Distributed Concurrency Control Based on Limited Wait-Depth

Peter A. Franaszek, *Fellow, IEEE,* Jayant R. Haritsa, *Member, IEEE,* John T. Robinson, and Alexander Thomasian

Abstract— **The performance of high-volume transaction processing systems for business applications is determined by the degree of contention for hardware resources as well as for data. Hardware resource requirements may be met cost-effectively with a data-partitioned or shared-nothing architecture. However, the two-phase locking (2PL) concurrency control method may restrict the performance of a shared-nothing system more severely than that of a centralized system due to increased lock holding times. Deadlock detection and resolution are an added complicating factor in shared-nothing systems. In this paper, we describe distributed Wait-Depth Limited (WDL) concurrency control (CC), a locking-based distributed CC method that limits the wait-depth of blocked transactions to one, thus preventing the occurrence of deadlocks. Several implementations of distributed WDL which vary in the number of messages and the amount of information available for decision making are discussed. The performance of a generic implementation of distributed WDL is compared with distributed 2PL (with general waiting policy) and the Wound-Wait CC method through a detailed simulation. It is shown that distributed WDL behaves similarly to 2PL for low lock contention levels, but for substantial lock contention levels (caused by higher degrees of transaction concurrency), distributed WDL outperforms the other methods to a significant degree.**

Index Terms— **Concurrency control, distributed algorithms, distributed databases, performance evaluation, simulation, two-phase locking.**

I. Introduction

HIGH-END transaction processing systems for business applications (such as banking, airline reservations, etc.) have stringent requirements for CPU processing power, I/O bandwidth, high availability, and cost effectiveness. Architectures for this purpose have evolved in three categories which are sometimes referred to as *Shared Everything (SE), Shared Disk (SD),* and *Shared Nothing (SN)* systems [16]. SN or *data partitioned* systems include distributed databases, but also have been used as a system design paradigm (e.g., Tandem multicomputers, Teradata's DBC/1012, and multicomputers with hypercube interconnection topologies). It has been argued that the SN paradigm is superior to the other two from the viewpoint of cost effectiveness, scalability, and availability.

Manuscript received May 14, 1991; revised June 8, 1992. A preliminary version of this paper was presented at the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 1992.

P. A. Franaszek, J. T. Robinson, and A. Thomasian are with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

J. R. Haritsa is with the Systems Research Center, University of Maryland, College Park, MD 20742.

IEEE Log Number 9213611.

Although SE and SD systems serve as the workhorse for today's high-end transaction processing systems, we are concerned with high-performance transaction processing in SN systems. This is because the increasing demand for higher transaction throughput from the viewpoint of processing power and I/O bandwidth can be met cost-effectively by SN systems. The performance of SN systems in a transaction processing environment is affected by the number of internode messages generated by transactions. The cost of sending and receiving messages tends to be nonneglible [9] and constitutes a significant CPU processing overhead that does not arise in centralized systems. On the other hand, there is the advantage of low cost per MIPS microprocessor technology, which makes SN systems attractive for processing high volumes of transactions [8] as well as data-intensive queries.

Two-phase locking (2PL) with the general waiting policy is the prevalent Concurrency Control (CC) method in commercial database systems.[1] It has been shown in numerous studies (see, e.g., [4], [17], [19]) that the performance of a system with 2PL may be constrained by data rather than hardware resource contention. In fact, SN systems are more susceptible to thrashing (degradation in system performance) than centralized systems because of the increased lock holding times due to internode communication and commit protocols [2], and possible delays in deadlock detection and resolution. Several ways to cope with this problem are as follows.

1) The use of different types of locks. Finer locking granularity (e.g., record versus page level locking), less restrictive locking modes (e.g., shared versus exclusive locks), semantics-based locks (increment and decrement locks which take advantage of commutativity), or specialized locks for indexes and other data structures [2] are some examples.

2) The use of CC methods other than 2PL. A large number of CC methods have been proposed [2]. It can be concluded from the studies reported in [4], [7] that in a system with high data contention, significant improvements in performance (compared to 2PL) are possible by utilizing data prefetching or judicious restarts of transactions (as described below). The adoption of these methods requires additional CPU processing capacity to tolerate the wasted processing due to transaction restarts.

A class of CC methods that take advantage of *access invariance* are described in [5]. Briefly, access invariance

[1] While all CC methods considered in this paper are based on two-phase locking, we use 2PL to refer to the standard locking policy where a transaction encountering a lock conflict is blocked and restarts are initiated only when there is a deadlock.

implies that a transaction will access the same set of database objects when it is executed at different times in a relatively short time interval. Such CC methods potentially have two execution phases, where the first phase does not involve CC and serves the purpose of prefetching data for the second execution phase, which uses some CC method, e.g., 2PL. With full access invariance, all of the data required for the second execution will be found in the database buffer (in main memory), and the execution time of this phase will be an order of magnitude shorter than the first phase. Therefore, the mean lock holding times in the second phase are an order of magnitude shorter than they would have been if locks were acquired in the first phase, which implies a significant reduction in lock contention. This effect is quantified in [20]. In fact, an appropriate CC method such as optimistic die [5] may be used during the first execution phase, in which case a successfully validated transaction may commit at the completion of its first phase. Given that the second phase is based on locking, lock preclaiming may be used to prevent deadlocks since the data required for execution are known at the end of the first execution phase. This hybrid CC scheme was shown to outperform 2PL in a high-performance SN system with data partitioning [18]. Due to the usual disadvantages associated with implementing optimistic methods [10], [13], this method will not be considered further in this work.

The Wait-Depth Limited (WDL) CC method for centralized databases (described in Section II-D) limits the wait-depth of blocked transactions, and is shown in [6], [7] to have superior performance with respect to 2PL, other locking methods such as running priority [4], and even optimistic CC methods (see, e.g., [2]) (when the hardware resources of the system are finite). In this paper, we propose an appropriate modification of WDL to SN systems, which has the twin goals of maintaining the main characteristics of WDL, while minimizing the number of additional messages that would be required for a straightforward implementation. For example, the difference of current time and the starting time of the current invocation of a transaction is used to indicate its progress instead of the number of locks held by the transaction in centralized systems [4], [6] (this progress information is used in deciding which transaction should be restarted to limit the wait-depth). In addition, schemes based on distributed decision making reduce the number of internode messages, but may incur more restarts than are absolutely necessary to limit the wait-depth to one. We also propose alternate implementations of distributed WDL that eliminate the possibility of multiple restarts at the cost of additional complexity and/or extra messages.

Simulation is used to compare the performance of distributed WDL with the distributed 2PL and the Wound-Wait methods [14]. 2PL was chosen because it is the protocol used in almost all transaction processing systems. Deadlock detection in distributed 2PL tends to be complex. Alternative deadlock resolution schemes are based on centralized and distributed combining of wait-for graphs or using timeouts (see Section II-B). An advantage of the Wound-Wait (WW) [14] and distributed WDL methods (with a wait-depth of one, as discussed later) with respect to 2PL is that they are *deadlock-free*. Furthermore, in the case of WW, the decision as to which transaction is to be restarted is done locally (at the node where the lock conflict occurs), without requiring additional messages. Our choice of CC methods covers the three main categories proposed in [4] of priority-less (2PL), strict priority (WW), and approximate essential blocking (distributed WDL).

A large number of papers have been written describing new distributed CC methods and comparing their performance through analysis or simulation. Some of the early work dealing with performance issues of distributed CC methods is surveyed in [15]. A more recent comparative study of CC methods and a survey of other works appears in [3]. A simulation study dealing with the effect of locking on the performance of an SN system is reported in [11].

The paper is organized as follows. Section II describes the distributed WDL method, and also includes a brief description of the 2PL and WW methods. Section III describes the model for the computer systems, the database, and the transaction characteristics considered in the simulation study. Simulation results are described in Section IV. Alternative implementations of distributed WDL are described in Section V. Conclusions appear in Section VI.

II. Distributed Concurrency Control Algorithms

Our study compares the performance of distributed WDL with respect to two well-known CC methods that, like distributed WDL, use locking as the underlying synchronization mechanism. The selected methods are the distributed 2PL [2] and the Wound-Wait (WW) method [14]. In this section, we first describe the general structure of a distributed transaction in our model. A brief description of the 2PL and WW CC methods is then presented, followed by a detailed description of distributed WDL.

A. Structure of Distributed Transactions

Each distributed global transaction consists of a *master* (or coordinator) process and a set of subtransactions (or cohort processes). The transaction runs at one of the nodes of the system, making database calls to the DBMS at the local (resp. remote) nodes to access local (resp. remote) data. Only sequential transaction execution with a single end-of-transaction commit point is considered here, for the sake of simplicity. We are not concerned with transactions involving user interactions, such as long-running transactions arising in computer-aided design applications, but rather with "short" transactions arising in business applications which have stringent response time requirements. For all the CC methods, the two-phase commit protocol [2] is used to ensure transaction atomicity.

The data distribution across the nodes of SN systems such as database machines is based on hashing applied to the primary key field in a relation. This generally implies a uniform distribution of accesses to the objects in the database. In the case of specific applications and transaction types, the data (e.g., relations or fragments of them) may be allocated so as to enhance locality of access. The latter allocation will be used in our study since it allows us to study the effect of locality

of access. Furthermore, we will assume that the data are not replicated, thereby requiring execution at a unique node for each data item that is accessed by a transaction.

When a new transaction arrives at one of the nodes of the system, it is assigned a timestamp. The timestamp is constructed by appending the node identifier of the parent node to the current system clock time at that node, thus ensuring that all transaction timestamps are unique. The global transaction's timestamp is also assigned to all of its cohorts.

B. The Two-Phase Locking (2PL) Concurrency Control Method

This is the commonly used CC paradigm in distributed databases. Transactions set locks directly at the primary execution node and indirectly through their subtransactions at other nodes. All locks are held until a transaction is either successfully committed or it is absorbed (strict 2PL) [2].

When there is a lock conflict, the transaction requesting the lock is blocked, and a request to this effect is posted in an FCFS queue. Since deadlocks are a possibility with 2PL, a centralized deadlock detection scheme may be adopted. Alternatively, deadlock detection may be carried out by the various nodes in a round-robin fashion [3]. Distributed deadlock detection methods will further reduce communication overhead for deadlock detection. Timeouts due to their simplicity are the most popular method for deadlock resolution in SN systems. It is noted in [11] that it is difficult to determine an appropriate timeout interval.

C. The Wound-Wait (WW) Concurrency Control Method

The WW CC method described in [14] is an effective method for preventing deadlocks in a distributed database based on local decisions made at the node where the lock conflict occurred. This is accomplished by utilizing priorities in resolving lock conflicts based on transaction timestamps. When an older transaction requests a lock on an object which is locked by a younger transaction in a conflicting mode, then the younger transaction is aborted. Younger transactions, however, are made to wait for older transactions when they request conflicting locks on data items held by older transactions. Deadlocks are eliminated since any cycle of waiting transactions would have to include at least one instance of an older transaction waiting for a younger transaction and such instances are prevented by this CC method.

An improvement in performance is achieved by ordering lock requests according to transaction timestamps (with older transactions being placed ahead of younger transactions in the queues for locks). When a transaction is restarted, it retains the timestamp that was associated when it first entered the system.

D. The Distributed WDL Concurrency Control Method

In this section, we first describe the centralized WDL method. In Section II-D2), we discuss the length function for distributed WDL, which is described in Section II-D3). This is followed by an illustrative example, a discussion of the issue of multiple transaction restarts to resolve the same lock conflict, and a high-level comparison of various methods.

1) Centralized WDL: The WDL(d) CC methods described in [6], [7] constitute a family of CC methods which restrict the wait-depth to d levels (only $d = 1$ is considered here), and in addition use a judicious victim selection policy to choose the transaction to be restarted such that wasted processing is minimized. Lock conflicts resulting in a violation of the wait-depth limit are resolved in WDL by comparing the progress made by the transactions involved in the lock conflict or their "length" (denoted by $L(T)$ for transaction T). The centralized WDL method can be specified succinctly by considering the wait-for trees associated with two active transactions T and T' as shown in Fig. 1a(a). Transaction T (resp. T') is blocking $n \geq 0$ (resp. $m \geq 0$) transactions. Next, T' makes a lock request and encounters a lock conflict with either transaction T or one of the n transactions blocked by it. The following rules cover all possible cases.

1) Case of Fig. 1a(b)

a) $m = 0$: T' waits (the wait tree is of depth 1).

b) $m > 0$: Restart T' unless $L(T') \geq L(T)$ and, for each $i, L(T') \geq L(T_i')$, in which case restart T.

2) Case of Fig. 1a(c):

a) $m = 0$: Restart T_1 unless $L(T_1) \geq L(T)$ and $L(T_1) \geq L(T')$, in which case restart T.

b) $m > 0$: Restart T' unless $L(T') \geq L(T_1)$ and, for each $i, L(T') \geq L(T_i')$, in which case restart T_1.

A pictorial representation of the WDL paradigm is given in Fig. 1(b).

What we just described is a specific instance of the wait-depth limited policies that were proposed in [6], [7]. Alternatively, it is possible to consider only two (rather than three or more) transactions at a time in resolving lock conflicts [21].[2] Referring to Fig. 1(a), when T' has a lock conflict with T (resp. T_1) as in Fig. 1a(b) [resp. Fig. 1a(c)], we first check if it is blocking other transactions (i.e., $m > 0$). If so, we restart T' if $L(T)$ is smaller than $L(T')$ [resp. $L(T_1)$]. Otherwise, the transaction holding the lock (T or T_1) is restarted. In the case that T' is not blocking other transactions, however, we check whether the transaction holding the lock is active or blocked [as in Fig. 1a(c)]. If it is active, no action is taken. If it is blocked, we restart T' if $L(T') < L(T_1)$; otherwise, T_1 is restarted.

For centralized database systems, this simplified scheme provides performance that is close to that of the original WDL method [21]. When implemented in a distributed environment, however, the message complexity (the number of messages required for conflict resolution as discussed in Section V-C) of the simplified method is identical to that of the original method. Given that the simplified method does not result in a reduction in messages complexity and its performance (in centralized systems) is inferior to the original WDL, we do not consider the simplified WDL in this study.

2) The Length Function in Distributed WDL: In the centralized case, it is convenient to define $L(T)$ as the number of locks held by T, and this has been shown to yield good performance [6], [7]. However, in the distributed case, given a particular subtransaction, determination of the total number

[2]The primary reason for considering this scheme in [21] was to simplify the analysis required for estimating transaction restart probabilities.

of locks held by all subtransactions of the global transaction would involve excessive communication, and in any case, the information could be obsolete by the time it was finally collected. Therefore, for distributed WDL, a length function based on time will be used, as follows. Each global transaction will be assigned a starting time (for its latest invocation if a transaction is restarted), and this starting time will be included in the startup message for each subtransaction, so that the starting time of a global transaction will be locally known at any node executing one of its subtransactions. Given a transaction, its length is defined as the difference of current time and the starting time of the global transaction.

We expect that transaction length defined in this fashion will be highly correlated with the total number of locks held by all subtransactions of a global transaction, and therefore will have similar performance characteristics when used as a WDL length function (note that subtransactions are executed sequentially in our model). This conjecture is verified by the simulation results in Section IV. In the case of centralized WDL, the cumulative number of locks requested by a transaction was also considered in [6], [7]. This assures a gain in transaction priority as the duration of its stay in the system increases, such that a transaction is not delayed in the system indefinitely due to restarts. It was observed, however, that this length function provides performance which is inferior to the one based on the number of locks obtained in the latest invocation. Furthermore, restart waiting (delaying the restart of an aborted transaction until its conflicting transactions are completed) makes the possibility of repeated restarts highly unlikely. In a distributed system, randomly generated delays before transaction restart are appropriate for this purpose when the conflicting transaction(s) are not local. Although distributed clock synchronization has been widely studied, extremely accurate clock synchronization is not required for our purposes since typical time-of-day clocks, correctly set to an external standard reference time, would suffice.

3) Distributed WDL: The following notation and conventions will be used in explaining the distributed WDL paradigm.

1) At any point in time, there is a set of global transactions $\{T_i\}$.

2) Each transaction T_i has an originating or primary node, denoted by $P(T_i)$, with starting time denoted by $t(T_i)$.

3) If T_i has a subtransaction at node k, this subtransaction is denoted by T_{ik}.

4) There are two CC subsystems at each node k, the LCC (local CC) which manages locks and wait relations for all subtransactions T_{ik} executing at node k, and the GCC (global CC) which manages all wait relations that include any transaction T_i with $P(T_i) = k$, and that makes global restart decisions for any of the transactions in this set of wait relations.

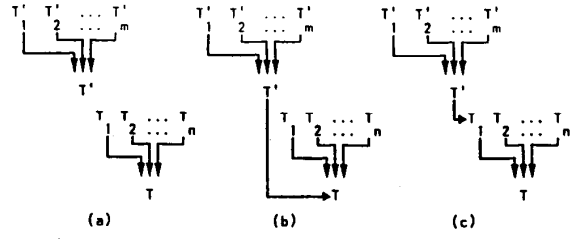
5) There is a *send* function that transparently sends messages between subsystems whether they are at the same or different nodes.

The general idea of the distributed WDL method is that: 1) whenever an LCC schedules a wait between two subtransactions, this information is sent via messages to the GCC's of the primary nodes of the corresponding global transactions, and 2) each GCC will asynchronously determine if transactions should be restarted, using its waiting and starting time information. Due to LCC's and GCC's operating asynchronously, conditions may temporarily arise in which the wait-depth of subtransactions is greater than one; however, such conditions will eventually be resolved either by a transaction committing or by being restarted by a GCC. The operation of the distributed WDL method will now be described in more detail.

In addition to the usual functions of granting lock requests, scheduling subtransaction waits, and releasing locks as part of subtransaction commit or abort processing, each LCC does the following: whenever a wait $T_{ik} \rightarrow T_{jk}$ is scheduled, the message $(T_i \rightarrow T_j, P(T_j), t(T_j))$ is sent to the GCC at node $P(T_i)$, and the message $(P(T_i), t(T_i), T_i \rightarrow T_j)$ is sent to the GCC at node $P(T_j)$, unless $P(T_i) = P(T_j)$, in which case only one message $(T_i \rightarrow T_j)$ is sent to the GCC at node $P(T_i)\,(= P(T_j))$.

Each GCC dynamically maintains a wait graph of global transactions which is updated using the messages it receives from LCC's of the form just described. Note that starting time and primary node information is included in these messages for those transactions that have a primary node different from that of the node to which the message was sent, so that each GCC has starting time and primary node information available for all transactions in its wait graph. Each GCC analyzes this wait information, either periodically or every time a message is received, and using the WDL method, determines whether transactions should be restarted. Periodic checking has the potential of combining messages associated with multiple transactions together, such that the number of internode messages is reduced. A similar effect can be achieved by *bundling* several messages into one before transmission. While reducing communication overhead, both methods have the disadvantage of increasing the chances of the wait-depth criterion being temporarily violated.

Whenever it is decided that a transaction T_i should be restarted, a restart message for T_i is sent to node $P(T_i)$. However, no wait relations are modified by the GCC at this time (since T_i could currently be in a commit or abort phase); instead, the status of T_i is marked as pending. Actual commit or abort (followed by restart) of a transaction T_i is handled by the transaction coordinator at node $P(T_i)$. Commit is initiated upon receiving successful completion messages from all subtransactions; abort is initiated upon receiving a restart message from some GCC (or also possibly due to receiving an abort message from some other transaction coordinator at a subtransaction node, for example, due to a disk error). The commit or abort is handled by communicating with the transaction coordinators and LCC's at each subtransaction node using known techniques (the two-phase commit protocol [2]). Additionally, it is necessary to send the appropriate information to each GCC that is currently maintaining wait information for T_i. This can be determined locally using the wait information maintained by the GCC at node $P(T_i)$: the GCC's for the primary nodes of the transactions that are waiting on T_i or on which T_i is waiting must be notified. Each such GCC removes T_i from its wait graph and acknowledges. In the case of transaction restart, restart can be initiated after

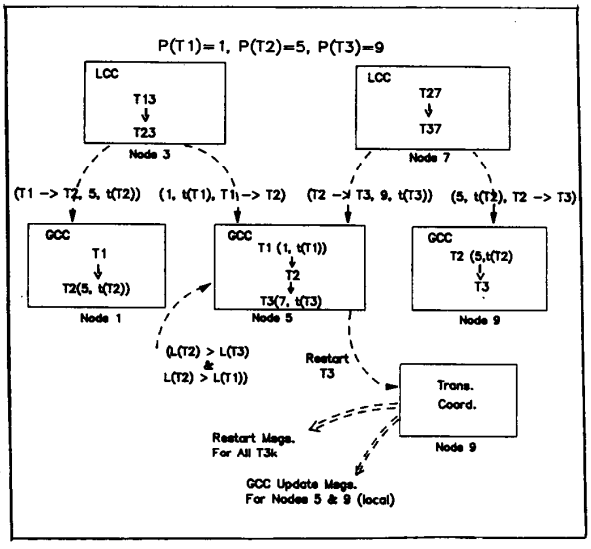


(a)

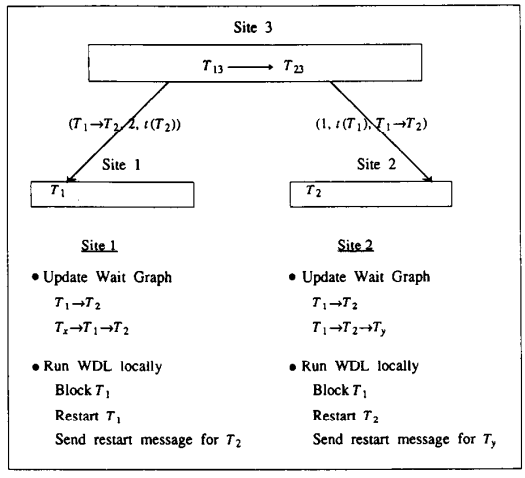
Requesting Transaction: T_R
Holding Transaction: T_H

	WAIT GRAPH	ACTION
(1)	$T_R \longrightarrow T_H$	block T_R
(2)	$T_X \longrightarrow T_R \longrightarrow T_H$	if $L(T_R) > Max(L(T_H), L(T_X))$ restart T_H else restart T_R
(3)	$T_R \longrightarrow T_H \longrightarrow T_Y$	if $L(T_H) > Max(L(T_R), L(T_Y))$ restart T_Y else restart T_H
(4)	$T_X \longrightarrow T_R \longrightarrow T_H \longrightarrow T_Y$	same as for (2)

(b)



(c)



(d)

Site 1	Site 2	
$T_x \rightarrow T_1 \rightarrow T_2$	$T_1 \rightarrow T_2 \rightarrow T_y$	
Restart	Restart	
T_1	T_2	Two restarts
T_1	T_y	
T_2	T_y	
T_2	T_2	

(e)

Fig. 1. (a) Initial and temporary states for WDL. (b) Operation of the WDL method. (c) Simple example of distributed WDL method. (d) Basic operations in distributed WDL. (e) Multiple restart problem in WDL.

receiving acknowledgment from all subtransaction nodes and each such GCC.

The above can be illustrated by a simple example, as illustrated in Fig. 1(c). As shown, there are three transactions T_1, T_2, T_3, with primary nodes 1, 5, and 9, and with various subtransactions possibly scattered around the system. Only those subtransactions that enter a wait relation are indicated in the figure.

1) At node 3, T_{13} requests a lock held in an incompatible mode by T_{23}, the LCC schedules $(T_{13} \rightarrow T_{23})$, and messages are sent as shown to the GCC's at nodes $P(T_1)$ and $P(T_2)$.

2) Concurrently, at node 7, T_{27} requests a lock held in an incompatible mode by T_{37}, the LCC schedules $(T_{27} \rightarrow T_{37})$, and messages are sent as shown to the GCC's at nodes $P(T_2)$ and $P(T_3)$.

3) At some later time, these various messages are received and wait graphs are updated by the GCC's at nodes 1, 5, and 9. After both messages for the GCC at node 5 are received, there is a wait chain of depth 2, as shown in the figure.

4) The GCC at node 5 determines, using local current time and the recorded starting time for each transaction (since $P(T_2) = 5$, its starting time is available locally), that $L(T_2) > L(T_3)$ and $L(T_2) > L(T_1)$. Therefore, following the WDL CC method, it decides to restart T_3, and sends a restart message to the transaction coordinator at node $P(T_3) = 9$.

5) The transaction coordinator at node 9 receives the restart message and begins transaction restart by sending restart messages for all nodes executing a subtransaction T_{3k} and GCC update messages to the local GCC and the one at node 5.

Note that, in practice, situations could develop that would be far more complex than that of this simple example: due to GCC's operating independently and asynchronously, decisions could be made concurrently by two or more GCC's to restart different transactions in the same wait chain, a situation that would not occur in the centralized case. The case when this situation arises is illustrated in Fig. 1(d) and (e). Nodes 1 and 2 receives messages from Node 3 about the conflict between transactions T_1 and T_2, and incorporating this new conflict information results in the wait-for graphs shown in Fig. 1(d). In this scenario, as per the basic WDL method [see Fig. 1(c)], Node 1 will decide to either restart T_1 or send a restart message for T_2 to Node 2. At the same time, Node 2 will decide to either restart T_2 or send a restart message for T_y to its parent node. The important point to note is that for three of the four possible restart combinations, two transactions are restarted. If we consider the conflict from a global perspective, however, we see that the resultant wait chain is identical to Case (4) in Fig. 1(b), and that only *one* of T_1 or T_2 need have been restarted to satisfy the limit on wait-depth. Since CC performance is usually dominated by the way in which simple cases are handled, we expect the distributed WDL method described here to have a performance characteristic similar to the centralized WDL method.

To summarize, in WW, all conflicts are decided locally at the conflict node, and conflict information does not have to be transmitted to other nodes. In 2PL also, all conflicts are decided locally, with the only overhead being the periodic transmission of wait graphs for deadlock detection. For WDL, however, each conflict could result in as many as two messages having to be transmitted (if the parent nodes of the conflicting transactions and the conflict node are all different). Additional messages may be required if the transaction to be restarted has a different primary execution node from the primary node of the conflicting transactions (for example, when the primary node for T_y in Fig. 1(d) is different from Node 2). This may cause increased communication costs and delays in resolving data conflicts. Also, since nodes possess only *parts* of the global wait chain, decisions could be made concurrently by two or more nodes to restart different transactions in the same wait chain. Therefore, there may be more restarts of transactions than would strictly be necessary to satisfy the limit on the depth of wait chains. Alternative distributed WDL implementations that alleviate the drawbacks of the *Basic* distributed WDL method are described in Section V.

III. The Distributed Database Model

A detailed simulation model of a distributed DBMS was developed for studying the performance behavior of the distributed 2PL, WW, and WDL CC methods. The general structure of the model is shown in Fig. 2(a). In this model, the database is partitioned among a number of nodes, each of which has a complete local DBMS. The nodes communicate with each other using messages transmitted on an interconnection network. The database itself is modeled as a collection of pages. A transaction consists of a sequence of data accesses, which involves a lock request, accessing the data item, followed by a period of CPU processing.

The model of the local database system as shown in Fig. 2(b) consists of seven components: an *application manager* that generates transactions; an *execution manager* that translates each transaction into a set of calls to the DMS (data management system) and also models transaction initiation, commit, and abort; a *DMS manager* that models the data management services for the database; a *buffer manager* that models the buffer allocation and replacement policies; a *concurrency control (CC) manager* that implements the details of the CC methods; a *recovery manager* that controls the logging process; and lastly, a *resource manager* that models the CPU and disk resources and services the hardware requests of all the other modules. In addition to these per-node components, the model also has a *network manager* that models the behavior of the underlying communication subnetwork and interfaces with the resource manager module at each node. The following subsections describe the details of the hardware resource configuration, the database access pattern, and the transaction workload generation process.

A. The Computer System Model

The system model and the settings for the simulation parameters are as follows:

1) Multisystem Configuration: There are $N = 4$ computer systems, consisting of tightly coupled multiprocessors with $P = 4$ processors per system. The total processing capacity per system is varied to study its effect on the relative performance

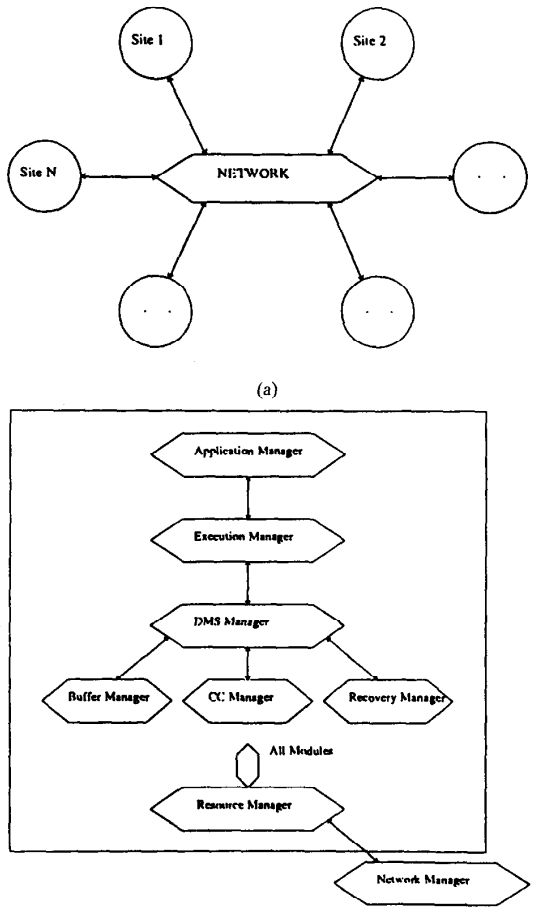


Fig. 2. (a) Distributed DBMS model structure. (b) Single site model structure.

of CC methods. Since it is expected that processor speeds will increase over the next decade, we will study the effect of increased processing capacity on the maximum throughputs achievable by different methods.

2) Intersystem Communication: A high bandwidth interconnection network which introduces negligible delay interconnects the computer systems. We take into account, however, the CPU overhead to send and receive messages (similar assumptions are made in [3], among others). Transaction execution requests and messages for data access have the same priority at the CPU. Messages related to CC, however, are assigned higher priority to facilitate speedy conflict resolution and reduce lock holding times.

3) I/O Subsystem: The disk service time including any queueing delays is assumed to be fixed and equal to 20 ms in the simulator.

4) Database Cache: A database cache with an LRU policy for caching local data is available at each node. High contention items (see Section III-B) tend to be always in the cache, $F_{DB\_high} \simeq 1.00$, while the hit ratio for low contention items is $F_{DB\_low} = 0.50$. The cache is large enough that data referenced by in-progress transactions are not replaced before they are completed.

5) Logging and Recovery: Nonvolatile (random access) storage is considered for logging, thereby circumventing the need for synchronous disk I/O. Logging time is therefore an order of magnitude smaller than the time required to write onto disk and it is ignored in the simulator. Note that this results in reducing lock holding time for all of the CC methods.

B. The Database Access Model

The database model considered in this study is described below.

1) Database Objects: Data items (e.g., disk pages) constitute the unit of locking. We distinguish high and low contention data items based on their access frequency by transactions. At each system, there are $D_{\text{high}} = 256$ (resp. $D_{\text{low}} = 7936$) data items in the high (resp. low) contention category. A fraction $F_{\text{high}} = 0.25$ (resp. $F_{\text{low}} = 0.75$) of all transaction accesses are uniformly to high (resp. low) contention items. Therefore, the level of data contention is determined by the high contention data items since they are accessed roughly ten times more frequently than low contention items.[3] As could be expected, when D_{high} is large, resulting in low levels of data contention, all CC methods provide the same performance. A small value of D_{high} is modeled in the experiments described here in order to highlight differences in the performance of the methods.

The overall cache hit ratio for a transaction executing for the first time (i.e., not a restarted transaction) is $P_{\text{hit}} = F_{DB\_low} \times F_{\text{low}} + F_{DB\_high} \times F_{\text{high}} = 0.625$ (typical of some high-end transaction processing systems). This hit ratio also applies to data accesses at remote nodes.

2) Access Mode: All data items are accessed in exclusive mode since we are interested in the relative performance of the CC methods. Shared accesses would have resulted in a reduction in the data contention level, but this would require an appropriate choice of the fraction of shared lock requests and more complicated conflict resolution, especially in the case of WDL.

3) Deadlock Detection: Deadlock detection is required only for 2PL since WW and WDL prevent deadlocks. In our simulation implementation, the deadlock detection is immediate, that is, a deadlock is detected as soon as a lock conflict occurs and a cycle is formed. Also, the overhead for detecting deadlocks is set to zero. These simplifications are justifiable because the frequency of deadlocks tends to be negligibly small, at least for the locking modes considered here [20].[4] The choice of a victim in resolving a deadlock is made based on transaction timestamps: the youngest transaction in the cycle is restarted to resolve the deadlock. When a transaction is restarted, it retains the timestamp that it was assigned when it first entered the system. Deadlock detection and resolution are handled in this fashion in order to

[3] A restarted transaction makes the same sequence of data accesses as the original transaction, that is, there is no resampling of data items.

[4] Most deadlocks are attributable to the conversion of shared to exclusive locks, and can be prevented by introducing update locks.

observe how WW and WDL compare with the "best" possible performance of 2PL.

C. The Transaction Processing Model

The construction and characteristics of the transaction workload are described below.

1) Transaction "Arrivals": We consider a closed system with M transactions in each system (and $N \times M$ transactions in the complex), i.e., a completed transaction is immediately replaced by a new transaction at the same system. This implies that we have a system with a fixed number of users and zero "think times." The parameter M is varied to study the effect of transaction concurrency on performance.[5]

2) Transaction Classes: There are multiple transaction classes based on transaction size, that is, the number of data items (n_c) accessed by a transaction in class c. Transactions are introduced into the system with frequencies $f_c, c = 1, \cdots, C$ according to what might be expected in a stream of arriving transactions. Transaction sizes are 4, 8, 16, and 32 with associated frequencies (0.20, 0.20, 0.35, 0.25). This geometric progression of transactions sizes yields a high variability in transaction size, while using only a small number of transaction classes. In addition to this four-class distribution, we also experimented with two other distributions: uniform and fixed. For the uniform distribution, transactions sizes are uniformly distributed between 8 and 24 (inclusive), while for the fixed distribution, all transactions are of size 16. These settings ensure that the *mean* transaction size of all the distributions is 16.

The advantage of using the frequency-based model instead of modeling a fixed number of transactions in each class is twofold: 1) we have the assurance that various CC methods process the same mix of transactions, and 2) the overall throughput can be used to compare the relative performance of CC methods.

3) Transaction Processing Stages:

Transaction Initialization: This requires CPU processing only, and the pathlength for this stage is $I_{init1} = 100\,000$ instructions. If a transaction is restarted due to any reason, $I_{init2} = 50\,000$ instructions are executed as part of its initialization phase.

Data Processing: There are n steps in this stage, where n is the number of data items accessed by the transaction (from local or remote partitions). Each transaction is routed to the system at which it exhibits a high degree of locality. The fraction of local accesses at each system is F_{local}, while the remaining $1 - F_{\text{local}}$ accesses are uniformly distributed over the remaining systems.

A data item may be available in the database cache, in which case the pathlength per data item is $I_{\text{cache}} = 20\,000$. This includes the overhead for CC. Otherwise, when data have to be accessed from disk, an additional $I_{\text{disk}} = 5000$ instructions are required (the processing required to retrieve cached data is considered to be negligible). In addition, it takes $I_{\text{send}} = 5000$ instructions to send or receive a message. Therefore, 20 000 instructions are executed for intersystem communication when the data are not available locally.

Transaction Completion: The CPU processing in this stage requires $I_{\text{complete}} = 50\,000$ instructions. In case a transaction has accessed local data only, it may commit at this point without requiring a two-phase commit protocol. Commit processing requires $I_{\text{commit}} = 5000$ instructions to force a log record onto stable storage.

If multiple systems are involved in processing a transaction as part of two-phase commit, $I_{\text{pre-commit}} = 5000$ instructions are executed at the primary node of transaction execution (mainly to write a pre-commit log record). There is also a per-system overhead of I_{send} and I_{receive} to send and receive PRECOMMIT messages. Pre-commit processing at secondary nodes from which data were accessed requires $I_{\text{remote}} = 5000$ instructions, which includes writing pre-commit records. Each remote system, after forcing modified data onto stable storage, sends an ACK message to the primary system, which in turn sends a COMMIT message to all of the nodes involved after forcing a commit record onto the log. On receiving this message, each system releases all locks that are held locally by the committing transaction.

Since WDL and WW are restart-oriented policies, provisions are made to reduce the overhead of restarting a transaction. We postulate that a no-steal policy is followed by the transaction processing system [2], and that the *undo records* are held in main storage while the transaction is active. It follows that transaction restarts can be performed without requiring disk I/O. We incur $I_{\text{restart}} = 5000$ instructions at the primary node of transaction execution and the sites of subtransaction execution to account for the overhead of releasing locks and undoing updates to modified pages.

IV. Simulation Results

In this section, we present performance results for the distributed 2PL, WW, and WDL CC methods obtained from a simulator written in DeNet [12]. The performance metric employed in comparing the CC methods is the overall system throughput across all N nodes as a function of the aggregate system Multiprogramming Level (MPL). The mean transaction response time follows easily from Little's law. In particular, we are interested in the *peak* throughput that is achievable by each of the CC methods as it determines the limit on system performance due to contention for data and resources. Due to the symmetric nature of the workload and the database system, the mean throughput at each node is the same and is $1/N$ of the overall throughput. Furthermore, due to conservation of flow, the throughput of each class of transactions is proportional to its fractional contribution in the input workload. Each simulation was run until steady-state behavior (whenever available, which excludes the thrashing region for 2PL). The batch means method was used to obtain relative half-widths of 5% about the mean throughput at 90% confidence level. The simulations also generated a host of other statistical information, including resource utilization, the *restart ratio*, defined as the ratio of the number of transaction

[5] It follows from Little's law that a nonzero think time simply has the effect of reducing the system throughput. A zero think time (and otherwise no constraints on the degree of concurrency) tends to reduce simulation cost by removing variability in the number of concurrent transactions.

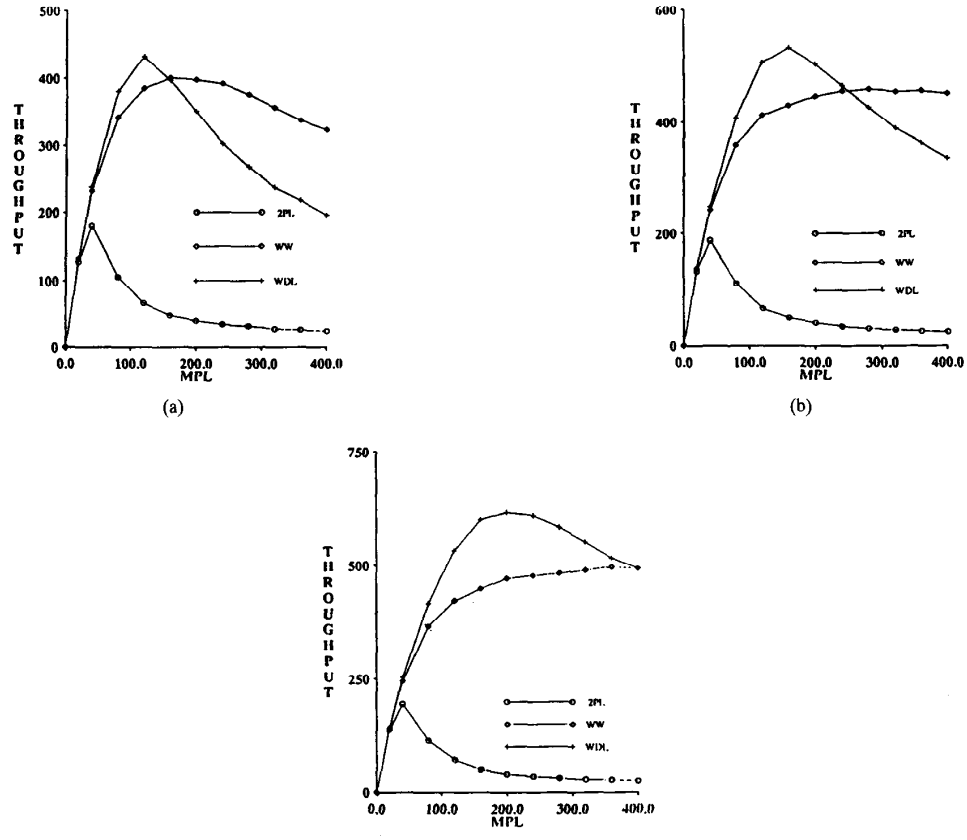


Fig. 3. Throughput. (a) 50 MIPS/CPU. (b) 100 MIPS/CPU. (c) 200 MIPS/CPU.

restarts and the number of transactions completed, mean transaction blocking time, etc. These secondary measures help in explaining the behavior of the CC methods under various loading conditions, but are reported here to a limited extent due to space limitations. The experiments investigated the effects of variations in system processing capacity, data locality, message costs, and transaction size distribution.

A. The Effect of Increased Processing Capacity

Our first set of experiments profiled the performance of the CC methods as a function of system processing capacity. The experiments were conducted for varying processor speeds, using the four-class distribution and keeping all other parameters at the levels specified in Section III. Fig. 3(a)–(c) present the transaction throughputs obtained under each CC method for per-processor speeds of 50, 100, and 200 MIPS, respectively (note that there are $P = 4$ processors per node and $N = 4$ nodes). In Fig. 4(a)–(c), the corresponding restart ratios for each of these experiments are shown. This metric helps to analyze how heavily a CC method is biased towards using either restarts or blocking as the method of conflict resolution. Note that the number of transaction restarts is not an adequate indicator of wasted processing. Therefore, Fig. 5(a)–(c) present the *processor utilization* characteristics for this set of experiments. In these utilization figures, three curves are shown for each CC method. First, the *total* utilization (solid line) indicates the actual processor utilization generated by the CC method; second, the *useful* utilization (dashed line) plots the resource usage made by those transaction executions that resulted in completion (i.e., they exclude the resources spent on work that was later undone by restarts); and, finally, the *message* utilization (dotted line) plots the fraction of the total resource utilization that is spent in the processing of messages. This breakup of processor utilization helps to identify the source of performance limitations and the overheads associated with each CC method.

The throughput results [Fig. 3(a)–(c)] indicate that the throughput for each CC method initially increases as the system MPL is raised, but peaks after the MPL is raised sufficiently high and decreases for MPL's beyond this point (the reader is reminded that this is so for a high lock contention environment, and for very low contention levels, all methods provide an effective throughput which follows the throughput characteristic). These trends are similar to those seen in centralized DBMS [1], [7], [6] and can be

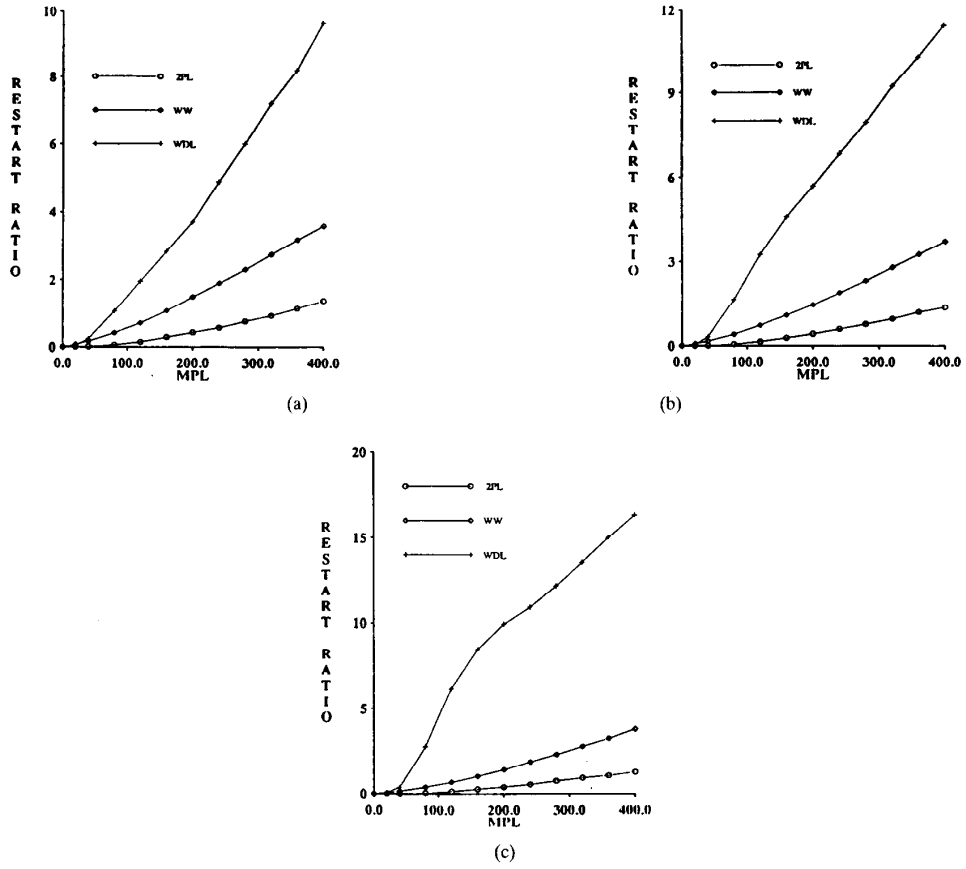


Fig. 4. Restart ratio. (a) 50 MIPS/CPU. (b) 100 MIPS/CPU. (c) 200 MIPS/CPU.

explained as follows: the initial increase is due to the fact that better performance is obtained through increased degree of concurrent transaction processing. There is little hardware resource and lock contention initially, but the contention increases with increased MPL. Provided that there was no lock contention, the system throughput increases with MPL, and ultimately attains asymptotic behavior beyond the point at which the bottleneck resource (processors) saturates (this is at least true for the simplified computer system model considered in most simulation studies, e.g., we assume there is adequate database buffer space). When lock contention is taken into account, system throughput may actually decrease with increased MPL. There are two factors contributing to this phenomenon. In the case of 2PL, where transaction restarts are rare [20] and wasted processing is negligible, the degradation in performance (reduction in throughput in a closed system) is due to the fact that the mean number of active transactions may actually decrease as the number of transactions activated in the system is increased [4], [17], [19]. In the case of a CC method, which uses restarts, the throughput in a closed system increases up to the point where the bottleneck resource (the CPU) saturates, but beyond this point, the system throughput decreases due to unnecessary restarts. This phenomenon can be prevented by limiting the number of transactions activated in the system or by using *restart waiting* [6], [7], i.e., a transaction which was aborted due to a lock conflict with one or more transactions is delayed until the conflicting transactions are completed.

Considering the performance of the CC methods individually, we observe that the peak throughput attained by 2PL is considerably smaller than that of the other CC methods.[6] At low MPL's, since few transactions are blocked and there is little wasted work due to deadlock-resolution restarts, 2PL behaves as well as the other CC methods. As the system MPL is increased, however, the number of blocked transactions in the system increases steeply, causing the throughput to level off. For MPL's beyond this peak throughput, a sharp fall in transaction throughput is seen and constitutes the *thrashing region* for 2PL (see, e.g., [19]). An important point to observe here is that the performance of 2PL shows only negligible improvement with an increase in processor speeds (e.g., compare Fig. 3(a) and 3(c)). The reason that 2PL is unable to take advantage of increased resource capacity is that its conflict resolution mechanism results in most of the

[6]In the case of low MIPS systems (and high MPL), 2PL may slightly outperform WDL (peak 2PL throughput may exceed peak WDL throughput by a few percent [7]) because of the wasted processing that is incurred by WDL.

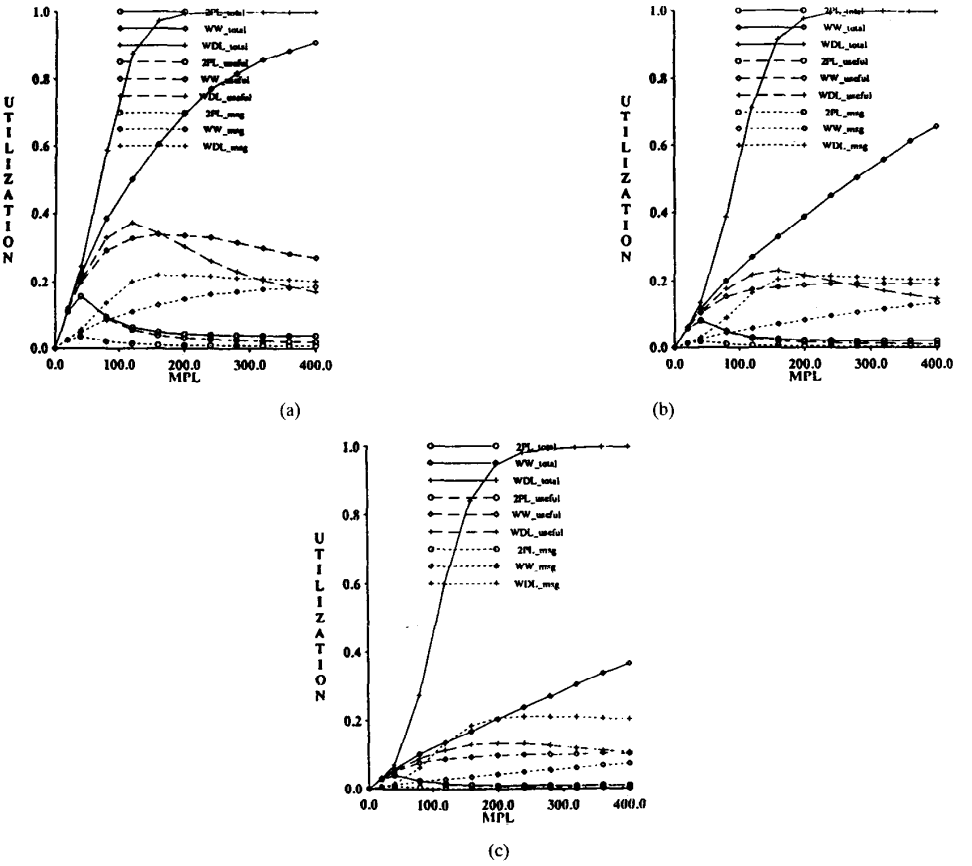


Fig. 5. Utilization. (a) 50 MIPS/CPU. (b) 100 MIPS/CPU. (c) 200 MIPS/CPU.

transactions being blocked under high lock contention [4], [17], [1], [7], [6], [19]. In this situation, it is not possible to gain higher concurrency by just adding resources to the system since there are no transactions available to make use of the additional capacity when the level of lock contention is high. This explanation is confirmed by looking at the CPU utilization graphs for 2PL in Fig. 5(a)–(c), which show that the total utilization of 2PL decreases as the processor speeds are increased, thus resulting in maintaining essentially the same throughput characteristic. Since restarts in 2PL are caused only when deadlocks occur, its restart ratio numbers [Fig. 4(a)–(c)] are significantly smaller than those of the other CC methods.

Turning our attention now to WW, we observe that it delivers a peak throughput intermediate to that of WDL and 2PL. Due to the significant restart component of its conflict resolution policy, which allows for higher levels of concurrent transaction execution, it is able to increase its use of system resources when the MPL is raised. In addition, its peak throughput performance improves, to a limited extent, with an increase in processing capacity. Once the processing capacity reaches sufficiently high values, however, the peak throughput of WW remains virtually the same and is unaffected by the availability of faster resources. An interesting characteristic of the throughput profile of WW is that its degradation for MPL's beyond the peak is very gradual and occurs at a much smaller rate than those of the other CC methods. The reason for the observed behavior is that when the lock contention is the primary performance limiting factor, the maximum number of concurrent transactions in WW asymptotically reaches $1/p$, where p is the pairwise probability of conflict among transactions [4]. Therefore, only for the case of processor speed being 50 MIPS, which causes the resources to be heavily utilized [see Fig. 5(a)] do we see a fall in throughput at MPL's beyond that of the peak throughput. For faster processor speeds [Fig. 5(b)–(c)], lock contention is the main performance limiting factor, and the throughput characteristic of WW flattens out at high MPL's.

Finally, with regard to WDL, we observe that it delivers a peak throughput greater than the other CC methods for the set of processor speeds considered in these experiments. More importantly, the performance of WDL *improves* with increased processor speeds, which means that unlike 2PL and WW, WDL is capable of utilizing additional resource capacity to achieve high throughputs. Therefore, as the processing capacity of the system is increased, WDL performs increasingly better than the other two CC methods. From Fig. 4(a)–(c)

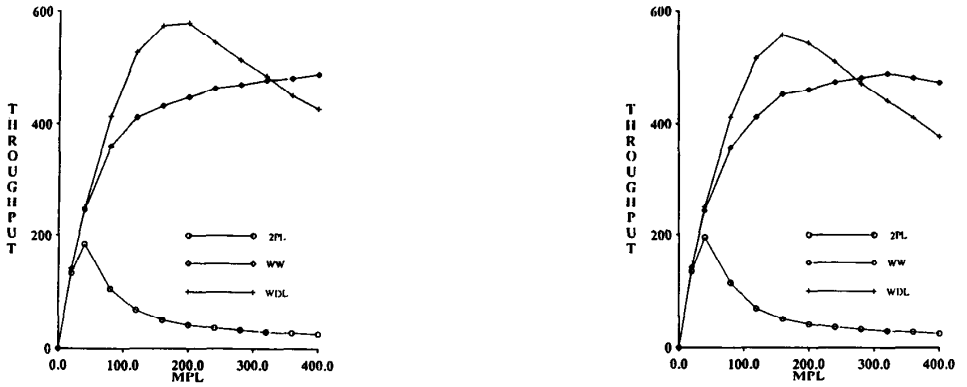


Fig. 6. (a) Locality 0.25 (200 MIPS/CPU). (b) MsgCost = 20 000 (200 MIPS/CPU).

and 5(a)–(c), it can be observed that WDL has significantly higher processor utilization and restart ratio characteristics than the other CC methods. The reason for this behavior is that WDL attempts to approximate the *essential blocking* property [4] by ensuring that wait-chains never involve more than two transactions. At high MPL's, when the number of lock conflicts is extremely high, this wait-chain limiting policy results in a high restart rate. The increased restart rate also means that fewer transactions are blocked, thereby resulting in an increase in the number of active transactions and higher resource utilization. Due to this ability of distributed WDL to fully utilize the resources, its performance noticeably degrades beyond the peak throughput since increases in MPL after this point result in a significant increase in *both* data and resource contention. An important point to note here is that, as observed in Section II-D, some of the restarts of WDL are unnecessary and are caused by the distributed nature of the conflict resolution algorithm. Elimination of such restarts may help further improve the performance of distributed WDL. Alternative conflict resolution protocols that eliminate these unnecessary restarts (at the cost of extra delay and increased number of messages) are described in Section V.

B. Message Costs and Locality

In previous studies of distributed database systems (e.g., [3]), it has been observed that system performance may be quite sensitive to data locality and message costs. Therefore, in our second set of experiments, we investigated the performance effects of having either similar degrees of data locality or higher message costs than those used in the baseline set of experiments.

The first experiment investigated the performance of the CC methods when the locality is reduced from 0.75 to 0.25 (uniform distribution across four nodes), while the second experiment increased the CPU cost per message from 5000 instructions to an artificially high value of 20 000 instructions (factor of four). Both these experiments were conducted for all the processor speeds of the first experiment, but due to space limitations, only the graphs obtained for a processor speed of 200 MIPS are shown here in Fig. 6(a) and (b). Comparing these figures with Fig. 3(c), we observe that although the absolute performance of all the CC methods is adversely affected, the relative behavior of the CC methods does not change significantly.

It should be noted, however, that the performance of distributed WDL is impacted more severely than those of the other CC methods. This is due to the fact that its message complexity is greater than that of the other schemes since its conflict resolution mechanism involves sending messages to the parent nodes of the conflicting transactions every time a conflict occurs. Therefore, with either decreased locality or increased message cost, the performance of WDL is more seriously affected. This is also confirmed by comparing the message utilization of WDL as compared to WW and 2PL in Fig. 5(a)–5(c). Since our primary aim, however, is to maximize peak system throughput, and as we are willing to devote resources towards this end, WDL appears to be the CC method of choice since it outperforms 2PL or WW if the resource capacity is sufficiently large. This feature was observed in all of our experiments.

C. Transaction Size Distribution

In our final set of experiments, we investigated the effect of having a transaction distribution different from the four-class distribution used in the baseline set of experiments. Experiments were conducted with both the uniform distribution and the fixed distribution. Fig. 7(a) and 7(b) present the results of these experiments for a processor speed of 200 MIPS, with all other parameters being at the levels specified in Section III. From these figures, we observe again that while the absolute throughputs of the CC methods are affected by the size distribution, their relative behavior remains qualitatively the same as that seen in the baseline experiments.

From the three sets of experiments described above, we conclude that for systems having sufficient processing capacity, WDL delivers a significantly higher peak throughput than

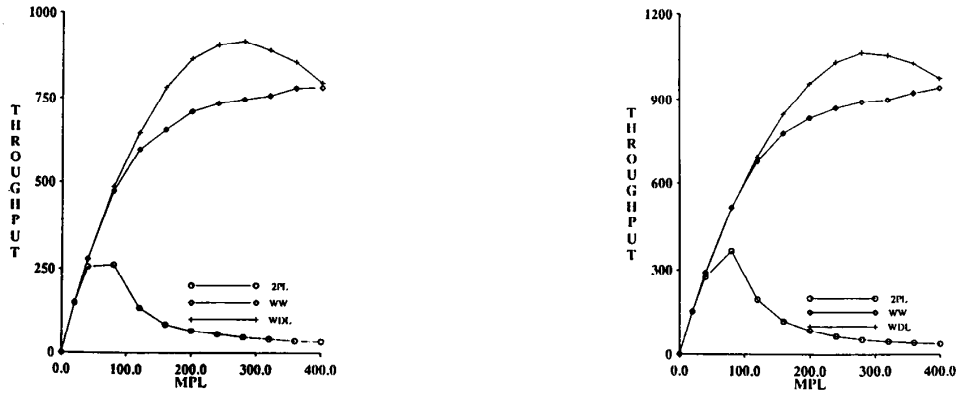


Fig. 7. (a) Uniform Tx size (200 MIPS/CPU). (b) Fixed Tx size (200 MIPS/CPU).

both WW and 2PL. It should also be noted that, while the performance degradation of WDL for MPL's beyond the peak is worse than that of WW, this degradation can be eliminated by using restart waiting [5], [6]. Since information about the completion of transactions at remote nodes is not readily available, randomly generated delays before transaction restart are appropriate in this case.

V. Alternative Distributed WDL Protocols

The simulation results presented in the previous section showed that the distributed WDL concurrency control protocol can provide significant performance benefits over traditional distributed locking algorithms. In this section, we go on to discuss modifications to the Basic protocol that could result in further improving the performance of distributed WDL. We will first informally motivate and describe the modifications, and then conclude by presenting an analysis of these modifications.

A. Eliminating Multiple Restarts

The Basic distributed WDL protocol allows the conflict resolution mechanism at each node to operate asynchronously and independently of the other nodes. A drawback of this scheme, however, is that since each node keeps wait graphs only for transactions that originate at that node, it is possible that *more* transactions may be restarted than are strictly necessary to limit wait chains to a maximum length of one. This "multiple restart" problem was illustrated earlier in Fig. 1(d) and (e).

We describe here three alternative distributed WDL protocols that attempt to address the multiple restart problem. The protocols make different tradeoffs in the number of messages used for conflict resolution and the delay in conflict resolution. To illustrate the functioning of the protocols, we will focus our attention on the case where the conflict node and the parent nodes of the conflicting transactions are all different. The details of the protocols are presented below (in the pictures illustrating the protocols, the dotted lines representing messages indicate messages that *may* need to be sent, while the full lines representing messages indicate messages that *have* to be sent).

1) Table Protocol: In the *Table Protocol*, shown in Fig. 8, information about a conflict is sent to the parent nodes of both the conflicting transactions, just as in the standard protocol. On receipt of this information, each parent node updates its local wait graph and executes the WDL algorithm locally. However, the WDL decision is not implemented right away. Instead, the decision of each parent node is transmitted to the complementary parent node. At both nodes, using the local decision and the remote decision as indexes into a "Conflict Decision Table" (Fig. 8), a *consensus* decision is implemented. If the table entry is "Block T_1," then both nodes do not take further action. If the table entry is "Restart T_1 ," Node 1 implements the decision, and correspondingly, if the table entry is "Restart T_2," Node 2 handles the restart. Finally, if the table recommendation is "Restart T_y ," Node 2 restarts T_y if it is a locally originating transaction; otherwise, it sends a restart message to the parent node of T_y.

Due to the consensual nature of the decision process, the Table Protocol eliminates the problem of multiple restarts. It is also a completely general protocol since even if the internal mechanisms of the WDL algorithm were to be altered, the cooperative decision-making ensures proper coordination among the conflicting parent nodes. Note that the Conflict Decision Table is very simply derived by determining what the correct WDL decision would have been if a global perspective of the wait chain were available.

The benefits of the Table Protocol are gained, however, at the expense of an increased number of messages (due to decision transmissions) and delays in conflict resolution (due to having to wait for the complementary node decision) as compared to the Basic WDL protocol.

2) Sequential Protocol: As mentioned earlier, the Table Protocol is a general protocol that handles the multiple restart problem inherent in the distributed nature of the Basic WDL algorithm. However, for the particular WDL decision process

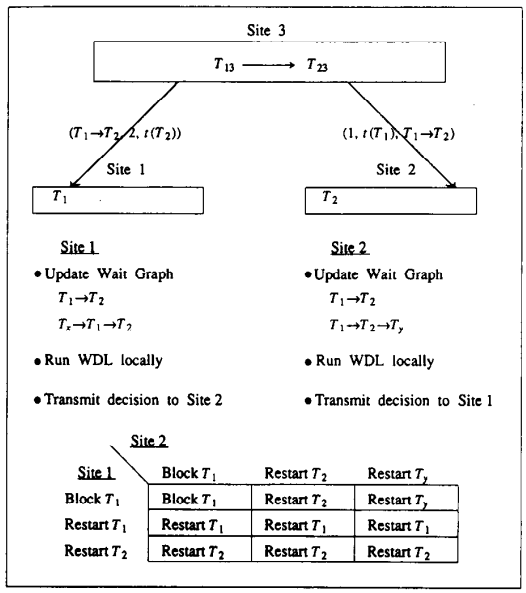


Fig. 8. Table protocol for distributed WDL.

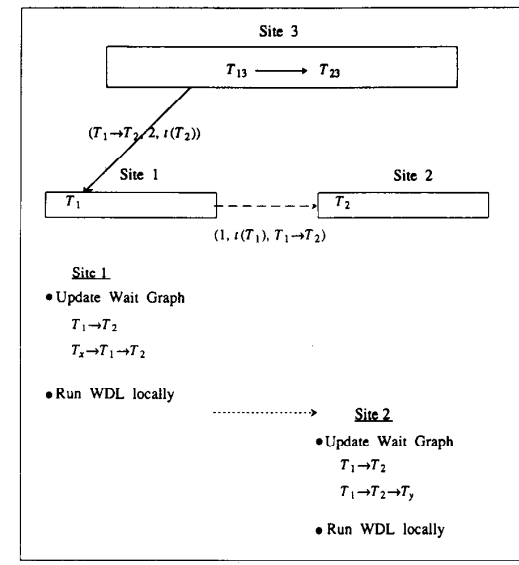


Fig. 9. Sequential protocol for distributed WDL.

of Fig. 1(b), we can improve on the Table Protocol by recognizing that Node 2 needs to be involved in the decision process *only* if Node 1 reaches a "block T_R decision" after executing the WDL algorithm on its piece of the wait graph. The *Sequential Protocol* takes advantage of this feature of WDL. In this protocol, shown in Fig. 9, information about a conflict is sent *only* to the parent node of the transaction whose data request caused the conflict. Accordingly, in Fig. 9, Node 3 sends the information about the conflict between T_1 and T_2 to Node 1 alone. On receiving the conflict message, Node 1 updates its wait graph and executes the WDL algorithm on the resultant graph. If the WDL decision is "Restart T_1," the decision is implemented locally. If the decision is "Restart T_2," a restart message for T_2 is sent to Node 2. However, if the decision is "Block T_1," then the conflict information is forwarded to Node 2. On receiving this message, Node 2 updates its local wait graph, executes WDL on it, and implements the resulting decision.

The Sequential Protocol eliminates multiple restarts by making WDL decisions in *sequence*. It also reduces the number of messages for conflict resolution (e.g., if Node 1 decides to restart T_1, then Node 2 does not even get to know of the occurrence of the conflict). Compared to the Basic WDL protocol, a drawback of the Sequential Protocol is that the delay in conflict resolution may be increased in some cases due to Node 2 obtaining knowledge of the conflict only after hearing from Node 1. However, it is no worse than the Table Protocol in this respect, and in cases where the WDL decision is handled completely by Node 1, the delay is less than that of the Table Protocol.

3) Local Protocol: A further improvement on the Sequential Protocol can be made by recognizing that Node 3, the conflict node, could potentially resolve the conflict *locally* if it possessed information about transaction T_1's associated wait graph. Therefore, in the *Local Protocol*, shown in Fig. 10, when a lock request is made by a transaction to a remote node, information about the current wait graph status of the requesting transaction is sent along with the request. Accordingly, in Fig. 10, the wait chain associated with T_1 is sent to Node 3 along with its data request, and this information is incorporated into the wait graph maintained by Node 3. When the conflict between T_1 and T_2 occurs, the WDL algorithm is executed locally at Node 3. Based on the WDL decision, one of the following courses of action is taken: 1) if the decision is "Restart T_1," then T_1 is removed from the queue of waiters for the conflict lock, and a message is sent to Node 1, which then handles the lock release at other nodes; 2) if the decision is "Restart T_2," the conflict lock is locally released and a message is sent to Node 2, which then handles the lock release at the remaining nodes; or 3) if the decision is "Block T_1," however, the remainder of the protocol is identical to that of the Sequential protocol.

The advantages of the Local Protocol are that it eliminates multiple restarts, reduces the number of messages needed for conflict resolution, and reduces the conflict resolution delay. In fact, unlike the Basic, Table, and Sequential protocols, where *all* conflicts have to be reported to the parent nodes of one or both of the conflicting transactions, the Local Protocol resolves conflicts at the conflict node itself in some cases. This aspect of the Local protocol may considerably reduce message traffic and delay in resolving conflicts.

A disadvantage of the Local Protocol, however, is that when WDL is executed at the conflict node, it uses "old" information about the wait graph of the requesting transaction. This could result in restarts that may be unnecessary from the viewpoint of the *current* state of the global wait graph (e.g., T_1 may be blocking T_x at the time of data request to Node 3, but this

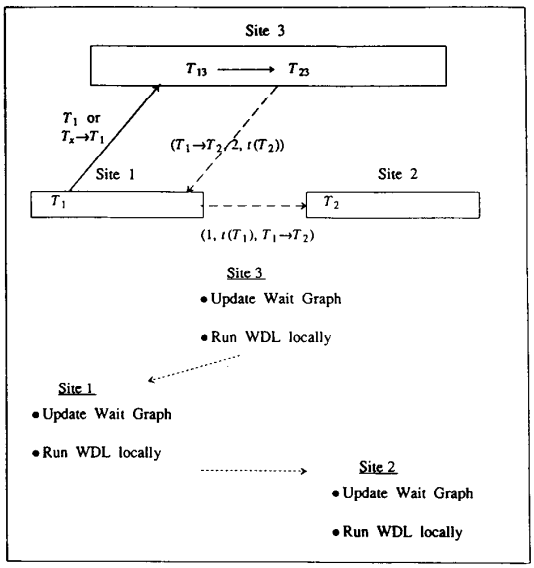


Fig. 10. Local protocol for distributed WDL.

may no longer be true at the time of conflict between T_1 and T_2). Also, in cases where the final conflict resolution decision is made by Node 2, the Local Protocol has increased latency, as compared to the Basic WDL protocol, in resolving conflicts due to the delay caused by the sequential passing around of the conflict information. However, it is no worse than either the Sequential Protocol or the Table Protocol in this respect.

In summary, the Table Protocol offers a general solution to the multiple restart problem, while the Sequential Protocol and the Local Protocol are optimizations that are based on the specific WDL algorithm presented in this paper. In the following subsection, we outline a further optimization to the above set of modified protocols.

B. Fast Wait-Chain Breaking

In the Basic distributed WDL protocol, the release of locks held by a transaction that is scheduled for restart occurs only when its parent node sends a lock release message to each of its subtransaction nodes. However, the decision to restart the transaction may have been arrived at (by a different node) considerably earlier. For example, in the Sequential Protocol (Fig. 9), Node 1 may decide to restart transaction T_2, but the restart process takes place only after Node 1 sends a "Restart T_2 " message to Node 2. This delay in implementing the restart decision increases the time period during which wait-chains of length greater than one exist, and therefore degrades performance. It would therefore be helpful if, in the above example, Node 1 could itself initiate the restart of T_2. This is not possible since Node 1 does not know the locations of all of T_2's subtransactions. The important point to note, however, is that we desire quick release primarily for the lock that corresponds to a wait-chain of length greater than one, that is, the lock on which T_1 and T_2 are conflicting. Release of this "critical" lock *can* be initiated by Node 1 since it knows the location of the conflict, i.e., Node 3. To elaborate, Node 1 can send a transaction restart message to Node 2, the parent node of T_2, and *simultaneously* send a lock release message to Node 3 for transaction T_2's subtransaction T_{23}. When Node 2 receives the transaction restart message from Node 1, it sends a lock release message for T_2 to all the remaining subtransaction nodes of T_2 apart from the conflict node. Therefore, there are savings in both the number of messages (this could be particularly significant if transactions typically have only a few remote subtransactions) and in the delay in conflict resolution.

In summary, a conflict decision node can send a restart message to the parent node of a remotely originating transaction and simultaneously send a lock release message for this transaction to the conflict occurrence node, thereby reducing the delay in the breaking of wait chains. This improvement can be added to the modified protocols described above. For the Table Protocol, the optimization comes into play only when the final decision is to restart T_y and the parent node of T_y is different than that of T_2. In the case of the Local and Sequential Protocols, however, in addition to the above case, the optimization also takes effect whenever Node 1 decides to restart T_2.

A point to note here is that if a node receives a lock release message for a subtransaction that is in the commit process, the lock release message is ignored since the subtransaction will soon be releasing its locks.

C. Protocol Analysis

In the protocol descriptions above, we informally highlighted the features of the various modified protocols by considering the case where the conflict node and the parent nodes of the conflict transactions were all different. We now go on, in this section, to provide a more concrete analysis of the expected performance behavior of the modified algorithms. To this end, we have presented, in Fig. 11, a detailed tabulation of the message, delay, and restart characteristics of the various distributed WDL protocols over *all* possible conflict situations. In Fig. 11, each protocol is characterized by three columns: CRO, CRM, and CRD. The first column, CRO (Conflict Resolution Outcome), describes the possible outcomes of the WDL decision process. The possible outcomes are to Block (B), to Restart $T_1\,(R1)$, to Restart $T_2\,(R2)$, or to Restart $T_y\,(Ry)$. The modified protocols (Table, Sequential, and Local) ensure that only one of these outcomes finally takes effect. With the Basic protocol, however, a combination of two outcomes is implemented when the parent nodes of the conflicting transactions are different (all possible combinations of outcomes are enumerated in the table). The second protocol column in Fig. 11, CRD (Conflict Resolution Delay), is the time period (measured in number of message transmission delays) between the occurrence of a lock conflict and the completion of the WDL decision process at the conflict and parent nodes (this includes the time for the physical breaking of the wait chain in the cases where the conflict results in

CONFLICT SCENARIO	BASIC PROTOCOL			TABLE PROTOCOL			SEQUENTIAL PROTOCOL			LOCAL PROTOCOL		
I. P2≠Py≠N2y	CRO	CRD	CRM	CRO	CRD	CRM	CRO	CRD	CRM	CRO	CRD	CRM
a) P1=P2=N12	B	0	0	B	0	0	B	0	0	B	0	0
	R1	0	0	R1	0	0	R1	0	0	R1	0	0
	R2	0	0	R2	0	0	R2	0	0	R2	0	0
	Ry	2	2	Ry	1	2	Ry	1	2	Ry	1	2
b) (P1=P2)≠N12	B	1	1	B	1	1	B	1	1	B	1	1
	R1	2	2	R1	2	2	R1	2	2	R1	1,2	1,2
	R2	2	2	R2	2	2	R2	2	2	R2	1,2	1,2
	Ry	3	3	Ry	2	3	Ry	2	3	Ry	2	3
c) (P1=N12)≠P2	B,B	1	1	B	2	2	B	1	1	B	1	1
	B,R2	2	2	R1	2	2	R1	0	0	R1	0	0
	B,Ry	3	3	R2	2	2	R2	1,2	1,2	R2	1,2	1,2
	R1,B	1	1	Ry	2	4	Ry	2	3	Ry	2	3
	R1,R2	2	2									
	R1,Ry	3	3									
	R2,B	2	2									
	R2,R2	2	3									
	R2,Ry	3	5									

CONFLICT SCENARIO	BASIC PROTOCOL			TABLE PROTOCOL			SEQUENTIAL PROTOCOL			LOCAL PROTOCOL		
I. P2≠Py≠N2y	CRO	CRD	CRM	CRO	CRD	CRM	CRO	CRD	CRM	CRO	CRD	CRM
d) P1≠(P2=N12)	B,B	1	1	B	2	2	B	2	2	B	2	2
	B,R2	1	1	R1	2	2	R1	2	2	R1	1,2	1,2
	B,Ry	2	3	R2	2	2	R2	2	2	R2	0,2	0,2
	R1,B	2	2	Ry	3	4	Ry	3	4	Ry	3	4
	R1,R2	2	2									
	R1,Ry	2	4									
	R2,B	2	2									
	R2,R2	2	2									
	R2,Ry	2	4									
e) P1≠P2≠N12	B,B	1	2	B	2	4	B	2	2	B	2	2
	B,R2	2	3	R1	3	5	R1	2	2	R1	1,2	1,2
	B,Ry	3	4	R2	3	5	R2	2,3	3	R2	1,2,3	1,2,3
	R1,B	2	3	Ry	3	6	Ry	3	4	Ry	3	4
	R1,R2	2	4									
	R1,Ry	3	5									
	R2,B	3	4									
	R2,R2	2	4									
	R2,Ry	3	6									

Protocols

CONFLICT SCENARIO	
II. P2=Py≠N2y	All entries remain identical to that for P2=Py=N2y except that: 1) The CRD and CRM are both reduced by 1 for CRO=(*,Ry) in BASIC protocol. 2) The CRM is reduced by 1 for CRO=(Ry) in all the modified protocols.
III. P2=Py=N2y	All entries remain identical to that for P2=Py=N2y except that: 1) The CRD is reduced by 2 for the CRO=(B,Ry) and by 1 for CRD=(R1,Ry) and CRO=(R2,Ry) in the BASIC protocol. The CRM is reduced by 2 for (*,Ry). 2) The CRD is reduced by 1 for CRO=(Ry) and the CRM is reduced by 2 for CRO=(Ry) in all the modified protocols.

Symbol	Meaning
P1	$P(T_1)$
P2	$P(T_2)$
N12	Conflict node ($T_1 \rightarrow T_2$)
Py	P(Ty)
N2y	Conflict node ($T_2 \rightarrow T_y$)
B	Block
R1	Restart T_1
R2	Restart T_2
Ry	Restart Ty

Legend for Figure 11

Fig. 11. Conflict resolution costs for distributed WDL implementations.

wait-chains of length greater than one). The last protocol column, CRM (Conflict Resolution Messages), is the number of messages that are sent in order to resolve the conflict during the Conflict Resolution Delay (CRD) period.

1) Table Construction: In this section, we will illustrate how the table entries in Fig. 11 describing the conflict resolution delay and message numbers for each conflict situation are derived for the various protocols.[7] We use case I-e) in Fig. 11 as our example (this case captures the situation where the conflict node and the parent nodes of the conflicting nodes are all different). Assume that the right conflict resolution decision (from a global perspective of the wait chain) is to restart T_1, i.e., the outcome of the WDL decision process should be $R1$, and that Node 3 is the conflict node. If we examine the Basic Protocol, an $R1$ decision takes two conflict information messages (one to each parent node), and then a message from Node 1 to Node 3 to remove T_{13} from the queue of lock waiters. Therefore, the delay in implementing the decision to restart R_1 is CRD = 2 (since the conflict information messages are sent in parallel), while the number of messages involved in the decision process is CRM = 3. In addition, based on the WDL decision at Node 2, additional messages may be sent for restarting either T_2 or T_y. A decision to restart T_2, for example, will take an additional message to Node 3 to release T_{23}'s locks. Therefore, the CRM for the Basic Protocol for an $R1$ decision could be 3, 4, or 5, based on the WDL decision at Node 2.

Moving on to the Table protocol, it is easily determined that it takes five messages to implement an $R1$ decision (two conflict information messages from Node 3 to Nodes 1 and 2, two messages for exchange of WDL decision between Node 1 and Node 2, and one more message from Node 1 to Node 3 to remove T_{13} from the queue of lock waiters), thereby resulting in CRM = 5. Further, the delay in the decision process is CRD = 3 since the conflict information messages and the conflict decision messages are sent in parallel.

Turning our attention to the Sequential Protocol, we find that it takes two messages to implement the $R1$ decision, one for the conflict information to be transmitted from Node 3 to Node 1, and the second message from Node 1 to Node 3 to remove T_{13} from the queue of lock waiters. Therefore, the delay in implementing the decision is CRD = 2, while the number of messages is CRM = 2.

Finally, for the Local Protocol, it is possible that the conflict is resolved at Node 3 itself. If this is so, only one message has to be sent from Node 3 to Node 1 informing it of the $R1$ decision, resulting in CRD = 1 and CRM = 1. If Node 3 cannot resolve the conflict itself, however, then the number of messages and the delay is identical to that of the Sequential Protocol.

In a similar fashion, we can derive the message complexity and delay involved in conflict resolution for the various combinations of conflict node location, parent node locations, and WDL outcome.

2) Performance Expectations: By comparing the entries for the various protocols in Fig. 11, we can make several observations about their expected performance. First, the message complexity of the Sequential Protocol is strictly (i.e., under all conflict situations) less than that of the Table Protocol, while the message complexity of the Local Protocol is strictly less than that of the Sequential Protocol. Second, the message delay of the Local Protocol is strictly less than that of the Sequential Protocol, which in turn is strictly less than that of the Table Protocol. Therefore, the Local Protocol has the best performance (among the modified algorithms) for both of these measures. As pointed out earlier, however, the Local Protocol has a hidden cost in that it may generate false (unnecessary) restarts due to using "old" information about the wait graph of the conflicting transactions. The significance of this cost will be determined by the execution time of subtransactions, since the longer the execution time, the greater the possibility of the wait graph having changed between the time that the subtransaction began executing and the time the conflict occurred. For business applications, however, where transactions are typically simple in structure, we expect subtransaction execution times to be relatively small, and therefore false restarts may occur only infrequently. Therefore, from an overall perspective, the Local Protocol appears to be the most promising candidate among the modified protocols.

If we compare the Local Protocol with the Basic Protocol, we note that its message complexity is strictly less than that of the Basic Protocol. However, the delay of the Local Protocol can be, based on the conflict situation, either more or less than that of the Basic Protocol. If the final decision is to block, for example, the Local Protocol has one more delay than the Basic Protocol (if Node 1 is different than Node 3). In contrast, if the final decision is to restart T_1, then the delay may be one less than or the same as that of the Basic Protocol. For the case where the final decision is to restart T_2, the delay of the Local Protocol may be either one less than, the same, or one more than that of the Basic Protocol. Finally, in the case where the final decision is to restart T_y, the delay is less than or the same as that of the Basic Protocol.

As can be deduced from the above discussion, the only situation where the delay of the Local Protocol is strictly worse than the Basic Protocol is when the final decision is to block, and is limited to the case where the conflict node is different than the parent node of the lock requesting transaction. On the other hand, unlike the Basic Protocol, the Local Protocol does not suffer from the problem of multiple restarts.

Given the above characteristics of the protocols, it is our expectation that the Local Protocol would improve on the performance of the Basic Protocol, at least in the parameter regions in which we expect future high-performance systems to operate. Of course, the actual extent of performance improvement needs to be evaluated with a performance study. While a detailed quantitative analysis of the message and delay characteristics of the various protocols is outside the scope of this paper, we present here the method by which these relative performance measures could be computed.

A recent study presents the analysis of (a slightly modified version) of WDL in a centralized system [21]. A byproduct of the analysis is the relative frequency of events which occur upon a lock conflict (transaction blockings and different types

[7] We assume, in this tabulation, that the optimization for fast wait-chain breaking is incorporated in the modified protocols.

of restarts). Aside from the frequency of restarts, we need to know the probabilities associated with different configurations given in Fig. 11. This aspect of the analysis can also be undertaken using techniques similar to those described in [22], where for a given locality of access, the distribution of the number of accesses to remote nodes is computed. Given the relative frequencies of restarts, the probabilities of various configurations can be used to obtain estimates for CRM and CRD for the different implementations of distributed WDL. Due to feedback effects, such results should be considered with caution. However, these analytical results can be used as components of an analytic solution for distributed WDL. A complete analysis of distributed WDL methods is beyond the scope of this paper.

VI. Conclusions

Distributed Wait-Depth Limited (DWDL) CC, a new distributed concurrency control method that is designed for high lock contention environments, is described in this paper. The DWDL method selectively utilizes transaction restarts to prevent the performance degradation that is caused by transaction blocking which occurs with two-phase locking with the general waiting policy. In addition, DWDL ensures that deadlocks (local or distributed) are prevented from occurring by limiting the wait-depth of blocked transactions to no more than one.

In distributed systems, message costs, in terms of processing overhead and internode communication delay, are relatively high in practice. Therefore, in designing the distributed WDL protocol, appropriate modifications were made to the centralized WDL paradigm to minimize the number of messages, while retaining desirable WDL properties. These changes include the use of transaction arrival time, instead of the number of locks held, as an indication of transaction progress.

Detailed simulation results showed that distributed WDL outperforms both standard 2PL and the wound-wait method in high MIPS systems with a high degree of lock contention. The improvement in performance was observed to increase with increased system processing capacity.

DWDL's improvement in performance with respect to standard locking schemes is attained at the cost of extra processing, which is due to the messages required for updating conflict graphs and restarting transactions. This extra processing is almost exclusively in the form of CPU overhead since we postulate large database buffers such that, given a high degree of access invariance [5], [7], the need for additional disk accesses is obviated. Given recent hardware trends of increasing CPU MIPS and decreasing semiconductor memory costs, this additional CPU processing may be an acceptable approach for achieving higher transaction throughputs, without requiring the redesign of transactions. However, it should be noted that, due to DWDL's greater message complexity, its performance is more susceptible to reductions in the locality of transaction data access and increases in the cost of sending messages than the other standard locking-based CC methods considered in this paper.

The basic DWDL algorithm has some drawbacks in that it may restart more transactions than are strictly necessary for restricting transaction wait-depths to one. Further, in order to minimize the delay in conflict resolution, it utilizes more messages than may be necessary for resolving conflicts. The performance of DWDL could perhaps be further improved by addressing these drawbacks. To this end, we proposed three alternate distributed WDL schemes and presented an empirical comparison of their performance (based on the number of messages and delay involved in conflict resolution) with respect to each other and the Basic method. One of these methods, the Sequential Protocol, appears particularly attractive, and in our future research, we plan to carry out a complete performance study of these alternative protocols.

A significant reduction in WDL overhead can be attained by using a periodic policy for propagating lock conflict information, as is done for conflict resolution in distributed systems [2]. Since the probability of encountering more complex graphs is more likely in the case of periodic conflict resolution, alternate schemes such as the sequential scheme and the more general data conflict resolution rules presented in [6], [7] may potentially provide improvements to the Basic distributed WDL paradigm in this case.

Acknowledgment

The simulator used in this study is an extension of a DeNet based simulator of a centralized DBMS developed by R. Jauhari, H. Pirahesh, and C. Mohan at IBM Almaden Research Center-ARC. The authors thank ARC for the donation of their simulator. We also thank M. Livny for his assistance with DeNet.

References

[1] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Trans. Database Syst.*, vol. 12, pp. 609–654, Dec. 1987.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

[3] M. J. Carey and M. Livny, "Distributed concurrency control performance: A study of algorithms, distribution, and replication," in *Proc. 14th Int. Conf. Very Large Data Bases,* Los Angeles, CA, Aug. 1988, pp. 13–25.

[4] P. A. Franaszek and J. T. Robinson, "Limitations of concurrency in transaction processing," *ACM Trans. Database Syst.,* vol. 10, pp. 1–28, Mar. 1985.

[5] P. A. Franazsek, J. T. Robinson, and A. Thomasian, "Access invariance and its use in high contention environments," in *Proc. 6th IEEE Data Eng. Conf.,* Los Angeles, CA, Feb. 1990, pp. 47–55.

[6] ——, "Wait depth limited concurrency control," in *Proc. 7th IEEE Conf. Data Eng.,* Kobe, Japan, Apr. 1991, pp. 92–101.

[7] ——, "Concurrency control for high contention environments," *ACM Trans. Database Syst.,* vol. 17, pp. 304–345, June 1992.

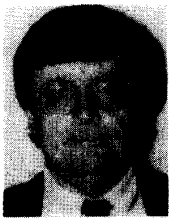
[8] P. Heidelberger and M. S. Lakshmi, "A performance comparison of multimicro and mainframe database architectures," *IEEE Trans. Software Eng.,* vol. 14, pp. 522–531, Apr. 1988.

[9] J. N. Gray, "The cost of messages," in *Proc. 7th Annu. Symp. Principles of Distributed Computing,* Toronto, Ont., Canada, Aug. 1988, pp. 1–7.

[10] T. Haerder, "Observations on optimistic concurrency control schemes," *Inform. Syst.,* vol. 9, no. 2, pp. 111–120, 1984.

[11] B. C. Jenq, B. C. Twitchell, and T. W. Keller, "Locking performance in a shared nothing parallel database machine," *IEEE Trans. Knowledge Data Eng.,* vol. 1, pp. 530–543, Dec. 1989.

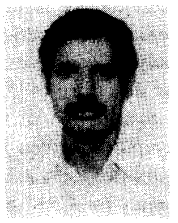
[12] M. Livny, *DeNet User's Guide, Version 1.0,* Dep. Comput. Sci., Univ. Wisconsin, Madison, 1988.
[13] C. Mohan, "Less optimism about optimistic concurrency control," in *Proc. 2nd Int. Workshop Res. Issues in Data Eng.,* Tempe, AZ, Feb. 1992, pp. 199–204.
[14] D.J. Rosenkrantz, R.E. Stearns, and P. M. Lewis, II, "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.,* vol. 3, pp. 178–198, June 1978.
[15] K.C. Sevcik, "Comparison of concurrency control methods using analytic models," in *Information Processing 83: Proc. 9th IFIP World Congr.,* R.E.A. Mason, Ed., Paris, France, Sept. 1983, pp. 847–858.
[16] M. Stonebraker, "The case for shared nothing," *Database Eng. Bull.,* vol. 9, pp. 4–9, Mar. 1986.
[17] Y. C. Tay, N. Goodman, and R. Suri, "Locking performance in centralized databases," *ACM Trans. Database Syst.,* vol. 10, pp. 415–462, Dec. 1985.
[18] A. Thomasian and E. Rahm, "A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking," in *Proc. 10th Int. Distributed Computing Conf.,* Paris, France, May 1990, pp. 294–301.
[19] A. Thomasian, "Performance limits of two-phase locking," in *Proc. 7th IEEE Conf. Data Eng.,* Kobe, Japan, Apr. 1991, pp. 426–435.
[20] A. Thomasian and I.K. Ryu, "Performance analysis of two-phase locking," *IEEE Trans. Software Eng.,* vol. 17, May 1991.
[21] A. Thomasian, "Performance analysis of locking policies with limited wait-depth," in *Proc. ACM SIGMETRICS/Performance '92 Conf.,* Newport, RI, June 1992, pp. 115–127.
[22] ——, "On the number of remote sites accessed in distributed transaction processing," *IEEE Trans. Parallel Distributed Syst.,* to be published. Also, IBM Res. Rep. RC 15430, Hawthorne, NY, Jan. 1990.



Peter Franaszek (S'63–M'66–SM'89–F'90) received the Sc.B. degree from Brown University, and the M.A. and Ph.D. degrees from Princeton University.

He is Manager of Systems, Theory and Analysis in the Computer Sciences Department at the IBM Thomas J. Watson Research Center. His interests include analysis and design principles in computer system organization, algorithms, communication networks, and coding. He has received a variety of IBM awards for this work in the areas of algorithms, interconnection networks, concurrency control principles, and coding theory. In 1991, he was elected to the IBM Academy of Technology. He was the recipient of the 1989 Emmanuel R. Piori award of the IEEE for his contributions to the theory and practice of digital recording codes. During the academic year 1973–1974, he was on leave from IBM to Stanford University as Consulting Associate Professor of Computer Sciences and Electrical Engineering. Prior to joining IBM, he was a member of technical staff at Bell Telephone Laboratories.

Dr. Franaszek is a member of Tau Beta Pi and Sigma Xi.



Jayant R. Haritsa (S'91–M'91) received the B.S. degree in electronics and communications engineering from the Indian Institute of Technology, Madras, in 1985, and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin, Madison, in 1987 and 1991, respectively.

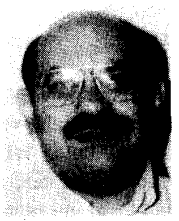
He is currently a Post-Doctoral Fellow with the Systems Research Center, University of Maryland, College Park. During 1988 and 1990, he spent summers at the Microelectronics and Computer Technology Consortium and at the IBM T. J. Watson Research Center, respectively. His research interests include database systems, real-time systems, network management, and performance modeling.

Dr. Haritsa is a member of the ACM.

John T. Robinson received the B.S. degree in mathematics from Stanford University, Stanford, CA, in 1974, and the Ph.D. degree in computer science from Carnegie-Mellon University in 1982.

Since 1981 he has been with the IBM T. J. Watson Research Center, Yorktown Heights, NY. His current research interests include database systems, file systems, parallel and distributed processing, and design and analysis of algorithms.

Dr. Robinson is a member of the ACM and the IEEE Computer Society.



Alexander Thomasian received the B.S.E.E. degree from the University of Tehran, Iran, and the M.Sc. and Ph.D. degrees in computer science from the University of California, Los Angeles.

He is a member of the research staff in the Systems Analysis Department at the IBM T. J. Watson Research Center. He was a faculty member at Case Western University and the University of Southern California and a senior staff scientist at Burroughs (now Unisys) Corporation. He has also been an adjunct faculty member in the Computer Science Department at Columbia University. His current research interests are in the area of performance analysis and design of parallel and distributed systems, with emphasis on databases. He has received an Outstanding Innovation Award from IBM and published about 70 papers. He has given tutorials on high-end transaction and query processing systems at several conferences sponsored by IEEE. He serves on the Program Committees of the 1993 Data Engineering and the 1993 Distributed Computing Conference as Vice-Chair of Performance Modeling and Evaluation.

Dr. Thomasian is a member of the IEEE Computer Society and the ACM.