Reducing System Overheads in Home-based Software DSMs *

Weiwu Hu Weisong Shi and Zhimin Tang
Institute of Computing Technology
Chinese Academy of Sciences
E-mail:{hww,wsshi,tang}@water.chpc.ict.ac.cn

Abstract

Software DSM systems suffer from the high communication and coherence-induced overheads that limit performance. This paper introduces our efforts in reducing system overheads of a home-based software DSM called JIAJIA. Three measures, including eliminating false sharing through avoiding unnecessarily invalidating cached pages, reducing virtual memory page faults with a new write detection scheme, and propagating barrier message in a hierarchical way, are taken to reduce the system overhead of JIAJIA. Evaluation with some well-known DSM benchmarks reveals that, though varying with memory reference patterns of different applications, these measures can reduce system overhead of JIAJIA effectively.

1 Introduction

Software DSMs suffer from the high communication and coherence-induced overheads that limit performance. The communication cost of software DSMs is evidently high, large granularity of coherence causes problems of false sharing, encoding and decoding *diffs* in multiple writer protocols are time consuming, and enforcing coherence through virtual memory page protection and page fault introduces much system overhead. Many techniques, such as multiple writer protocol[4], lazy release consistency[9], runtime write trapping[19], and hardware support[3], have been proposed to reduce false sharing, minimize remote communication, and hide communication latency.

This paper introduces our efforts in reducing system overheads of a software DSM called JIAJIA[7]. JIAJIA is a home-based software DSM system in which physical memories of multiple computers are combined to form a larger shared space. It implements the lock-based cache coherence protocol for scope consistency[8]. The protocol is lock-based because it totally eliminates directory and maintains coherence through accessing write notices kept on the lock. In a recently released version of JIAJIA, measures are taken to reduce its system overhead. These measures include optimizations to avoid unnecessarily invalidating cached pages through minimizing write notices sent from the lock releasing processor to the lock manager and from the lock manager to the acquiring processor, a cache only write detection (CO-WD) scheme to reduce the number of page faults caused by writing home pages, and a hierarchical scheme to speedup the propagation of barrier messages.

Evaluation with some widely accepted DSM benchmarks such as SPLASH2 program suite and NAS Parallel Benchmarks indicate that, though varying with memory reference patterns of different applications, these measures can reduce system overhead of JIAJIA effectively. Among nine applications we tested, the first optimization method achieves a 5% $\sim$ 50% reduction of execution time in seven applications, the second method achieves a 2% $\sim$ 48% improvement in three applications, and the third method achieves a 1% $\sim$ 3% improvement in three applications.

The rest of this paper is organized as follows. The following Section 2 introduces the JIAJIA software DSM system on which our experiments are based. Section 3 illustrates measures taken to reduce system overhead of JIAJIA. Section 4 presents experiment results and analysis. The conclusion of this paper is drawn in Section 5.

2 The JIAJIA Software DSM System

2.1 Memory Organization

Figure 1 shows JIAJIA's organization of the shared memory. Unlike other software DSMs that adopt the COMA-like memory architecture, JIAJIA organizes the shared memory in a NUMA-like way. In JIAJIA, each shared page has a fixed home node and homes of shared pages are distributed across all nodes. References to local part of shared memory always hit locally. References to remote shared pages cause these pages to be fetched from its home and cached locally. When cached, the remote page is kept at the same user space address as that in its home node. In this way, shared address of a page is identical in all processors, no address translation is required on a remote access.

In JIAJIA, shared pages are allocated with the `mmap()` system call. Each shared page has a fixed global address which determines the home of the page. Initially, a page is mapped to its global address only by its home processor. Reference to a non-home page causes the delivery of a SIGSEGV signal. The SIGSEGV handler then maps the fault page to the

*The work of this paper is supported partly by National Climbing Program of China, National Natural Science Foundation of China, and President Young Creation Foundation of the Chinese Academy of Sciences.

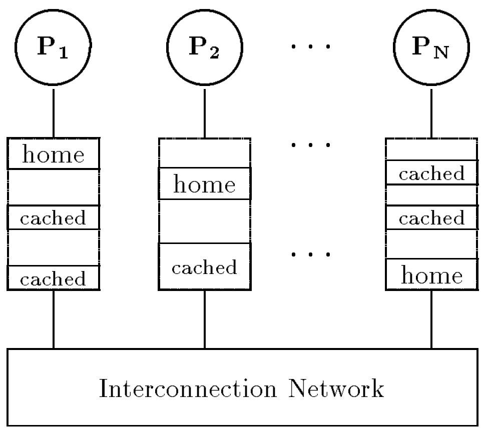

Figure 1. Memory Organization of JIAJIA

global address of the page in local address space. Since the total shared memory allocated may be larger than the physical memory of one host, mapping too many remote pages will break the system down. To avoid this, each host maintains a "cache" data structure to record all locally kept non-home pages. Any locally kept remote page must find a position in the local cache. If the number of locally kept remote pages is larger than the maximum number allowed, some aged cache pages must be replaced (unmapped) to make room for the new page.

With the above memory organization, JIAJIA is able to support shared memory that is larger than physical memory of one machine. In other software DSMs such as TreadMarks, CVM, and Quarks, the shared space is limited by the physical memory of one machine because no cache replacing mechanism is implemented and hence each host should have the capability of holding all shared pages. Besides, in these sytems, each processor maintains a large local page table to keep directory information (to locate a page on a page fault), twins, diffs, protect states, and local and global addresses of all pages. The size of this page table scales linearly with the number of shared pages. In JIAJIA, homes of shared pages are distributed across all processors and the page table contains only information about "cached" pages. For each cached page, the page table only keeps its address, protect state, and a twin for writable pages[1].

Another important feature of JIAJIA's memory organization is that it allows the programmer to flexibly control the initial distribution of homes of shared locations. The basic shared memory allocation function of JIAJIA allows the programmer to allocate a certain size of shared memory block by block across all processors. The processor from which the allocation starts can also be indicated.

[1]In the recent version of JIAJIA, there is also a page table of all shared pages. However, each item of the page table requires only six bytes.

2.2 Cache Coherence Protocol

Based on the observation that the benefit of complex design of a software DSM system may be offset by the system overhead caused by the complexity, the cache coherence protocol of JIAJIA is designed as simple as possible. Instead of supporting multiple consistency models and adaptive write propagation strategies as some previous DSM systems did, JIAJIA takes a fixed memory consistency model and fixed write propagation (write invalidate) strategy. Multiple writer technique is employed to reduce false sharing.

JIAJIA implements the scope consistency (ScC)[8] which is even lazier than lazy release consistency (LRC)[9]. Adopting the ScC greatly simplifies the lock-based cache coherence protocol of JIAJIA. In TreadMarks which implements LRC, complex data structures such as *intervals* and *vector timestamp* is employed to record the "happen-before-1" relation of memory accesses among different processors. In JIAJIA, ScC does not require to implement the complete "happen-before-1" relation. Instead, it only requires previous intervals related to the acquired lock to be visible to the acquiring processor on an acquire.

The distinguishing characteristic of JIAJIA's cache coherence protocol is its lock-based feature, i.e., it does not rely on any directory information to maintain coherence. Instead, coherence is maintained through writing and reading *write-notices* on the lock. The lock-based protocol can be summarized as follows.

In the protocol, each page has a fixed home and can be cached by a non-home processor in one of three states: Invalid (INV), Read-Only (RO), and Read-Write (RW). As a special kind of shared object, each lock also has a home node.

On a release, the releaser performs a comparison of all cached pages written in this critical section with their twins to get *diffs* of this critical section. These *diffs* are then sent to homes of associated pages. After all *diffs* have been applied to home pages, a release message is sent to the home of the associated lock to release the lock. Besides, the releaser piggybacks write-notices of the critical section on the release message notifying the modifications in the critical section.

On an acquire, the acquiring processor sends a lock acquiring request to the lock manager. The requesting processor is then stalled until it is granted the lock. When granting the lock, the lock manager piggybacks write-notices associate with this lock on the granting message. After the acquiring processor receives this granting message, it invalidates all cached pages that are notified as obsolete by the associated write-notices.

A barrier can be viewed as a combination of an unlock and a lock. Arriving at a barrier ends an old "critical section", while leaving a barrier begins a new one. In this way, two barriers enclose a critical section. On a barrier, all write notices of all locks are cleared.

On a read miss, the fault page is fetched from the home in RO state in the local memory.

On a write miss, if the written page is not presented or is in INV state in the local memory, it is fetched from the home in RW state. If the written page is in RO state in the local memory, the state is turned into RW. A write-notice is recorded about this page and a twin of this page is created before written.

It can be seen from the above description that, compared with directory-based protocols, all coherence related actions in our protocol are taken in synchronization points. In this way, the lock-based protocol has least coherence related overheads for ordinary read or write miss. Besides, the lock-based protocol is free from the overhead of maintaining the directory.

3 Reducing System Overhead

3.1 Reducing False Sharing

False sharing constitutes a major overhead in software DSMs due to large granularity of coherence and high cost of communication. In JIAJIA, coherence is maintained through requiring the lock releasing processor to send write notices generated in the associated critical section to the lock manager and the lock acquiring processor to invalidate its locally cached data copies according to the write notices of the lock. To reduce false sharing, write notices that will cause superfluous invalidation should not be included in the message from the lock releasing processor to the lock manager, or from lock manager to the lock acquiring processor. We take the following measures to reduce write notices sent to acquiring processor on a lock or barrier.

In the basic protocol, a cached page is invalidated on an acquire or on a leaving of barrier if there is a write notice in the lock indicating that this page has been modified. However, if the modification is made by the acquiring processor itself, and the page has not been modified by any other processors, then the invalidation is unnecessary since the modification has already been visible to the acquiring processor. With this optimization, a processor can retain the access right to pages modified by itself on passing an acquire or a barrier.

In the basic protocol, write notices produced in a critical section is sent to the barrier manager on an arriving of barrier. However, if a page is modified only by its home node, and there is no other processors have read the page since last barrier, then it is unnecessary to send the associated write notice to the barrier manager on the arriving of barrier. To do this, a *read notice* is recorded in the home of a page any time a remote get page request is received and served. The *read notice* is then used to decided whether a write notice introdiced by the home processor is sent to the barrier manager on arriving of a barrier.

The *incarnation number* method implemented in Midway[2] and ScC[8] is adopted to eliminate unnecessary invalidation on locks. With this method, each lock is associated with an incarnation number which is incremented every time the lock is transferred. Besides, each processor maintains a local incarnation number for each lock. When a write notice is recorded in the lock, the current incarnation number of the lock is recorded as well. On a lock acquire, the acquiring processor includes its incarnation number of the lock in the acquiring message. On lock grant, the current incarnation number of the lock and those write notices which have an incarnation number larger than the incarnation number in the request is sent to the acquiring processor. The acquiring processor then sets its local incarnation number of the lock to the one received from the lock and invalidates cached pages accordingly.

3.2 Reducing Write Detection Overhead

The multiple writer protocol needs to detect writes to shared memory so that the protocol can be activated to correctly propagate the writes. Two write detection schemes are implemented in JIAJIA. One is the traditional virtual memory page fault write detection (VM-WD) scheme which identifies writes to shared pages through page faults. The other is a cache only write detection (CO-WD) scheme which does not write-protect home pages but invalidates all cached pages at the beginning of an interval.

With the VM-WD scheme, both home and cached shared pages are initially write-protected at the beginning of an interval. For a write fault on a cached page, a *twin* of the page is created and a write notice is recorded for this page in the SIGSEGV handler. Write protection on the shared page is then removed so that further writes to this page can occur without page faults. At the ending of the interval, a word-by-word comparison is performed between the written page and its twin to produce *diff* about this page. Write notices and *diff*s are then sent to the lock manager and page home respectively. If the SIGSEGV signal is caused by a write fault on a home page, then a write notice is recorded for this page and write protection on the shared page is removed. At the ending of the interval, write notices about home pages are sent to the associated lock.

The above VM-WD scheme detects writes through page faults and entails additional runtime overhead on the protocol. Our previous experiments show that, write-protecting home pages causes significant overheads for applications with large shared data set and good data distribution so that most writes hit in the home. The CO-WD scheme reduces the overhead of home pages write detection at the cost of some extra cache miss. In the CO-WD scheme, all cached shared pages are conservatively assumed to be obsolete and are invalidated at the beginning of an interval. No write detection about home pages is required in CO-WD because the purpose of detecting write notices of home pages is to maintain coherence through invalidating associated cached pages, and the CO-WD scheme has already invalidated all cached pages when starting an interval. Only writes to cached pages are detected to generate *diff*s which are sent to their home at the end of an interval. *Diff*s are generated through comparing the dirty page with its twin as in the VM-WD scheme.

3.3 Hierarchical Propagation of Barrier Messages

Barrier provides a convienent yet expensive way of inter-processor synchronization. Normally, there is a barrier manager to handle barrier requests from all processors. In the lock-based protocol of JIAJIA, the barrier manager is also responsible for combining write notices received from all processors and sending these write notices to each processor one-by-one. Processing barrier messages in the sequential way makes the barrier manager a potential bottleneck of the system when the number of processors is large.

To alleviate the bottleneck problem of the barrier manager, JIAJIA provides a tree structured propagation of barrier messages. It maps all hosts into a binary tree, with the barrier manager in the tree root. When a processor arrives at a barrier, it does not send barrier request directly to the barrier manager. Instead, it sends barrier request to its parent host in the tree. After receiving barrier requests from both of its children, the parent host combines write notices received from its children and then delivers the request to its parent. All processors arrive at a barrier when the root host receives barrier requests from both of its children. Similarly, barrier acknowledgements are propagated from the root to each node of the tree in an inverse way.

4 Performance Evaluation

The evaluation was done in the Dawning-1000A parallel machine developed by the National Center of Intelligent Computing Systems. The machine has eight nodes each with a 200MHz PowerPC 604 processor and 256MB locak memory. These nodes are connected through a 100Mbps switched Ethernet.

4.1 Applications

We port some widely accepted DSM benchmarks to evaluate the performance of JIAJIA. This paper shows the results of nine applications, include Water, Barnes[2], and LU from SPLASH and SPLASH2[17, 18], EP and IS from NAS Parallel Benchmarks[1], SOR, TSP, and ILINK from Rice University[15], and a real application EM3D from Institute of Electronics, Chinese Academy of Sciences.

EM3D is the parallel implementation of FDTD (Finite Differene Time Domain) algorithm to compute the resonant frequency of a waveguide loaded cavity. The three electronic field components, three magnetic field components, and eight coefficient components are allocated in shared space. The electronic and magnetic field components are updated alternatively in each iteration. Barriers are used for synchronization. Only ten iterations are run in our test, while the real run requires 12000 iterations.

Table 1 shows characteristics and sequential run time of the benchmarks. As indicated in the table, the 8192×8192 LU and SOR cannot be run on single machine due to memory size limitation and the corresponding sequential run times are estimated values.

4.2 Overall Performance of JIAJIA

Table 2 shows eight processor execution results of the original JIAJIA (JIA$_{base}$), JIAJIA with the write notice minimization improvement (JIA$_w$), JIA$_{wc}$ which uses the cache-only write detection scheme on the basis of (JIA$_w$), and JIA$_{wcb}$ which is the tree structured barrier message propagation version of JIA$_{wc}$. Eight-processor execution time, speedup, message counts, message amounts, SIGSEGV signal counts, and remote get page request counts are shown in Table 2. Figure 2 shows execution time breakdown of JIA$_{base}$, JIA$_w$, and

[2]To get better speedup, the tolerance parameter in `sample.in` is set to 0.2.

Table 1. Characteristics of the Benchmarks

Appl.	Size	Mem.	Barrs	Locks	S. time
Water	1728 mole.	484KB	35	65	178.00
Barnes	16384	1636KB	28	8	413.24
LU	$2048 * 2048$	32MB	128	0	84.86
LU	$8192 * 8192$	512MB	512	0	5464.80*
EP	2^{24}	4KB	1	1	49.69
IS	2^{24}	4KB	30	10	30.10
SOR	$2048 * 2048$	16MB	200	0	68.44
SOR	$8192 * 8192$	256MB	200	0	1235.76*
TSP	`-f20 -r15`	788KB	0	140	175.36
ILINK	`LGMD-1-2-3`	12MB	740	0	650.94
EM3D	$120 * 60 * 416$	160MB	20	0	65.20

$*$: Estimated from 4096×4096 LU and SOR sequential time (683.10 and 308.94 seconds respectively). Sequential time of 8192×8192 LU and SOR are unavailable due to memory size limitation.

JIA$_{wc}$ for tested benchmarks except EP and IS. Computation time, remote request service time, SIGSEGV service time, synchronization time, and other overhead are present. Other overhead in the figure is caused by factors such as processor cache miss latency, interrupt time, TLB miss time, etc.

It can be seen from Table 2 that, for most tested applications, JIAJIA achieves satisfied performance and speedup.

Both Water and Barnes are N-body problems and are characterized with tight sharing. As has been stated, the coherence protocol of JIAJIA takes all coherence related actions in synchronization points and has least message overheads in ordinary write and read operations. In Water and Barnes, the number of shared pages is small, and this small number of pages are referenced frequently by multiple processors. Hence, the overhead at synchronization point is not too high, and page faults caused by ordinary write and read operations introduce least messages in JIAJIA. As a result, compared to other software DSM systems, JIAJIA achieves an acceptable speedup in Water and Barnes.

In LU and SOR, matrices are initially distributed across processors in the way such that each processor keeps in its home the data it processes. As a result, computation-to-communication ratios of both LU and SOR are $O(N)$ where N is the problem size. Hence, speedups of both LU and SOR are acceptable and scale with the problem size. Another reason for the acceptable speedups of LU and SOR is that no *diff* production is required for home pages in home-based software DSMs[20]. Frequent inter-processor barrier synchronization contributes the main reason for the moderate speedup of 2048×2048 LU and SOR. Besides, in LU, only the updating trailing submatrix computation phase of each step is fully parallelized.

Both EP and IS have separate computation and communication phases, i.e., computation of EP and IS happen locally and involves no communication, communication happens only at the end of computation. EP achieves a speedup of 8 because the communica-

Table 2. Eight-Processor Execution Results

Appl.	**Eight-proc. Time**				**Eight-proc. Speedup**				**Message #**			
	JIA$_{base}$	JIA$_w$	JIA$_{wc}$	JIA$_{wcb}$	JIA$_{base}$	JIA$_w$	JIA$_{wc}$	JIA$_{wcb}$	JIA$_{base}$	JIA$_w$	JIA$_{wc}$	JIA$_{wcb}$
Water	27.75	26.47	30.62	—	6.41	6.72	5.81	—	11918	10828	22000	—
Barnes	71.89	68.53	70.90	—	5.75	6.03	5.83	—	37752	37018	54307	—
LU2048	26.56	25.04	24.64	24.27	3.20	3.39	3.44	3.50	25992	25992	25992	25992
LU8192	964.32	909.39	878.95	876.23	5.67	6.01	6.22	6.24	386666	386664	386616	386616
EP	6.30	6.30	6.31	—	7.89	7.89	7.80	—	77	77	77	—
IS	4.87	4.84	4.85	—	6.18	6.22	6.21	—	1050	1050	1050	—
SOR2048	33.52	21.97	11.45	11.16	2.04	3.12	5.98	6.13	8413	8411	8412	8414
SOR8192	486.21	265.64	166.20	163.18	2.54	4.65	7.44	7.58	8413	8413	8413	8413
TSP	45.99	33.25	47.65	—	3.81	5.27	3.69	—	37062	20290	39375	—
ILINK	230.24	188.63	294.15	293.60	2.83	3.45	2.21	2.22	177392	140164	249228	249228
EM3D	49.84	25.00	18.74	—	1.31	2.61	3.48	—	7602	4479	16499	—
	Message Amt.(KB)				**SIGSEGV #**				**Get Page #**			
	JIA$_{base}$	JIA$_w$	JIA$_{wc}$	JIA$_{wcb}$	JIA$_{base}$	JIA$_w$	JIA$_{wc}$	JIA$_{wcb}$	JIA$_{base}$	JIA$_w$	JIA$_{wc}$	JIA$_{wcb}$
Water	19108	16850	40125	—	5325	4892	10075	—	3385	2847	8430	—
Barnes	82099	80569	116461	—	34508	34144	39442	—	18208	17844	26487	—
LU2048	100406	99950	99830	99830	32663	32663	12072	12072	12072	12072	12072	12072
LU8192	1593850	1569946	1567908	1567908	1108876	1108875	189696	189696	189721	189720	189696	189696
EP	60	60	60	—	22	22	21	—	14	14	14	—
IS	896	896	895	—	230	230	210	—	140	140	140	—
SOR2048	23225	11837	11763	11772	412000	412000	2800	2800	2800	2800	2800	2800
SOR8192	91997	46146	46135	46162	164080	164080	2800	2800	2800	2800	2800	2800
TSP	59382	24265	64025	—	16530	8312	17194	—	14082	5682	15231	—
ILINK	632606	418714	928756	928756	95870	86758	117709	117709	73191	54577	109109	109109
EM3D	19809	8282	33250	—	186071	184510	9830	—	3512	1690	7970	—

tion and computation ratio of EP is low. In IS, we keep the number of buckets at 1024 which makes the communication amount relatively small compared to the computation work of counting 2^{24} keys. As a result, a speedup of 6.2 is achieved.

TSP specializes in that all inter-processor synchronizations are taken through locks. In TSP, each processor frequently reads from and writes to the pool of tours and the priority queue, causing tight sharing of pages (a page can store 27 paths in TSP[15]). As a result, when entering a critical section, a cached page is normally invalidated because it has been written by other processors. Again, the speedup of TSP is acceptable because the lock-based cache coherence protocol of JIAJIA has least overhead on ordinary read and write miss.

The speedup of ILINK is not so high compared to other software DSM systems such as TreadMarks[5]. We ascribe the main reason to the relatively little problem scale in our evaluation (LGMD-1-2-3 vs. LGMD-2-10-7) and slow communication (100Mbps Ethernet) to computation (0.4GFLOPS single node performance) ratio in our environment. Besides, the difference of memory organization between JIAJIA and TreadMarks may also contribute the reason.

4.3 Effect of Reducing False Sharing

Table 2 shows that, JIA$_w$ outperforms JIA$_{base}$ significantly ($18\% \sim 50\%$) in SOR, TSP, ILINK, and EM3D, slightly ($5\% \sim 6\%$) in Water, Barnes, and LU, while the difference between JIA$_{base}$ and JIA$_w$ is trivial in EP and IS.

In LU and SOR where each processor only writes to the part of data kept in its home, the *read notice* optimization technique contributes the main reason for the better performance of JIA$_w$ than JIA$_{base}$. It can be seen from Table 2 that, in LU and SOR, JIA$_w$ and JIA$_{base}$ have the same number of messages, SIGSEGVs, and remote get page requests. However, JIA$_w$ transfers less message amounts than JIA$_{base}$ because with the *read notice* technique, JIA$_w$ only sends write notices of boundary pages to the barrier, while JIA$_{base}$ sends write notices of all home pages to the barrier. Figure 2 also shows that the synchronization overhead of JIA$_w$ is much less than that of JIA$_{base}$ in LU and SOR.

The incarnation number technique makes JIA$_w$ significantly outperforms JIA$_{base}$ in TSP. In JIAJIA, write notices in all locks are cleared only on a barrier which is not the synchronization method in TSP. As a result, the tight sharing pattern of TSP causes write notices of a lock accumulate as more releases are performed, and some cached pages of a processor are unnecessarily invalidated on an acquire of the lock in JIA$_{base}$. In JIA$_w$, on the other hand, the incarnation number technique of JIA$_w$ helps to reduce unnecessary invalidations because only write notices with an incarnation number larger than the acquiring proces-

Figure 2. Breakdown of Parallel Execution Time

sor's local incarnation number of the lock are sent to the acquiring processor. It can be seen from Table 2 that, JIA$_w$ has much less remote get page requests than JIA$_{base}$ in TSP. As a result, JIA$_w$ has less SIGSEGV overheads than JIA$_{base}$, as indicated in Figure 2.

Statistics in Table 2 shows that remote get page requests and message amounts of JIA$_w$ are less than that of JIA$_{base}$ in ILINK and EM3D in which barrier is the mere synchronization method. The less number of remote accesses of JIA$_w$ than JIA$_{base}$ is caused mainly by the optimization which keeps a cached single-writer page valid on the writing processor on a leaving of barrier. Besides, with the *read notice* technology, JIA$_w$ sends much less write notices than JIA$_{base}$ on barriers, causing the synchronization time of JIA$_w$ to be much less than that of JIA$_{base}$ in ILINK and EM3D. Figure 2 shows that both SIGSEGV time and synchronization time of JIA$_w$ is much less than that of JIA$_{base}$ in ILINK and EM3D.

4.4 Effect of Cache-Only Write Detection

It can been seen from Table 2 that, CO-WD outperform VM-WD in LU, SOR, and EM3D, while VM-WD outperforms CO-WD in Water, Barnes, TSP, and ILINK.

In LU and SOR, matrices are distributed across processors in a way that each processor only writes to its home part of the matrices in the computing. Since the computation of an iteration is synchronized with barriers and passing a barrier causes all shared pages to be write-protected in VM-WD, page fault occurs for writing each home page in an iteration. The CO-WD scheme, on the other hand, does not write protect shared pages on a barrier, and writing to home pages of a processor can process smoothly without any intervention. Table 2 shows that the CO-WD scheme causes much less SIGSEGVs than the VM-WD scheme, though remote get page request numbers are same for VM-WD and CO-WD in LU and SOR. It can be derived from Table 2 that, JIA$_{wc}$ requires 37% $\sim$ 48% less time than JIA$_w$ in SOR, and 2% $\sim$ 3% less time than JIA$_w$ in LU. Figure 2 shows that JIA$_{wc}$ has much less SIGSEGV overhead than JIA$_w$ in LU and SOR.

In Water, Barnes, TSP and ILINK, the number of shared pages is not large and hence the advantage of CO-WD over VM-WD in keeping home pages writable on a synchronization point is not significant. On the other hand, since CO-WD invalidates all cached pages on an acquire, it causes much more remote page faults and consequently messages than VM-WD for applications with irregular memory reference pattern. As a result, both the SIGSEGV time and server time of JIA$_{wc}$ is large than JIA$_w$ in these applications, as indicated in Figure 2.

In EM3D, CO-WD also introduces more remote page faults than VM-WD. However, since EM3D requires 160MB shared memory, VM-WD causes much more SIGSEGVs than CO-WD. Figure 2 shows that the overhead of additional virtual memory page faults in VM-WD is similar to that of the additional getting remote pages in CO-WD. CO-WD performs better than VM-WD mainly because it has less other overheads in EM3D.

4.5 Effect of Hierarchical Barrier Message Propagation

Since the improvement of JIA$_{wcb}$ over JIA$_{wc}$ is the hierarchical propagation of barrier messages, JIA$_{wcb}$ is tested only for applications with more than 100 barriers.

As can be seen from Table 2, JIA$_{wcb}$ outperforms JIA$_{wc}$ slightly (around 1% $\sim$ 3%) for all applications we tested. This slight improvement is acceptable because the advantage of JIA$_{wcb}$ over JIA$_{wc}$ is not significant for eight processors. In our test of passing 1000 barriers, JIA$_{wcb}$ finishes in about 3 seconds while it takes JIA$_{wc}$ about 6 seconds to finish. It is expected that the difference between JIA$_{wcb}$ and JIA$_{wc}$ will be more significant when more processors are involved.

5 Conclusions and Future Work

The above evaluation and analysis can be summarized as follows.

The methods of reducing false sharing through minimizing the number of write notices propagated between lock manager and lock releaser (acquirer) is effective for different kinds of applications.

CO-WD works well for applications with good data distribution and most writes hit in home pages, such as LU and SOR. However, it is sensitive to data distribution and performs terribly bad when most shared memory reference happens in cache, as in TSP.

Tree structured barrier message propagation performs slightly better than propagating barrier messages sequentially for all applications tested. It is expected to bring more advantages when more processors are involved.

JIAJIA achieves satisfied speedup for most applications. This is mainly because the simplicity of the lock-based cache coherence protocol entails little system overheads. Other factors, such as the home-based memory organization which requires no *diff* generating for home pages[20], the uniformed local and global addresses mapping of JIAJIA, and the flexible API which allow the programmer to control initial distribution of shared data, also help to improve performance.

With the NUMA-like memory organization, JIAJIA can combine memories of multiple processor to form a large shared memory.

Our future work include improving JIAJIA system, optimizing the lock-based cache coherence, and developing a hardware prototype to implement the protocol. Further information about JIAJIA is available at `http://www.ict.ac.cn/chpc/index.html`.

Acknowledgement

The authors are greatly indebted to Professor Liviu Iftode for his constructive comments on the earlier version of this paper. An allocation of computer time from the National Center of Intelligent Computing systems (NCIC) of China is gratefully acknowledged.

References

[1] D. Bailey, *et. al.*, "The NAS Parallel Benchmarks", *Technical Report 103863*, NASA, July 1993.

[2] B. Bershad, *et. al.*, "The Midway Distributed Shared Memory System", in *Proc. of the 38th IEEE Int'l CompCon Conf.*, pp. 528–537, Feb. 1993.

[3] R. Bianchini, *et. al.*, "Hiding Communication Latency and Coherence Overhead in Software DSMs", in *Proc. of ASPLOS'96*, Oct. 1996.

[4] J. Carter, *et. al.*, "Implementation and Performance of Munin", in *Proc. of the 13th ACM Sym. on Operating Systems Principles*, pp. 152–164, Oct. 1991.

[5] S. Dwarkadas, *et. al.*, "Parallelization of General Linkage Analysis Problems", *Human Heredity*, pp. 127-141, 44(1994).

[6] K. Gharachorloo, *et. al.*, "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors", in *Proc. of ISCA'90*, pp. 15–26, May 1990.

[7] W. Hu, *et. al.*, "A Lock-based Cache Coherence Protocol for Scope Consistency", *Journal of Computer Science and Technology*, Vol. 13, No. 2, pp. 97–109, Mar. 1998.

[8] L. Iftode, *et. al.*, "Scope Consistency: A Bridge Between Release Consistency and Entry Consistency", in *Proc. of the 8th Annual ACM Sym. on Parallel Algorithms and Architectures*, June 1996.

[9] P. Keleher, *et. al.*, "TreadMarks Distributed Shared Memory on Standard Workstations and Operating Systems", in *Proc. of the 1994 Winter Usenix Conf.*, pp. 115–131, Jan. 1994.

[10] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models", in *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, pp. 91–98, May 1996.

[11] D. Khandekar, "Quarks: Distributed Shared Memory as a Building Block for Complex Parallel and Distributed Systems", Master's thesis, Department of Computer Science, The University of Utah, Mar. 1996.

[12] G. Lathtop, *et. al.*, "Strategies for Multilocus Analysis in Humans", PNAS 81(1994), pp. 3443–3446.

[13] D. Lenoski, *et. al.*, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessors", in *Proc. of ISCA'90*, pp. 148–158, June 1990.

[14] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", in *Proc. of ICPP'88*, Vol. 2, pp. 94–101, Aug. 1988.

[15] H. Lu, *et. al.*, "Quantifying the Performance Differences Between PVM and TreadMarks", *Journal of Parallel and Distributed Computing*, Vol. 43, No. 2, pp. 65–78, June 1997.

[16] A. Schaffer, *et. al.*, "Avoiding Recomputation in Genetic Linkage Analysis", *Human Heredity*, pp. 225–237, 44(1994).

[17] J. Singh, *et. al.*, "SPLASH: Stanford Parallel Applications for Shared Memory", *Computer Architecture News*, 20(1):5–44, March 1992.

[18] S. Woo, *et. al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations", in *Proc. of ISCA'95*, pp. 24–36, 1995.

[19] M. Zekauskas, *et. al.*, "Software Write Detection for a Distributed Shared Memory", in *Proc. of the 1st Int'l Sym. on Operating System Design and Implementation*, pp. 87–100, Nov. 1994.

[20] Y. Zhou, *et. al.*, "Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems", in *Proc. of the 2nd USENIX Sym. on Operating System Design and Implementation*, Seattle, Oct. 1996.