Transparent Data Selection and Regional Locality in Distributed Shared Memory

Zhiyi Huang†, Chengzheng Sun‡, Abdul Sattar‡, and Ian McDonald†

†Dept of Computer Science
University of Otago, Dunedin, New Zealand
Email:{hzy,jrm}@cs.otago.ac.nz
‡School of Computing & Information Technology
Griffith University, Brisbane, Qld 4111, Australia
Email:{scz,sattar}@cit.gu.edu.au

Abstract. *This paper discusses transparent* Data Selection *achieved by exploring* Regional Locality *in Distributed Shared Memory (DSM). Data Selection means processors propagate to each other only these shared data objects which are really shared among them.* Regional Locality *is the program behaviour in which a set of addresses that are accessed in one critical or non-critical region will be very likely accessed as a whole in the same critical region or other non-critical regions. Three update propagation protocols based on Regional Locality are discussed in terms of achieving transparent Data Selection in Distributed Shared Memory systems. These protocols include: Selective Lazy/Eager Update Propagation protocol, First Hit Update Propagation protocol, and Second Hit Update Propagation protocol. Our experimental results indicate that these protocols can achieve transparent Data Selection for many Distributed Shared Memory concurrent programs. We have also shown that the proposed protocols outperform the existing update propagation protocols based on temporal locality. Implementing transparent Data Selection at both run-time and compile-time would be an interesting future research direction.*

Key Words: Distributed Shared Memory, Data Selection, Regional Locality

1 Introduction

Many weaker sequential consistency models [4, 6, 2, 10, 9] have been proposed in Distributed Shared Memory (DSM) systems. The goal of these models is to achieve Sequential Consistency (SC) [11] on networks of workstations as efficiently and conveniently as possible. These models can take advantage of explicit synchronisation primitives, e.g., *acquire*, *release*, and *barrier*, to achieve time selection, processor selection, and data selection [13].

To explain the above three selection techniques, we consider a DSM system with four processors P_1, P_2, P_3, and P_4, where P_1, P_2, and P_3 share a data object x, and P_1 and P_4 share a data object y. Suppose all memory accesses to the shared data objects x and y are *serialised* among competing processors by means of synchronisation operations to avoid data races. Under these circumstances, the following selection techniques can be used: (1) **Time selection**: Updates on a shared data object by one processor are made visible to the public *only at the time* when the data object is to be read by other processors. For example, updates on x by P_1 may be propagated outward only at the time when either P_2 or P_3 is about to read x. (2) **Processor selection:** Updates on a shared data object are propagated from one processor to *only one other processor* which is the next one in sequence to read the shared data object. For example, updates on x by P_1 may be propagated to only P_2 (but not to P_3) if P_2 is the next one in sequence to read x. (3) **Data selection:** Processors propagate to each other *only these shared data objects* which are really shared among them. For example, P_1, P_2, and P_3 may propagate to each other only data object x (not y), and P_1 and P_4 propagate to each other only data object y (not x). To improve the performance of the strict SC model, a number of weaker SC models have been proposed, such as Weak Consistency (WC) [4], Eager Release Consistency (ERC) [6], Lazy Release Consistency (LRC) [10], Entry Consistency (EC) [2], and Scope Consistency (ScC) [9]. They perform one or more of the above three selection techniques while appearing to be sequentially consistent. We call these weaker SC models *selective SC models*. Table 1 summarises the selection techniques used by these selective SC models.

Model	TS	PS	DS
SC	No	No	No
WC	Sync. time	No	No
ERC	Rel. time	No	No
LRC	Acq. time	Next proc.	No
EC	Acq. time	Next proc.	Lock-data assoc. (user-annotation)
ScC	Acq. time	Next proc.	Scope-data assoc. (user-annotation)

Table 1. Selection techniques used in existing selective SC models

All existing selective SC models achieve time/processor/data selection by requiring programmers to manually annotate the programs so that time/processor

/data selection can be combined with synchronisation operations. For example, the WC model requires programs to explicitly access special synchronisation variables before and after accessing normal shared variables, so that the memory system is explicitly notified to propagate updates at synchronisation access time. In the ERC model, programs are required to call an *acquire* primitive before accessing shared data objects and call a *release* primitive after accessing shared data objects, so that the memory system is explicitly notified of the entry and exit of a critical region, and can select the exit time to propagate updates. The LRC and EC models achieve both time and processor selections by requiring programs to explicitly call *acquire* and *release* primitives at the entry and exit of a critical region, respectively, so that the memory system can propagate updates only to the next processor at the entry time (instead of at the exit time as in the ERC model). Data selection in the EC model is achieved by requiring the programmer to explicitly associate synchronisation objects with shared data objects. The ScC model made one step toward (partially) transparent data selection by taking advantage of the consistency scopes implicitly defined by synchronisation primitives, but programmers may still have to explicitly define additional consistency scopes in programs due to correctness considerations. Although the programmer annotation approach can achieve time/processor/data selection effectively, the programmer has to be very careful about these annotations to ensure the correctness of the program. This imposes an extra burden on programmers and increases the complexity of parallel programming.

The goal of our research is to design and implement an efficient DSM system based on a *transparent selective SC* approach, which is able to achieve both high performance and programming convenience by automatically selecting the right time, right processor, and right data for maintaining a sequentially consistent shared memory. Toward this end, we distinguish two types of programmers' annotations: one is the synchronisation annotations which are required by both the correctness of parallel programs (to avoid data races) and the correctness of memory consistency; and the other is the annotations which are required only by the correctness of memory consistency. For the first type of annotations, such as the *acquire* and *release* synchronisation primitives in the ERC, LRC, EC and ScC models, the DSM system can take advantage of them to achieve time/processor selection without imposing any additional burden on programmers. However, for the second type of annotation, such as the association between synchronisation objects and shared data objects for data selection in the EC model, and the additional consistency scopes in the ScC model, they are truly an extra burden to programmers and should be replaced by automatic associations via run-time detection (and/or compile-time analysis). As the first step, we focus on the protocols and techniques used to achieve *transparent data selection* under the condition of synchronised shared memory accesses. In other words, parallel programs are assumed to use synchronisation primitives, such as *acquire* and *release*, to avoid data races (as in the ERC, LRC, EC, and ScC models), but no programmers' annotation is required to associate shared data objects with synchronisation operations, or to define consistency scopes.

2 Data selection

In general, data selection can reduce the amount of data propagated among processors since it propagates only those shared data objects among processors. In page-based DSM systems in particular, data selection can reduce the effect of false sharing (False sharing occurs when two processors update different data objects that lie in the same memory consistency unit, e.g. a page). Let us again consider the example in the previous section. We assume data objects x and y lie in the same page p. Though P_3 and P_4 don't share any data objects, they now share the same page p. When P_3 modifies page p by updating x, the updates of x have to be propagated to P_4 in order to make the copy of p consistent in P_4, even though P_4 does not need those updates at all. This effect of false sharing can be reduced if we can distinguish updates on different data objects located in the same page and then selectively propagate them.

In page-based DSM systems, consistency can be maintained by either update protocol or invalidate protocol. In update protocol, when a page is updated its updates are propagated to update copies of the page in other processors. In invalidate protocol, when a page is updated its copies in other processors are invalidated by invalidation notices; when an invalid copy of the page is accessed the updates of the page are propagated. Therefore, in page-based DSM systems, we distinguish between two kinds of data selection: selection of invalidation notices, and selection of updates. Selection of invalidation notices means we selectively propagate invalidation notices according to the true sharing of data objects. Selection of updates means we selectively propagate updates of data objects according to the true sharing among processors. In update protocol, we can perform selection of updates. In invalidate protocol, we can perform both selection of updates and selection of invalidation notices.

Selection of updates can be further divided into *lazy* and *eager* update selection. *Lazy* update selection means only when a data object is accessed by a processor, its updates are propagated to the processor. For example, when P_4 accesses data object y, it requests updates of y from P_1, and then P_1 propagates updates of y to P_4. *Eager* update selection means a data object is known *in advance* to be accessed by a processor, and its updates are propagated to the processor before it is accessed. For example, if P_1 knows in advance P_4 will access data object y, P_1 can eagerly propagate updates of y to P_4 before P_4 accesses and requests y. *Eager* update selection can be more efficient than *lazy* update selection because it can propagate updates of several data objects in a single message. However *eager* update selection risks propagating useless data if it can not accurately detect in advance which data objects will be accessed by which processors. In addition, *lazy* update selection can be easily achieved in invalidate protocol.

Transparent data selection can be implemented at run-time and/or compile-time. Run-time implementation can select useful data objects more accurately than its compile-time counterpart, but its run-time overhead may sometimes overshadow the benefit of data selection. A combination of run-time and compile-time implementation would be a promising approach in the future.

In this paper we discuss protocols and techniques which can achieve transparent data selection (especially eager update selection) at run-time by taking advantage of a program behaviour called *Regional Locality.* The rest of this paper is organised as follows. We introduce *Regional Locality* in Section 3. Then we discuss update propagation protocols based on *Regional Locality* in Section 4. These protocols are compared with related work in Section 5. Experimental results are analysed and discussed in Section 6. Finally, the major conclusions and future work are presented in Section 7.

3 Regional Locality

Reference locality [12] in program behaviour has been studied and explored extensively in memory design, code optimisation, multiprogramming, etc. There are two broad classifications of locality: *temporal locality*, which means an address accessed in the past is likely to be accessed in the near future; and *spatial locality*, which says an address nearby in memory space to the one just accessed is likely to be accessed in the near future. In addition to *temporal locality* and *spatial locality*, many Distributed Shared Memory (DSM) concurrent programs exhibit the third kind of reference locality – *Regional Locality* in their execution. Before explaining *Regional Locality*, we need to give a brief introduction to regions in the execution of DSM programs.

An execution of a DSM concurrent program can be viewed as a sequence of regions which are delimited by synchronisation primitives, such as *acquire, release* and *barrier.* A critical region begins with an *acquire* and ends with a *release*, while a non-critical region begins with a *release* (out-most one in nested critical regions) or a *barrier* and ends with an *acquire* (out-most one in nested critical regions) or a *barrier.* We say two critical regions are the same if both of them are protected by the same lock.

Regional Locality is the program behaviour in which a set of addresses that are accessed in one critical or non-critical region will be very likely accessed as a whole in the same critical region or other non-critical regions. For instance, in a page-based DSM system, suppose processor P_1 enters a critical region and accesses pages $\{m_1, m_2, ..., m_n\}$ during the execution of the critical region, and processor P_2 enters the same critical region afterwards. P_2 will most likely access pages $\{m_1, m_2, ..., m_n\}$ during the execution of this critical region, since the same critical region usually protects the same set of data objects. Similar behavior also exists in non-critical regions of a DSM program. For example, suppose processor P_1 enters a non-critical region and accesses pages $\{m_1, m_2, ..., m_n\}$ during the execution of the non-critical region, and processor P_2 enters another non-critical region afterwards. Since data objects accessed in a non-critical region often migrate together from one processor to another processor, which is regulated by the programmer to avoid data race in non-critical regions, when P_2 accesses one or two members of the page set $\{m_1, m_2, ..., m_n\}$, it will very likely access every member of the set $\{m_1, m_2, ..., m_n\}$.

Regional Locality is similar to temporal locality in the sense that it acquires the knowledge of locality from the past execution of the program. Their difference is that temporal locality uses all the addresses accessed by a processor in the past as one locality group for the processor itself, while *Regional Locality* divides into groups the addresses accessed by a processor in the past according to their occurring program regions and uses these groups as locality groups for all processors. Like other kinds of locality, *Regional Locality* can also be explored to improve performance of DSM programs. In this paper we explore *Regional Locality* in update propagation in DSM systems.

4 Update propagation based on Regional Locality

A DSM update propagation protocol determines when and how updates on one copy of a page are propagated to other copies of the same page on other processors. Updates on a page can be represented by a *single-writer* scheme or by a *multiple-writer* scheme [3]. An update propagation protocol can be integrated with either a single-writer scheme or a multiple-writer scheme.

There have existed a number of different protocols for propagating updates in DSM systems [7]. One protocol, adopted by the TreadMarks DSM system [1], works as follows: when an old copy of a page needs to be renewed, the old copy is invalidated first; only when the invalidated old copy is really accessed by a processor and a page fault occurs, are the updates of the page sent to the processor. We call this protocol as the *Lazy Update Propagation* (LUP) protocol since it propagates updates lazily when updated pages are accessed. LUP is actually an invalidate protocol without eager update selection.

In LUP each page fault involves an update requesting message to a remote processor, and an update propagating message from a remote processor. The large number of messages caused by page faults influence seriously the performance of DSM systems. If we can apply eager update selection and prefetch updates of several pages in a single page fault, we can reduce page faults and the messages caused by page faults. In this way the performance of the DSM system will be significantly improved. The challenge here is to prefetch as many useful updates as possible while avoiding prefetching useless updates. It is important to be aware that prefetch is a double-edged sword in the sense that prefetch of useful updates can improve performance while prefetch of useless updates may on the contrary seriously degrade the performance. The effect of prefetch depends on the accuracy of update selection. However, it is non-trivial for update selection to detect which updates are useful and which ones are useless to a processor.

In the following sections we use *Regional Locality* as a heuristic in update selection to detect which updates will be needed in the future execution of a processor. Based on this knowledge we prefetch useful updates in our novel update propagation protocols.

4.1 Update propagation in critical regions

In update propagation we are only concerned about the updated pages whose updates need to be propagated. To explore *Regional Locality* in update propagation in critical regions, every lock in a processor is associated with a *Critical Region Updated Pages Set (CRUPS)* which stores pages updated in a critical region. The CRUPSs actually keep the knowledge of *Regional Locality* in critical regions. A CRUPS is formed as follows. Before a processor enters a critical region by acquiring a lock, an empty CRUPS is created for the lock. If the processor updates a page during the execution of the critical region, the identifier of the page is recorded into the CRUPS of the corresponding lock. When the processor exits from the critical region, it stops recording in the CRUPS, but keeps the the contents of the CRUPS for use in the next acquisition of the same lock.

According to *Regional Locality*, we know when a processor enters a critical region it will very likely access the pages previously updated in the same critical region. So when a processor P_2 enters a critical region by acquiring a lock from another processor P_1, P_1 can assume that P_2 will access the pages in its CRUPS of the lock and thus piggy-backs the updates of these pages on the lock grant message. This idea is essentially a data prefetching technique based on the acquired knowledge of *Regional Locality.* Based on the above idea we proposed a hybrid update propagation protocol, called the *Selective Lazy/Eager Update Propagation (SLEUP)* in [13].

4.2 Update propagation in non-critical regions

To explore *Regional Locality* in update propagation in non-critical regions, we detect the pages updated in non-critical regions and aggregate them together. We propose a *Non-Critical Region Updated Pages Set (NCRUPS)* scheme for grouping pages updated in non-critical regions. In every processor we associate every non-critical region with a NCRUPS. The NCRUPSs actually keep the knowledge of *Regional Locality* in non-critical regions. A NCRUPS is formed as follows. When a processor enters a non-critical region, a unique empty NCRUPS is created and assigned to the non-critical region; when a processor updates a page during the execution of a non-critical region, the identifier of the page is recorded into the corresponding NCRUPS; when a processor leaves a non-critical region, it stops recording into the corresponding NCRUPS but saves the NCRUPS for later use.

By using the NCRUPS scheme, we can group pages updated inside each non-critical region and optimally propagate updates of these pages to a processor when it is about to access them. We use some hints to decide whether a processor is about to access the pages in a NCRUPS so as to propagate all the updates of these pages to the processor. The first hint we use is the first page fault on any page in a NCRUPS. This hint suggests all the pages in the NCRUPS might be accessed soon by the processor according to *Regional Locality.* Therefore when a fault on a page in a NCRUPS occurs in a processor, we propagate the updates of all the pages in the NCRUPS to the processor. Based on the above idea, we proposed an update propagation protocol called *First Hit Update Propagation (FHUP)* in [8].

However the FHUP protocol does not have sufficient hints to detect correct knowledge of *Regional Locality* for false sharing access patterns, and may cause useless update propagation [8]. To overcome this drawback, we use both the first and the second page faults on pages in a NCRUPS as hints. That is, if a page in a NCRUPS is accessed in a non-critical region by a processor, and later another page in the same NCRUPS is accessed in the same non-critical region by the same processor, then all the pages in the NCRUPS are very likely to be accessed by the processor and therefore the updates of all the pages in the NCRUPS are propagated to the processor. Based on the above idea we proposed another update propagation protocol called *Second Hit Update Propagation (SHUP)* in [8]. The advantage of the SHUP protocol is that the second page fault is used to correctly detect *Regional Locality* and avoid useless update propagation.

5 Comparison with related work

A Lazy Hybrid (LH) protocol [5] is proposed based on temporal locality. The idea behind the LH protocol is that programs usually have significant temporal locality, and therefore any page accessed by a process in the past is likely to be accessed in the future. The LH protocol therefore selects updates of pages that have been accessed in the past (regardless whether or not in the same critical/non-critical region) by the processor acquiring a lock or arriving at a barrier, and piggy-backs the updates on grant messages. The similarity between LH and our protocols is that both of them use some kinds of locality heuristics to prefetch updates of pages. The major difference between LH and our protocols is the following: the former uses a heuristic without distinguishing the accessed pages which are in the same critical/non-critical region from these pages which are not, while the latter makes this distinction based on *Regional Locality* and hence can be more accurate in selecting the updates for prefetch. Since the heuristic in the LH protocol is very speculative, it can cause useless update propagation, and thus degrades the performance of the underlying DSM system. This point has been verified by our experimental results.

6 Experimental results

All our protocols are implemented in TreadMarks. The Lazy Hybrid protocol is also implemented in TreadMarks in order to compare *Regional Locality* with temporal locality in DSM. All these protocols are evaluated with the Lazy Update Propagation (LUP) protocol adopted in TreadMarks, which does not explore any locality and is a benchmark for those exploring locality.

The experimental platform consists of 8 SGI workstations running IRIX Release 5.3. These workstations are connected by a 10 Mbps Ethernet. Each of them has a 100 MHz processor and 32 Mbytes memory. The page size in the virtual memory is 4 KB.

We used 8 applications in the experiment: *TSP, BT, QS, Water, FFT, SOR, Barnes, IS*, among which the source code of *TSP, QS, Water, FFT, SOR, Barnes, IS* are provided by TreadMarks research group. All the programs are written in C.

application	protocol	Time (secs)	Total Data (bytes)	Updates Data (bytes)	Page Fault	Mesgs
TSP	LUP	15.86	1267683	448958	1029	2846
	LH	8.63	1287368	463437	355	1405
	SLEUP	7.33	1252737	443896	245	1209
BT	LUP	82.92	39511375	8921228	26478	96979
	LH	72.08	40964979	9390072	13918	68542
	SLEUP	69.71	39148835	8761972	6469	53925
QS	LUP	20.09	10153006	6100023	3046	10432
	LH	15.52	10844953	6962709	962	6095
	SLEUP	13.36	9165498	5354832	956	5936
	SLEUP+FHUP	14.96	11596416	7838800	829	5447
	SLEUP+SHUP	12.38	9282800	5430895	930	5886
Water	LUP	32.59	11717602	9980061	4314	24495
	LH	36.82	14535830	12590288	2137	21668
	SLEUP	31.07	11834142	9981561	3024	21920
	SLEUP+FHUP	31.92	13759521	11607920	1733	18906
	SLEUP+SHUP	30.63	12159638	9979899	1992	19764
FFT	LUP	4.44	3220826	2188032	557	2135
	LH	9.26	5540076	4487644	174	1735
	FHUP	4.87	3902122	2820048	291	1603
	SHUP	4.60	3306240	2188032	557	2136
SOR	LUP	13.70	7391113	14140	203	4301
	LH	15.10	7934204	473636	16	4992
	FHUP	14.53	7556048	134885	203	4302
	SHUP	13.84	7416629	14140	203	4303
Barnes	LUP	49.38	50943423	37198386	12791	75943
	LH	X	X	X	X	X
	FHUP	48.14	55534888	37687510	12640	74318
	SHUP	49.17	55136208	37199430	12763	75659
IS	LUP	113.42	71732008	69626536	4444	11305
	LH	120.15	75004402	73404180	192	8044
	FHUP	108.20	72100823	69626536	2774	7965
	SHUP	110.62	72223052	69623400	3998	10384

Table 2. Performance Statistics for applications

Among these applications, *TSP* and *BT* only use locks for synchronisation, and *QS* uses one lock to protect a task queue, Water uses both locks and barriers for synchronisation, and *FFT, SOR, Barnes*, and *IS* only use barriers for synchronisation. The FHUP and SHUP protocols are not applied to *TSP* and

BT since there is no update on shared memory in non-critical regions in these two applications. Also since there are no critical regions in *FFT, SOR, Barnes*, and *IS*, the SLEUP protocol is not applied to them.

The experimental results are given in Table 2. In the table, the item *Time* is the total running time of an application program; the *Total Data* is the sum of total message data; the *Updates Data* is the sum of total propagated updates data; the *Page Fault* is the number of page faults; and the *Mesgs* is the total number of messages;

6.1 Regional Locality

From the experimental results we know *Regional Locality* exists in many DSM concurrent programs. Among the applications with *Regional Locality* are *TSP*, *BT*, *QS*, *Water*, *Barnes*, and *IS*. By applying SLEUP, FHUP and SHUP, which explore *Regional Locality*, the average improvement on the performance of these applications is 20.2%. The maximum improvement is up to 53.8% (*TSP*). Particularly, by exploring *Regional Locality*, the number of page faults and the number of messages are reduced to 46% and 66% respectively in average. There is no improvement on the performance of some applications, such as *FFT* and *SOR*, because they don't have any *Regional Locality*.

6.2 Regional Locality vs. temporal locality

Protocols based on *Regional Locality* outperform those based on temporal locality for all of our applications. Compared with LUP, LH degrades the performance of many programs, such as *Water*, *FFT*, *SOR*, *Barnes*, *IS*. (Because message buffers overflow at barriers, we have not provided running results of *Barnes* based on the LH protocol [1]). The average degradation is 34.4%, and the maximum degradation is up to 108.6% (*FFT*). The reason for the degradation is that LH propagates a large number of useless updates. The average amount of useless updates propagated in LH is 27.8% of the total propagated updates. Even though LH can improve some applications, such as *TSP*, *BT*, and *QS*, its performance is still not as good as SLEUP/SHUP/FHUP. The performance of SLEUP/SHUP/FHUP is 17.7% better than that of LH on average. The average amount of updates propagated in SLEUP/SHUP/FHUP is 29.7% less than that in LH. Even though in some applications, such as *FFT*, *SOR* and *IS*, the number of page faults and the number of messages in LH are less than those in SLEUP/SHUP/FHUP, the overall performance of SLEUP/SHUP/FHUP is better than that of LH since LH propagates a large number of useless updates.

From the above discussion we know temporal locality is more speculative than *Regional Locality.* Temporal locality does not have as accurate a knowledge of the to-be-accessed data as *Regional Locality.* This inaccuracy of temporal locality causes useless update propagation and degrades the performance of DSM systems.

[1] The buffer overflows because of too much (useless) update propagation at barriers in LH, and therefore its performance will be further degraded at barriers

6.3 Data selection and Regional Locality

The accuracy of data selection in SLEUP/SHUP/FHUP depends on the detection of *Regional Locality*. We use the CRUPS scheme to detect *Regional Locality* in critical regions, and use the NCRUPS scheme and the first/second page fault to detect *Regional Locality* in non-critical regions. The accuracy of data selection affects the performance of those protocols. On one hand, incorrect data selection causes useless update propagation. For example, for *FFT* and *SOR* the FHUP protocol detects the incorrect knowledge of *Regional Locality* and selects the incorrect data objects. So FHUP propagates useless updates and degrades performance in these two applications. On the other hand, incomplete data selection hinders the improvement of performance. For instance, SHUP can not perform data selection as immediate as FHUP in *IS* and *Barnes*. So SHUP does not perform as well as the FHUP for these two applications.

From the above discussion we know, even though both FHUP and SHUP are based on *Regional Locality*, data selection in FHUP is more speculative while data selection in SHUP is more conservative in terms of data selection. Their merits become prominent in different applications.

The overhead of the CRUPS scheme is very small because it takes advantage of the write-protection mechanism provided in the TreadMarks system. There is some overhead for bookkeeping the *remote NCRUPS list* in the NCRUPS scheme. For example, for *FFT* and *SOR* where there is no *Regional Locality*, SHUP slightly degrades their performance (3.6% degradation for *FFT*, 1.0% degradation for *SOR*) because of this bookkeeping overhead.

7 Conclusions

In this paper, we have discussed transparent data selection and its implementation based on the program behaviour – *Regional Locality* and evaluated this new class of reference locality in update propagation in DSM systems. We have discussed three novel update propagation protocols, SLEUP, FHUP, and SHUP, which achieve transparent data selection based on *Regional Locality* in DSM systems. The experimental results indicate:

- *Regional Locality* exists in the execution of many Distributed Shared Memory concurrent programs. Update propagation protocols exploring *Regional Locality* significantly improve the performance of the DSM systems.
- Data selection techniques based on *Regional Locality* outperform those based on the more speculative temporal locality. Protocols exploring temporal locality cause performance degradation for many applications in our experiment.

Our future research is to implement transparent data selection at both runtime and compile-time.

Acknowledgements

This research is supported by an ARC (Australian Research Council) large grant (A49601731), and a NCGSS grant by Griffith University.

References

1. C.Amza, et al: "TreadMarks: Shared memory computing on networks of workstations," *IEEE Computer,* 29(2):18-28, February 1996.
2. B.N. Bershad, et al: "The Midway Distributed Shared Memory System," *In Proc. of IEEE COMPCON Conference,* pp528-537, 1993.
3. J.B. Carter, J.K. Bennett, and W. Zwaenepoel: "Techniques for reducing consistency-related information in distributed shared memory systems," *ACM Transactions on Computer Systems,* 13(3):205-243, August 1995.
4. M.Dubois, C.Scheurich, and F.A.Briggs: "Memory access buffering in multiprocessors," *In Proc. of the 13th Annual International Symposium on Computer Architecture*, pp.434-442, June 1986.
5. S. Dwarkadas, et al: "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology", *In Proc. of the 20th Symposium on Computer Architecture*, pp.144-155, May 1993.
6. K. Gharachorloo, D.Lenoski, J.Laudon: "Memory consistency and event ordering in scalable shared memory multiprocessors," *In Proc. of the 17th Annual International Symposium on Computer Architecture,* pp15-26, May 1990.
7. Z. Huang, W.-J. Lei, C. Sun, and A. Sattar: "Heuristic Diff Acquiring in Lazy Release Consistency Model," *In Proc. of 1997 Asian Computing Science Conference (ASIAN'97),* LNCS 1345, Springer-Verlag, pp98-109, Dec. 1997.
8. Z. Huang, C. Sun, and A. Sattar: "Exploring Regional Locality in Distributed Shared Memory," *In Proc. of 1998 Asian Computing Science Conference (ASIAN'98),* Dec. 1998.
9. L. Iftode, J.P. Singh and K. Li: "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," *In Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures,* 1996.
10. P. Keleher: "Lazy Release Consistency for Distributed Shared Memory," *Ph.D. Thesis,* Rice Univ., 1995.
11. L. Lamport: "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers,* 28(9):690-691, September 1979.
12. J.R. Spirn: "Program Locality and Dynamic Memory Management," *PhD thesis*, Princeton University, 1973.
13. C. Sun, Z. Huang, W.-J. Lei, and A. Sattar: "Towards Transparent Selective Sequential Consistency in Distributed Shared Memory Systems," *In Proc. of the 18th IEEE International Conference on Distributed Computing Systems,* Amsterdam, May 1998.