Distributed Concurrency Control with Limited Wait-Depth

P. A. Franaszek, J. R. Haritsa, J. T. Robinson, and A. Thomasian*

IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598, USA

Abstract

We describe Distributed Wait-Depth Limited (DWDL) concurrency control, a locking based method that limits the wait-depth of blocked transactions to one, which assures that deadlocks are resolved as part of regular transaction processing. The performance of DWDL is compared with that of distributed two-phase locking (2PL) and the wound-wait concurrency control method through a detailed simulation. Our results show that DWDL behaves similarly to 2PL for low data contention levels, but at high lock contention levels DWDL outperforms the other methods to a significant degree.

1. Introduction

High-end transaction processing systems have stringent requirements for CPU processing power, I/O bandwidth, high availability, etc. It has been argued [13] that the Shared Nothing (SN) or data partitioned systems are superior to the Shared Everything (SE) or centralized systems and Shared Disk (SD) or data sharing systems from the viewpoint of cost effectiveness, scalability, availability, etc. (however, SD systems, especially when equipped with a shared intermediate storage, can be more robust than SN systems to load imbalance [17]). The performance of SN systems in a transaction processing environment is affected by the number of internode messages generated by transactions, since the cost of sending and receiving messages tends to be non-negligible [8] and may constitute 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 [7].

Two-phase locking (2PL) is the prevalent Concurrency Control (CC) method in commercial database systems. It has been shown in numerous studies (see e.g., [3], [14], [15]) that the performance of a system with 2PL may be constrained by data rather than hardware resource contention. SN systems are more susceptible to performance degradation than centralized systems due to the increased lock holding time. In this paper we adapt the Wait-Depth Limited (WDL) Concurrency Control (CC) method for data partitioned systems and compare its performance with that of other methods.

The WDL CC method (described in Section 2) limits the wait-depth of blocked transactions and is shown in [4,5] to have superior performance with respect to 2PL, other locking methods such as running priority [3], and even optimistic CC methods (when the hardware resources of the system are finite). In this paper we propose an appropriate modification of centralized WDL for distributed database 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.

Simulation is used to compare the performance of DWDL with that of 2PL and the Wound-Wait (WW) method [11]. A key issue in distributed 2PL is the issue of deadlock detection and resolution. Alternative deadlock resolution schemes are based on centralized and distributed combining of wait-for graphs or using timeouts. An advantage of the WW and DWDL methods 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 reached at the node where the lock conflict occurs (without requiring additional messages). Our choice of CC methods covers the three main categories of priority-less (2PL), strict priority (WW), and approximate essential blocking (DWDL) [3].

A large number of papers have been written describing new distributed CC methods and comparing their performance through analysis or simulation. Early works dealing with performance issues of distributed CC methods are surveyed in [12]. A more recent comparative study of CC methods and a survey of other works appears in [2]. A simulation study dealing with the effect of locking on the performance of SN systems is reported in [9].

In Section 2 we describe the DWDL method. The model for the computer systems, the database, and the transaction characteristics considered in the simulation study are described in Section 3. Simulation results are presented in Section 4 and conclusions appear in Section 5.

2. Distributed WDL (DWDL) Concurrency Control

In this section, we first describe the general structure of a distributed transaction in our model, followed by a detailed description of DWDL.

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 (remote) nodes to access local (remote) data. For simplicity, only sequential transaction execution with a single end-of-transaction commit point is considered here. For all the CC methods the two-phase commit protocol is used to assure transaction atomicity.

* Dr. Haritsa was a summer employee at IBM Research in 1990. Current address: Systems Research Center, University of Maryland, College Park, MD 20742, USA.

0-8186-2865-0 /92 $3.00 © 1992 IEEE

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.

The WDL(d) CC methods [4,5] constitute a family of CC methods, which restrict the wait-depth to d levels (only $d = 1$ is considered here). 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) and restarting the transaction which has made less progress.

Consider two active transactions T' and T with m and n transactions waiting on each, respectively (where m or n could be zero) as shown in Figure 1.a. Since for WDL(1) no wait trees greater than one are allowed, there cannot be any transactions waiting on T'_i or T_j and thus this represents the general case for two active transactions. Supposing that T' makes a lock request that conflicts with T or with one of the T_j, then if $m > 0$, temporary states in which wait trees of 2 or 3 can occur are shown in Figures 1.a(b,c) (if $m = 0$ than the wait-depths are one less). Note that for Figure 1.a(c) we must have $n > 0$ in order for this state to arise, also without loss of generality we assume that the conflict is with T_1. In case T' is blocked by one of the transactions blocked by it, the resulting deadlock is resolved by aborting the transaction with the smaller length. One set of rules for implementing WDL(1) is as follows:

1. Case of Figure 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 Figure 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.

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 [4,5]. However in the distributed case, given a particular subtransaction, determination of the total number 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 is used, as follows. Each global transaction is assigned a starting time (for its latest invocation if a transaction is restarted) and this starting time is included in the startup message for each subtransaction so that the starting time of a global transaction is locally known at any node executing one of its subtransactions. The length of a transaction 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 is 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

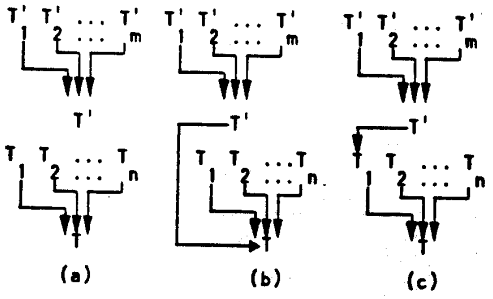

Figure 1a: Initial and temporary states for WDL(1).

model). This conjecture is verified by the simulation results in Section 4. In the case of centralized WDL the cumulative number of locks requested by restarted transactions was also considered [4,5]. 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. 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.

The following notation and conventions is used in explaining the DWDL 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 DWDL method is that (1) whenever a LCC schedules a wait between two subtransactions, this information is sent via messages to the GCCs 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 LCCs and GCCs 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 DWDL 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 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 LCCs 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 than 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 every time a message is received, and using the WDL method determines whether transactions should be restarted.

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 LCCs at each subtransaction node using known techniques. 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 GCCs 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 receiving acknowledgment from all subtransaction nodes and each such GCC.

The above can be illustrated by a simple example, as illustrated in Figure 1b. 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 GCCs 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 GCCs 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 GCCs 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 trans-

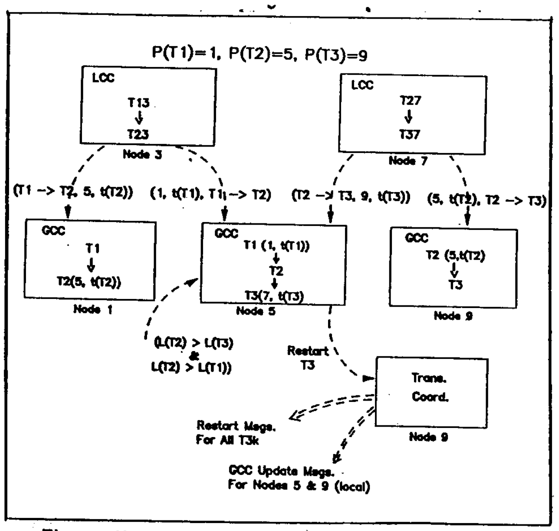

Figure 1b: A simple example of distributed WDL CC

action (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 GCCs operating independently and asynchronously, decisions could be made concurrently by two or more GCCs 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 Figure 2. Nodes 1 and 2 receive 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 Figure 2. In this scenario, as per the basic WDL method (refer to Figure 1a), 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

Site 1	Site 2
Restart T_1	Restart T_2
Restart T_1	Restart T_y
Restart T_2	Restart T_y
Restart T_2	Restart T_2

two transactions are restarted. If we consider the conflict from a global perspective, however, we see 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 DWDL method described here to have a performance characteristic similar to the centralized WDL method.

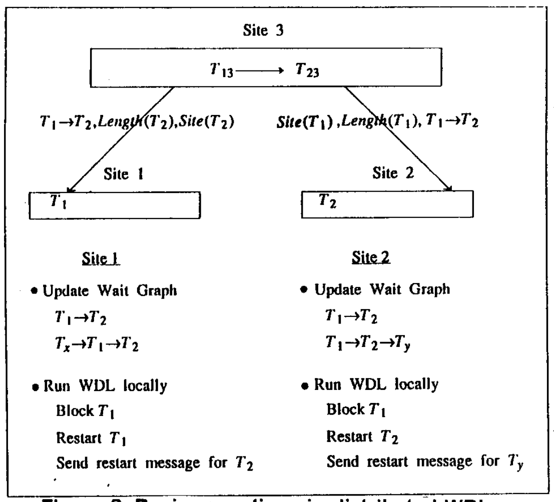
Site 3
$T_{13} \rightarrow T_{23}$
$T_1 \rightarrow T_2, Length(T_2), Site(T_2)$
$Site(T_1), Length(T_1), T_1 \rightarrow T_2$
Site 1
Site 2
T_1
T_2
Site 1
Site 2
• Update Wait Graph
$T_1 \rightarrow T_2$
$T_x \rightarrow T_1 \rightarrow T_2$
• Update Wait Graph
$T_1 \rightarrow T_2$
$T_1 \rightarrow T_2 \rightarrow T_y$
• Run WDL locally
Block T_1
Restart T_1
Send restart message for T_2
• Run WDL locally
Block T_1
Restart T_2
Send restart message for T_y

Figure 2: Basic operations in distributed WDL

3. *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 (further references to 2PL, WW, and WDL are to their distributed versions). 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.

3.1 The Computer System Model

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

1. **Multi-system configuration:** There are $N = 4$ computer systems, consisting of tightly-coupled multiprocessors with $P = 4$ processors per system.
2. **Inter-system 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 [2] among others).
3. **I/O subsystem:** The disk service time including any queueing delays is assumed to be fixed and equal to 20 milliseconds 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 3.2) are always in the cache, 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 is not replaced before they are completed.
5. **Logging and recovery:** Non-volatile (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.

3.2 The Database Access Model

The database model considered in this study is described below:

1. **Database Objects:** We distinguish high and low contention data items based on their access frequency by transactions. At each system there are $D_{high} = 256$ ($D_{low} = 7936$) data items in the high (low) contention category. A fraction $F_{high} = 0.25$ ($F_{low} = 0.75$) of all transaction accesses are uniformly to high (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. 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 $P_{hit} = F_{DB\_low} \times F_{low} + F_{high} = 0.625$ (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, rather than the absolute performance attained by the methods.
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 [16]. 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 is handled in this fashion in order to observe how WW and WDL compares with the "best" possible performance of 2PL.*

3.3 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.
2. **Transaction Classes:** We consider four transaction classes based on transaction size. Transaction sizes and associated frequencies are as follows 4(0.20), 8(0.20), 16(0.35), 32(0.25), respectively.
3. **Transaction Processing Stages:**

   Transaction initialization: This requires CPU processing only and the pathlength for this stage is

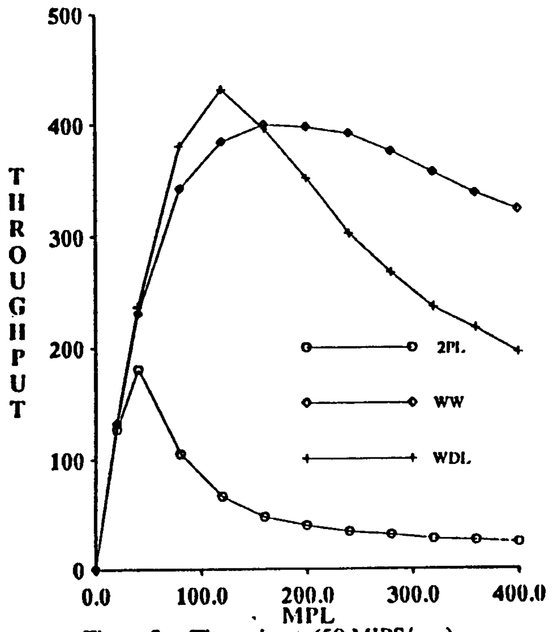

Figure 3a: Throughput (50 MIPS/cpu)

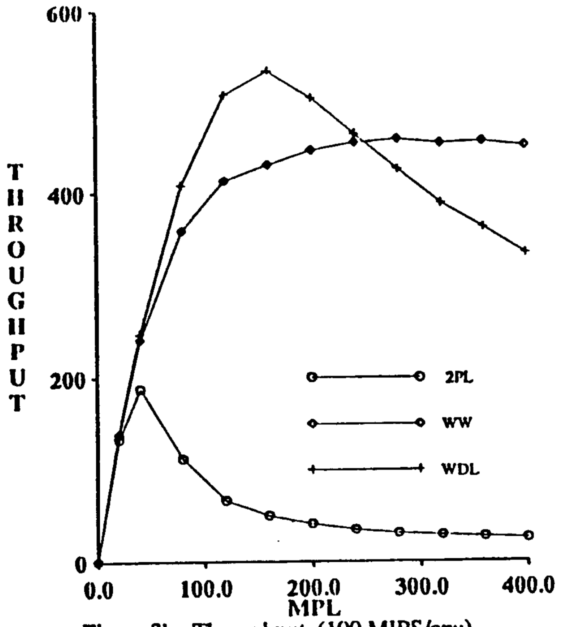

Figure 3b: Throughput (100 MIPS/cpu)

I_{init1} = 100,000 instructions. If the transaction is restarted due to failed validation or having been selected as the victim for deadlock resolution, then I_{init2} = 50,000. Restarted transactions follow the same access pattern as their original incarnation.

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 $1 - F_{local}$ 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_{cache} = 20,000. This includes the overhead for CC. Otherwise, when data has to be accessed from disk, an additional I_{disk} = 5000 instructions are required (the processing required to retrieve cached data is considered to be negligible). In addition, it takes I_{send} = 5000 instructions to send or receive a message. Therefore, 20,000 instructions are executed for inter-system communication when the data is not available locally.

Transaction completion: The CPU processing in this stage requires $I_{complete}$ = 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_{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, $I_{pre-commit}$ = 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 was accessed requires I_{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.

4. Simulation Results

In this section, we present performance results for the 2PL, WW and DWDL CC methods obtained from a simulator written in DeNet [10]. 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 Multi-Programming Level (MPL). 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 hardware resources. Each simulation was run until steady state behavior is achieved (this 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 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.

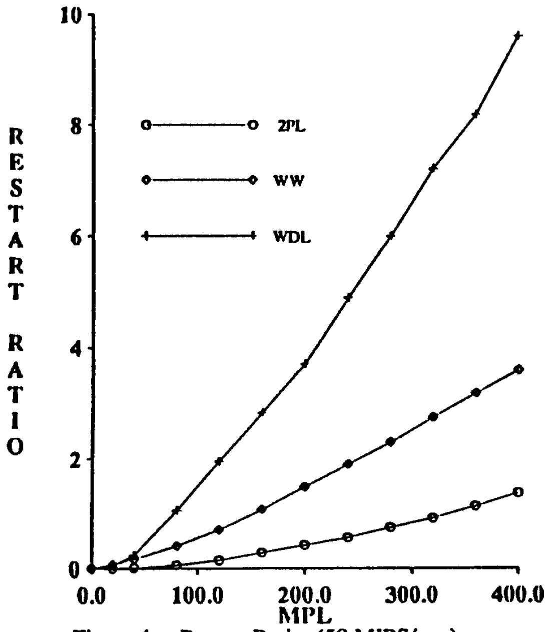

Figure 4a: Restart Ratio (50 MIPS/cpu)

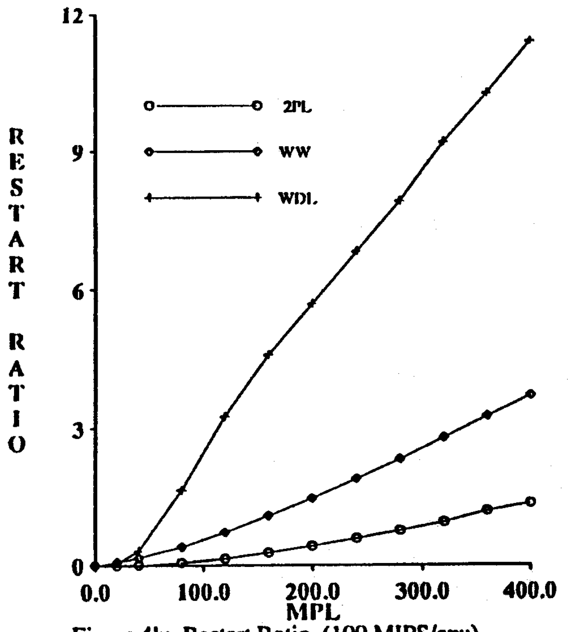

Figure 4b: Restart Ratio (100 MIPS/cpu)

Our 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 3. Figures 3a and 3b present the transaction throughputs obtained under each CC method for per-processor speeds of 50 and 100 MIPS, respectively. In Figures 4a and 4b, 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. Note that the number of transaction restarts is not an adequate indicator of wasted processing, therefore, in Figures 5a and 5b we 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.

We observe that the peak throughput attained by 2PL is considerably smaller than that of the other CC methods. At low MPLs, 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 MPLs beyond this peak throughput, a sharp fall in transaction throughput is seen and constitutes the *thrashing region* for 2PL (see e.g., [15]). An important point to observe here is that the performance of 2PL shows only negligible improvement with increase in processor speeds (compare Figures 3a and 3b). The reason that 2PL is unable to take advantage of increased resource capacity is that its conflict resolution mechanism results in most of the transactions being blocked under high lock contention [3,14,1,4,5,15]. 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 utilization graphs for 2PL in Figures 5a and 5b, 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 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.

Finally, with regard to WDL, we observe that it delivers a peak throughput greater than the other CC methods for the set of processors 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

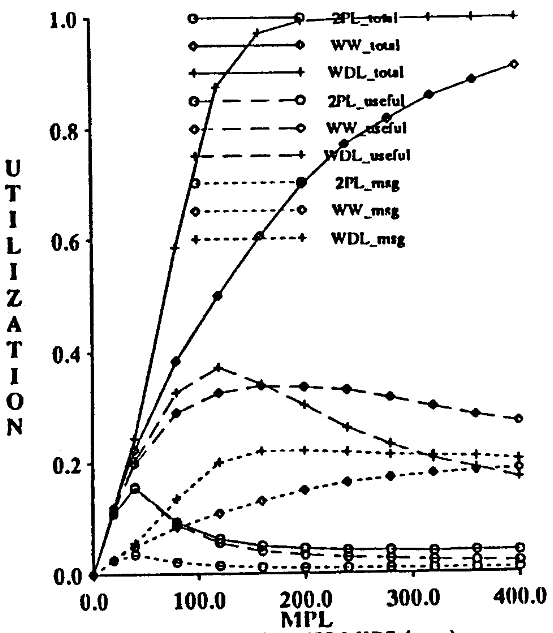

Figure 5a: Utilization (50 MIPS / cpu)

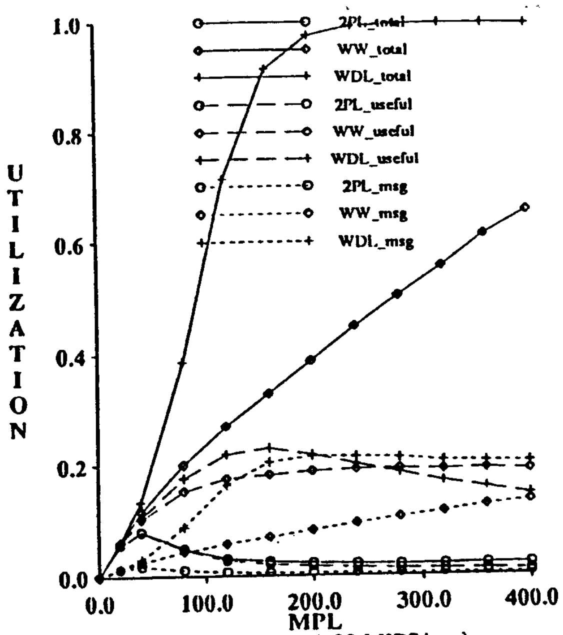

Figure 5b: Utilization (100 MIPS/cpu)

the system is increased, WDL performs increasingly better than the other two CC methods. From Figures 4a and 4b and 5a and 5b, 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 [3] by ensuring that wait-chains never involve more than two transactions. At high MPLs, 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 more parallelism and higher resource utilization. Due to this ability of DWDL 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 2, 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 DWDL. Alternative conflict resolution protocols that eliminate these unnecessary restarts (at the cost of extra delay and increased number of messages) are described in [6]. In [6] we also investigate the effect of varying message costs, locality, and the distribution of transaction sizes.

5. Conclusions

A new distributed CC method, Distributed WDL (DWDL), is presented in this paper. DWDL utilizes transaction restarts to prevent thrashing in high lock contention environments. Deadlocks (local or distributed) are also prevented since the wait-depth of blocked transactions is kept at one.

If the cost of sending messages and the associated delay is negligibly small then the centralized WDL method [4,5] can be approximated rather closely in a distributed environment. Since this is usually not the case in practice, it is desirable to utilize a DWDL implementation that requires a reduced number of inter-node messages, but bases decisions on less accurate information. Appropriate modifications are therefore made to the centralized WDL paradigm to minimize the number of extra messages, while retaining desirable WDL properties.

Detailed simulation results showed that DWDL outperforms standard 2PL and the wound-wait method in high MIPS systems and that the difference in performance increases as the processing capacity of the system is increased.

Acknowledgements

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. 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 TODS 12*,4 (Dec. 1987), 609-654.

[2] M. J. Carey and M. Livny. "Distributed concurrency control performance: A study of algorithms, distribution, and replication," *Proc. 14th Int'l VLDB Conf.* Los Angeles, CA, Aug. 1988, pp. 13-25.

[3] P. A. Franaszek and J. T. Robinson. "Limitations of concurrency in transaction processing," *ACM TODS 10*,1 (March 1985), 1-28.

[4] P. A. Franazsek, J. T. Robinson, and A. Thomasian. "Wait depth limited concurrency control," *Proc. 7th IEEE Conf. Data Engineering*, Kobe, Japan, April 1991, pp. 92-101.

[5] P. A. Franazsek, J. T. Robinson, and A. Thomasian. "Concurrency control for high contention environments," *ACM TODS 17*,2 (June 1992), also *IBM Research Report RC 14665*, Hawthorne, NY, 1989.

[6] P. A. Franazsek, , J. R. Haritsa, J. T. Robinson, and A. Thomasian. "Distributed concurrency control based on limited wait depth," *IBM Research Report RC 16881*, Hawthorne, NY, 1991.

[7] P. Heidelberger and M. S. Lakshmi. "A performance comparison of multimicro and mainframe database architectures," *IEEE TSE 14*,4 (April 1988), 522-531.

[8] J. N. Gray. "The cost of messages," *Proc. 7th Annual Symp. of Principles of Distributed Computing*, Toronto, Canada, August 1988, pp. 1-7.

[9] B. C. Jenq. B. C. Twitchell, and T. W. Keller. "Locking performance in a shared nothing parallel database machine," *IEEE TKDE 1*,4 (Dec. 1989), 530-543.

[10] M. Livny. *DeNet User's Guide*, Version 1.0, Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1988.

[11] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. "System level concurrency control for distributed database systems," *ACM TODS 3*,2 (June 1978), 178-198.

[12] K. C. Sevcik. "Comparison of concurrency control methods using analytic models." *Information Processing 83*, (R. E. A. Mason, ed.), Paris, France, Sept. 1983, pp. 847-858.

[13] M. Stonebraker. "The case for shared nothing," *Database Eng. Bulletin 9*,1 (March 1986), 4-9.

[14] Y. C. Tay, N. Goodman, and R. Suri. "Locking performance in centralized databases," *ACM TODS 10*,4 (Dec. 1985), 415-462.

[15] A. Thomasian. "Performance limits of two-phase locking," *Proc. 7th IEEE Conf. Data Eng.*, Kobe, Japan, April 1991, pp. 426-435.

[16] A. Thomasian and I. K. Ryu. "Performance analysis of two-phase locking," *IEEE TSE 17*,5 (May 1991), 386-402.

[17] P. S. Yu and A. Dan. "Comparison on the impact of coupling architectures to the performance of transaction processing systems," *4th Int'l HPTS Workshop*, Pacific Grove, CA, Sept. 1991.