7030/01E and by HKU under Large Equipment Grant 01021001.

mize the communication and coordination overheads in maintaining the single object heap illusion.

Many distributed JVMs use a page-based DSM to build the GOS [9,11]. This is an easy approach because all the memory consistency and cache coherence issues are handled by the page-based DSM. It however suffers from problems due to a mismatch between the object-based memory model of Java and the underlying page-based implementation of the distributed object heap subsystem. One of these is the false sharing problem which occurs because of the incompatible sharing granularities of the variable-sized Java objects and the fixed-size virtual memory pages [14]. This mismatch has also prevented further optimizations in the cache coherence protocol implementing the Java memory model.

Object-based DSMs can be good candidates for implementing the GOS. Most existing object-based DSM systems are language-based [15–18], and rely on the compiler to extract object sharing information in the user's program. Such information comes usually from annotations by the programmers. Therefore, the approach cannot fit into our distributed JVM scenario because as a runtime component, the GOS cannot rely on the programmer to provide the sharing information.

Scientific applications exhibit diverse execution patterns. To execute these applications efficiently in software DSM systems, many cache coherence protocols have been proposed. Home-based protocols [19] assign a home node to each shared data object from which all copies are derived. It is widely believed that home-based protocols are more scalable than homeless protocols [20], for the reason that the former has less memory consumption and can eliminate `diff` accumulation. The home in a home-based protocol can be either fixed [19] or mobile [21]. There is also variation for the coherence operations, such as a multiple-writer protocol, or a single-writer protocol. The multiple-writer protocol introduced in Munin [17] supports concurrent writes on different copies using the diff technique. It may however incur heavy diff overhead compared with conventional single-writer protocols. Another choice is between the update protocol (e.g., Orca [15]) and the invalidate protocol used in many page-based DSM systems such as TreadMarks [20] and JUMP [21]. The update protocol can do prefetching to make the data available before the access, but it may send much unneeded data when compared with the invalidate protocol. Indeed, the choice of a good coherence protocol is often application-dependent. That is, the particular memory access patterns in an application speak for the more suitable protocol. That motivates us to go after an adaptive protocol.

In this paper, we propose a new global object space design for the distributed JVM. In our design, we use an object-based adaptive cache coherence protocol to implement the Java memory model. We believe that adaptive protocols

are superior to non-adaptive ones due to their adaptability to object access patterns in applications. An adaptive cache coherence protocol is able to detect the current access pattern and adjusts itself accordingly. Some DSMs (e.g., Munin [17]) support multiple cache coherence protocols and allow the programmer to explicitly associate a specific protocol with the shared data. This is not transparent to the programmer and it is difficult to dynamically switch between different protocols in response to changes in the access pattern. Several page-based DSM systems [22][23] support adaptive protocols that can automatically adapt to the access pattern at runtime. However, the access pattern observed by these systems is the page-level approximation of the actual pattern. They may not be effective if the approximation deviates substantially from the actual pattern, which can easily be the case if the application has fine-grained sharing granularity. An object-based adaptive protocol, on the other hand, should be more flexible.

The challenges of designing an effective and efficient adaptive cache coherence protocol are: (1) whether we can determine those important access patterns that occur frequently or those that contribute a significant amount of overhead to the GOS, and (2) whether the runtime system can efficiently and correctly identify such target access patterns and apply the corresponding adaptations in a timely fashion.

To further understand the first challenge and to overcome it, we propose the *access pattern space* as a framework to characterize object access behavior. This space has three dimensions—number of writers, synchronization, and repetition. We identify some basic access patterns along each dimension: multiple-writers, single-writer, and read-only for the number-of-writers dimension; mutual exclusion and condition for the synchronization dimension; and patterns with different numbers of consecutive repetitions for repetition dimension. Some combination of different basic patterns along the three dimensions then portrays an actual runtime memory access pattern. This 3-D access pattern space serves as a foundation on which we can identify those significant object access patterns in the distributed JVM. We can then choose the right adaptations to match with these access patterns and improve the overall performance of the GOS.

To meet the second challenge, we take advantage of the fact that the GOS is implemented by modifying the heap subsystem of the JVM. Our adaptive protocol can leverage all runtime object types and access information to efficiently and accurately identify the access patterns worthy of special focus. We leverage runtime object connectivity information to detect *distributed-shared objects* (DSOs). DSOs are the objects that are reachable from at least two threads located at different cluster nodes in the distributed JVM. The identification of DSOs allows us to handle the memory consistency problem more precisely and efficiently. For example, in Java, synchronization primitives are

not only used to protect critical sections but also to maintain memory consistency. Clearly, only synchronization of DSOs may involve multiple threads on different nodes. Thus, the identification of DSOs can reduce the frequency of consistency-related memory operations. Moreover, since only DSOs that are replicated on multiple nodes would be involved in consistency maintenance, the detection of DSOs therefore leads to a more efficient implementation of the consistency protocol.

We apply three different protocol adaptations to the basic home-based multiple writer cache coherence protocol in three respective situations in the access pattern space: (1) *object home migration* which optimizes the single-writer access pattern by moving the object's home to the writing node according to the access history; (2) *synchronized method migration* which chooses between default object (data) movement and optional method (control flow) movement in order to optimize the execution of critical section methods according to some prior knowledge; (3) *connectivity-based object pushing* which scales the transfer unit to optimize the *producer-consumer* access pattern according to object connectivity information.

The rest of the paper is organized as follows. Section 2 introduces the access pattern space. Section 3 defines DSO, and explains the lightweight DSO detection scheme and how we use the concept of DSO to address both the memory consistency issue and the memory management issue in the GOS. Section 4 presents the adaptive cache coherence protocol. We conducted experiments to measure the performance of the prototype based on our design, which we report in Section 5. In section 6, related work is discussed and compared with our GOS. The final section gives the conclusion and presents a possible agenda for future work.

2 Access Pattern Specification

In this section, we first introduce the Java memory model which influences memory behavior, and then propose the access pattern space for specifying object access behavior in Java. Although we discuss access patterns in the context of Java, the access pattern space concept should be applicable to other shared memory systems.

2.1 Java Memory Model

The Java memory model (JMM) defines memory consistency semantics of multi-threaded Java programs. There is a lock associated with each object

in Java. Based on the JMM proposed in [24], when a thread T_1 acquires a lock that was most recently released by another thread T_2, all writes that are visible to T_2 at the time of releasing the lock become visible to T_1. This is the release consistency [25].

The Java language provides the *synchronized* keyword, used in either a synchronized method or a synchronized statement, for synchronization among multiple threads. Entering or exiting a synchronized block corresponds to acquiring or releasing a lock of the specified object. A synchronized method or a synchronized statement is used not only to guarantee exclusive access in the critical section, but also to maintain memory consistency of objects among all threads that have performed synchronization operations on the same lock.

We follow the operations defined in the JVM specification to implement this memory model. Before a thread releases a lock, it must copy all assigned values in its private *working memory* back to the *main memory* which is shared by all threads. Before a thread acquires a lock, it must flush (invalidate) all variables in its working memory; and later uses will load the values from the main memory. Therefore, an object's access behavior can be described as a set of reads and writes performed on the object, with interleaving synchronization actions such as locks and unlocks. Locks and unlocks on the same object are executed sequentially. Among all the accesses from different threads, a partial order is established by the synchronization actions.

The complexity of the implementation of the JMM stems from the fact that these reads and writes as well as locks and unlocks may be issued concurrently by multiple threads. In particular, the locks and unlocks invoked on a particular object may influence other objects' access behavior of different threads. A straight-forward implementation of the JMM will result in poor performance, especially in a cluster environment.

2.2 Access Pattern Space

Three orthogonal dimensions capturing the characteristics of object access behavior can be defined: *number of writers*, *synchronization*, and *repetition*. They form a 3-dimensional access pattern space, as shown in Fig. 1.

Number of writers. This says how many nodes there are in which some thread is writing to the object. We distinguish three cases:

- *Multiple writers*: the object is written by multiple nodes.
- *Single writer*: the object is written by a single node. *Exclusive access* is a special case where where the object is accessed (written and read) by only

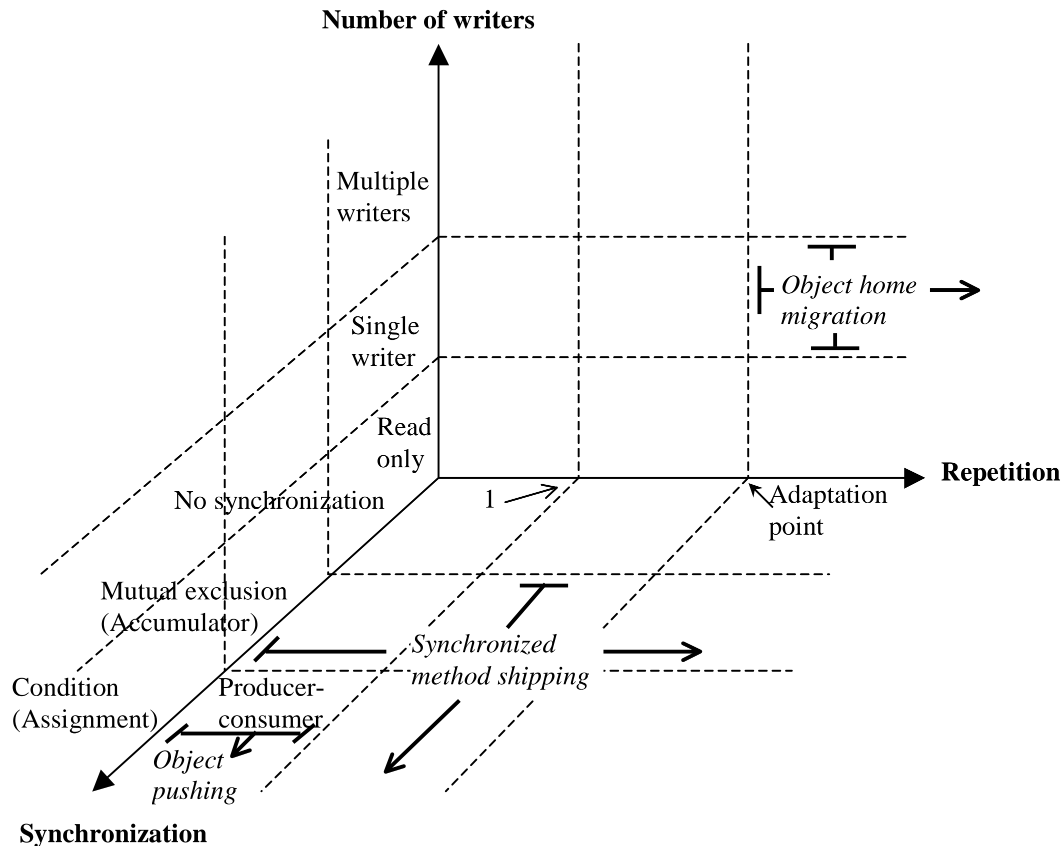

Fig. 1. The object access pattern space: items in normal font are the access patterns; items in italic font are the corresponding adaptations; the basic protocol is not shown.

one node. Object home migration is used to optimize this pattern, which will be discussed in Section 4.1.

- *Read only*: no node writes to the object.

Synchronization. This characterizes the execution order of accesses by different threads. When the object is accessed by multiple threads and at least one thread is a writer, the threads must be well synchronized to avoid data race. There are three cases:

- *Accumulator*: the object accesses are mutually exclusive. The object is updated by multiple threads concurrently, and therefore all the updating should happen in a critical section. That is, the read/write should be preceded by a lock and followed by an unlock.
- *Assignment*: the object accesses obey the precedence constraint. The object is used to safely transfer a value from one thread to another thread. The source thread writes to the object first, followed by the destination thread reading it. Synchronization actions should be used to enforce that the write happens before the read according to the memory model. Java provides the `wait` and `notify` methods in the `Object` class to help implement the

assignment pattern.
- No synchronization: synchronization is unnecessary.

We use synchronized method shipping to optimize synchronization-related patterns, which will be discussed in Section 4.2.

Repetition. This indicates the number of consecutive repetitions of an access pattern. It is desirable that an access pattern will repeat for a number of times so that the GOS will be able to detect the pattern using history information and then to apply optimization on the re-occurrence of the pattern. Such a pattern will appear on the right side of the *adaptation point* along the repetition axis. The adaptation point is an internal threshold parameter in the GOS. When the pattern repeats for more times than what the adaptation point indicates, the corresponding adaptation will be automatically performed. The single-writer pattern can be optimized using this approach. On the other hand, some important patterns appear on the left of the adaptation point, such as the producer-consumer pattern, which is also called the single assignment. We use connectivity-based object pushing to optimize those patterns that have little repetition as we cannot rely on the history information to detect them. The detail is presented in Section 4.3.

3 Distributed-shared Object

In this section, we define distributed-shared object and the benefits it brings to our GOS. We then present a lightweight mechanism for the detection of DSOs and the basic cache coherence protocol used in the GOS.

3.1 Definitions

In the JVM, *connectivity* exists between two Java objects if one object contains a reference to another. Therefore, we can conceive the whole picture of an object heap to be a *connectivity graph*, where vertices represent objects and edges represent references. *Reachability* describes the transitive referential relationship between a Java thread and an object based on the connectivity graph. An object is *reachable* from a thread if its reference resides in the thread's stack, or if there is some path existing in the connectivity graph between this object and some known reachable object.

By the escape analysis technique [26], if an object is reachable from only one thread, it is called *thread-local* object. The opposite is a *thread-escaping* object, which is reachable from multiple threads. Thread-local objects can be

separated from thread-escaping objects at compile time using escape analysis.

In a distributed JVM, Java threads are distributed to different nodes, and so we need to extend the concepts of thread-local object and thread-escaping object. We define the following.

- A *node-local object* (DSO) is an object reachable from thread(s) in the same node. It is either a thread-local object or a thread-escaping object.
- A *distributed-shared object* (NLO) is an object reachable from at least two threads located at different nodes.

3.2 Benefits from Detection of DSOs

The detection of DSOs can help reduce the memory consistency maintenance overhead. According to the JVM specification, there are two memory consistency problems in a distributed JVM. The first one, *local consistency*, exists among working memories of threads and the main memory inside one node. The second one, *distributed consistency*, exists among multiple main memories of different nodes. The issue of local consistency should be addressed by any JVM implementation, whereas the issue of distributed consistency is only present in the distributed JVM. The cost to maintain distributed consistency is much more than that of its local counterpart due to the communication incurred. As we have mentioned before, synchronization in Java is used not only to protect critical sections but also to enforce memory consistency. However, synchronization actions on NLOs do not need to trigger distributed consistency maintenance, because all threads that are able to acquire or release the lock of an NLO must reside in the same node, and therefore would not experience distributed inconsistency throughout.

Only DSOs are involved in distributed consistency maintenance since they have multiple copies in different nodes. With the detection of DSOs, only DSOs need to be visited to make sure that they are in a consistent state during distributed consistency maintenance.

According to the JVM specification, one vital responsibility of the GOS is to perform automatic memory management in the distributed environment—*distributed garbage collection* (DGC) [27]. The detection of DSOs also helps improve the memory management in the GOS in this regard. Being aware of the existence of DSOs, local garbage collectors can perform asynchronous collection of garbage. The detection of DSOs enables independent memory management in each node.

In the distributed JVM, whether an object is a DSO or an NLO is determined by the relative location of the object and the threads reaching it. Compile-time solutions, such as escape analysis, are not useful as the location of objects and threads can only be determined at runtime. We propose a runtime lightweight DSO detection scheme which leverages Java's runtime type information.

Java is a strongly typed language. Each variable, either object field that is in the heap or thread-local variable in some Java thread stack, has a type. The type is either a reference type or a primitive type such as integer, char, or float. The type information is known at compile time and written into class files generated by the compiler. At runtime, the class subsystem builds up type information from the class files. Thus, by looking up runtime type information, we can identify those variables that are of the reference type. Therefore, object connectivity can be determined at runtime. The object connectivity graph is dynamic since connectivity between objects may change from time to time through the reassignment of objects fields.

DSO detection is performed when there are some JVM runtime data to be transmitted across node boundary, which could be thread stack context for thread relocation, object content for remote object access, or diff data for update propagation. On both the sending and the receiving side, these data are examined for identification of object references contained within. A transmitted object reference indicates the object is a DSO since it is reachable from threads located at different nodes. On the sending side, if the object has not been marked as a DSO, it is marked at this moment. On the receiving side, when a received remote reference first emerges, an empty object of corresponding type will be created to be associated with it, so that the reference will not become a dangling pointer. The object's access state will be set to be invalid. When it is accessed later, its up-to-date content will be "faulted in". In this scheme, only those objects whose references appear in multiple nodes will be identified as DSOs.

We detect DSO in a lazy fashion. Since at anytime it is unknown whether an object will be accessed by its reaching thread in the future or not, we choose to postpone the detection to as close to the actual access as possible, thus making the detection scheme lightweight.

To correctly reflect the sharing status of objects in the GOS, we rely on distributed garbage collection to convert a DSO back to an NLO. If all the cached copies of a DSO have become garbage, the DSO can be converted back to an NLO. A DGC algorithm, *indirect reference listing* (IRL) [28], is adopted to collect DSOs that have turned into garbage. With the IRL in place, each node

independently garbage-collects its local heap using a mark-sweep collector [29]. Timely invocation of DGC can avoid the unnecessary overheads in handling the consistency problem.

3.4 Basic Cache Coherence Protocol

Our basic cache coherence protocol is a home-based, multiple-writer cache coherence protocol. Fig.2 shows a state transition graph depicting the lifecycle of an object from its creation to possible collection based on the proposed DSO concept.

An object is the unit of coherence. When a DSO is detected, the node where the object is first created is made its home node. The home copy of a DSO is always valid. A non-home copy of a DSO can be in one of three possible access states: invalid, read (read-only), or write (writable). Accesses to invalid copies of DSOs will fault in the contents from their home node. Upon releasing a lock of a DSO, all updated values to non-home copies of DSOs should be written to their corresponding home nodes. Upon acquiring a lock, a flush action is required to set the access state of the non-home copies of DSOs invalid, which guarantees that 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 written to the corresponding home nodes. In this way, a thread is able to see the up-to-date contents of the DSOs after it acquires the proper lock. Note that along the number-of-writers dimension in the access pattern space, the multiple-writers pattern can be treated as a generalized form of all patterns, with the single-writer pattern and the read-only pattern being special cases (with some dumb writers).

Since a lock can be considered a special field of an object, all the operations on a lock, including acquire, release, as well as `wait` and `notify` that are the methods of the `Object` class, are executed in the object's home node. Thus, the object's home node acts as the object's lock manager. A multiple-writer protocol permits concurrent writing to the copies of a DSO, which is implemented using the *twin* and *diff* technique [20]. On the first write to a non-home copy of the DSO, a twin will be created, which is an exact copy of the object. On lock acquiring and releasing, the diff, i.e., the modified portion of the object, is created by comparing the twin with the current object content word by word, and sent to the home node.

With the availability of object type information, it is possible to invoke different coherence protocols according to the type of the objects. For example, immutable objects, such as instances of class `String`, `Integer`, and `Float`, can be simply replicated and treated as an NLO. Some objects are consid-

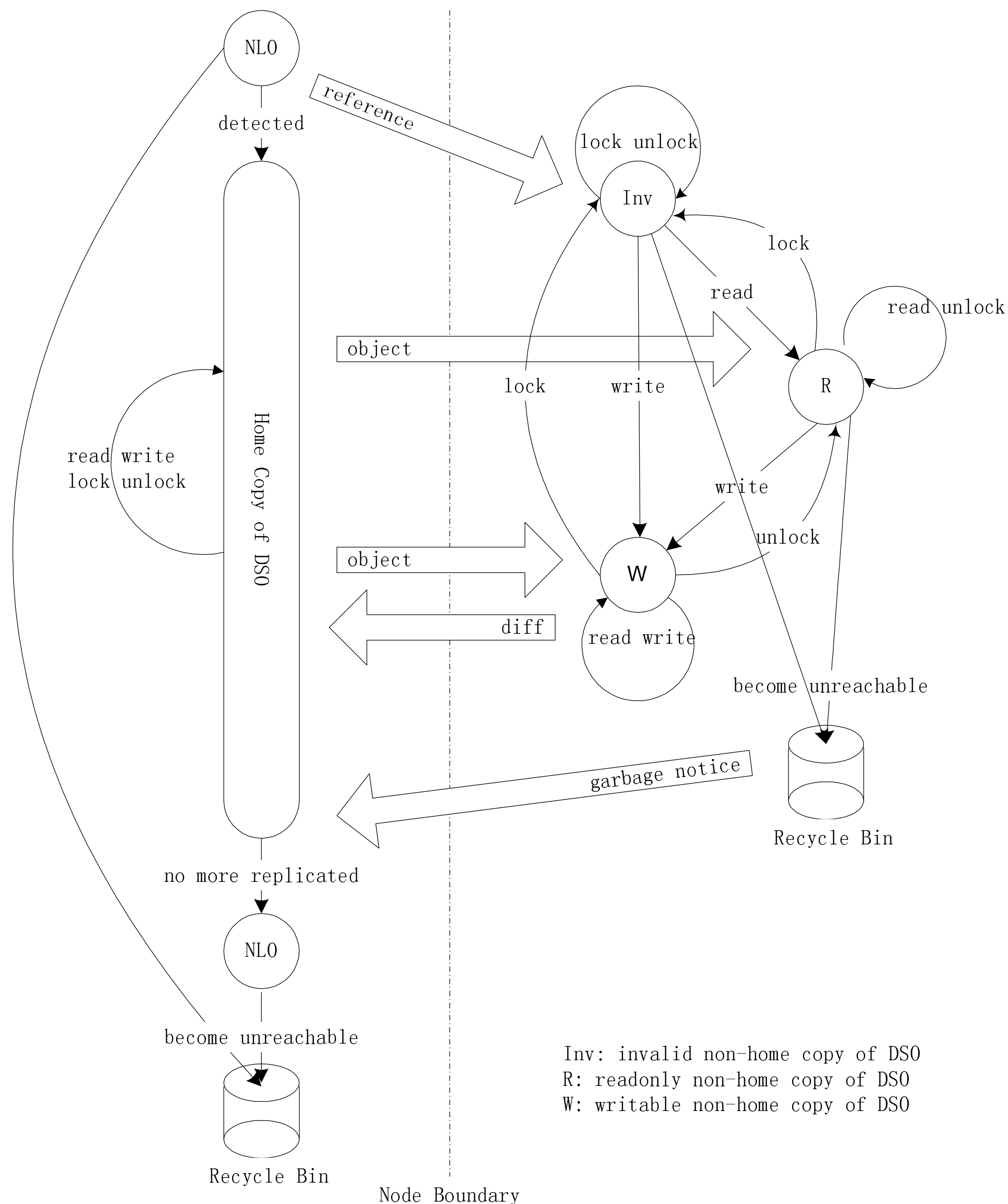

Fig. 2. State transition graph depicting object lifecycle in the GOS

ered node-dependent resources, such as instances of class `File`. When node-dependent objects are detected as DSOs, object replication should be denied. Instead, accesses to them should be transparently redirected to their home nodes. This is an important issue in the provision of a complete single system image to Java applications.

4 Adaptive Cache Coherence Protocol

In the last section, we presented the home-based multiple-writer cache coherence protocol for dealing with the consistency issues. However, as explained before, the non-adaptive protocol can not be optimal in all circumstances. The adaptive protocols are superior to non-adaptive ones because of their adaptability to applications' access patterns. In this section, we discuss the adaptations we add to the basic protocol based on the proposed access pattern space.

4.1 Object Home Migration

With a home-based cache coherence protocol, each DSO has a home node to which all writes are propagated and from which all copies are derived. Therefore, the home node of a DSO plays a special role among all nodes holding a copy. Accesses happening in the non-home nodes will incur communication with the home node, while accesses in the home node can proceed in full speed.

We propose a runtime mechanism to determine the optimal location of the home of an object and perform object home migration accordingly. Object home migration may have negative impacts on performance. In order to notify a node which is not aware of the home migration, a redirection message should be sent. Improper migration will result in a large number of unnecessary redirection messages in the network. Therefore, we only apply object home migration to those DSOs exhibiting the single-writer access pattern. If a DSO exhibits the multiple-writers pattern, all the non-home nodes can still communicate with the original home node in order to obtain the up-to-date copy and propagate the writes. It does not matter which is the home node as long as long as the home node is one of the writing nodes.

If a DSO exhibits the single-writer pattern and its home is made the only writing node, the overhead of creating and applying diff can be eliminated. If the DSO further exhibits an exclusive access pattern, all the accesses will happen in the home node, and therefore no communication will be necessary.

In order to detect the single-writer access pattern, the GOS monitors all home accesses as well as non-home accesses at the home node. With the cache coherence protocol, the object request can be considered a remote read, and a diff received on synchronization points can be considered a remote write. To monitor the home accesses, the access state of the home copy will be set to `invalid` on acquiring a lock and to `read` on releasing a lock. Therefore, home access faults can be trapped and a return can be made after the access is recorded.

To minimize the overhead in detecting the single-writer pattern, the GOS records consecutive writes that are from the same remote node and that not interleaved by the writes from other nodes. We follow a heuristic that an object is in the single-writer pattern if the number of consecutive writes exceeds a predefined threshold. A relatively small threshold is used because the number of consecutive writes reflects the synchronization periods during which the object was only updated by that node.

The cost incurred by object home migration is due to monitoring and recording the consecutive writes. This cost happens when the object request message or the diff message arrives, as well as the first local write on at-home DSO happens. Compared with the communication overhead, this cost is negligible since only a few instructions suffice to update the object's current consecutive write counter.

If the single-writer pattern is detected, upon request from the writing node, not only is this object delivered in a reply but also a home migration notification would be issued. A forwarding pointer is left in the original home node to refer to the new home.

4.2 Synchronized Method Migration

Synchronized method migration is not meant to directly optimize synchronization related access patterns such as assignment and accumulator. Instead, it optimizes the execution of the synchronized method itself, which is usually related to those access patterns.

Java's synchronization primitives, including synchronized block, as well as the `wait` and `notify` methods of the `Object` class, are originally designed for thread synchronization in a shared memory environment. The synchronization constructs built upon them are inefficient in a distributed JVM that is implemented in a distributed memory architecture like clusters.

Fig. 3 shows the skeleton of a Java implementation of the barrier function. The execution cannot continue until all the threads have invoked the `barrier` method. We assume the instance object is a DSO and the node invoking `barrier` is not its home node. On entering and exiting the synchronized `barrier` method, the invoking node will acquire and then release the lock of the `barrier` object, while maintaining distributed consistency. In line 8, the `barrier` object will be faulted in. It is a common behavior that the locked object's fields will be accessed in a synchronized method. In line 9 and line 11, the synchronization requests `wait` and `notifyAll` respectively, will be issued. The `wait` method will also trigger an operation to maintain distributed con-

```
1    class Barrier {
2          int count;             // the number of threads to barrier
3          private int arrived; // initial value equals to 0
4
5          public synchronized void barrier() {
6                try {
8                     if (++arrived < count)
9                          wait();
10                    else {
11                         notifyAll();
12                         arrived = 0;
13                    }
14              } catch (Exception e) { }
15         }
16   }
```

Fig. 3. Barrier class

sistency according to the JMM. [2] Therefore, there are four synchronization or object requests sent to the home node and multiple distributed consistency maintaining operations are involved.

Migrating a synchronized method of a DSO to its home node for execution will combine multiple round-trip messages into one and reduce the overhead for maintaining distributed consistency. While object shipping is the default behavior in the GOS, we apply method shipping particularly to the execution of synchronized methods of DSOs. With the detection of DSOs, this adaptation is feasible in our GOS.

The method shipping will cause the workload to be redistributed among the nodes. However, the synchronized methods are usually short in execution time and can only be sequentially executed by multiple threads; therefore, synchronized method migration will not affect the load distribution in the distributed JVM.

4.3 Connectivity-based Object Pushing

Some important patterns, such as the single-writer pattern, tend to repeat for a considerable number of times, therefore giving the GOS the opportunity to detect the pattern using history information. However, there are some significant access patterns that do not repeat. These latter patterns cannot be detected using access history information.

[2] According to the JMM, `wait` behaves as if the lock is released first and acquired later.

Connectivity-based object pushing is applied in our GOS to the situations where no history information is available. Essentially, object pushing is a prefetching strategy which takes advantage of the object connectivity information to more accurately pre-store the objects to be accessed by a remote thread, therefore minimizing the network delay in subsequent remote object accesses. Connectivity-based object pushing actually improves the reference locality.

The *producer-consumer* pattern is one of the patterns that can be optimized by connectivity-based object pushing. Similar to the assignment pattern, the producer-consumer pattern obeys the precedence constraint. The write must happen before the read. However, in the producer-consumer pattern, after the object is created, it is written and read only once, and then turned into garbage. Therefore, producer-consumer is single-assignment. The producer-consumer pattern is popular in Java programs. Usually, in a producer-consumer pattern, one thread produces an object tree, and prompts another consuming thread to access the tree. In the distributed JVM, the consuming thread suffers from network delay when requesting objects one by one from the node where the object tree resides.

In order to apply connectivity-based object pushing, we follow the heuristic that after an object is accessed by a remote thread, all its reachable objects in the connectivity graph may be "consumed" by that thread afterwards. Therefore, upon request for a specific DSO in the object tree, the home node pushes all the objects that are reachable from it to the requesting node.

Object pushing is better than pull-based prefetching which relies on the requesting node to specify explicitly which objects to be pulled according to the object connectivity information. A fatal drawback of pull-based prefetching is that the connectivity information contained in an invalid object may be obsolete. Therefore, the prefetching accuracy is not guaranteed. Some unneeded objects, even garbage objects, may be prefetched, which will end up wasting communication bandwidth. On the contrary, object pushing gives more accurate prefetching since the home node has the up-to-date copies of the objects and the connectivity information in the home node is always valid.

In our implementation, we rely on an optimal message length, which is the preferred aggregate size of objects to be delivered to the requesting node. Reachable objects from the requested object will be copied to the message buffer until the current message length is larger than the optimal message length. We use a breadth-first search algorithm to select the objects to be pushed. If these pushed objects are not DSOs yet, they will be detected. This way, DSOs are eagerly detected in object pushing.

Since object connectivity information does not guarantee that future accesses

are bound to happen, object pushing also risks sending unneeded objects. To reduce such negative impacts, the GOS will not push large-size objects. It will also not perform object pushing upon request of an array of reference type, e.g., a multi-dimension array, since such an array usually represents some workload shared among threads with each thread accessing only a part of it.

5 Performance Evaluation

In this section, we present the performance of the GOS and the effects of the adaptations discussed in Section 4.

Our distributed JVM implementation is based on the Kaffe JVM [30] which is an open-source JVM. The GOS is integrated with the bytecode execution engine in interpreter mode. A Java application is started in one cluster node. When a Java thread is created, it is automatically dispatched to a free cluster node to achieve parallel execution. Unless specified otherwise, the number of threads created is the same as the number of cluster nodes in all the experiments. This arrangement should give us the best possible performance in most cases. We conducted the performance evaluation on the HKU Gideon 300 cluster [31], which is a cluster of PCs with Intel 2GHz P4 CPU, running Linux kernel 2.4.18, and connected by a Fast Ethernet.

Our application suite consists of four multi-threaded Java programs: (1) ASP, to compute the shortest paths between any pair of nodes in a graph (of 1024 nodes) using a parallel version of Floyd's algorithm; (2) SOR, which performs red-black successive over-relaxation on a 2-D matrix (2048×2048) for a number of iterations; (3) Nbody, to simulate the motion of particles (2048 of them) to gravitational forces between each other over a number of simulation steps using the algorithm of Barnes & Hut; (4) TSP, to solve the Traveling Salesman Problem by finding the cheapest way of visiting all the cities (12 of them) and returning to the starting point with a parallel branch-and-bound algorithm.

5.1 Application Performance

Fig. 4 shows the efficiency curves for each application. The sequential performance is measured using the original Kaffe JVM.

In the figure, we observe high efficiency can be achieved when we use up to 32 nodes for Nbody and TSP. Among the four applications, TSP is the most computationally intensive program. Therefore, it is able to achieve over 80% of efficiency even for a relatively small problem size. TSP prunes large parts

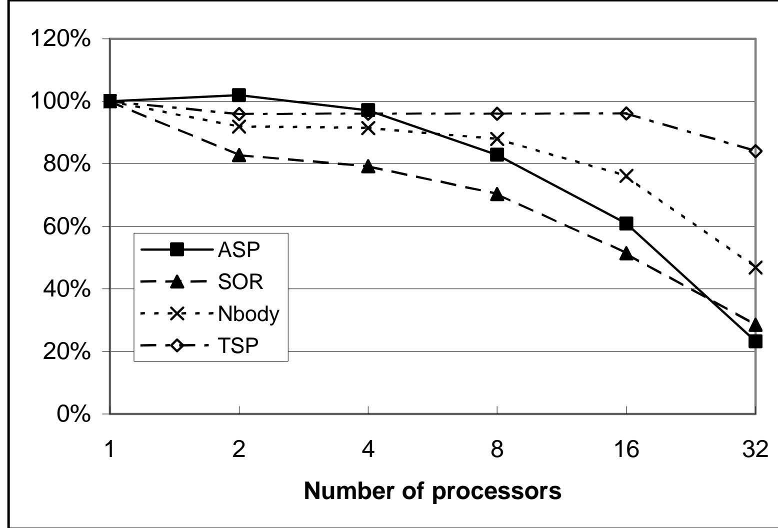

Fig. 4. Efficiency

of the search space by ignoring partial routes that are already longer than the current best solution. The program divides the whole search tree into many small ones to build up a job queue in the beginning. Every thread will obtain jobs from this queue until it becomes empty. In Nbody, the measured efficiency is higher than 40% in the 32-node case. The efficiency is mainly affected by the construction of the quadtree in each simulation step as it cannot be parallelized. When the main thread conducts the construction, all other threads are waiting. The efficiency decreases while the number of processors increases.

When 32 nodes are used, the efficiencies of SOR and ASP drop below 40%. In SOR and ASP, even though the workload is distributed equally among the threads, they both suffer from the intensive synchronization overhead caused by the embedded barrier operations. Further analysis is conducted according to the timing breakdown of various overheads incurred during the execution, as shown in Fig. 5.

Note that all applications are implemented in a structured and synchronized manner. Each thread acquires all the needed data objects before it performs the computation, and all threads are synchronized in some way before it starts the next iteration. Therefore, we are able to break the total execution time into four parts, where `Comp` denotes the average computation time in each node; `Obj` the average object access time in each node to fault in up-to-date copies of invalid objects; `Syn` the time spent on synchronization operations, such as lock, unlock, wait, and notify; and `GC` the average garbage collection overhead measured at each node. The percentages of the four measurements with respect to the total execution time are displayed in Fig. 5.

Notice that not every application requires the GC. The Obj and Syn portions

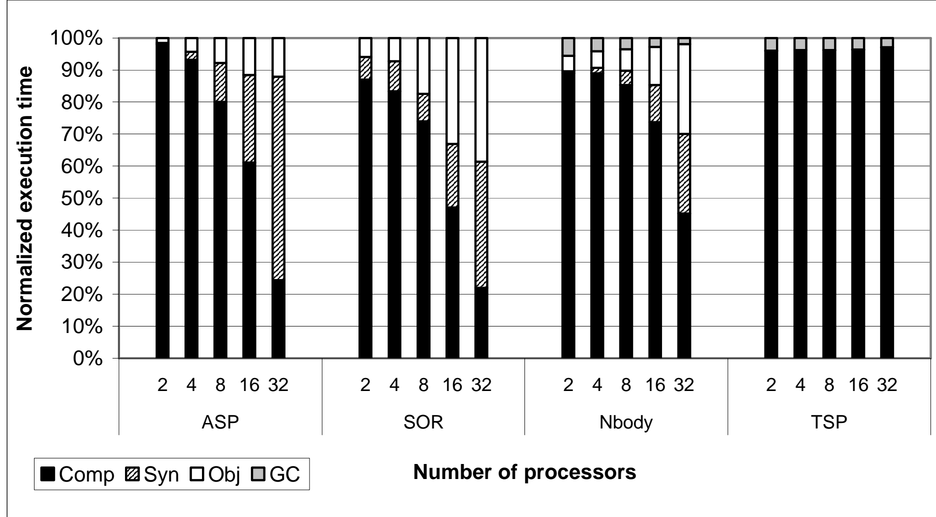

Fig. 5. Breakdown of normalized execution time against number of processors

are the GOS overhead to maintain a global view of a virtual object heap shared by physically distributed threads. The Obj and Syn portions not only include the necessary local management cost and the time spent on the wire for moving the protocol-related data, but also the possible waiting time on the requested node.

ASP requires n iterations to solve an n-node graph problem. There is a barrier at the end of each iteration, which requires participation of all threads. The Java language does not directly provide any barrier operation among threads, and so the barrier is implemented using synchronized primitives, as shown in Fig. 3. We can see in Fig. 5 that the Syn portion increases rapidly as we increase the number of processors. In SOR, there are two barriers in each iteration. The situation of SOR is similar to that of ASP. The Syn operation contributes a significant portion to the execution time when scaled to a large number of processors.

Nbody also involves synchronization in each simulation step. The synchronization overhead becomes a significant part of the overall execution time when we increase the number of processors. We also observe that when we scale up the number of processors, the GC portion shrinks due to the reduced memory requirement on each node. TSP is a computationally intensive program, and the GOS overhead accounts for less than 5% of the total execution time.

In order to measure the DSO detection overhead, we run the applications on two nodes, and configure them so that all the to-be-shared objects are allocated in one node while the working thread is running in the other node. This way we can have the largest percentage of DSOs and the largest communication traffic during the execution. Then we add up the DSO detection overhead on the two nodes, and compare it against the corresponding sequential execution time

of the applications. The percentage of DSO detection overhead against the sequential execution time is less than 3% for all applications (more precisely, 1.07% for ASP, 1.68% for SOR, 0.85% for Nbody, and 2.25% for TSP). We can see that the DSO detection overhead is quite small. For each application, we choose a very small problem size, and therefore the communication-to-computation ratio is relatively large. Since the DSO detection overhead always co-exists with the communication, as the problem size scales up, the proportion of DSO detection overhead in the execution time will decrease accordingly.

5.2 Effects of Adaptations

In the experiments, all adaptations are disabled initially; and then we would enable the planned adaptations incrementally. Fig. 6 shows the effects of adaptations on the communication, in terms of the number of messages exchanged between cluster nodes and the communication traffic caused during the execution. Fig. 7 shows the effects of adaptations on the execution time. We present the normalized execution time against different problem sizes. In ASP, we scale the size of the graph; in SOR, we scale the size of the 2-D matrix; in Nbody, we scale the number of the bodies; in TSP, we scale the number of the cities. All data are normalized to that when none of the adaptations are enabled. All tests run on 16 processors. In the legend, "No" denotes no adaptive protocol enabled, "HM" denotes object home migration, "SMM" denotes synchronized method migration, and "Push" denotes object pushing.

As can be seen in the figures, object home migration greatly improves the performance of ASP and SOR. In ASP and SOR, the data are in the 2-D matrices that are shared by all threads. In Java, a 2-D matrix is implemented as an array object whose elements are also array objects. Many of these array objects exhibit the single-writer access pattern after they are initialized. However, their original homes are not the writing nodes. Object home migration automatically makes the writing node the home node in order to reduce communication. We can see that object home migration dramatically reduces the number of messages, the communication volume, as well as the execution time. Also the effect of object home migration is amplified when the problem size is scaled up in ASP and SOR.

In Nbody, the single-writer access pattern is insignificant, and therefore the effect of object home migration cannot be observed. In TSP, all threads have the chance to update the DSO storing the current minimal tour. However, a certain thread may update it for several times consecutively. In that situation, the multiple-writers object dynamically changes its pattern to single-writer for a short while, and then changes back to the multiple-writers pattern. Subsequent accesses will not claim any performance improvement if they do

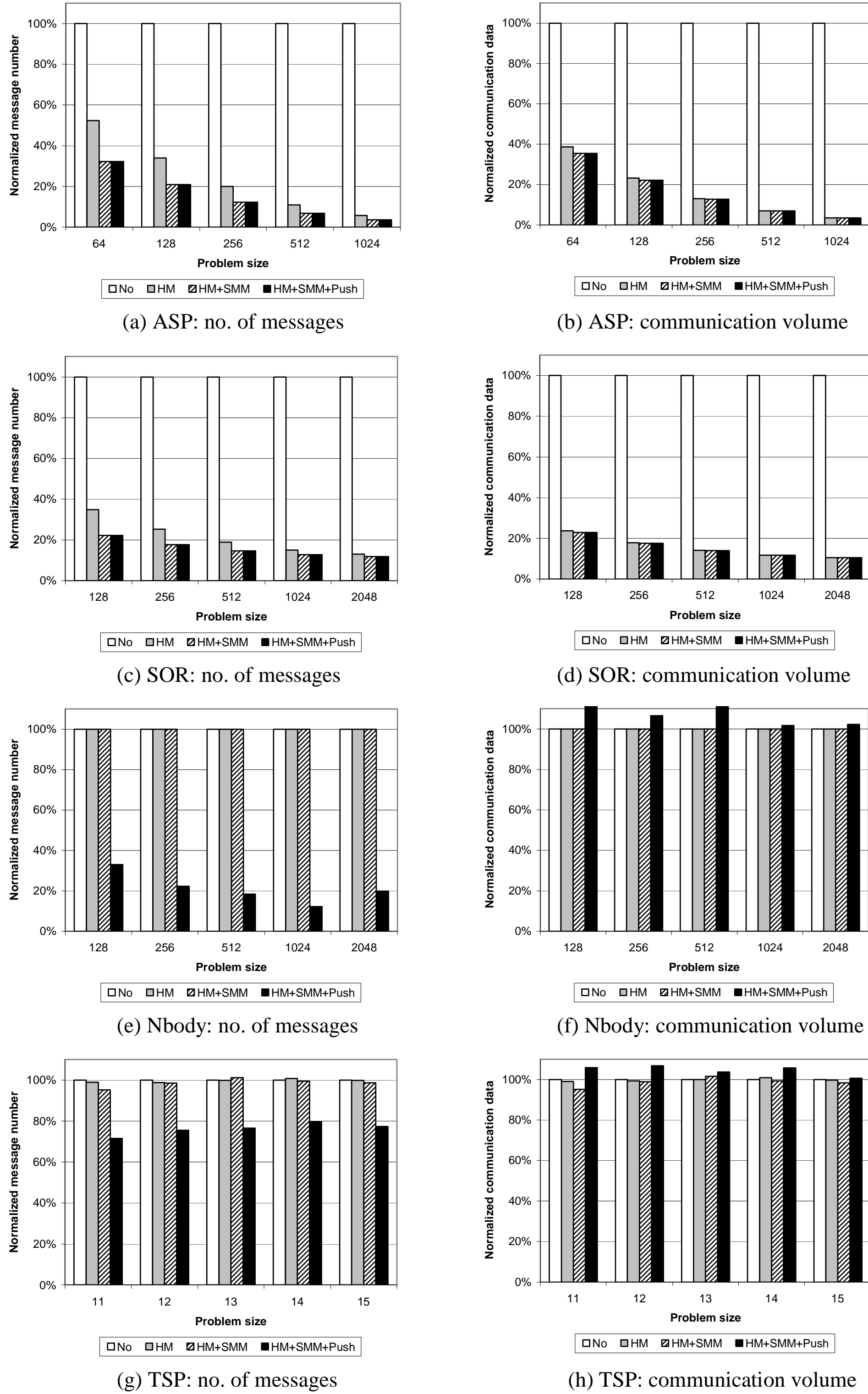

Fig. 6. Effects of adaptations w.r.t. communication

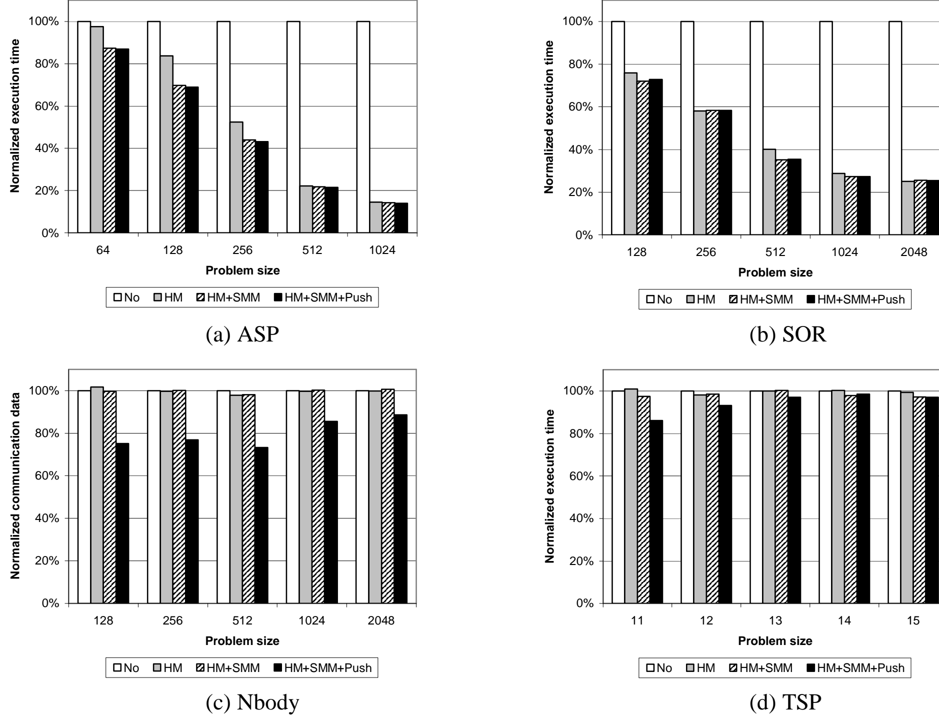

Fig. 7. Effects of adaptations w.r.t. execution time

not exhibit such a pattern after home migration. They may even experience some performance loss because some overhead to locate the new home will be incurred. TSP is one case showing the possible negative impact of home migration, where our heuristic to detect the single-writer pattern fails. But as can be seen in Fig. 7(d), the negative impact is well contained, less than one percent in TSP.

Synchronized method migration optimizes the execution of a synchronized method of a non-home DSO. Although it does not reduce the communication volume, it reduces the number of messages significantly, as can be seen in the ASP and SOR cases. We also observe in Fig. 7(a) and (b) that synchronized method migration improves ASP and SOR's overall performance to some extent, particularly when the problem size is small. ASP requires n barriers for all the threads in order to solve an n-node graph. The synchronization is quite heavy in ASP. So synchronized method migration has more positive effect on ASP. When the problem size is scaled up, the communication-to-computation ratio decreases, thus the adaptation effect becomes not so evident. The synchronization overhead comprises not only the processing and transmission time, but also the waiting time. Sometimes the synchronization overhead is dominated by the waiting time, which cancels out the benefit from synchronized method migration. Nbody's synchronization uses synchronized block instead of synchronized method, and so synchronized method migration has no effect here. TSP's situation is similar to Nbody's.

Connectivity-based object pushing is a prefetching strategy which takes advantage of the object connectivity information to improve reference locality. Particularly, it improves the producer-consumer pattern greatly. Nbody is a typical application of the producer-consumer pattern. In Nbody, a quadtree is constructed by one thread and then accessed by all other threads in each iteration. The quadtree consists of a lot of small-sized objects. We can see that object pushing greatly reduces the number of messages for Nbody. Since object pushing may push unneeded objects as well, the amount of communication increases slightly. The improvement on execution time due to object pushing is also significant in Nbody, as seen from Fig. 7(c). However, when the problem size is scaled up, the communication-to-computation ratio decreases, thus the effect of object pushing decreases. Notice that communication is relatively little in TSP. Although object pushing could decrease the number of messages, the improvement on the total execution time due to this optimization is still limited. Compared with Nbody and TSP, most DSOs in ASP and SOR are array objects, and object pushing is not performed on them to reduce the impact of pushing unneeded objects.

6 Related Work

Java's popularity and ever-advancing performance make Java a promising candidate for high performance computing. There are many research projects targeting at high performance Java computing in distributed or parallel environments [32].

A distributed JVM transparently exploits multi-threading support in Java to deliver high-performance parallel computing in distributed environments. In a related effort, our team implemented the JESSICA system [11] which leverages a page-based DSM [21] to build the GOS. All objects are allocated in the distributed shared memory. Each node manages a segment of shared memory and creates new objects in its own segment. Although this approach greatly alleviates the burden of constructing the GOS because all the cache coherence issues, such as object addressing, faulting, replication, and transmission, can be managed by the page-based DSM, it suffers from certain problems. First, the false sharing problem is serious due to the incompatible sharing granularity of Java and that of the page-based DSM. In comparison, the GOS described in this paper can be considered an object-based DSM. False sharing therefore is not a significant issue. Second, due to the multi-threading nature, Java's synchronization primitives may not be mappable to those provided by the page-based DSM systems that do not support multi-threading. Moreover, as a low-level support layer, the page-based DSM is not aware of the runtime information in JVM, which makes it difficult to look for opportunities to improve the performance of the GOS as we have done in this present work. The

detailed analysis of various factors contributing to the efficiency of using a page-based DSM to build the GOS can be found in [14]. Java/DSM [9] has also built its GOS on top of a page-based DSM.

cJVM [10] uses a master-proxy object model and a method shipping approach to implement the GOS. A proxy object will be created locally on accessing a remote object and the remote object becomes the master object. Method invocation of the proxy object as well as field accessing to the proxy object are shipped to the node where the master object resides. Several optimization techniques were applied to reduce the amount of such shipping [33]. No consistency issue is involved in this approach. To compare, we adopt a more aggressive object caching mechanism, and rely on our adaptive cache coherence protocol to handle the consistency issue. Since the method shipping approach may forward the execution flow to the node where the master object resides, the workload distribution is determined by the distribution of master objects in cJVM. Load balancing may be difficult to achieve without an effective strategy enforced by either the programmer or some runtime mechanism. In contrast, our synchronized method migration can be considered a selective method shipping approach, where only those synchronized methods can be the candidates for method shipping. The synchronized methods are usually short in execution time and can only be sequentially executed by multiple threads; therefore, synchronized method migration will not affect the load distribution in the distributed JVM.

Some other approaches (e.g., Jackal [8], Hyperion [7]) compile multi-threaded Java program into native code that can run on the cluster. In these systems, JVM is not involved in the execution while a software DSM is employed to provide the GOS service. Jackal uses a fine-grain DSM to build the GOS. The coherence unit is a fixed-size region of 256 bytes. Most of the effort to improve performance is done at compile time. Jackal's compiler performs two optimizations: object-graph aggregation and automatic computation migration. These two optimizations are similar to our connectivity-based object pushing and synchronized method migration. Object-graph aggregation uses a heap approximation algorithm [34] to identify those connected objects. However, the heap approximation algorithm cannot distinguish between different objects that are created in the same allocation site. Thus this approach is effective only for the situation where the related objects are from different allocation sites. In contrast, our object pushing is a runtime approach and has no such drawback. Hyperion uses a centralized table of objects in each node, and references to objects are indexed in this table. This introduces a redirection overhead on object accessing and may weaken cache locality. No adaptation is incorporated into the cache coherence protocol in Hyperion.

Most existing object-based DSM systems are language-based. They are either new parallel programming languages (e.g., Orca [15], Jade [16]), or modifica-

tions of programming languages such as C (e.g., Munin [17], Midway [18]). In both cases, the compiler or the preprocessor is leveraged to extract programmer-annotated object sharing information to direct the runtime system to choose the proper object placement, object replication policy, or even a cache coherence protocol if there are multiple protocols available. On the contrary, our GOS does not require the Java programmer to explicitly provide any information related to distributed shared objects. Our GOS itself is able to detect distributed access patterns and optimize them at runtime. Moreover, our GOS is flexible in the way that it can dynamically change the protocol if the object's access behavior changes.

Orca's runtime system can dynamically decide whether to replicate a shared object in all nodes, or to withhold the replication. In the latter case, both the compiler and runtime information are leveraged to make a decision as to where to place the object. Our GOS follows a simpler but more flexible approach compared with Orca's. In our GOS, shared objects are cached on demand, and the runtime system takes the responsibility to choose the optimal home node for the object. Also, our GOS incorporates more adaptations in other dimensions.

Munin can optimize some object access patterns. However, it requires the programmer to explicitly annotate the object with pattern declarations. Munin enumerates four access pattern declarations: *conventional*, *read-only*, *migratory*, and *write-shared*. Each pattern has its own protocol. Among them, read-only, conventional and write-shared correspond to the three patterns along the number-of-writers dimension in our access pattern space, while migratory corresponds to the accumulator pattern. Munin applies a multiple-writer protocol that corresponds to our basic protocol to the write-shared pattern, and a single-writer protocol to the conventional pattern. For the migratory pattern, the objects are migrated from machine to machine as critical regions are entered and exited.

SAM [35] is an object-based DSM runtime system that also has the support to optimize some object access patterns. SAM enumerates two patterns, corresponding to the producer-consumer pattern and the accumulator pattern in our GOS. SAM lets user explicitly tie the synchronization to the object accesses in order to perform efficient distributed shared object accesses. SAM can conduct some prefetching either based on the linkage between the lock and the data, or by the user request through some particular library calls.

Several page-based DSM systems [22][23] implement adaptive coherence protocols for a page-based access pattern at runtime. In the context of page-based DSMs, accesses to different objects residing at the same page are mingled at the page level. It is difficult to detect access patterns in applications with fine-grain sharing. In our GOS, on the other hand, accesses to different objects

can be distinguished. Furthermore, the object type information is available at runtime. Therefore, object access patterns can be detected more precisely and efficiently.

DOSA [36] implements a fine-grain DSM support for typed languages such as Java. Its aim is to keep sharing granularity at the object level but still rely on the virtual memory mechanism to check the access state as in a page-based DSM. It introduces a level of indirection on object access. Accesses to objects will go through a handle table to locate an object's actual address. Although software access check is not involved, this approach adds an additional indirection overhead to object accesses and impairs cache locality.

7 Conclusion and Future Work

This paper presents the design of a global object space for a distributed JVM. With the help of runtime object connectivity information, distributed-shared objects are separated from node-local objects to facilitate efficient consistency maintenance and memory management.

We study the object access patterns via the access pattern space. Given the space as a framework, we were able to apply three adaptations to the cache coherence protocol to achieve optimization of certain patterns. The single-writer access pattern is a popular pattern existing among the regular structured applications such as SOR and ASP. The object home migration method is useful in improving the performance of the single-writer pattern. With our GOS's ability to identify the object connectivity information at runtime, connectivity-based object pushing not only can optimize the producer-consumer access pattern, but also improve the reference locality in general. The effect of synchronized method migration is less obvious than that of the other two adaptations, but it still can improve the performance of synchronization related patterns to some extent, especially in applications with heavy synchronizations, such as ASP. After all these adaptations are enabled, considerable performance improvements have been observed.

In our future work, we plan to investigate more the optimization opportunities in the access pattern space. For example, read-only access patterns can be detected using the method similar to that for detecting the single-writer pattern. We can disable the flush operation on read-only distributed-shared objects upon synchronization until further notification. Having observed the fixed relationship between object access and synchronization, we can perform prefetching to be triggered by synchronization. Therefore, distributed-shared objects presenting the accumulator pattern should be prefetched on acquiring the corresponding lock, and those showing the assignment pattern should be

prefetched on releasing the corresponding lock.

In the current implementation, the GOS is integrated with the bytecode execution engine in interpreter mode. We plan to integrate the GOS with a bytecode execution engine in JIT mode. In JIT mode, software checking of object access state will likely be a significant overhead. However, this checking overhead can be much reduced by the JIT compiler. For example, access checks on elements of an array or fields of an object can be batched. Such techniques have already been demonstrated in some software DSMs, such as Shasta [37]. The JIT compiler may provide more optimization opportunities. For example, the objects that will be accessed in one method can be identified during JIT compilation and prefetched on demand at the later execution.

References

[1] G. Bracha, J. Gosling, B. Joy, G. Steele, The Java Language Specification, Second Edition, Addison Wesley, 2000.

[2] Sun Microsystems, Inc., The Java Hotspot Performance Engine Architecture (Oct. 1999).

[3] Java Grande Forum, http://www.javagrande.org/.

[4] R. Pozo, Numeric and Performance Issues of Java, Workshop of Clusters and Computational Grids for Scientific Computing 2002.

[5] IEEE Computer Society Task Force on Cluster Computing, Cluster Computing White Paper (Mar. 2000).

[6] G. Bell, J. Gray, High Performance Computing: Crays, Clusters and Centers. What Next? (2001).

[7] M. MacBeth, K. McGuigan, P. Hatcher, Executing Java Threads in Parallel in a Distributed-Memory Environment, in: Proc. of IBM Center for Advanced Studies Conference, 1998.

[8] R. Veldema, R. F. H. Hofman, R. Bhoedjang, H. E. Bal, Runtime Optimizations for a Java DSM Implementation, in: Java Grande, 2001, pp. 153–162.

[9] W. Yu, A. Cox, Java/DSM: A Platform for Heterogeneous Computing, in: Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation, 1997.

[10] Y. Aridor, M. Factor, A. Teperman, cJVM: a Single System Image of a JVM on a Cluster, in: Proc. of International Conference on Parallel Processing, 1999.

[11] M. J. M. Ma, C.-L. Wang, F. C. M. Lau, JESSICA: Java-Enabled Single-System-Image Computing Architecture, Journal of Parallel and Distributed Computing 60 (10) (2000) 1194–1222.

[12] K. Hwang, H. Jin, E. Chow, C. Wang, , Z. Xu, Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space, IEEE Concurrency Magazine 7 (1) (1999) 60–69.

[13] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Second Edition, Addison Wesley, 1999.

[14] W. Cheung, C. Wang, F. Lau, Annual Review of Scalable Computing, Vol. 4, World Scientific, 2002, Ch. Building a Global Object Space for Supporting Single System Image on a Cluster.

[15] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, M. F. Kaashoek, Performance Evaluation of the Orca Shared Object System, ACM Transactions on Computer Systems 16 (1).

[16] M. C. Rinard, D. J. Scales, M. S. Lam, Jade: A High Level Machine-Independent Language for Parallel Programming, Computer 26 (6) (1993) 28–38.

[17] J. B. Carter, J. K. Bennett, W. Zwaenepoel, Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems, ACM Transactions on Computer Systems 13 (3) (1995) 205–243.

[18] M. J. Zekauskas, W. A. Sawdon, B. N. Bershad, Software Write Detection for a Distributed Shared Memory, in: Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI), 1994.

[19] L. Iftode, Home-based Shared Virtual Memory, Ph.D. thesis, Princeton University (August 1998).

[20] P. Keleher, S. Dwarkadas, A. L. Cox, W. Zwaenepoel, TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, in: Proc. of the Winter 1994 USENIX Conference, 1994, pp. 115–131.

[21] B. Cheung, C. Wang, K. Hwang, A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations, in: International Conference on Parallel and Distributed Processing Techniques and Applications, 1999, pp. 821–827.

[22] C. Amza, A. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, W. Zwaenepoel, Adaptive Protocols for Software Distributed Shared Memory, in: Proceedings of IEEE, Special Issue on Distributed Shared Memory, Vol. 87, 1999, pp. 467–475.

[23] L. R. Monnerat, R. Bianchini, Efficiently Adapting to Sharing Patterns in Software DSMs, in: the 4th IEEE International Symposium on High-Performance Computer Architecture, 1998.

[24] J. Manson, W. Pugh, Core Semantics of Multithreaded Java, in: ACM Java Grande Conference, 2001.

[25] P. Keleher, A. L. Cox, W. Zwaenepoel, Lazy Release Consistency for Software Distributed Shared Memory, in: Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92), 1992, pp. 13–21.

[26] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, S. P. Midkiff, Escape Analysis for Java, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999, pp. 1–19.

[27] D. Plainfoss, M. Shapiro, Survey of Distributed Garbage Collection Techniques, in: Proc. of Interational Workshop on Memory Management, 1995.

[28] J. M. Piquer, I. Visconti, Indirect Reference Listing: A Robust Distributed GC, in: Euro-Par '98 Parallel Processing, 1998.

[29] P. R. Wilson, Uniprocessor Garbage Collection Techniques, in: Proc. Int. Workshop on Memory Management, no. 637, Springer-Verlag, Saint-Malo (France), 1992.

[30] Kaffe Java Virtual Machine, http://www.kaffe.org.

[31] The HKU Gideon 300 Cluster, http://www.csis.hku.hk/˜clwang/gideon300-main.html.

[32] M. Lobosco, C. L. Amorim, O. Loques, Java for High-Performance Network-Based Computing: A Survey, Concurrency and Computation: Practice and Experience (14) (2002) 1–31.

[33] Y. Aridor, M. Factor, A. Teperman, T. Eilam, A. Schuster, Transparently Obtaining Scalability for Java Applications on a Cluster, Journal of Parallel and Distributed Computing 60.

[34] R. Ghiya, L. J. Hendren, Putting Pointer Analysis to Work, in: 25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1998, pp. 121–133.

[35] D. J. Scales, M. S. Lam, The Design and Evaluation of a Shared Object System for Distributed Memory Machines, in: Operating Systems Design and Implementation, 1994, pp. 101–114.

[36] Y. C. Hu, W. Yu, D. Wallach, A. Cox, W. Zwaenepoel, Runtime Support for Distributed Sharing in Typed Languages, in: the Fifth ACM Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, 2000.

[37] D. J. Scales, K. Gharachorloo, C. A. Thekkath, Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, in: Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII), 1996, pp. 174–185.