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

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

Abstract

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* in distributed JVM, which virtualizes a single Java object heap across machine boundaries to facilitate transparent object accesses. We leverage runtime object connectivity information to detect objects, called *distributed-shared objects*, that are reachable from threads at different nodes to address memory consistency issues in the distributed JVM. With the detection of distributed-shared objects, we can reduce memory consistency maintenance overhead and provide efficient distributed garbage collection.

We propose a framework to characterize object access patterns, along three orthogonal dimensions. Under the framework, we propose and apply three effective adaptations to the cache coherence protocol to optimize several access patterns that frequently appear during application execution, including an *object home migration* method that adapts to the single-writer access pattern, *synchronized method migration* that allows the remote execution of a synchronized method at the home node of its locked object, and *object pushing* that uses the object connectivity information to adapt to the producer-consumer access pattern.

Keywords: Java, cluster computing, Distributed JVM, Distributed shared memory, Adaptive cache coherence protocol.

1 Introduction

The Java programming language [7] provides multi-threading support for explicit parallelism in a program. Parallel programs in Java are generally more portable than those using other parallel languages or runtime libraries such as PVM, MPI, or software DSM. Recent advances in Java compiling

and execution technology, such as just-in-time compiler and the hotspot technology [35], are making Java an attractive tool for high performance computing [14]. Some performance benchmark results even indicates that Java can outperform the C programming language in some numerical computation programs for scientific and engineering applications [30].

In recent years, cluster [17] [6] has gradually been accepted as a scalable and affordable parallel computing platform by both academia and industry. A lot of effort [24][36][39][2][23] has been devoted to supporting transparent and parallel execution of multi-threaded Java programs in a cluster of computers. Among many such projects, the Hyperion [24] and Jackal [36] systems compile multi-threaded Java programs directly into distributed applications in native code; Java/DSM [39], cJVM [2], and JESSICA [23] are *distributed JVMs* that conform to the JVM specification [21] but runs on a cluster of computers. As a middleware for the cluster, a distributed JVM presents a *single system image* (SSI) to Java applications on top. With ideal SSI, not only the Java threads created within one program can be distributed to different cluster nodes to achieve a high degree of execution parallelism, but cluster-wide resources such as memory, I/O, and network bandwidth can be aggregated to solve large-size problems. Given that the distributed JVM conforms to the JVM specification, any pure Java program can run on the distributed JVM without any modification in both the Java source code and the bytecode.

The above paradigm of parallel Java programming will boost programming productivity. First, the steep learning curve does not exist in this paradigm since the programmers do not need to learn a new language, new libraries, or tools in order to develop parallel programs. Second, this paradigm works in the convenient mode that programs can be developed in a single machine, and then submitted to a parallel computer for execution. Finally, many existing multi-threaded Java applications, especially server applications, can be automatically parallelized with minimal effort.

In a distributed JVM, the shared memory characteristics of Java threads call for a *global object space* (GOS) that "virtualizes" a single Java object heap. The heap spans the whole cluster in order to facilitate transparent object access. The GOS indeed is 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. The performance of the distributed JVM hinges on the GOS's ability to minimize the communication and coordination overheads in maintaining the single object heap illusion.

Many distributed JVMs use a page-based DSM to build the GOS. 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 the mismatch between the object-based memory model of Java and the underlying page-based implementation. For instance, the false

sharing problem occurs because of the incompatible sharing granularities of the variable-sized Java objects and the fix-sized virtual memory pages. This mismatch has also prevented further optimizations on the cache coherence protocol that implements the Java memory model. A more effective and efficient solution for object sharing among distributed Java threads is needed.

In this paper, we propose a new global object space design for 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 applications' object access patterns.

Many cache coherence protocols have been proposed in the field of software distributed shared memory. Home-based protocols [18] 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 kind has less memory consumption and eliminates diff accumulation. The home in a home-based protocol can be either fixed [18] or migratable [9]. There are also choices for the coherence operations, such as that between a multiple-writer protocol or a single-writer protocol. The multiple-writer protocol introduced in Munin [8] supports concurrent writes on different copies using the diff techniques. It may however incur heavy diff overhead compared with conventional single-writer protocol. Another choice is between the update protocol (e.g., Orca [5]) and the invalidate protocol in many page-based DSM systems (e.g., [20][9]). The update protocol can do prefetching to make the data available before the access, but it may send much unneeded data compared with the invalidate protocol.

We found the performance of a coherence protocol to be often application-dependent. That is, the particular memory access patterns in a application determine which protocol is more efficient and therefore suitable. That motivated us to go after an adaptive protocol.

An adaptive cache coherence protocol is able to detect the current access pattern and adjusts itself accordingly. Some DSMs (e.g., Munin [8]) 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 cannot dynamically switch between different protocols in response to changes in access pattern. Several page-based DSM systems [1][26] support adaptive protocols that can automatically adapt to the access pattern at runtime. However, the access pattern these systems observe is the page-level approximation of the actual pattern. They may not be effective if the page-level approximation is different from the actual pattern, such as when the application has fine-grained sharing granularity. An object-based adaptive protocol should be more flexible, which is what we have designed and implemented.

The challenges of designing an effective and efficient adaptive cache coherence protocol are: (1) whether we can determine those adaptations that are able to optimize the access patterns in the context where the protocol is applied, and (2) whether the runtime system can efficiently identify those target access patterns and apply the corresponding adaptations.

To understand more the first challenge and to subsequently 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 dimension of number of writers; mutual exclusion and condition for the dimension of synchronization; and patterns with different number of consecutive repetitions for the dimension of repetition. The combination of different basic patterns then portrays an actual access pattern. This access pattern space serves as a systematic foundation on which we can identify those significant object access patterns existing in distributed JVM. We can then choose the right adaptations to match with and optimize these access patterns.

To meet the second challenge, we take advantage of the fact that our GOS is implemented by modifying the heap subsystem of JVM. Our adaptive protocol can leverage all runtime object types and access information to efficiently and accurately identify the access patterns we are interested in.

We leverage runtime object connectivity information to detect objects we call *distributed-shared objects* (DSO). DSOs are defined to be those reachable from at least two threads located at different cluster nodes in the distributed JVM. The distinction of DSOs allows us to more efficiently address the memory consistency issue. In Java, synchronization primitives are not only used to protect critical sections but also to maintain memory consistency. We show that only locks on DSOs need to trigger distributed memory consistency maintenance. The detection of DSOs cuts down on the frequency of distributed consistency maintenance. Since only DSOs that may be replicated on multiple nodes need to be involved in consistency maintenance, the detection of DSOs therefore leads to a more efficient consistency protocol.

We apply three different protocol adaptations to the basic home-based multiple writer cache coherence protocol in three different situations respectively in the access pattern space: (1) *object home migration* which optimizes the single writer access pattern by adapting the object's home to the writing node according to the access history; (2) *synchronized method migration* which chooses between default object shipping and method shipping in order to optimize the execution of critical section methods according to some a priori knowledge; (3) *object pushing* which adapts the transfer unit to optimize *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 as well as 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 since the memory model 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 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. The Java language provides the *synchronized* keyword, used in either a synchronized method or a synchronized statement, for synchronization. Entering into and exiting from a synchronized block correspond to acquiring and releasing the lock of the corresponding object respectively. In Java, synchronization is used not only to protect critical sections, but also to maintain memory consistency among threads.

Our GOS follows the JMM proposed in [25]. When a thread T_1 acquires a lock that was most recently released by another thread T_2, all writes that were visible to T_2 at the time of releasing the lock become visible to T_1. This is the release consistency [19].

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* shared by all threads. Before a thread acquires a lock, it must flush (throw away) all variables in its working memory, and therefore later uses will load the values from the main memory.

2.2 Access Pattern Space

An object's access behavior can be described as a set of reads and writes performed on the object, interleaved with synchronization actions such as locks and unlocks. These reads and writes as well as locks and unlocks may be issued concurrently by multiple threads. The locks and unlocks may not

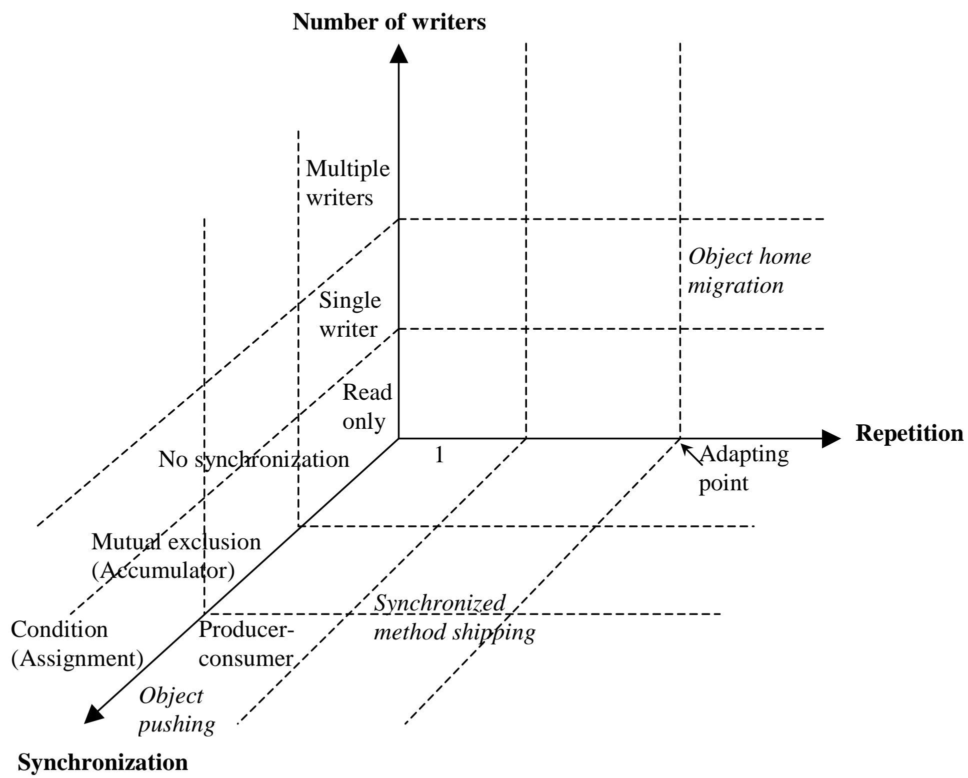


Figure 1: Object access pattern space (The items in italic font are the corresponding adaptations. The basic protocol is not shown.)

be invoked on that particular object, but they will still influence the object's accesses according to the JMM. 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.

Three orthogonal dimensions defining the characteristics of object access behavior can be observed: *number of writers*, *synchronization*, and *repetition*. We therefore propose a 3-dimensional access pattern space based on these three characteristics, as shown in Fig. 1.

Number of writers characterizes how many nodes there are in which a thread can be 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 in single writer pattern, where the object is accessed by only one node.
- *Read only*: no node writes to the object.

Synchronization characterizes the execution order of accesses from 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. We distinguish 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. Notice that although Java implements the *monitor* concept, it does not require that the lock protecting the critical section should belong to the object accessed inside.
- *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.

Repetition characterizes the number of consecutive repetitions of an access pattern. It is desirable that the access pattern will repeat for a considerable number of times so that the GOS will be able to detect the pattern using the history and to apply some adaptation to optimize the

pattern's reoccurrence. Such a pattern will appear on the right side of the adapting point along the repetition axis. The adapting point is an internal threshold in the GOS. When the pattern repeats for more times than the adapting point, the corresponding adaptation will be automatically performed. For instance, the single writer pattern can be optimized using this approach.

However, there are some significant access patterns that repeat only once, such as the *producer-consumer* pattern. 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 detection of the producer-consumer pattern therefore cannot depend on access history information, and other information and heuristics should be used instead.

3 Distributed-shared Object

In this section, we define distributed-shared object (DSO), and discuss the benefits of distinguishing between DSOs and NLOs (node-local objects), as well as the detection of DSOs. We then present our basic cache coherence protocol and the distributed garbage collection scheme in the GOS.

3.1 Definition

In JVM, *connectivity* exists between two Java objects if one object contains a reference to another. Therefore, we can conceive the whole picture of object heap to be a *connectivity graph*, where the vertices represent objects and the 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 (1) its reference resides in the thread's stack, or (2) there is some path existing in connectivity graph between it and some known reachable object.

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 some escape analysis technique [11].

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 distributed-shared object and node-local object, as follows.

Node-local object is an object reachable from thread(s) in the same node. It is either a thread-local object or a thread-escaping object.

Distributed-shared object 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 threads' working memories 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. The issue of distributed consistency only emerges in 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 NLO do not need to trigger distributed consistency maintenance, because all threads able to acquire or release the lock of an NLO must reside in the same node and therefore do 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 they are in the consistent state in 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) [29]. The detection of DSO 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 on garbage. The detection of DSOs enables independent memory management on each node. The memory management and distributed garbage collection will be discussed in Section 3.5.

3.3 Lightweight Detection

In distributed JVM, whether an object is a DSO or NLO is determined by the relative locations of the object and the threads reaching it. The locations of objects and threads can only be determined at runtime. We propose a runtime lightweight DSO detection scheme which leverages on Java's type information which is available at runtime.

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, the object connectivity graph is available at runtime. The graph is dynamic since connectivity between objects may change from time to time through the reassignment of objects fields.

The DSO detection is performed when there are some JVM runtime data to be transmitted across node boundary, which are either thread stack context for thread relocation or object content for remote object access. On both the sending and the receiving side, these data are examined to identify 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 its 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 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 detected as DSOs.

We detect DSO in a lazy fashion. Since it is still 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.

3.4 Basic Cache Coherence Protocol

Fig.2 illustrates the lifecycle of an object in our GOS from its creation to possible collection. If an object is a DSO, after it is detected, it will be replicated in multiple nodes and suffer from the consistency problem. This subsection discusses our basic cache coherence protocol to handle the consistency problem. The garbage collection of both NLOs and DSOs will be discussed in the next subsection.

In the dimension of number of writers in the access pattern space, the multiple-writers pattern can be considered the generalized form of all patterns. Both the single-writer pattern and the read-only pattern are special cases of the multiple-writers pattern, but with some dumb writers. Therefore, our basic cache coherence protocol is a home-based multiple-writer cache coherence protocol. Its state transition graph is included in Fig. 2. The 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

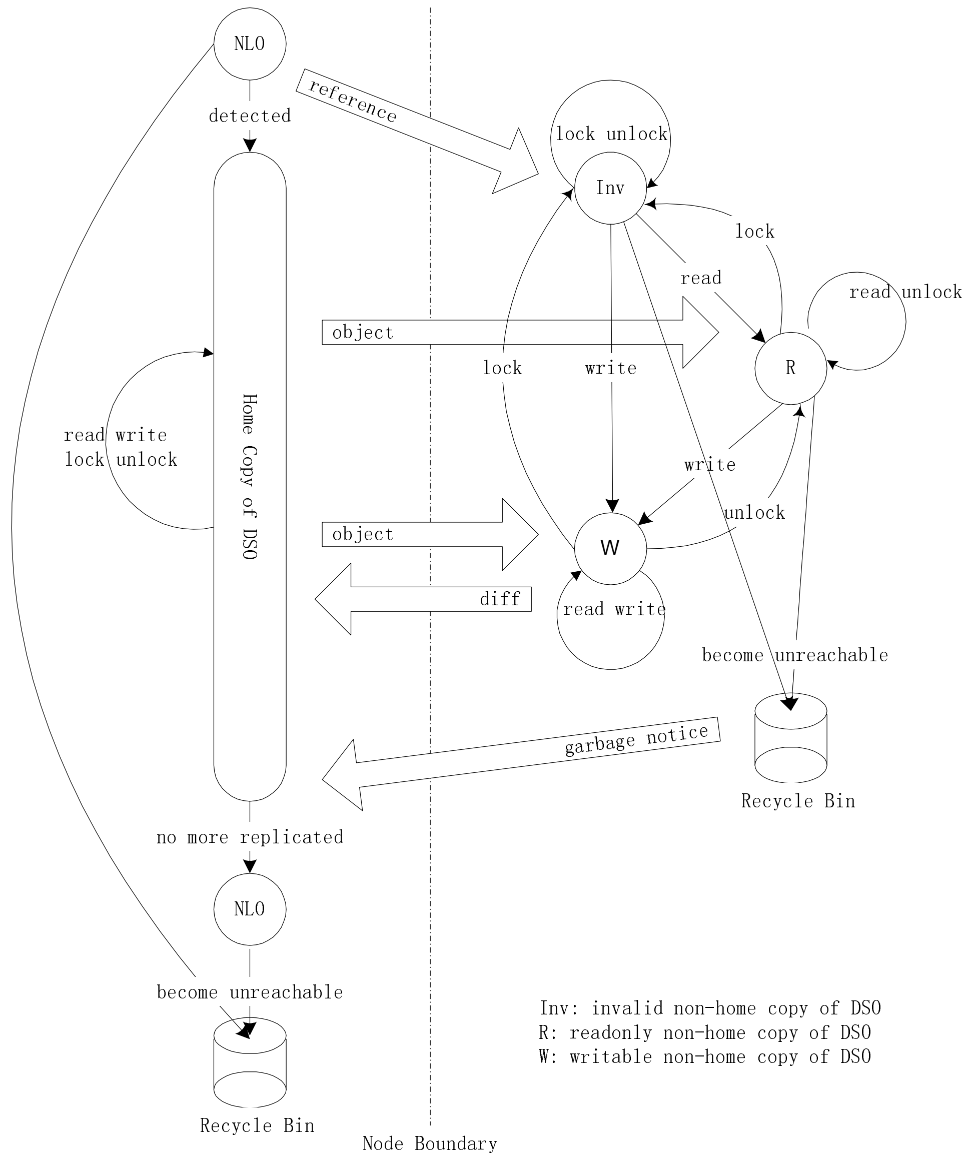


Figure 2: Object lifecycle in GOS

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 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.

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 at 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.

Due to 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 the instances of class `String`, `Integer`, and `Float`, can be simply replicated and treated as an NLO afterwards. Some objects are considered node-dependent resources, such as the 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 guaranteeing complete single system image to Java applications.

3.5 Distributed Garbage Collection

Unlike other GOS approaches [23][39][36], we do not assume among the nodes a shared virtual memory address space where the copies of an object occupy the same virtual memory addresses in all the nodes. Instead, we make use of Java's runtime type information to unambiguously identify pointers, i.e. object references in Java context. By that we are able to do pointer translation across node boundary and relocate objects to different addresses in different nodes. In this way, the heap management of each node is totally decoupled, and all the nodes are coordinated to present a huge virtual heap.

A DGC algorithm, *Indirect Reference Listing* [28], is adopted to collect DSOs that have turned into garbage. We use an indirect reference listing (IRL) algorithm to maintain a distributed reference diffusion tree for each DSO. Each copy of a DSO would maintain two lists, an import list recording where its reference comes from, and an export list recording where its reference is sent to. In a DSO's reference diffusion tree, every vertex represents a

node possessing one of its copies. The root of the tree is its home node. An edge in the tree represents that the reference is transmitted from one node to another node. The sending node adds the receiving node into its export list, while the receiving node adds the sending node into its import list. If the node to be added is already in the list, the addition has no effect.

When a non-home copy of a DSO meets the following two conditions, it can be reclaimed locally and a *garbage notice* will be sent to its parent in the diffusion tree: (1) its export list is empty; and (2) it is not reachable from the local root set, which can be determined by the local collector. If one node receives a garbage notice of a DSO, it will remove the sending node from the DSO's export list. When the export list of the home copy of a DSO becomes empty, it is converted to an NLO. IRL cannot collect a cycle of garbage DSOs whose home nodes are different. However, this is not a serious problem in the GOS.

The transmission path of a DSO reference may form some cycle among the nodes. The export list in every node in the cycle is non-empty and all the copies will be put into the local root sets. The result is that this DSO will never be reclaimed even it is not reachable from anywhere. In order to avoid such cycles polluting the structure of the diffusion tree, we make sure that each node can only have one valid parent in the tree. If a DSO reference arrives from a node different from the current parent, the sender will not be added into the import list. Instead, the receiver prepares a pseudo garbage notice for the sender, since the sender has already added the receiver into the export list. Having received the pseudo garbage notice, the sender can remove the receiver from its export list.

The major overheads of the IRL are due to maintaining import and export lists for every DSO as well as sending garbage notices. The list maintenence coexists with the reference transmission. Compared with the transmission, the maintenence overhead is negligible. The garbage notices can be buffered and piggybacked on coherence messages. So the IRL will not contribute significant overhead to the GOS.

With the IRL in place, each node independently garbage-collects its local heap using a mark-sweep collector [38]. All the DSO copies with non-empty export list would be put into the root set. All the non-home copies of DSOs that are inconsistent with their home copies, i.e. in write state, would also be put into the root set [13] to tolerate the inconsistency.

4 An adaptive Cache Coherence Protocol

Only DSOs would suffer from consistency problems since they are replicated in multiple nodes. In the last section, we have presented the home-based multiple writer cache coherence protocol for handling the consistency issues. However, as we explained before, the non-adaptive protocol can never be

optimal in all circumstances with different access patterns. The adaptive protocols are superior to non-adaptive ones due to their adaptability to applications' access patterns. In this section, we discuss the adaptations we apply to the basic protocol within the framework of the 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.

In the GOS, a runtime mechanism is applied to determine the optimal home of a DSO and to perform object home migration automatically, which is to reselect a node as the home node.

We only apply object home migration to those DSOs exhibiting the single-writer access pattern. If the DSO exhibits the multiple-writer pattern, all the non-home nodes will communicate with the home node in order to obtain the up-to-date copy and propagate the writes. Therefore, it does not matter which is the home node as long as the home node is one of the writing nodes. Moreover, object home migration may have negative impacts on performance. In order to notify a node not aware that the object home has already been migrated to a new home, a redirection message should be sent. Improper migration will lead to a lot of redirection overhead.

If a DSO exhibits the single-writer pattern and its home is made the only writing node, the diff creation and application overhead 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 incurred.

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 the diff received on synchronization points 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, the 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. The number of consecutive writes is the number of synchronization during which the object was only updated by that node. We follow a heuristic that an object is in the single-writer pattern if the number of consecutive writes exceeds a predefined threshold.

If the single-writer pattern is detected, the object home is migrated to

the writing node. 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 can be a building block of those access patterns.

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

Fig. 3 shows the skeleton of a Java implementation of the barrier function. The execution can not continue until all the threads invoke the `barrier` method. We suppose 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, as well as maintain distributed consistency. In line 8, `barrier` object will be faulted-in. It is a common behavior that the locked object's fields will be accessed in synchronized method. In line 9 or line 11, another synchronization request, either `wait` or `notifyAll`, will be issued. The `wait` method will also trigger the operation to maintain distributed consistency according to the JMM[1]. Therefore, there are four synchronization or object requests sent to the home node and multiple distributed consistency maintaining operations involved.

Migrating a synchronized method of a DSO to its home node for execution will combine multiple message roundtrips into one and reduce the overhead to maintain 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 redistribute among the nodes. However, the synchronized methods are usually short in terms of the execution time and can only be sequentially executed by multiple threads, therefore, synchronized method migration will not affect the load distribution in distributed JVM.

4.3 Object Pushing

The producer-consumer pattern, which is single assignment, is a significant access pattern in Java programs. Usually, in a producer-consumer pattern,

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

```
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   }
```

Figure 3: Barrier class

one thread prepares an object tree, and notifies another thread to access the tree. In distributed JVM, the performance suffers from the consuming thread requesting objects in the tree one by one from the node where the producing thread resides. Because the producer-consumer pattern only repeats once, we can not do prediction according to the history.

We use object pushing to optimize the producer-consumer pattern. Object pushing speculates on future accesses according to runtime object connectivity information instead of the access history. 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 requested for a DSO, the home node pushes all the objects that are reachable from it to the requesting node. Essentially, object pushing is a prefetching strategy which takes advantage of the reference locality existing in Java program execution to achieve an aggregation effect on communication.

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 result in wastage of communication bandwidth. On the contrary, object pushing provides more accurate prefetching since the home node has the up-to-date copies of the objects and the connectivity information at the home node is always valid.

In the implementation, we rely on 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. 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. In 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 risks sending unneeded objects. To reduce the negative impact of pushing unneeded objects, the GOS will not push large-size objects. The GOS will also not perform object pushing upon request of arrays of reference type, e.g. multi-dimension arrays, since arrays of reference type usually represent some workload shared among threads.

5 Performance Evaluation

In this section, we study the performance of the GOS as well as the effects of the adaptations discussed in Section 4.

Our distributed JVM implementation is based on the Kaffe JVM [37] which is an open source JVM. The GOS is integrated with the bytecode execution engine which is in interpreter mode. In multi-threaded Java applications, when a Java thread is started, it can be automatically dispatched to some underloaded cluster node to achieve maximum parallelism and load balance.

We conducted the performance evaluation on the HKU Gideon 300 cluster [12], which is a cluster of PCs with Intel 2GHz P4 CPU, running Linux kernel 2.4.18, connected by a fast ethernet.

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

5.1 Application Performance

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

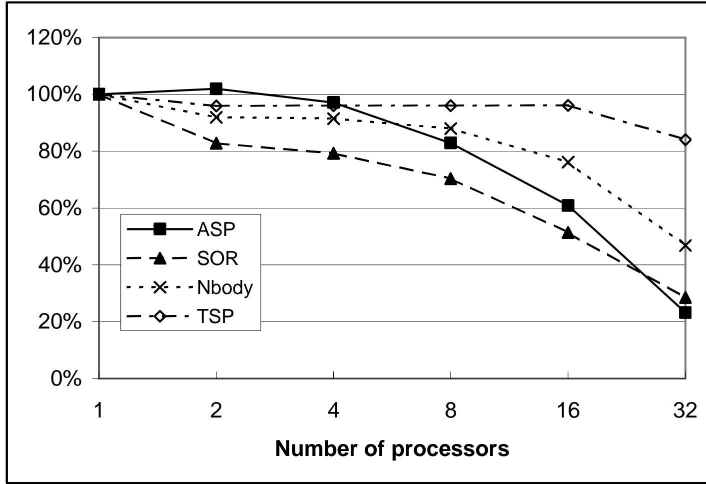


Figure 4: Efficiency

Fig. 5 shows the average percentage of normalized execution time breakdown against number of processors for the four applications. In the legend, Comp denotes the computation time; Obj denotes the time to request an up-to-date copy of an invalid object; Syn denotes the time spent on synchronization operations, such as lock, unlock, wait, and notify; GC is the garbage collection overhead. Notice that not every application requires the GC. The Obj and Syn portions 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 processing and the time spent on the wire, but also the possible waiting on the requested node. For example, if the requested node is busy with other operations, or the request cannot be fulfilled immediately (e.g., a lock request for a lock held by others), the request has to suspend. The DSO detection overhead as well as the overhead to translate object reference across node boundary are always present with the communication. They are however insignificant when compared with the communication cost.

ASP requires n iterations to solve an n-nodes graph problem. The graph is represented by the distance matrix. The workload is distributed equally among the threads row wise. 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, which requires all threads' participation. The Java language does not directly provide any barrier operation among threads, and so the barrier should be implemented using synchronized primitives, as shown in Fig. 3. We can see in Fig. 5 that the Syn portion increases faster than the number of processors. The efficiency of synchronization, especially the barrier operation, for the case of large number of processors is critical to the performance of the distributed JVM.

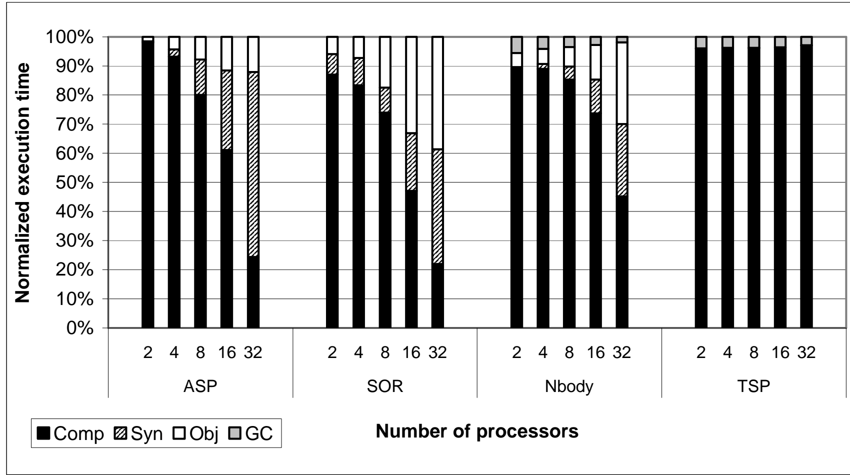


Figure 5: Normalized execution time breakdown against number of processors

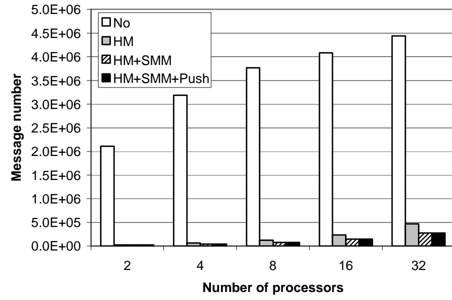
SOR performs red-black successive over-relaxation on a 2-D matrix for a number of iterations. The workload is distributed equally among all the threads row wise. There are two barriers in each iteration. The situation of SOR is similar to that of ASP. The Syn operation contributes a significant percentage to the execution time when scaled to a large number of processors.

Nbody also involves synchronization in each simulation step. Moreover, in Nbody, the construction of the quadtree in each simulation step cannot be parallelized. When the main thread conducts the construction, all other threads are waiting. The efficiency decreases while the number of processors increases. This is another factor affecting the efficiency curve of Nbody. As Nbody is scaled up to more processors, the GC portion scales down due to larger aggregated heap size.

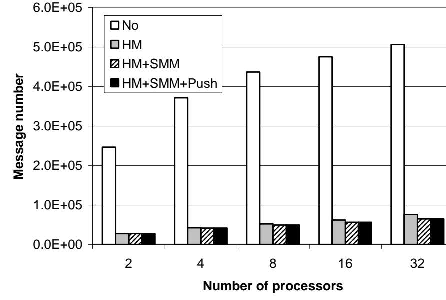
TSP prunes large parts of the search space by ignoring partial routes already longer than the current best solution. The program divides the whole search tree into many small ones to build up a job queue in the beginning. Every thread will get jobs from this queue until the queue is empty. TSP is a computation intensive program when compared with the other applications. Therefore, it is able to achieve an almost horizontal efficiency curve even for a relatively small problem size.

5.2 Effects of Adaptations

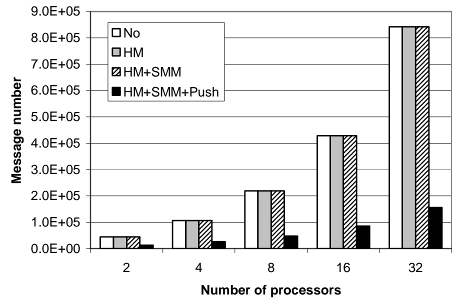
In the experiments, we incrementally enable the planned adaptations. All adaptations are disabled initially. Then object home migration is enabled, followed by synchronized method migration which is enabled while object



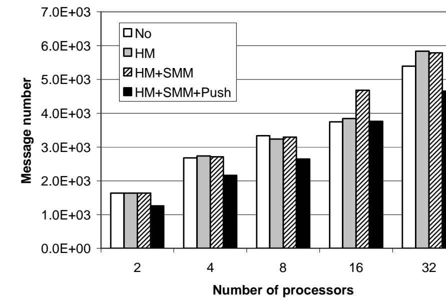
(a) ASP



(b) SOR

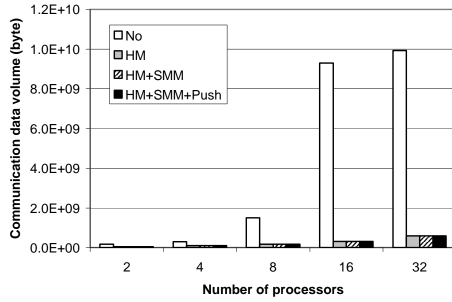


(c) Nbody

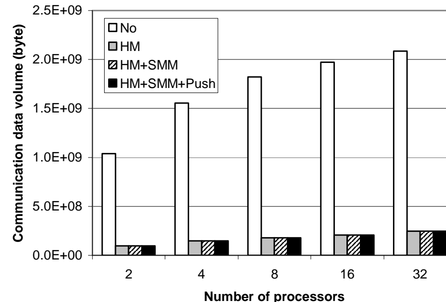


(d) TSP

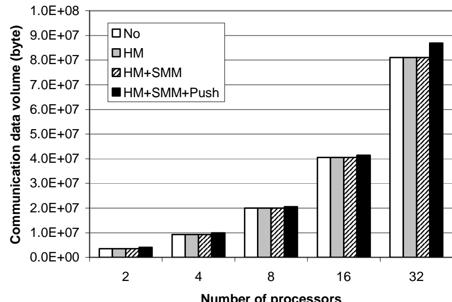
Figure 6: Effects of adaptations w.r.t. the message number



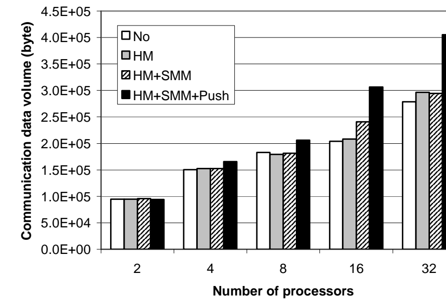
(a) ASP



(b) SOR

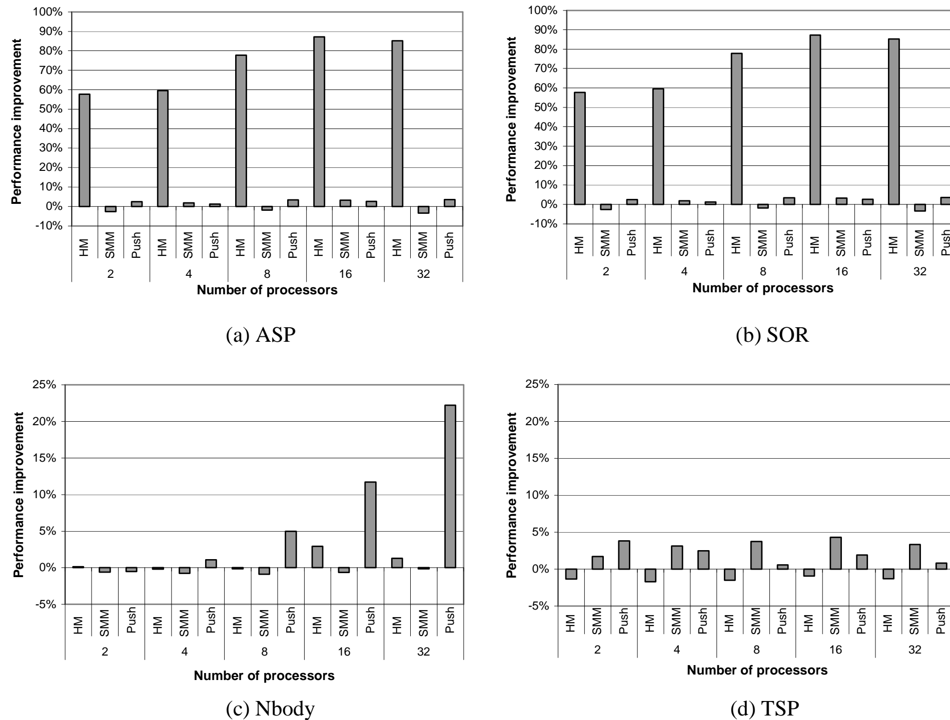


(c) Nbody



(d) TSP

Figure 7: Effects of adaptations w.r.t. the communication data volume



(a) ASP (b) SOR

(c) Nbody (d) TSP

Figure 8: Relative performance improvement of adaptations

home migration is in effect. Finally, we enable object pushing, and all adaptations are in effect. In the legend, No denotes no adaptive protocol enabled, HM denotes object home migration, SMM denotes synchronized method migration, and Push denotes object pushing.

Fig. 6 and 7 respectively show the effects of adaptations on the number of messages and the communication data volume. We present the data against the number of processors.

Fig. 8 shows the relative performance improvement in terms of decreased execution time for all adaptations against the number of processors. In the figure, a column presents the percentage of decrease in execution time when the corresponding adaptation is enabled against the execution time before it was enabled. The negative column indicates that the execution time increases when the adaptation is enabled.

As seen from the figures, object home migration greatly improves the performance of ASP and SOR. In ASP and SOR, the data are represented by 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 those 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 traffic. We can see that object home migration dramatically reduces the number of messages and the communication volume, as well as the execution time. 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, we can say that the multiple-writer object dynamically changes its pattern to single-writer for a short while, and then changes back to multiple-writer. Future accesses will not gain any performance improvement if they will not exhibit such a pattern after home migration, but may 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. This negative impact cannot be totally avoided due to our heuristics. But as can be seen in Fig. 8(d), the negative impact is quite limited, less than two 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 data volume, it reduces the number of messages significantly, as seen in the ASP and SOR cases. As shown in Fig. 8(b), synchronized method migration improves SOR's overall performance to some extent. However, the decrease in execution time due to synchronized method migration is not clear in ASP. As seen from Fig. 8(a), synchronized method migration may either increase or decrease the execution time. We may attribute this to ASP's heavy synchronization operations. ASP requires n barriers for all threads in order to solve an n-node graph. As we mentioned before, 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 the benefit from synchronized method migration. Nbody's synchronization uses synchronized block instead of synchronized method, and so synchronized method migration should have no effect on it. TSP has very limited communication and synchronization. Synchronized method migration also improves TSP's performance a little.

Object pushing optimizes the producer-consumer access pattern and improves reference locality by aggregating small object messages. 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 size 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 communication amount increases a little. The gain of object pushing in terms of decrease in execution time is also very obvious in Nbody, as seen from Fig. 8(a). When Nbody is scaled up to more processors, the communication effort due to the producer-consumer pattern increases proportionally. Therefore, the effect

of object pushing is amplified. Object pushing also improves access locality for objects with referential relationship. Notice that the effort of communication is relatively small in TSP. Although the adaptations decreases the communication time, the total execution time decrease due to adaptations 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. Object pushing has a little positive effect on ASP and a little negative effect on SOR.

6 Related Work

Java's popularity and ever-advancing performance make Java a promising candidate for high performance computing. There are many research projects [22] targeting at high performance Java computing in distributed or parallel environments. Some extend the Java language grammar to meet this challenge. For example, JavaParty [27] requires programmers to explicitly add the `remote` keyword in front of all distributed shared objects in Java source code. Then a specialized preprocessor will translate the source code to a distributed application using RMI [34]. Many others are aware of the inadequacy of Java classes to support distributed computing. They tried to invent new classes to fix the problem. For example, mpiJava [4] implements a Java interface for the native MPI library. However, all the above approaches are not transparent to the Java programmer. They require the programmer to handle issues related to the distributed environment explicitly. On the contrary, a distributed JVM transparently exploits multi-threading inherent in Java programs to implement high performance parallel computing in distributed environments. One of the core components of a distributed JVM is the GOS which virtualizes a single object heap.

In a related effort, we implemented the JESSICA system [23] which leverages a page-based DSM, JUMP [9], 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 without interaction with others. The copies of an object reside at the same virtual memory address in each node. Although this approach greatly alleviates the burden of constructing GOS because all the cache coherence issues, such as object addressing, faulting, replication, and transmission, can be managed by page-based DSM, it suffers from many problems. First, 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 if the page-based DSM does 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 paper. The detailed analysis of various factors contributing to the efficiency of using a page-based DSM to build the GOS can be found in [10]. Java/DSM [39] has also built its GOS on top of a page-based DSM.

cJVM [2] uses a master-proxy object model and a method shipping approach to implement the GOS. Method invocation of and 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 [3]. This approach is appropriate for the sequential consistency memory model. However, under the proposed Java memory model, i.e. release consistency, this approach is not efficient since every object access and method invocation may require communication. A more aggressive object caching mechanism, like that in our GOS, should be adopted. Since the method shipping approach may forward the execution flow to the node where the master object reside, the workload distribution is determined by the distribution of master objects in cJVM. Load balance may be difficult to achieve without an effective strategy enforced by either the programmer or some runtime mechanism.

Some other approaches reply on compiler techniques to transparently run multi-threaded Java applications on a cluster. They compile a multi-threaded Java program to distributed native code. In these systems, JVM is not involved in the execution while a software DSM is employed to provide the GOS service.

Jackal [36] directly compiles Java source code to native code. Similar to JESSICA, all nodes share a virtual memory address space in Jackal, and each node manages a segment of the address space. Different from JESSICA, 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, whose aims are similar to those of our object pushing and synchronized method migration. Object-graph aggregation uses a heap approximation algorithm [15] to identify those related objects. However, the heap approximation algorithm cannot distinguish between different objects that are created at the same allocation site. Thus this approach is effective only for the situation when the related objects are from different allocation sites. In contrast, our object pushing is a runtime approach and has no such drawback. Similar to Jackal, Hyperion [24] compiles Java bytecode to C source code, and then to native code.

Orca [5] and Jade [31] are object-oriented parallel programming languages. With distributed JVM as the executing platform, Java can also be considered a parallel programming language. Java's popularity makes it more promising to be acceptable by the parallel computing community than

Orca and Jade.

Munin [8] and SAM [33] are object-based DSMs with supports to optimize some object access patterns. However, they require the programmer to explicitly annotate the object with some pattern declaration. Compared to our GOS, their approach is neither transparent to the programmer nor flexible in a dynamic situation. Munin enumerates five access pattern declarations: *conventional*, *read-only*, *migratory*, *write-shared*, and *synchronization*. Each pattern has its own protocol. Among them, read-only, conventional and write-shared correspond to the three patterns along the dimension of the number of writers in our access pattern space, while migratory corresponds to the accumulator pattern. Synchronization is actually the declaration for synchronization variables. SAM enumerates two patterns, values with a single-assignment semantics and accumulator. The former corresponds to the producer-consumer pattern in the access pattern space.

Several page-based DSM systems [1][26] implement adaptive coherence protocols for a per-page 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 of 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. Similar to our access pattern space, sharing pattern categorization is proposed to specify access patterns in ADSM [26]. However, sharing pattern categorization is only based on the association between locks and the data they protect. Hence, it corresponds to the synchronization dimension in our access pattern space. The producer-consumer pattern in ADSM is different from ours in definition. In fact theirs should be single-writer, multiple-readers, and multiple-assignment according to our access pattern space. Comparatively, our definition of producer-consumer pattern gives restriction on its repetition.

DOSA [16] implements a fine-grain DSM support for typed language 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 accessing. 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 distributed JVM. With the help of runtime object connectivity information, distributed-shared

objects are separated from node-local objects to facilitate efficient memory consistency maintenance and distributed garbage collection.

We propose the access pattern space as a framework to characterize object access patterns. Given the framework, we were able to apply three adaptations to the cache coherence protocol to optimize some significant patterns, including an object home migration method that adapts to the single-writer access pattern, synchronized method migration that allows the remote execution of a synchronized method at the home node of its locked object, and object pushing that uses the object connectivity information to adapt to the producer-consumer access pattern. After all these adaptations are enabled, considerable performance improvements have been observed.

In our future work, we plan to investigate more on optimization opportunities in terms of adaptations to object access patterns under the framework of 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 on 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, the software check of object access state will become a significant overhead. However, this check overhead can be greatly 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 DSM, such as Shasta [32]. 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 if demanded at the later execution.

The GOS's current DGC algorithm is based on indirect reference listing. Each distributed-shared object should maintain a list of the nodes that import its reference. This is a potential problem that hinders the scalability of GOS. In the future, we plan to adopt indirect reference counting, where each distributed-shared object only needs to remember how many times its reference is transmitted to other nodes. Compared with indirect reference listing, indirect reference counting requires a smaller memory footprint when the GOS is scaled to larger-size cluster.

References

[1] C. Amza, A.L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. In *Proceedings of IEEE, Special Issue on Distributed Shared Memory*, volume 87, pages 467–475, March 1999.

[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] Yariv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster. Transparently Obtaining Scalability for Java Applications on a Cluster. *Journal of Parallel and Distributed Computing*, 60, Oct. 2000.

[4] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava: An Object-Oriented Java interface to MPI. In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, April 1999.

[5] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), Feberury 1998.

[6] G. Bell and J. Gray. High performance computing: Crays, clusters and centers. what next?, 2001.

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

[8] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[9] B. Cheung, C.L. Wang, and Kai 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 (PDPTA'99)*, 1999.

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

[11] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

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

[13] Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. In *First Symposium on Operating Systems Design and Implementation*, pages 229–241, Monterey, CA, 1994. ACM Press.

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

[15] 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.

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

[17] IEEE Computer Society Task Force on Cluster Computing. *Cluster Computing White Paper*, Mar. 2000.

[18] L. Iftode. *Home-based Shared Virtual Memory.* PhD thesis, Princeton University, August 1998.

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

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

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

[22] M. Lobosco, C. L. Amorim, and O. Loques. Java for high-performance network-based computing: a survey. *Concurrency and Computation: Practice and Experience*, (14):1–31, 2002.

[23] 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.

[24] 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.

[25] Jeremy Manson and William Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.

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

[27] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[28] Jose M. Piquer and Ivana Visconti. Indirect Reference Listing: A robust Distributed GC. In *Euro-Par '98 Parallel Processing*, September 1998.

[29] David Plainfoss and Marc Shapiro. Survey of Distributed Garbage Collection techniques. In *Proc. of Interational Workshop on Memory Management*, 1995.

[30] Roldan Pozo. Numeric and Performance Issues of Java. http://www.cs.utk.edu/~dongarra/lyon2002/Pozo.ppt.

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

[32] D. J. Scales, K. Gharachorloo, and 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)*, pages 174–185, 1996.

[33] Daniel J. Scales and Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Operating Systems Design and Implementation*, pages 101–114, 1994.

[34] Inc. Sun Microsystems. *Java Remote Method Invocation Specification*. 1999.

[35] Sun Microsystems, Inc. *The Java Hotspot Performance engine Architecture*, Oct. 1999.

[36] Ronald Veldema, Rutger F. H. Hofman, Raoul Bhoedjang, and Henri E. Bal. Runtime optimizations for a Java DSM implementation. In *Java Grande*, pages 153–162, 2001.

[37] Kaffe Java virtual machine. http://www.kaffe.org.

[38] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[39] 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.