ORION: An Adaptive Home-based Software DSM

M.C. Ng
School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119620
ngmingch@comp.nus.edu.sg

W.F. Wong
School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119620
wongwf@comp.nus.edu.sg

ABSTRACT

As more and more software DSM systems with their unique APIs surface, it becomes imperative for the industry to come up with a standardized API to facilitate users in using different types of DSM systems. A multithreaded software DSM, Orion, has been developed to provide POSIX-thread (pthread) like API which avoids creating another unique set of API and helps in porting pthread programs to a distributed environment. Orion implements home-based consistency model, which is a recent development in the DSM field that has open up many areas for further research and development. In this paper, we also present 2 adaptive schemes for home-based DSM systems: home migration and dynamic adaptation between write-invalidation and write-update protocols. The two fully automatic schemes aim to involve minimal user intervention and yet deliver good performances with some speedups ranging from 2% to 79% observed in some 8 benchmarks tested.

Keywords

Distributed shared memory, virtual shared memory, release consistency, home-based release consistency and multithreading.

1. INTRODUCTION

It would not be an exaggeration to say that in the software DSM community today there are no less than 10 systems around with each bearing its own API. While a fair proportion of the functions in these APIs are by and large the same in functionality, conformity is still lacked in them. Though this problem is not new, it was not brought up openly for more attentions to be given to until recently when OpenMP is speculated to be the API for software DSM in the future. A closely related effort, which started 2 years ago, was undertaken by us to develop an API that conforms to the *POSIX-thread* (*pthread*) standard. The reason for selecting pthread, apart from avoiding creating yet another unique set of API, was the multithread trend that was going through the industry. Pthread, unlike HPF and other highly specialized languages, is used in more varieties of software that are not necessarily scientific in nature, such as a web server running on a dual-processor system. The net result is thus a multithreaded software DSM system that provides pthread-like API which we dubbed it *Orion*. As of now, we had implemented about 50% of the complete repertoire of pthread functions that are the commonly used and essential ones.

In the course of development we had also focused on building 2 adaptive schemes for home-based software DSM systems because Orion implemented home-based eager release consistency. One of the major challenges persistently faced by the community is the cost of network communication necessitated by the protocol as well as induced by false sharing. Basically, the cost of network communication can be broken down into two factors:

- the number of messages transacted;
- the amount of data transferred

The advent of the release consistency (RC) was a major milestone in the history of software distributed shared memory. It reduces the network data and its natural support for multiple writer protocol mitigates the problem of write false sharing. Of the two, the former is more costly in terms of performance loss. To these we can add a third – *the timeliness of data*. Software DSMs rely on the use of signal handling mechanisms that are costly. If the data is available when it is needed, this can be reduced. This probably explains why in certain applications, the write update protocol will perform better than the write-invalidate protocol even if more data is transferred. We shall see how timeliness of data can be improved in the later part of the paper.

One of the important variant of RC is the *home-based release consistency* model (HRC). Unlike in the traditional RC model, each page in a HRC has an assigned home. It should be noted that the discussion of RC here is limited to the write-invalidate RC models. The main advantage of HRC over RC is that after communicating diffs to the homes, they can be discarded. In contrast, RC needs to use an aging mechanism to ensure that diffs do not occupy too much storage. However in HRC, sending diffs home is mandatory which can be a performance-loss factor. Published data have shown that for certain applications lazy release consistency model (LRC) performs better than home-based lazy release consistency model (HLRC) [4], while for some other applications the reverse is true [2]. The main cause for this difference is the choice of the home for a page, which varies from application to application. This points to the need for the system to dynamically adapt to the needs of the applications in order to achieve better performances.

The idea of automatic adaptive protocol is not new. As a matter of fact, many automatic adaptive schemes [5,6,7,8,9,10,11] have been proposed. These include adaptation between the single-writer and multiple-writer protocols, adaptation between the write-invalidate and write-update protocols, and even process/thread migration. In this paper, besides introducing our software DSM system, we have also proposed 2 adaptive schemes for home-based software DSM. We have identified that HRC exhibits an important feature that aids adaptation. The home of a page serve as a natural point from which the data access pattern information for that page can be collected. The adaptive schemes that we are proposing take advantage of this feature. A minor enhancement called the *partial page update* is also proposed in the bid to reduce sending unnecessary network data as a result of the typical entire page transfer upon receiving a page request at the home node. In the rest of the paper, we shall outline our proposals and present some experiment data on their effectiveness.

2. ORION

This section briefly introduces our software DSM system, Orion, which is designed to provide a pthread-like API. It currently implements home-based eager release consistency (HERC) supporting the multiple writer, write-invalidate and write-update protocols. Typically in an Orion application, user codes interfaces with the Orion API, which in turn makes use of functions from pthread and *data passing interface* (DPI) libraries.

An Orion program typically starts with a number of nodes or hosts specified by the user. Depending on the number of nodes specified, the master node that is ranked 0 remotely spawns the same program on other nodes and assigns them a different rank each. The DPI library undertakes the task of automatic remote process spawning during startup. Previously, *message-passing interface* (MPI) was used but it was eventually discarded for its lack of multithread support and significant loss of performance due to its internal overhead. DPI is a separate network communications library written to provide the basic send, buffered send and receive functions found in MPI. Some group communication functions like barrier, all-gather and other functions are provided too. Each message carries a tag that is also the thread id of the receiving thread. Most DPI functions use TCP/IP (i.e. Transport Control Protocol/Internet Protocol) to ensure reliable and ordered delivery of large data. Since TCP is a point-to-point communication protocol, a TCP link from one node to another is being set up during startup. UDP/IP (i.e. User Datagram Protocol/Internet Protocol) is mainly used for multicasting in situations where it is critical to send one short message through the network to inform all remote nodes to proceed with some tasks simultaneously.

User Codes		
Orion Layer		
POSIX-Thread (Pthread) Library	**Data Passing Interface**	
	Transport Control Protocol (TCP) Layer	**Universal Datagram Protocol (UDP) Layer**
	Internet Protocol (IP) Layer	

Figure 1: The software layers of Orion

Currently, Orion allows user to use up to 20000 of virtual pages for shared memory that is equivalent to 157MB on machine that has page size of 8192 bytes. Internally, another 20000 are reserved for twins that are necessary for supporting multiple-writer in the system.

2.1 Overview of the Internals of Orion

Orion has one functional unit that is actively running in background as a daemon thread. It is responsible for maintaining memory consistency, managing distributed threads and many others. Each node has a separate running copy of this thread termed as the *service thread* (ST). Generally, server-client architecture is adopted for the network communication model in the entire system. The servers are the STs while the clients can either be the user threads or some remote STs. An example of a user thread communicating with an ST is lock acquisition. In Orion, a mutex (i.e. lock) has a designated ST that is responsible for managing it. When a client wishes to lock or release a mutex, a message must be sent to the ST. On the other hand, ST-ST type of communication is usually oblivious to the users. An example of it is home migration of a page. When an ST decides to shift the home of a page to another node, it will inform the latter's ST of the change.

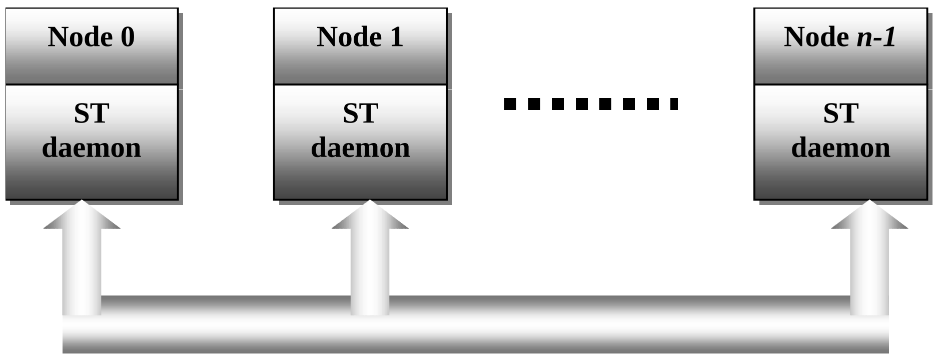

Figure 2: The ST daemon

Some services provided by ST include

1. Thread management (spawning and joining),
2. Mutex management (creation, locking and unlocking),
3. Condition variable management (creation, waiting and signalling),
4. Barrier management,
5. Page management (page request, home migration of page, write protocol adaptation of page), and
6. Page statistics collection (remote diffs update monitoring and page request monitoring).

2.2 Maintaining Shared Memory Consistency

In simple terms, maintaining shard memory consistency is about delivering the correct data (i.e. changes) to the correct place at the correct point of time. Changes are write operations. In page-based software DSM systems, a change really means a modification of a page. To detect whether a page has been modified, shared pages are initially write protected so that a page fault signal can be caught when an initial write is performed on it.

Since Orion implements HERC, every page has a globally appointed home. If a write fault is detected, the page index is added to *orion_modified_page*, a list of modified pages, and a twin of the page is created and kept. Write permission is granted to this page so that any subsequent write to the same page will not generate any page fault.

When a thread performs mutex operation or barrier rendezvous, some actions are taken to maintain shared memory consistency. These are essentially the synchronization points. Firstly, diffs are identified or extracted by doing a word-by-word comparison between the modified page and its corresponding twin. Diffs are sent back to the designated home where they are applied. The ST of the home receives the diffs, decodes and applies them to the correct position in the shared memory. If the home of a page is the node itself, then diffs extraction for the page is skipped.

The second step to be performed when a synchronization point is reached is the dissemination of invalidation messages (a.k.a write notices). Invalidation messages can only be sent when the diffs are successfully been applied. Thus in this respect, all the homes that receive diffs must acknowledge when they have applied. This must be observed to avoid possible data race conditions. Invalidation messages can be dispatched once acknowledgements have been received. The ST of each process is responsible for receiving the invalidation messages and making those pages specified in the invalidation message invalid if they are not based locally.

Reading an invalid page will cause the page fault handler to send a request to the home of the page. The ST of the home then responds by sending only the modified region of the requested page over to the requester since Orion implements partial page update enhancement, which will be explained later. Upon receiving the update, read permission is then granted to the page. A write operation performed on a write-protected page involves adding an entry to the list of modified pages, twin creation and making the page write-permissible. A write to an invalid page is treated as a read to an invalid page and then a write to a write-protected page. Hence, it involves 2 page faults.

2.3 Relative Performance of Orion

This section accesses the performance of Orion in comparison to *Treadmarks* [12] which implements lazy release consistency. A total of six benchmarks from Treadmarks and NAS [15] were executed on 4 nodes of the 32-node Fujitsu AP3000. They are the successive-over-relaxation (SOR), Gauss Elimination (GAUSS), 3D Fast Fourier Transform (3D-FFT), integer sorting (IS), conjugate gradient (CG) and multigrid (MG). Each node in the experiment featured an UltraSparc 143-MH $_z$ processor and 128MB of physical memory. The nodes were connected via a high-speed network capable of delivering data at 200MB/sec. Table 1 shows some micro-benchmarks and the costs of some basic operations in Orion.

Micro-benchmark	Min. cost (*u*sec)	Basic operation	Cost (*u*sec)
Page transfer bandwidth	18 MB/sec	*mprotect* (8192)	49
Page fetch	2456	*memcpy* (8192)	58
Barrier (12 nodes)	7098	Thread suspension	628
Remote lock acquire	625	Thread continuation	28
		Read fault	250
		Write fault	250

Table 1: Micro-benchmarks and costs of basic operations in Orion

Table 2 shows the problem sizes used and the execution times during the comparison. The initial home assignment in this experiment was round robin.

Applications	Sizes/iterations	Treadmarks (sec)	Orion (sec)
SOR	1024X2048, 100	108.520	111.560
GAUSS	512X512, 512	5.664	7.181
3DFFT	32x32x32, 100	25.090	25.600
IS	22x16, 50	30.986	31.021
MG	32x32x32, 5	1.46	1.58
CG	14000x14000	45.168	45.820

Table 2: Relative performance of Orion

3. ADAPTIVE PROTOCOLS FOR HRC SOFTWARE DSM

In one of the study conducted by the Treadmarks team [4], it was pointed out that the choice of home was a crucial performance-determining factor. Some HRC systems allow the users to specify how the homes should be distributed before execution. This strategy depends on the user 's expertise and will continue to work satisfactorily if the nature of the application 's memory references remains unchanged or static. However, in designing an efficient HRC, there are 2 things that should be avoided. They are firstly *improper initial home assignment*, and secondly *static home assignment*. Hence, if the application's memory access pattern changes, regardless of how perfect the initial home placement may be, performance will be dented.

There are three possible ways to improve the initial home assignment, as listed below:

1. *User intervention*. User provides information on home assignment prior to execution.
2. *First touch home assignment* . Whichever is the first node to write to a page during execution will automatically be the home of that page.
3. *Compiler inference* . A modified compiler, after analyzing the access pattern of an application, instructs the software DSM system on how to assign home prior to execution.

Intervention from the user and first touch home assignment method are well understood, while compiler inference technique remains an area for active research. However, a good initial home assignment is not sufficient to promise good performance in applications with dynamic access patterns. Therefore, it is the issue of automatic adaptation that we wish to explore.

The basis upon which our adaptive schemes work is none other than identification of certain memory access patterns (MAP). In most modern scientific applications, repetitions or loops of codes are common. This makes the MAP fairly predictable. Hence, knowing when and which process will need the data will contribute to better data distribution and better performance. For shared pages, there can be multiple producers that produce some data shared by either zero, one or many consumers (i.e.

demand). We limit our discussion here to one producer. If this supply-and-demand MAP is kept stable, adaptive schemes can easily be formulated. We have categorized the MAP of an application from the *producer's point of view* into 4 main groups:

i) *zero consumer,*

ii) *one persistent consumer,*

iii) *multiple persistent consumers, and*

iv) *non persistent consumer*

Based on these 4 groups of MAPs, we have developed 2 adaptive schemes for HRC.

3.1 System Statistics and Parameters

The MAP monitoring procedure, performed at the home node, primarily involves keeping track of some system statistics. This MAP monitoring is performed on every node to keep track of some statistics of pages that based locally (i.e. the home of this page is the node itself). The statistical information involved on the adaptive actions triggering procedures are:

- *Remote update count* (RUC) for each memory page. It is an array with each element, *k*, keeping track of the number of times updates/ diffs from node *k* have been received since the last local access at home (i.e. the page has been accessed by the home node itself).
- *Page access count* (PAC) for each memory page. PAC includes both the local and external access. An external access refers to the remote request for a fresh copy of a page from the home. It is an array with each element referring to a remote node.

To capture absolute and accurate local access pattern at home, read accesses need to be detected as well. This is not required in a traditional DSM system. Typically in a home-based DSM system, pages based locally are either write-protected or read/write accessible. However, to detect the initial local access to an updated page, pages that are based locally have to be made write/read inaccessible. This incurs additional page faults that are serious detriments to performance. The alternative is thus to capture only initial write access which does not incur additional page faults and helps in providing hints to the DSM system on the local access pattern, which may not be absolute but sufficient for adaptation schemes to work with. Our software DSM system adopts the second approach.

The system parameters include:

- *Page request sampling period* for each memory page. Within this sampling period, the PAC of each page is accumulated upon each page request. If the period is set as *Y*, then the Y^{th} page request is considered the end of a period and the PAC is reset after appropriate adaptive actions have been taken, if necessary. Local access is also considered a page request.

	Events	RUC of page *k*	PAC of page *k*	Page *k* access permission	Page request sampling period	Conditions & actions
1	Local access detection (node 0)	Reset all elements to 0	Increment $PAC_{[0]}$ by 1	Change page access permission	Increment by 1	NA
2	Remote diffs update by node *M* (*M*≠0)	Increment $RUC_{[M]}$ by 1	NA	Set page access permission to inaccessible (to detect local access)	NA	If $RUC_{[M]}$ exceeds HMTL and write policy is write-invalidate protocol, event 4 is triggered off.
3	Page request by node *M* (*M*≠0)	NA	Increment $PAC_{[M]}$ by 1	NA	Increment by 1	If page request sampling period for page *k* ends, event 5 is triggered off.
4	Home migration	Reset all elements to 0	NA	Remove read protection	NA	Home is migrated to node *M*
5	Write protocol switch and home migration evaluation	NA	Resets all elements to 0 after evaluation	NA	Reset page request sampling period to 0 after evaluation	If $PAC_{[0]}$ is non-zero and $PAC_{[M]}$/sampling period exceeds WPTL, include node *M* is partial update list. If $PAC_{[0]}$=0 and $PAC_{[M]}$/sampling period exceeds HMTL, migrate home to node *M*(*only one non-zero PAC value*).

Table 3: System statistics and events

- *Home migration triggering limit* (HMTL). This is the limit set to activate the home migration adaptive action.
- *Write protocol switch triggering limit* (WPTL). This is the limit set to activate the write protocol switch adaptive action.

Assuming that the events are happening to page *k* whose home is node 0, Table 3 shows a tabular summary on the collection of system information and the conditions that trigger off the adaptive actions.

3.2 Home Migration

It is quite obvious that the HLRC has one serious shortcoming. That is the choice of home for a page. Home migration is an adaptive scheme we propose to reduce unnecessary network traffic. By monitoring the MAP, we can make intelligent guess of which pages are frequently or seldom fetched by other processes.

3.3 Dynamic Adaptation Between WI and Partial WU protocols

Write protocol switch is an adaptive scheme that improves the timeliness of data or data availability. Typically, a page fault involves a round-trip to the home that is expensive. This scheme aims to improve performance through cutting down the time spent on handling page fault and the half round-trip time during which a page request message is sent to the home.

HRC sends out invalidation messages after diffs have settled at their homes. Note that diffs are simply updates and it is just a convenient extension to the consistency model by sending diffs to all nodes to implement write-update (WU) protocol. Subsequent invalidation messages should therefore exclude pages that are maintained by WU protocol. Our scheme switches the write protocol of a page between write-invalidate and *partial write-update* (PWU) that allows only selective nodes to be updated simultaneously with the home. A switch occurs when the percentage of PAC of one node within the page request sampling period exceeds the threshold level. This node will be included in an update list that is replicated on every node. This approach also eliminates sending unnecessary page updates to nodes that may not require them, as would happen in full WU protocol. Note that full WU protocol is actually a subset of partial WU protocol and hence, if the page is not maintained in full WU protocol, it should be included in the invalidation messages as well.

Switching the write protocol of a page from PWU protocol back to WI protocol is easy. The non-trivial part is deciding when to do it. When a page is in partial WU protocol, nodes in the update list do not go to the home to fetch the page anymore, and as such the home does not have enough information to decide if the write protocol of the page should revert to WI. One possible way is the nodes that are receiving updates, apart from the home, can voluntarily request the home to drop itself from the update list if the node sees that it is unproductive for diffs to be sent to itself. The way to determine if such an action is necessary is described as follows. Suppose node *M* is listed in the update list for a page *X*. Each time node *M* (i.e. not the home) receives diffs from node *N*, it keeps track of the $\text{RUC}_{[N]}$ value of the page, which reflects the number of times a page has been updated by node *N* since the last local access. If this counter exceeds a certain threshold level, the node can request the home node to drop *itself* (i.e. node *M*) from the update list. All nodes will be informed of the change in the write policy change initiated by the home node later. If a local access is made on node *M*, the entire array of RUC counters should be reset to zero.

3.4 One Producer-Zero Consumer MAP

Next we shall see how the different adaptive schemes relate to the different MAPs. As an illustration, suppose there are 4 nodes as shown in figure 3. Node 0 is actively writing to page *X* that is home-based at node 2. Each time it reaches a synchronization point, diffs are sent to node 2 but page *X* is never or seldom fetched by other nodes (including node 2 itself).

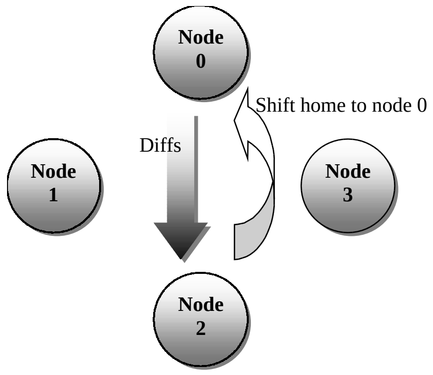

Figure 3: One producer-zero consumer MAP

Clearly, sending diffs of page *X* from node 0 to node 2 is unproductive, and shifting the home to node 0 will definitely result in lesser redundant network communication. In such a case, node 2 can unilaterally decide to shift the home for page X to node 0 when it finds that the $\text{RUC}_{[0]}$ of page *X* exceeds HMTL. This MAP is commonly found in applications like a benchmark called SOR in which one node only uses some 'edge' values among the blocks of values computed by other nodes. Those non-'edge' values are produced by one node, but useless to the others.

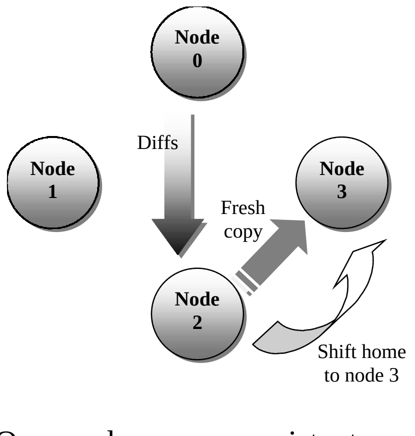

Figure 4: One producer-one persistent consumer MAP

3.5 One Producer-One Persistent Consumer MAP

Consider a slightly different scenario in which node 3 persistently requests for fresh copies of page *X* from node 2. Node 2, which does not need diffs from node 0, acts as a buffering zone. Based on the high $PAC_{[3]}$ (i.e. page access count value for node 3) and zero $PAC_{[home]}$ (i.e. the PAC value for home) for page *X*, node 2 can shift the home of node 3.

3.6 One Producer-Multiple Persistent Consumers MAP

The one producer-multiple persistent consumers (i.e. more than 1 persistent consumers) MAP should be tackled by dynamic adaptation between WI and partial WU protocols since home migration is not suitable in this case. Figure 5 shows the scenario where node 0 is the producer while the other nodes are consumers. In this case, we make a switch to full WU based on the heuristics described earlier.

3.7 One Producer-Non Persistent Consumer MAP

Lastly, the recommended approach for dealing with the situation of one producer-non persistent consumer is not to do anything, i.e. maintain the page in WI protocol. In fact, this approach is inherently built-in when the system periodically evaluates whether the page should be maintained in WI or partial WU protocol based on the PAC elements of the remote nodes. Since most consumers or nodes do not consistently request for the page, the average PAC elements will be low. Therefore, no change of write protocol is required.

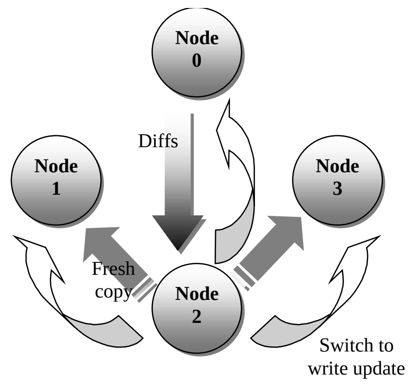

Figure 5: One producer-multiple persistent consumers MAP

3.8 Partial Page Update

Presently, a page request results in a transfer of one physical memory page size of data across the network. In some cases, differences between the present and previous versions of a page are minor. In view of this shortfall, the HRC DSM system may be designed to send the *modified region* of a page instead of the entire page upon request. However, there are some additional trivial steps involved during diffs reception and page retrieval procedures. For a start, each page will have 2 associated indices pertaining to each foreign node. One index called the *start index* points to the start of a modified region, while the other one called the *end index* points to the last byte of the modified region. The start and end indices are initialized to some values such that the latter is greater than the former. This means that no change has been made to the page and therefore, if a page request for this page is received, the page content is not sent.

There are 2 situations in which the indices will be modified. The first situation is during diffs reception. When the home decodes the diffs and applies them to the respective page, it will know the starting and ending points of the modified region specified by the diffs. Note that the diffs are arranged in the order of address. If the starting point is address-wise lower than the start index, then start index is set to the starting point of the modified region. If the end index is address-wise lower than the ending point, then the end index is also changed. Hence, the start and end indices grow upward and downward respectively to the top and bottom of a page, but never exceed them. Note that the same procedure is repeated for the pair of indices of each foreign node entry pertaining to the page.

The second situation is when a write fault is detected on a page that is based locally. In this case, the start and end indices are set to 0 and the standard page size minus 1 (i.e. entire page is modified). This is done for efficiency reason. Since the page is based locally, there is no need to extract diffs during synchronization, which is otherwise performed on other nodes. If diffs extraction is performed on pages that are based locally just to update the start and end indices, time will be wasted. Hence, it is recommended that if a page is modified by the home node, then the entire page is sent as in a standard home-based consistency model. Again the same procedure is repeated for the pair of indices of each foreign node entry pertaining to the page.

During page retrieval, the home picks out the pair of indices belonging to the requesting node. It only sends the offset and the content of the modified region as hoarded by the start and end indices, which are then reset to values such that end index is greater than the start index if the access permission of the page is not read/write acccessible.

This technique involves little overheads such as updating the array of start and end indices of a page during diffs reception. In terms of network transferred data, 2 more bytes of overhead necessitated by the offset indication per respond to page request will be incurred in exchange for the benefit of sending only the modified region that is always smaller or equal to the size of a page.

4. PERFORMANCE OF THE 2 ADAPTIVE SCHEMES

We had tested the proposed schemes using Orion. Eight benchmarks were used this time round. The 2 additional benchmarks were Barnes-Hut, and Water from the SPLASH [16] suite. The data sizes used and execution statistics of the benchmarks under the no adaptive scheme influences are shown in Tables 4 and 5 respectively.

The hardware and OS platforms remain the same as in the comparison experiment between Orion and Treadmarks. The main objective of this benchmarking process is to verify and to show how well the two adaptive schemes can improve the performance.

The process does not involve any user intervention in providing clues for the DSM system in initial home assignment. The initial home assignment is round-robin assignment.

Applications	Size (# of iterations)	Synchronization point
SOR	1024X2048 (100)	201 barriers
GAUSS	1024X1024 (1024)	1026 barriers
FFT	128X128X128 (10)	24 barriers
IS	22X17 (50)	252 barriers
WATER	512 (10)	73 barriers, 3860 locks
BARNES-HUT	65536	20 barriers
MG	128X128X128 (5)	124 barriers
CG	14KX14K (10)	792 barriers

Table 4: Problem sizes and iterations of benchmarks

Applications	Execution time (sec)	Amount of data transferred (MB)	No. of msg	No. of page request
SOR	334.008	59	175762	1824
GAUSS	2083.563	670	225399	5513
FFT	133.220	112	18951	4898
IS	201.966	108	24992	8698
WATER	332.490	27	152138	2998
BARNES-HUT	52.760	32	12733	4227
MG	80.250	50	13729	1477
CG	259.877	90	30779	10615

Table 5: Statistics of benchmarks executed without adaptive actions

4.1 Results

Table 6 summarizes the main performance results of the proposed scheme. It has been observed that, compared to the basic home-based DSM system without adaptive action, a speedup ranging from 2% to 79% was observed.

In the case of SOR, GAUSS and MG, reductions in the overall network traffic brought upon the expected speedups. Interestingly, in some applications like FFT and IS, increment in amount of data transferred or number of transacted messages did not cause them to perform any worse. In general, a reduction in network traffic should give rise to better performance, but the converse may not be true. Consider the time spent on waiting for diffs or pages can substantially be reduced if they are already available when they are needed. This is the key rationale to page and diffs prefetching techniques. The dynamic write protocol adaptation scheme we proposed can potentially increase network traffic in Orion. The reason is in Orion the size of the diffs extracted from a modified page can be greater than the page size since the diff granularity is set at 2 bytes. If a page contains a part of a large array of 4-byte integers, and the changes to the elements are small, the problem of large amount of diffs can very well arise. Suppose a page frequently accessed by some nodes is maintained under write-update protocol, more data is thus sent. The primary aim of this adaptation is to reduce external page request.

It has also been noted that the adaptive schemes in Orion could not effectively handle applications with migratory memory access pattern as exhibited in IS and WATER. The reason is the schemes could not react fast enough to take appropriate actions such as home migration. Suppose in one phase a page is only exclusive accessed by one node and in the next phase which is only one synchronization point away, it will be accessed by another node. Collection of access pattern at the home node cannot pinpoint the best home for that page. Hence, more often than not, the write policy adaptation scheme eventually kicks in and results in maintaining partial write-update protocol for the page that is not entirely appropriate. Admittedly, our adaptive schemes in Orion are by no means complete to cover all types of access patterns. One of the future work will be to handle migratory data through adaptation between home and homeless protocol or between single- and multiple-writer.

Applications (executed with adaptive actions)	Speedup (%)	Reduction in data transacted(%)	Reduction in no. of msg (%)	Reduction in external page request (%)	No. of home migration	No of write protocol change
SOR	42	87	36	44	775	9
GAUSS	79	96	83	24	387	7
FFT	6	7	-7	47	5291	2033
IS	9	-63	-6	94	65	187
WATER	4	17	-17	80	142	78
BARNES-HUT	2	-27	-9	60	763	2842
MG	40	50	23	40	3719	400
CG	11	12	2	92	13	40

Table 6: Performance results of benchmarks executed with 2 adaptive actions

5. CONCLUSION

In this paper, we presented two dynamic adaptive schemes for home-based software distributed shared memory system. They take advantage of the fact that the home of a page is a natural point to maintain memory access pattern information regarding individual pages. Using this information, pages can migrate to new homes dynamically. The second scheme supports partial write-update protocol. For the same page, on those nodes where the page is accessed frequently, the write-update protocol is used, whereas on the others the initial write-invalidate protocol is maintained. Our experiments showed that the proposed schemes could effectively reduce the number of external page requests, as shown in table 6, thereby reducing the network communication overheads. The proposed schemes are general and can be implemented on any home-based software DSM.

6. REFERENCES

[1] R. Samantha, A. Bilas, L. Iftode, and J. P. Singh. "Home-based SVM Protocols for SMP Clusters: Design and Performance." Fourth Internal Symposium on High Performance Computer Architecture, February 1998.

[2] Y. Zhou, L. Iftode, and K. Li. "Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems." In Proceedings of the Second USENIX Symposium on Operating System Design and Implementation, pages 75-88, November 1996.

[3] P. Keleher. "Lazy Release Consistency for Distributed Shared Memory." Ph.D thesis, Rice University, 1994.

[4] A.L. Cox, E. Lara, C. Hu, and W. Zwaenepoel. "A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory." In Proceedings of the Fifth High Performance Computer Architecture Conference, January 1999.

[5] S. Dwarkadas, H.H. Lu, A.L. Cox, R. Rajamony, and W. Zwaenepoel. "Combining Compile-time and Run-time Support for Efficient Software Distributed Shared Memory." In Proceedings of IEEE, Special Issue on Distributed Shared Memory, pages 476-486, March 1999.

[6] J.H. Kim, and N.H. Vaidya. "Adaptive Migratory Scheme for Distributed Shared Memory." Technical report 96-023, November 1996.

[7] C. Amza, A.L. Cox, S. Dwarkadas, L.J. Jin, K. Rajamani, and W. Zwaenepoel. "Adaptive Protocols for Software Distributed Shared Memory." In Proceedings of IEEE, Special Issue on Distributed Shared Memory, pages 467-475, March 1999.

[8] L.R. Monnerat, and R. Bianchini. "Efficiently Adapting to Sharing Patterns in Software DSMs." In Proceedings of the 4th International Symposium on High Performance Computer Architecture, February 1998.

[9] J.H. Kim, and N.H. Vaidya. "Towards an Adaptive Distributed Shared Memory." Technical report 95-037, September 1995.

[10] K. Thitikamol and P. Keleher. "Thread Migration and Communication Minimization in DSM Systems." In The Proceedings of the IEEE, March 1999.

[11] S.J. Eggers, and R.H. Katz. "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation." In Proceedings of the 15th Annual International Symposium on Computer Architecture, pages 373-383, May 1998.

[12] C. Amza, A. Cox, S. Dwarkadas, H.H. Lu, R. Rajamony, W.M. Yu, and W. Zwaenepoel. "TreadMarks: Shared Memory Computing on Networks of Workstations." IEEE Computer, vol. 29 no. 2, pages 18-28, February 1996.

[13] W.W Hu, W.S. Shi, and Z.M. Tang. "JIAJIA: An SVM System Based on A New Cache Coherence Protocol." In Proceedings of the High Performance Computing and Networking (HPCN'99), April 1999.

[14] M.R. Eskiciogiu, T.A. Marsiand, W.W. Hu, and W.S Shi. "Evaluation of JIAJIA Software DSM System on High Performance Architectures." In Proceedings of the 12th Annual International Symposium on High Performance Computer Systems and Applications, page 76,May 1998.

[15] D.H. Bailey, J. Barton, T.A. Lansinski, and H. Simon. "The NAS Parallel Benchmark." RNR technical report RNR-91-002, Januray 1991.

[16] J.P. Singh, W-D Weber, and A. Gupta. "SPLASH: Standford Parallel Applications for Shared Memory." Standard University CSL-TR-92-626, June 1992.