Toward Transparent Selective Sequential Consistency in Distributed Shared Memory Systems

Chengzheng Sun, Zhiyi Huang, Wanju Lei, and Abdul Sattar
School of Computing & Information Technology
Griffith University, Brisbane, Qld 4111, Australia
Email:{scz,hzy,wlei,sattar}@cit.gu.edu.au

Abstract

This paper proposes a transparent selective sequential consistency approach to Distributed Shared Memory (DSM) systems. First, three basic techniques — time selection, processor selection, and data selection – are analyzed for improving the performance of strictly sequential consistency DSM systems, and a transparent approach to achieving these selections is proposed. Then, this paper focuses on the protocols and techniques devised to achieve transparent data selection, including a novel Selective Lazy/Eager Updates Propagation protocol for propagating updates on shared data objects, and the Critical Region Updated Pages Set scheme to automatically detect the associations between shared data objects and synchronization objects. The proposed approach is able to offer the same potential performance advantages as the Entry Consistency model or the Scope Consistency model, but it imposes no extra burden to programmers and never fails to execute programs correctly. The devised protocols and techniques have been implemented and experimented with in the context of the TreadMarks DSM system. Performance results have shown that for many applications, our transparent data selection approach outperforms the Lazy Release Consistency model using a lazy or eager updates propagation protocol.

Keywords: distributed shared memory, consistency model, time selection, processor selection, data selection, selective lazy/eager updates propagation.

1 Introduction

A Distributed Shared Memory (DSM) system provides application programmers the illusion of shared memory on top of message passing distributed systems, which facilitates the task of parallel programming in distributed systems. DSM has been an active area of research in parallel and distributed computing, with the goals of making the DSM systems more convenient to program and more efficient to implement [15, 2]. 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 program.* This means that in a SC-based DSM system, memory accesses from all processors may be interleaved in any sequential order which is consistent with each processor's memory access order, and the memory access orders observed by all processors are the same. One way to fully 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.

In practice, not all parallel applications are interested in seeing all memory updates made by other processors, let alone seeing them in order. Many parallel applications regulate their accesses to shared data by synchronization, so not all valid interleavings of their memory accesses are relevant to their real executions. Therefore, it is actually not necessary for the DSM system to force a processor to propagate **all** its updates to **every** other processor (with a copy of the shared data) at **every** memory update time. Under certain conditions, the DSM system can select the *time*, the *processor*, and the *data* for making shared memory updates public to improve the performance while still

appearing to be sequentially consistent. For example, consider a DSM system with four processors P_1, P_2, P_3, and P_4, where P_1, P_2, and P_3 share a data object x, and P_1 and P_4 share a data object y. Suppose all memory accesses to the shared data objects x and y are *serialized* among competing processors by means of synchronization operations to avoid data races. Under these circumstances, the following three basic techniques can be used: (1) **Time selection**: Updates on a shared data object by one processor are made visible to the public *only at the time* when the data object may be read by other processors. For example, updates on x by P_1 may be propagated outward only at the time after P_1 has finished updating x and has released the lock on x. (2) **Processor selection:** Updates on a shared data object are propagated from one processor to *only another processor* which is the next one in sequence to read the shared data object. For example, updates on x by P_1 may be propagated to only P_2 (but not to P_3) if P_2 is the next one in sequence to read x. (3) **Data selection:** Processors propagate to each other *only these shared data objects* which are really shared among them. For example, P_1, P_2, and P_3 may propagate to each other only data object x (not y), and P_1 and P_4 propagate to each other only data object y (not x). To improve the performance of the strict SC model, a number of weaker DSM consistency models have been proposed [6, 9, 13, 3, 11], which perform one or more of the above three selection techniques while appearing to be sequentially consistent. In the following discussion, these weaker DSM consistency models are called *selective SC models*.

The Weak Consistency (WC) model [6] is a selective SC model which performs *time selection*. The WC model requires a processor to propagate all its updates to other processors only at *synchronization time*, rather than at any memory access time. With time selection, updates 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 [9] takes *time selection* one step further than the WC model by distinguishing two different synchronization accesses: *acquire* and *release* accesses, which are the entry and exit of a critical region respectively. The ERC model requires that share memory updates be propagated outward only at *release access time*. In other words, the ERC model is more time-selective than the WC model by propagating updates 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 [13] improves the ERC model by performing both *time selection* and *processor selection*. Instead of propagating updates to all other processors at release access time as in the ERC model, the LRC model postpones updates propagation till another processor has successfully performed an *acquire* access. At successful acquire accesses, the DSM system is able to know precisely which processor is the next one to access the shared data, so updates can be propagated only to that particular processor (or no propagation at all if the next processor is the current processor), thus achieving *processor selection* in the LRC model. By sending updates only to the processor which has just entered a critical region, more messages can be reduced in the LRC model.

The Entry Consistency (EC) model [3] is very similar to the LRC model in propagating updates only to the next processor entering a critical region. In addition to *time selection* and *processor selection*, the EC model also performs *data selection* by propagating only these shared data objects which are associated with a critical region. These associations allow the EC model to avoid moving more data than necessary. With additional data selection, the EC model can be more efficient than the LRC model [4].

The Scope Consistency (ScC) model [11] is able to offer most of the potential performance advantages of the EC model by means of the time, processor, and data selections, and it also improves the programmability of the EC model by requiring programmers to attach consistency scopes with code sections, instead of data.

The selection techniques used by alternative consistency models are summarized in Table 1.

Model	TS	PS	DS
SC	No	No	No
WC	Sync. time	No	No
ERC	Rel. time	No	No
LRC	Acq. time	Next proc.	No
EC	Acq. time	Next proc.	Lock-data assoc. (user-annotation)
ScC	Acq. time	Next proc.	Scope-data assoc. (user-annotation)

Table 1: Selection techniques used in existing consistency models

All existing selective SC consistency models achieve

time/processor/data selection by requiring programmers to manually annotate the programs so that time/processor/data selection can be combined with synchronization operations. For example, the WC model requires programs to explicitly access special synchronization variables before and after accessing normal shared variables, so that the memory system is explicitly notified to propagate updates at synchronization access time. In the ERC model, programs are required to call an *acquire* primitive before accessing shared data objects and call a *release* primitive after accessing shared data objects, so that the memory system is explicitly notified of the entry and exit of a critical region, and can select the exit time to propagate updates. The LRC and EC models achieve both time and processor selections by requiring programs to explicitly call *acquire* and *release* primitives at the entry and exit of a critical region, respectively, so that the memory system can propagate updates only to the next processor at the entry time (instead of at the exit time as in the ERC model). Data selection in the EC model is achieved by requiring the programmer to explicitly associate synchronization objects with shared data objects. 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 explicitly define additional consistency scopes in programs due to correctness considerations. Although the programmer annotation approach can achieve time/processor/data selection effectively, the programmer has to be very careful about these annotations to ensure the correctness of the program. This imposes extra burden on programmers and increases the complexity of parallel programming.

The goal of our research is to design and implement an efficient DSM system based on a *transparent selective SC* approach, which is able to achieve both high performance and programming convenience by automatically selecting the right time, right processor, and right data for maintaining a sequentially consistent shared memory. Toward this end, we distinguish two types of programmers' annotations: one is the synchronization annotations which are required by both the correctness of parallel programs (to avoid data races) and the correctness of memory consistency; and the other is the annotations which are required only by the correctness of memory consistency. For the first type of annotations, such as the *acquire* and *release* synchronization primitives in the ERC, LRC, EC and ScC models, the DSM system can take advantage of them to achieve time/processor selection without imposing additional burden on programmers. However, for the second type of annotations, such as the association between synchronization objects and shared data objects for data selection in the EC model, and the additional consistency scopes in the ScC model, they are truly extra burden to programmers and should be replaced by automatic associations via run-time detection (and/or compile-time analysis). In addition, not all parallel programs require synchronized accesses to shared data to ensure correctness. Chaotic relaxation algorithms, for example, allow competing accesses to shared data without imposing ordering constraints [16]. Such algorithms work even if some read accesses do not return the most recent value. For these non-synchronizing parallel programs, additional mechanisms are needed by the transparent selective SC DSM system to achieve time/processor/data selection. In this paper, we will focus on the protocols and techniques used to achieve transparent data selection under the condition of synchronized shared memory accesses. In other words, parallel programs are assumed to use synchronization primitives, such as *acquire* and *release*, to avoid data races (as in the ERC, LRC, EC, and ScC models), but no programmers' annotation is required to associate shared data objects with synchronization operations, or to define consistency scopes.

The rest of this paper is organized as follows. In Section 2, two different protocols for propagating updates in DSM systems – *Lazy Updates Propagation* and *Eager Updates Propagation* – are first analyzed, and then a novel *Selective Lazy/Eager Updates Propagation* protocol is proposed to achieve data selection without affecting the correctness of program execution. Next, the techniques for automatically detecting the associations between shared data objects and critical regions are devised and discussed in Section 3. Our approach is compared to related work, such as the LRC, EC and ScC models, in Section 4. Experimental results are discussed in Section 5. Finally, the major contributions of this paper and future work are summarized in Section 6.

2 Updates propagation protocols for supporting data selection

A DSM updates propagation protocol determines when and how updates on one copy of a page are propagated to other copies of the same page on other processors. Updates on a page can be represented by

the updated page itself if there is only one writable copy of a page at any given time (i.e., a *single-writer scheme* [15, 8]), or by the *diff* obtained from the updated page and its *twin* if there are multiple writable copies of a page on different processors (i.e., a *multiple-writer* scheme [5]). An updates propagation protocol can be integrated with either a single-writer scheme or a multiple-writer scheme.

In the following discussion on updates propagation protocols, we assume a DSM system which achieves both *time selection* and *processor selection* by requiring programs to explicitly call the *acquire* and *release* primitives at the entry and exit of a critical region, respectively, and shared memory updates are propagated only to the next processor at the entry time.

2.1 The lazy and eager updates propagation protocols

There have existed a number of different protocols for propagating updates in DSM systems [10]. One protocol, adopted by the TreadMarks DSM system [1], works as follows: when an old copy of a page needs to be renewed, the old copy is invalidated first; only when the invalidated old copy is really accessed by a processor, are the updates of the page sent to the processor. We call this protocol as the *Lazy Updates Propagation* (LUP) protocol since it propagates updates lazily when updated pages are accessed. More precisely, the LUP protocol can be specified as follows.

Protocol 1 *The LUP protocol*
For any pair of processors P_1 and P_2 in a DSM system, suppose P_1 has left a critical region by calling *release*, and P_2 is the next processor to enter this critical region by calling *acquire*. The LUP protocol works as follows:

1. At the entry of the critical region, invalidation notices of all updated pages are propagated from P_1 to P_2, and all corresponding copies at P_2 are invalidated.

2. During the execution of the critical region, when an invalidated page is accessed, a page fault triggers the propagation of the updates of the missing page from P_1 to P_2. □

Alternatively, when a page needs to be renewed, the updates of the page, instead of the invalidations, are propagated immediately to update the old copy at the entry of a critical region. This protocol is called the *Eager Updates Propagation* (EUP) protocol since it propagates updates eagerly before updated pages are accessed. The EUP protocol can be specified more precisely as follows.

Protocol 2 *The EUP protocol*
For any pair of processors P_1 and P_2 in a DSM system, suppose P_1 has left a critical region by calling *release*, and P_2 is the next processor to enter this critical region by calling *acquire*. The EUP protocol works as follows:

1. At the entry of the critical region, updates of all pages are propagated from P_1 to P_2, and all corresponding copies at P_2 are updated.

2. During the execution of the critical region, no page fault or further updates propagation occurs. □

To illustrate the behavior of LUP and EUP, consider Program 1, in which both P_1 and P_2 access data objects x and y, and the two processors use *lock-1* to serialize their accesses. For simplicity, we assume that the shared data objects x and y are located on two distinct pages.

Program 1 *Memory access pattern in favor of EUP*

```
P1                P2
...               ...
Acquire(1);       Acquire(1);
write(x);         read(x);
write(y);         read(y);
Release(1);       Release(1);
...               ...
```

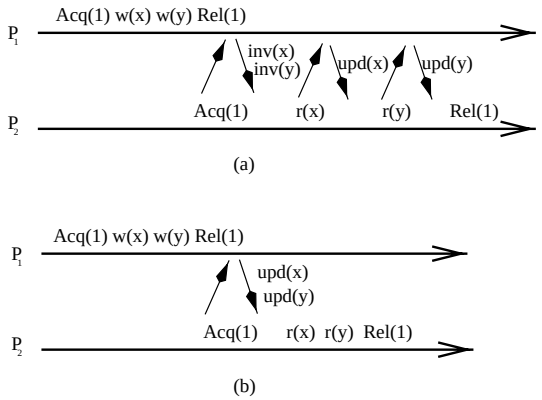

Figure 1: The message passing in executing Program 1 (a) under LUP, and (b) under EUP

The message passing between processors P_1 and P_2 using LUP is illustrated in Fig. 1(a). Suppose P_1 gets the lock first. Since the underlying DSM system can achieve both *time* and *processor* selections, the updates on x and y are propagated from P_1 to P_2 only when P_2 has acquired the lock. According to LUP, an invalidation message for x and y (actually piggybacked on the lock grant message) is first sent to P_2

and the page copies for x and y on P_2 become invalidated. Later when P_2 really accesses x and y, two page faults occur. Each page fault triggers two messages: one from P_1 to P_2 to request the updates, the other from P_2 to P_1 to propagate the updates.

On the other hand, the message passing between processors P_1 and P_2 for executing the same program using EUP is illustrated in Fig. 1(b). As explained above, when P_2 has acquired the lock, the updates of x and y need to be propagated to P_2. According to EUP, the updates of x and y, instead of their invalidations, are eagerly sent from P_1 to P_2. Consequently, no page fault occurs when accessing x and y by P_2.

For Program 1, EUP is obviously more efficient than LUP because the number of messages in EUP (2 messages) is much smaller than that in LUP (6 messages), and there is no page fault in EUP but two page faults in LUP. However, there exist other shared memory access patterns which may be in favor of LUP, instead of EUP. As shown in Program 2, the two processors P_1 and P_2 share data object y protected by *lock-2*. In addition, P_1 has access to data object x protected by *lock-1*, which may be shared with other processors (not shown in this picture).

Program 2 *Memory access pattern in favor of LUP*

```
P1                P2
...               ...
Acquire(1);       Acquire(2);
write(x);         read(y);
Release(1);       Release(2);
...               ...
Acquire(2);
write(y);
Release(2);
...
```

The message passing between P_1 and P_2 for executing Program 2 using LUP is illustrated in Fig. 2(a). After P_2 has acquired *lock-2*, P_1 sends invalidations of both x and y to P_2 and the page copies for x and y on P_2 become invalidated. Later only the updates of y are sent to P_2 since P_2 only accesses y. Through the separation of page invalidation and updates propagation, unnecessary updates transfers can be avoided in LUP.

However, EUP can cause unnecessary updates transfers between processors in this program. The updates of both x and y are sent eagerly from P_1 to P_2 after P_2 has acquired *lock-2*, as illustrated in Fig. 2(b). Obviously, the propagation of updates of x is unnecessary since P_2 will not access the page for x. The overhead of unnecessary updates propagation may make

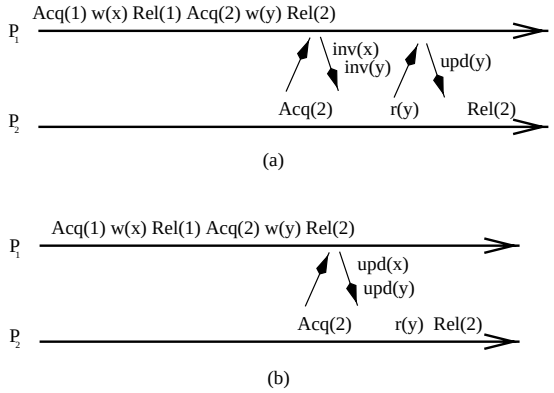

Figure 2: The message passing in executing Program 2 (a) under LUP, and (b) under EUP

EUP perform worse than LUP (see Section 5 for experimental performance results).

2.2 The selective lazy/eager updates propagation protocol

From above discussions we know that LUP can avoid useless updates propagation, but may cause more page faults and send more messages. On the other hand, EUP can avoid page faults and reduce the number of messages, but may propagate useless updates. There are favorable scenario for either LUP or EUP, but none of them can be an eternal winner with respect to the performance. In this section, we propose a hybrid updates propagation, called the *Selective Lazy/Eager Updates Propagation* (SLEUP) protocol, which is able to take advantage of the merits of both LUP and EUP to avoid useless updates propagation and to avoid page faults and reduce the number of messages. The SLEUP protocol relies on the knowledge about the association of synchronization objects and shared data objects. The *lock-data* association knowledge can be obtained transparently without programmers' annotations (to be discussed in the next section). Suppose the knowledge of *lock-data* association for a critical region is known, the SLEUP protocol can be specified as follows.

Protocol 3 *The SLEUP protocol*
For any pair of processors P_1 and P_2 in a DSM system, suppose P_1 has left a critical region by calling *release*, and P_2 is the next processor to enter this critical region by calling *acquire*. The SLEUP protocol works as follows:

1. At the entry of a critical region,
   (a) updates of these pages which are known to be associated with the current critical region

are propagated from P_1 to P_2, and all corresponding copies at P_2 are updated.

(b) invalidation notices of these updated pages which are unknown to be associated with the current critical region are propagated from P_1 to P_2, and all corresponding copies at P_2 are invalidated.

2. During the execution of the critical region, when an invalidated page is accessed, a page fault triggers the propagation of the updates of the missing page from P_1 to P_2. □

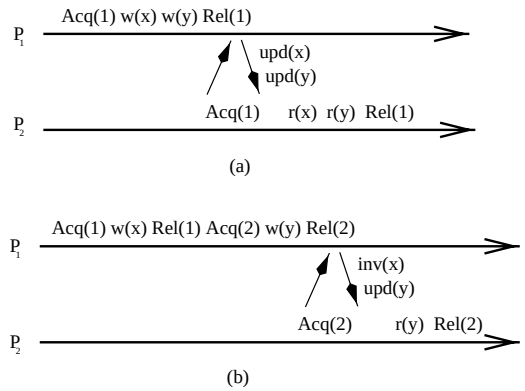

Figure 3: The message passing under SLEUP (a) in executing Program 1, and (b) in executing Program 2

For example, to execute Program 1 using SLEUP, if it is known in advance the pages for x and y are going to be used in P_2 after acquiring *lock-1*, the updates of x and y will be propagated from P_1 to P_2 together with the lock grant message. This is obviously the optimal arrangement for Program 1, as shown in Fig. 3(a). Also, for Program 2, if it is known in advance only the page for y is going to be used in P_2 after *lock-2* is acquired, only the updates of y are propagated to P_2, and the invalidation of x is also sent to P_2. Since P_2 does not access x during the execution of the critical region, no page fault occurs and no lazy updates acquiring messages are generated. This is obviously the optimal arrangement for Program 2, as shown in Fig. 3(b).

The SLEUP protocol performs best if the knowledge about the *lock-data* association is both *correct* and *complete*, i.e., none of the pages not really associated with the current critical region has been mistakenly regarded as associated, and all shared data pages really associated with the current critical region have been known as associated. However, if the knowledge about the *lock-data* association is *incorrect*, i.e., some pages not really associated with the current critical region has been mistakenly regarded as associated, the SLEUP protocol will propagate some useless updates as in the EUP protocol, but incorrect association has no effect on the correctness of the program execution. On the other hand, if the knowledge about the *lock-data* association is *incomplete*, i.e., some pages really associated with the current critical region has not been known, some page faults may occur and lazy updates acquiring messages may be generated as in the LUP protocol, but this incomplete association has no effect on the correctness of the program execution either. In summary, the degree of correctness and completeness of *lock-data* association knowledge has only effects on the performance of the SLEUP protocol, but has no any effect on the correctness of the program execution. Therefore, when the SLEUP protocol is integrated with a transparent approach to the detection of *lock-data* association, which may not be 100% correct or complete (see the next section), the SLEUP protocol may never fail to execute the program correctly and it can achieve data selection in critical regions without imposing any extra burden to programmers.

3 Automatic detection of lock-data associations

As mentioned in previous sections, the SLEUP protocol assumes that the *lock-data* associations have already been known. In this section, we propose a scheme for automatically detecting *lock-data* associations. The heuristic used in this scheme is that *the pages previously updated in a critical region by a processor will be most likely accessed the next time in the same critical region by the same/another processor.* Different critical regions are assumed to be delimited by the pairs of *acquire* and *release* primitives with different *lock identifiers*.

To implement this heuristic, each lock in a processor is associated with a *Ciritcal Region Updated Pages Set* (CRUPS). Initially every CRUPS is empty. During the execution of a critical region, the CRUPS of a lock keeps accumulating the identifiers of the updated pages. When a critical region is entered by a processor, the contents of the corresponding CRUPS obtained in the previous execution of this critical region by the lock granting processor will be used as the knowledge of *lock-data* associations by the SLEUP protocol. More precisely, the CRUPS scheme is described as below.

The CRUPS scheme

For any pair of processors P_1 and P_2 in a DSM sys-

tem, suppose P_1 has left a critical region by calling *release*, and P_2 is the next processor to enter this critical region by calling *acquire*. Let $CRUPS_{P_1}$ be the critical region updated page set for this critical region executed at P_1, and $CRUPS_{P_2}$ be the critical region updated page set for this critical region executed at P_2.

1. P_2 acquires the lock from P_1.
2. After receiving the lock acquiring message from P_2, P_1 propagates to P_2 the updates of these pages in $CRUPS_{P_1}$, and the invalidations of these updated pages not in $CRUPS_{P_1}$.
3. After receiving the lock grant message and handling the updates and invalidations according to the SLEUP protocol, P_2 empties its current $CRUPS_{P_2}$ for this lock.
4. During the execution of the critical region, P_2 records into $CRUPS_{P_2}$ the identifier of every updated page detected by an *write detection* scheme (to be discussed shortly).
5. At the exit of the critical region, P_2 stops recording in $CRUPS_{P_2}$, but keeps the contents of $CRUPS_{P_2}$ for use in the next acquisition of the same lock. □

To detect the write accesses during the execution of a critical region, the CRUPS scheme takes advantage of the following two existing mechanisms needed by the 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 missing page's identifier into the corresponding lock's CRUPS, as well as fetching the missing 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 updated page's identifier into the corresponding lock's CRUPS, as well as making a twin of the accessed page in the multiple-writer mode or obtaining the ownership of the accessed page in the single-writer mode.

Because the above two mechanisms have already been provided by the underlying DSM system, there is little extra overhead for recording the identifiers of accessed pages. However, the above access detection scheme is incomplete in the sense that some memory accesses, such as a write on a writable (non-protected) page, may not be detected. This incompleteness can be remedied by a more elaborate write detection mechanism [17].

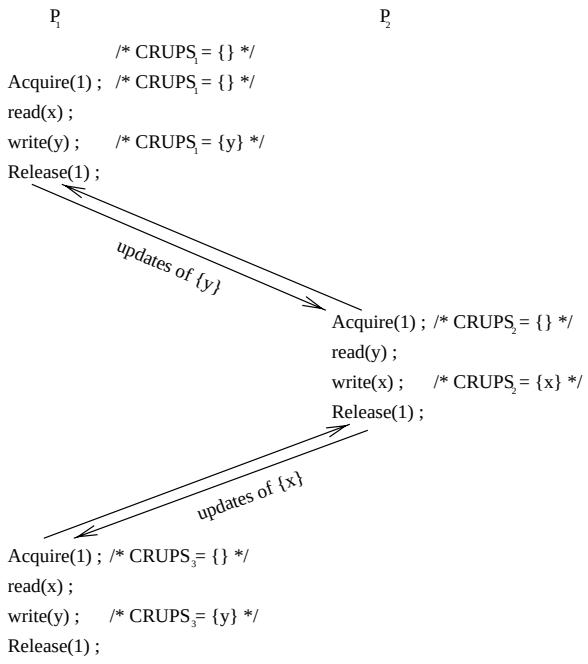

Figure 4: An example for the CRUPS scheme

To illustrate how the CRUPS scheme works, we give an example in Fig. 4. Suppose P_1 reads x and writes y, and P_2 reads y and writes x. For the first time when P_1 enters the critical region, its write on y is detected and therefore y is recorded into $CRUPS_1$. When P_2 acquires the lock, P_1 piggy-backs updates of y on the lock grant message. When P_2 receives $CRUPS_1$, it empties $CRUPS_1$. During the execution of this critical region at P_2, the write on x is detected and therefore x is recorded into $CRUPS_2$. When P_1 acquires the lock again, P_2 piggy-backs updates of x on the lock grant message. The experimental results in Section 5 demonstrate the effectiveness of the CRUPS scheme.

4 Comparison with LRC, EC and ScC

Compared to the LRC model using the LUP protocol, the SLEUP protocol integrated with the transparent CRUPS detection scheme is more efficient since it can reduce the number of page faults and lazy updates acquiring messages (see the experimental results

in Section 5). To improve the performance of the LRC model, a Lazy Hybrid (LH) protocol was proposed in [7, 13]. The LH protocol tries to reduce the number of page faults by speculatively prefetching data before it is requested. The idea behind the LH protocol is that programs usually have significant temporal locality, and therefore any page accessed by a process in the past is likely to be accessed in the near future. The LH protocol therefore selects updates of pages that have been accessed in the past (regardless whether or not in the same critical region) by the processor acquiring a lock, and piggy-backs the updates on the lock grant message. The similarity between LH and SLEUP is that both of them use some kinds of locality heuristics to select updates of pages for piggy-backing on the lock grant message. The major difference between LH and SLEUP is the following: the former uses a heuristic without distinguishing the accessed pages which are protected by the same lock from these pages which are not, but the latter makes this distinction and hence can be more accurate in selecting the updates for eager propagation. Since the heuristic in the LH protocol is very speculative, it can cause unnecessary updates transfers, thus degrading its performance. This point has been confirmed by our experimental results in Section 5.

Integrated with the transparent CRUPS detection approach, the SLEUP protocol can achieve data selection without imposing any extra burden on programmers, and may never fail to execute the program correctly. In contrast, the EC model also uses *lock-data* associations to achieve data selection, but it requires programmers to correctly annotate the *lock-data* associations of the program. This represents not only a substantial burden for programmers, but it also may fail to execute the program correctly if the programmers' annotation is not accurate.

The ScC model is a step toward transparent data selection, but it still requires programmers to correctly define the consistency scopes in programs. When programming in the ScC model, programmers have to be concerned about both the synchronization scope (e.g. a critical region), which is required by the correctness of the program to avoid data races, and the consistency scope, which is required by the correctness of the ScC model. The degree of transparency in achieving data selection in the ScC model depends on the degree of match between synchronization scopes and consistency scopes in programs. If all synchronization scopes match the consistency scopes in a program, a synchronization scope defined by the use of *acquire* and *release* primitives effectively defines a consistency scope, which can be taken advantage of by the ScC run-time system to achieve scope consistency. However, if synchronization scopes do not completely match consistency scopes in a program, programmers have to either use *acquire* and *release* primitives to define additional synchronization scopes (thus additional consistency scopes), or use consistency-special primitives like *open_scope* and *close_scope* to explicitly define additional consistency scopes to ensure the correctness of the program, which is a nontrivial task to programmers. Therefore, the major differences between the ScC model and our approach in this paper are: (1) our approach is completely transparent in achieving data selection, whereas the ScC model is only partially transparent; and (2) our approach may never fail to execute a program which is correct from the synchronization point of view, whereas the ScC model may produce wrong results for a program which is correct from the synchronization point of view, but which contains incorrect definitions of consistency scopes.

5 Experimental results

In this section, we presents an experimental evaluation of LUP, EUP, LH, and SLEUP protocols. All these protocols are implemented in TreadMarks [1]. The experimental platform consists of 8 SGI workstations running IRIX Release 5.3, which are connected by a 10 Mbps Ethernet. Each of the workstations has a 100 MHz processor and 32 Mbytes memory. The page size in the virtual memory is 4 KB.

TreadMarks has adopted a *multiple-writer* scheme [5], which was proposed to minimize the effect of *false sharing* problem [12]. 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. So in the multiple-writer scheme the page $diff$, instead of the whole page, is used to renew an old copy.

We use 4 applications in the experiment: *TSP, QS,* BT and *Water*. *TSP, QS*, and *Water* are provided by TreadMarks research group. All the programs are written in 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 op-

APP	Prot	Time (Sec.)	PF	Mesgs	DiffD (KB)	TotD (KB)
TSP	LUP	15.86	1029	2846	449	1268
	EUP	6.39	7	734	456	1250
	LH	8.63	355	1405	463	1287
	SLEUP	7.33	245	1209	444	1253
QS	LUP	20.09	3046	10432	6100	10153
	EUP	42.08	103	7607	26549	30330
	LH	15.52	962	6095	6963	10845
	SLEUP	13.36	956	5936	5355	9165
BT	LUP	82.92	26478	96979	8921	39511
	EUP	83.34	352	43505	13537	44206
	LH	72.08	13918	68542	9390	40965
	SLEUP	69.71	6469	53925	8762	39149
Water	LUP	32.59	4314	24495	9980	11718
	EUP	40.49	2301	20461	16013	17807
	LH	32.83	3757	23354	11579	11915
	SLEUP	31.07	3024	21920	9982	11834

Table 2: Performance Statistics for applications

erates 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 timestep, the intra- and inter-molecular forces incident on a molecule are computed. These applications are representative in either numerical computing, e.g., *Water, QS*, or symbolic computing, e.g., *TSP, BT*.

Table 2 gives the performance results for the various protocols. In the table, the *Time* is the total running time of an application program; the *PF* is the number of page faults; the *Mesgs* is the total number of messages; the *DiffD* is the sum of total $diff$ data; and the *TotD* is the sum of total message data.

SLEUP vs. LUP: SLEUP outperforms the LRC model using LUP for every application. The reason is that SLEUP piggy-backs useful updates on lock grant messages before they are requested, and therefore the number of page faults is reduced in SLEUP. Table 2 shows the number of page faults in SLEUP is significantly less than that in LUP (76.2% less in TSP, 75.6% less in BT, 68.6% less in QS, and 29.9% less in $Water$). Consequently the number of messages in SLEUP has been greatly reduced. Also from Table 2 we notice that there is no significant change of the amount of $diff$ data between LUP and SLEUP. This suggests that SLEUP does not send useless updates based on the CRUPS scheme for automatic detection of *lock-data* associations.

SLEUP vs. LH: SLEUP outperforms LH for every applications. The reason is that SLEUP can selectively propagate updates more accurately by using the CRUPS-based *lock-data* associations. From Table 2, we notice that LH normally sends more useless $diff$ data and has more page faults than SLEUP (30.0% more $diff$ data in QS, 16.0% more $diff$ data in $Water$, 7.2% more $diff$ data in BT, 4.3% more $diff$ data in TSP, 115.1% more page faults in BT, 44.9% more page faults in TSP, 24.2% more page faults in $Water$, and 0.6% more page faults in QS). Accordingly, LH has more messages and propagates more data than SLEUP.

About EUP: The performance of EUP is normally the worst among the four protocols, since it blindly and eagerly propagates updates without any *data selection*. As a result, EUP has sent a large amount of useless updates in these applications (its $diff$ data is five times of SLEUP's $diff$ data in QS, see Table 2). Even though the number of page faults in EUP is the least among the protocols, its number of messages sometimes is even greater than SLEUP (see the number of messages in QS in Table 2). The reason is the $diff$ data propagated at an *acquire* access can not be accommodated in one message (the maximum size of a UDP message is 32768 in SGI workstations), and have to be sent in several messages, which increases the total number of messages. The only exception that EUP performs better is TSP, because in TSP every processor almost needs to access every shared data object in a critical region and therefore TSP is in favor of eager updates propagation.

6 Conclusions

In this paper, we have proposed and discussed a novel transparent selective sequential consistent approach to DSM systems. We identified three basic techniques — time selection, processor selection, and data selection – for improving the performance of strict sequential consistency systems, and proposed a transparent approach to achieving these selections without imposing extra burden to programmers. Furthermore, we devised the protocols and techniques for achieving transparent data selection, including a novel SLEUP protocol for propagating updates on shared data, and the CRUPS-based techniques for automatically detecting

the associations between shared data objects and synchronization objects. The proposed approach is able to offer the same potential performance advantages as the Entry Consistency model or the Scope Consistency model, but it does not require programmers to provide any additional annotations for synchronization-data associations or consistency scopes, and it never fails to execute a parallel program which is correct from the synchronization point of view, but which may be incorrect from the synchronization-data association or the consistency scope point of views. The devised protocols and techniques have been implemented and experimented with in the context of the TreadMarks DSM system. Performance results have shown that for many applications, the proposed transparent data selection approach using SLEUP outperforms the LRC model using LUP, EUP, or LH. We are currently investigating additional run-time detection techniques combined with compile-time analysis to achieve more accurate association of shared data objects with synchronization operations.

Acknowledgments

The work reported in this paper is supported in part by an ARC (Australian Research Council) large grant (A49601731), and a NCGSS grant by Griffith University.

References

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

[2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum: "Orca: A language for parallel programming of distributed systems," *IEEE Transactions on Software Engineering,* vol. 18, pp. 190-205, March 1992.

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

[4] B.N. Bershad, et al: "The Midway Distributed Shared Memory System," *In Proc. of IEEE COMPCON Conference,* pp528-537, 1993.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel: "Techniques for reducing consistency-related information in distributed shared memory systems," *ACM Transactions on Computer Systems,* 13(3):205-243, August 1995.

[6] M.Dubois, C.Scheurich, and F.A.Briggs: "Memory access buffering in multiprocessors," *In Proc. of the 13th Annual International Symposium on Computer Architecture*, pp.434-442, June 1986.

[7] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel: "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology," In *Proc. of the 20th Annual International Symposium on Computer Architecture,* pp. 144-155, May 1993.

[8] B. Fleisch and R.H. Katz: "Mirage: A coherent distributed shared memory design," *In Proc. of the 12th ACM Symposium on Operating Systems Principles,* pp211-223, Dec. 1989.

[9] K. Gharachorloo, D.Lenoski, J.Laudon: "Memory consistency and event ordering in scalable shared memory multiprocessors," *In Proc. of the 17th Annual International Symposium on Computer Architecture,* pp15-26, May 1990.

[10] Zhiyi Huang, Wanju Lei, Chengzheng Sun, and Abdul Sattar: "Heuristic Diff Acquiring in Lazy Release Consistency Model," *In Proc. of 1997 Asian Computing Science Conference,* LNCS 1345, Springer-Verlag, pp98-109, Dec. 1997.

[11] L. Iftode, J.P. Singh and K. Li: "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," *In Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures,* 1996.

[12] T.E. Jeremiassen and S. Eggers: "Reducing false sharing on shared memory multiprocessors through compile-time data transformations," *In Proc. of the 5th Symposium on the Principles and Practice of Parallel Programming,* July 1995.

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

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

[15] K.Li, P.Hudak: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems,* Vol. 7, pp321-359, Nov. 1989.

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

[17] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad: "Software write detection for distributed shared memory," *In Proc. of the 1st OSDI Symposium,* Nov. 1994.