Migrating-Home Protocol for Software Distributed Shared Memory

BENNY WANG-LEUNG CHEUNG, CHO-LI WANG AND FRANCIS CHI-MOON LAU

Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong.

The efficiency of Software Distributed Shared Memory (DSM) is often limited by the excessive amount of network communication in maintaining the memory consistency of the system. Two of the most popular software solutions to reduce redundant data traffic are the proper use of relaxed memory consistency models and coherence protocols. In this paper, we propose the migrating-home protocol for a relaxed memory consistency model, scope consistency. The protocol allows the processor storing the most up-to-date copy of a page to change among processors, so as to adapt better to the memory access patterns of DSM applications. Our proposals have been implemented in a DSM system running on a 16-node Pentium III 450MHz PC cluster. We analyzed not only the execution time of the benchmark programs, but we also adopted a new approach to analyze the communication and page fault patterns. It is shown that our DSM system reduces the amount of network communication and handles page faults more efficiently. They provide concrete evidence in explaining the substantial performance improvement obtained by our system.

Keywords: clustering computing, distributed shared memory, memory consistency model, coherence protocol, home-based protocol, migrating-home protocol, JUMP.
1. INTRODUCTION

Software Distributed Shared Memory (DSM) offers the abstraction of a globally shared memory across physically distributed memory applications by a software layer (Figure 1). It exempts programmers from handling explicit data communication in writing parallel programs, making it an attractive parallel programming paradigm on a cluster of PCs or workstations. However, early DSM systems often perform poorly, as they tend to communicate excessive amount of data among machines through the network to maintain memory consistency.

There are several software approaches to alleviate this network bottleneck. One of them relies on the use of memory consistency models, which are rules specifying how the memory system will appear to the programmers. Instead of using the strict sequential consistency (SC) [1] as adopted by the first DSM system, IVY [2], many later DSM systems make use of the relaxed models. For example, the Munin DSM system [3] made use of the eager release consistency model (ERC), while TreadMarks [4] employed lazy release consistency (LRC) [5], which is weaker than ERC. Midway [6] went further and used the entry consistency model (EC) [7], which is even weaker and more efficient than LRC. Unfortunately, the programming interface associated with EC is not easy to use, as it requires explicit binding between synchronization variables (locks) and shared memory variables. Scope consistency (ScC) [8] has thus been proposed, aiming at achieving good programmability and performance.
Apart from the use of relaxed memory consistency models, we can improve DSM performance through an efficient coherence protocol, which defines rules to assist the implementation of the memory consistency model. For instance, the home-based protocol is more efficient than the homeless protocol [9] in implementing ScC.

In this paper, we introduce a coherence protocol called migrating-home protocol, in which the location storing the most up-to-date copy of each page in shared memory can be changed among processors. Through the use of this protocol, better adaptation to the memory access patterns of DSM applications can be achieved. This is demonstrated through a reduction in the execution time of DSM applications. We also apply a new approach to analyze the communication and page fault patterns on a number of DSM programs. It is shown that under the migrating-home protocol, the amount of data traffic in the network can be reduced. Page faults can be handled more efficiently as well, as more page faults can be served locally without communicating with other processors.

For the rest of this paper, Section 2 overviews ScC and the 2 existing coherence protocols for its implementation. Section 3 discusses the migrating-home protocol in detail. Section 4 describes the implementation of the protocol and the testing environment. The
performance results and analysis are provided in Section 5. The new approach of page fault analysis will be addressed in Section 6. This is followed by the related work in Section 7. Finally, we conclude this paper in Section 8.

2. BACKGROUND

In this section, we shall briefly overview scope consistency (ScC), together with a discussion of the home-based and homeless protocols for implementing ScC to place the ground of our research.

2.1 Scope Consistency (ScC)

Scope consistency (ScC) achieves high performance and good programmability through the use of the scope concept, which reduces the amount of data propagation among processors, and fits naturally to the lock mechanism.

In ScC, a scope is a limited view of memory with respect to which memory references are performed. Updates made within a scope are guaranteed to be visible only within the same scope. For example, in Figure 2, all critical sections guarded by the same lock comprise a scope. The locks in a program thus determine the scopes implicitly, making the scope concept easy to understand. In addition, barriers define a global scope, which covers the entire program. Thus at a barrier, all the updates on the shared objects made by every processor will be propagated to the others.

A scope is said to be opened at an acquire operation (lock acquire or leaving a barrier),
and is closed at a release (lock release or approaching a barrier). Having all these concepts, ScC is defined as follows:

*When processor Q opens a scope that is previously closed by another processor P, the updates made within the same scope in P is propagated to Q.*

This means the updates made outside the same scope will not be propagated at the time of the acquire. An example demonstrating this fact is shown in Figure 2.

In this program, only the update of z by P0 is propagated to P1 at the acquire of the lock, since only z is updated within the same scope. In comparison, the updates of all x, y and z are propagated under LRC. Therefore ScC reduces the amount of data communication within the cluster, and becomes more efficient than LRC. But the programming interface is exactly the
same as that for release consistency. There is no explicit binding as needed by EC. Hence ScC retains good programmability as well.

2.2 Home-Based vs. Homeless Coherence Protocols

The memory consistency models discussed above define the policies on the behavior of the shared memory abstraction as viewed by the users. However, we still need to define the mechanisms, i.e. the data structures and algorithms used in implementing the models. This leads to the existence of coherence protocols. We shall discuss two different categories of coherence protocols, namely the home-based and the homeless protocols.

In page-based DSM, a machine may access a shared memory area not in its main memory. This results in a page fault. To serve the page fault, a copy of the memory page causing the fault with the most up-to-date contents must be brought in from the remote, so that the access can continue without error. There are different ways to get the page from remote. In *home-based protocols* such as the one adopted by JIAJIA V1.1 [10], each page in the shared memory space is assigned a processor to store the most up-to-date copy of a page. This processor, fixed at application initialization time, is known as the *home* of the page. All the updates on a page will be propagated to the home processor in the form of diffs [4] when the processor making the update performs a synchronization operation. Later, when a processor accesses the page, it generates a page fault and the page request will be forwarded to the home of the page. The home processor replies by sending a clean copy of the page that causes the fault. Figure 3(a) shows how the home-based protocol serves a page fault.

On the other hand, a homeless protocol such as the one used in TreadMarks does not possess the concept of home. No processor is responsible for holding the most up-to-date
copy of a page. In order to serve a page fault, the faulting processor has to contact all the peers that have recently updated that page. All these processors handle the request by sending the updates made on the page in the form of diffs to the faulting processor. The faulting processor then applies these updates in order as specified by the timestamps attached, so that the clean copy of the page can be obtained. Figure 3(b) shows the events for serving a page fault in a homeless protocol.

Research [11] shows that the home-based protocol is more efficient than the homeless protocol by sending fewer messages in the network. In particular, it reduces the communication overhead in serving a page fault by requesting only one processor for a copy of the page. Moreover, home-based protocols are easier to implement than homeless protocols, since there is no need to implement timestamps in home-based protocols to deal with the order of the updates, as homeless protocols do.
(a) **Under the Homeless Protocol**

![Diagram](image)

Figure 3. Comparison of the homeless protocol and the home-based protocol. (a) In the homeless protocol, the page fault request in \( P_3 \) has to be served by communicating with multiple processors (\( P_1 \) and \( P_2 \)). (b) In the home-based protocol, the page updates are propagated to the home processor of the page (\( P_0 \)) at the release operation of \( P_1 \) and \( P_2 \). The following page fault in \( P_3 \) is served only by communicating with the home processor \( P_0 \).
3. THE MIGRATING-HOME PROTOCOL

This section studies the proposed migrating-home protocol [12], in which the location of the home processor is changeable among the processors during application execution.

3.1 The Migrating-Home Concept

The fixed-home concept introduced in the home-based protocol, though helps in improving the performance over its homeless counterpart, may not adapt well to the access patterns of applications. In particular, if the home processor is not involved in accessing the page, a message containing the page updates (diffs in our example) is always sent from the processor updating the page to the home processor at synchronization time. In return, an acknowledgement known as diff grant is sent from the home of the page to the processor making the update. This idea is further explained using a simple example in Figure 4(a).
Figure 4. (a) An example for the home-based protocol, (b) Under the migrating-home concept, the twin, diff and diff grant are saved.

This pair of messages can be saved if we grant the home to the current page requester (like $P_0$ in our example) when the original home ($P_2$) serves the page fault. This is done by setting a flag in the message when $P_2$ sends back the copy of the page to $P_0$. We say that the home is migrated from $P_2$ to $P_0$. This means that the copy of page $x$ obtained by $P_0$ will be regarded as the master copy of the page and its contents are most up-to-date. Therefore, the diff and diff grant messages need not be sent when $P_0$ issues the lock release operation after the write, as shown in Figure 4(b). Moreover, the twin operation (which is a memory copy operation) needs not be performed at $P_0$ either, since $P_0$ is the home and contains the master copy at the time of the write. It can thus directly write to the copy of the page it gets.

This example brings about the migrating-home concept, which can be described by the
following statement:

*When a processor requests a page from its current home processor, the requester can become the new home.*

Figure 5(a) shows this concept again graphically. In the diagram, the circular token denotes the home of page $X$. It is moved from the original home processor $P_2$ to the new home processor $P_0$ when the latter makes a page request.

However, other processors may not be aware of this home migration. For example, $P_1$ may request page $x$ after the home change. It may get the outdated copy if it requests the page from the previous home $P_2$. To avoid this, $P_0$ needs to send a *migration notice* to all the other nodes at a release operation (such as lock release or reaching a barrier), as shown in Figure 5(b). The migration notice tells the processors that the home of a page has been changed. As we shall see next, the migration notices are short and they replace many of the lengthy diffs.

3.2 Migration Notices

Figure 5. The migrating-home concept: (a) Home migration, (b) At a release.
A migration notice is used to inform other processors about the home change of a page. The only piece of information needed to be carried in each migration notice is the page ID to specify which page has its home changed. Thus the migration notice is very short, and minimizes the amount of data communication through the network.

In DSM applications, it is usual that more than one page will have their home migrated to a processor within a critical section. Sending the migration notices of each page one by one can cause an excessive amount of time spent in the communication startup. As the sending of migration notices is delayed until a release operation takes place at the new home, we can make use of the short-length feature of migration notices by concatenating multiple notices together to form a single message.

3.3 Dealing with False Sharing

In the previous discussion of the migrating-home protocol and migration notices, we have not yet dealt with the case when false sharing occurs. That is, when two or more processors write to different locations of the same page before synchronization takes place. We have encountered such a situation in the example shown in Figure 3. One of the most popular ways in dealing with false sharing is the diffing technique [4]. In the migrating-home protocol, diffing is also used in solving the false sharing problem, with certain rules added to maintain memory consistency of the DSM system.

To show how the migrating-home protocol deals with false sharing, we look at an example in Figure 6, where P0 and P1 both ask for a page X before any one of them synchronizes. In this false-sharing situation, the migrating-home protocol still grants the first
requester $P_0$ as the new home. The late requester $P_1$ will receive a copy of the page from the previous home $P_2$ and the ID of the new home processor, so that $P_1$ can send the diff to the new home when it synchronizes. The copy of the page obtained by the late requester $P_1$ is still clean, as long as $P_1$ does not access the variables updated by $P_0$ (such as $X_0$) before $P_0$ synchronizes. In fact, $P_1$ is not supposed to access $X_0$, since the behavior is undefined under the definition of LRC or ScC.

Later, when $P_1$ approaches the synchronization point, it calculates and sends the diff to the new home. The diff grant is then replied as an acknowledgment. Finally, the new home $P_0$ synchronizes and sends the migration notice to the other processors. The previous home $P_2$ replies the migration notice by telling $P_0$ that $P_1$ has a copy of $X$ too. This helps $P_0$ to determine whether its copy of page $X$ has got the most updated contents. If it has, the page can be migrated to other processors upon further request.

![Diagram](image_url)  

Figure 6. Illustrating how the migrating-home protocol deals with false sharing.
It looks as if the reply of the migration notice made by $P_2$ is redundant since $P_1$ has sent the diff to $P_0$. However, $P_0$ may synchronize before $P_1$. Thus the reply is needed to guarantee that the new home $P_0$ knows whether the page content is clean.

To summarize the migrating-home protocol with false sharing support, the home of a page can be migrated to the page requester if and only if:

- The processor being requested the page is the current home of the page, and
- The contents of the entire page are already clean (i.e., most up-to-date).

### 4. IMPLEMENTATION AND TESTING

The migrating-home protocol discussed in Section 3 has been implemented on the JUMP software DSM system. A number of benchmark applications are ported to test its performance. In this section, we discuss the implementation of JUMP, as well as the environment used in our testing in detail.

#### 4.1 Implementation of JUMP

JUMP, which stands for JIAJIA Using Migrating-Home Protocol, is a software DSM system modified from JIAJIA V1.1 with the implementation of the migrating-home protocol. As JUMP is modified from the JIAJIA DSM system, it inherits most of the implementation features in JIAJIA. Like JIAJIA, JUMP is a user-level, page-based DSM system appearing in the form of a runtime library. It is built on the UNIX operating system, and makes use of the UNIX virtual memory manager and system calls to achieve the shared memory abstraction. It
adopts the scope consistency as the memory model, and false sharing is handled by the differencing technique. BSD sockets provided by the UNIX operating system are used for communication among processors. JUMP also shares the same system structure as JIAJIA, which includes a communication subsystem, a memory management subsystem and a synchronization subsystem. The only difference between JIAJIA and JUMP is that JUMP employs the migrating-home protocol, which is embedded within the memory subsystem. The synchronization subsystem of JUMP is also modified to interact with the memory subsystem, such that locks and barriers are provided as synchronization facilities.

4.2 Performance Testing

We conducted a test to see the performance of JUMP by executing a suite of 6 benchmark applications. The performance of JUMP with JIAJIA V1.1 is compared to find out if the migrating-home protocol can improve the DSM performance over the home-based protocol.

The test was performed on a commodity cluster built using 16 Pentium III 450MHz PCs. They are connected using Fast Ethernet through a 24-port IBM 100-based switch. Each machine has 128MB main memory, and runs a copy of the Linux Kernel 2.2.14 as the operating system.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arg.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The six benchmark applications, as shown in Table 1, are described as follows:

(1) **MM**: MM is an application which multiplies two \(n \times n\) matrices (namely \(Q\) and \(S\)) using \(p\) processors, storing the results in a third resulting matrix \(R\). All the matrix elements are initialized as shared memory. However, at the initialization, while the values of the elements in \(S\) are assigned in a row major fashion, the values of the elements in \(Q\) are initialized in a column-major fashion. This means that the two matrices are in fact \(Q^T\) and \(S\). Many implementations of MM divide the resulting matrix \(R\) into \(p\) parts, and each processor calculates one part of \(R\). Our implementation takes a different approach. The source matrices \(Q\) and \(S\) are divided into \(p\) parts, and each processor only accesses one part of \(Q\) and \(S\) to calculate a subtotal value of each element in the resulting matrix \(R\). The subtotal values calculated from each of the \(p\) processors are stored temporarily in local memory, and are summed up together to form the resulting matrix \(R\) at the final stage.

(2) **ME**: ME performs the merge sort on \(n\) integers using \(p\) processors. The \(n\) integers, appeared as \(p\) sorted arrays, are held by the \(p\) processors at the starting phase. At each stage,
2 arrays held by adjacent processors are merged together as one sorted array by one of the processors. Hence the merging is done in \((\log p)\) stages. Notice that in this program, using more processors will slow down the execution, since an extra stage of merging will be introduced when the number of processors doubles.

(3) **RX:** RX performs radix sort on \(n\) 32-bit integers generated by \(p\) processors. As each 32-bit number to be sorted can be expressed using 8 hexadecimal digits, the sorting is divided into 8 stages, with one of the digits (4 bits) being sorted at each stage. Each processor uses \(p\) buckets allocated in the shared memory to sort the numbers, and distribute them between stages.

(4) **LU:** LU is a program which factorizes an \(n \times n\) matrix \(M\) using \(p\) processors by a technique known as LU-factorization. Each element in the matrix is initialized as a double-precision floating point number, and the whole matrix is allocated in the shared memory space. The factorization process is divided into \(n\) stages. In each stage \(s\), the value of the elements in \(M\) will be updated according to the formulae:

\[
M_{sj} \leftarrow \frac{M_{sj}}{M_{ss}} \quad \text{for} \ (j > s), \ \text{and}
\]

\[
M_{ij} \leftarrow M_{ij} - M_{sj} \times M_{is} \quad \text{for} \ (i > s \ \text{and} \ j > s)
\]

Each processor is responsible to compute the intermediate values for certain rows of elements in the matrix \(M\). The values will be written back to the matrix. After all \(n\) stages, the matrix \(M\) will be completely factorized. Finally, the processor \(P0\) is responsible for verifying if the result of \(M\) is correct.

(5) **BK:** BK performs bucket sort on \(n\) integers. Each of the \(p\) processors involved in the execution handles 256 buckets for sorting and data distribution. After data distribution takes
place, each bucket holds the numbers within a certain range. The numbers in each bucket are then sorted by bubble sort. This gives the sorted result of the numbers when the buckets are accessed with a particular sequence.

(6) **SOR:** The full name of SOR is known as the red-black successive over-relaxation application. Our program is performed on two $n \times n$ matrices, one known as the red matrix, and the other called the black matrix. At each stage of the program, the values of the elements in each of the two matrices are updated according to the values of the elements in the other matrix. This routine is performed for 20 iterations.

5. PERFORMANCE RESULTS

This section addresses the results of the performance testing as described in the previous section. We shall compare the performance between the migrating-home protocol in JUMP with the home-based protocol in JIAJIA V1.1 by analyzing the execution time, number of messages, and volume of data communicated.

5.1 Execution Time Analysis

The execution time of the benchmark suite under JUMP and JIAJIA V1.1 on the Linux cluster is shown in Figure 7 using logarithmic-10 scale. We also compare their execution time by using a ratio, expressed as the execution time of an application under JIAJIA V1.1 over the execution time of the same application under JUMP, with the same $n$ and $p$ values. A ratio over 1 means JUMP has an improvement in performance over JIAJIA.

From the graphs, JUMP performs better than JIAJIA V1.1 for most of the applications.
Since the protocol is the only difference between the two systems, this shows that in general, the migrating-home protocol in JUMP is more efficient than its home-based protocol in JIAJIA. However, the degree of performance improvement varies with applications. This is discussed in more detail as follows.

1. **MM**: JUMP has a small performance improvement over JIAJIA V1.1 in running the MM application. Most of the data points only exhibit a small throughput improvement of about 5-10%, which is relatively small when compared to other applications. This is because in MM, each processor spends a long time computing the local matrix. This intense computation dominates the total execution time of the MM program, especially when the number of processors is small and the problem size is large. However, the local matrix computation is not affected much by the protocol used. Therefore, the heavy computation component reduces the effect of protocol used on the performance of the matrix multiplication program.

   However, for small problems with the $n/p$ ratio smaller than 16, the MM application runs even slower under the migrating-home protocol. This is because for small problems, each processor writes to one or two pages in each critical section. At the release, the migration notices of these pages need to be sent to all the other processors, but there are only one or two migration notices in each of these messages. The migrating-home protocol is unable to take advantage of the feature of concatenating multiple migration notices together in a single message. Hence the extra communication startup cost becomes considerably high.
Figure 7. Graphs showing the comparison of the execution time of each benchmark application under JUMP and JIAJIA V1.1 in terms of the ratio (time under JIAJIA V1.1 / time under JUMP).
(2) **ME:** Unlike MM, the migrating-home protocol has a much better improvement in the performance of the merge sort program, in which JUMP exhibits a 33-234% performance improvement over JIAJIA. This is because the ME program is more communication-intensive than MM. Moreover, the performance improvement of ME under JUMP tends to increase with more processors and a larger problem size.

We also notice that ME runs more slowly under both JIAJIA and JUMP with more processors. This is because ME makes one more stage of merging when the number of machines is doubled, resulting in extra execution time.

(3) **RX:** JUMP exhibits a small performance improvement not exceeding 16.3% over JIAJIA V1.1 in the execution of the RX program. For most problem sizes under the 2 and 4-processor case, JUMP is able to execute RX faster than JIAJIA. However, when the number of processors increases to 8 or 16, JUMP suffers from a small performance degradation of about 5-10% in general.

(4) **LU:** The LU factorization benchmark shows the largest fluctuation in the performance improvement of JUMP over JIAJIA. For small problem sizes ($n = 64$ and 128), JUMP suffers from degradation in performance when compared with JIAJIA. However, when the problem size $n$ increases to 256 or more, the performance improvement of JUMP becomes positive and is increasing drastically. At $n = 1024$, JUMP even completes the LU application more than 10 times faster than JIAJIA with 8 or 16 processors, which is the largest improvement encountered in the testing. This observation is in fact caused by a drastic reduction in the amount of communication made within the network. Further communication analysis in the next part shows that the migrating-home protocol in JUMP is able to reduce more than 95% of the network communication in LU for large problems.
(5) **BK:** For the bucket sort program, the migrating-home protocol in JUMP exhibits a modest performance improvement over JIAJIA V1.1 not exceeding 30%. Except for the data point with $n = 256K$ and $p = 16$, where JUMP experiences a very small performance degradation, bucket sort benefits the migrating-home protocol with all problem size and processor combinations tested. The reason for the performance degradation at the smallest $n/p$ ratio case is similar to that in MM: Not enough migration notices are available to concatenate as a single message. The extra overhead in sending the migration notices cannot be hidden.

(6) **SOR:** The SOR application experiences the best performance gain under JUMP in general among the 6 applications used. Most of the data points in SOR perform twice as fast when JUMP replaces JIAJIA V1.1. Like the general trend shown in most applications, a higher performance ratio is obtained when the problem size is increased.

### 5.2 Communication Overhead

To understand how the migrating-home protocol improves the efficiency in the execution of DSM applications, we shall investigate the amount of communication made by JUMP during the execution of the benchmark programs. We compare both the number of messages and the communication volume (that is, the total amount of bytes transmitted) within the cluster by JUMP and JIAJIA V1.1. The results are expressed as message ratio and communication volume ratio of JUMP over JIAJIA for each benchmark, and are shown as graphs in Figure 8 and Figure 9 respectively.

From the data, it can be seen that for MM and RX, JUMP sends more distinct messages than JIAJIA V1.1 for the same application over the same problem size. This is shown by a message ratio larger than 1. The only exceptions are the RX benchmark for sorting 4M
integers with 2 or 4 processors in use, at which JUMP sends fewer messages. This means that
the migrating-home protocol generates more messages to be transmitted in the cluster than
the home-based protocol in these 2 benchmark applications. The observation is not surprising
due to the broadcast nature of migration notices in the migrating-home protocol.

For ME and LU, JUMP also sends more messages than JIAJIA V1.1 for small problem
sizes. However, this is no longer true when the problem size grows larger. For example,
JUMP is able to send no more than 43.9% of the total number of messages needed to be sent
by JIAJIA in the execution of ME with \( n = 4M \). For LU with \( n = 1024 \), JUMP even needs to
send less than 16.2% of messages sent by JIAJIA to complete the task. The underlying reason
is that although there are many short migration notices needed to be sent under JUMP, the
migrating-home protocol adapts to the memory access patterns of ME and LU so well that the
save in the page requests and page grant messages exceeds the extra migration notices
generated.

Although JUMP may send more or less messages than JIAJIA depending on the
application, we observe a general trend that the message ratio of JUMP over JIAJIA
decreases with the increase in problem size. This can be explained by the fact that large
problems use more pages in shared memory. Hence there is a larger potential for larger
problems to take advantage of the short migration notices by concatenating them together as
a single message, as discussed in Section 3.

Next, we also compare the communication volume generated in the network by JUMP
and JIAJIA. We observe that JUMP sends fewer bytes than JIAJIA V1.1 over most of the
applications. This matches the aim of the migrating-home protocol in reducing the data
volume communicating within the network. The short migration notices replace the lengthy
diffs generated under the home-based protocol in most cases, and hence the protocol succeeds in transmitting fewer bytes within the cluster.

We also find that the reduction in communication volume produced by JUMP over JIAJIA V1.1 becomes more and more significant when the problem size increases. Using the ME application with 16 processors as an example, JUMP sends about 33.3% of the communication volume transmitted by JIAJIA when the problem size equals 2M, but this value drops to 28.5% when 4M integers are sorted. In the extreme case, JUMP can save 97% or even more bytes that JIAJIA needs to send in the execution of LU with $n = 1024$. This observation, together with the trend that JUMP tends to send less messages than JIAJIA for large problems, suggest that the migrating-home protocol favors the execution of large applications.

We have observed that the migrating-home protocol in JUMP may send more messages than the home-based protocol in JIAJIA, depending on the application and the problem size, but the communication volume generated by JUMP is less than JIAJIA. These two factors have a contrasting effect on the actual time spent in the communication. This is because the time needed for a message to be sent from a processor to another through the network can be decomposed into 2 parts: a constant startup cost, and a transmission cost directly proportional to the number of bytes sent. From the execution time of the benchmarks, it can be concluded that the reduction in the amount of bytes communicated has a more dominating effect than the increase in the number of messages sent. This phenomenon becomes more significant in systems with low-latency communication support. Under such systems, the migrating-home protocol is able to perform more efficiently than the home-based protocol.
Figure 8. Comparing the number of messages sent under JUMP and JIAJIA V1.1, expressed by the ratio of the no. of messages sent under JUMP to that under JIAJIA V1.1 for each application.
Figure 9. Comparing the volume of data communicated under JUMP and JIAJIA V1.1, expressed by the ratio of the data volume under JUMP to that under JIAJIA V1.1 for each application.
6. PAGE FAULT ANALYSIS

In analyzing the performance of a DSM system, apart from the execution time and communication analysis, the number of page faults encountered, together with the way they are dealt with by the system can also provide useful evidence. This section takes a look at the different types of page fault that can occur under a DSM system.

6.1 Different Types of Page Faults

In JUMP or JIAJIA, a page fault arises when a shared memory page being accessed (read or written) is not within the local memory or is marked dirty. A page fault can also occur when an attempt is made to write on a page which is write-protected. Page faults with different causes are treated differently by the DSM system. We classify them into 3 main categories PF1, PF2 and PF3 as follows.

**PF1**: A PF1 fault is a page fault that can be served by the local processor, which is the home of the page. It arises only due to the violation in access permission, that is, when we write on a page which has been write-protected. The solution is just to disable the write protection on the page. No remote processors have to be contacted to serve the fault, and no messages have to be sent through the network. An example is shown in Figure 10.

**PF2**: A PF2 fault is a page fault that can also be served by the local processor. However, unlike PF1, the local processor is not the home of the page. The processor is still able to serve the fault because it has cached a clean copy of the page in its local memory. Hence there is no need to contact the remote home processor for getting a copy of the page. However, the local processor is not the home of the page, no matter the migrating-home protocol or the home-
Based protocol is used. Therefore, it is inevitable that the faulting processor has to send a diff message to the remote home processor when it issues a release operation, as shown in Figure 11. The sending of the diff message takes time, and hence PF2 faults have to take longer time to serve than PF1 faults in an indirect sense.

\[ P_0 (\text{Home of } X) \]
\[ P_1 \]
\[ \begin{array}{c}
\text{Twin} \\
\text{Clean page in cache}
\end{array} \]

Figure 10. Serving the PF1 page fault. No remote processors are involved.

\[ P_0 \]
\[ P_1 (\text{Home of } X) \]
\[ \begin{array}{c}
\text{Acquire (L0)} \\
W(x0)2
\end{array} \]
\[ \begin{array}{c}
\text{Release (L0)} \\
\text{No Diff sent !!}
\end{array} \]

Figure 11. Serving the PF2 page fault. \( P_0 \) has cached a copy of page \( X \).

\textbf{PF3:} The third type of page fault is one which has to be served by the remote processor. A page request message has to be sent to the current home processor of the page, and upon receiving the request, the home processor replies with a copy of the required page. If the migrating-home protocol is used, information about the home migration is also appended in the replying message. It can be sure that PF3 page faults take the longest time to be served, since it involves a pair of messages communicating between two processors. However, there
are still four possibilities which can happen after the page fault. The characteristics of the 4 sub-categories of PF3 faults are summarized in Table 2, and are discussed in detail below.

**PF3A:** If a processor generates a page fault and requests a page, and the home is migrated to the faulting processor during the page fault is being served, there is no need for the new home to further send out diff messages at the time a release operation is issued even it writes on this page. This situation can only happen in the migrating-home protocol (Figure 12(a)).

<table>
<thead>
<tr>
<th>Page Fault Type</th>
<th>PF3</th>
<th>PF3B</th>
<th>PF3C</th>
<th>PF3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occur on Read</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Occur on Write</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Occur in Home-Based Protocol</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Occur in Migrating-Home Protocol</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Requests Page from Remote</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Involves Home Migration</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Involves Diff and Diff Grant</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

Note: (1) The Diff produced under PF3D faults are empty diffs, with no information if the page is not written before a release operation occurs.
Figure 12. Two possibilities of the PF3 page fault. (a) If the home is migrated to the page requester when serving the fault, no diff is sent. (b) Home is not granted to page requester, causing the diff and diff grant at release time.
Figure 13. The other two possibilities of the PF3 page fault. (a) A read fault does not cause a diff to be sent at synchronization under the home-based protocol. (b) Under the migrating-home protocol, if the home is not granted to the faulting processor, an empty diff has to be sent at the release time.

**PF3B:** If home migration is not migrated when the write fault is being served, as the faulting processor is not the home of the page, it has to send a diff message containing the updates of the page to the remote home processor of the page when it issues a release. The scenario is shown in Figure 12(b). Notice that **PF3B** can happen in both the migrating-home protocol and the home-based protocol, but for the migrating-home protocol, it only happens when false sharing exists, and the home is not migrated when the page is granted from the remote.
Otherwise, the faulting processor should have been granted the home as the page fault is served.

**PF3C and PF3D:** The third and fourth possibility arises when home migration does not take place, and the page fault is a read fault. Under the home-based protocol, no diff is sent since there is no update on the page. This is shown in Figure 13(a). Under the migrating-home protocol, we treat a read fault in the same way as a write fault. This can eliminate the need to send the page fault request and page grant messages in case the page will be written later. However, if the processor getting the page is not granted the home of the page, and it does not write the page before synchronization, then at synchronization time, an *empty diff* will be sent to the home processor of the page. An empty diff does not contain any page update information, it only signals the home processor of a page that the empty diff sender has finished accessing a copy of the page. The scenario is depicted in Figure 13(b). Since the cost of this type of page faults is different under the two protocols, in order to differentiate them, we shall call the remote read fault under the home-based protocol PF3C, while the remote read fault under the migrating-home protocol is named PF3D.

Among the three types of page faults, it is easy to spot that the PF1 fault bears the least cost in terms of the number of messages sent and also the time taken to serve the fault. Both PF2 and PF3 take longer time to serve, but it is difficult to compare the cost of the two. The reason is three-folded. Firstly, although PF2 faults can be served locally, it has to generate a diff message (and hence a diff grant too) in the network. In comparison, a PF3 fault, regardless of the sub-category it belongs to, always generate at least a pair of messages (the page request and page grant) when it is being served. Secondly, for PF2 faults, the copy of the page cached may reside in the disk rather than in the main memory due to the virtual
memory management of the underlying operating system. In such case, PF2 faults will take a much longer time to get served. Finally, there are four possibilities that can occur for a PF3 fault, as shown in Figures 12 and 13. This further complicates the overhead comparison with PF2 faults. However, one thing can be sure is that if we can change some of the PF2 or PF3 faults to PF1 faults, the time needed to deal with the page faults and its related activities will be reduced. The performance in executing the DSM applications can hence be improved. We shall investigate if the migrating-home protocol is capable of reducing the cost associated by the page faults.

6.2 Page Fault Breakdown

Figure 14 shows the comparison of the number of page faults generated in every application under JUMP and JIAJIA. To further analyze the number of each type of page faults generated under the two protocols, Table 3 lists the breakdown of various types of page faults for each benchmark program under JUMP and JIAJIA using 16 processors.

From the graph, we see that in most cases, JUMP generates more page faults than JIAJIA in executing the same benchmark application over the same problem size. When 2 processors are used, JUMP causes a maximum of 29.4% more page faults than JIAJIA V1.1. As $p$ increases to 4, this difference narrows down as JUMP only generates at most 16.2% more page faults than JIAJIA. This value further decreases to about 6.9% and 5.5% when 8 and 16 processors are employed respectively. The trend indicates that the number of page faults generated in JIAJIA increases faster than that generated in JUMP as more processors are used in executing the DSM applications.
Figure 14. Comparing the total number of page faults generated under JUMP and JIAJIA, which is expressed by the ratio (page faults made under JUMP / page faults made under JIAJIA V1.1).
Table 3. Page fault breakdown for each application under JUMP and JIAJIA V1.1 using 16 processors.

<table>
<thead>
<tr>
<th>Appl.</th>
<th>Size</th>
<th>Page Fault Breakdown under JUMP</th>
<th>Page Fault Breakdown under JIAJIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>n</td>
<td>PF1</td>
<td>PF2</td>
</tr>
<tr>
<td>MM</td>
<td>64</td>
<td>75</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>256</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>964</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>3856</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>15424</td>
<td>0</td>
</tr>
<tr>
<td>ME</td>
<td>256K</td>
<td>1040</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>512K</td>
<td>2080</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>4160</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2M</td>
<td>8320</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4M</td>
<td>16640</td>
<td>0</td>
</tr>
<tr>
<td>RX</td>
<td>256K</td>
<td>2398</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>512K</td>
<td>4200</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>7826</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2M</td>
<td>15070</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4M</td>
<td>29533</td>
<td>0</td>
</tr>
<tr>
<td>LU</td>
<td>64</td>
<td>308</td>
<td>1570</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>2471</td>
<td>4709</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>20558</td>
<td>10701</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>130849</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>916354</td>
<td>0</td>
</tr>
<tr>
<td>BK</td>
<td>256K</td>
<td>8505</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>512K</td>
<td>8794</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>9375</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2M</td>
<td>10520</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4M</td>
<td>10204</td>
<td>0</td>
</tr>
<tr>
<td>SOR</td>
<td>512</td>
<td>9164</td>
<td>568</td>
</tr>
<tr>
<td></td>
<td>768</td>
<td>21828</td>
<td>579</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>39396</td>
<td>557</td>
</tr>
<tr>
<td></td>
<td>1536</td>
<td>90212</td>
<td>548</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>161533</td>
<td>531</td>
</tr>
</tbody>
</table>

To explain the extra page faults introduced by the migrating-home protocol, we look at the example in Figures 15 and 16. Suppose \(P0\) is the original home of page \(X\). Under the
migrating-home protocol used in JUMP, after page \(X\) is migrated to some other processors for more than once, \(P0\) wants to read the same page as well. It then generates another page fault, and the home is migrated back. This scenario is shown in Figure 15. However, under the home-based protocol with the same access pattern, the home stays unchanged in \(P0\) throughout the program execution and \(P0\) can read without generating the extra page fault, as shown in Figure 16. When more processors are involved, the chance for the original home processor to access the page again is in general smaller due to two possible reasons. First, the page can be accessed in turn by more processors, making the same processor to access the same page less frequently, as in the case of MM. Second, with the increase of processors, there is a higher chance for the page to be initially allocated to a processor which does not access the page. This is true in both JUMP and JIAJIA, where the home allocation of a page takes an arbitrary round-robin fashion. If the home processor of a page does not access the page, the saving of a page fault in favor of the home-based protocol cannot happen. The RX program experiences this type of access. In either situation, less extra page faults are invoked when more processors are used.
Figure 15. Demonstrating the extra page fault generated by the migrating-home protocol. Under the migrating-home protocol, $P_0$ causes a remote page fault when it reads $x_0$. 
When we study the page fault breakdown statistics in Table 3, it is found that the migrating-home protocol in JUMP is able to generate less PF2 faults by converting them into PF1 ones. This conclusion is derived from the data, as for most applications, the sum of the PF1 and PF2 faults generated by JIAJIA in an application is equal or very close to the number of PF1 faults generated by JUMP for the same application. As a PF2 fault causes a diff later in the execution while a PF1 fault does not, the migrating-home protocol is able to perform more efficiently. Furthermore, the number of PF2 page faults accounts for a considerable proportion of the total number of page faults occurred in JIAJIA. For example, when 16 processors are used, 26-88% of all the page faults generated in JIAJIA are PF2 faults.
faults. By converting them into a less time-costly form, JUMP reduces the need to send lengthy diffs through the network and hence improves the overall performance of the applications significantly.

Another observation is that although both JUMP and JIAJIA generate remote PF3 faults during application execution, the page faults bear different natures. The PF3 faults introduced by JIAJIA are either PF3B or PF3C faults. The PF3B fault requires a diff to be made and sent to the remote home processor, and can become the potential performance bottleneck of the DSM, particularly when the diff is lengthy. However, for the case of JUMP, most of the remote PF3 faults made are mainly PF3A ones. For each of these page faults, the home of the requested page has been migrated to the faulting processor. This implies that no diffs will be generated, and less time can be spent for serving the page faults, thus improving DSM application performance.

In summary, although the migrating-home protocol in JUMP may produce more page faults than the home-based protocol used by JIAJIA during program execution, most of the page faults need shorter time to be served. Fewer diffs are generated as well. The time saved in serving the page faults compensates the extra page faults generated. Hence the migrating-home protocol is able to improve the overall efficiency of the DSM system.

7. RELATED WORK

TreadMarks [4] makes use of the homeless protocol discussed in Section 2 to implement the lazy release consistency model. The popularity of the system has triggered various studies on its design and implementation. Coherence protocol has then become one of the major
fields of research in DSM. Iftode [13] has studied the homeless protocol in TreadMarks. He criticized that a page requester may have to gather diffs from different processors upon a page fault can be a potential performance bottleneck.

Hence he proposed a new cache coherence protocol known as the Automatic Update Release Consistency (AURC) for implementing lazy release consistency on the SHRIMP multicomputer [14] by making use of the automatic update hardware mechanism provided by the machine. AURC can gain a better performance than the homeless protocol. In AURC, a home processor is selected to store the master copy of every page. The automatic update mappings provided by SHRIMP are then set such that writes to the other copies of the page are automatically propagated to the home processor immediately. The home copy of the page is hence always kept up-to-date, while the other copies will be updated by fetching the home copy on demand. This scheme enhances the performance in two ways: First, the update is done by hardware and does not cause any software overhead. Second, communication is needed between the page requester and the home processor of the page only.

Although AURC is dedicated to the SHRIMP multicomputer, which possesses a specialized automatic update hardware, the idea of a home for each shared memory page inspires later research efforts. The most remarkable one is the home-based lazy release protocol (HLRC) proposed by Zhou and Iftode [11], which is a protocol implementing lazy release consistency using the home-based approach, with no specialized hardware support needed. The underlying concept adopted by HLRC has been dealt with when we discussed the home-based protocol in Section 2. Zhou and Iftode also showed in their paper that the home-based protocol is more efficient than the homeless approach.

The idea of HLRC was also adopted by the first few versions of JIAJIA (before V2.1). However, instead of implementing the lazy release consistency model, JIAJIA used the
home-based approach to realize the scope consistency model. This results in a more efficient implementation, since scope consistency produces less data propagation than lazy release consistency at lock acquire or barrier synchronization.

Later versions of JIAJIA also adopt the concept of home migration of shared memory pages, in order to achieve better adaptation to the memory access patterns of DSM applications. One of them is the *home migration* protocol implemented by JIAJIA V2.1 [15]. This protocol shares the same objective with the migrating-home protocol proposed in this research, but its implementation is rather different. Instead of migrating the home of a page eagerly in serving a page fault, JIAJIA V2.1 migrates the home of a page if and only if that page has been written by only one processor between two barriers. The home is migrated to that only writer when the second barrier synchronization takes place. This method attempts to embed the migration notices within the barrier message, so that no extra messages are incurred. However, the main drawback is that this stricter rule for home migration does not apply when two or more processors write to the same page within two barriers. It does not work on applications that use locks either.

We also tested the performance of JIAJIA V2.1 and compared it with that of JUMP, using the same testing environment mentioned in Section 5. The results are summarized in [16], and it shows that JUMP outperforms JIAJIA V2.1 in all applications except RX, in which JUMP loses by 2.0-22.9%. Thus the migrating-home protocol in JUMP is more efficient than the home migration protocol in JIAJIA V2.1 in general.

8. CONCLUSIONS
This paper has proposed the migrating-home protocol, in which the home of each page in shared memory can be migrated to another processor when it accesses the page. It reduces the amount of data communicating among machines through the network to maintain memory consistency. We have demonstrated that the proposed enhancement is practical, as the migrating-home protocol has been implemented into the real-life DSM systems, namely JUMP. Performance testing and analysis show that the migrating-home protocol adapts better to the memory access patterns of most DSM applications than the traditional home-based protocol. This leads to an improvement in the performance of JUMP. Further analysis shows that the migrating-home protocol is able to convert some of the remote page faults to local ones, hence reducing the time in serving page faults. Moreover, as the home of each page can be migrated to a processor which accesses the page, some of the page updates need not to be sent to the home processor, hence reducing data traffic.

REFERENCES


of the 2nd Symposium on Operating Systems Design and Implementation (OSDI’96), pages 75-88, October 1996.


