# A Novel Adaptive Home Migration Protocol in Home-based DSM*

Weijian Fang Cho-Li Wang Wenzhang Zhu Francis C.M. Lau
Department of Computer Science
The University of Hong Kong
{wjfang+clwang+wzzhu+fcmlau}@cs.hku.hk

## Abstract

*Home migration is used to tackle the home assignment problem in home-based software distributed shared memory systems. We propose an adaptive home migration protocol to optimize the single-writer pattern which occurs frequently in distributed applications. Our approach is unique in its use of a per-object threshold which is continuously adjusted to facilitate home migration decisions. This adaptive threshold is monotonously decreasing with increased likelihood that a particular object exhibits a lasting single-writer pattern. The threshold is tuned according to the feedback of previous home migration decisions at runtime. We implement this new adaptive home migration protocol in a distributed Java Virtual Machine that supports truly parallel execution of multi-threaded Java applications on clusters. The analysis and the experiments show that our new home migration protocol demonstrates both the sensitivity to the lasting single-writer pattern and the robustness against the transient single-writer pattern. In the latter case, the protocol inhibits home migration in order to reduce the home redirection overhead.*

***Keywords***: *Cluster Computing, Distributed Shared Memory, Home-based cache coherence protocol.*

## 1. Introduction

In recent years, computer cluster has gradually been accepted as a scalable and affordable parallel computing platform by both academia and industry [5]. Message passing is one of the major programming paradigms on clusters, with which the programmers are required to write explicit code to send and receive data in order to coordinate processes in different cluster nodes.

As an alternative, software *Distributed Shared Memory* (DSM) [1] systems promise a higher programmability compared with the message passing paradigm. Software DSM provides a globally shared memory abstraction across physically distributed memory machines, and parallel programmers do not need to write explicit communication code. However, designing a high-performance software DSM system is far from trivial because many issues specific to DSM, such as programmer interface, memory consistency, and cache coherence, need to be addressed effectively.

The *memory consistency model* of a DSM system provides a formal specification of how the memory system will appear to the programmers [3]. From the viewpoint of programmers, *sequential consistency* is the most intuitive model, which requires that the memory accesses within each individual process follow the program order and writes be made atomically visible to all the processes. Though intuitive, sequential consistency suffers from poor performance due to excessive data communication among machines [13]. In order not to suffer such inefficiency, attempts were made to relax the memory order constraints as imposed by sequential consistency. *Lazy release consistency* (LRC) [11] is one of the state-of-the-art relaxed consistency models widely used in software DSM systems. In LRC, a write will not be propagated to another process until that process acquires a lock.

TreadMarks [12] adopts a *multiple-writer* cache coherence protocol to implement lazy release consistency. TreadMarks uses *twin* and *diff* techniques to support multiple processes writing on the same shared virtual memory page simultaneously due to *false sharing*. On a *write fault* to a local cached page, a copy of that page, called twin, is created. Later, the diff, which is the local updates ever performed, can be figured out by comparing the current page with the previously saved twin. The protocol is considered to be *homeless* because the diffs are saved and managed at each process.

Although TreadMarks' homeless protocol can greatly alleviate the false sharing problem, it may still suffer from heavy communication and protocol overheads [10]. In or-

* This research is supported by Hong Kong RGC Grant HKU-7030/01E and HKU Large Equipment Grant 01021001.

der to serve a page fault, the faulting process has to fetch the diffs from each process that has updated the page before the fault according to LRC, which causes multiple round-trip messages. Each diff needs to be applied once at each process that fetches that diff, which amounts to a large overhead. In addition, the diffs could consume a lot of memory, and cleaning the useless diffs may trigger a global garbage collection.

In order to address the above problems, a home-based protocol to implement LRC, which is called HLRC, was proposed [10]. In the home-based protocol, each shared coherence unit has a home to which all writes (diffs) are propagated and from which all copies are derived. It has been shown that the home-based protocol is more scalable than the homeless protocol because the home-based protocol maintains a simpler state, sends fewer messages, has a lower diff overhead, and consumes much less memory [10].

The asymmetry between the home copy and non-home copies in home-based protocols raises the *home assignment* problem. In home-based protocols, the home copy is always valid. Accesses at the home node never incur communication overhead, while accesses at non-home nodes will trigger communication with the home node. Therefore, which node to act as the home will change the coherence data communication pattern, and influence the application performance. In fact, the optimal home assignment is determined by the memory access patterns of the application. This inspires some dynamic home assignment protocols that are able to adapt to runtime memory access patterns [9, 6, 15, 7].

In DSM applications, the *single-writer* access pattern happens if the shared coherence unit is only updated by one process for a certain period. It does not prohibit the shared coherence unit from being read by multiple processes at the same time. A few research projects [9, 4, 14] have demonstrated that the single-writer pattern is common in DSM applications. In this paper, we propose a novel home migration protocol to optimize the single-writer pattern. We only target the single-writer pattern because home migration makes little difference in the multiple-writer situation so long as the home node is one of the writers.

At runtime, an object can exhibit different access patterns during its lifetime. For example, an object can be updated by multiple writers concurrently, and then by a single writer exclusively; or an object can be updated by different writers sequentially, each persisting for sometime. Since home migration has to have the effect that the other processes would be informed of the new home, improper home migrations will degrade the performance by introducing a host of messages for new home notification. Therefore, it is a challenge to exploit the single-writer property as much as possible and at the same time maintain an acceptable level of home migration overhead.

Our design of an efficient and precise home migration protocol introduces a per-object home migration threshold that is monotonously decreasing with increased likelihood that the corresponding object presents a lasting single-writer pattern. The threshold is continuously adjusted according to the feedback of previous home migration decisions. We show that this protocol is sensitive to the lasting single-writer pattern because the protocol can detect it as early as possible and the corresponding home migration is timely made. The protocol is also *robust* as it can avoid unnecessary home migrations, particularly when the applications exhibit the transient single-writer pattern.

We have implemented this home migration protocol in the global object space (GOS) of a distributed Java Virtual Machine (JVM) [7]. The distributed JVM transparently runs unmodified multi-threaded Java applications on a cluster, where Java threads are distributed to different cluster nodes to achieve parallelism. In the distributed JVM, the shared memory nature of Java threads calls for a GOS that "virtualizes" a single Java object heap spanning the entire cluster to facilitate transparent object accesses issued by distributed Java threads. The GOS is indeed a DSM service in an object-oriented system. The memory consistency of GOS follows Java consistency that resembles LRC.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 discusses the home-based protocol and the home migration concept, as well as our previous approach. Section 4 elaborates on our new home migration protocol with the adaptive threshold. Section 5 presents the performance evaluation and in-depth discussion. Section 6 concludes the paper.

## 2. Related Work

In JiaJia [9], which is a page-based DSM system, those pages that are written by only one process between two barriers are recognized by the barrier manager and their homes are migrated to the single writing process. New home notifications are piggybacked on barrier messages. Similar to our approach, JiaJia's home migration protocol only optimizes the single-writer pattern. However, JiaJia's approach relies on the barrier synchronization. It will not work if the application does not use barriers or the DSM infrastructure does not expose the barrier function. For example, in our case, the Java programmers have to implement the barrier by using more primitive synchronization operations such as lock/unlock/wait. Furthermore, since all the single-writer detection work is done centrally at the barrier manager, it may cause considerable overhead when there are a fair number of processes as well as shared pages.

JUMP [6] adopts a migrating-home protocol in that the process requiring the page becomes the new home. The new home notification is broadcast to other processes at synchro-

nization points. Although this approach results in less diffing operations because the writes probably happen at the home node, the home migration decision ignores the inherent memory access patterns of the application. If the accesses by the process at the new home do not persist, home migration will not improve the performance; instead, it could suffer from heavy home notification overhead. The worst case happens when the shared page is written by processes sequentially, which produces numerous home notification messages.

Similar to our distributed JVM, Jackal [15] allows unmodified multi-threaded Java programs to run on distributed memory parallel machines. However, Jackal follows a different approach by directly compiling Java programs into distributed native code. In terms of functionalities, Jackal's runtime system is comparable to our GOS. Jackal's runtime system enables an optimization called *lazy flushing*. The home of a shared coherence unit is fixed. However, if a unit is not shared by any other node and some node requests a copy for write access, the requesting node becomes the exclusive owner. The later reads and writes will be performed locally as if they were at the home. If other nodes want to share the unit, the current exclusive owner needs to be notified. Like JUMP's migrating-home protocol, the drawback of lazy flushing is that it ignores the application's inherent access patterns. Frequent transitions from and to an exclusive owner will cause a lot of communication overhead, and thus the number of transition are set to a maximum of five times in Jackal.

In homeless protocols that implement LRC, some page-based DSM systems [4, 14] can adapt between the single-writer protocol and the multiple-writer protocol. The single-writer protocol does not use twin and diff techniques. Instead, one process must get the ownership of a shared page before writing on it. The single-writer pattern may benefit from the single-writer protocol because diff overhead and diff accumulation can be avoided. In the context of page-based DSMs, accesses to different objects residing at the same page are mingled at the page level. So it is difficult for them to detect access patterns in applications with fine-grain sharing. In [4], the DSM system switches to the single-writer protocol when it observes that the diff overhead is larger than that of requesting the whole page. In [14], the DSM system switches to the single-writer protocol based on the approximate association between locks and the data they protect.

## 3. Home Migration Basics

In this section, we first introduce the concept of home migration, then discuss several mechanisms to notify the new home after the home migration. We will also present our previous home migration protocol.

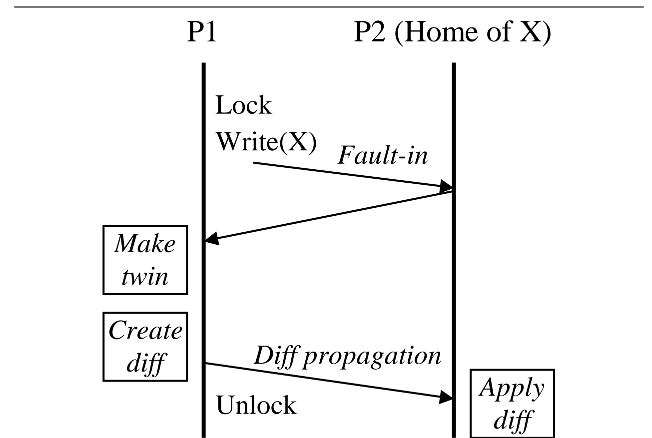



**Figure 1. Home-based Protocol for LRC with multiple-writer support**

### 3.1. Home Migration Concept

Figure 1 illustrates the home-based multiple-writer protocol that implements LRC. In the figure, `X` represents some shared coherence unit, which could be either an object or a virtual memory page. Its home is at the processor where process `P2` resides. Assuming the write on `X` performed by process `P1` causes a fault, because either the local cached copy is outdated according to LRC or `X` is not cached at all, `P1` will then fault-in the valid copy from `X`'s home, `P2`. Before `P1` could write on the newly fetched copy, it needs to create a twin, which is simply a copy of `X`. Later, when `P1` releases the lock, it eagerly creates the diff, which is the difference between the current `X` and the previously saved twin, and sends the diff to the home. And the diff will be applied to the home copy of `X` at the home.

If `P1` is the only writer of `X`, we can migrate `X`'s home from `P2` to `P1`, to avoid the communication overhead including faulting in the shared data and the diff propagation, the diff overhead including creating and applying the diff, and the memory consumption caused by the twin and the diff.

On the other hand, if both `P1` and `P2` write on `X`, it does not matter which node to become the home.

### 3.2. Home Location Notification Mechanism

We assume there is a way to determine the initial home for each unit. For example, all units are initially assigned a home node by a well known hash function. If the home of a shared coherence unit is subject to migration, we need some mechanism to inform other nodes of the new home location. There are three mechanisms: *broadcast*, *home manager*, and *forwarding pointer*.

**Broadcast** On home migration, no particular action needs to be done. The new home location is later broadcast to all the nodes at some appropriate time such as synchronization points.

**Home Manager** The most updated home location of a unit is always recorded in a designated manager node, which is known to all nodes. On home migration, the new home location is posted to the manager node. If later a process cannot locate the home of some shared unit, it can visit the manager node to find out where the current home is.

**Forwarding Pointer** On home migration, a forwarding pointer is left in the former home to point to the new home. If later a process visits the former home, it can always be redirected to the current home via the given forwarding pointer.

With the broadcast and home manager mechanisms, it is possible that the broadcast or update to the manager happens after some node tries to fault in a copy from the home node. Then the former home is already obsolete, but the new home is still not known. This situation needs to be handled carefully, for example, by waiting for sometime before repeating the fault-in again. Notice that this situation will not happen with the forwarding pointer mechanism.

Of the three mechanisms, which is superior depends on the memory access patterns of the applications and how frequent the home migration is. If after a home migration, all the other nodes need to visit the new home, then the broadcast mechanism is superior to the others because a well implemented broadcast operation should be efficient for notifying all. Otherwise, the heavyweight broadcast may cause a large overhead. The merit of the forwarding pointer mechanism is that it does not need to broadcast the new home location on home migration. However, the redirection effect may cascade where multiple home migrations may form a distributed chain of home forwarding pointers. Therefore, a process may be redirected multiple times before coming upon the current home, which is called *redirection accumulation*. It could cause significant overhead when home migration happens frequently. The manager mechanism strikes a balance between the home notification cost and the home miss cost. However, on a home miss, the process needs to visit the old home, the manager, and the new home in sequence, which is heavyweight compared with the broadcast mechanism and the forwarding pointer mechanism in the absence of redirection accumulation.

### 3.3. Home Migration with Fixed Threshold

In our previous work [7], we proposed a home migration protocol to optimize the single-writer pattern. We designed and implemented a GOS support for distributed JVM running on clusters. To match the Java memory model, the coherence unit in our GOS is a Java object. We use a home-based cache coherence protocol to implement the Java memory model that resembles LRC. We also use twin and diff techniques to support concurrent multiple writers on the same object. In our experiments, the home migration optimization gives very positive performance improvement.

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-only* on releasing a lock. Therefore, the home access faults can be trapped and returned after the access is recorded. We call the write fault at home node the *home write*, and the read fault at home node the *home read*, respectively.

At the home node, we define an object's *consecutive remote writes* to be those issued from the same remote node and not interleaved with the writes from either the home node or other remote nodes. Note that under the Java memory model, the remote writes are only reflected to the home node on synchronization points. Therefore the number of consecutive writes is the number of synchronizations during which the object is only updated by that node. At runtime, the GOS continuously monitors consecutive remote writes for each object. We also introduce a predefined home migration threshold that represents some prior knowledge on the single-writer pattern. We follow a heuristic that an object is in the single-writer pattern if the number of consecutive remote writes exceeds the home migration threshold. If the single-writer pattern is detected, when the object is requested again by the writing node, not only the object is replied, but also its home is migrated. We adopt the forwarding pointer mechanism to notify others of the new home location. When an obsolete home node is requested for an object, it simply replies with the valid home node location.

## 4. Home Migration with Adaptive Threshold

Our previous home migration protocol uses a fixed home migration threshold that represents some prior knowledge on the single-writer pattern. As soon as the number of observed consecutive remote writes is larger than the threshold, the home migration will happen. This home migration protocol is different from all the other approaches discussed in the related work section by its unique continuous runtime adaptation to the per-object access pattern. The assumption is that the access history can be used to predict the future behavior.

However, this protocol is still not satisfactory. Above all, it is difficult to decide the fixed home migration threshold. If it is too large, which implies a lazy migration policy, the home migration will be less sensitive to the single-writer pattern, thus causing unnecessary remote access overhead. If the home could be migrated earlier, more remote accesses could be transformed to local accesses. On the contrary, if the threshold is too small, it implies an eager migration policy. Although sensitive to the single-writer pattern, it will be less capable of avoiding unnecessary home migrations. If the single-writer pattern is transient in that it repeats for a very limited times, then the threads on the new home node may not perform any more accesses after the home migration. Thus the home migration decision will not gain any performance improvement, but suffer from the home redirection overhead. We observe that the transient single-writer pattern is not worthy of home migration. The home migration protocol should capitalize on the lasting single-writer pattern.

The challenge here is to choose a threshold that yields both sensitivity and robustness for the single-writer pattern. By robustness we mean taking no migration action for the transient single-writer pattern, and by sensitivity the approach responds actively to the lasting single-writer pattern. Furthermore, it is anticipated that different objects may have different access behaviors. It is more reasonable to use different thresholds on different objects.

Based on the above discussion, we propose a novel home migration protocol with an adaptive threshold. The adaptive threshold is monotonously decreasing with increased likelihood that an object presents the lasting single-writer pattern. A lower threshold will allow home migration to happen more quickly. The adaptive threshold is continuously adjusted at runtime according to the feedback of previous home migration decisions for each object.

### 4.1. Runtime Feedback

In order to measure the feedback of previous home migration decisions, the GOS will observe *exclusive home writes* and *redirected object requests* at runtime.

We define exclusive home write to be that there is no remote write between an exclusive home write and an earlier home write. Clearly, exclusive home writes reflect the single-write pattern happening at the home node. So it represents a positive feedback of previous home migration decisions.

A redirected object request reflects the home redirection effect due to home migration. It represents a negative feedback of previous home migration decisions. Redirected object requests take the redirection accumulation into account. For example, if an object request is redirected for three times before reaching the current home node, the number of redirected object requests will be considered to be three instead of one.

In addition, it is observed that exclusive home writes and redirected object requests are associated with different costs. The home redirection overhead, which is measured by redirected object requests, is equal to the round-trip time for a unit-sized message. The benefits due to home migration are from eliminated pairs of object fault-ins and diff propagations. They are measured by exclusive home writes, and are related to the object size. Therefore, we introduce the *home access coefficient* which is the overhead ratio of one eliminated pair of object fault-in and diff propagation to one home redirection. Here we mainly consider the communication overhead.

### 4.2. Formalization of Adaptive Home Migration Protocol

We formalize the idea of object home migration with adaptive threshold as follows. For each object, we have:

- $C_i$ : the number of *consecutive remote writes* since the $(i-1)th$ home migration.
- $T_i$ : the value of the adaptive *home migration threshold* since the $(i-1)th$ home migration.
- $T_{init}$ : the initial threshold, which is equal to 1.
- $R_i$ : the number of *redirected object requests* since the $(i-1)th$ home migration.
- $E_i$ : the number of *exclusive home writes* since the $(i-1)th$ home migration.
- $\alpha$ : the *home access coefficient*. Its deduction is shown in the appendix.
- $\lambda$ : the *feedback coefficient*. It is set to 1 to make the home migration threshold be sensitive enough to the feedback of previous home migrations.
- $m_{\frac{1}{2}}$ : the *half-peak length* in bytes, which is the message length required to achieve half of the asymptotic bandwidth [8].

Home migration decision is taken when the following condition is met:

$$C_i = T_i \quad (1)$$

Adaptive home migration threshold, $T_i$, is calculated by

$$T_i = \max\{(T_{i-1} + \lambda(R_i - \alpha E_i)), T_{init}\} \quad (2)$$

where

$$T_0 = T_{init} = 1 \quad (3)$$

and

$$\alpha \approx 2 + \lfloor \frac{sizeof(object)}{m_{\frac{1}{2}}} \rfloor \quad (4)$$

Equation (2) is the core of the above equations, which determines the adaptive home migration threshold. Both the positive feedback (exclusive home writes) and the negative feedback (redirected object requests) of previous home migrations will affect the current threshold. The positive feedback tends to indicate that the object presents a lasting single-writer pattern, thus decreases the threshold. Remember the threshold is monotonously decreasing with increased likelihood of the lasting single-writer pattern. While the negative feedback tends to indicate that the object presents the transient single-writer pattern, thus increases the threshold. We also take the home access coefficient into account. Whenever the home migration condition, i.e., Equation (1), is met, a home migration takes place. All these computations are done by the GOS at the home node of the object.

The initial threshold is set to 1 in order to speed up the initial data relocation if possible. It is possible that the initial data layout is not optimal with respect to the data access behavior, particularly when the writing nodes of single-writer objects are not their home nodes. A small initial home migration threshold could alleviate this situation. We rely on the adaptive threshold mechanism to adjust the threshold automatically after the initial home migration.

# 5. Performance Evaluation

Our distributed JVM implementation is based on the Kaffe JVM [2] which is an open-source JVM. The GOS is integrated with the bytecode execution engine in just-in-time compiler mode. The detailed implementation of the GOS is described in our previous paper [7]. 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. When an object is created, the creation node becomes its default home node. Exceptionally, we distribute the homes of large objects, such as array objects, among the nodes in a round-robin fashion in order to achieve load balance.

We conducted the performance evaluation on a PC cluster with 2GHz Pentium 4 processors, running Linux kernel 2.4.22, and connected by a high-performance Foundry Networks' Fast-Ethernet switch.

The home migration protocol introduced in this paper is very lightweight. All the computations related to the adaptive home migration threshold, which are mainly simple integer arithmetic, coexist with the communication. Compared with the communication overhead, their overhead is almost negligible. The GOS needs to allocate memory for the adaptive threshold, consecutive remote writes, redirected object requests, and exclusive home writes, for each shared Java object. The GOS distinguishes *distributed shared objects* among all objects at runtime, which are reachable from at least two threads residing on different cluster nodes. Only distributed shared objects will have this extra memory consumption. So the memory consumption of the adaptive home migration protocol is well contained.

## 5.1. Application Performance

We evaluate several multi-threaded Java applications on our distributed JVM. The applications include (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 of size $2048 \times 2048$ for a number of iterations; (3) Nbody, to simulate the motion of 2048 particles due 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 shortest way of visiting 12 cities and returning to the starting point with a parallel branch-and-bound algorithm. Several optimizations, including home migration, synchronized method shipping, and connectivity-based object pushing, are applied in the GOS. The detailed analysis of performance result without the adaptive home migration protocol can be found in our previous paper [7].

Figure 2 shows the execution times against the number of processors when the home migration protocol is enabled and disabled respectively. NoHM represents the situation when home migration is disabled. HM represents the situation that the adaptive threshold home migration protocol proposed in this paper is enabled.

Among all four applications, home migration improves the performance of ASP and SOR a lot. 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. The shared data are allocated to different cluster nodes in a round robin manner initially. However, their original homes are not the writing nodes. The home migration protocol automatically makes the writing node the home node to eliminate remote accesses.

Home migration has little impact on the performance of the other two applications, Nbody and TSP, due to the lack of single-writer pattern in them. This fact also indicates that our home migration protocol has little negative side effect because of its lightweight design.

In Figure 3, we further compare the performance of two home migration protocols. One is FT representing the fixed threshold home migration protocol used in our previous paper [7], where we set the fixed threshold to 2 in order to

ASP

Execution time (second)

Number of processors

NoHM

HM

SOR

Execution time (second)

Number of processors

NoHM

HM

Nbody

Execution time (second)

Number of processors

NoHM

HM

TSP

Execution time (second)

Number of processors

NoHM

HM

**Figure 2. Effect of home migration protocol**

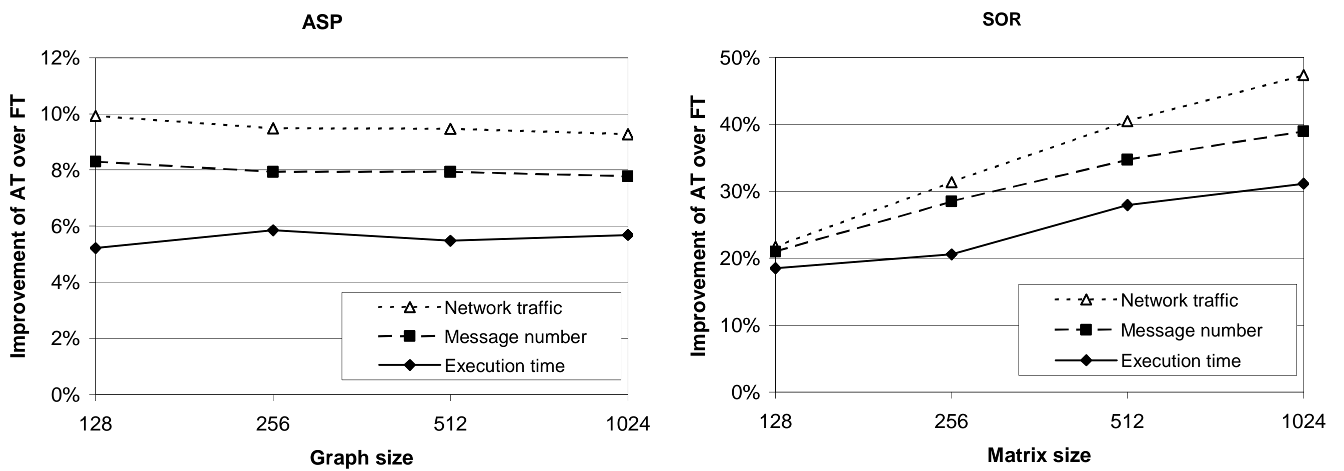



**Figure 3. Comparison of home migration protocols against problem size**

avoid the heavy home redirection overhead due to unnecessary home migration decisions. The other is AT representing the home migration protocol with adaptive threshold proposed in this paper. We present the improvement of AT over FT in terms of reduced execution time, number of messages, and network traffic 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 this experiment, both ASP and SOR run on eight cluster nodes.

As can be seen in Figure 3, AT improves the performance of ASP and SOR compared with FT. The major reason comes from that FT uses a threshold that is not suitable to these applications. In FT, we choose 2 for a balance between sensitivity and robustness of the fixed threshold home migration protocol. However, in ASP and SOR, with 2 as the threshold actually postpones the data relocation in the initial phase. In other words, with 2 as the threshold is not sensitive enough in this case. Here, a smaller threshold performs better. Notice that 2 for the threshold may be a good choice for some other applications. Instead, AT does not have such problem of choosing a appropriate threshold value. In AT, we choose the smallest threshold to speed up the initial data relocation and rely on the adaptive protocol to continuously tune the threshold according to runtime observations.

In ASP and SOR, since the performance improvement of AT over FT comes from the eliminated remote accesses due to timely home migration by AT, it should depend on the communication overhead in each iteration which is a function of the problem size. This is what we observe in SOR: the performance improvement increases when the problem size is scaled up. However, the situation in ASP is a little more complicated. In ASP, we observe that the performance improvement almost stays constant when the problem size is scaled up. This is because ASP requires $n$ iterations to solve an $n$-node graph problem and the performance improvement due to quicker home migration in AT is amortized among all iterations.

### 5.2. Sensitivity and Robustness Analysis

In order to clearly examine the performance difference between home migration protocols with different fixed thresholds and that with the adaptive threshold, we carefully design a synthetic benchmark program that predominantly presents the single-writer pattern. Figure 4 shows the source code skeleton run by each thread. In the benchmark, after a thread acquires the lock of object `lock0`, it will update a shared counter for a number of times, which we refer to as the *repetition* of the single-writer pattern. It is represented by `r` in the code. The home migration protocols try to change the home of this shared counter object to improve the performance. In order to reflect these updates to the home copy, each update is enclosed in a synchronized block. Notice after this thread releases `lock0`, it may acquire it again, or another thread may get the chance to acquire it. For example, if the repetition of single-writer pattern is 4, the actual consecutive writing times could be a multiple of 4, such as 8, 16. This happens randomly at runtime. We also embed some computation in the benchmark to make it more realistic. We measure the performance of different home migration protocols against different repetitions of single-writer pattern.

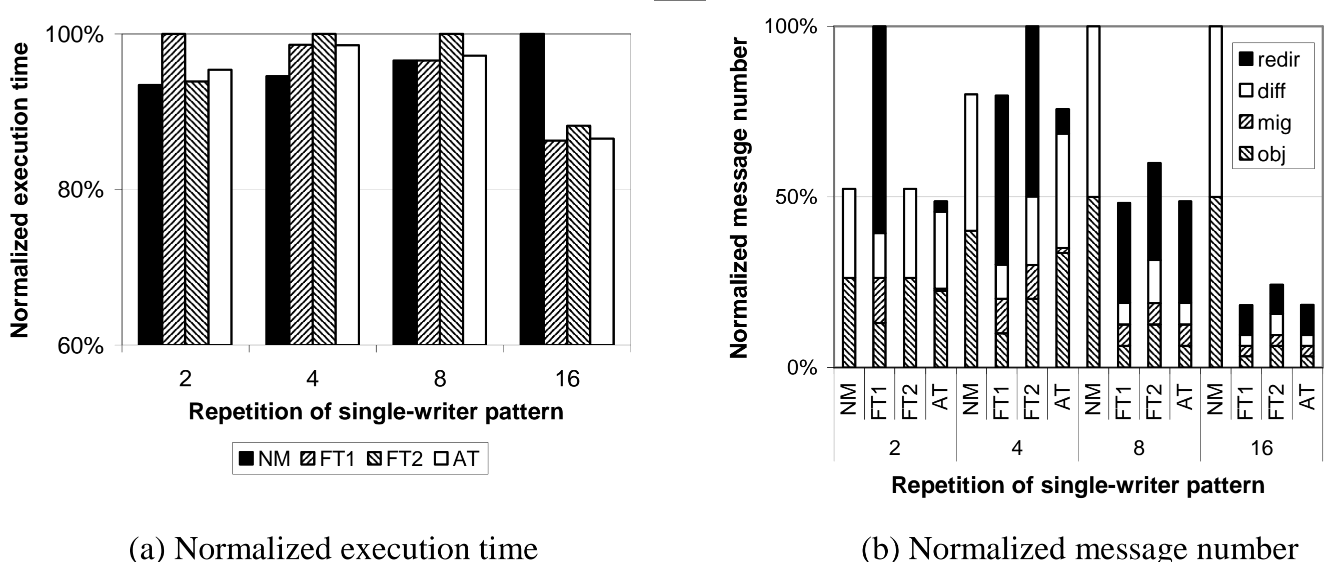
```
while (true) {
    synchronized (lock0) {
        if (counter.internal >= n) {
            break;
        }
        counter.internal++;
        for (int j=0; j<r-1; j++) {
            synchronized (lock1) {
                counter.internal++;
            }
        }
    }
    // Some simple arithmetic
    // computation goes here.
}
```

**Figure 4. Source code skeleton run by each thread**

In the experiment, we start with eight working threads all running on the nodes other than the one where the application is started. All synchronization operations are distributed ones that are sent to the node where the application is started. So all the performance differences come from the effects of different home migration protocols.

Figure 5 (a) shows the normalized execution time against different repetitions of the single-writer pattern. NM denotes no home migration. FT1 denotes home migration with a fixed threshold of 1. FT2 denotes home migration with a fixed threshold of 2. FT1 always performs home migration more eagerly than FT2. AT denotes the protocol proposed in this paper. For each repetition, the times are normalized to the largest one among them.

Figure 5 (b) shows the normalized message number against different repetitions of the single-writer pattern. For each repetition, the message numbers are normalized to the largest one among them. We further break down the messages into four categories: `obj` denotes normal object fault-in without home migration happening at the same time, `mig` denotes object fault-in with home migration, `diff` denotes




(a) Normalized execution time

(b) Normalized message number

**Figure 5. Effects of home migration protocols against repetition of single-writer pattern**

diff propagation, and `redir` denotes object home redirection. We do not consider synchronization messages because they are invariable in all cases as mentioned before.

In the message breakdown, the communication overhead without home migration includes `obj` and `diff`. They are the overheads that the home migration protocol tries to reduce. Under situations with home migration, the total number of object fault-in equals to `obj` plus `mig`, and `redir` is the negative impact of home migration.

We have several observations with Figure 5. Firstly, when the repetition of the single-writer pattern is large enough, e.g. 16, the benefit from home migration is quite obvious. As seen, 87.2% of object fault-ins and diff propagations are eliminated by FT1. In other words, remote read/write changes to home read/write. We can expect even better performance improvement due to home migration when the repetition is larger.

Secondly, when the repetition of the single-writer pattern is not large enough, the benefit from home migration may not pay off when compared to the home redirection overhead. Particularly, when the object's home and the lock's home are at the same node, as in the situation without home migration, the diff propagation can be piggybacked on synchronization messages. This explains why home migration protocols incur much less messages but still perform roughly the same as that without home migration when the repetition of the single-writer pattern is 8.

Thirdly, in all cases, FT1 is more sensitive than FT2 towards the single-writer pattern in that the message numbers of object fault-in and diff propagation in FT1 are less than those in FT2. FT1 changes more remote read/write to local read/write. When the repetition is relatively large, such as 8 and 16, AT performs as well as FT1 in this aspect. This fact confirms our claim that AT presents good sensitivity towards the lasting single-writer pattern.

Finally, when the repetition is relatively small, such as 2 or 4, i.e. the transient single-writer pattern, fixed threshold home migration protocols incur a lot of redirection overhead. This shows that fixed threshold protocols usually do not have robustness against the transient single-writer pattern, except in some individual cases, e.g., FT2 prohibits home migration when the repetition is two. As we can see, AT demonstrates better robustness than fixed threshold protocols in this aspect. AT is able to detect the transient single-writer pattern and strike a good balance between performing home migration to reduce remote accesses and prohibiting home migration to reduce the redirection overhead. When the repetition is relatively small, such as 2 or 4, AT greatly reduces the home redirection messages.

## 6. Conclusion

In this paper, we propose a home migration protocol with adaptive threshold, which can be used to optimize the single-writer access pattern in home-based DSMs. An adaptive threshold is introduced for each object and it is monotonously decreasing with increased likelihood that the object presents the lasting single-writer pattern. The threshold is continuously adjusted according to the feedback of previous home migration decisions at runtime.

The experiments confirm our claim that this protocol is intelligent in that it is robust against the transient single-writer pattern, thus avoiding unnecessary home migrations;

and it is sensitive to the lasting single-writer pattern, thus eliminating remote object accesses.

Our research shows that by exploiting abundant runtime information in the strongly-typed Java language, such as per-object access patterns, we can design more intelligent cache coherence protocols for DSM systems. However, this is rather difficult or even impossible in page-based DSMs which emulate a flat shared memory space.

In the future, we will test this protocol in more real, complicated DSM applications to further evaluate its performance. We will research on other heuristics to improve the effectiveness of home migration and to try to reduce its negative impacts.

## References


[1] Distributed Shared Memory Homepage. http://www.ics.uci.edu/~javid/dsm.html.

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

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] C. Amza, A. 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.

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

[6] B. Cheung, C. Wang, and 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*, pages 821–827, 1999.

[7] W. Fang, C.-L. Wang, and F. C. Lau. On the Design of Global Object Space for Efficient Multi-threading Java Computing on Clusters. *Parallel Computing*, 29:1563–1587, November 2003.

[8] R. Hockney. A Framework for Benchmark Performance Analysis. *Supercomputer*, IX-2(48):9–22, 92.

[9] W. Hu, W. Shi, and Z. Tang. Home Migration in Home-based Software DSMs. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

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

[11] 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.

[12] 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.

[13] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[14] 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.

[15] R. Veldema, R. F. H. Hofman, R. Bhoedjang, and H. E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Java Grande*, pages 153–162, 2001.


## Appendix

## A. Deduction of the Home Access Coefficient

Hockney [8] has proposed a model to characterize the communication time (in $\mu$s) for a point-to-point operation as follows, where the communication overhead, $t(m)$, is a linear function of the message length $m$ (in bytes).

$$t(m) = t_0 + \frac{m}{r_\infty} \tag{5}$$

$t_0$ is the *start-up time* in $\mu$s.

$r_\infty$ is the *asymptotic bandwidth* in MB/s.

Recall that the *home access coefficient* is the overhead ratio of one eliminated pair of object fault-in and diff propagation to one home redirection. Here we mainly consider the communication overhead. We assume the object size is $o$, the diff size is $d$, and the home redirection is a unit-sized message. Then we have

$$\alpha = \frac{(t_0 + \frac{o}{r_\infty}) + (t_0 + \frac{d}{r_\infty})}{t_0 + \frac{1}{r_\infty}} \tag{6}$$

$$= \frac{2t_0 r_\infty + (o + d)}{t_0 r_\infty + 1} \tag{7}$$

The half-peak length, denoted by $m_{\frac{1}{2}}$ bytes, is the message length required to achieve half of the asymptotic bandwidth. It can be derived using the relationship

$$m_{\frac{1}{2}} = t_0 r_\infty \tag{8}$$

Based on $m_{\frac{1}{2}} \gg 1$ and $o > d$, we derive Equation (4), which we restate here:

$$\alpha \approx 2 + \lfloor \frac{o}{m_{\frac{1}{2}}} \rfloor \tag{9}$$