

View-based Consistency and False Sharing Effect in Distributed Shared Memory

Z. Huang[†], C. Sun[‡], M. Purvis[†], and S. Cranefield[†]

[†]Departments of Computer & Information Science
University of Otago
Dunedin, New Zealand

[‡]School of Computing & Information Technology
Griffith University
Brisbane, Qld 4111, 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 requires that the data objects of a view are made up-to-date only before a processor accesses them. Compared with other memory consistency models, the View-based Consistency model can achieve data selection without user annotation and reduce more false sharing effect.

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 [14, 7, 2, 5, 4, 1, 16, 17]. A DSM system provides 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 the DSM systems more convenient to use and more efficient to implement [9, 16]. In this paper, we propose a View-based Consistency (VC) model for DSM, which is a significant step towards our goal.

2 Motivation

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 [15]. The Sequential Consistency (SC) model [13] 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*. This means that in a SC-based DSM system, memory accesses from all processors may be interleaved in any sequential order which 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 seeing them in order. Many parallel applications regulate their accesses to shared data by synchronization, so not all valid interleavings of their memory accesses are relevant to their real executions. Therefore, it is actually 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 *processor*, and the *data* for making shared memory updates public to improve the performance while still appearing to be sequentially consistent. For example, 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 *serialized* among competing processors by means of synchronization operations to avoid data races. Under these circumstances, the following three basic 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 another 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 [6, 8, 12, 3, 11], 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 [16]. We argued in reference [16] that a consistency model should not impose any extra burden on programmers, such as annotation of lock-data association in Entry Consistency (EC) [3] and scope-data association in Scope Consistency (ScC) [11]. In this paper, we propose a *View-based Consistency (VC)* model which, besides time selection and processor selection, can transparently achieve data selection.

3 View-based Consistency (VC)

During the execution of a parallel program, multiple processors work on and 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/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.

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 barriers¹. 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, 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 $V_1 \cap V_2$ only contains read-only data objects; otherwise a data race happens. Fig. 1 shows a snapshot of *views* of processors in shared memory. The overlapped part of different *views* only contains read-only data objects.

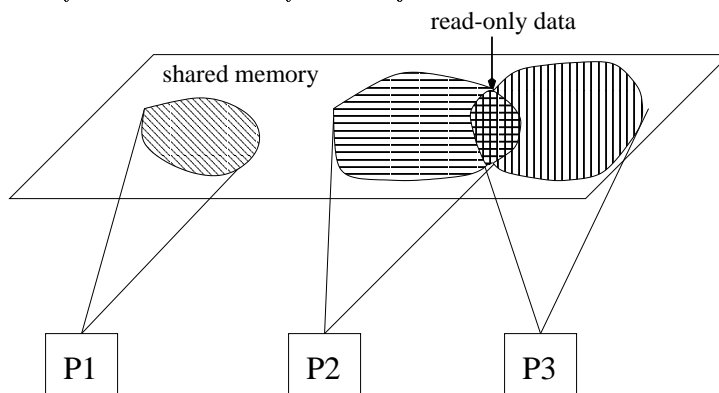


Figure 1: A snapshot of processors' *views* in a program execution

Many DSM systems require explicit *acquire*, *release* and *barrier* in DSM programs to achieve weak Sequential Consistency. An execution of such a DSM program can be 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*. 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 because of nested critical regions.

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.

¹A 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*.

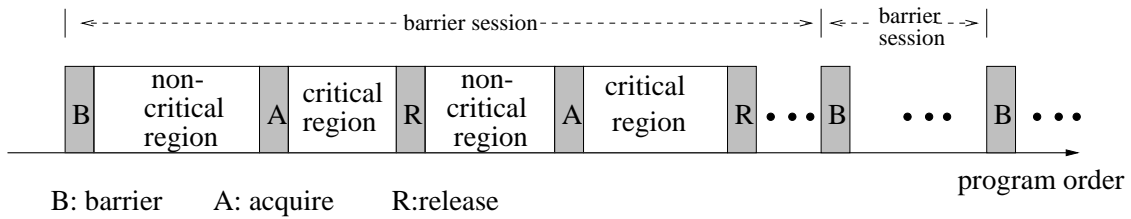


Figure 2: A view of a program execution based on region

2. Explicit acquisition by calling the *acquire* primitive. Exclusive access is guaranteed by 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.

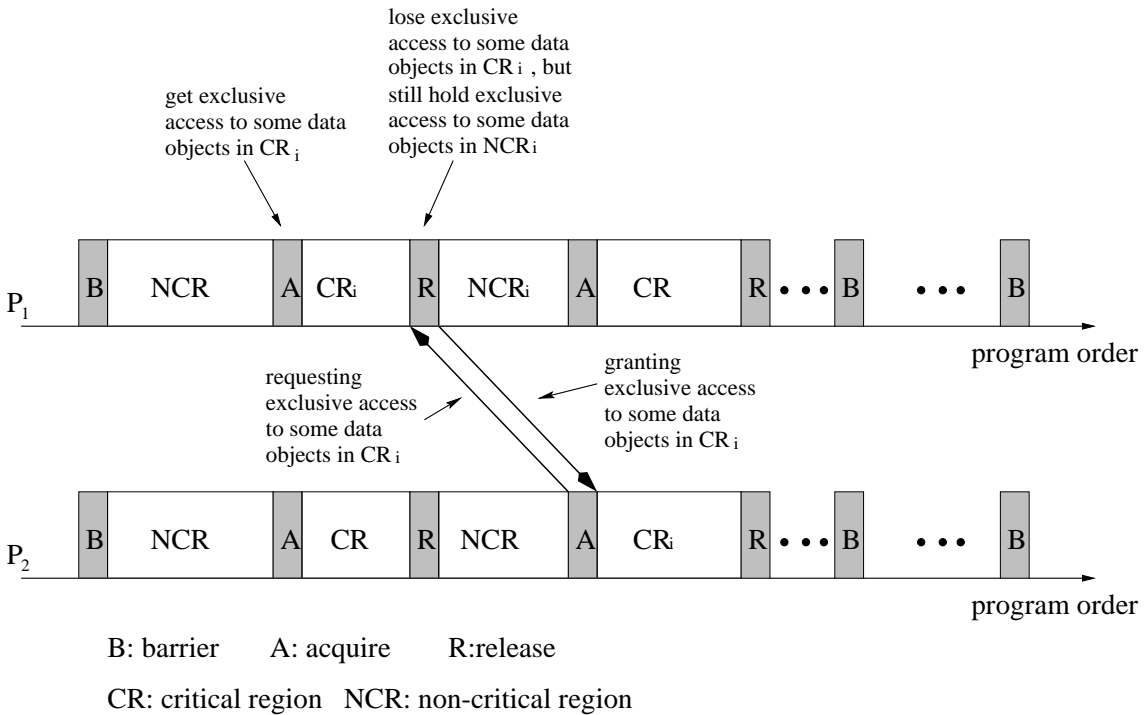


Figure 3: Views and their transitions

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 (CRV)* and *Non-critical Region Views (NRV)*. 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.

Definition 1 *Critical Region View (CRV)*

Besides read-only data objects, a processor’s CRV includes 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)*

Besides read-only data objects, a processor’s NRV includes 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 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 made up-to-date.
- The VC model guarantees the same execution result as the Sequential Consistency for a data-race-free DSM program. *Read/write* data objects are accessed sequentially by exclusive access and they are made up-to-date once the exclusive access to them is gained.
- 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.

4 Comparison of related models

Among the different consistency models, only ScC [11] and EC [3] 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 *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 $S = \{s_1, \dots, s_n\}$. Because it is quite common for a processor to access only some data objects in S after it acquires lock l , we can suppose the accessed data objects are $S' \subset S$. 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 sharing² in page-based DSM. It can reduce the false sharing effect in the following two levels:

1. Restrict the propagation of invalidation notices. Only the invalidation notices that are useful for making up-to-date the data objects in a processor's new *view* are propagated to the processor;
2. Restrict the effective scope of invalidation notices. Only the invalidation notices that are useful for making up-to-date the data objects of the current *view* of a processor are effective in the current region of the processor.

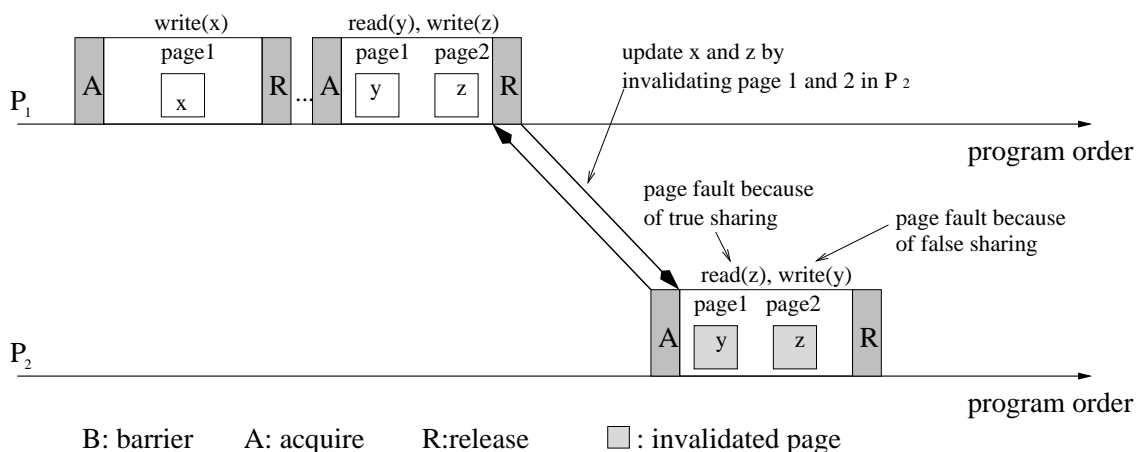


Figure 4: False sharing effect in LRC

²False sharing occurs when two processors update different shared data objects that lie in the same memory consistency unit (e.g. a page).

Fig. 4 presents an example of the false sharing effect in *Lazy Release Consistency (LRC)* [12]. In the example, data objects x and y lie in the same page. Processor P_1 updates x in the first critical region and then updates z in the second critical region. After P_1 exits from the second critical region, P_2 enters the same critical region. Though P_2 only accesses data objects y and z , LRC updates data objects x and z in P_2 by invalidating both page 1 and page 2. When P_2 accesses y , it has a page fault on page 1, which is caused by false sharing.

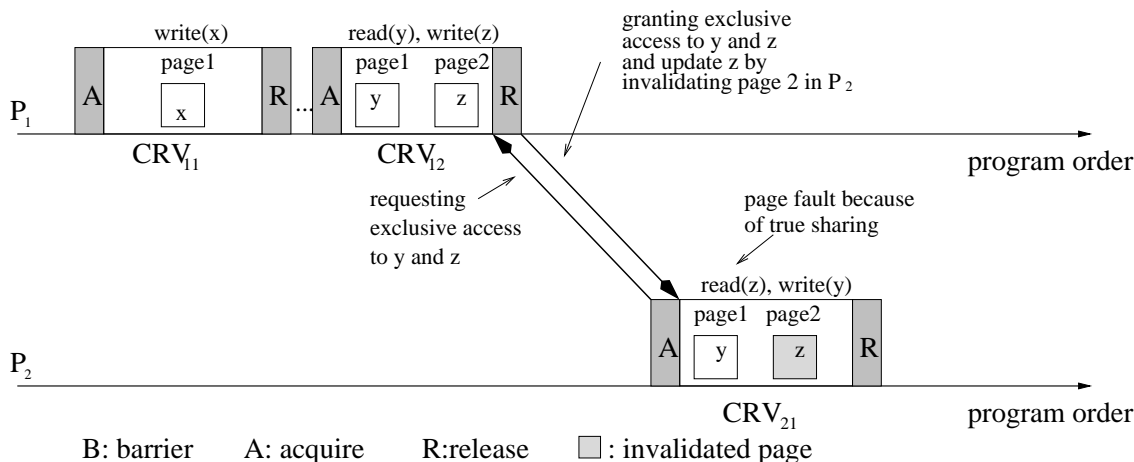


Figure 5: Reduce the false sharing effect in VC (1)

For the same example, Fig. 5 shows how VC removes this effect of false sharing. VC only requires updating the data objects to which a processor has exclusive access. In the example P_2 has got exclusive access to data objects y and z when entering a critical region. Since y is not updated by P_1 , VC only updates z by invalidating page 2. By preventing the propagation of the invalidation notice for page 1, which has no use for updating y and z in the *view* of P_2 , VC can avoid the page fault on page 1, which occurs in LRC when P_2 accesses y .

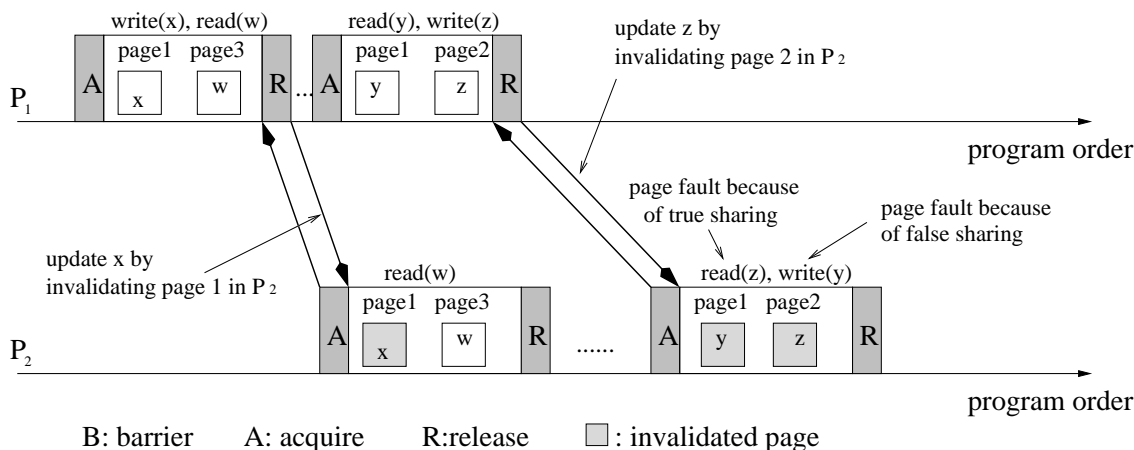


Figure 6: False sharing effect in both EC and ScC

While ScC [11] and EC [3] can remove the above false sharing effect, they may fail to do so in other situations. Fig. 6 presents another example of the false sharing effect in these models. In the example data objects x and y lie in the same page. P_1 updates x in the first critical region and then updates z in

the second critical region. According to EC and ScC, x and w are associated with the first critical region, and y and z are associated with the second critical region. When P_2 enters the first critical region, x is updated by invalidating page 1; however x is not accessed by P_2 . The invalidation notice for page 1 is still effective in the second critical. Then when P_2 accesses y in the second critical region, a page fault on page 1 occurs, which is caused by the invalidation notice produced for updating x . This page fault is of no use for updating y and it is a false sharing effect.

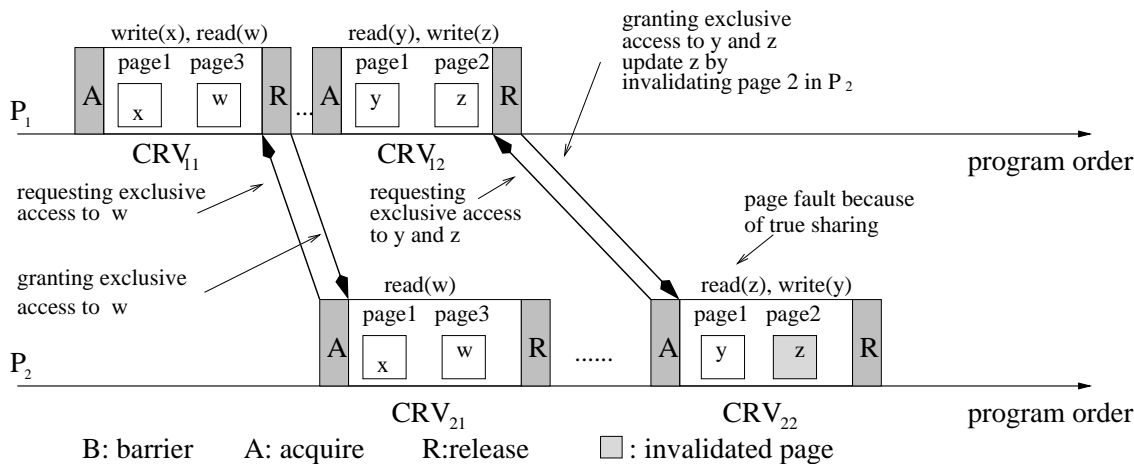


Figure 7: Reduce the false sharing effect in VC (2)

Once again, VC can remove the above false sharing effect by restricting the propagation of invalidation notices or by restricting the effective scope of invalidation notices. In Fig. 7 since x is not in the *view* CRV_{21} of P_2 , it should not be updated in P_2 . So the invalidation notice for page 1 is not propagated to P_2 , and P_2 has no page fault on page 1 in the second critical region. Moreover, suppose x is mistakenly detected as in the *view* CRV_{21} and then the invalidation notice for page 1 is propagated to P_2 . If we can detect that the invalidation notice for page 1 has no use for updating y and z in the *view* CRV_{22} , VC makes it ineffective in the second critical region. So there is no page fault when P_2 accesses y , and the above false sharing effect is removed.

In the following section, we discuss the issues in the implementation of the VC model.

5 Implementation issues

There are two technical issues in the implementation of VC. One is *view detection*, and the other is *view transition*. View 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 make all the data objects of its new *view* up-to-date. Any correct implementation of the VC model should guarantee that before a processor enters a new region, *view* detection and *view* transition are achieved correctly.

View detection can be implemented at compile-time or at run-time. Through analysis of the program at compile-time, data dependency can be detected and could be used for *view* detection. At run-time we can record the updated data objects in every region and calculate the *view* in each region, though sometimes the calculated *view* is greater than the real one. The more accurate the *view* is detected, the more data selection is achieved in the VC model. We will discuss the techniques for *view* detection in a

later paper.

In *view* transition, to make data objects up-to-date, we can use either the update protocol or the invalidation protocol [16]. If the update protocol is used, we should propagate merely the updates on data objects of the new *view* to the processor. If the invalidation protocol is used, we should propagate merely the invalidation notice involved with the new *view*, and restrict the effect of other unrelated invalidation notices. These techniques can be developed by adapting the protocols proposed in [9, 10].

6 Conclusion

In this paper we have proposed a novel *View-based Consistency (VC)* model for DSM. Compared with other DSM consistency models, this model can achieve data selection without user annotation and reduce more false sharing effect by only updating data objects in the *view* of a processor. Like some other weak Sequential Consistency models, the VC model can guarantee the same execution result as the Sequential Consistency model for data-race-free programs. Further research should be carried out on automatic *view* detection and *view* transition so as to implement VC efficiently.

References

- [1] C.Amza, et al: "TreadMarks: Shared memory computing on networks of workstations", *IEEE Computer*, 29(2):18-28, February 1996.
- [2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum: "Orca: A language for parallel programming of distributed systems", *IEEE Transactions on Software Engineering*, vol. 18, pp.190-205, March 1992.
- [3] 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.
- [4] B.N. Bershad, et al: "The Midway Distributed Shared Memory System", *In Proc. of IEEE COMPCON Conference*, pp.528-537, 1993.
- [5] 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*, Pages 152-164, Oct. 1991.
- [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] 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.
- [8] 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.
- [9] 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, pp.98-109, Springer Verlag, 1997.

- [10] Z. Huang, C. Sun, and A. Sattar: "Exploring regional locality in distributed shared memory", *In Proc. of 1998 Asian Computing Science Conference*, Lecture Notes in Computer Science 1538, pp.142-156, Springer Verlag, 1998.
- [11] 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.
- [12] P. Keleher: "Lazy Release Consistency for Distributed Shared Memory", *Ph.D. Thesis*, Dept of Computer Science, Rice Univ., 1995.
- [13] L. Lamport: "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Transactions on Computers*, 28(9):690-691, September 1979.
- [14] K.Li, P.Hudak: "Memory Coherence in Shared Virtual Memory Systems", *ACM Trans. on Computer Systems*, Vol. 7, pp.321-359, Nov. 1989.
- [15] D. Mosberger: "Memory consistency models", *Operating Systems Review*, 17(1):18-26, Jan. 1993.
- [16] 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.
- [17] A.S. Tanenbaum: *Distributed Operating Systems*, Prentice Hall, 1995.