Department of Computer Science,
University of Otago

Technical Report OUCS-2004-03

A View-based Consistency Model based on Transparent Data Selection in Distributed Shared Memory

Authors:
Z. Huang
Department of Computer Science, University of Otago
C. Sun
School of Computing & Information Technology Griffith University, Brisbane, Australia
S. Cranefield, and M. Purvis
Department of Information Science, University of Otago

Status: Not yet submitted

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/trseries/

A View-based Consistency Model based on Transparent Data Selection in Distributed Shared Memory

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, in which a new concept, view, is coined. A view is a set of data objects that a processor has the right to access in the shared memory. The View-based Consistency model only requires that the data objects of a processor's view are updated before a processor accesses them. In this way, it can achieve the maximum relaxation of constraints on modification propagation and execution in data-race-free programs. This paper first briefly reviews a number of related consistency models in terms of their use of three techniques – time, processor and data selection – which each eliminate some unnecessary propagation of memory modifications while guaranteeing sequential consistency for data-race-free programs. Then, we present the View-based Consistency model and its implementation. In contrast with other models, the View-based Consistency model can achieve transparent data selection without programmer annotation and can offer the maximum performance advantage. Differences among related work are discussed through illustrative examples.

Performance evaluation has shown that our implementation of the View-based Consistency model outperforms the Lazy Release Consistency model, and that the View-based Consistency model has advantages over optimal consistency protocols such as the Affinity Entry Consistency protocol. Finally we summarises our contributions and points out our future direction of implementation effort for distributed shared memory systems.

Key Words: Distributed Shared Memory, Sequential Consistency, Relaxed Sequential Consistency, Entry Consistency, Scope Consistency, Lazy Release Consistency, Time selection, Processor selection, Data selection, View-based Consistency, View detection, View transition, False Sharing

1 Introduction

A Distributed Shared Memory (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. 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 [21, 12, 20, 9, 7, 6, 3, 14, 15].

The consistency model of a DSM system specifies ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems' programming convenience and implementation efficiency [22]. The Sequential Consistency (SC) model [19] 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* [19]. This means that in an SC-based DSM system, memory accesses from different processors may be interleaved in any sequential order that is consistent with each processor's order of memory accesses, and the orders of memory accesses observed by different processors are the same. One way to strictly implement the SC model is to ensure all memory modifications be totally ordered and memory modifications generated and executed at one processor be propagated to and executed in that order at other processors instantaneously. This implementation is correct but it suffers from serious performance problems [25].

In practice, not all parallel applications require each processor to see all memory modifications 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 modifications to **every** other processor (with a copy of the shared data) at **every** memory modification time. Under certain conditions, the DSM system can select the *time*, the *processor*, and the *data* for propagating shared memory modifications in order to improve the performance while still appearing to be sequentially consistent [24]. 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, P_2, and P_3 share a data object $x1$, and P_1 and P_4 share a data object y, as shown in Fig. 1. The data objects z and $x2$ are shared among processors at a later time not shown in this scenario.

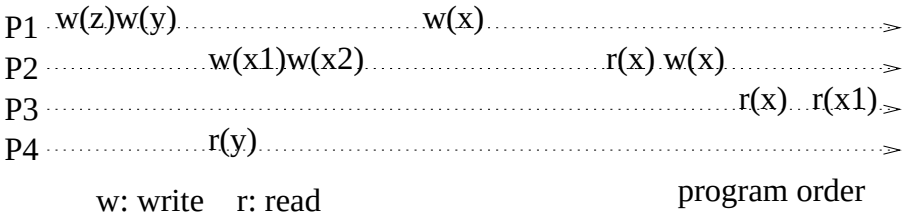

Figure 1: A scenario of a DSM program

Suppose all memory accesses to shared data objects 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 [24]. **Time selection**: Modifications on a shared data object by one processor are propagated to other processors *only at the time* when the data object is to be read by them. For example, modifications 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. **Processor selection:** Modifications 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, modifications on x by P_1 may be propagated to P_2 (but not to P_3) if P_2 is the next one in sequence to read x. **Data selection:** Processors propagate to each other *only those shared data objects* that 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 Relaxed Sequential Consistency (RSC) models have been proposed [10, 13, 18, 5, 17], which perform one or more of the

above three selection techniques. RSC models can be also called conditional Sequential Consistency models because they 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 modification propagation and execution. That means modifications generated and executed by a processor may not be propagated to and executed at other processors immediately. Most RSC models can guarantee Sequential Consistency for data-race-free programs that are *properly labelled* [13] (i.e., explicit primitives, provided by the system, should be used for synchronization in the programs).

However, in these RSC models data selection is achieved by programmer annotation. The programmer is required to provide annotations of the association between data objects and synchronization objects, such as locks and barriers, so that the DSM system can know which data objects should be propagated when a synchronization object is accessed. This type of annotation is an extra burden on the programmer and causes an increase of the complexity of parallel programming. The ease of programming would be improved if the need for human annotation could be replaced with automatic detection of the association. Automatic detection can be done at run time and/or compile time. Compile-time detection may cause a slower compiling process, but has little run-time overhead. Run-time detection may have more run-time overhead, but can get more accurate information. This paper proposes a novel View-based Consistency (VC) model which can achieve, with run-time automatic view detection, data selection transparently in addition to time and processor selection.

The rest of this paper is organised as follows. Section 2 briefly presents the background to our work, especially the classification of RSC models in terms of time, processor, and data selection. Section 3 presents the VC model [16], whose properties are described in detail. Section 4 discusses implementation issues and presents our implementation of the VC model. Section 5 describes the differences among related work through illustrative examples. Section 6 presents and evaluates performance results. Finally, the major contributions of this paper and some suggestions for future work are summarized in Section 7.

2 Relaxed Sequential Consistency Models

During the execution of a DSM parallel program, multiple processors communicate with each other through the virtual shared memory. In shared memory some data objects are *read-only*, and some are *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.

RSC models distinguish *synchronization* data objects from *ordinary* data objects in shared memory. Synchronization data objects, such as locks and barriers[1], are those that are explicitly used to enforce exclusive access to other data objects. The rest of the data objects in shared memory are called ordinary data objects. RSC models require sequential consistency for the synchronization data objects in synchronization primitives, such as *acquire* and *release*. RSC models assume there is only one memory copy for the synchronization data objects and accesses to the same data object are executed sequentially. For the storage of ordinary data objects, RSC models adopt a replicated architecture to improve performance under the distributed environment. Copies are stored in the local memory of each participating processor. Modifi cations generated and executed locally at some time by one of the processors may be propagated to and executed by other processors at a later time. This delay may cause the violation of sequential consistency, but, if handled properly, can be used to improve the performance of the DSM system. In order to achieve sequential consistency while taking advantage of the delay, RSC models do not allow any data race on ordinary data objects in the program. Exclusive access to ordinary data objects has to be guaranteed by explicitly using synchronization primitives. If data races on ordinary data objects are prevented by using the synchronization primitives in the program, most RSC models can guarantee sequential consistency for the ordinary data objects. In summary, RSC models strictly implement sequential consistency for synchronization data objects, while they implement sequential consistency for ordinary data objects under the condition that there is no data race on them. This can be guaranteed by *properly labelling* the program with the synchronization primitives. A program is said to be *properly labelled* if there is no data race in the program due to the use of the synchronization primitives such as *acquire, release* and *barrier* [13]. An execution of such a DSM program can

[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*.

be viewed as a sequence of *barrier sessions* as shown in Fig. 2.

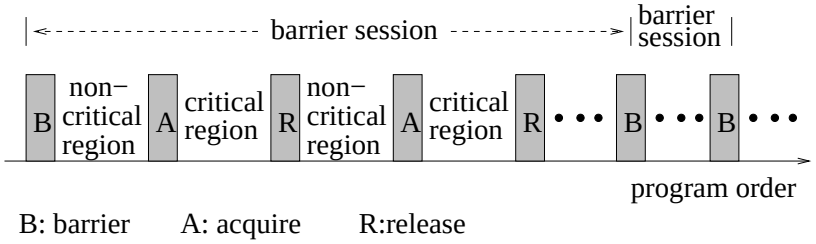

Figure 2: A view of a program execution based on the concept of region

A barrier session begins with a *barrier* and ends with another *barrier*. Inside a barrier session there is a sequence of regions comprising critical regions and non-critical regions. These regions are delimited by *acquire, release* and *barrier* primitives. A critical region is a section of code that ensures that only one processor executes the section of code at any time. It begins with an *acquire* and ends with a *release*. A non-critical region begins with a *release* (the outermost one when there are nested critical regions) or a *barrier* and ends with an *acquire* (the outermost one when there are nested critical regions) or a *barrier*. A non-critical region does not overlap with any critical region, but a critical region may be nested within another critical region.

In an execution of a properly labelled program, accesses to data objects may have some causal ordering. The causal ordering is essential to the definition of data races and data-race-free programs. The RSC models take advantage of the assumption that programs are data-race free to optimize modification propagation and execution.

Definition 1 *Causal Ordering Relation "$\rightarrow$"*

At a high level we can model a DSM program as a set of sequences of actions (one sequence for each processor). These actions include memory accesses and synchronization primitives such as *acquire*, *release* and *barrier*. Given two actions a_1 and a_2, generated by processors i and j, respectively, then $a_1 \rightarrow a_2$ if and only if (1) $i = j$, and the generation of a_1 happened before the generation of a_2 in program order; or (2) $i \neq j$, a_1 is a *release*, a_2 is an *acquire* on the same lock as a_1, and a_1 releases the lock to a_2; or (3) there exists an access a_0, such that $a_1 \rightarrow a_0$ and $a_0 \rightarrow a_2$.

Definition 2 *Previous and concurrent actions*

Given any two actions a_1 and a_2, (1) a_1 is said to be *previous* to a_2 if and only if $a_1 \rightarrow a_2$; (2) a_1 and a_2 are said to be *concurrent* (written $a_1 \| a_2$) if and only if neither $a_1 \rightarrow a_2$ nor $a_2 \rightarrow a_1$.

Definition 3 *Conflicting accesses, data race and data-race-free programs*

Two memory accesses a_1 and a_2 *conflict* if and only if a_1 and a_2 access the same memory location and at least one is a write. If there are two conflicting accesses a_1 and a_2 in an execution of a program, and $a_1 \| a_2$, then we say a *data race* occurs in the execution. A program is *data race free* if and only if every possible execution of the program has no data race.

From the above definitions we know that if no data race occurs, then when a processor modifies a data object inside a critical region, other processors will not access that data object. Thus it is not necessary to propagate modifications during the execution of a critical region. The RSC models have taken advantage of this relaxation of modification propagation to achieve time, processor, and data selection. They can guarantee sequential consistency for data-race-free programs while improving their performance. In the following we briefly describe the RSC models in terms of the three selection techniques.

The Weak Consistency (WC) model [10] is an RSC model which achieves time selection. The WC model requires a processor to propagate all its modifications to other processors only at synchronization time, rather than at every memory modification time. With time selection, modifications on shared data objects can be accumulated and only the final results are propagated in batches at synchronization time. In this way, the number of messages in the WC systems can be greatly reduced compared to that in strict SC systems.

The Eager Release Consistency (ERC) model [13] 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. The ERC model requires that shared memory modifications be propagated outward only at *release* time. In other words, the ERC model is more time selective than the WC model by propagating modifications outward 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 memory system.

The Lazy Release Consistency (LRC) model [18] improves the ERC model by performing both time and processor selection. Instead of propagating modifications to all other processors at *release* time as in ERC, LRC postpones the propagation of modifications until another processor has successfully performed an *acquire*. At a successful *acquire*, the DSM system is able to know precisely which processor is the next one to access the shared data, so modifications can be propagated only to that particular processor (or no propagation at all is required if the

next processor is the current processor), thus achieving processor selection in the LRC model. By sending modifications only to the processor that is entering a critical region, a greater reduction in the number of messages can be achieved in the LRC model.

The Entry Consistency (EC) model [5] is very similar to LRC in propagating modifications only to the next processor entering a critical region. In addition to time and processor selection, the EC model also performs data selection by propagating only those shared data objects that are associated with a synchronization data object such as a lock. These associations are annotated by the programmer. Due to the inclusion of data selection, the EC model can be more efficient than the LRC model [6].

The Scope Consistency (ScC) model [17] is able to offer most of the potential performance advantages of the EC model by means of time, processor, and data selection, and it also improves the programmability of the EC model by requiring programmers to attach consistency scopes with code sections, instead of data. For critical regions, data objects can be automatically associated with scopes; but the programmer has to annotate the scopes explicitly for non-critical regions. If the annotation is not provided or is incorrect, ScC does not guarantee sequential consistency for the program.

From the above discussion we can see that constraints on modification propagation and execution have become more and more relaxed. This relaxation allows DSM systems to perform time, processor, and data selection. To achieve these selections RSC models require programmers to annotate the programs manually so that selections can be combined with synchronization primitives. For example, the ERC model requires programs to be properly labelled with system-provided synchronization primitives, so that the DSM system is explicitly notified of a processor's entry to and exit from a critical region and can thereby select the exit time to propagate modifications. 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 requiring programmer annotation which imposes an extra burden on programmers and increases the complexity of parallel programming.

We distinguish two types of programmer annotations: synchronization annotations which are required to ensure both the correctness of parallel programs (to avoid data races) and the correctness of memory consistency; and annotations which are required only for the correctness of memory consistency. For the first type of annotations, such as *acquire* and *release* primitives in the ERC, LRC, EC and ScC models, the DSM system can take advantage of them to achieve time and/or processor selection without imposing an additional burden on programmers. However, for the second type of annotations, such as the association between synchronization objects and data objects for data selection in the EC model, and the additional consistency scopes in the ScC model, they are truly an extra burden on programmers and increase the complexity of parallel programming. They should be replaced by automatic associations via run-time detection and/or compile-time analysis [11].

In the following section we present the View-based Consistency model, which achieves time, processor, and data selection and removes the extra burden on programmers required by previous data selection techniques.

3 View-based Consistency

The View-based Consistency (VC) model [16] has been proposed to achieve data selection transparently without programmer annotation. Similar to other RSC models, it guarantees Sequential Consistency for properly labelled data-race-free programs.

A *view* is a set of ordinary data objects that a processor has the *right* to access in the shared memory at a particular point in time. 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*.

In general we say that the view of a processor is a set of data objects that the processor has the right to access. In data-race-free programs, we can say that the view comprises data objects that the processor *will access* in some following period of execution, since the processor should have got the access right to the data objects.

At any time point of an execution in a data-race-free program, 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 would occur. This is illustrated in Fig. 3.

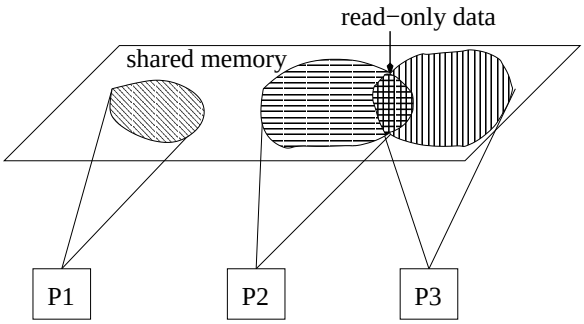

Figure 3: A snapshot of processors' views in a data-race-free program

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 by making use of 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 guaranteed by the programmer who controls access to the data object by associating it with the status of another critical-region-protected object. For example, exclusive access to a task can be guaranteed by removing the task from a lock-protected task queue (and thus the status of the task queue has been changed).

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

According to the above observation, views can be handled and processed based on regions (critical and non-critical). For convenience of description in this paper, we classify views into *Critical Region Views (CRVs)* and *Non-critical Region Views (NRVs)*. A CRV is the view of a processor while it executes a critical region, and an NRV is the view of a processor while it executes a non-critical region. In data-race-free programs, a CRV consists of the data objects accessed in the critical region, and an NRV consists of the data objects accessed in the non-critical regions.

Consistency maintenance in VC requires updating data objects of the view before a processor enters a region. More precisely, the following consistency conditions are given for the View-based Consistency (VC) model. Any implementation of VC should satisfy these conditions.

Definition 4 *Conditions for View-based Consistency*

- Before a processor P_i is allowed to enter a critical or 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 causal order.
- Before a processor P_i is allowed to pass a barrier primitive, all previous *write* accesses must *be performed with respect to* P_i according to their causal order.
- The sequential consistency of synchronization data objects must be 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.

Inside a barrier session, based on the first condition, a processor is guaranteed to access the up-to-date version of the data objects in its view.

The second condition states that when a processor reaches a barrier, it should be able to *see* all the previous modifications made by other processors. This condition is also required in EC, ScC and LRC.

SC correctness of VC Processors are synchronised to modify the same view one after another, but may modify different views concurrently in any data-race-free program. Based on this observation, for any parallel execution of a DSM program under VC, we can produce a global sequential order of the modifications on views, in which the modifications on the same view are ordered in the same way as the synchronised order of the parallel execution, and the modifications on different views are put in program order if they are executed sequentially in the program, otherwise they are parallel and put in any order. Parallel modifications on different views can be executed in any order, which will not affect the execution result. Obviously, according to the consistency conditions for VC, the parallel execution result of the program under VC is the same as the above sequential execution of the modifications. Therefore, a global

sequential order has been found to match the parallel execution result under VC. According to the definition of the SC model, VC can guarantee Sequential Consistency for data-race-free programs.

The first consistency condition of VC also indicates that, when a processor is entering a new region, it is only required to update those data objects in the corresponding new view. In this way, VC achieves time selection (at the entry of a region), processor selection (the next processor which has the right to access the view), and data selection (only the data objects of the view).

Any implementation of the VC model should conform with the above consistency conditions. There are two important technical issues in the implementation: view detection and view transition. View detection means identifying all the data objects in the new view of a processor before it enters a new region. In data-race-free programs, the view consists of the data objects to be accessed by the processor in the new region. Thus view detection is a prediction of the data objects that will be accessed by the processor during execution of the new region. View transition means updating all the data objects in the new view of a processor before it enters a new region. In short, before a processor enters a new region, its view should have been detected; when a processor enters a new region, the view transition should have been achieved. Any implementation of the VC model should guarantee that view detection and transition are implemented correctly.

Correctness and accuracy are two important issues in view detection. A correct view should include all data objects that a processor has the right to access. An accurate view should include and only include those data objects. The correctness of view detection must be satisfied in VC implementation, while inaccuracy of view detection may only affect its performance (e.g., propagation of irrelevant modifications of data objects). Of course, the performance also depends on the effectiveness of view detection and view transition. We will discuss techniques for view detection and transition in Section 4.

In the following discussion, we assume the view of every processor is correctly and accurately detected under an ideal situation, in order to explain how the generic VC model works.

Fig. 4 shows how the VC model works using the same scenario shown in Fig. 1. In Fig. 4, we assume the accesses to data object $x1$ are synchronised by the status of x, as in the third way for acquiring exclusive access described early in this section. Under this assumption it is

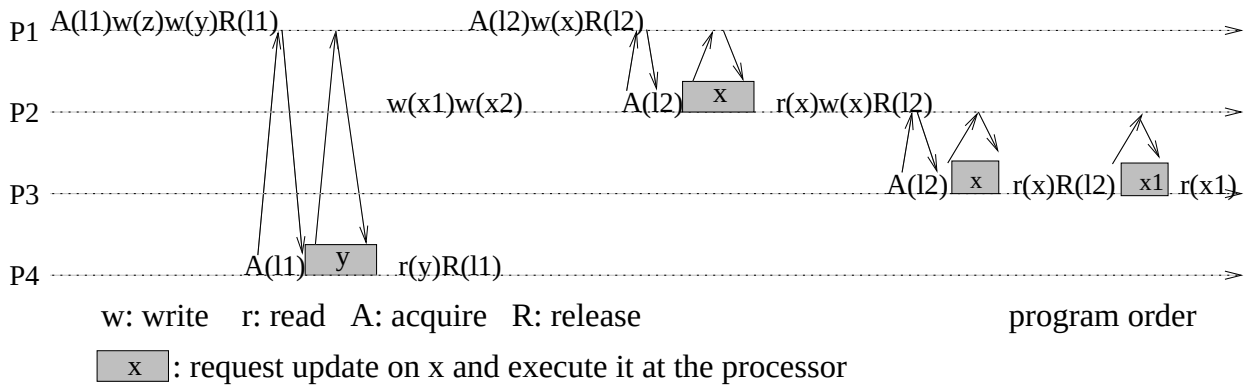

Figure 4: View-based Consistency in action

not possible for $P2$ and $P3$ to race for $x1$.

We assume there are two critical regions which are protected by locks $l1$ and $l2$, respectively, and P_2 and P_3 access data objects in non-critical regions. We also assume that the view of every processor has been detected correctly at each step. The detected view of P_4 includes y when it enters the critical region, the detected view of P_2 and P_3 includes x when they enter their critical regions, and the detected view of P_3 in the non-critical region includes $x1$. Before P_4 executes the critical region, only the modification of y is propagated to the processor to update its view. For P_2 and P_3, only the modification on x is propagated to them to update their views in the same critical region. When P_3 enters the non-critical region, only the modification on $x1$ is propagated. The lock acquisition and modification propagation are separate in the figure, but they can be combined in an implementation in order to improve DSM performance.

Comparison with related models Among the RSC models, only ScC [17] and EC [5] can achieve data selection. However, they cannot guarantee SC correctness for data-race-free programs (see Section 5 for details). In order to guarantee SC correctness, extra programmer annotations are required in programs. To selectively update data objects, EC uses *guarded shared data* D_S and ScC uses *scope*. Guarded shared data D_S is a set of data objects associated with a synchronisation data object s. This association is specified by the programmer. A scope in ScC consists of some program sections protected by a synchronization data object such as a lock. Both D_S and *scope* are static and fixed for a particular synchronization data object or a critical region. Even if some data objects are not going to be accessed by a processor in a critical region, they are updated simply because they are associated with the lock or the critical region. For example, in Fig. 4, suppose y and z are associated with lock $l1$ in EC. Thus after P_4 acquires $l1$, modifications on y and z are propagated to it, even though it is not going to

access z. This circumstance is not uncommon in parallel programs. In contrast, a view in VC is dynamic and may be different every time the same critical region is entered. For example, in Fig. 4, although P_1 updates both z and y in the $l1$-related critical region, P_4's view only includes y when P_4 enters the $l1$-related critical region. Therefore only the modification on y is propagated to P_4 under VC.

Compared to LRC, VC can reduce more of the false-sharing[2] effect in page-based DSM systems (see Section 5 for details). The false-sharing effect is the propagation of useless modifications, which is caused by false sharing.

Programming interface The VC model provides the same programming interface as LRC, ERC and WC. It can guarantee Sequential Consistency for data-race-free programs that are properly labelled. In contrast, 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 annotations are not correctly provided by the programmer (see Section 5 for details).

From the above discussion, we know that, in the generic VC model (an ideal implementation of VC) where views can be accurately detected, no useless modifications will be propagated among processors. Therefore, the generic VC model can achieve the maximum data selection. In other words, the generic model can achieve the maximum relaxation of constraints on modification propagation and execution in data-race-free programs. As a consequence, we believe the generic VC model can be used as a framework for future research on DSM implementations.

To summarise, we make the following statements which are true in the generic VC model.

- The view of a processor is constant within a region.
- When a processor enters a new region, only the data objects of its new view are updated.
- Sequential Consistency for data-race-free programs is guaranteed.

[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.

- Time, processor, and data selection are achieved. Data selection is achieved without programmer annotations.
- Maximum relaxation of constraints on modification propagation and execution for data-race-free programs is achieved.

In the following section, we will discuss our implementation of the VC model.

4 Implementation

We have implemented the VC model based on TreadMarks [3], which is a page-based DSM system. In TreadMarks, a *diff* is used to represent modifications on a page. Initially a page is write-protected. When a write-protected page is first modified by a processor, a *twin* of the page is created and stored in the system space. When the modifications 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.

In our implementation, a page is treated as the basic unit of data objects. Thus a view in the implementation consists of pages. The diffs can be regarded as modifications on pages.

4.1 View detection

In consistency maintenance we only need to know which are the modified data objects and then update them. Likewise, to maintain the consistency of a view, we only need to update the modified data objects in the view. Therefore, we are not interested in the unchanged pages of a view and thus only the modified pages are recorded in our implementation of view detection.

In our implementation, view detection is achieved at run time. To detect modified pages in a view, our implementation takes advantage of the following two existing mechanisms in TreadMarks:

1. When a write access is performed on an invalidated page, a page fault will occur. A page fault handler will request the diffs of the page from other processors. We extended the page fault handler to store the page's identifier into the corresponding view.

2. When a write access is performed on a write-protected page, a protection violation interrupt will occur. An interrupt handler will make a twin of the page under the multiple-writer scheme [8]. We extended the interrupt handler to store the page's identifier into the corresponding view.

Since the above two mechanisms already existed in TreadMarks, there was little overhead for storing the identifiers of modified pages.

4.1.1 View detection in critical regions

Views in critical regions are relatively easy to detect. All the pages previously modified in the same critical region are included in the corresponding CRV, because they will be very likely accessed later by a processor executing the same critical region, although program logic like *if* statements may affect the access pattern of a processor. However, the pages merely modified from outside a critical region are excluded from the CRV, since their diffs shouldn't be used to update the CRV, as this would indicate there is a data race in the program.

We construct a CRV by recording pages modified inside a critical region. However, if a page is already writable before a new region is entered, that page will not be detected and stored in the CRV or NRV if it is modified in the region. To detect all modified pages in a region, we make all writable pages write-protected (read-only) when entering the region. This is the additional overhead required for view detection. It is proportional to the number of times critical regions are executed in the program. Fortunately, the overhead will be balanced by the performance gain which is also proportional to the number of times critical regions are executed. Our experimental results demonstrate this additional overhead is relatively small.

Based on the above write-protection mechanism, our view detection algorithm works as below. A page set S_V is associated with each lock protecting a critical region. Initially S_V is empty. During an execution of the critical region, if a page p_i is detected as modified, then $S_V = S_V \cup \{p_i\}$. The page set S_V includes all the pages modified in the critical region. Since a processor entering the critical region will very likely access the pages in S_V, S_V is used as the detected CRV, and is reasonably accurate.

4.1.2 View shrinkage in non-critical regions

Unfortunately, views in non-critical regions are not as easy to detect as in critical regions, because there are no obvious connections between non-critical regions. The modified pages in previous non-critical regions may not belong to the view of a successive non-critical region. However, as for a CRV, it is certain that an NRV should only include pages modified in previous non-critical regions, since modifications made in critical regions shouldn't be used to update an NRV, otherwise, it indicates that a data race occurs in the program. Of course, a modified page may be included in both a CRV and an NRV or in two CRVs because of the false-sharing problem.

In our implementation, an NRV initially consists of all pages previously modified in non-critical regions and thus a detected NRV may be larger than the real one. This means that when a processor executes a non-critical region it may not access some pages in the NRV. This inaccuracy only affects the performance (less data selection and no reduction of false-sharing effect in non-critical regions), not the correctness of the implementation.

To improve the performance of our implementation, we use Regional Locality [15] to dynamically shrink the NRV during execution of a non-critical region. Regional Locality is based on the observation that a set of pages that are accessed in a non-critical region will be very likely accessed as a whole in other non-critical regions. For example, suppose processor P_1 enters a non-critical region and accesses pages m_1, m_2, ..., m_n during 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, as regulated by the programmer to shift workload among processors, 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 that set.

To exploit Regional Locality in non-critical regions, we detect the pages modified in non-critical regions and aggregate them. We adopt a Modified Pages Set (MPS) for grouping pages modified in non-critical regions. An MPS is formed as below. When a processor enters a non-critical region, a unique empty MPS is created for the non-critical region. If the processor modifies a page during execution of the non-critical region, the identifier of the page is stored into the MPS. When the processor exits from a non-critical region, the MPS becomes complete and is stored for later use.

We use some hints to decide whether we should shrink the view of a processor in a non-critical region. The first hint we use is the access to any page in an MPS (called the *first hit* of the MPS). This hint suggests that the full set of the pages in the MPS might be the real view of the processor executing the non-critical region. If the accessed page belongs to multiple MPSs, we use the access to another page (called the *second hit* of the MPS) to confirm which MPS might be the real view. Once an MPS is assumed to be the real view of the processor, the processor reduces its view to the MPS. The shrunken view can be used to prefetch diffs of the view in the non-critical region. We will discuss how to take advantage of the shrunken view in view transition.

The detected views are propagated to a processor when it calls *acquire*. Thus there are no extra messages for view propagation, except for small data of views piggy-backed on the lock release message.

4.2 View transition

View transition is to make a view (a set of pages) up to date. Before a new region is entered, view transition must be done. Modification propagation and execution protocols (also called consistency protocols) are used to achieve view transition, where pages that are required to be made consistent by the consistency model are made up to date. The two common modification propagation and execution protocols are the invalidation and the update protocol. If the invalidation protocol is adopted, when a page is required to be made consistent (up to date) by the consistency model, it is invalidated first and the diffs are propagated and applied by a page fault later. If the update protocol is adopted, when a page is required to be made consistent, the diffs are propagated and applied immediately.

From an implementation point of view, if a view is accurately detected and its pages are to be accessed by the processor, it would be more efficient to use the update protocol to propagate and apply the diffs, rather than using the invalidation protocol. However, if the detected view is larger than the real view, the invalidation protocol may be more efficient. Since the detected view has pages that will not be accessed by the processor, the update protocol will thus propagate useless diffs. In contrast, the invalidation protocol propagates (small) invalidation notices first to invalidate the pages in the detected view, but the diffs of a page are propagated only when the page is in the real view and thus accessed by the processor. In this way, the prop-

agations of useless diffs are avoided in the invalidation protocol, with the overhead of more messages (and page faults). If the useless diffs are huge, there is a chance for the invalidation protocol to be more efficient.

The update protocol is generally suitable for VC, just as the invalidation protocol is for LRC (see Section 5 for details). However, since the detected NRVs in our implementation are not accurate and are normally larger than the real ones in our implementation, we adopted the invalidation protocol for the view transition into non-critical regions. The invalidation notices for the NRV of a processor are propagated to the processor when it calls *acquire*, but are disabled in critical regions. They are enabled and effective in any non-critical regions. When a processor enters a non-critical region, the pages with the disabled invalidation notices are invalidated. However, these notices will be disabled again in the following critical region if their pages are not accessed in the non-critical region nor belong to the following CRV. Obviously, compared with TreadMarks, enabling/disabling invalidation notices is another overhead in our implementation. Again this overhead is proportional to the number of times critical regions are executed.

During execution of the non-critical region, the NRV may shrink to one of the MPSs. Once the NRV shrinks, we prefetch the diffs of the pages in the shrunken NRV and apply them to their pages. This optimisation doesn't change the effectiveness of the invalidation notices of other pages.

5 Comparison with related work

Our implementation can improve performance at two levels: the consistency model and consistency protocol levels. At the consistency model level, it only updates pages in the view of a processor entering a new region. The detected CRVs are generally accurate and the update protocol is used to update them. The accurate CRVs detected in our implementation help reduce the false-sharing effect, as will be discussed shortly. In LRC all previously modified pages must be updated and the invalidation protocol is used to alleviate the overhead of propagation of useless diffs.

At the consistency protocol level, besides the update protocol adopted for CRVs, our implementation uses shrunken views to prefetch the diffs in non-critical regions. This technique is

similar to some optimal consistency protocols [23, 14, 4], which have been proposed to improve the performance of the LRC model. These protocols choose optimal ways to propagate diffs, e.g., prefetching of diffs. However, they work at the level of modification propagation in LRC, instead of at the level of a consistency model. They don't change the consistency conditions of LRC. Therefore, they inherit the false-sharing effect from LRC. Though they can reduce the number of messages and the overhead of page faults by prefetching, they can't remove the false-sharing effect in LRC, i.e., the propagation of useless diffs. These techniques are complementary to our VC implementation and may be useful to further optimize the diff propagation in non-critical regions in our implementation.

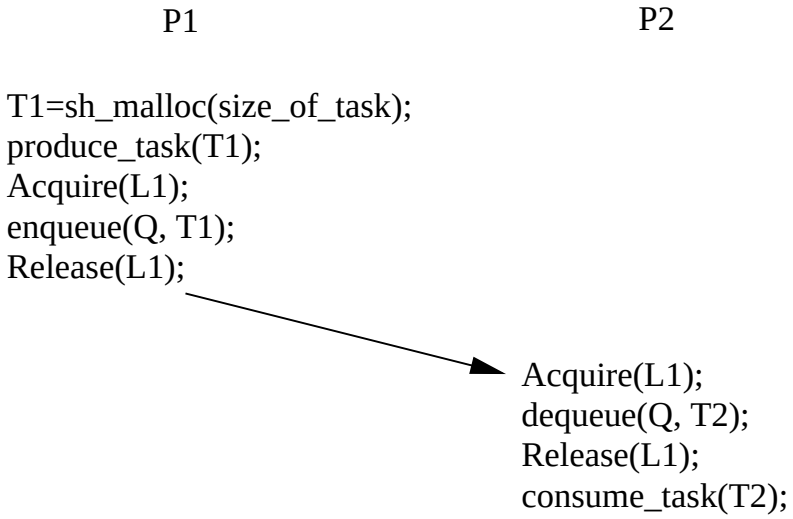

Figure 5: A task queue program

To explain the subtle differences among related models, we will discuss a task queue program shown in Fig. 5. To make the comparison fair, in the discussion we assume the invalidation protocol is used with the models being compared. This assumption is valid, since RSC models address when to make which pages consistent in page-based DSM systems and thus are independent of modification propagation and execution protocols (also called consistency protocols) such as the invalidation or update protocol, although those protocols may affect the performance of their implementations.

In the program in Fig. 5, the variable Q is used as a task queue. Processor $P1$ produces a task $T1$ and puts it into Q, and then processor $P2$ gets a task $T2$ from Q and consumes the task, where $T1$ and $T2$ are local variables that store the addresses of the shared data objects for tasks.

Fig. 6 illustrates the read/write accesses to the shared data objects in the task queue program.

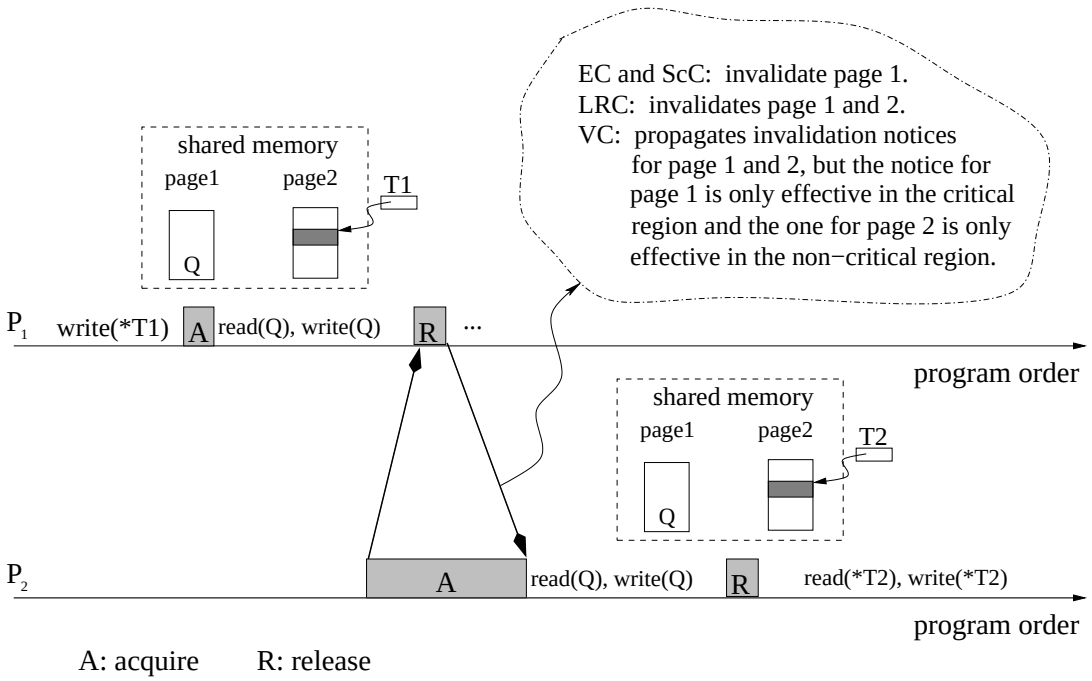

Figure 6: Differences among EC, ScC, LRC, and VC

It is used to explain how different RSC models maintain consistency in a page-based DSM. To make the subtle differences more clear, we assume $T1$ and $T2$ happen to point to the same task data which is located in page 2, while Q is in page 1.

In EC, the programmer is required to annotate that Q is associated with lock $L1$, while in ScC this association can be detected at run-time. In both EC and ScC, only Q is required to be made up to date before $P2$ enters the critical region. Therefore, the invalidation notice for page 1 is propagated to $P2$ and page 1 is made invalid there. A later page fault on page 1 will bring it up to date. Since page 2 is not invalidated, $P2$ will not read the up-to-date task data pointed to by $T2$. Thus SC cannot be guaranteed for this data-race-free program under EC and ScC.

In LRC, all previous write accesses, including the writes on Q and the task data pointed by $T1$, must be performed before the acquire is performed in $P2$. Thus the invalidation notices for page 1 and 2 are propagated and page 1 and 2 are made invalid. Later page faults will bring them up-to-date.

In VC, all previous write accesses to the data objects in a view must be performed before a processor enters the corresponding region. Run-time view detection in VC can determine that page 1 (Q) belongs to the CRV and that page 2 (the task data pointed to by $T1$) belongs to the NRV. Thus the invalidation notices for page 1 and 2 are propagated to $P2$, however, the notice for page 1 is only effective in the critical region to bring Q up to date, and the one for

page 2 is only effective in the non-critical region to bring the task data up to date. Mindful readers may realise that the more complicated processing of the invalidation notices in VC is not superior to LRC in this example, given the extra overhead of enabling and disabling invalidation notices. However, the individual treatments of the notices for different regions can result in less false-sharing effect in many other situations, as will be described shortly.

From the above descriptions we know the RSC models are different in terms of when to make which pages up to date, i.e., their consistency conditions are different. However, there is another important aspect of difference among them regarding data selection. Through user annotation in EC, run-time detection of scope-data association in ScC, or run-time view detection in VC, these models have firm knowledge that page 1 (Q) will be very likely accessed in the critical region, which can be readily used to improve their implementations. For example, instead of adopting the invalidation protocol which causes one page fault and one modification (diff) request in the above example, they can adopt the update protocol which can piggy-back the diffs on the lock granting message, i.e., lock release (though extra messages may be required if diffs are huge) and thus save two messages (diff request and reply) and one page fault.

On the other hand, LRC has no such knowledge and thus doesn't know which diffs will be useful. Therefore, the invalidation protocol can help avoid propagation of useless diffs in such a situation. Of course, some extra work can be done to improve the diff propagation in LRC. Examples are the Affinity Entry Consistency (AEC) protocol [23] which can pre-send the modifications based on Lock Acquirer Prediction (LAP), and the Heuristic Diff Acquiring (HDA) protocol [14] which adopts the update protocol for pages modified in critical regions and piggy-backs the diffs on the lock granting message. However, these improvements don't change the consistency conditions in LRC such as which pages need to be made up to date at *acquire* time. The following example will illustrate the difference between LRC and VC in terms of reducing the false-sharing effect and the difference between VC and optimal diff propagation (or consistency) protocols such as AEC.

Fig. 7 presents an example of the false sharing effect in LRC. In the example, data objects X and Y lie in the same page. Processor P_1 modifies X in the first critical region $CR1$ and then modifies Z in the second critical region $CR2$. After P_1 exits from $CR2$, P_2 enters it. Although P_2 only accesses data objects Y and Z, according to LRC, both page 1 and page 2 of P_2 should

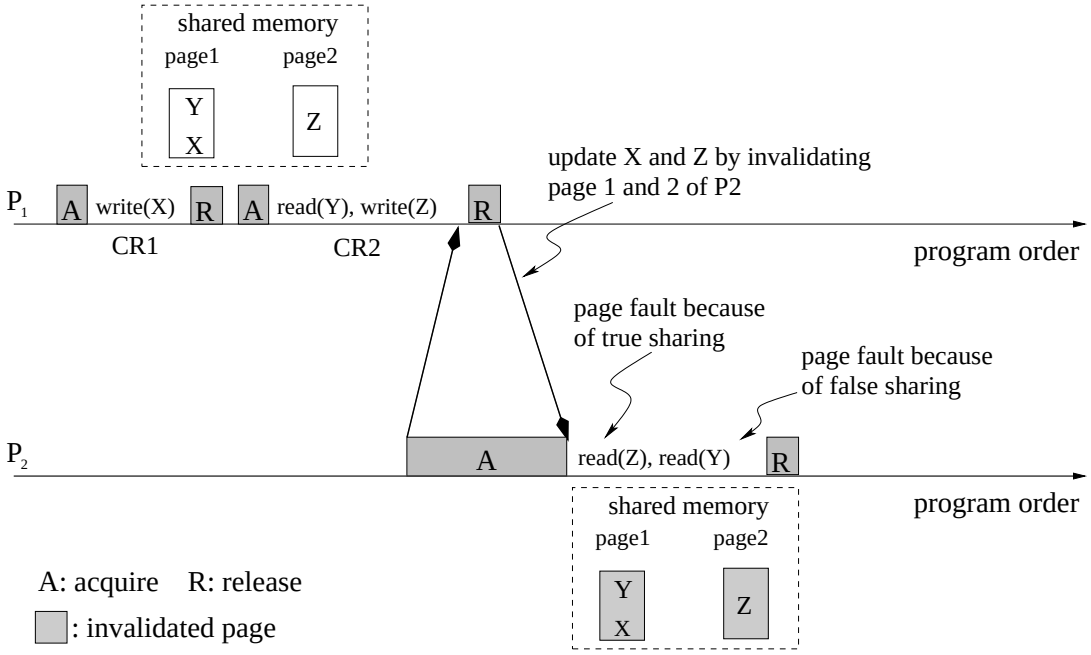

Figure 7: False sharing effect in LRC

be invalidated. When P_2 reads Y, there is a page fault on page 1, which is caused by false sharing.

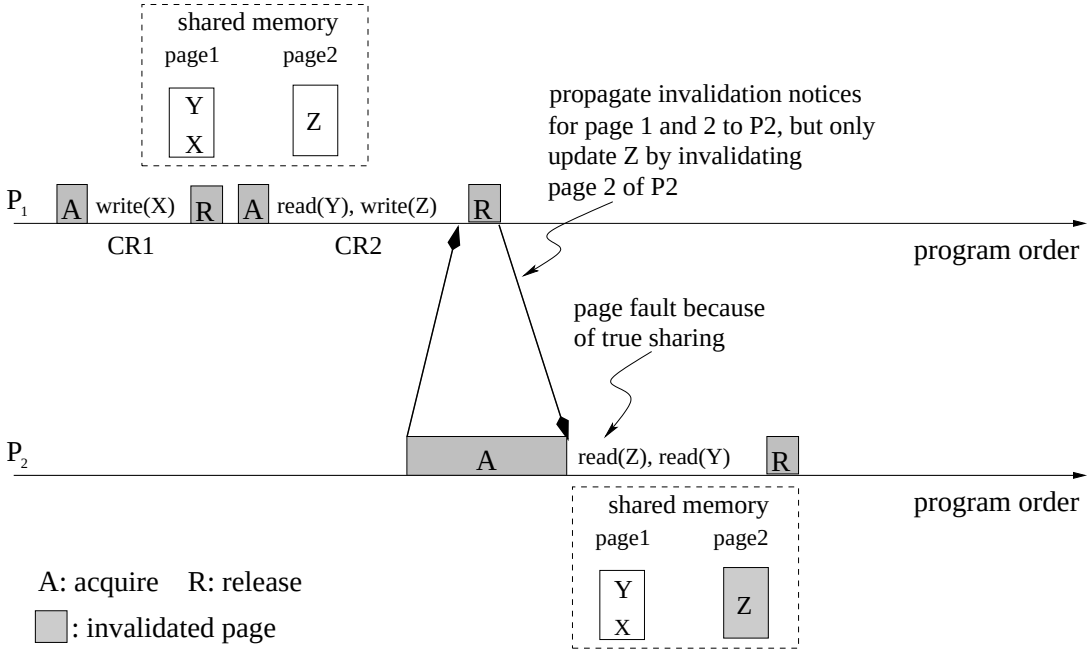

Figure 8: Reduction of the false sharing effect in VC

For the same example, Fig. 8 shows how VC removes that effect of false sharing. VC only requires updating the data objects of the current view of a processor. The view of P_2 in $CR2$ only includes data objects Y and Z. Since Y is not modified by P_1, VC only updates Z by invalidating page 2, though the invalidation notice for page 1 is also propagated (but disabled)

and may be effective later to bring X up to date in P_2. By restricting the effective scope of the invalidation notice for page 1 (which is of no use for updating Y and Z in the view of P_2), VC can avoid the page fault on page 1 caused by false sharing. Naturally, as we said before, the update protocol should be adopted to make the implementation of VC more effi cient with the knowledge of views.

As we mentioned before, LRC can be improved by adopting more complicated diff propagation protocols, such as AEC [23] and HDA [14]. AEC pre-sends diffs from critical regions to the processors that will likely enter the same critical regions based on Lock Acquirer Prediction (LAP). For the example in Fig. 7, AEC may be able to predict that P_2 will enter $CR2$ and thus the diff of page 2 can be piggy-backed on the lock release message and applied to page 2 before P_2 is entering $CR2$. In this way, the number of page faults and diff request/reply messages can be reduced. Similarly, HDA uses history records to pre-send likely-useful diffs to only the next processor entering the same critical region. Though the performance of LRC can be improved by the above optimal diff propagation protocols, they don't change the consistency conditions of LRC. This is the essential difference between consistency models and diff propagation (or consistency) protocols. Therefore, in Fig 7, the invalidation notice for page 1 is still required (by LRC) to be effective in AEC and HDA. Consequently, the page fault on page 1 will occur due to false sharing in AEC and HDA, which reflects their essential difference from VC.

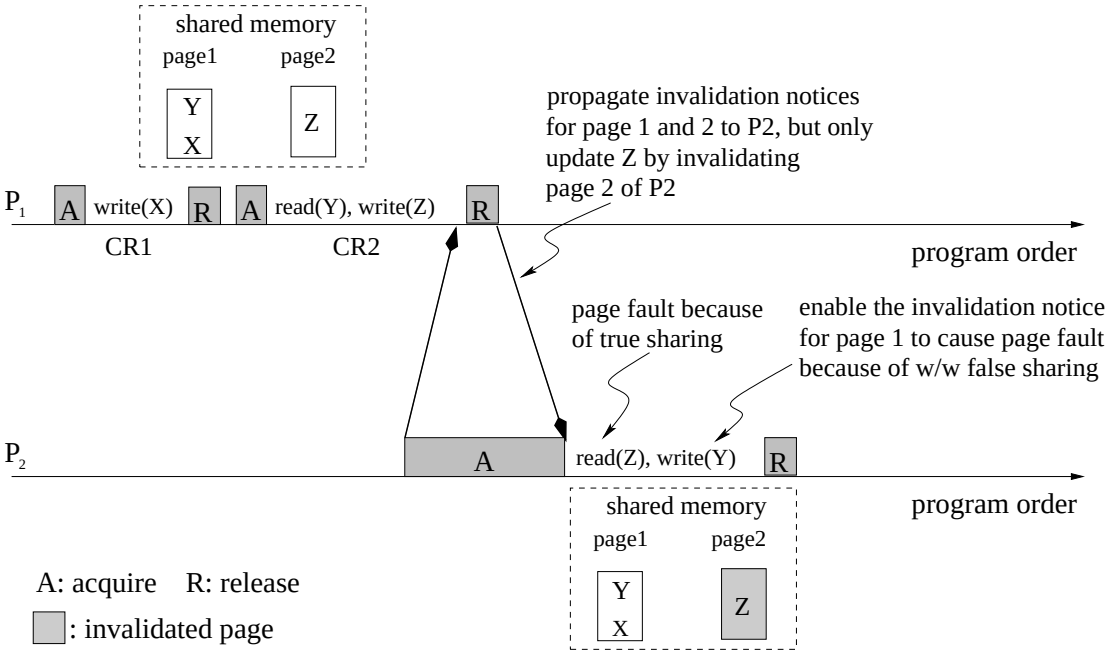

Figure 9: The w/w false sharing effect in the VC implementation

There are two kinds of false-sharing effect, write/read (w/r) and write/write (w/w). A w/r 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. For example, the false-sharing effect in Fig. 7 is w/r. A w/w 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. While our current implementation of VC can reduce w/r false-sharing effect as shown in Fig. 8, it can't reduce the w/w false-sharing effect since a coarse-grained diff management scheme is adopted. For example, in Fig. 9 (which is similar to the scenario in Figs. 7 and 8, except P_2 writes rather than reads Y), when P_2 writes Y on page 1, any implementation of VC will face two choices. One is to ignore the disabled invalidation notice for page 1, make a twin of page 1 and make a diff later. This scheme has to distinguish the diffs from different views and a complicated, fine-grained diff management will be inevitable. The benefit of the scheme is the reduction of the w/w false-sharing effect.

The alternative choice is to enable the invalidation notice for page 1, bring it up to date through the page fault before making its twin, as shown in Fig. 9. While the scheme simplifies diff management, it tolerates the w/w false-sharing effect. We adopted the latter scheme in our current implementation, but will investigate fine-grained diff management in the future.

SC correctness of our VC implementation Our implementation can normally guarantee Sequential Consistency for data-race-free programs. It guarantees that pages that are previously modified in a critical region are updated when a processor is entering the same critical region. That means, pages that are not modified in the critical region are not put into the CRV in the view detection. For most data-race-free programs, it is true that data objects required to be updated and then accessed in a critical region are only those previously modified in the same critical region. Our VC implementation can guarantee Sequential Consistency for those data-race-free programs. However, there may be cases where pages modified in non-critical regions are accessed in a critical region in a data-race-free program, in which cases our implementation can't guarantee Sequential Consistency. For example, if we modify the program in Fig. 5, so that when $P2$ dequeues a task it also checks the task data in the critical region. In that case, the processor won't read the up-to-date task data in the critical region (but will read the up-to-date task data in the following non-critical regions) in our implementation, though there is no data

race in the program. But we claim this kind of program is very peculiar, since the critical region is designed to protect only the task queue, not the task data. Fortunately, we haven't found this kind of peculiar programs in our applications yet.

If the above peculiar type of program needs to be supported, we can enhance our implementation with compile-time analysis. Once the above access pattern is found at compile time, the compiler can direct the DSM to enable the invalidation notices of all modified pages, which will guarantee Sequential Consistency for those programs.

It is worth noting that the above problem with SC correctness of our implementation doesn't affect the SC correctness of the generic VC model.

6 Experimental evaluation

In this section, we present an experimental evaluation of the LRC model and our implementation of VC. Both of these have been implemented based on TreadMarks [3]. The experimental platform consists of 8 PCs running Red Hat Linux 6.1, which are connected by a 10 Mbps Ethernet. Each of the PCs has a 500 MHz processor and 128 Mbytes of memory. The page size in the virtual memory is 4 KB.

We chose five applications in the experiment: *TSP*, *QS*, *BT*, *Water* and *IS*. *TSP*, *QS*, *Water*, and *IS* are provided by the TreadMarks research group. All the programs are written in the C language. *TSP* is the Travelling Salesperson Problem, which finds the minimum cost path that starts at a designated city, passes through every other city exactly once, and returns to the original city. *QS* is a task-queue style program. It uses a recursive sorting algorithm that operates by repeatedly partitioning an unsorted input list into a pair of unsorted sublists, such that all of the elements in one of the sublists are strictly greater than the elements of the other, and then putting the sublists into a task queue. It recursively takes a sublist from the queue and sorts the sublist, until the task queue is empty. *BT* is an algorithm that creates a fixed-depth binary tree. In the algorithm, multiple processes explore a binary tree to search for unexpanded nodes. If a process finds an unexpanded node, it expands the node and creates new unexpanded nodes. The algorithm terminates when the fixed-depth binary tree is established. *Water* is a molecular dynamics simulation. Each time step, the intra- and inter-molecular forces incident on a molecule are computed. *IS* (Integer Sort) ranks an unsorted sequence of N keys. The

rank of a key in a sequence is the index value i that the key would have if the sequence of keys were sorted. All the keys are integers in the range [0, B_{max}] and the method used is bucket sort. These applications are representative of both numerical computing (*Water, QS* and *IS*), and symbolic computing (*TSP* and *BT*). The performance results are listed in Table 1.

App	Seq. Time (Sec.)	Model	Time (Sec.)	Diff_Req	RPF	RFS	Mesgs	TFS
TSP	1.89	LRC	2.54	962	-	-	2763	58
		VC_i	2.56	960	-	0	2756	
		VC	1.65	25	937	0	911	
QS	0.29	LRC	7.09	3267	-	-	12209	2
		VC_i	7.15	3330	-	0	12375	
		VC	4.59	791	1044	0	5301	
BT	0.16	LRC	28.26	11437	-	-	79468	4347
		VC_i	27.59	11347	-	792	79426	
		VC	25.73	7429	3441	776	69342	
Water	10.25	LRC	19.86	12428	-	-	96600	6
		VC_i	19.91	12423	-	3	96600	
		VC	19.09	11891	511	3	95478	
IS	-	LRC	113.42	4444	-	-	11305	-
		VC	108.20	2774	1569	0	7965	

Table 1: Performance statistics for applications on eight processors

Other applications, such as *FFT*, *SOR* and *Barnes*, were not chosen since they don't have locks (critical regions) and thus they have no performance gain in our implementation. Since the extra overhead of view maintenance is proportional to the number of times critical regions are executed, our implementation is no worse for these problems than LRC. Though *IS* also has no locks, it was chosen to demonstrate the performance gain of adopting view shrinkage in non-critical regions. Its experimental results have been cited from previous work [15]. Though view shrinkage is not the major contribution of this paper, we should mention that, from our earlier results [15], it doesn't work very well for *FFT* and *SOR*, as the detected MPSs include pages that will not be accessed in the following barrier sessions (see Huang et al. [15] for

details).

We designed the experiment in order to demonstrate the performance gain of VC at both consistency model and consistency protocol levels. We also collected evidence of false-sharing effect in our applications in order to demonstrate the potential performance gain of VC. The extra overhead of view maintenance is also indicated from our results.

Table 1 shows the performance results for LRC and our VC implementation. VC_i is the VC implementation based on the invalidation protocol, which is used to investigate the performance gain of VC at the consistency model level and to indicate the extra overhead of view maintenance. *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 overall improvement in our VC implementation, *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. The second column *Seq. Time* is the running time of the sequential execution of the applications. Except for *TSP*, the sequential execution of the applications is faster than their parallel execution on 8 PCs, which demonstrates the need for improving the DSM performance on cluster computers.

VC_i vs. LRC

Table 1 shows that some applications, such as *TSP* and *QS*, don't benefit from the implementation of VC_i, because they have no w/r false-sharing effect in critical regions. As discussed in Section 5, our implementation can only reduce w/r false-sharing effect.

In addition, in the task-queue style program *QS*, the inaccurate NRVs can't help our implementation to reduce the false-sharing effect, though *QS* has w/r false-sharing effect in non-critical regions.

For these two applications, 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 indicates that the overhead of view maintenance (including view detection and view transition) is only a trivial portion of the expense of the whole system.

However, for *BT* which has high w/r and w/w false-sharing effects, VC_i can improve its performance up to 2.4%. Though this performance gain is not very significant, it can only be achieved at the level of the consistency model and can't be achieved by the optimal diff propagation protocols [23, 14, 4] such as AEC. This performance gain will be increased if the

w/w false-sharing effect can be reduced or the detected NRVs can be more accurate.

For *Water*, VC_i is slightly (0.25%) worse than LRC. Though VC_i has reduced the false-sharing effect (reduction of three page faults) in *Water*, the overhead of view maintenance overshadowed the benefit.

VC vs. LRC

In Table 1, VC is our implementation with the optimal diff propagation protocol as described in Section 4.2, where the update protocol is adopted for updating CRVs. This optimisation is very natural in our implementation, since the CRVs are readily available and predict the pages to be very likely accessed.

Table 1 shows VC outperforms LRC for all five applications tested. VC has improved the performance significantly compared with LRC (35% for *TSP*, 35.3% for *QS*, 9% for *BT*, 3.9% for *Water*, and 4.6% for *IS*). The number of *diff* request messages in VC is significantly less than that in LRC (97.4% less in *TSP*, 75.8% less in *QS*, 35% less in *BT*, 4.3% less in *Water*, and 37.6% less in *IS*). The detected CRVs have helped the update protocol to reduce the *diff* request messages. Consequently the total number of messages in VC has been greatly reduced compared with LRC.

The major reason for the above improvement is that the diffs of the pages to be accessed in critical regions (i.e., CRVs) are piggy-backed on the lock release messages and applied to the pages immediately. Thus the number of diff request/reply messages and the overhead of page faults are reduced in those applications.

It is worth noting that the above performance gain due to the use of the update protocol for CRVs may be achieved by the optimal diff propagation protocols [23, 14, 4].

False-sharing effect

There are two applications, *BT* and *Water*, that demonstrate the reduction of the false-sharing effect achieved in our implementation. *BT*, which uses locks to protect nodes in the binary tree, has serious false-sharing effect, while *Water* has less false-sharing effect. For *BT*, the performance improvement due to reduction of the false-sharing effect is 18.4% of the total improvement. This significant proportion has demonstrated the advantage of the VC model

over the consistency protocols, e.g. AEC, in terms of the programs with serious false-sharing effect.

To get an impression of the false-sharing effect in our applications, we have recorded the total number of page faults that are due to the false-sharing effect inside critical regions. In Table 1, *RFS* is also the number of page faults that are due to the w/r false-sharing effect inside the critical regions; *TFS* is the number of page faults that are due to all false-sharing effects (including w/r and w/w false sharing) inside the critical regions. The results show the reduced w/r false-sharing effect is only a small portion of the total false-sharing effect for half of our applications (0% for *TSP*, 0% for *QS*, 18.2% for *BT*, and 50% for *Water*) inside the critical regions. For the applications with serious false-sharing effect, such as *BT*, there is still a great room for performance improvement which can only be achieved at the level of the consistency model in VC.

We were not able to collect evidence for the false-sharing effect in non-critical regions, but we believe applications such as *QS* and *Water* have signifi cant false-sharing effect in non-critical regions.

7 Conclusions

In this paper, we have presented a new RSC model, the VC model, for DSM systems. RSC models have relaxed the constraints on modifi cation (diff) propagation and execution for data-race-free programs that are properly labelled while still guaranteeing sequential consistency for those programs. We have briefly discussed each of them in terms of three selection techniques. Among previous RSC models, only EC and ScC can perform data selection, but 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. It can achieve data selection without programmer annotations and reduces more false sharing effect than LRC. Its only consistency requirement is that all the data objects in the view of a processor must be updated during view transition. In this way, it can achieve the maximum relaxation on the consistency requirements and produces more room for optimisation in DSM implementations. Therefore, the generic VC model can be used as a framework for future research on DSM implementations.

We have implemented the VC model based on TreadMarks. Techniques for the implementation of VC have been discussed in this paper. The differences between our VC implementation and related work were illustrated through examples. Performance results have shown that our VC implementation generally outperforms the LRC model and has advantages over the optimal diff propagation protocols [23, 14, 4]. The results have also demonstrated that the extra overhead of view maintenance in our VC implementation is relatively trivial.

Further research should be carried out under the framework of the VC model. First, fine-grained diff management needs to be investigated. The current implementation tolerates the w/w false sharing effect due to the adoption of simple diff management. More elaborate diff management may help to reduce more of the false-sharing effect. Second, mechanisms for detection of accurate NRVs need to be investigated. Finally, Run-time and compile-time techniques [11] need to be developed for accurate view detection.

References

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

[2] S.V. Adve and K. Gharachorloo: "Shared memory consistency models: a tutorial", *IEEE Computer,* 29(12), pp.66-76, December 1996.

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

[4] C. Amza, et al.: "Adaptive Protocols for Software Distributed Shared Memory", *Proc. of IEEE,* Special Issue on Distributed Shared Memory, 87(3), pp.467-475, March 1999.

[5] 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.

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

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

[8] 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.

[9] P. Dasgupta, et al.: "The design and implementation of the Clouds distributed operating system", *Computing Systems Journal*, 3, Winter 1990.

[10] 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.

[11] 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.

[12] 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.

[13] K. Gharachorloo, D. Lenoski, and 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.

[14] 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.

[15] 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.

[16] Z. Huang, C. Sun, M. Purvis, and S. Cranefi eld: “View-based Consistency and its Implementation”, *In Proc. of the First IEEE/ACM International Symposium on Cluster Computing and the GRID*, Brisbane, May 2001.

[17] 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.

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

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

[20] D. Lenoski, et al.: “The Stanford DASH multiprocessor”, *IEEE Computer*, 25(3), pp.63-79, March 1992.

[21] K. Li, and P. Hudak: “Memory Coherence in Shared Virtual Memory Systems”, *ACM Trans. on Computer Systems*, Vol.7, pp.321-359, November 1989.

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

[23] C.B. Seidel, R. Bianchini, and C.L. Amorim: “The Affi nity Entry Consistency Protocol”, *In Proc. of the 1997 International Conference on Parallel Processing*, August 1997.

[24] 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.

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