# Performance of Cache Coherency Schemes in a Shared Disks Transaction Environment*

Haengrae Cho

Department of Computer Engineering, Yeungnam University
Kyongsan, Kyongbuk 712-749, Korea

## Abstract

*In a shared disks environment (SDE), the computing nodes are locally coupled via a high-speed network and share a common database at the disk level. To reduce the amount of expensive and slow disk I/O, each node caches database pages in its main memory buffer. This paper focuses on SDE that uses record-level locking as a concurrency control. While the record-level locking can guarantee higher concurrency than page-level locking, it may result in heavy message traffic. In this paper, we first propose a cache coherency scheme that can reduce the message traffic in the standard locking. Then the scheme is extended to the context where lock retention and lock de-escalation are adopted. Using a distributed database simulation model, we evaluate the performance of the proposed schemes under a wide variety of database workloads.*

## 1 Introduction

One approach to improve the capacity and availability of a single transaction system is to couple multiple computing nodes. There are two primary flavors of the coupling: *shared nothing* [13] and *shared disks* [10]. In shared nothing, each data partition is owned by a single node, and only the node can directly read and modify its partition. When several partitions are accessed, a two-phase commit protocol is needed. Cache and lock management are done locally by each partition server. In shared disks, all coupled nodes are connected via a high-speed network and share a common database at the disk level. Nodes need to hold locks on shared data, both when active transactions need the data and when the data is cached at the nodes. Such shared access requires distributed locking and cache management.

Our focus in this paper is on caching and locking, and on how to reduce their overhead in a shared disks environment (SDE) while providing high concurrency. The SDE has been adopted in a number of commercial products and research prototypes. They are IBM's IMS/VS Data Sharing product [14], Amoeba research project [12], DEC's VAX DBMS and VAX Rdb/VMS [4], and Oracle's parallel server [9].

The remainder of this paper is organized as follows: Section 2 presents the issues about caching and locking in SDE. Section 3 describes the details of coherency control schemes proposed in this paper. The simulation model is developed in Section 4, and Section 5 analyzes the performance of the proposed schemes using the simulation model. A summary and concluding remarks appear in Section 6.

## 2 Caching and Locking in SDE

This paper considers an SDE where a record-level locking is adopted as a concurrency control. While the SDE with record-level locking can guarantee higher concurrency, it could suffer from heavy message traffic [10]. The message traffic results from two overheads: *locking overhead* and *coherency control overhead*. In this section, we describe each overhead in detail and present previous works.

### 2.1 Caching

Caching may substantially reduce the amount of disk accesses by utilizing the locality of reference. However, since a particular page may be simultaneously cached in different nodes, modification of the page in any buffer invalidates copies of that page in other nodes. This necessitates the use of a *cache coherency scheme* so that the nodes always see the most recent version of database pages.

The coherency control overhead comes from implementing latches in the SDE to satisfy the physical consistency of a page. Specifically, to achieve the cache coherency, each node is required to verify page validity when a record of the page is accessed. If its cached page is out-of-date, the node should receive recent version of the page from other nodes. Note that both procedures may cause message traffic between nodes.

*This research was supported by the Korean National Technology Development Project, sponsored by the Korean Ministry of Information and Communication under contract AB-97-B-0067.

There are two approaches of cache coherency schemes for fine-granularity locking. They are *merging* scheme [1] and *write token* scheme [7]. The merging scheme permits simultaneous updates on the same page. Since writing out partially up-to-date pages could lead to lost updates, all modifications must be merged before the page is written to the shared disks. As is discussed in [1], merging can be CPU intensive and may also require additional disk I/O to access logs of each update.

In the write token scheme, simultaneous updates on the same page are prohibited. This is achieved by the notion of *physical lock* (P lock). To update a page, a node is required to become the "owner" of that page by requesting P lock with exclusive mode; when a different node tries to update the page or to read the recent version of the page, it must be sent the most recent copy of the page by the current owner. By permitting only one P lock per page with exclusive mode, the physical consistency of a page can be satisfied. Note that the write token scheme may suffer from the heavy message traffic due to frequent page transfers between nodes.

### 2.2 Locking

Note that the lock granule of record-level locking is smaller than that of page-level locking. This means that the record-level locking requires large number of messages for lock requests and lock responses between nodes and a global lock manager (GLM).

There are two approaches to reduce the locking overhead in SDE. They are *lock caching* [3] and *lock de-escalation* [5]. In the lock caching, locks can be cached by a node even after the requesting transaction has committed. Then the node can handle lock requests of later transactions *locally*. When a node requests a lock that conflicts with one or more locks that are currently cached at other nodes, the GLM "calls back" the conflicting locks by sending requests to the nodes that have those locks cached. The lock request is granted only when the GLM has determined that all conflicting locks have been released, so the callback scheme ensures that nodes can never concurrently hold conflicting locks.

The basic idea of lock de-escalation is to obtain locks at the coarsest granularity (e.g., file) for which no concurrency control conflict exists in order to minimize interaction with the GLM. However, to avoid undue data contention, locks can later be "de-escalated" into finer grained locks (e.g., records) if subsequent conflicts. More elegant ideas have been proposed to support fine-grained sharing in a page server object-oriented DBMS [1]. However, as we have noted previously, it adopts merging scheme that might result in poor performance in the SDE.

## 3 Cache Coherency Schemes

In this section, we propose a cache coherency scheme, named *cache check scheme* (CCS) that can alleviate the heavy message traffic of the write token scheme (WTS). We first present a system architecture. Then we describe the basic idea of CCS, where the locking strategy is similar to the WTS. Finally, we extend the CCS where the notion of lock caching and lock de-escalation are adopted.

### 3.1 System Architecture

The SDE consists of multiple loosely coupled nodes sharing a common database at the disk level. Standard two-phase locking is used for concurrency control where a global lock manager (GLM) is assumed to be available. The GLM not only provides global locking functions for every node, but also tracks which node has a valid copy of a page. Each node maintains a local buffer and caches a part of the database in this buffer. An LRU buffer replacement scheme is used by each node for its local buffer management. To access a data page, a transaction requests a copy of the item from the local buffer manager. The buffer manager returns a copy of the requesting transaction if the page is present in the buffer. Otherwise, a copy of that page is brought from the shared disk or other node's buffer to the local buffer. In either case, the newly accessed page is placed at the top of the LRU stack.

We assume *no force* buffer management policy [8], where a transaction is allowed to commit even though all pages modified by it are not written to disk. Note that every page in the database has a page_LSN field which contains the log sequence number (LSN) of the log record that describes the latest update to that page [8]. By comparing page_LSNs of two pages, we can determine which page is more recent.

The GLM maintains two lock tables: (1) logical lock table, that registers record locks held by active transactions, and (2) physical lock table, that registers the recent page_LSN, page owner, and copy set of cached page. A *copy set* of a page represents a set of nodes caching recent version of the page. If there is a page owner, it is always included in the copy set of the page. The copy set may also include other nodes.

### 3.2 Basic Idea

Before describing the details of the CCS, we first illustrate the problem of transferring unnecessary messages in the WTS. Example 1 shows this problem.

**Example 1:** Suppose a page $P_A$ contains three records $R_1$, $R_2$, and $R_3$, and a transaction $T_1$ at node $N_1$ commits after updating $R_1$ to $R_1'$. As a result, a new page_LSN of $P_A$ is registered in the GLM. After that, suppose a transaction $T_2$ at node $N_2$ tries to read $R_2$ in $P_A$ that was also cached at $N_2$. Since the page_LSN of $P_A$ at $N_2$ is smaller than that at the GLM, the copy of $P_A$ at $N_1$ should be transferred to $N_2$. Note that the page transfer from $N_1$ to $N_2$ is not necessary. This is because both copies of $P_A$ at $N_1$ and $N_2$ have the same version of $R_2$. $P_A$ needs to be transferred only if $T_2$ tries to read $R_1$ or to update any records in $P_A$. □

The CCS allows a node to receive a page only if the node does not cache the recent version of a *record* to be accessed; in contrast, WTS makes a node receive a page if the node does not cache the recent version of the *page*. By transferring a page only if the cached record has been updated by other nodes, the CCS can eliminate the unnecessary message transfer.

In the CCS, when a transaction commits, a commit message is sent to the GLM. The commit message includes a list of [record identifier, page_LSN of a page that includes the record] for each updated record. The GLM stores this information at the *record cache table* (RCT). The RCT has a role to maintain the recent page_LSN of every cached updated record.

When a transaction $T_i$ at node $N_i$ tries to access a record $R_i$ at page $P_A$, a lock request message is sent to the GLM. The lock request message is a 5-tuple: [transaction identifier, node identifier, lock mode, record identifier, page_LSN of a page that includes the record]. The lock mode is $S$ for read lock and $X$ for write lock. If $P_A$ is not cached at $N_i$, page_LSN is set to "-1". Let page_LSN($P_A$) be the value of page_LSN transferred from $N_i$ to the GLM, and page_LSN(RCT) be the value of page_LSN of $R_i$ maintained at the RCT. If $R_i$ has not been updated recently, page_LSN(RCT) is defined as $\phi$. Once the lock for $R_i$ is accepted, one of the following cases may be happened.

1. $S$ lock $\wedge$ page_LSN(RCT) $= \phi$: This implies that $R_i$ has not been updated recently. If $N_i$ caches $P_A$ (i.e., page_LSN($P_A$) $\neq -1$), the GLM sends a lock grant message to $N_i$, and then $N_i$ can read $R_i$ from its cached $P_A$. Otherwise, the GLM sends a page transfer message to $N_i$. $N_i$ now should receive recent version of $P_A$ from some node in the copy set of $P_A$.[1] If the copy set of $P_A$ is empty, the GLM sends a disk access message to $N_i$ and then $N_i$ should read $P_A$ from the shared disks. In the latter two cases, the GLM registers $N_i$ to the copy set of $P_A$.

2. $S$ lock $\wedge$ page_LSN($P_A$) $\geq$ page_LSN(RCT): This implies that the version of $R_i$ at $N_i$ is recent. So the GLM sends only the lock grant message to $N_i$, and $N_i$ is allowed to read $R_i$ from its cached $P_A$ without any overhead.

3. $S$ lock $\wedge$ page_LSN($P_A$) $<$ page_LSN(RCT): $R_i$ has been updated by other nodes and thus the version of $R_i$ at $N_i$ is not recent. $N_i$ should get the recent version of $P_A$ from some node in the copy set or from the shared disks. In both cases, $N_i$ is included in the copy set of $P_A$.

4. $X$ lock: This case is similar to WTS. If $N_i$ is not a page owner of $P_A$, the ownership is changed to $N_i$ after the current page owner issues, if necessary, a log force for ensuring the write-ahead logging protocol. Now, if $N_i$ is included in the copy set of $P_A$, the GLM sends a lock grant message to $N_i$, and then $N_i$ can update $R_i$. Otherwise, $N_i$ should get the recent version of $P_A$ from some node in the copy set or from the shared disks. In every case, $N_i$ becomes a page owner of $P_A$, and the copy set of $P_A$ is changed to include only $N_i$.

[1]The GLM selects a node *randomly* from the copy set that transfers $P_A$ to $N_i$. This is because if a fixed node (e.g., page owner) responds every page request, the node will suffer from frequent interrupts from the GLM. Furthermore, the queuing delay of each page request would also increase.

Consider the Example 1 again. In CCS, a commit message [$R_1$, new page_LSN of $P_A$] is transferred to the GLM when $T_1$ commits. Then the GLM stores "new page_LSN of $P_A$" to the RCT. After that, $T_2$ can read $R_2$ at its cached copy of $P_A$ even though $N_2$'s version of $P_A$ is not recent. This is because $R_2$ has not been updated recently, and thus the RCT does not maintain page_LSN about $R_2$ (case 1). On the other hand, when $T_2$ tries to read $R_1$, it sends a lock request message [$S$, $R_1$, page_LSN of $P_A$ at $N_2$] to the GLM. After the lock is granted, the GLM compares "new page_LSN of $P_A$" at the RCT and "page_LSN of $P_A$ at $N_2$" transferred by $T_2$. Since "new page_LSN of $P_A$" at the RCT is larger, $N_2$ has to receive recent version of $P_A$ from $N_1$ (case 3).

## 3.3 Extension

In this section, we extend the basic CCS to the context where the notion of lock caching and lock de-escalation are used. We first describe our locking strategy and then present the extended CCS.

### 3.3.1 Locking Strategy

The basic idea of our locking strategy is to lock a page if possible, rather than to lock a record; hence, subsequent lock requests for records of the page can be handled locally. Note that the owner of page lock is a *node*, while the owner of record lock is a *transaction*. The GLM maintains page locking information, and each node registers its read and write locks locally at the record-level granularity. If the page locks need to be de-escalated, the local lock information should be transferred to the GLM.

We support five lock modes for a page: $IS$, $IX$, $S$, $SIX$, and $X$. The lock compatibility is equal to that in the traditional multi-granularity locking. *IS* lock implies that some records on the page are currently being read, while *IX* lock means that there are some record updates. Note that in both lock modes, the node should forward each (record) lock request to the GLM. However, the other lock modes may not require communication with GTM. If the node has an $S$ or *SIX* lock on a page, $S$ lock request on a record of the page can be handled locally. On the other hand, the node forwards $X$ record lock requests to the GLM. In this case, the $S$ page lock will be upgraded to *SIX* mode. $X$ lock on a page implies that the node can handle locally all the lock requests on records of the page.

Page locks of *IS* or *IX* modes are released when the requesting transaction is committed. However, page locks of the other modes are cached even after the requesting transaction has committed.[2] Since the cached page locks may cover several record locks, the record locks can also be cached. This implies the GLM and other nodes may regard that both locks of a node have same duration.[3] While our approach can support simple lock modes compared to [6], it is rather tricky to coordinate page lock and record lock. We now describe this issue in detail.

Suppose a node requests an $S$ lock on record $R_1$ of page $P_A$ to the GLM. This implies that the node may have no lock, *IS* lock, or *IX* lock on $P_A$. If the record lock can be granted, the GLM may escalate the page lock of the node. Suppose that MAX($P_A$) means the strongest lock mode among locks held by other nodes. Table 1(a) shows page lock escalation for $S$ record lock request. In the table, "IMP" means that the combination is impossible. "CB" means that the GLM *calls back* the conflicting locks by sending requests to the owner nodes of those locks. Then the nodes de-escalate their locks. To do so, they obtain record-level locks at the GLM for any records that they have accessed while holding the page lock. After de-escalation is done, the GLM checks for record-level conflicts and then performs lock escalation once again for new MAX($P_A$). The escalated page lock mode is piggybacked with the lock grant message. Table 1(b) shows page lock escalation for $X$ record lock request.

### 3.3.2 Extended CCS

Suppose a transaction $T_i$ at node $N_i$ tries to access a record $R_i$ at page $P_A$. If $N_i$ does not have corresponding page lock on $P_A$, the lock request message is sent to the GLM. The structure of lock request message is same to the basic CCS. Then the GLM handles the lock request according the locking strategy of Section 3.3.1. Suppose *P-Lock* implies the resulting page lock that will be allowed to $N_i$, page_LSN(GLM) implies the page_LSN of $P_A$ at the physical locking table, and page_LSN(RCT) is the value of page_LSN of $R_i$ maintained at the RCT. Once the lock for $R_i$ is accepted, one of the following cases may be happened.

1. $S$ lock $\wedge$ *P-Lock* $=$ $S$ or *SIX*: If page_LSN($P_A$) is equal to page_LSN(GLM), the GLM sends an $S$ lock grant message on $P_A$, and $N_i$ can read $R_i$ from its cached $P_A$. Otherwise, the GLM sends a page transfer message together with the lock grant message to $N_i$. The recent version of $P_A$ is transferred from some node in the copy set of $P_A$. If the copy set is empty and page_LSN($P_A$) is not recent, the GLM sends a disk access message together with the lock grant message to $N_i$.
2. $S$ lock $\wedge$ *P-Lock* $=$ *IS* or *IX*: This case is same to the basic CCS, and page_LSN($P_A$) is compared with page_LSN(RCT). For every case, a lock grant message on $R_i$ is transferred to $N_i$.
3. $X$ lock: If page_LSN($P_A$) is not equal to page_LSN(GLM), $N_i$ should have recent version of $P_A$ from the other nodes or disks. $N_i$ becomes the page owner of $P_A$.

Note that in case 1 page_LSN($P_A$) is compared with page_LSN(GLM) not with page_LSN(RCT). Hence, unlike the basic CCS, $P_A$ may be transferred even though $R_i$ is recent. This is because $S$ or *SIX* lock on $P_A$ implies that every record of $P_A$ will be read locally without communicating to the GLM. By transferring the most recent page, it is guaranteed that every record of the page is recent.

Table 1: Page Lock Escalation

| Current Lock Mode | MAX($P_A$) | | | | | |
|---|---|---|---|---|---|---|
| | Null | IS | IX | S | SIX | X |
| Null | S | S | IS | S | IS | CB |
| IS | S | S | IS | S | IS | IMP |
| IX | S | SIX | IX | IMP | IMP | IMP |

(a) $S$ record lock request

| Current Lock Mode | MAX($P_A$) | | | | | |
|---|---|---|---|---|---|---|
| | Null | IS | IX | S | SIX | X |
| Null | X | SIX | IX | CB | CB | CB |
| IS | X | SIX | IX | CB | CB | IMP |
| IX | X | SIX | IX | IMP | IMP | IMP |
| S | X | SIX | IMP | CB | IMP | IMP |
| SIX | X | SIX | IMP | IMP | IMP | IMP |

(b) $X$ record lock request

[2]The *SIX* lock is downgraded to $S$ lock, when the updating transaction commits.

[3]Note that this is not true in [6], where page lock and record lock are assumed to have different duration. As a result, in [6], a number of complicated lock modes are proposed that combine page lock and record lock.

## 4 Simulation Model

We evaluate the performance of coherency control schemes using a simulation model. Figure 1 shows the simulation model for the SDE. The simulation model was implemented using the CSIM discrete-event simulation package [11].

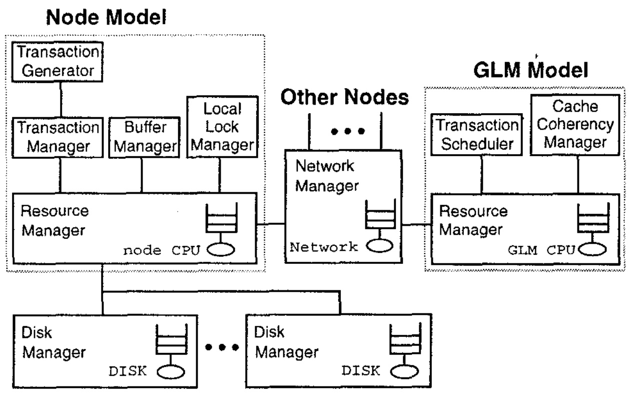

Figure 1: Simulation Model of SDE

We model the SDE consisting of a single GLM plus a varying number of nodes, all of which are connected via a local area network. The disks are shared by every node. The model for each node consists of a *buffer manager*, which manages the node buffer pool using an LRU policy, and a *resource manager*, which models CPU activity and provides access to the shared disks and the network. The *transaction generator* of each node has a role to generate transactions, each of which is modeled as a sequence of database operations, i.e., each lock request is followed by a database access operation (a read or a write). For each transaction, the *transaction manager* forwards lock request messages and commit messages to the GLM. The *local lock manager* is required only in the extended CCS, where record-level locking functions are performed.

The GLM has a role to perform the concurrency control and the cache coherency control. The *transaction scheduler* supports two-phase locking and deadlock resolution. The lock granularity is defined as a record. Transactions wait on any conflicting lock request, and are aborted only in case of deadlocks. Each deadlock is checked for each wait using the wait-for graph, and a transaction making the request is selected as a victim. Locks are released together at the end-of-transaction. In case of extended CCS, the transaction scheduler also implements the page locking strategy of Section 3.3.1. The *cache coherency manager* captures the semantics of a given coherency control scheme. We have implemented a total of three cache coherency managers, including basic CCS, extended CCS, and WTS.

Table 1 shows the simulation parameters for specifying the resources and overheads of the system and

Table 2: Simulation Parameters

| System Parameters | | |
|---|---|---|
| *LCPUSpeed* | Instruction rate of node CPU | 30 MIPS |
| *GCPUSpeed* | Instruction rate of GLM CPU | 100 MIPS |
| *NetBandwidth* | Network Bandwidth | 10 Mbps |
| *NumNode* | Number of computing nodes | 2 - 15 |
| *NumDisk* | Number of shared disks | 4 |
| *MinDiskTime* | Minimum disk access time | 0.01 sec |
| *MaxDiskTime* | Maximum disk access time | 0.03 sec. |
| *PageSize* | Size of a page | 4096 bytes |
| *RecPerPage* | Number of records per page | 20 records |
| *DBSize* | Number of pages in database | 1024 (4 MB) |
| *BufSize* | Per-node buffer size | 1 MB |
| *FixedMsgInst* | Number of instr. per message | 20,000 instr. |
| *PerMsgInst* | Additional instr. per message | 10,000 per page |
| *CtrlMsgInst* | Size of a control message | 256 bytes |
| *LockInst* | No. of instr. per lock/unlock pair | 300 instr. |
| *PerIOInst* | Number of instr. per a disk I/O | 5000 instr. |
| **Transaction Parameters** | | |
| *CreationDelay* | Transaction inter-arrival delay | 0 second |
| *TrxSize* | No. of pages accessed by a trx. | 10 records |
| *TrxSizeDev* | Transaction size deviation | 0.1 |
| *PageLocality* | No. of records accessed per page | 1 - 4 |
| *WriteOpPct* | Probability of updating record | 0.02 - 0.5 |
| *TrxPerNode* | No. of trx. executed per node | 10 |

the settings that will be used for experiments. Many of the parameter values are adopted from [1].

We assume that GLM's CPU performs much better than each node's CPU. The reason is to prevent the GLM from being the performance bottleneck; hence, our experiment results could also be applied to the multi-GLM architecture. The number of shared disks are set to 4, and each disk has a FIFO queue of I/O requests. Disk access time is drawn from a uniform distribution between 10 milliseconds to 30 milliseconds.

The *network manager* is implemented as a FIFO server with 10 Mbps bandwidth. The CPU cost to send or to receive a message via the network is modeled as a fixed per-message instruction count plus an additional per-byte instruction increment. Note that the protocol processing (i.e., CPU overhead) dominates the on-the-wire time for messages in network.

The average number of pages accessed by a transaction is determined by a uniform distribution between *TranSize* $\pm$ *TranSize* $\times$ *TRSizeDev*. The *PageLocality* parameter specifies a range of values for the number of records to be accessed per page by a transaction. The parameter *WriteOpPct* represents the probability of updating a record. We will evaluate the performance of each coherency control scheme by changing *WriteOpPct* from 0.02 to 0.5.

The performance metrics used in the experiments are the *throughput rate* and the *response time* (turnaround time). The throughput rate is measured as the number of transactions that successfully commit per second. The response time in second is measured as the difference between when a transaction is submitted and when the transaction successfully commits. The time includes any time spent in the queue and time spent due to restarts. Since we are simulating a closed queuing system, the throughput rate is the inverse of the response time; hence, we do not explicitly show the response time.

A form of the batch mean method was used for the statistical analysis of simulation results. Each simulation was run for 30 batches until the number of committed transactions are 2000. The results of the first 200 transactions (10 percent) were discarded in order to account for initial transient conditions. Using this method, we were able to produce tight 90 percent confidence intervals for our simulation results.

## 5 Experiments and Results

In this section, our simulation model of the SDE is used to explore the relative performance of coherency control schemes. The compared schemes are write token scheme (WTS), cache check scheme (CCS), and extended cache check scheme (ECCS). To analyze the tradeoffs among the three schemes, their performance will be examined under a wide variety of database workloads. In the following subsections, we first describe the characteristics of each workload, and then discuss the experiment results on the workload.

### 5.1 HICON Workload

This workload models an application where all nodes have the *same* data access skew and the degree of data contention is very high as a result. For every transaction, most of the database accesses go to the hot set. Specifically, 80 percents of every transaction's accesses go to about 20 percents of database (200 pages). Note that analyzing the performance in this workload is very important, since the high contention application is one of the major applications of the SDE using fine-granularity locking.

Figure 2 shows the transaction throughput at this workload when the probability of updating a record (*write probability*) is varied. If the write probability is low, record updates are few and data contention is negligible. In this region, ECCS outperforms the other schemes significantly, because most of record locks can be handled locally and cached page locks are rarely de-escalated. Furthermore, the buffer hit ratio is higher in this region; hence disk I/O can also be reduced. The performance of CCS and WTS are almost identical in this region. There are not many page transfers to differentiate the schemes.

As the write probability is increased, throughput goes down because more updates bring more work (page transfers and I/Os) and also more data contention. ECCS worsens dramatically with write probability. Note that locks can be rarely cached in high

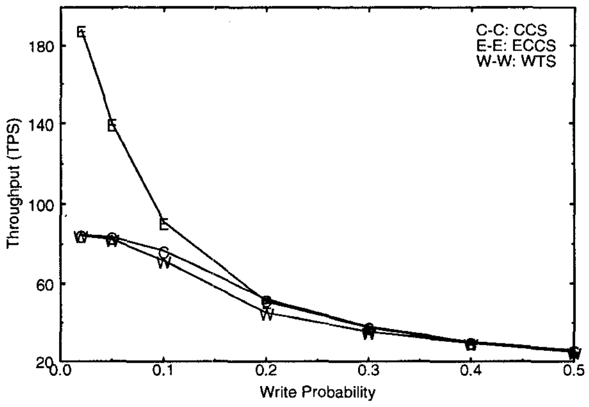

Figure 2: HICON - Throughput

contention workload if the write probability is high. Furthermore, the cached locks may be callbacked immediately due to frequent lock conflicts. As a result, at high write probability, ECCS works similar to CCS as Figure 2 shows. WTS suffers from frequent page transfers in medium write probability, while the performance of CCS downgrades slowly because it transfers a page only if the cached record is not recent. However, if the write probability is very high, the performance difference of WTS and CCS is disappeared. This is because both schemes run similarly for update operations and the number of update operations increase.

Figure 3 shows the transaction throughput at high contention workload for various number of nodes. The write probability was set to 0.1 and 0.2. As the number of nodes increase, both CCS and ECCS perform better than WTS. If there are large number of nodes, more transactions can be executed concurrently and the number of transactions accessing the same page is increased. As a result, the problem of unnecessary page transfer in WTS can be exacerbated. One interesting observation is that CCS outperforms ECCS at large number of nodes and high write probability. In this region, ECCS suffers from frequent callback requests. On the other hand, at low write probability, ECCS outperforms the other schemes significantly as the number of nodes increase until 10 because the cached locks can be rarely callbacked. After that, the performance of ECCS downgrades because large number of nodes bring more data contention.

### 5.2 HOTCOLD Workload

This workload models an environment where transactions referencing similar data are clustered together to be executed on the same node; hence, transactions running on different nodes should mainly access disjoint portions of the shared database. As a result, *node-specific locality of reference* can be exploited. If the transactions with high affinity to a given set of pages are executed in the same node, high local buffer

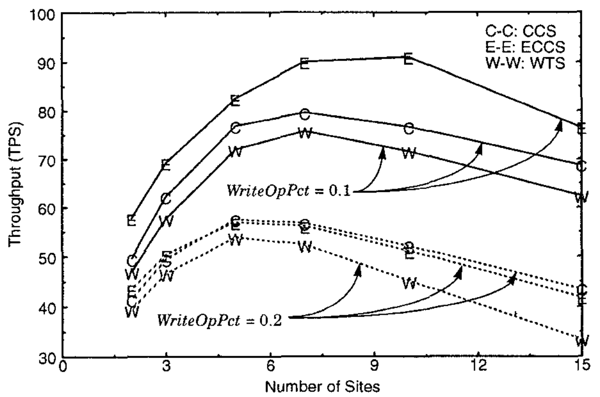

Figure 3: HICON - Varying the number of sites

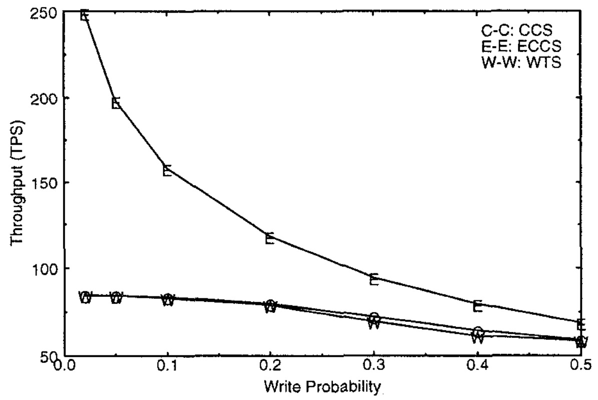

Figure 4: HOTCOLD - Throughput

hit ratio will be achieved and the message traffic due to page transfers will be reduced. To implement this workload, each node has an affinity for its own preferred region of the database, directing 70 percents of its accesses to that specific region and only 20 percents to the database as a whole. The remaining 10 percents go to the shared region, such as system catalogs, file directories, internal nodes of B-tree index, and so on. The shared region occupies about 10 percents of the database (100 pages), and the specific region allocated to each node is about 4.5 percents (45 pages).

Figure 4 shows the experiment results of this workload by varying the write probability while the number of nodes were set to 10. As expected, the performance of every scheme is improved dramatically due to reduced message traffic for page transfers and low lock conflict ratio. In particular, the performance of ECCS is improved more than the other schemes. This is because the node-specific locality of reference increases the effectiveness of lock caching and the cached locks are rarely callbacked due to low lock conflict ratio. The performance difference of CCS and ECCS is reduced compared to the HICON workload. This results from the following two reasons. First, the probability of accessing the same page by transactions in different nodes is reduced. Next, in CCS, the amount of disk I/Os may be relatively large compared to WTS. In WTS, if a node is not included in the copy set, the node should receive a recent version of the page before reading a record of the page. Even though this may result in heavy message traffic, the number of nodes in the copy set must increase. Furthermore, the transferred page is put on top of LRU buffer of the received node; hence, there are more chances for a node to find the recent version of the page from other nodes instead of the disks. On the other hand, CCS allows a node to read a cached record if the record was not updated by other nodes, thereby the number of nodes in the copy set must be relatively small. Therefore, if a node has to access the recent version of a page, CCS has larger probability for accessing disks than WTS.

Figure 5 shows the throughput results of CCS and WTS when the buffer size is changed. If the buffer size is small, CCS is outperformed by WTS because CCS suffers from frequent disk I/Os. As the buffer size is increased, the performance of CCS is improved more than WTS. This is due to the fact that the probability of a node to find recent pages from its buffer or other nodes' buffer is increased. In other words, the amount of disk I/Os can be reduced significantly in CCS by increasing the buffer size. Furthermore, note that CCS uses the buffer space more efficiently than WTS on read operations.

## 5.3 UNIFORM Workload

The last experiment was performed on UNIFORM workload, where all nodes access data uniformly throughout the entire database. For each transaction, 90 percents of its operations access the entire database except the shared region, and the remaining 10 percents go to the shared region of the database. Similar to the affinity clustering workload, the shared region occupies about 10 percents of the database.

Figure 6 shows the experiment results of uniform workload by varying the write probability. The number of nodes were set to 10 again. The performances of all schemes get worse considerably compared to the HOTCOLD workload. This is because of lack of locality resulting in the large amount of disk I/Os due to frequent page replacements and heavy message traffic for page transfers. As a result, each transaction holds its own locks relatively long, thereby the average locking delay must be increased, too. The performance of ECCS is nearly equal to that of CCS after the write probability is 0.1. The lock caching is not effective because the (new) cached locks are callbacked before they are used again locally. Furthermore the locks cannot be cached at high write probability. CCS behaves similar to WTS for every write probability. Remember that both CCS can outperform WTS if there are more chances for a node to be able to access its cached page, even though the page is not a recent version. However, in UNIFORM workload, a page that is not located in

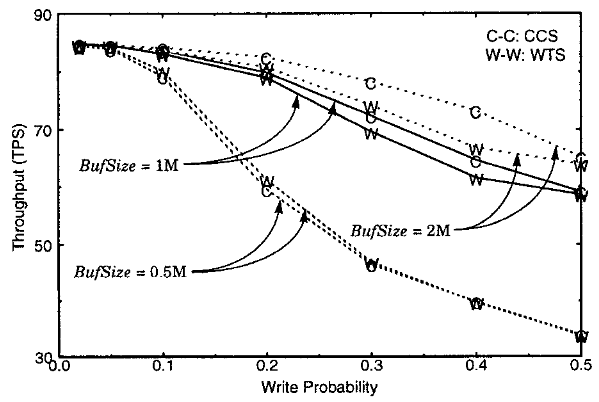

Figure 5: HOTCOLD - Effect of buffer size

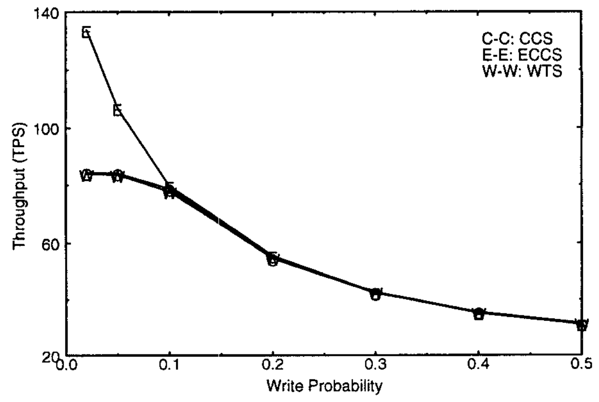

Figure 6: UNIFORM - Throughput

the shared region has often been replaced to a node's buffer before the node accesses the page again. This means that most of page accesses must lead to page transfers or shared disks accesses.

## 6 Conclusions

In this paper, we have proposed two cache coherency schemes, named *cache check scheme* (CCS) and *extended cache check scheme* (ECCS) in a shared disks environment (SDE) with fine granularity locking. Both schemes are based on the traditional write token scheme (WTS). While the fine-granularity locking is well matched to the requirements of the SDE such as high performance and high resource usage probability, it has been criticized for heavy message traffic due to locking overhead and frequent page transfers for maintaining the cache coherency. CCS can reduce the message traffic by receiving a page only if the node does not cache the recent version of a *record* to be accessed; in contrast, WTS makes a node be sent a page if the node does not cache the recent version of the *page*. ECCS further improves the performance by supporting the notion of lock caching and lock de-escalation.

We have explored the performance of CCS and ECCS under a wide variety of database workloads and system configurations using a distributed database simulation model. The basic results obtained from the experiments can be summarized as follows. First, for the range of workloads examined, ECCS stands out as the best at low write probability. Next, both CCS and ECCS exhibit substantial performance improvements over WTS for workloads with high locality of references, such as high contention workload and hot-cold workload. This result is very encouraging because (1) the SDE using record-level locking is usually expected to perform well in the high contention applications, and (2) one must strive to achieve node-specific locality of reference by exploiting the affinity-based transaction routing as far as possible. Last, any performance gains that can be achieved by CCS are offset due to the limited availability of each node's buffer.

## References

[1] M. Carey, M. Franklin and M. Zaharioudakis, Fine-Grained Sharing in a Page Server DBMS, in: *Proc. of ACM SIGMOD* (1994) 359-370.

[2] A. Dan and P. Yu, Performance Analysis of Buffer Coherency Policies in a Multisystem Data Sharing Environment, *IEEE Trans. on Parallel and Distributed Systems* 4(3) (1993) 289-305.

[3] A. Dan and P. Yu, Performance Analysis of Coherency Control Policies Through Lock Retention, in: *Proc. of ACM SIGMOD* (1992) 114-123.

[4] N. Kronenberg, M. Levy and D. Strecker, VAX clusters: A Closely Coupled Distributed System, *ACM Trans. on Computer Systems* 4(2) (1986) 130-146.

[5] A. Joshi, Adaptive Locking Strategies in a Multi-node Data Sharing Environment, in: *Proc. of 17th Int'l Conf. on VLDB* (1991) 181-191.

[6] D. Lomet, Private Locking and Distributed Cache Management, in: *Proc. of 3rd Int'l Conf. on Parallel and Distributed Information Systems* (1994) 151-159.

[7] C. Mohan and I. Narang, Recovery and Coherency Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment, in: *Proc. of 17th Int'l Conf. on VLDB* (1991) 193-207.

[8] C. Mohan, *et al.*, ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, *ACM Trans. on Database Systems* 17(1) (1992) 94-162.

[9] *Oracle 7 Parallel Server Concepts and Administration*, Oracle Corp. part A42522-1 (1996).

[10] E. Rahm, Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems, *ACM Trans. on Database Systems* 18(2) (1993) 333-377.

[11] H. Schwetman, *CSIM User's Guide for use with CSIM Revision 16* (MCC, 1992).

[12] K. Shoens *et al.*, The AMOEBA Project, in: *Proc. of IEEE CompCon* (1985) 102-105.

[13] M. Stonebraker, The Case for Shared Nothing, *IEEE Database Engineering Bulletin* 9(1) (1986).

[14] J. Strickland, P. Uhrowczik and V. Watts, IMS/VS: An Evolving System, *IBM Systems Journal* 21(4) (1982) 490-510.