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 relaxed memory consistency models and traffic-thrifty coherence protocols. In this paper, we propose the *migrating-home protocol* for a relaxed memory consistency model, the scope consistency model. The protocol allows the processor storing the most up-to-date copy of a page to change from one processor to another, so as to adapt to the memory access patterns of DSM applications. The new protocol has 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 also the communication and page fault patterns via a new analysis approach. It is shown that our DSM system reduces the amount of network communication and handles page faults more efficiently. The benchmark results provide concrete evidence for 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 computing nodes (Figure 1). It exempts programmers from handling explicit data communication in parallel programs, making it an attractive parallel programming paradigm on a cluster of PCs or workstations. Early DSM systems, however, often failed to perform satisfactorily, as they tended to communicate excessive amount of data among machines because of the need to maintain memory consistency.

[*] The content of this research has been presented in the 1999 and 2000 *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, USA.

[+] This research is supported in part by the Hong Kong RGC Grant HKU-7030/01E and an AOE grant from the HKSAR Government.

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 in terms of its semantics. The first DSM systems, IVY [2], adopted the strict *sequential consistency* (SC) [1]. Because of SC's inefficiencies, later systems turned to the *relaxed* models. For example, the Munin DSM system [3] used the *eager release consistency* (ERC) model; TreadMarks [4] employed *lazy release consistency* (LRC) [5], which is weaker (i.e., more relaxed) than ERC. Midway [6] used the *entry consistency* (EC) model [7], which is even weaker and appears to be 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 good programmability and high performance.

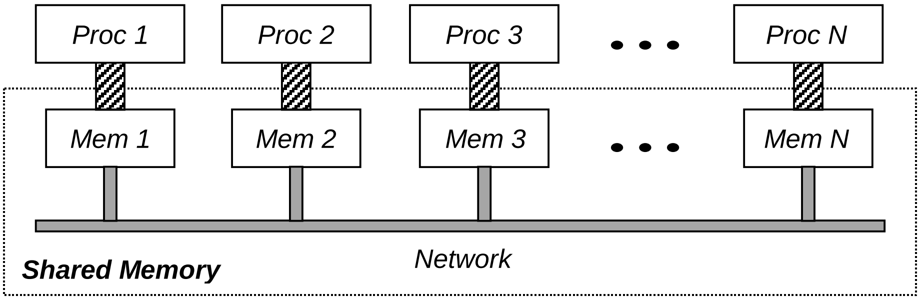


Figure 1. The distributed shared memory abstraction: Each processor sees a shared address space, delineated by the dashed outline, rather than a collection of distributed address spaces.

Apart from the use of relaxed memory consistency models, we can improve DSM performance through an efficient *coherence protocol*, which defines rules for a correct implementation of the underlying 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 the *migrating-home protocol*. Using this protocol, the location storing the most up-to-date copy of each page in the shared memory can be changed from one processor to another depending on the need. The protocol allows better adaptation to the memory access patterns of DSM applications to 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 of a number of DSM programs. It is shown that with the use of 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 communication with other processors.

For the rest of this paper, Section 2 overviews ScC and the two existing coherence protocols and aspects of their 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 their analysis are provided in Section 5. Section 6 discusses the new approach of page fault analysis and the results. Section 7 presents the related work. 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.

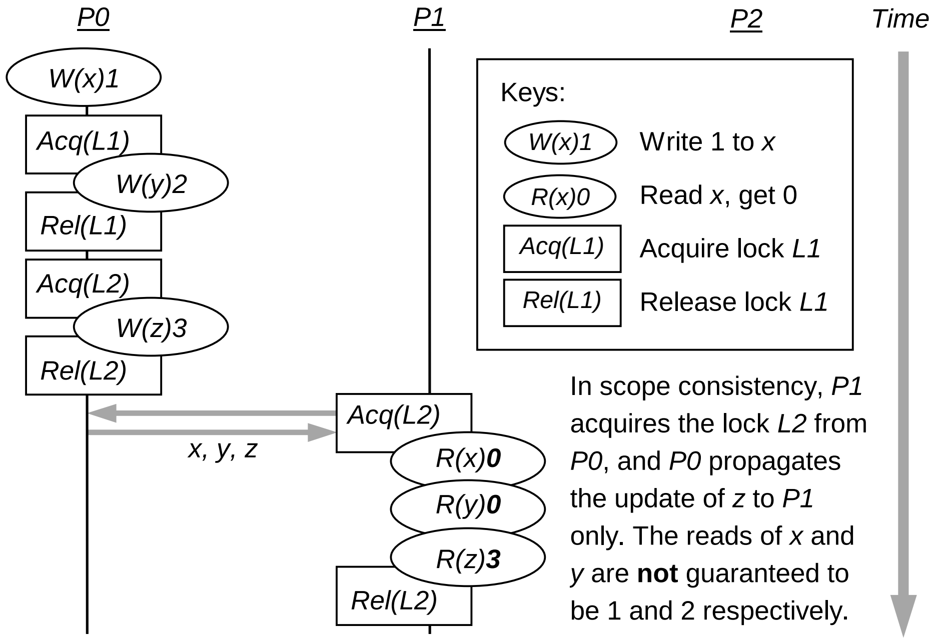


Figure 2. An example illustrating the scope consistency (ScC).

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.

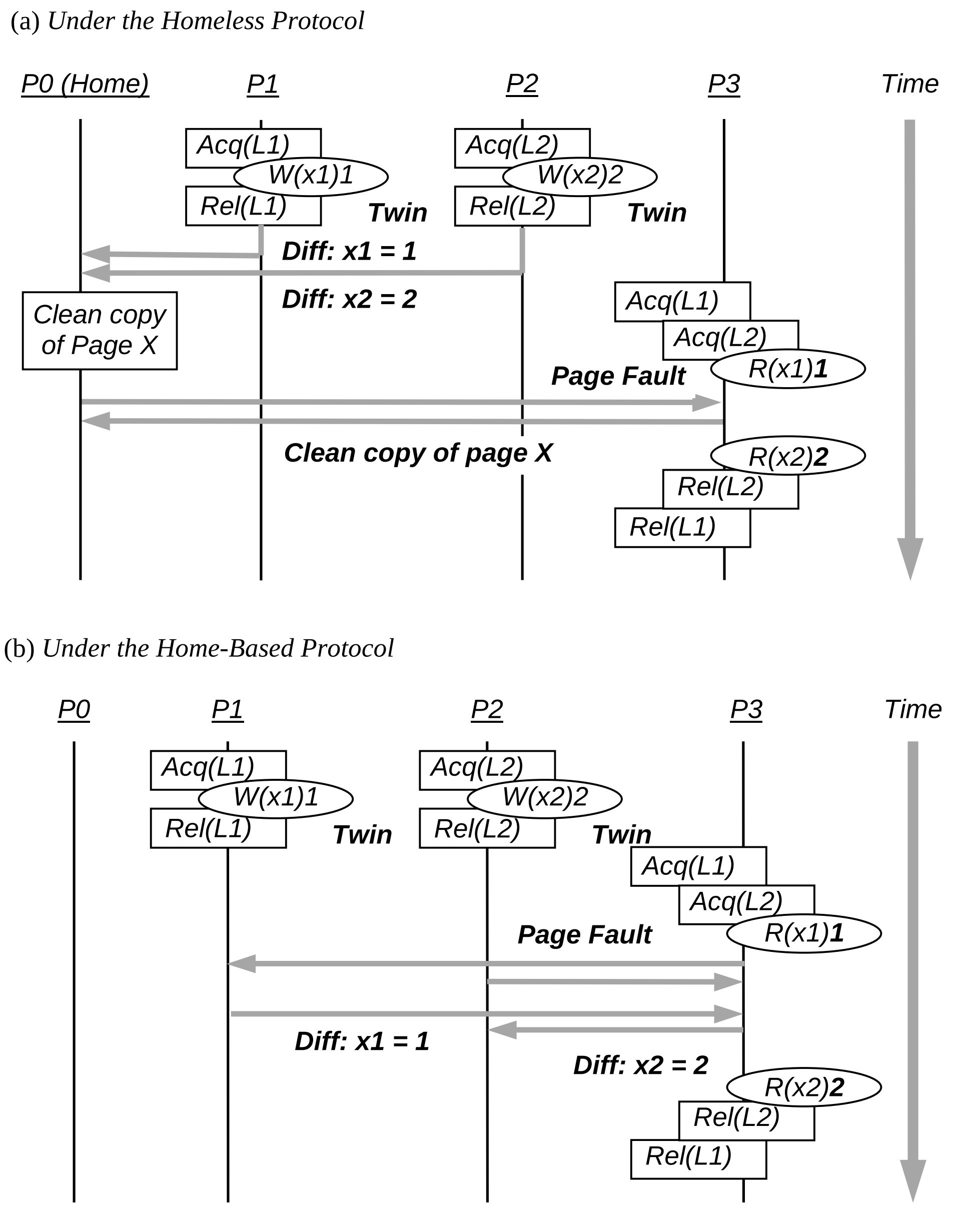
2.2 Home-Based vs. Homeless Coherence Protocols

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*

(b) *Under the Home-Based Protocol*

Figure 3. Comparison of the homeless protocol and the home-based protocol. (a) In the homeless protocol, the page fault request in *P3* has to be served by communicating with multiple processors (*P1* and *P2*). (b) In the home-based protocol, the page updates are propagated to the home processor of the page (*P0*) at the release operation of *P1* and *P2*. The following page fault in *P3* is served only by communicating with the home processor *P0*.

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).

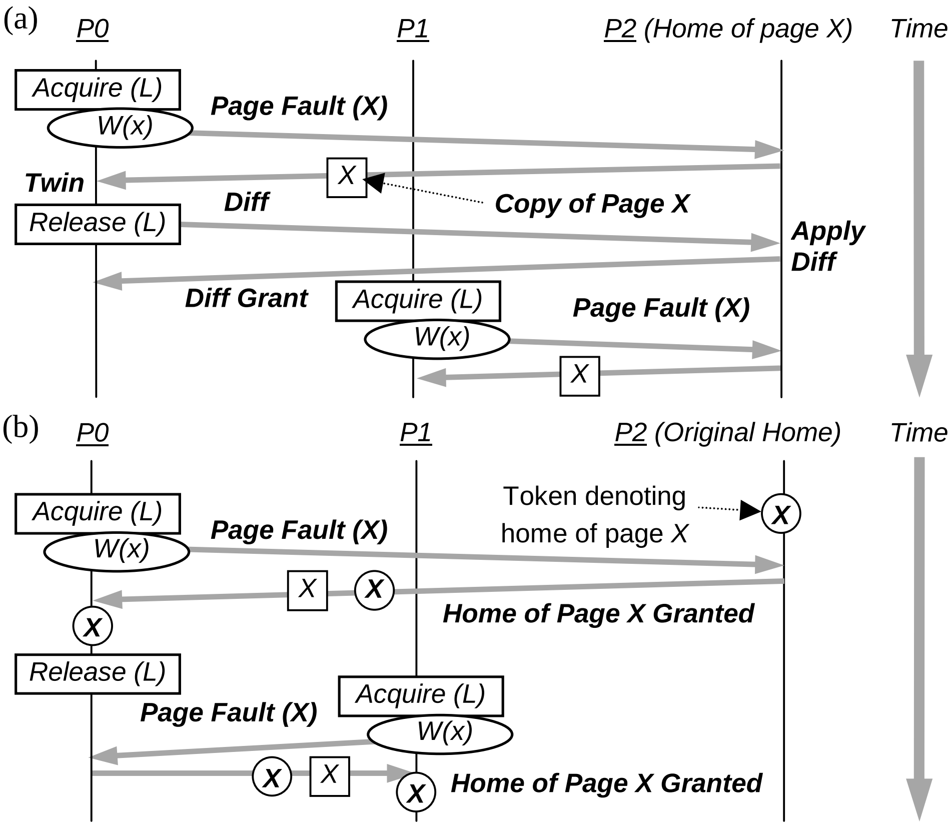


Figure 4. (a) An example for the home-based protocol, (b) Under the migrating-home concept, the twin, diff and diff grant are saved.

This pair of messages can be saved if we grant the home to the current page requester (like *P0* in our example) when the original home (*P2*) serves the page fault. This is done by setting a flag in the message when *P2* sends back the copy of the page to *P0*. We say that the home is *migrated* from *P2* to *P0*. This means that the copy of page *X* obtained by *P0* 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 *P0* 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 *P0* either, since *P0* 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 *P2* to the new home processor *P0* when the latter makes a page request.

However, other processors may not be aware of this home migration. For example, *P1* may request page *X* after the home change. It may get the outdated copy if it requests the page from the previous home *P2*. To avoid this, *P0* 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.

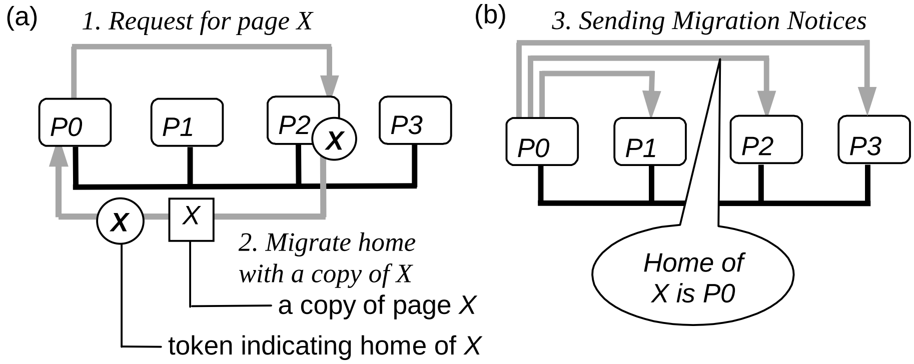

(a)
1. Request for page X
P0
P1
P2
X
P3
X
X
2. Migrate home with a copy of X
a copy of page X
token indicating home of X
(b)
3. Sending Migration Notices
P0
P1
P2
P3
Home of X is P0

Figure 5. The migrating-home concept: (a) When page *X* is requested, the home is migrated together with the transmission of the page copy. (b) At lock release, the new home sends migration notices to all peers, stating the new home location of page *X*.

3.2 Migration Notices

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 *P0* as the new home. The late requester *P1* will receive a copy of the page from the previous home *P2* and the ID of the new home processor, so that *P1* can send the diff to the new home when it synchronizes. The copy of the page obtained by the late requester *P1* is still clean, as long as *P1* does not access the variables updated by *P0* (such as *x0*) before *P0* synchronizes. In fact, *P1* is not supposed to access *x0*, since the behavior is undefined under the definition of LRC or ScC.

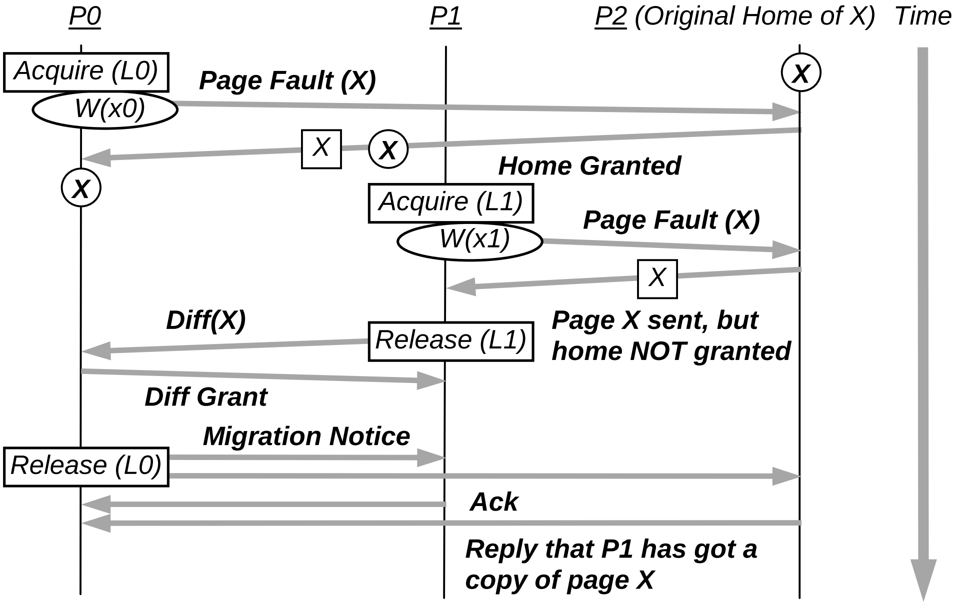


Figure 6. Illustrating how the migrating-home protocol deals with false sharing. Here *x0* and *x1* are two variables in page *X*.

Later, when *P1* 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 *P0* synchronizes and sends the migration notice to the other processors. The previous home *P2* replies the migration notice by telling *P0* that *P1* has a copy of *X* too. This helps *P0* 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.

It looks as if the reply of the migration notice made by *P2* is redundant since *P1* has sent the diff to *P0*. However, *P0* may synchronize before *P1*. Thus the reply is needed to guarantee that the new home *P0* 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 **J**IAJIA **U**sing **M**igrating-Home **P**rotocol, is a software DSM system modified from JIAJIA V1.1 [10] with the implementation of the migrating-home protocol. 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 diffing technique. The same system structure of JUMP consists of a *communication subsystem*, a *memory management subsystem* and a *synchronization subsystem*. The communication subsystem deals with data communication among nodes in the system, while the memory management subsystem handles memory consistency issues. The migrating-home protocol implemented by JUMP forms a unique and crucial component in the memory subsystem. The synchronization subsystem of JUMP works closely with the other two subsystems, and it provides locks and barriers as synchronization facilities.

4.2 Performance Testing

We evaluated the performance of JUMP by executing a suite of six benchmark applications. The performance of JUMP and 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 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 1. Description of the six benchmark applications.

Name	Arg.	Description
MM	*n*, *p*	Matrix Multiplication of two $n \times n$ matrices using *p* processors
ME	*n*, *p*	Merge Sort on *n* integers in *p* sorted lists using *p* processors
RX	*n*, *p*	Radix Sort of *n* 32-bit integers using *p* processors
LU	*n*, *p*	LU-factorization of an $n \times n$ matrix using *p* processors, with the results verified for correctness
BK	*n*, *p*	Bucket Sort of *n* integers using *p* processors with $256 \times p$ buckets
SOR	*n*, *p*	Red-Black Successive Over Relaxation on two $n \times n$ matrices using *p* processors, with the main loop iterating for 20 times

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. 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, two 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 ($1 \le s \le n$), the value of the elements in M will be updated according to the formulae:

$M_{sj} \leftarrow M_{sj} / M_{ss}$ for ($j > s$), and
$M_{ij} \leftarrow M_{ij} - M_{sj} \times M_{is}$ for ($i > s$ 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* × *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 reports the results of the performance testing as described in the previous section. We shall compare the performance between and 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 the execution time under the two systems by dividing the execution time of an application under JIAJIA V1.1 by the execution time of the same application under JUMP, with the same *n* and *p* values. We call the result obtained *performance ratio*, and is presented in the line chart in Figure 7. A performance 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 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.

(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. Also, in each stage of the merging, half of the shared memory pages are accessed by their home node under migrating-home protocol, hence reducing the communication and page fault overhead. 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 with p = 2 or 4, 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. This can be accounted for by the extra number of page faults generated, as described in Section 6.2.

(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 traffic made within the network during program execution. 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.

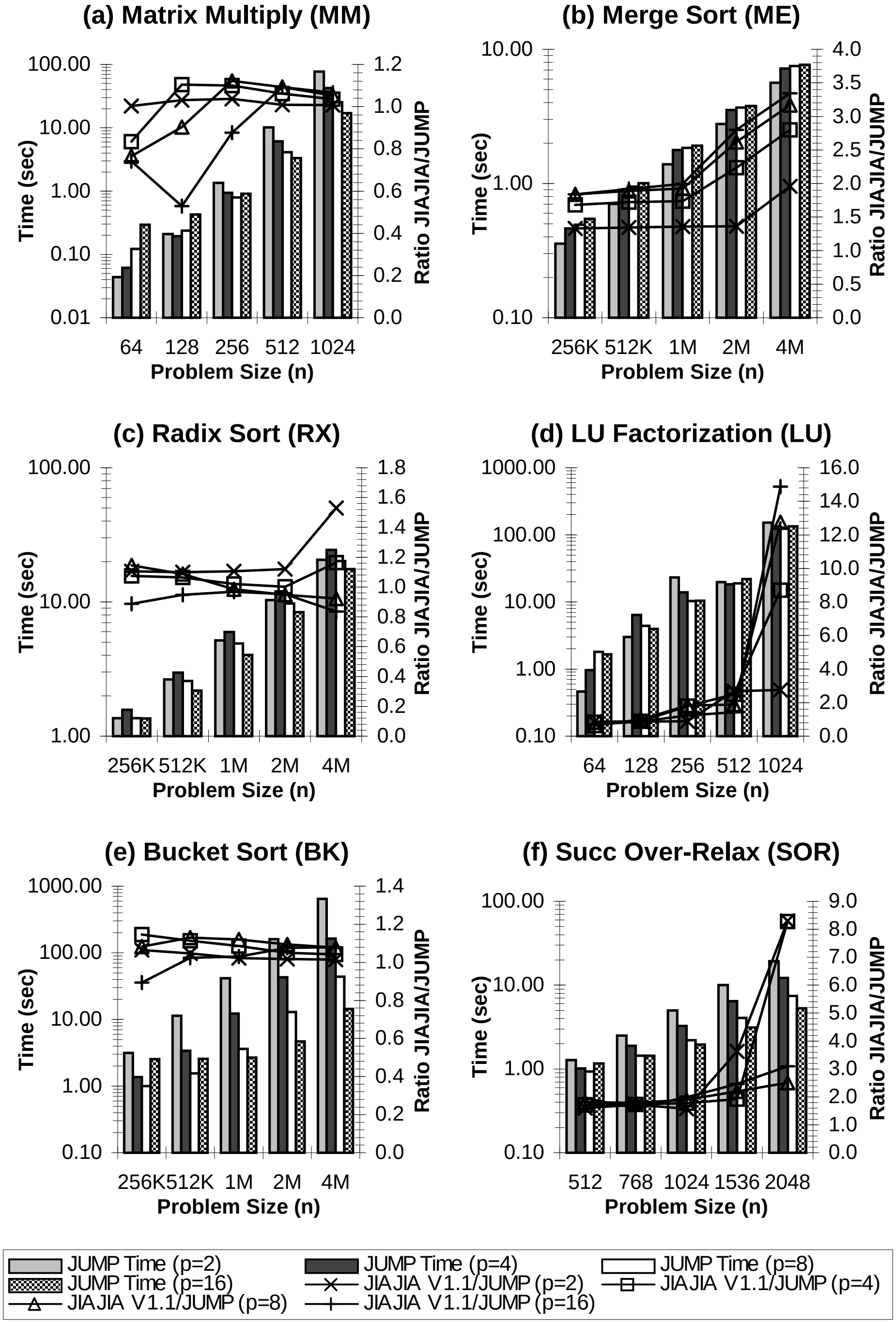


Figure 7. Graphs showing the execution time of applications under JUMP (bar chart), and also the comparison of the execution time of each benchmark application under JUMP and JIAJIA V1.1 (line chart) in terms of the performance ratio (time under JIAJIA V1.1 / time under JUMP).

(6) ***SOR:*** The SOR application experiences the best performance gain under JUMP in general among the six applications used. Most of the tests in SOR perform twice as fast when JUMP replaces JIAJIA V1.1. This is because most shared memory pages in SOR are accessed solely by a single process throughout the execution. Under the migrating-home protocol, the accessing processor must have cached such pages, and it must become the home of those pages after the first access, making further accesses local and thus reducing communication and page fault overhead. Moreover, 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. This means the migrating-home protocol generates more messages to be transmitted in the cluster than the home-based protocol in these two benchmark applications. The observation is not surprising due to the broadcast nature of migration notices in the migrating-home protocol. The only exception is the RX benchmark with n = 4M and p = 2 or 4, at which JUMP sends fewer messages.

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 sends 1/6 of the total number of messages sent by JIAJIA to complete the task. There are two main reasons. First, migration notices for multiple pages can be concatenated as a single message under such a large problem size, so that the extra number of messages generated can be kept to a minimum. And second, the migrating-home protocol adapts well to the memory access patterns of ME and LU, leading to a massive drop in the number of remote page requests and page grant messages generated.

We also 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.

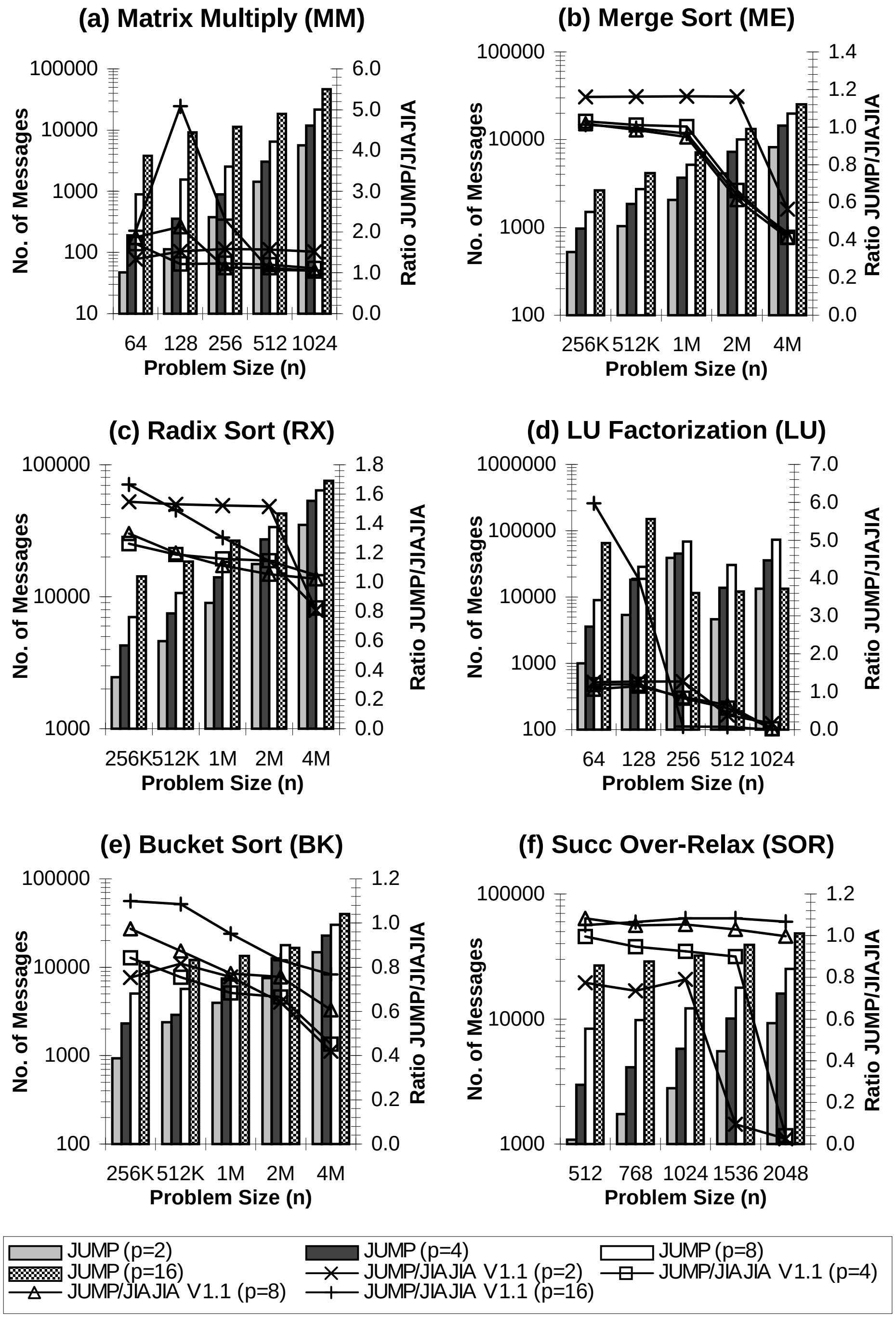


Figure 8. Graphs showing the number of messages sent under JUMP (bar chart), and also the comparison of the number of messages sent under JUMP and JIAJIA V1.1 (line chart), expressed by the ratio (number of messages sent under JUMP / number of messages sent under JIAJIA V1.1).

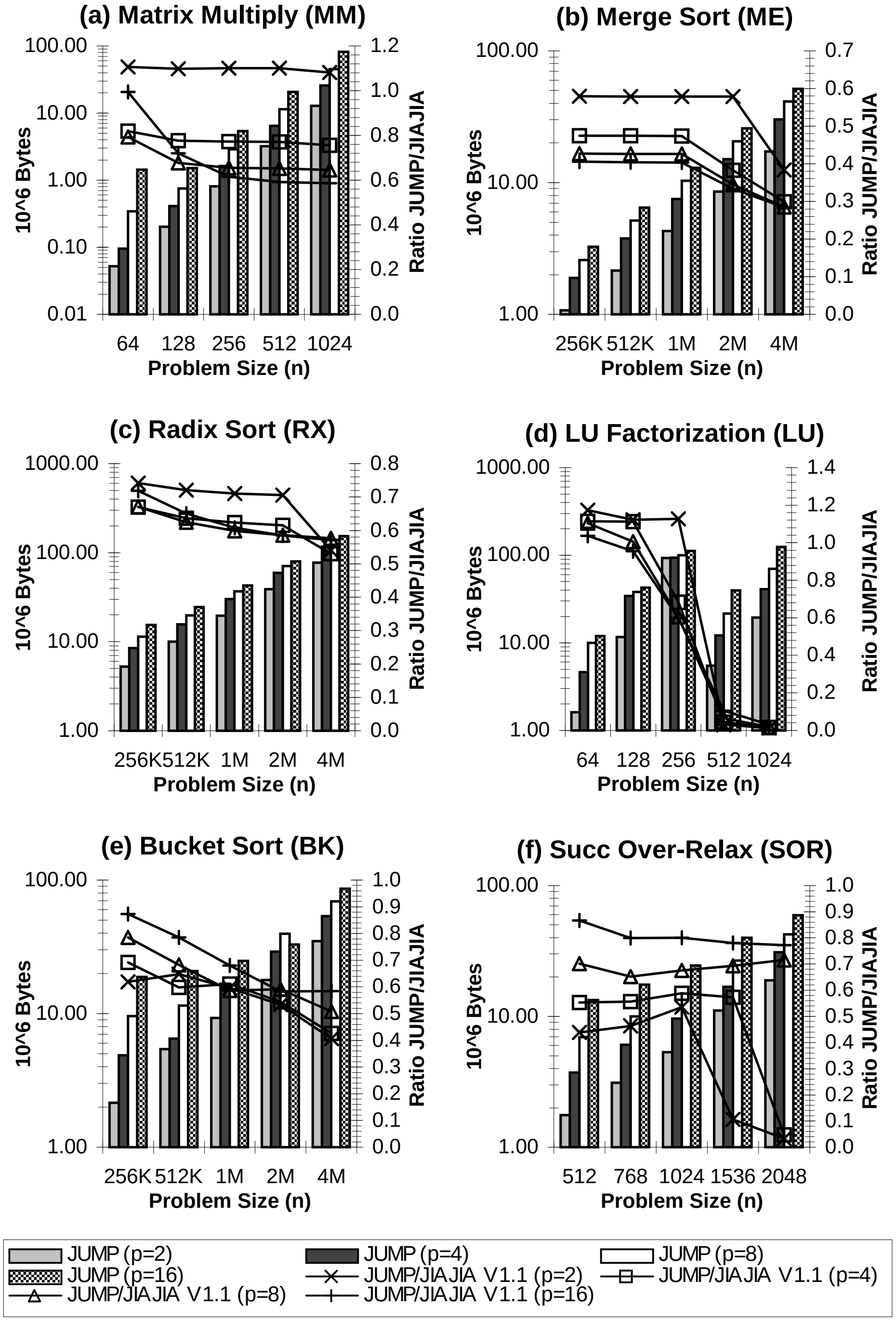


Figure 9. Graphs showing the total data traffic sent under JUMP (bar chart), and also the comparison of the total data traffic sent under JUMP and JIAJIA V1.1 (line chart), expressed by the ratio (total data traffic sent under JUMP / total data traffic sent under JIAJIA V1.1).

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, when sorting 2M integers, JUMP sends about 33.3% of the communication volume transmitted by JIAJIA. But when 4M integers are sorted, JUMP only sends 28.5% of the data traffic that is transmitted by JIAJIA through the network. In the extreme case, JUMP can save 97% of 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 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 two 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. Moreover, the extra number of messages sent under JUMP tends to be short messages, most of them being migration notices. Hence JUMP is expected to perform even better in systems with low-latency communication support, since the communication latency, which is the dominant factor in sending short messages, can be reduced under low-latency communication support.

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 three 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. Note that in most cases, PF1 faults are only required due to book-keeping purpose by the DSM system. For example, the system needs to trap the first write of a page within the critical section guarded by a lock, so that this information can be recorded and sent to later acquirers of the same lock, in order to invalidate the dirty copies of the page.

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.

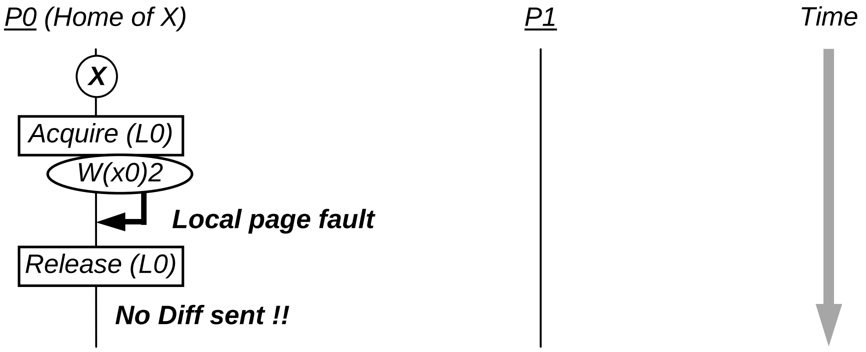


Figure 10. Serving the PF1 page fault. No remote processors are involved.

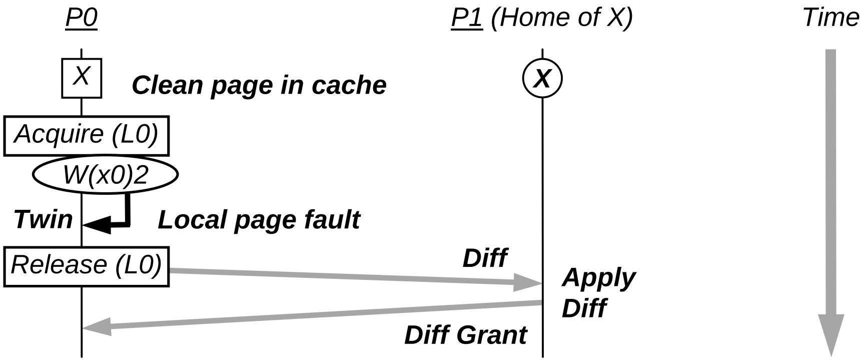


Figure 11. Serving the PF2 page fault. *P0* has cached a copy of page *X*.

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.

Table 2. Comparison among the four sub-categories of the PF3 page faults.

Page Fault Type	PF3A	PF3B	PF3C	PF3D
Occur on Read	✓	×	✓	✓
Occur on Write	✓	✓	×	×
Occur in Home-Based Protocol (JIAJIA)	×	✓	✓	×
Occur in Migrating-Home Protocol (JUMP)	✓	✓	×	✓
Requests Page from Remote	✓	✓	✓	✓
Involves Home Migration	✓	×	×	✓
Involves Diff and Diff Grant	✓	✓	×	✓ (1)
Note: (1) The Diff produced under PF3D faults are empty diffs, with no page update information if the page is not written before a release operation occurs.				

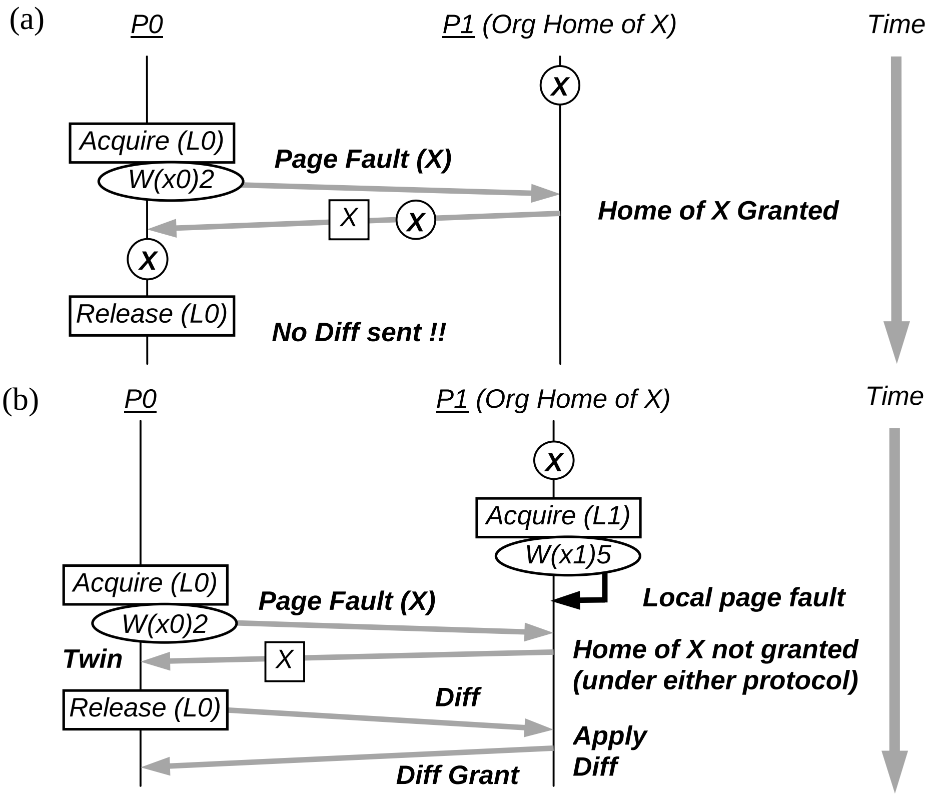


Figure 12. Two possibilities of the PF3 page fault. (a) PF3A: If the home is migrated to the page requester when serving the fault, no diff is sent. (b) PF3B: Home is not granted to page requester, causing the diff and diff grant at release time.

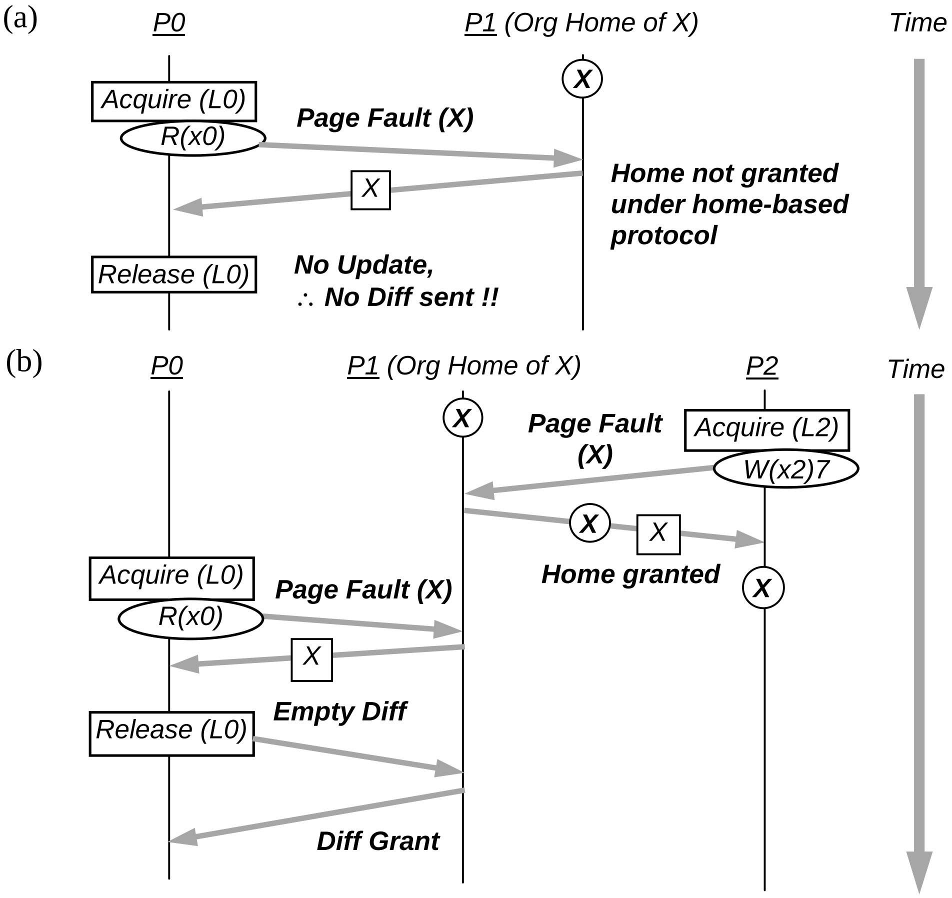


Figure 13. The other two possibilities of the PF3 page fault. (a) PF3C: A read fault does not cause a diff to be sent at synchronization under the home-based protocol. (b) PF3D: Under the migrating-home protocol, if the home is not granted to the faulting processor, an empty diff has to be sent at the release time.

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)).

PF3B: If the home 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 message and page grant message 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). 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. As the disk access time may be even slower than the network speed, PF2 faults may 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.

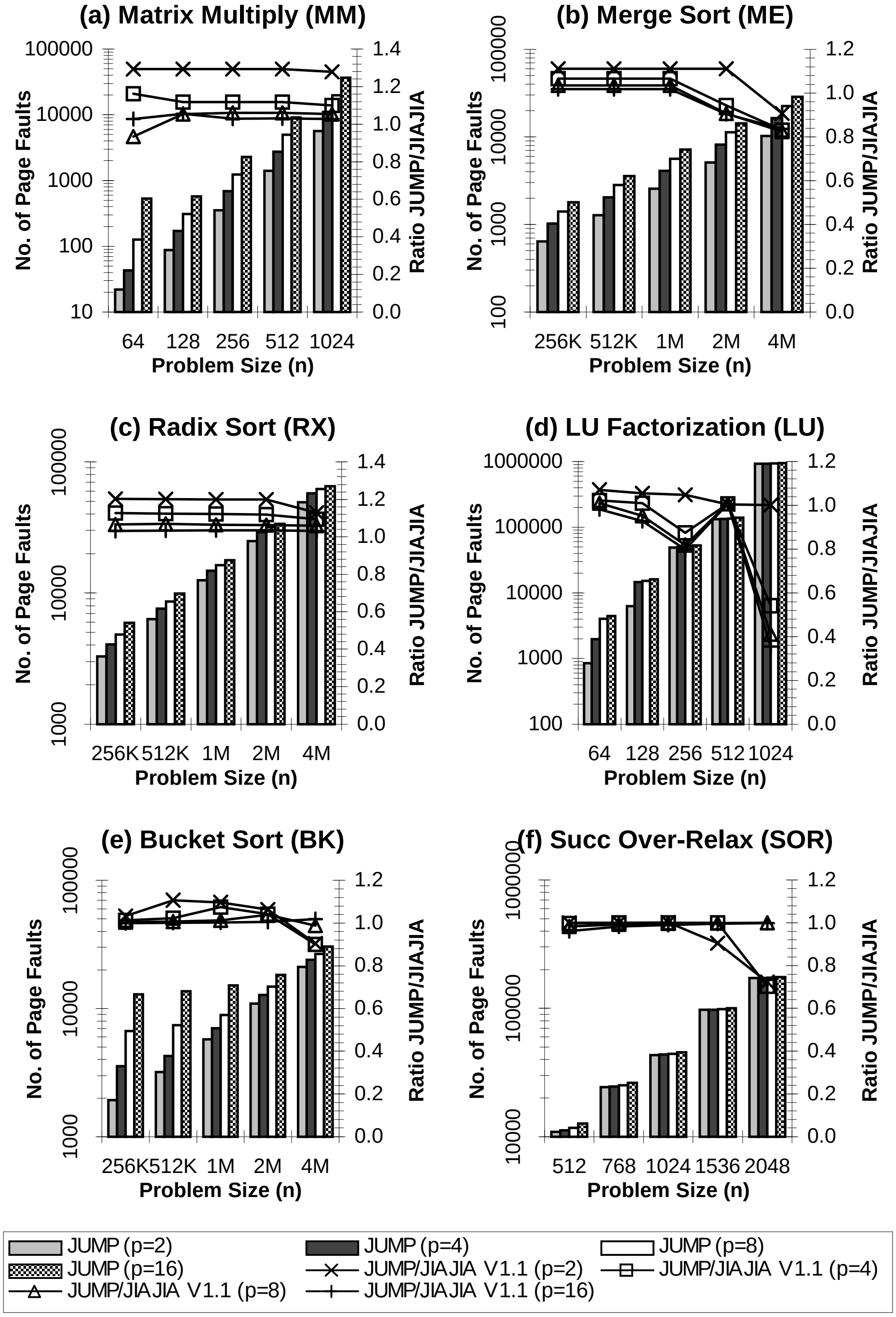


Figure 14. Graphs showing the total number of page faults under JUMP (bar chart), and also the comparison of the total number of page faults under JUMP and JIAJIA V1.1 (line chart), expressed by the ratio (number of page faults under JUMP / number of page faults under JIAJIA V1.1).

Table 3. Page fault breakdown for each application under JUMP and JIAJIA V1.1 using 16 processors.

Appl.	Size	Page Fault Breakdown under JUMP					Page Fault Breakdown under JIAJIA			
Name	n	PF1	PF2	3A	3B	3D	PF1	PF2	3B	3C
MM	64	75	163	239	35	22	16	225	21	258
	128	256	0	315	0	0	16	240	30	255
	256	964	0	1320	0	0	64	900	180	1074
	512	3856	0	5280	0	0	256	3600	720	4290
	1024	15424	0	21120	0	0	1024	14400	2881	17235
ME	256K	1040	0	752	0	0	80	960	240	480
	512K	2080	0	1504	0	0	160	1920	480	960
	1M	4160	0	3008	0	0	320	3840	960	1920
	2M	8320	0	6016	0	0	640	6280	3320	5614
	4M	16640	0	12032	0	0	1280	8921	10279	13805
RX	256K	2398	0	3546	0	0	188	2403	483	2692
	512K	4200	0	5701	0	0	320	4079	714	4475
	1M	7826	0	10022	0	0	573	7439	1200	8047
	2M	15070	0	18674	0	0	1085	14182	2154	15223
	4M	29533	0	35992	0	0	2121	27618	4081	29723
LU	64	308	1570	1905	72	608	110	1908	7	2522
	128	2471	4709	6716	636	1567	511	7620	30	9195
	256	20558	10701	18184	416	3495	2049	30600	120	34038
	512	130849	0	2035	0	6997	8209	122640	480	7545
	1024	916354	0	7423	0	21338	57266	859088	3700	1751935
BK	256K	8505	0	4375	0	0	4388	4121	259	4108
	512K	8794	0	4853	0	0	4423	4376	497	4332
	1M	9375	0	5807	0	0	4488	4888	976	4782
	2M	10520	0	7734	0	0	4622	5902	1940	5682
	4M	10204	0	20198	0	0	1280	8921	10534	9095
SOR	512	9164	568	2858	56	0	714	10191	541	1720
	768	21828	579	3854	38	9	1596	22172	1140	1880
	1024	39396	557	5487	54	12	2732	38992	1982	2280
	1536	90212	548	9076	62	11	6092	87032	4384	2840
	2048	161533	531	13627	55	10	10796	154212	7748	3381

Although more page faults are generated under JUMP than JIAJIA in general, as we shall see later, most of the page faults are PF1 faults. Hence they are in fact generated for DSM book-keeping purposes rather than real page misses, and hence they need less time to get served.

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, the percentage of extra page faults invoked decreases when more processors are used. However, the *actual* number of extra page faults is still on the increase in RX. This explains why JUMP executes RX with a slightly poorer performance than JIAJIA in the 8 and 16-processor cases.

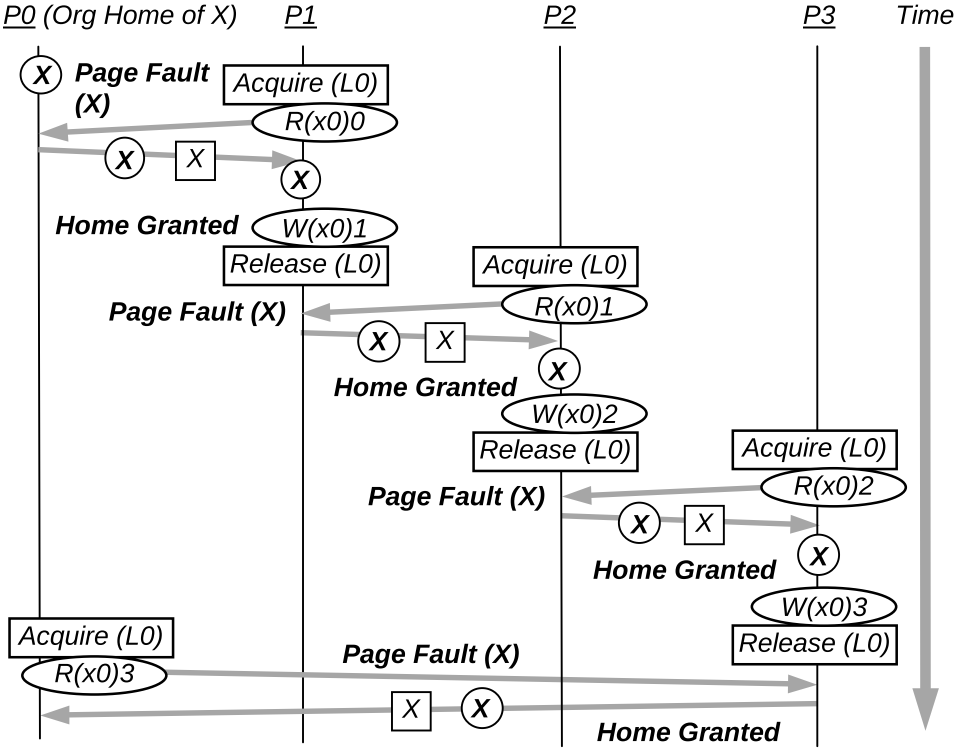


Figure 15. Demonstrating the extra page fault generated by the migrating-home protocol. Under the migrating-home protocol, *P0* causes a remote page fault when it reads *x0*.

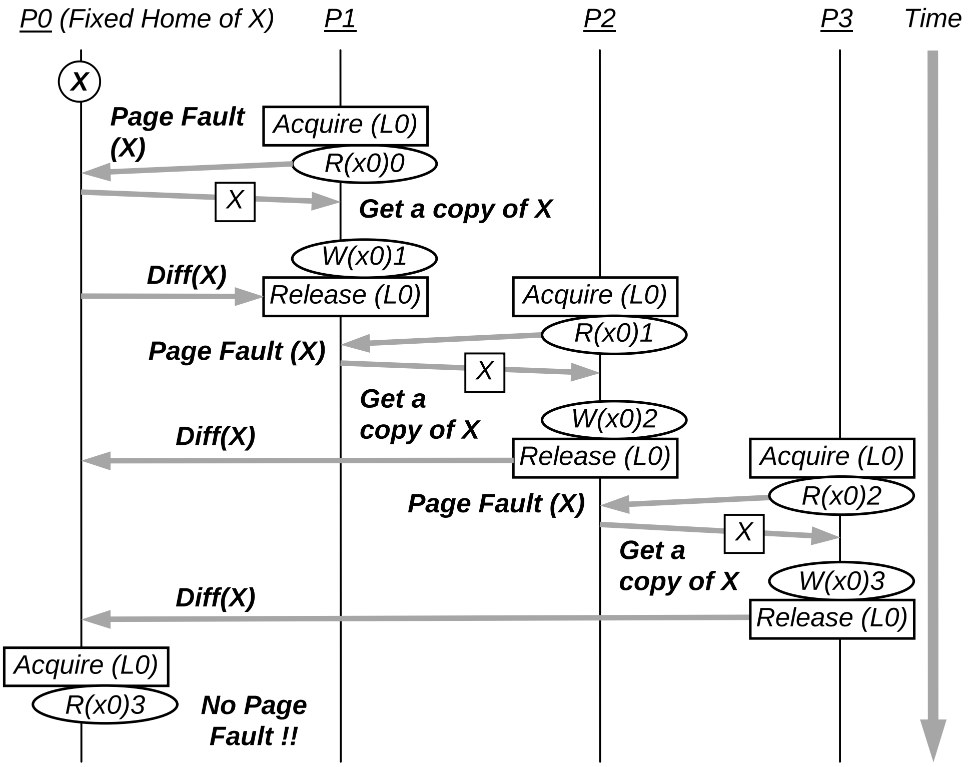


Figure 16. Demonstrating the extra page fault generated by the migrating-home protocol. For the home-based protocol, the remote PF3 page fault does not occur since it already contains the most up-to-date copy of the page *X*.

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. 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 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 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. We have discussed the underlying concept adopted by HLRC when the home-based protocol is introduced in Section 2.2. 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 [10] (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 [8].

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 a page in the shared memory can be migrated to another processor when the processor accesses the page. It reduces the amount of data communication among machines needed to maintain memory consistency. We have demonstrated that the protocol as has been incorporated in the JUMP DSM system is practical. 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 in five out of six applications tested. 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 that accesses the page, many of the page updates need not be sent to the home processor, hence reducing data traffic. In the best case as seen in the test, JUMP can save more than 97% of the total data traffic through the network during program execution. The performance obtained has motivated us to port the JUMP DSM system as a support layer for the Global Object Space (GOS) in the JESSICA project [17].

REFERENCES

1. L. Lamport. How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690-691, September 1979.

2. K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. On Computer Systems*, Vol. 7, no. 4, pages 321-359, 1989.

3. J. B. Carter, J. K. Bennett and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symposium on Operating Systems Principles* (*SOSP-13*), pages 152-164, October 1991.

4. 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, January 1994.

5. P. Keleher, A. L. Cox, W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture* (*ISCA'92*), pages 13-21, May 1992.

6. B. N. Bershad, M. J. Zekauskas and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE International Computer Conference* (*COMPCON Spring'93*), pages 528-537, February 1993.

7. B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Conistency for Distributed Memory Multiprocess-ors. CMU-CS-91-170.

8. L. Iftode, J. P. Singh and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proc. of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures* (*SPAA'96*), pages 277-287, June 1996.

9. P. Keleher. The Impact of Symmetry on Software Distributed Shared Memory. In *The Journal of Parallel and Distributed Computing (JPDC)*, 60(11): 1388-1419, 2000.

10. W. Hu, W. Shi and Z. Tang. A Lock-based Cache Coherence Protocol for Scope Consistency. *Journal of Computer Science and Technology*, 13(2):97-109, March 1998.

11. Y. Zhou, L. Iftode and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation* (*OSDI'96*), pages 75-88, October 1996.

12. B. Cheung, C. L. Wang and K. Hwang, A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations. In the *1999 International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA'99*), Las Vegas, Nevada, USA.

13. L. Iftode, J. P. Singh, J. P. and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proc. of the 23rd Annual International Symposium on Computer Architecture* (*ISCA'96*), pages 122-133, May 1996.

14. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21st Annual Symposium on Computer Architecture*, pages 142-153, April 1994.

15. W. Hu, W. Shi and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *Proc. of the High-Performance Computing and Networking Europe 1999* (*HPCN'99*), pages 463-472, April 1999.

16. B. Cheung, C. L. Wang and K. Hwang, JUMP-DP: A Software DSM System with Low-Latency Communication Support. In the *2000 International Conference on*

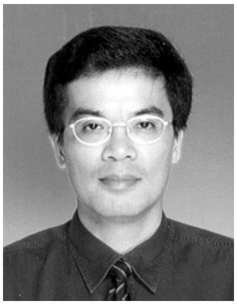
Parallel and Distributed Processing Techniques and Applications (*PDPTA’2000*), Las Vegas, Nevada, USA.

17. B.W.L. Cheung, C.L. Wang, and F.C.M. Lau. Building a Global Object Space for Supporting Single System Image on a Cluster, *Annual Review of Scalable Computing*, Chapter 6, Volume 4, Year 2002.

AUTHORS

Benny Wang-Leung Cheung (張宏亮) is currently a PhD student in the Department of Computer Science and Information Systems (CSIS) in the University of Hong Kong (HKU). His research topic is on Distributed Shared Memory (DSM) for Cluster Computing. He obtained the B.Eng degree in Computer Engineering in 1997, and the M.Phil degree in Computer Science in 2000, both in HKU as well. During his M.Phil study, he has developed the JUMP DSM System for Linux PC cluster. He is now developing a new DSM system, which provides large shared memory space, with low overhead in synchronization and maintaining memory consistency.

Dr. Cho-Li Wang (王卓立) received his B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985. He obtained his M.S. and Ph.D. degrees in Computer Engineering from University of Southern California in 1990 and 1995 respectively. His current research involves: High-Speed Cluster Networking, Distributed Java Virtual Machine, Parallel and Distributed Web Server System, Software Architecture for Network Computing and Embedded System. He is a member of the executive committee for IEEE Task Force on Cluster Computing (TFCC) and region coordinator (Hong Kong). He is also a member of ApGrid.



Dr. Francis Chi-Moon Lau (劉智滿) received the B.Sc. degree from Acadia University, Canada, and the M.Math. and Ph.D. degrees from the University of Waterloo. He joined the Department of Computer Science and Information Systems at The University of Hong Kong in 1987, where he is now the head of the department. His research interests are in parallel and distributed computing, object-oriented programming, operating systems, and Web and Internet computing. He is a member of the IEEE.