A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations

Benny Wang-Leung Cheung, Cho-Li Wang and Kai Hwang
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
{wlcheung+clwang+kaihwang}@csis.hku.hk

Abstract

The performance of a software *Distributed Shared Memory* relies on an efficient memory consistency model, and a suitable protocol for implementing the model. This paper studies a new concept of a *migrating-home* protocol for implementing the *scope consistency* model. In this protocol, the home, which is the location of the most up-to-date copy of each memory page, can change among processors. We implement this idea on a cluster of UltraSPARC I model 140 workstations with ATM network. The migrating-home protocol outperforms the home-based approach and adapts better to the memory access patterns for most applications. In particular, it reduces communication overheads in forwarding page updates among the processors. Page faults can also be handled more efficiently. These factors reduce a considerable amount of data communicating within the cluster.

1 Introduction

Cluster or network of workstations has been becoming an appealing computing platform for cost-effective parallel computing. But their usability and programmability depend much on the programming environment. One of the most popular parallel programming paradigms is software *Distributed Shared Memory* (DSM), which offers an abstraction of a globally shared memory across physically distributed memory machines.

In a DSM system, multiple copies of memory pages reside in different machines. A *memory consistency model* formally specifies how the memory system will appear to the programmer [1]. Memory consistency models affect the efficiency of DSM. The first DSM system, *IVY*, adopted *sequential consistency* (SC) [2]. However, its performance was poor due to excessive data communication among the machines. Later systems, such as *TreadMarks* [3] used the more relaxed *lazy release consistency* (LRC) model [4] to improve system efficiency. The *Midway* DSM system [5], on the other hand, employed *entry consistency* (EC) [6], which is more efficient than LRC. Unfortunately, the programming interface associated with EC is not easy to use. *Scope consistency* (ScC) has thus been proposed [7], aiming at achieving good programmability and performance.

To implement ScC, two *protocols*, namely the *home-based* and the *homeless* protocols, can be employed [8]. For the home-based (or fixed-home) protocol, a machine in the cluster is designated to keep the most up-to-date copy of every logical page in the system. Hence, when a processor requests a page, it only needs to communicate with the home. In comparison, to serve a page fault under the homeless protocol, a processor must send out requests to every processor which has updated the page, collects the updates from all other processors, and then applies the updates in sequence to obtain the most updated version of the page. Although both protocols adopt the *diffing* technique to handle the problem of *false sharing* [3], research [9] shows that the home-based protocol outperforms the homeless one since home-based protocol incurs less communication in the network.

This paper discusses a new *migrating-home protocol* for implementing ScC. In this protocol, the location storing the most up-to-date copy of each memory page is changeable among processors, so that in most cases, the processor requiring the page becomes the new home of the page. Processors send out short *migration notices* to notify others about the home change. With this feature, our protocol is able to perform less diffing operations and fewer lengthy diffs are sent among the machines. The protocol further reduces the communication overhead by concatenating multiple migration notices with the remaining diffs as a single message for transmission. It can also handle page faults in a more efficient way because more page faults can be served locally without sending extra messages to other processors. All these factors help to reduce the total amount of data communicating within the cluster effectively.

We implement the migrating-home protocol on the JUMP system, which runs on a cluster of 8 UltraSPARC I model 140 workstations with ATM networking. Four benchmark applications with different memory access patterns are tested. The migrating-home protocol adapts to the access patterns of most applications better than the original home-based protocol. This is shown by the testing results, in which the migrating-home protocol speeds up the execution time over the home-based approach for most applications. The maximum reduction in execution time is 67.5%. Our experiments also show that the migrating-home protocol can save up to 97% of the total amount of data sent within the cluster.

For the rest of the paper, Section 2 introduces the ScC model and the home-based protocol for implementing the model. Section 3 describes our migrating-home protocol in detail. Section 4 discusses the implementation of JUMP, the testing environment and benchmark applications. Section 5 addresses the testing result by comparing JUMP and its counterpart using the home-based protocol. Section 6 summarizes our conclusions.

2 Scope Consistency (ScC) and the Home-Based Protocol

In this section, we briefly describe the scope consistency model [7]. We also discuss the original home-based protocol [8] used to implement scope consistency to place the ground of our research.

The objective of *scope consistency* (ScC) is to provide high performance like EC and good programmability like LRC. This is achieved by the use of the *scope* concept, which reduces the amount of data updates sent among the processors, and fits naturally to the synchronization provided by 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 1, all critical sections guarded by the same lock comprise a scope. In other words, the locks used in a program implicitly determines the scopes, making the scope concept easy to understand. The acquire operation *opens* a scope, while the release operation *closes* the scope. From the above concepts, the consistency rule of ScC is defined as follows:

When a processor Q opens a scope previously closed by another processor P, P will propagate the updates made within the same scope to Q.

As an illustration, consider the example in Figure 1 again. In this example under ScC, *P0* does not propagate the updates for *x* and *y* to *P1* (while *P0* does under LRC), since the updates are not in the same scope (Scope 2). Therefore, the reads of *x* and *y* made by *P1* may give results other than 1.

ScC is more efficient than LRC, since ScC does not propagate the updates made in scopes other than the one being opened. On the contrary, LRC propagates all updates before the release. Moreover, ScC provides good programmability, since it retains the same programming interface as LRC. It does not require explicit binding of variables to locks as EC does. Hence we consider ScC a promising memory model for DSM.

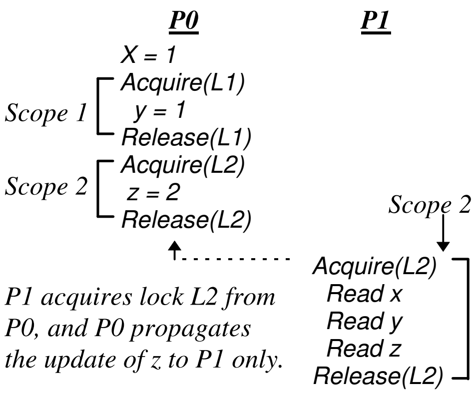

Figure 1. An Example for Scope Consistency.

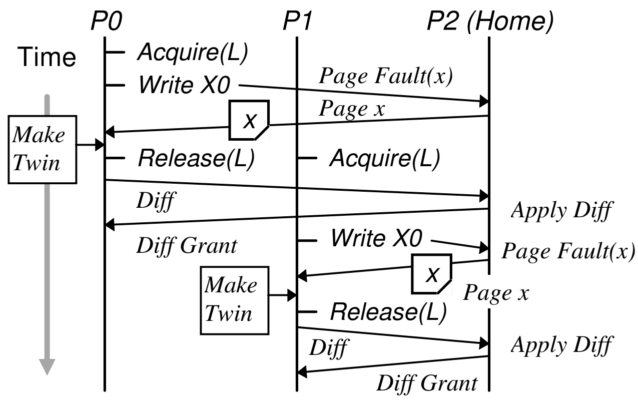

Figure 2. An Example for the Home-Based Protocol.

The original implementation of ScC is based on the *home-based protocol* [8], with the *diffing* technique to handle *false sharing* [3]. For each page in the memory, the protocol assigns a processor in which the most up-to-date copy subsides. This processor is the *home* of the page, fixed at the start of application execution. All other processors will send the page updates to the home at synchronization time. Upon a page fault, they can get a clean copy of the page by requesting the home processor. While in homeless protocols, no processor is responsible to keep the most updated copy of the page. To serve a page fault, a processor requests all the other members in the cluster to send the updates made by each of them, and then apply these updates in sequence to create a clean copy of the page. This introduces a high communication overhead and complicates the implementation as well. For example, we need to implement the timestamps on the diffs, so as to identify the order of the updates.

To illustrate the home-based protocol, we focus on a basic scenario of shared memory access in Figure 2. Assume variable *X0* is in page *x*. In the fixed-home protocol, processor *P0* tries to write to *X0* and generates a page fault. *P0* then requests the page from *P2*, the home of *x*. *P2* replies with a copy of the page. When *P0* gets the page, it makes a duplicate called a *twin* and performs the write. Later, at the release of lock *L*, *P0* compares the updated page with its twin and sends out the page difference, known as the *diff*, to *P2*. Finally, *P2* receives the diff and acknowledges *P0* with a *diff grant* message.

3 The Migrating-Home Protocol

Although the home-based protocol described in the previous section is more efficient than the homeless one, the concept of a fixed home can hinder the efficiency of the protocol. In this section, we study a new *migrating-home protocol*, in which the home is changeable among processors. From here onwards, we refer to the home-based protocol as the fixed-home protocol, as opposed to the migrating-home protocol.

3.1 The Migrating-Home Concept

The fixed-home concept does not adapt well to the access patterns of many applications. Its inflexibility introduces a pair of diff and diff grant messages communicating among processors, as shown in the previous example in Figure 2. These messages can be saved if we grant the home to *P0* when *P2* serves the page fault. As *P0* has the most updated copy of page *x*, the diff and diff grant messages need not be sent, as shown in Figure 3. This is the migrating-home concept -- when a processor asks for a page from its current home processor, the requesting processor can become the new home. Figure 4(a) shows this concept graphically. In the diagram, the circular "token" denotes the home of page *x*. It is moved from *P2* to *P0* to indicate the change of home.

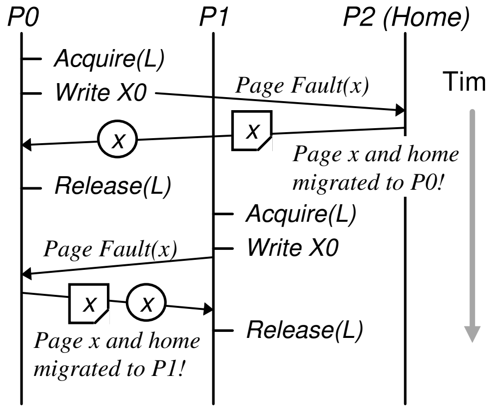

Figure 3. An Example Illustrating the Migrating-Home Concept

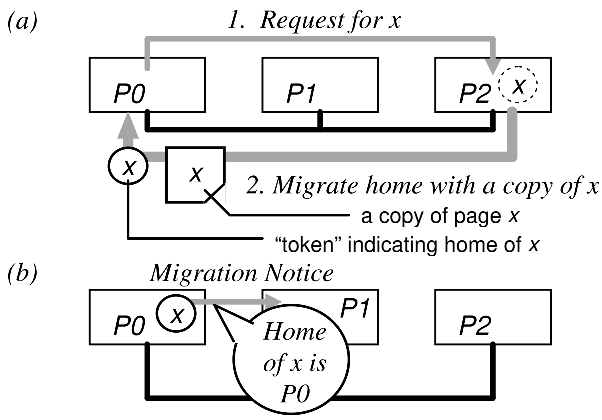

Figure 4. The Migrating-Home Concept: (a) Home Migration, (b) At Lock Release.

The migrating-home protocol causes a problem when *P1* requests page *x* after the home change, as shown in Figure 3. It may get an 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 synchronization time, as shown in Figure 4(b). This replaces the lengthy diffs, which can be the same size as the page itself. As a result, the migrating-home protocol reduces the number of bytes sent within the cluster. This improves the DSM performance. Moreover, as a DSM application usually accesses many memory pages, the migration notices can be merged together to form a single message to reduce the communication startup cost.

3.2 Solution for False Sharing

To deal with false sharing, the migrating-home protocol also uses the diffing technique, with additional rules to maintain memory consistency. When two processors ask the home for the same page before synchronization, the first becomes the new home, as shown in Figure 5(a). The late requester will receive a copy of the page from the previous home and also the new home location (Figure 5(b)). Therefore, it can send the diff to the new home. Under ScC, the copy obtained by the late requester is clean if the two processors do not access the same variable.

Moreover, the communication overhead of the diff can be reduced by concatenating it with the migration notices before sending when they have the same destination.

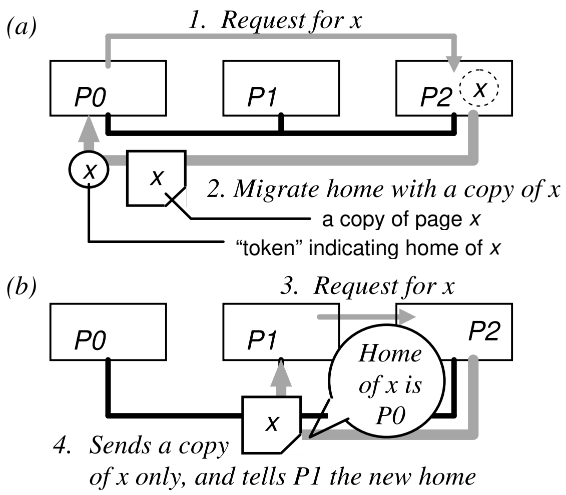

Figure 5. Handling the False Sharing Problem.

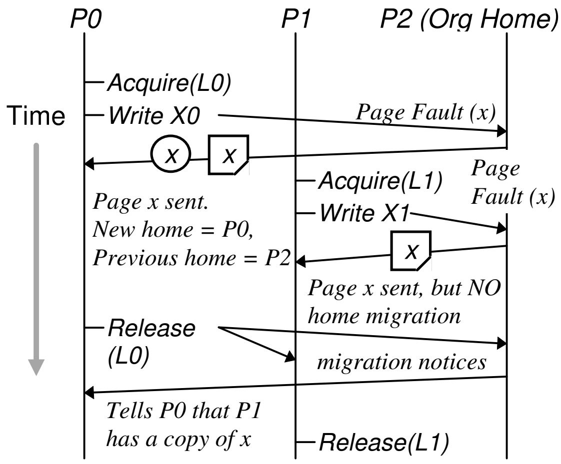

Figure 6. The Migrating-Home Protocol with False Sharing Support

Event	***Action***
Page fault	Send a request to the home of page
Request for page *x* from processor *P* received	1) If I am home of *x*, grant a copy of *x* to *P*. If (set of processors getting a copy of *x* = set of processors that have sent diff for *x*), then migrate the home to *P*, and I become the previous home of *x*. Record the ID of processor *P* otherwise. 2) If I am the previous home of page *x*, then grant a copy of *x* to *P*, and tell *P* about the new home. Record the ID of processor *P*. 3) If not 1) or 2), reject request, tell *P* to ask the new home.
Lock release	For every page *x*: 1) If I am home of *x*, send migration notice to every other processor. 2) Otherwise, send diff of *x* (if any) to the home of *x*.
Diff of page *x* received	I am the home of *x*, thus, send back the diff grant message.
Migration notice of page *x* received	1) If I am home of *x*, reply sender with the processors that have received a copy of *x* from me. Cancel the previous home flag. 2) Otherwise, note the home change.
Getting the page *x*	1) If home not granted, make a twin of *x* before writing the page 2) Otherwise, just write on the page.

Table 1. Table showing the Major Events and Actions in the Migrating-Home Protocol.

Table 1 shows the algorithm discussed above in tabular form. To illustrate the algorithm further, we give an example as shown in Figure 6. *P0* first writes to variable *X0*, causing a page fault for *x*. It requests *P2* for the page. *P2* sends *P0* a copy of *x* and grants *P0* the new home of *x*. Then, before *P0* releases lock *L0*, *P1* writes to *X1*, which is in page *x* too. In this false sharing situation, *P1* asks *P2* for the page and gets the copy, in which *X1* is still clean. However, *P2* cannot grant *P1* as the home of page *x*, since *P2* is no longer the home. Instead, *P2* informs *P1* that *P0* is now the new home of the page, so that *P1* can send the diff to *P0* at the release of lock *L1*.

At the release of *L0*, *P0* sends a migration notice to all processors, including *P2*. *P2* replies by notifying *P0* the processors that have got a copy from *P2*. *P0* can then decide whether it allows home change when it receives a fault on that page later. If some processors have not sent the diff, such as *P1* in this example, home migration cannot take place. On the other hand, after *P2* replies the migration notice from *P0*, *P2* will reject any page fault request of *x* from other processors, because *P2* has no way to inform the new home *P0* again. In such case, the previous home will reply the requester the new home of the page. This rarely happens, since all the processors usually should have received the migration notice. Hence they should have known the new home location.

4 Implementation and Benchmark Programs

We have implemented the migrating-home protocol by modifying the JIAJIA DSM system [9] to form the JUMP (JIAJIA Using Migrating-Home Protocol) system. JIAJIA is a software DSM system developed by the Chinese Academy of Sciences in 1998. It is a user-level C language runtime library built on the UNIX operating system. JIAJIA implements the scope consistency model with the home-based protocol. Like TreadMarks, JIAJIA handles false sharing by the diffing technique. It provides locks, barriers and condition variables as synchronization facilities.

We run the JUMP system on a cluster of 8 UltraSPARC I model 140 machines. Each of the workstations has 64MB of main memory and a page size of 8K bytes. Each machine runs the Sun OS 5.6 operating system. The network we use is a 16-port FORE ATM ASX1000 switch. The port bandwidth can achieve 155Mbps.

To test the performance of JUMP, we use four benchmark applications written in C, as described in Table 2. In the testing, we execute each of these programs with various problem sizes on 2, 4 and 8 nodes. We also run the same programs on the original JIAJIA implementation using the same environment.

Name	***Arg.***	***Problem Description***
MM	n, p	Matrix Multiplication of two $n \times n$ matrices using p processors
LU	n, p	LU Factorization on an $n \times n$ matrix using p processors
ME	n, p	Merge Sort on n integers using p processors (merge p sorted lists)
RX	n, p	Radix Sort on n integers using p processors

Table 2. Description of the Four Testing Applications

5 Results and Analyses

This section analyzes the performance of the migrating-home protocol on the JUMP system. We present the timing data, the total number and size of messages sent, as well as the number of page faults generated.

5.1 Execution Time Analysis

The decrease in execution time in JUMP as compared with JIAJIA depends on the application, as shown in Figure 7. Overall, JUMP has a much better performance over JIAJIA on most of the applications. For ME, JIAJIA runs 10.8% to 35.5% faster for small problem sizes with $n \leq 2M$. When $n = 4M$, JUMP runs the program in less than half the time needed by JIAJIA. JUMP even beats JIAJIA by running up to 67.5% faster on LU.

In RX, JUMP outperforms JIAJIA steadily by no more than 9.8%. The result is similar for MM, in which JUMP is at most 9.0% faster. The only exception is when JUMP executes MM with small matrices (n = 64 or 128) using 8 processors. At these data points, JUMP suffers a small performance degradation of 5% or less.

5.2 Communication Analysis

Table 3 summarizes the number of messages (*# Msgs*) sent for each application under JUMP and JIAJIA, together with their aggregate size (*Mbytes*) for the 8-processor case. We notice that with the exception of ME, JUMP sends a larger number of messages than JIAJIA for most of the data points. However, these messages are short ones, as JUMP sends fewer bytes than JIAJIA. Also, there is a large difference in the number of messages sent between JUMP and JIAJIA for small problem sizes. However, this gap becomes closer or even disappears for larger problems. This trend is observed in both LU and RX. This is because we take advantage of the short migration notices by concatenating them together, as discussed in Section 3.2.

A third point observed is that the migrating-home protocol in JUMP sends fewer bytes than the fixed-home counterpart in JIAJIA, especially on large problem sizes. For example, JUMP sends 42% to 66% fewer bytes than JIAJIA for the ME application. JUMP even achieves a 97% save for LU on large problem size (n = 1024). This reduced amount of data communication accounts for the performance improvement achieved by JUMP.

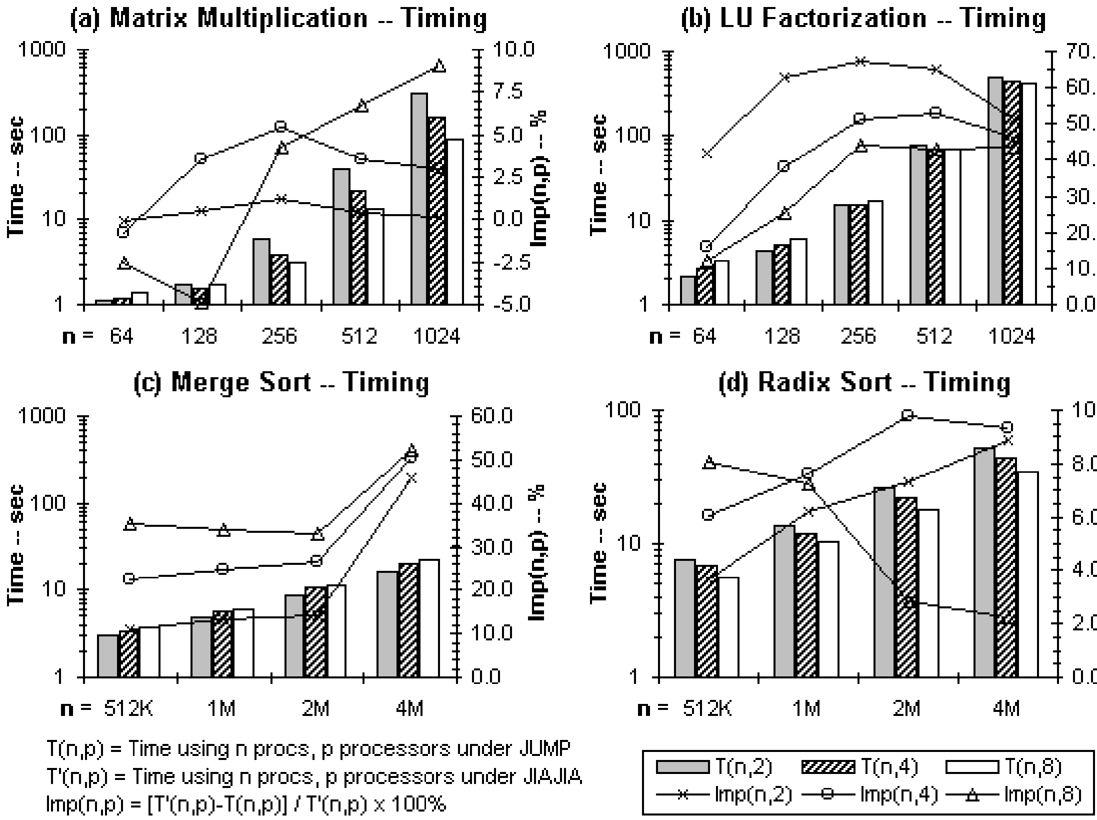

Figure 7. Performance Improvement of the Testing Applications
(Note that the bar chart, which shows the execution time, is in log scale.)

Appl. Name	Size (n)	JUMP						JIAJIA					
		# Msgs	MBytes	PF1	PF2	PF3	Total	# Msgs	MBytes	PF1	PF2	PF3	Total
MM	64	696	0.746	12	45	77	134	471	0.715	8	49	73	130
	128	1288	0.681	64	0	91	155	455	0.993	8	56	77	141
	256	1897	2.838	228	0	392	620	1236	4.031	32	196	358	586
	512	3829	11.216	912	0	1568	2480	3402	17.349	128	784	1428	2340
	1024	11557	44.731	3648	0	6272	9920	11208	69.315	512	3136	5712	9360
ME	512K	1503	5.073	800	0	608	1408	1645	12.017	128	672	560	1360
	1M	2719	10.134	1600	0	1216	2816	3213	24.035	256	1344	1120	2720
	2M	5151	20.256	3200	0	2432	5632	6349	48.066	512	2688	2240	5440
	4M	10015	40.500	6400	0	4864	11264	17944	118.473	1024	4165	7185	12374
RX	512K	6986	22.350	2123	0	2680	4803	5680	33.641	325	1896	2279	4500
	1M	10733	39.010	3952	0	4684	8636	9809	62.911	578	3469	4033	8080
	2M	18294	72.479	7593	0	8710	16303	18283	122.147	1081	6598	7626	15305
	4M	33610	139.656	14878	0	16790	31668	35336	240.622	2109	12866	14813	29788
LU	64	3665	4.571	2081	0	540	2621	2639	4.267	289	1792	427	2508
	128	7307	9.352	8257	0	1106	9363	5237	12.506	1089	7168	875	9132
	256	14822	20.021	32769	0	2371	35140	12680	56.263	4113	28656	1889	34658
	512	29932	41.345	130945	0	4899	135844	40868	351.686	16353	114592	3903	134848
	1024	59921	83.993	523905	0	9956	533861	196488	2597.700	65601	458304	7931	531836

Table 3. Comparison of Communication and Page Fault Statistics between JUMP and JIAJIA (p = 8)

5.3 Page Fault Analysis

The number of page faults is also useful in explaining the performance improvement of JUMP. In shared memory, there are 3 types of page faults as described below:

- ***PF1*:** Page faults that can be served by the local processor, which is the home of the page. They arise due to violation of access permission.
- ***PF2*:** Page faults that can be served by the local processor, which is not the home of the page. The local processor can serve the fault because it has cached a clean copy of that page.
- ***PF3*:** Page faults that have to be served by the remote processor.

Among the three types of page faults, *PF3* takes a long time to be served, as it involves a pair of messages sending between two processors to serve the page fault. However, serving a *PF2* type page fault can be more time-consuming although it can be served locally. This is because the page may be in the disk cache, and it costs a long time to bring the page from the disk back to the physical memory. Moreover, since the home is not local, the processor making the page fault must send a diff to the home of the page at synchronization time. The diff and the diff grant messages increases the cost of *PF2* faults. To improve the DSM performance, minimizing the number of *PF2* faults is equally important as that for *PF3* faults.

Table 3 shows the number of each type of page faults occurred in the testing applications under JUMP and JIAJIA. From the data, applications under JUMP produce 6% to 7% more page faults than JIAJIA. However, JUMP is able to convert most of the *PF2* faults in JIAJIA to *PF1* faults. As the number of *PF2* faults in JIAJIA accounts for 33% to 86% of the total number of page faults for the benchmark applications, JUMP significantly reduces the need to send lengthy diffs. Hence it improves the overall timing performance.

6 Conclusion

The protocol used in implementing a memory consistency model is as important as the model itself. It has a vital effect on the overall performance of the DSM system. The JIAJIA DSM system uses a fixed-home protocol to implement scope consistency. Although it is more efficient than TreadMark's homeless protocol, the concept of a fixed home for each memory page does not adapt well to the access patterns of the applications. This paper discussed the new idea of a migrating-home protocol. In the protocol, the home of each page can be migrated to another processor when it accesses the page. We show that the migrating-home protocol reduces the number of bytes sent in the cluster. In addition, the new protocol saves time in serving page faults by sending less diffs. With these factors, the migrating-home protocol improves the execution time of DSM applications substantially.

Reference

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

[2] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690-691, Sep 1979.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18-28, Feb 1996.

[4] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS91-170, Carnegie Mellon University, Sep 1991.

[5] B. N. Bershad, M. J. Zekauskas and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conference*, p.528-537, Feb 1993.

[6] P. Keleher, A. L. Cox and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture*, p.13-21, May 1992.

[7] 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 Symp. on Parallel Algorithms and Architectures*, Jan 1996.

[8] Y. Zhou, L. Iftode and K. Li. Performance Evaluation of the Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, p.75-88, 1996.

[9] W. Hu, W. Shi, Z. Tang and M. Li. A Lock-based Cache Coherence Protocol for Scope Consistency. *Journal of Computer Science and Technology*: 13(2), 1998.