JUMP-DP: A Software DSM System with Low-Latency Communication Support

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

Abstract: *Communication overhead within the network is always a main source of performance bottleneck for software DSM systems. In this paper, we introduce the JUMP-DP DSM system, which is an integration of the migrating-home protocol implemented by the JUMP DSM system, together with the low latency Socket-DP communication support. Testing results show that JUMP-DP not only reduces the amount of data communicating among processors like JUMP does, but the low-latency Socket-DP component in JUMP-DP is also capable of reducing the software protocol overhead in network communication. Therefore, JUMP-DP is able to further enhance the performance of DSM applications substantially over JUMP, which makes use of traditional BSD Sockets.*

Keywords: migrating-home protocol, low-latency communication.

1 Introduction

Software Distributed Shared Memory (DSM) is an attractive parallel programming paradigm on a cluster of PCs or workstations. It offers a shared memory abstraction over machines with physically distributed memory. Programmers view the whole system sharing a unified piece of memory, with no need to handle explicit data communication in writing programs.

However, the performance of DSM is often a major concern by users, as DSM tends to communicate excessive amount of data among processors to maintain memory consistency. This network bottleneck can be alleviated in three ways. In terms of hardware, we can use a high-bandwidth network to speed up the data transmission within the cluster. In the software aspect, we can reduce the amount of data communication in the network, or the software protocol overhead in the communication. The software solutions are more tempting, since performance can be improved with lower cost.

Many efforts have been made to reduce the amount of data transmission in DSM systems. Some employed a relaxed memory consistency model such as *lazy release consistency* (*LRC*) [1] or *scope consistency* (*ScC*) [2]. Others propose efficient memory coherence protocols. There is also work on reducing the software protocol overhead in network communication, such as *Active Messages* (*AM*) [3] and *Fast Messages* (*FM*) [4]. Yet few systems attempted to integrate the two technologies together.

In this paper, we introduce the *JUMP-DP* DSM system, which supports the low-latency *Directed Point* (DP) [5] communication on the *JUMP* DSM system [6] through the use of *Socket-DP*. It also adopts the migrating-home protocol in JUMP to implement ScC. JUMP-DP is implemented and tested on a cluster of 16 Pentium-III 450MHz PCs. Results show that JUMP-DP improves DSM performance by both software ways: a low-latency DP support together with an efficient migrating-home protocol.

For the rest of the paper, Section 2 gives an overview of the JUMP DSM system and its migrating-home protocol. Section 3 discusses Socket-DP, which is a layer built on top of DP with a BSD-Socket like interface. Section 4 describes the system structure of JUMP-DP. The performance testing and results are addressed in Section 5, followed by the related work in Section 6. We conclude this paper in Section 7.

2 Overview of JUMP

JUMP is a page-based software DSM system modified from JIAJIA [7] V1.1. It is built on top of UNIX, making use of its virtual memory manager and system calls to achieve the shared

The research was supported by the Hong Kong RGC grant HKU 7032/98E & HKU CRGC grant 335/065/0042.

memory abstraction on a cluster of homogeneous PCs or workstations. Like JIAJIA, JUMP adopts the ScC memory model. But instead of using the home-based protocol as adopted by JIAJIA V1.1, a migrating-home protocol was proposed and implemented in JUMP.

The migrating-home protocol is a memory coherence protocol, which specifies how the rules set by the memory consistency model are to be implemented. Various protocols can be adopted for implementing a consistency model, leading to different performance. Examples are the traditional homeless protocol [8] as adopted by TreadMarks [9], or the home-based protocol as adopted by JIAJIA. For each page in the shared memory space, the home-based protocol fixes a processor to hold the most up-to-date copy of the page. This processor is the *home* of the page. Under the home-based protocol, the updates made by every processor on a page must be propagated to the home processor at synchronization time. A processor can then serve a page fault by forwarding the request to the home processor of the page.

In contrast, the homeless protocol 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 the page fault, a processor has to contact all the peers which have updated that page. The other processors reply the request by sending the updates made on the page to the faulting processor (e.g. in the form of diffs [9]). The faulting processor then applies the updates in order (according to the timestamps specifying when the updates are made) to obtain the clean copy of the page. It is shown in [8] that the home-based protocol is more efficient than the homeless protocol, and as the home-based protocol needs not handle timestamps, it is easier to implement.

Although the home-based protocol is more efficient than the homeless protocol, the fact that a fixed home for every page throughout the program execution may not adapt well to the memory access pattern of the program. A page may never be accessed by its home processor. In such case, it is costly to serve multiple remote page faults throughout program execution. This is especially true for remote writes, as the updates need to be propagated back to the home processor at synchronization time.

Thus we proposed the migrating-home protocol, which allows the home location of a page to be migrated from a processor to another when the latter requests the page from the former, provided that certain conditions are met. The migrating-home concept is shown in Figure 1(a). In the figure, processor P writes to variable x in page X, and generates a page fault. Processor Q, being the original home of page X, receives the request from P. Q replies by sending a copy of page X to P, and migrates the home of the page to P. As P must access page X, it is an advantage for the home of the page to migrate to P, since any updates made by P to page X (in the form of diffs) need not be sent to other processors when P synchronizes. So the migrating-home protocol improves the DSM performance by reducing the amount of data communication made within the network.

To be more specific, the home of a page X will be migrated from processor Q to processor P when Q is serving a remote page fault from P, if and only if (1) Q is the home of X, and (2) all the other processors having a copy of X have sent the updates to Q. To compare the difference among the three protocols discussed in this section, the behavior for the home-based and homeless protocols in serving a page fault is shown in Figure 1(b) and 1(c) respectively. Readers interested in the migrating-home protocol may refer to [6] for more details.

It is shown in [6] that the migrating-home protocol reduces substantial amount of network communication, enhancing the performance of JUMP for most DSM applications over JIAJIA.

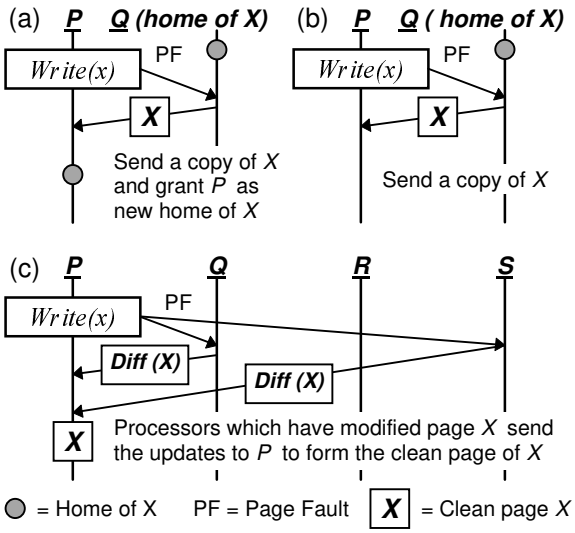

Figure 1. Serving a page fault in (a) the migrating-home protocol, (b) the home-based protocol, and (c) the homeless protocol.

3 Socket-DP

Socket-DP provides access to the low-latency Directed Point (DP) communication subsystem through traditional BSD Socket interface. With Socket-DP, users can access DP like ordinary BSD sockets, and Socket-DP will call the communication functions of DP for the user.

In Socket-DP, the creation and binding of a socket, as well as the sending and receiving of messages are all invoked using standard UNIX I/O system calls, just as BSD Sockets do. The only difference in the interface lies on the way to identify sockets. Instead of using IP address and port number, Socket-DP follows the syntax of DP and uses Node ID (NID) and Directed Point ID (DPID) to identify the target socket in the binding and message sending. The NID is used to identify each node in the cluster, while the DPID is used to identify every communication endpoint (i.e. socket) within a node.

Apart from a familiar programming interface, Socket-DP contains other features which are crucial for supporting the need of DSM:

- ***Signal handling for Asynchronous I/O:*** Asynchronous communication with signal handling is vital for DSM systems. When a message arrives, a signal is generated to inform the receiver. The process getting the signal should respond at once to reduce the waiting time of the sender. Signal handling for asynchronous I/O is thus vital for DSM. Socket-DP implements the signal handling feature by delivering a SIGIO signal to the receiving process on message arrival.
- ***Message disassembly and assembly:*** Each DP packet is of fixed size. For the Fast Ethernet implementation, the packet size is 1500 bytes. Long messages need to be cut explicitly into fragments before sending, and grouped at the receiver side. Socket-DP provides a simple mechanism for message disassembly and re-assembly to save programmers from the tedious work.
- ***The select() system call:*** Given a set of sockets, UNIX provides the *select()* system call to check if data has reached any of them. This call is used in JUMP to check any arrived messages not being served. Socket-DP provides a simple implementation for the *select()* system call. Unlike the *select()* system call for BSD Sockets which handles both message send and receive, the implementation of *select()* for Socket-DP only deals with message arrival. Thus the code becomes cleaner and more efficient.

To compare the performance of Socket-DP with BSD Sockets, we performed a simple test on the point-to-point round-trip communication time on both types of sockets as shown in Figure 2. The test is performed on two Pentium III 450MHz PCs connected by Fast Ethernet through an IBM 100-based switch. Each of the machines has 128 MB RAM and runs a piece of Linux 2.0.36 operating system. The results are plotted as a graph in Figure 3. It shows that Socket-DP has improved the performance over BSD sockets by more than 80% for messages not exceeding 32 bytes, as the DP layer saves a considerable amount of time in the software protocol overhead. For messages longer than 1KB, Socket-DP outperforms BSD Socket by 12.4-16.3%. Thus Socket-DP not only inherits DP low-latency communication feature, but it also provides a simple programming interface familiar to application users.

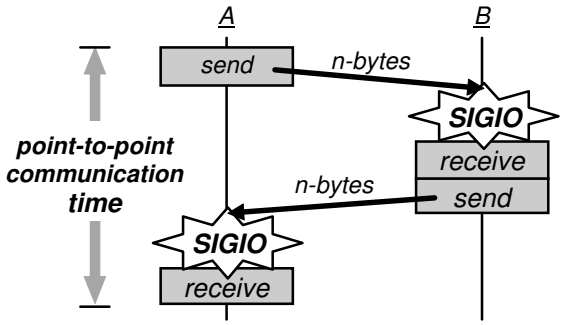

Figure 2. The Point-to-Point Round-Trip Test.

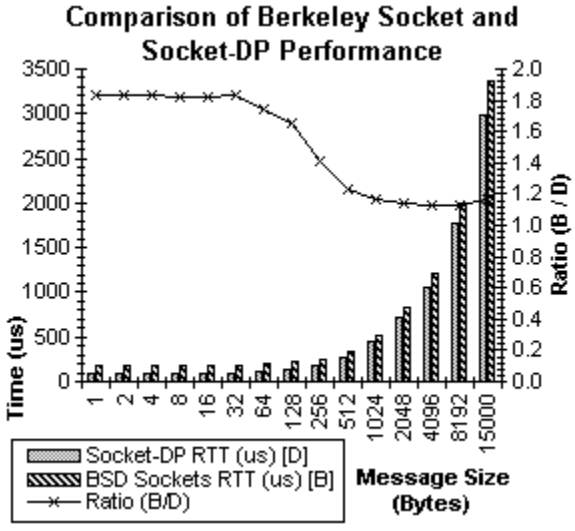

Figure 3. Comparison of RTT of BSD Socket and Socket-DP.

4 JUMP-DP

We embedded the DP communication support into JUMP to form the JUMP-DP DSM system. A low latency Directed Point support layer has been introduced into the system to back up JUMP communication subsystem. Instead of BSD Sockets, we use Socket-DP running on Fast Ethernet. As both types of sockets share a similar user interface, the effort in modifying the DSM system code is greatly simplified.

5 Testing and Results

We conducted a test to see the performance of JUMP-DP by comparing it with JUMP and JIAJIA V1.1 using BSD sockets in executing a suite of 6 benchmarks. The test was performed on a commodity cluster built using 16 Pentium III 450MHz PCs, connected using Fast Ethernet through a 24-port IBM 100-based switch. Each machine has 128MB of main memory. Each runs a copy of the Linux Kernel 2.0.36 as the operating system. All the three systems JIAJIA, JUMP and JUMP-DP are tested in this environment for a fair performance comparison.

The six benchmarks are described in Table 1. MM is a matrix multiplication program of 2 $n \times n$ matrices P and Q using p processors, with Q initialized as Q^T. Many implementations of MM divide the result matrix into p parts, and each processor handles the calculation of one part of the result matrix. Our implementation of MM takes a different approach. The two source matrices P and Q are divided into p parts, and each of the p processors only accesses one part of P and Q to calculate the subtotal value in its local matrix. The p local matrices are summed together to get the final result. ME performs merge sort on n integers appeared in p sorted lists using p processors. As two lists are merged at a time, the sorting is done in (log p) stages. RX does radix sort on n 32-bit integers. Each stage 4 bits are sorted, hence the sorting is done in 8 stages. LU performs LU-Factorization on an $n \times n$ matrix, and the results are verified for correctness. BK is a bucket sort program on n integers. Each of the p processors handles 256 buckets. After data distribution, each bucket holds the numbers within a certain range, and the numbers in a bucket are sorted by bubble sort. SOR is the red-black successive over-relaxation application performing on two $n \times n$ matrices. The main routine that performs the relaxation loops for 20 iterations.

The testing results are shown in Table 2. When we compare JUMP with JIAJIA V1.1, we can see that the migrating-home protocol in JUMP outperforms the home-based protocol in JIAJIA V1.1 for most of the applications. This matches the result in [6], in which a different testing environment is used. The performance improvement of JUMP over JIAJIA depends on the application and number of processors used. The largest improvement is observed at the LU Factorization application with n = 1024 and p = 16, in which JUMP runs nearly 15 times as fast as JIAJIA does. This is because the verification routine in LU generates an excessively large amount of remote page faults in JIAJIA, which can be much reduced under the migrating-home protocol in JUMP, since the home migration turns the subsequent page faults to local ones.

Next, as we compare the timing data for JUMP and JUMP-DP, we observe that for all testing benchmarks, JUMP-DP runs faster than JUMP using BSD Sockets. Most applications enjoy a 10-20% improvement in performance. This means the low-latency communication support by Socket-DP in JUMP-DP is capable of further enhancing the DSM performance.

Name	*Parameters*	*Description*
MM	*n, p*	Matrix Multiplication of two $n \times n$ matrices using p processors
ME	*n, p*	Merge Sort of n integers appeared in p sorted lists using p processors
RX	*n, p*	Radix Sort of n 32-bit integers using p processors in 8 stages
LU	*n, p*	LU Factorization of a $n \times n$ matrix using p processors with result verification
BK	*n, p*	Bucket Sort of n integers using p processors and $256 \times p$ 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

Table 1. Description of the six benchmark applications used.

Appl. Name	Size n	JIAJIA V1.1 Time (sec)			JUMP Time (sec)			JUMP-DP Time (sec)			JIAJIA V2.1 Time (sec)		
		$p = 4$	$p = 8$	$p = 16$	$p = 4$	$p = 8$	$p = 16$	$p = 4$	$p = 8$	$p = 16$	$p = 4$	$p = 8$	$p = 16$
MM	64	0.051	0.093	0.218	0.061	0.122	0.293	0.047	0.105	0.203	0.066	0.144	0.473
	128	0.215	0.214	0.226	0.195	0.237	0.428	0.171	0.201	0.332	0.215	0.220	0.330
	256	1.037	0.895	0.795	0.942	0.798	0.906	0.869	0.710	0.733	1.053	0.874	0.830
	512	6.465	4.489	3.640	6.089	4.108	3.330	5.766	3.760	2.906	6.405	4.495	3.587
	1024	44.026	26.695	18.155	42.520	25.331	17.034	41.355	24.418	15.749	44.284	27.310	18.837
ME	256K	0.777	0.909	1.005	0.462	0.494	0.547	0.408	0.439	0.491	0.799	0.918	1.004
	512K	1.545	1.789	1.931	0.897	0.946	1.007	0.775	0.835	0.888	1.553	1.792	1.940
	1M	3.081	3.561	3.824	1.775	1.847	1.918	1.531	1.612	1.701	3.093	3.557	3.837
	2M	7.875	9.634	10.565	3.521	3.681	3.775	3.062	3.203	3.341	6.200	7.310	7.802
	4M	20.096	23.750	20.648	7.176	7.494	7.673	6.218	6.627	6.875	15.655	18.259	19.154
RX	256K	1.685	1.555	1.204	1.568	1.358	1.356	1.287	1.131	1.142	1.274	1.240	1.329
	512K	3.163	2.803	2.067	2.973	2.578	2.180	2.545	2.134	1.836	2.380	2.176	2.086
	1M	6.100	4.844	3.908	5.981	4.913	4.038	4.941	4.065	3.424	4.644	4.146	3.787
	2M	11.999	9.202	7.972	11.983	9.716	8.368	10.091	8.116	7.410	9.238	8.505	7.545
	4M	28.400	18.662	14.649	24.414	20.219	17.550	20.358	16.885	14.591	24.476	17.957	16.104
LU	64	0.811	1.128	1.170	0.967	1.804	1.653	0.779	1.419	1.590	0.968	1.909	2.444
	128	5.715	4.388	3.422	6.405	4.395	3.972	5.568	3.750	3.960	6.059	6.971	8.119
	256	24.872	18.965	12.926	13.872	10.362	10.415	12.595	9.613	9.883	25.597	28.610	30.934
	512	45.718	35.582	30.916	18.267	18.836	21.853	17.706	18.054	20.581	18.259	19.063	22.342
	1024	1168.88	1646.13	1994.00	134.38	129.04	133.99	132.54	125.86	130.03	1094.15	1706.87	2147.94
BK	256K	1.559	1.082	2.271	1.361	1.000	2.543	1.245	0.874	2.341	2.391	3.709	7.812
	512K	3.782	1.754	2.622	3.399	1.552	2.563	3.277	1.413	2.281	4.773	4.713	8.276
	1M	13.275	4.048	2.773	12.224	3.614	2.690	11.795	3.554	2.683	13.948	7.794	9.828
	2M	45.073	14.117	5.041	42.917	12.899	4.683	42.356	12.461	4.428	47.668	19.394	14.102
	4M	169.83	47.440	15.584	162.89	44.041	14.430	161.97	43.366	14.340	175.67	58.849	30.967
SOR	512	1.745	1.684	2.273	1.019	0.934	1.164	0.914	0.843	1.096	1.027	1.101	1.615
	768	3.259	2.575	2.305	1.884	1.439	1.440	1.765	1.284	1.289	1.810	1.665	2.005
	1024	5.807	4.191	3.893	3.269	2.208	1.957	3.062	1.947	1.753	3.270	2.602	2.730
	1536	12.335	8.930	7.717	6.420	4.074	3.133	5.912	3.697	3.031	6.681	4.865	4.654
	2048	100.635	18.683	16.319	12.163	7.457	5.277	11.367	6.980	4.879	14.388	8.974	7.908

Table 2. Execution time of the 6 benchmark applications under JIAJIA V1.1, JUMP, JUMP-DP and JIAJIA V2.1 with 4, 8 and 16 processors.

If we compare the performance of JUMP-DP with JIAJIA V1.1 using BSD Sockets, a substantial improvement in performance can be obtained for most applications. This is shown more clearly as we look at the performance ratio, which is expressed as the execution time of an application under JIAJIA V1.1, divided by the execution time of the same application under JUMP-DP, using the same problem size n and number of processors p. A performance ratio over 1 means that JUMP-DP improves the application performance over JIAJIA V1.1. The higher the ratio, the more the improvement JUMP-DP makes.

The performance ratios of each application are plotted as graphs shown in Figure 4. From the graphs, it is observed that most applications achieve a performance ratio larger than 1. The performance improvement is achieved by two factors: the efficient migrating-home protocol which reduces the amount of communication among machines in the cluster, and the low-latency DP support which reduces the protocol overhead in the communication. Applications with heavy data communication and favorable memory access patterns are benefited most. Examples are ME and SOR. For ME, JUMP-DP executes the application 90.7-273% faster than JIAJIA V1.1, while for SOR, JUMP-DP beats JIAJIA by running 78.9-785.3% faster.

Finally, we consider the speedup of each application under JUMP-DP. We find that the speedup is application-dependent. BK obtains the best speedup due to the $\Theta(n^2)$ complexity of the bubble sort routine. The speedup for MM is also good since the amount of computation is relatively high. However, the execution time of LU does not improve much with more processors, as its verification routine dominates the execution and takes rather constant time. ME runs even slower with more processors since the number of merging stages increases.

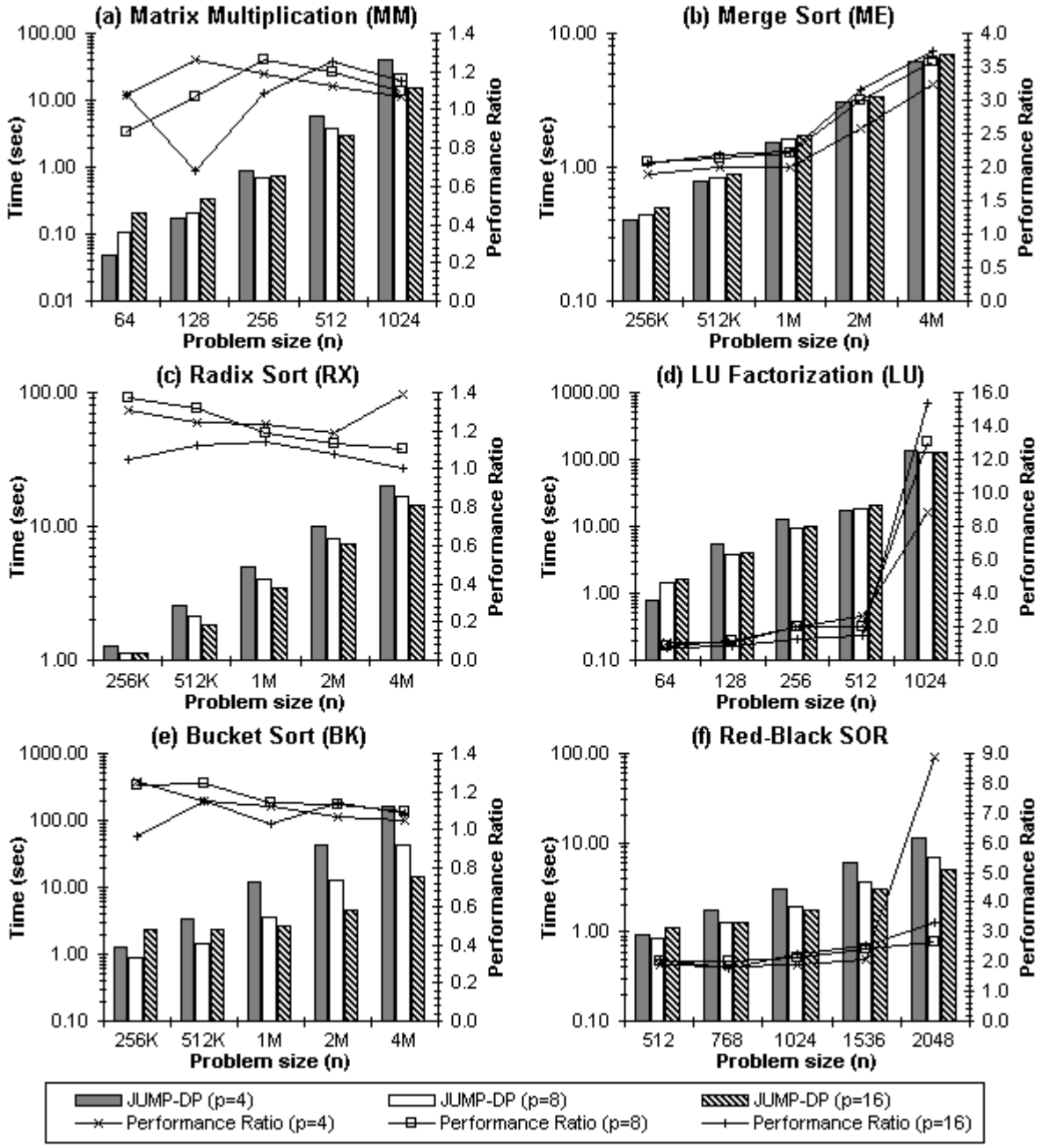

Figure 4. Comparing the execution time of the six benchmark applications under JUMP-DP and JUMP. The lines show the performance ratio of JUMP-DP over JIAJIA V1.1, obtained by (execution time of JIAJIA V1.1 / execution time of JUMP-DP). The bar chart shows the execution time of each application under JUMP-DP in log10 scale.

6 Related Work

There are some other works which deal with the home migration of shared memory pages 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 [10] released in 1999. It has the same objective as our migrating-home protocol in JUMP, but the mechanism is quite different. Instead of migrating the home of a page eagerly in serving a page fault, JIAJIA V2.1 migrates the home at the barrier after the remote page fault is served. The home is migrated if there is only one writer to a page. This less-aggressive strategy tries to reduce the broadcasting over-

head of the new home location by appending this information with the barrier grant message. But the main drawback is that the rule for home migration is too strict. If two processors write to the same page between two barriers, the home is not migrated. Also, applications using locks cannot be benefited from this protocol.

We tested the performance of JIAJIA V2.1 and compared it with our JUMP and JUMP-DP systems. The testing environment used is the same as mentioned in Section 5, and the timing results are also presented in Table 2. We find that our migrating-home protocol in JUMP outperforms the home migration protocol in JIAJIA V2.1 for most applications. The only exception is RX, where JIAJIA V2.1 beats JUMP and JUMP-DP by 2.0-22.9%.

7 Conclusions

We have proposed in this paper the JUMP-DP DSM system. It improves software DSM performance in two ways. First, JUMP-DP adopts the migrating-home protocol in JUMP to implement the relaxed ScC model. The aggressive strategy adopted by the migrating-home protocol is able to adapt better to the memory access patterns of DSM applications in most cases, hence reducing the amount of data communication within the network. Second, Socket-DP is employed to support low-latency communication in JUMP-DP, reducing the software communication overhead. Our testing shows that the two enhancements of JUMP-DP combined to improve the performance of DSM applications substantially.

References

[1] 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.

[2] 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.

[3] T. von Eicken, D. E. Cullerand, S. C. Goldstein and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th International Symposium of Computer Architecture*, May 1992.

[4] S. Pakin, M. Lauria and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, California, December 1995.

[5] C. M. Lee, A. Tam and C. L. Wang. Directed Point: An Efficient Communication Subsystem for Cluster Computing. In *Proc. of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 662-665, Las Vegas, October 1998.

[6] 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 9*), Las Vegas, Nevada, USA.

[7] 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.

[8] 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.

[9] 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.

[10] 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.