Selection-based Weak Sequential Consistency Models for Distributed Shared Memory

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

‡School of Computing & Information Technology
Griffith University, Brisbane, Australia

Abstract

Based on time, processor, and data selection techniques, a group of Weak Sequential Consistency models have been proposed to improve the performance of Sequential Consistency for Distributed Shared Memory. These models can guarantee Sequential Consistency for data-race-free programs that are properly labelled. This paper reviews and discusses these models in terms of their use of the selection techniques. Their programmer interfaces are also discussed and compared. Among them the View-based Consistency model is recognized as the model that can offer the maximum performance advantage among the Weak Sequential Consistency models. An implementation of the View-based Consistency model has been given. Finally this paper suggests future directions of implementation effort for Distributed Shared Memory.

Key Words: Distributed Shared Memory, Weak Sequential Consistency, View-based Consistency

1 Introduction

A Distributed Shared Memory (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. Distributed Shared Memory has become an active area of research in parallel and distributed computing with the goals of making DSM systems more convenient to program and more efficient to implement [15, 5, 4, 2, 18].

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 [16]. The Sequential Consistency (SC) model [14] 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 global 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 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 generated and executed at one processor be propagated to and executed at other processors instantaneously. This implementation is correct but it suffers from serious performance problems [19].

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 interleavings 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 *processor*, and the *data* for making shared memory updates public

to improve the performance while still appearing to be sequentially consistent [18]. 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, as shown in Fig. 1.

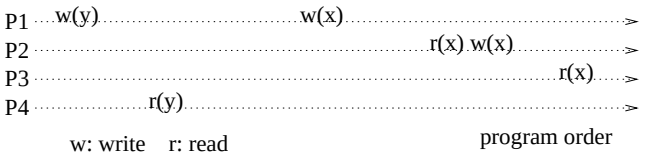

Figure 1: A scenario of a DSM program

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 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 Weak Sequential Consistency (WSC) models have been proposed [7, 9, 13, 3, 12], which perform one or more of the above three selection techniques. WSC models can be also called conditional Sequential Consistency models, which can guarantee Sequential Consistency for some class of programs that satisfy the conditions imposed by the models. These models take advantage of the synchronizations in data-race-free programs and relax the constraints on update propagation and execution. That means, updates generated and executed by a processor may not be propagated to and executed at other processors immediately. Most WSC models can guarantee Sequential Consistency for data-race-free programs that are *properly labelled* [7] (i.e., explicit primitives, provided by the system, should be used for synchronization in the programs).

In this paper we demonstrate the above three selections can help improve the performance of the WSC models. Previous papers [16, 1] have discussed consistency models, but none of them analyzed the complete set of WSC models in terms of the three selection techniques. The rest of this paper is organized as follows. Section 2 describes different WSC models in terms of the three selection techniques and the programmer interface. Section 3 discusses the differences and relationships among the WSC models. Section 4 gives one implementation of the View-based Consistency model. Finally Section 5 concludes the paper with future implementation effort on DSM.

2 WSC models

In the following sections we evaluate the WSC models in terms of the three selections and the programmer interface.

2.1 Weak Consistency

The Weak Consistency (WC) model [7], proposed in 1986, was the first WSC model. Rather than requiring an update to be propagated to and executed at other processors immediately, WC requires that all previously-generated updates be propagated to and executed at all processors before a synchronization primitive is allowed to be executed. Thus propagation of updates can be postponed until a synchronization primitive is to be executed.

WC can achieve time selection by propagating updates to other processors only at *synchronization time*, rather than at every update time. With time selection, updates can be accumulated and only the final results are propagated in batches at synchronization time. In this way, the number of messages in WC implementations can be greatly reduced compared to that in strict SC implementations.

WC requires programmers to use explicit primitives, such as *acquire* and *release*, for synchronization. No data race on ordinary data objects is allowed in the program. If a program meets these requirements, WC can guarantee Sequential Consistency for it.

2.2 Eager Release Consistency

The Eager Release Consistency (ERC) model [9] improves WC by removing the update propagation at *acquire* time. It

requires that all previously-generated updates must be propagated to and executed at all processors before a *release* is allowed to be executed. So update propagation can be postponed until a *release* is to be executed.

ERC takes time selection one step further than the WC model by distinguishing two different synchronization primitives: *acquire* and *release*, which are the entry and exit of a critical region respectively. ERC requires that updates be propagated to other processors only at *release* time. In other words, ERC is more time-selective than the WC model by propagating updates only at the exit of a critical region, instead of at both the entry and exit of a critical region as in the WC model, thus further reducing the number of messages in the system.

ERC has the same programmer interface as WC, though it removes update propagation at *acquire* time. It can guarantee Sequential Consistency for data-race-free programs that are *properly labelled*. A formal proof of this conclusion is provided in reference [9].

2.3 Lazy Release Consistency

The Lazy Release Consistency (LRC) model [13] does not require the update propagation at *release* time. It postpones the update propagation until a processor calls an *acquire*, at which time it knows which processor is the next one to need the updates. So LRC requires that before any access after an *acquire* is allowed to be executed, all previously-generated updates must be propagated to and executed at the processor executing the *acquire*. The Lazy Release Consistency (LRC) model [13] improves the ERC model by performing both time selection and processor selection.

LRC can achieve time selection similar to ERC, except the update propagation is further postponed until another processor has successfully executed an *acquire*.

LRC can achieve processor selection by postponing the update propagation until *acquire* time. At successful *acquires*, the DSM system is able to know precisely which processor is the next one to access the shared data objects, so updates can be propagated only to that particular processor (or no propagation at all if the next processor is the current processor). By sending updates only to the processor that has just entered a critical region, more messages can be reduced in the LRC model.

LRC has the same programmer interface as WC and ERC. It can guarantee Sequential Consistency for data-race-free programs that are *properly labelled*.

2.4 Entry Consistency

The Entry Consistency (EC) model [3] tried to remove the propagation of useless updates in LRC by requiring the programmer to annotate association between ordinary data objects and synchronization data objects (e.g. locks). When a processor acquires a synchronization data object, only the updates of the data objects that are associated with the synchronization data object are propagated to the processor. More precisely we say, EC requires that before any access after an *acquire* is allowed to be executed, all previously-generated updates of data objects that are associated with the corresponding synchronization data object, must be propagated to and executed at the processor executing the *acquire*.

EC achieves the same time selection and processor selection as LRC, since updates are propagated only to the next processor calling an *acquire*.

EC achieves data selection by only propagating updates of data objects that are associated with a synchronization data object. The association, provided by the programmer, helps the EC model remove the propagation of some updates useless to a processor. With additional data selection, EC can be more efficient than LRC.

In addition to requiring a program to be data-race free and properly labelled, EC requires the programmer to annotate the association between ordinary data objects and synchronization data objects in the program. If the association is correct, EC can guarantee Sequential Consistency for data-race-free programs; otherwise, Sequential Consistency is not guaranteed. The annotation of the association is normally regarded as an extra burden on the programmer.

2.5 Scope Consistency

The Scope Consistency (ScC) model [12] is very similar to EC, except it can partially automate the association between ordinary data objects and synchronization data objects by introducing the concept of *consistency scope*. ScC requires that before any access after an *acquire* is allowed to be executed, all previously-generated updates of data objects that belong to the corresponding scope, must be propagated to and executed by the processor executing the *acquire*.

Like EC, ScC only propagates the updates of data objects that are in the current consistency scope. The difference is that a consistency scope can automatically establish the association between critical regions and data objects. For

non-critical regions, however, scopes have to be explicitly annotated by the programmer.

ScC achieves the same time selection, processor selection and data selection as EC. So it can offer the same performance advantages as EC if the scopes are well detected or annotated in the program.

ScC improves the programmer interface of EC by requiring programmers to associate scopes with code sections, instead of data. For critical regions, scopes can be automatically associated; but the programmer has to annotate the scopes explicitly for non-critical regions. If the annotation is not correct, ScC can not guarantee Sequential Consistency for the program. ScC also requires the program to be data-race free and properly labelled.

2.6 View-based Consistency

The View-based Consistency (VC) model [11] is proposed to achieve data selection transparently without programmer annotation. A *view* is a set of ordinary data objects that a processor has the right to access in a data-race-free program. A processor's view changes when it moves from one region to another by calling *acquire* and *release*. VC requires that before a processor is allowed to enter a critical region or a non-critical region, all previously-generated updates of data objects that belong to the corresponding view, must be propagated to and executed at the processor.

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.

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.

VC achieves the same time selection and processor selection as LRC. It can be more selective than EC and ScC in terms of data selection.

VC has the same programmer interface as LRC, ERC, and WC. It can guarantee Sequential Consistency for data-race-free programs that are properly labelled. To achieve data selection transparently, VC relies on techniques for automatic view detection [11].

3 Discussion

From the history of WSC models we know that constraints on update propagation and execution have become more and more relaxed. This relaxation allows the DSM systems to perform time, processor, and data selections. Table 1 gives a summary of WSC models in terms of the three selection techniques.

Model	Time Sel.	Proc. Sel.	Data Sel.
SC	No	No	No
WC	Sync	No	No
ERC	Release	No	No
LRC	Acquire	Next proc.	No
EC	Acquire	Next proc.	Lock-data assoc. (programmer annot.)
ScC	Acquire	Next proc.	Scope-data assoc. (programmer annot.)
VC	Acquire	Next proc.	View-data assoc. (auto detection)

Table 1: Selection techniques used in existing consistency models

From the discussion in previous sections we also know that all existing WSC models achieve time/processor/data selection by requiring programmers to annotate the programs manually so that time/processor/data selection can be combined with synchronization primitives. For example, the ERC model requires programs to be properly labelled by system-provided synchronization primitives, so that the DSM system is explicitly notified of the entry and exit of a critical region and can thereby select the exit time to propagate updates. The EC model, furthermore, requires the programmer to associate synchronization data objects explicitly with ordinary data objects to achieve data selection. The ScC model made one step toward (partially) transparent data selection by taking advantage of the consistency scopes implicitly defined by synchronization primitives, but programmers may still have to define additional consistency scopes explicitly in programs in order to guarantee Sequential Consistency. We should be very cautious of the programmer annotation which may impose an extra burden on programmers and increase the complexity of parallel programming.

We distinguish two types of programmers' annotations: one is the synchronization 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* synchronization primitives in the ERC, LRC, EC and ScC models, the DSM system can take advantage of them to achieve time/processor selection without imposing an additional burden on programmers. However, for the second type of annotations, such as the association between synchronization data objects and ordinary 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 increase the complexity of parallel programming. They should be replaced by automatic associations via run-time detection (and/or compile-time analysis).

Among all these WSC models, only VC can achieve data selection without imposing any extra burden on the programmer. In an ideal implementation, where views can be accurately detected, VC can achieve the maximum data selection. In another word, VC achieves the maximum relaxation of constraints on update propagation and execution for data-race-free programs. So VC can be a reference model where other WSC models can know their distance from the ideal relaxation of constraints.

Based on the above analysis, the VC model appears to be a generic and appropriate framework for future DSM implementation. Since the VC model incorporates the features of the previous WSC models (shown in Table 1), the previous WSC models can 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.

4 Implementation of VC

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 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 [2], 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 [6].

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. According to our experimental results this additional overhead is trivial [11].

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. [18]

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[18]. It uses the update protocol for the modified pages in CRVs, but the invalidation protocol for the modified pages in NRVs.

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, we have to tolerate the write/write false-sharing effect. Thus the write/write false-sharing effect has not been removed in our implementation.

5 Conclusions and future directions

WSC models have relaxed the constraints on update propagation and execution for data-race-free programs that are properly labelled, but they can still guarantee Sequential Consistency for those programs. We have analyzed each of them in terms of the three selections and the programmer interface. From the discussion we know that more and more selections are performed in WSC models (as one reads top-down in Table 1), but data selection is performed in EC and ScC at the price of a complex programmer interface which imposes an extra burden on the programmer. The VC model has been proposed to remove such a burden by means of automatic view detection. Besides time selection and processor selection, VC tries to achieve the maximum data selection without imposing any extra burden on the programmer. It can serve as a reference model for other WSC models and as a target model for DSM implementation, because it is the model that has the maximum performance advantage among the WSC models.

We believe that future implementation efforts in DSM will be focused on how to achieve the performance advantage of VC with the least overhead. 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.[8], or run-time optimization, e.g.[17], 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.

References

[1] S.V. Adve and M.D. Hill: “A unified formalization of four shared-memory models”, *IEEE Transactions on Parallel and Distributed Systems*, vol.4(6), pp.613-624, June 1993.

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

[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] 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.

[7] 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.

[8] 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.

[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, 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] Z. Huang, C. Sun, M. Purvis, and S. Cranefield: "View-based Consistency and Its Implementation", *In Proc. of the First IEEE/ACM Symposium on Cluster Computing and the Grid*, pp.74-81, IEEE Computer Society, 2001.

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

[19] A.S. Tanenbaum: *Distributed Operating Systems*, Prentice Hall, 1995.