View-based Consistency and Its Implementation

Z. Huang†, C. Sun‡, S. Cranefield†, and M. Purvis†
†Departments of Computer & Information Science
University of Otago, Dunedin, New Zealand

‡School of Computing & Information Technology
Griffith University, Brisbane, Australia
Email:hzy@cs.otago.ac.nz, scz@cit.gu.edu.au,
{mpurvis, scranefield}@infoscience.otago.ac.nz

Abstract

This paper proposes a novel View-based Consistency model for Distributed Shared Memory. A view is a set of ordinary data objects that a processor has the right to access in a data-race-free program. The View-based Consistency model only requires that the data objects of a view are updated before a processor accesses them. Compared with other memory consistency models, the View-based Consistency model can achieve data selection without user annotation and can reduce much false-sharing effect. This model has been implemented based on TreadMarks. Performance results have shown that for all our applications the View-based Consistency model outperforms the Lazy Release Consistency model.

Key Words: Distributed Shared Memory, Sequential Consistency, False Sharing

1 Introduction

Distributed Shared Memory (DSM) has become an active area of research in parallel and distributed computing [16, 8, 4, 3, 1, 19]. A DSM system can provide application programmers the illusion of shared memory on top of message passing distributed systems, which facilitates the task of parallel programming in distributed systems. The goal of our research is to make DSM systems more convenient to use and more efficient to implement [10, 19]. In this paper, we propose a View-based Consistency (VC) model for DSM, which is a significant step toward our goal.

The consistency model of a DSM system specifies the ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems' programming convenience and implementation efficiency [17]. The Sequential Consistency (SC) model [15] has been recognized as the most natural and user-friendly DSM consistency model. The SC model guarantees that *the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its own program* [15]. This means that in a SC-based DSM system, memory accesses from all processors may be interleaved in any sequential order that is consistent with each processor's memory access order, and the memory access orders observed by all processors are the same. One way to strictly implement the SC model is to ensure all memory updates be totally ordered and memory updates performed at one processor be immediately propagated to other processors. This implementation is correct but it suffers from serious performance problems [17].

In practice, not all parallel applications require each processor to see all memory updates made by other processors, let alone to see them in order. Many parallel applications regulate their accesses to shared data by synchronization, so not all valid inter-leavings of their memory accesses are relevant to their real executions. Therefore, it is not necessary for the DSM system to force a processor to propagate *all* its updates to *every* other processor (with a copy of the shared data) at *every* memory update time. Under certain conditions, the DSM system can select the *time*, the *proces-*

sor, and the *data* for making shared memory updates public to improve the performance while still appearing to be sequentially consistent[19]. Under these circumstances, the following three basic techniques can be used: **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. **Processor selection:** Updates on a shared data object are only propagated from one processor to the processor that is the *next in sequence* to access the shared data object. **Data selection:** Processors only propagate to each other *those shared data objects* that are really shared among them.

To improve the performance of the strict SC model, a number of weaker SC models have been proposed [6, 9, 14, 2, 13], which perform one or more of the above three selection techniques while appearing to be sequentially consistent. However, none of them can achieve data selection without programmer annotation [19]. We argued previously [19] that a consistency model should not impose any extra burden on programmers, such as annotation of lock-data association in the Entry Consistency (EC) [2] and scope-data association in the Scope Consistency (ScC) [13] models. In this paper, we propose a *View-based Consistency (VC)* model which, besides time selection and processor selection, can transparently achieve data selection.

The rest of this paper is organized as follows. Section 2 describes in detail the VC model and its properties. In Section 3 the VC model is compared with some related models, e.g. EC and ScC, in terms of user annotation, data selection, interface for programmers, and false-sharing effect in Section 3. Issues regarding an implementation of VC are discussed and presented in Section 4. Performance results are presented and evaluated in Section 5. Finally, the major contributions of this paper and areas for future work are summarized in Section 6.

2 View-based Consistency

During the execution of a DSM parallel program, multiple processors communicate with each other through the shared memory. In shared memory some data objects are *read-only*, and some *read/write*. To prevent data races (where multiple processors read and write the same data object concurrently), a parallel program has to guarantee that a processor has gained exclusive access before accessing a *read/write* data object. This kind of parallel programs is called data race free.

We distinguish *synchronization* data objects from *ordinary* data objects in shared memory, just like many other DSM systems. Synchronization data objects are those which are explicitly used to enforce exclusive access to other data objects, such as locks and barriers[1]. The rest of the data objects in shared memory are called ordinary data objects. Exclusive access to the synchronization data objects is guaranteed by system-provided primitives, such as *acquire*, *release*, and *barrier*, while exclusive access to the ordinary data objects has to be guaranteed by using those system primitives. Like many Weak Sequential Consistency models[11], sequential consistency for the synchronization data objects is guaranteed by the system; however, sequential consistency for the ordinary data objects is achieved conditionally, depending on the underlying consistency model. Therefore, we only need to be concerned with the consistency of the ordinary data objects.

A *view* is a set of ordinary data objects a processor has the *right* to access in shared memory. We say a processor has the *right* to access some data object if and only if it has gained exclusive access to the data object or the data object is *read-only*. At any time point of an execution, suppose any two processors P_1 and P_2 have views V_1 and V_2 respectively. Then $V_1 \cap V_2$ must only contain read-only data objects; otherwise a data race occurs. Fig. 1 shows a snapshot of views of processors in shared memory. The overlapped part of different views only contains read-only data objects.

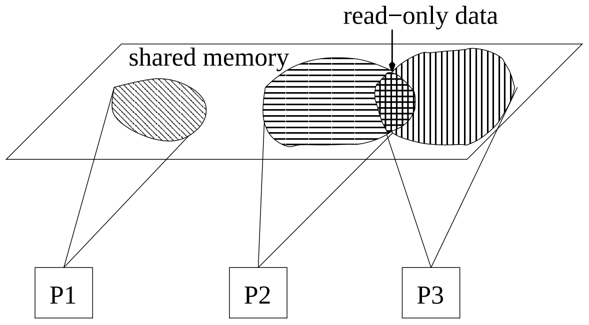

Figure 1: A snapshot of processors' views

Many DSM systems require explicit calls to *acquire*, *release* and *barrier* in programs to achieve weak sequential consistency. An execution of such a DSM program can be

[1]A barrier is a synchronization device that requires all processes to wait for the last of them to arrive at the same synchronization point. It can be implemented by *acquire* and *release*.

viewed as a sequence of *barrier sessions* shown in Fig. 2. A barrier session begins with a *barrier* and ends with another *barrier*. Inside a barrier session there is a sequence of regions which are delimited by *acquire, release* and *barrier* primitives. A critical region begins with an *acquire* and ends with a *release*, while a non-critical region begins with a *release* (the outermost one in nested critical regions) or a *barrier* and ends with an *acquire* (the outermost one in nested critical regions) or a *barrier*. A non-critical region does not overlap with any critical region, but a critical region may overlap with another critical region due to the possibility of nested critical regions.

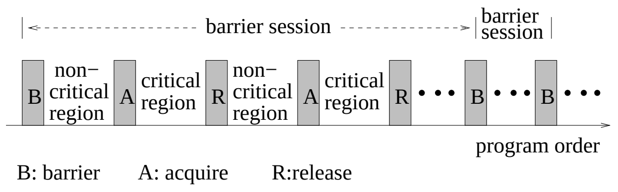

Figure 2: A view of a program execution based on the concept of region

In a DSM program, exclusive access to a data object can only be gained in the following three ways:

1. implicit assignment by the programmer inside a barrier session. Exclusive access is guaranteed by barriers.
2. explicit acquisition by calling the *acquire* primitive. Exclusive access is guaranteed by the lock mechanism of critical regions.
3. implicit acquisition by changing the status of data objects protected by critical regions. For example, exclusive access to a task from a task queue is guaranteed by removing the task from the lock-protected task queue.

Therefore, in an execution of a DSM program, only when a processor calls synchronization primitives, such as *barrier, acquire*, and *release*, does its view change, as shown in Fig. 3. A processor's view is constant inside a critical region or a non-critical region. Only when a processor moves from one region to another, does it gain or lose exclusive access to some data objects.

According to this observation, views can be classified as *Critical Region Views (CRVs)* and *Non-critical Region Views (NRVs)*. A processor's CRV is its view while it executes inside a critical region. A processor's NRV is its view while it executes inside a non-critical region. More precisely, the following definitions are given for CRV and NRV.

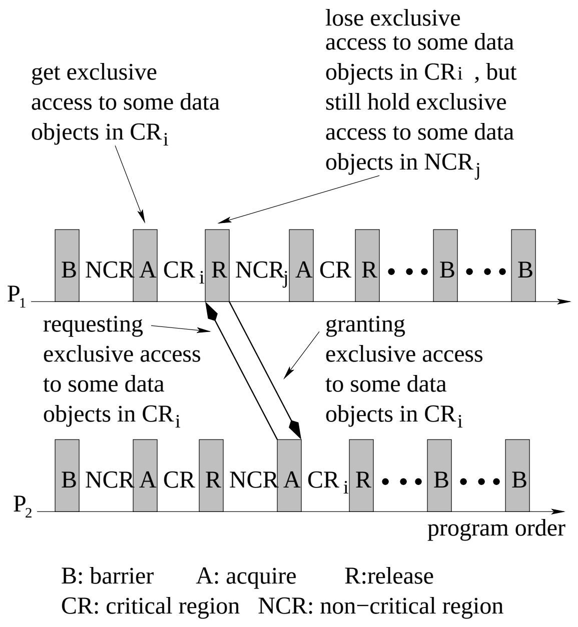
lose exclusive access to some data objects in CRi , but still hold exclusive access to some data objects in NCRj
get exclusive access to some data objects in CRi
P1
B NCR A CR i R NCRj A CR R B B
requesting exclusive access to some data objects in CRi
granting exclusive access to some data objects in CRi
P2
B NCR A CR R NCR A CR i R B B
program order
B: barrier
A: acquire
R:release
CR: critical region
NCR: non−critical region

Figure 3: Views and their transitions

Definition 1 *Critical Region View (CRV)*

A processor's CRV comprises read-only data objects and the data objects to which the processor has exclusive access guaranteed by the current critical region and the current barrier session.

Definition 2 *Non-critical Region View (NRV)*

A processor's NRV comprises read-only data objects and the data objects to which the processor has exclusive access guaranteed by the status of critical-region-protected data objects and the current barrier session.

Based on the definitions of CRV and NRV, we propose a *View-based Consistency (VC)* model with the following consistency conditions.

Definition 3 *Conditions for View-based Consistency*

- Before a processor P_i is allowed to enter a critical region or a non-critical region, all previous *write* accesses to the ordinary data objects of the CRV or NRV must *be performed with respect to* P_i according to their order.
- The sequential consistency of synchronization data objects is guaranteed by the implementation of the system primitives such as *acquire, release*, and *barrier*.

A write access to a memory location is said to *be performed with respect to* processor P_i at a time point when a subsequent read access to that location by P_i returns the value set by the write access.

The VC model has the following properties:

- In the VC model, only when a processor moves from one region to another region does its view change. A processor's view is constant within a region.
- In the VC model, when a processor changes to a new region all the data objects of its new view must be updated.
- The VC model guarantees the same execution result as the Sequential Consistency model for a data-race-free DSM program.
- The VC model can achieve time selection, processor selection, and data selection. Data selection can be achieved by updating only the data objects in the current view of a processor.

3 Comparison of related models

Among the different consistency models, only ScC [13] and EC [2] can achieve data selection. But the VC model is different from them in the following aspects.

User annotation: VC requires no user annotation to achieve data selection. EC requires the user to specify the association between a synchronization data object s and the shared data D_s, where s controls access to a critical region protecting D_s. This specification is essential for EC to achieve data selection. If the specification is not correct, EC can not achieve data selection correctly. ScC also requires the user to specify scope annotation for some programs, though it can detect scope automatically for some other programs.

Data selection: To selectively update data objects, VC uses a concept of view, while EC uses *guarded shared data* D_s and ScC *scope*. However, the view in VC is different from D_s in EC and the *scope* in ScC. Both D_s and *scope* are static and fixed with a particular synchronization data object or a critical region. Even if some data objects are not accessed by a processor in a critical region, they are updated simply because they are associated with the lock or the critical region. However, the view in VC is dynamic and may be different from region to region. Even for the regions protected by the same lock, the views in them are different and depend on the data objects actually accessed by the processor in the regions. Because of this difference, VC is more selective than EC and ScC in terms of data selection. For example, suppose lock l is used to protect a set of shared data objects $S = \{s_1, ...s_n\}$. Because it is common for a processor to access only some data objects in S after it acquires lock l, we can assume the set of accessed data objects is $S' \subset S$. Then when the processor enters the critical region, the D_s in EC and the scope in ScC are S, while the view in VC is S'. EC and ScC have to update all data objects in S, while VC only updates data objects in S'.

Interface for programmers: VC provides a simple and clear interface for the programmer: if a program is data race free, VC can guarantee the same execution result as Sequential Consistency. But EC requires the programmer to provide correct lock-data association. If the lock-data association is not correct, EC does not guarantee the correct execution of the program. Similarly, ScC does not guarantee the same execution result as Sequential Consistency for some data-race-free programs if explicit scope annotation is not correctly provided by the programmer.

Apart from the above differences, VC has more potential to reduce the effect of false sharing[2] in page-based DSM. It can reduce the false sharing effect in the following two ways:

1. Restrict the propagation of invalidation notices. Only the invalidation notices that are useful for updating the data objects in a processor's new view are propagated to the processor;
2. Restrict the effective scope of invalidation notices. Even though some invalidation notices have been propagated to a processor, only the invalidation notices that are useful for updating the data objects of the current view of the processor are effective in the current region of the processor.

We have shown examples in [12] to explain how VC can reduce false-sharing effect in the above two ways in contrast with LRC and ScC. In the following section, we discuss some issues in the implementation of the VC model.

4 Implementation

There are two technical issues in the implementation of VC. One is *view detection*, and the other is *view transition*. View

[2]False sharing occurs when one processor modifies a shared data object that lies in the same memory consistency unit (e.g. a page) as another shared data object lies, while another processor reads or writes the other shared data object.

detection means that before a processor enters a new region we should find out all the data objects in its new view. View transition means that when a processor's view changes we should update all the data objects of its new view. Any implementation of the VC model should guarantee that before a processor enters a new region, view detection and view transition are achieved correctly.

We have implemented the VC model in the framework of TreadMarks [1], which is a page-based DSM system. In our implementation of the VC model, we regard a page as the basic unit of data objects. Thus a view in our implementation consists of pages.

4.1 View detection

View detection is implemented at run time. In view detection, if a page is not modified it is not necessary to record it in a view, because it has no change and thus does not need consistency maintenance. Therefore, only the modified pages are recorded in a view in view detection.

To detect modified pages in view detection, our implementation takes advantage of the following two existing mechanisms needed by other schemes in the DSM system:

1. When a write access is performed on an invalidated page, a page fault will occur. The page fault handler in the DSM system can be extended to record the faulty page's identifier in the corresponding view, as well as fetching an updated copy of the faulty page from another processor.
2. When a write access is performed on a write-protected page, a protection violation interrupt will occur. The interrupt handler in the DSM system can be extended to record the modified page's identifier in the corresponding view, as well as making a twin of the accessed page in the multiple-writer scheme or obtaining the ownership of the accessed page in the single-writer scheme [5].

Because the above two mechanisms have already been provided by the underlying DSM system, there is little extra overhead for recording the identifiers of modified pages. However, if a page is already writable before a new view is entered, that page will not be detected and recorded in the new view if it will be modified in the view. To detect all modified pages of a view, we make all writable pages write-protected (read-only) before a new view is entered. This is the additional overhead required for view detection. From our experimental results we know this additional overhead is trivial.

The CRVs detected in our implementation are complete and accurate since a processor entering a critical region has exclusive access to those pages modified by other processors in the same critical region. Unfortunately, an NRV detected in our implementation consists of all pages modified by other processors in non-critical regions. That means a processor entering a non-critical region may not have exclusive access to some pages in its NRV. Thus a detected NRV may be bigger than the real one. This inaccuracy only affects the performance, not the correctness of our implementation.

4.2 View transition

Before a new view is entered view transition needs to be done. View transition can be either based on the *invalidation protocol*, which only invalidates those modified pages in the new view, or based on the *update protocol*, which only updates those modified pages in the new view. If the invalidation protocol is used in view transition, the pages that are not in the new view but are modified stay valid until some later view transition needs to invalidate them.

The update protocol is suitable for VC, as is the invalidation protocol for LRC, because VC has done data selection through the use of views and thus the pages in the new view are most likely to be accessed in the corresponding critical region. Therefore updating them straightforwardly helps to reduce the number of messages requesting updates and thus is more efficient than the invalidation protocol. [19]

However, since the detected NRVs are not accurate we adopt a hybrid protocol, which incorporates both the invalidation protocol and the update protocol, in our implementation. The hybrid protocol is similar to the SLEUP protocol[19]. It uses the update protocol for the modified pages in CRVs, but the invalidation protocol for the modified pages in NRVs.

5 Experimental results

In this section, we present an experimental evaluation of the LRC model and our implementation of the VC model. Both of them are implemented in TreadMarks [1]. The experimental platform consists of 8 PCs running Linux Red Hat 6.1, which are connected by a 10 Mbps Ethernet. Each of the PCs has a 500 MHz processor and 128 Mbytes memory. The page size in the virtual memory is 4 KB.

TreadMarks has adopted a multiple-writer scheme [5], which was proposed to minimize the effect of false sharing. In the multiple-writer scheme, initially a page is write-protected. When a write-protected page is first updated by a processor, a *twin* of the page is created and stored in the system space. When the updates on the page are needed by another processor, a comparison of the *twin* and the current version of the page is done to create a *diff*, which can then be used to update copies of the page in other processors. Thus in the multiple-writer scheme the page *diff*, instead of the whole page, is used to renew an old copy.

Since our implementation of VC is based on TreadMarks, we have to adapt to the multiple-writer scheme at the price of false-sharing effect. There are two kinds of false-sharing effect: write/read and write/write. Write/read false-sharing effect occurs when one processor modifies a shared data object that lies in the same memory consistency unit (e.g. a page) as another shared data object, while another processor reads the other shared data object. Write/write false-sharing effect occurs when one processor modifies a shared data object that lies in the same memory consistency unit (e.g. a page) as another shared data object, while another processor writes to the other shared data object. In our implementation we can completely remove the write/read false-sharing effect. However, to work with the multiple-writer scheme properly, our implementation has to tolerate the write/write false-sharing effect. Thus the write/write false-sharing effect has not been removed in our current implementation.

We used four applications in the experiment: *TSP*, *QS*, *BT* and *Water*. *TSP*, *QS*, and *Water* are provided by the TreadMarks research group. All the programs are written in the C language. *TSP* is the Travelling Salesperson Problem. *QS* is a recursive sorting algorithm. *BT* is an algorithm that creates a fixed-depth binary tree. *Water* is a molecular dynamics simulation. These applications are representative of both numerical computing (*Water* and *QS*), and symbolic computing(*TSP* and *BT*). Table 1 gives the performance results.

In the table, VC_i is the VC implementation based on the invalidation protocol, VC_h is the VC implementation based on the hybrid protocol. *Time* is the total running time of an application program, *Diff_Req* is the number of messages for *diff* requests, *RPF* is the reduction in page faults due to the use of the hybrid protocol in the VC model, *RFS* is the reduction in page faults due to the reduction of the false-sharing effect in the VC model, and *Mesgs* is the total number of messages.

APP	Model	Time (Sec.)	Diff_Req	RPF	RFS	Mesgs
TSP	LRC	2.54	962	-	-	2763
	VC_i	2.56	960	-	0	2756
	VC_h	1.65	25	937	0	911
QS	LRC	7.09	3267	-	-	12209
	VC_i	7.15	3330	-	0	12375
	VC_h	4.59	791	1044	0	5301
BT	LRC	28.26	11437	-	-	79468
	VC_i	27.59	11347	-	792	79426
	VC_h	25.73	7429	3441	776	69342
Water	LRC	19.86	12428	-	-	96600
	VC_i	19.91	12423	-	3	96600
	VC_h	19.09	11891	511	3	95478

Table 1: Performance Statistics for applications on eight processors

VC_h vs. LRC

VC outperforms LRC for all four applications tested. From Table 1 we know VC_h has improved the performance significantly compared with LRC (35% for TSP, 35.3% for QS, 9% for BT, and 3.9% for $Water$). The number of *diff* request messages in VC_h is significantly less than that in LRC (97.4% less in TSP, 75.8% less in QS, 35% less in BT, and 4.3% less in $Water$). The hybrid protocol has contributed very much to the reduction of *diff* request messages. Consequently the number of total messages in VC_h has been greatly reduced compared with LRC.

VC_i vs. LRC

The implementation of VC_i aims at investigating the extra overhead of maintaining the views in VC and the false-sharing effect of application programs.

From Table 1 we know some applications, such as TSP and QS, do not benefit from the implementation of VC_i because there is no false-sharing effect in TSP and no reduction in false sharing in QS due to the inaccuracy of NRV in the implementation. However, the performance of VC_i is not significantly worse than that of LRC (0.7% worse for TSP, and 0.8% worse for QS). This demonstrates that the overhead of view maintenance (including view detection and view transition) is only a trivial portion of the expense of the whole system.

As we mentioned early in this section, our current implementation of VC removes any write/read false-sharing effect, but does not remove the write/write false-sharing effect as a result of compromise with the multiple-writer scheme in TreadMarks. Thus the RFS showed in Table 1 is only the reduced number of page faults due to the reduction of the write/read false-sharing effect. Among the four applications, only BT and $Water$ have the write/read false-sharing effect, and 6% of page faults in BT are due to the write/read false-sharing effect. We have collected the total number of page faults that are due to false-sharing effect inside critical regions, and the results are shown in Table 2.

APP	TPF	RFS	TFS
TSP	1002	0	58
QS	3084	0	2
BT	13963	792	4347
Water	12046	3	6

Table 2: Number of page faults due to the false-sharing effect

In Table 2, TPF is the total number of page faults; RFS is the number of page faults that are due to the write/read false-sharing effect inside critical regions; TFS is the number of page faults that are due to all false-sharing effects (including write/read and write/write false-sharing) inside critical regions. From Table 2, we know the reduced write/read false-sharing effect in our current implementation of VC is only a small portion of the total false-sharing effect (0% for TSP, 0% for QS, 18.2% for BT, and 50% for $Water$). Further research will be needed to remove the write/write false-sharing effect in VC implementation.

Except for BT, however, the performance of most applications is less affected by the false-sharing effect inside the critical regions, considering the ratio of the total number of page faults that are due to the false-sharing effect inside critical regions to the total number of page faults (5.8% for TSP, 0.06% for QS, 31% for BT, and 0.05% for $Water$). Thus there is not much potential for VC to further improve their performance if the false-sharing effect inside the non-critical regions is not considered. Detecting the accurate NRVs is an important task to remove the false-sharing effect inside the non-critical regions and to further improve the performance of the applications.

6 Conclusion

In this paper we have proposed a novel *View-based Consistency (VC)* model for DSM and discussed important issues for its implementation. Compared with other DSM consistency models, this model can achieve data selection without user annotation and reduce more of false sharing effects. Its only consistency requirement is that all the data objects in a processor's new view must be updated during view transition. The further relaxation on consistency requirement enables VC to have more room for optimization in the implementation of DSM. The VC model can guarantee the same execution result as the Sequential Consistency model for data-race-free programs. Performance results have shown that for all our applications the VC model outperforms the LRC model. We have also demonstrated that the extra overhead of view maintenance is trivial.

The VC model appears to be the appropriate framework for future DSM implementation, since VC has the potential performance advantage to achieve the maximum relaxation of constraints on update propagation and execution for data-race-free programs. It is generic enough for the previous models to be considered as limited versions of the VC implementation. As a consequence it would appear that future implementation of DSM would best be devoted to optimizing data selection in the VC model.

Further research should be carried out under the framework of the VC model. (1) Accurate detection of NRVs. Run-time and compile-time techniques need to be developed for the detection. These techniques are different from previous work on compile-time optimization, e.g.[7], or run-time optimization, e.g.[18], which work at the level of update propagation protocol in LRC, instead of the level of a consistency model. (2) Efficient view representation. The current implementation uses a page as the basic unit of a view. A page is too coarse for the representation of views and may result in propagation of useless updates on the same page. (3) Reduction of the write/write false sharing. A new update representation scheme, rather than the single-writer and the multiple-writer schemes, is needed to reduce the write/write false sharing.

Acknowledgments

The authors would like to thank Kin Cheung and Malcolm Fraser for their assistance in the implementation of the VC model. The research is partially supported by an Otago Research Grant.

References

[1] C. Amza, et al.: "TreadMarks: Shared memory computing on networks of workstations", *IEEE Computer,* 29(2), pp.18-28, February 1996.

[2] B.N. Bershad, et al.: "Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors", *CMU Technical Report* CMU-CS-91-170, September 1991.

[3] B.N. Bershad, et al.: "The Midway Distributed Shared Memory System", *In Proc. of IEEE COMPCON Conference,* pp.528-537, 1993.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel: "Implementation and performance of Munin", *In Proceedings of the 13th ACM Symposium on Operating Systems Principles,* pp.152-164, Oct. 1991.

[5] 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), pp.205-243, August 1995.

[6] 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.

[7] S. Dwarkadas, et al.: "An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System", *In Proc. of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems,* Oct. 1996.

[8] B. Fleisch and R.H. Katz: "Mirage: A coherent distributed shared memory design", *In Proc. of the 12th ACM Symposium on Operating Systems Principles,* pp.211-223, Dec. 1989.

[9] 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,* pp.15-26, May 1990.

[10] 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,* Lecture Notes in Computer Science 1345, pp.98-109, Springer Verlag, 1997.

[11] Z. Huang, C. Sun, S. Cranefield, and M. Purvis: "Overview of weak sequential consistency models for distributed shared memory", in *Proc. of the 10th International Conference on Computing and Information*, November 2000.

[12] Z. Huang, C. Sun, M. Purvis, and S. Cranefield: "View-based Consistency and False Sharing Effect in Distributed Shared Memory", *Operating Systems Review*, 35(2), April 2001.

[13] 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.

[14] P. Keleher: "Lazy Release Consistency for Distributed Shared Memory", *Ph.D. Thesis,* Dept of Computer Science, Rice Univ., 1995.

[15] L. Lamport: "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Transactions on Computers*, 28(9), pp.690-691, September 1979.

[16] K. Li, P. Hudak: "Memory Coherence in Shared Virtual Memory Systems", *ACM Trans. on Computer Systems*, Vol. 7, pp.321-359, Nov. 1989.

[17] D. Mosberger: "Memory consistency models", *Operating Systems Review*, 17(1), pp.18-26, Jan. 1993.

[18] C.B. Seidel, R. Bianchini, and C.L. Amorim: "The Affinity Entry Consistency Protocol", *In Proc. of the 1997 International Conference on Parallel Processing*, August 1997.

[19] 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,* pp.572-581, Amsterdam, May 1998.