View-based Consistency and its Implementation

Z. Huang‡, C. Sun‡, M. Purvis‡, and S. Cranefield†
†Departments of Computer & Information Science
University of Otago
Dunedin, New Zealand
‡School of Computing & Information Technology
Griffith University
Brisbane, Qld 4111, Australia
Email: hzy@cs.otago.ac.nz, scz@cit.gu.edu.au,
{mpurvis,scranefield}@infoscience.otago.ac.nz

Abstract

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

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

1 Introduction

Distributed Shared Memory (DSM) has become an active area of research in parallel and distributed computing [15, 8, 2, 5, 4, 1, 17, 18]. A DSM system provides application programmers the illusion of shared memory on top of message passing distributed systems, which facilitates the task of parallel programming in distributed systems. The goal of our research is to make DSM systems more convenient to use and more efficient to implement [10, 17]. In this paper, we propose a View-based Consistency (VC) model for DSM, which is a significant step towards our goal.

The consistency model of a DSM system specifies the ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems' programming convenience and implementation efficiency [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 sequential order, and the operations of each individual processor appear in this sequence in the order specified by its own program [14]. This means that in a SC-based DSM system, memory accesses from all processors may be interleaved in any sequential order which is consistent with each processor’s memory access order, and the memory access orders observed by all processors are the same. One way to strictly implement the SC model is to ensure all memory updates be totally ordered and memory updates performed at one processor be immediately propagated to other processors. This implementation is correct but it suffers from serious performance problems [18].

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. For example, consider a DSM system with four processors \( P_1, P_2, P_3, \) and \( P_4 \), where \( P_1, P_2, \) and \( P_3 \) share a data object \( x \), and \( P_1 \) and \( P_4 \) share a data object \( y \). Suppose all memory accesses to the shared data objects \( x \) and \( y \) are serialized among competing processors by means of synchronization operations to avoid data races. Under these circumstances, the following three basic techniques can be used: (1) **Time selection:** Updates on a shared data object by one processor are made visible to the public only at the time when the data object is to be read by other processors. For example, updates on \( x \) by \( P_1 \) may be propagated outward only at the time when either \( P_2 \) or \( P_3 \) is about to read \( x \). (2) **Processor selection:** Updates on a shared data object are only propagated from one processor to the processor that is the next in sequence to access the shared data object. For example, updates on \( x \) by \( P_1 \) may be propagated to only \( P_3 \) (but not to \( P_2 \)) if \( P_2 \) is the next in sequence to read \( x \). (3) **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 weaker SC models have been proposed [7, 9, 13, 3, 12], which perform one or more of the above three selection techniques while appearing to be sequentially consistent. However, none of them can achieve data selection without programmer annotation [17]. We argued previously [17] that a consistency model should not impose any extra burden on programmers, such as annotation of lock-data association in the Entry Consistency (EC) [3] and scope-data association in the Scope Consistency (ScC) [12] models. In this paper, we propose a **View-based Consistency (VC)** model which, besides time selection and processor selection, can transparently achieve data selection.

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

## 2 View-based Consistency (VC)

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

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

A **view** is a set of **ordinary** data objects a processor has the **right** to access in shared memory. We say a processor

\(^1\)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.
has the right to access some data object if and only if it has gained exclusive access to the data object or the data object is read-only. At any time point of an execution, suppose any two processors \( P_1 \) and \( P_2 \) have views \( V_1 \) and \( V_2 \) respectively. Then \( V_1 \cap V_2 \) only contains read-only data objects; otherwise a data race happens. Fig. 1 shows a snapshot of views of processors in shared memory. The overlapped part of different views only contains read-only data objects.

![Diagram of processors and shared memory](image)

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

Many DSM systems require explicit acquire, release and barrier in DSM programs to achieve weak sequential consistency. An execution of such a DSM program can be viewed as a sequence of barrier sessions shown in Fig. 2. A barrier session begins with a barrier and ends with another barrier. Inside a barrier session there is a sequence of regions which are delimited by acquire, release and barrier primitives. A critical region begins with an acquire and ends with a release, while a non-critical region begins with a release (the outermost one in nested critical regions) or a barrier and ends with an acquire (the outermost one in nested critical regions) or a barrier. A non-critical region does not overlap with any critical region, but a critical region may overlap with another critical region due to the possibility of nested critical regions.

![Diagram of barrier session](image)

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

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

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

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

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

---

1. Critical Region Views (CRV)
2. Non-critical Region Views (NRV)
Figure 3: Views and their transitions

**Definition 1** Critical Region View (CRV)

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

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

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

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

**Definition 3** Conditions for View-based Consistency

- Before a processor $P_i$ is allowed to enter a critical region or a non-critical region, all previous write accesses to the ordinary data objects of the CRV or NRV must be performed with respect to $P_i$ according to their order.

- The sequential consistency of synchronization data objects is guaranteed by the implementation of the system primitives, such as `acquire`, `release`, and `barrier`.

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

The VC model has the following properties:

- In the VC model only when a processor moves from one region to another region, does its view change. A processor's view is constant within a region.

- In the VC model when a processor changes to a new region, all the data objects of its new view must be updated.
• The VC model guarantees the same execution result as the Sequential Consistency for a data-race-free DSM program. Read/write data objects are accessed sequentially by exclusive access and they are updated once the exclusive access to them is gained.

• The VC model can achieve time selection, processor selection, and data selection. Data selection can be achieved by updating only the data objects in the current view of a processor.

3 Comparison of related models

Among the different consistency models, only ScC [12] and EC [3] can achieve data selection. But the VC model is different from them in the following aspects.

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

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

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

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

1. Restrict the propagation of invalidation notices. Only the invalidation notices that are useful for updating the data objects in a processor's new view are propagated to the processor;

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

Fig. 4 presents an example of the false sharing effect in Lazy Release Consistency (LRC) [13]. In the example, data objects $x$ and $y$ lie in the same page. Processor $P_1$ updates $x$ in the first critical region and then updates $z$ in the second critical region. After $P_1$ exits from the second critical region, $P_2$ enters the same critical region. Though $P_2$ only accesses data objects $y$ and $z$, LRC updates data objects $x$ and $z$ in $P_2$ by invalidating both page 1 and page 2. When $P_2$ accesses $y$, it has a page fault on page 1, which is caused by false sharing.

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

Figure 6: False sharing effect in both EC and ScC
While ScC [12] and EC [3] can remove the above false sharing effect, they may fail to do so in other situations. Fig. 6 presents another example of the false sharing effect in these models. In the example data objects x and y lie in the same page. P1 updates x and w in the first critical region and then updates z in the second critical region. According to EC and ScC, x and w are associated with the first critical region, and y and z are associated with the second critical region. When P2 enters the first critical region, x is updated by invalidating page 1; however, x is not accessed by P2. The invalidation notice for page 1 is still effective in the second critical. Then when P2 accesses y in the second critical region, a page fault on page 1 occurs, which is caused by the invalidation notice produced for updating x. This page fault is of no use for updating y and it is a false sharing effect.

![Figure 7: Reduce the false sharing effect in VC (Example 2)](image)

Once again, VC can remove the above false sharing effect by restricting the propagation of invalidation notices or by restricting the effective scope of invalidation notices. In Fig. 7 since x is not in the view CRV\textsuperscript{1} of P2, it should not be updated in P2. So the invalidation notice for page 1 is not propagated to P2, and P2 has no page fault on page 1 in the second critical region. Moreover, suppose x is mistakenly detected as in the view CRV\textsuperscript{2} and then the invalidation notice for page 1 is propagated to P2. Since VC can detect that the invalidation notice for page 1 has no use for updating y and z in the view CRV\textsuperscript{2}, VC makes it ineffective in the second critical region. So there is no page fault when P2 accesses y, and the above false sharing effect is removed.

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

## 4 Implementation

There are two technical issues in the implementation of VC. One is view detection, and the other is view transition. View 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 correct implementation of the VC model should guarantee that before a processor enters a new region, view detection and view transition are achieved correctly.

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

### 4.1 View detection

View detection is implemented at run time. In view detection, if a page is not modified it is not necessary to record it in a view, because it has no change and thus does not need consistency maintenance. Therefore, only the modified pages are recorded in a view in view detection.
To detect modified pages in view detection, our implementation takes advantage of the following two existing mechanisms needed by other schemes in the DSM system:

1. When a write access is performed on an invalidated page, a page fault will occur. The page fault handler in the DSM system will 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 will 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 completely all the modified pages of a view, we make all writable pages write-protected (read-only) before a new view is entered, which is the additional overhead for view detection. From our experimental results we know this additional overhead is trivial.

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

4.2 View transition

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

The update protocol is suitable for VC, just as 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. [17]

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 proposed in [17]. It uses the update protocol for the modified pages in CRVs, but the invalidation protocol for the modified pages in NRVs.

5 Experimental results

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

TreadMarks has adopted a multiple-writer scheme [6], which was proposed to minimize the effect of false sharing. In the multiple-writer scheme, initially a page is write-protected. When a write-protected page is first updated by a processor, a twin of the page is created and stored in the system space. When the updates on the page are needed
by another processor, a comparison of the twin and the current version of the page is done to create a diff, which can then be used to update copies of the page in other processors. Thus in the multiple-writer scheme the page diff, instead of the whole page, is used to renew an old copy.

Since our implementation of VC is based on TreadMarks, we have to compromise with the multiple-writer scheme on the 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 lies, 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 lies, 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 maintain the consistency of a page in the multiple-writer scheme, our implementation has tolerated the write/write false-sharing effect. Thus the write/write false-sharing effect has not been removed in our current implementation.

We use four applications in the experiment: TSP, QS, BT and Water. TSP, QS, and Water are provided by the TreadMarks research group. All the programs are written in the C language. TSP is the Traveling Salesman 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 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 recursively invoking itself on the two unsorted sublists. 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. These applications are representative of both numerical computing, e.g., Water and QS, and symbolic computing, e.g., TSP and BT.

Table 1 gives the performance results for LRC and VC. In the table, VC.j is the implementation based on the invalidation protocol; VC_h is the implementation based on the hybrid protocol; the Time is the total running time of an application program; the Diff_Req is the number of messages for diff request; the RPF is the reduction in page faults due to the use of the hybrid protocol in the VC model; The RFS is the reduction in page faults due to the reduction of the false-sharing effect in the VC model. the Msgs is the total number of messages.

<table>
<thead>
<tr>
<th>APP</th>
<th>Model</th>
<th>Time (Sec.)</th>
<th>Diff_Req</th>
<th>RPF</th>
<th>RFS</th>
<th>Msgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP</td>
<td>LRC</td>
<td>2.54</td>
<td>962</td>
<td>-</td>
<td>-</td>
<td>2763</td>
</tr>
<tr>
<td></td>
<td>VC_j</td>
<td>2.56</td>
<td>960</td>
<td>-</td>
<td>0</td>
<td>2756</td>
</tr>
<tr>
<td></td>
<td>VC_h</td>
<td>1.65</td>
<td>25</td>
<td>937</td>
<td>0</td>
<td>911</td>
</tr>
<tr>
<td>QS</td>
<td>LRC</td>
<td>7.09</td>
<td>3267</td>
<td>-</td>
<td>-</td>
<td>12209</td>
</tr>
<tr>
<td></td>
<td>VC_j</td>
<td>7.15</td>
<td>3330</td>
<td>-</td>
<td>0</td>
<td>12375</td>
</tr>
<tr>
<td></td>
<td>VC_h</td>
<td>4.59</td>
<td>791</td>
<td>1044</td>
<td>0</td>
<td>5301</td>
</tr>
<tr>
<td>BT</td>
<td>LRC</td>
<td>28.36</td>
<td>11437</td>
<td>-</td>
<td>-</td>
<td>79468</td>
</tr>
<tr>
<td></td>
<td>VC_j</td>
<td>27.59</td>
<td>11347</td>
<td>-</td>
<td>792</td>
<td>79426</td>
</tr>
<tr>
<td></td>
<td>VC_h</td>
<td>25.73</td>
<td>7429</td>
<td>3441</td>
<td>776</td>
<td>69342</td>
</tr>
<tr>
<td>Water</td>
<td>LRC</td>
<td>19.86</td>
<td>12428</td>
<td>-</td>
<td>-</td>
<td>96600</td>
</tr>
<tr>
<td></td>
<td>VC_j</td>
<td>19.91</td>
<td>12423</td>
<td>-</td>
<td>3</td>
<td>96600</td>
</tr>
<tr>
<td></td>
<td>VC_h</td>
<td>19.09</td>
<td>11891</td>
<td>511</td>
<td>3</td>
<td>95478</td>
</tr>
</tbody>
</table>

Table 1: Performance Statistics for applications on 8 processors
<table>
<thead>
<tr>
<th>APP</th>
<th>TPF</th>
<th>RFS</th>
<th>TFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP</td>
<td>1002</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>QS</td>
<td>3084</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>BT</td>
<td>13963</td>
<td>792</td>
<td>4347</td>
</tr>
<tr>
<td>Water</td>
<td>12046</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: Total number of page faults due to false-sharing effect

**VC_{h} vs. LRC**

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

**VC_{i} vs. LRC**

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

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

As we mentioned in Section 4, our current implementation of VC removes any write/read false-sharing effect, but does not remove the write/write false-sharing effect as a result of compromise with the multiple-writer scheme in TreadMarks. Thus the RFS showed in Table 1 is only the reduced number of page faults due to the reduction of the write/read false-sharing effect. Among the four applications, only BT and Water have the write/read false-sharing effect, and 6% of page faults in BT are due to the write/read false-sharing effect. Actually we have collected the total number of page faults that are due to false-sharing effect inside critical regions, and the results are showed in Table 2 as below.

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

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

6 Conclusion

In this paper we have proposed a novel View-based Consistency (VC) model for DSM and implemented it based on TreadMarks. Compared with other DSM consistency models, this model can achieve data selection without user
annotation and reduce more false sharing effect. Its only consistency requirement is that all the data objects in a processor’s new view must be updated during view transition. The further relaxation on consistency maintenance in VC produces more room of optimization for the implementation of DSM. Like some other weak Sequential Consistency models, however, the VC model can guarantee the same execution result as the Sequential Consistency model for data-race-free programs. Important issues and schemes on the implementation of VC have been presented and discussed. Performance results have shown that for all our applications the VC model outperforms the LRC model. We have also demonstrated that the extra overhead of view maintenance in VC is trivial. Further research should be carried out on accurate detection of NRVs and the reduction of write/write false sharing.

References


