Distributed Transaction Processing on an Ordering Network

Rashmi Srinivasa, Craig Williams, Paul F. Reynolds Jr.
Department of Computer Science
University of Virginia, Charlottesville
{rashmi, craigw, reynolds}@virginia.edu

Technical Report CS-2001-08
February 2001

Abstract

The increasing demand for high throughputs in transaction processing systems leads to high degrees of transaction concurrency and hence high data contention. The conventional dynamic two-phase locking (2PL) concurrency control (CC) technique causes system thrashing at high data contention levels, restricting transaction throughput. Optimistic concurrency control (OCC) is an alternative strategy, but OCC techniques suffer from wasted resources caused by repeated transaction restarts. We propose a new technique, ORDER, that enlists the aid of the interconnection network in a distributed database system in order to coordinate transactions. The network in an ORDER system provides total ordering of messages at a low cost, enabling efficient CC. We compare the performance of dynamic 2PL and ORDER, using both an analytical model and a simulation. Unlike previously-proposed models for 2PL, our analytical model predicts performance accurately even under high data contention. We study the effects of various parameters on performance, and demonstrate that ORDER outperforms dynamic 2PL for a wide range of workloads.

1 Introduction

Concurrency control (CC) is an integral part of a database system, and is the activity of coordinating the actions of transactions that operate in parallel, access shared data, and potentially interfere with one another [BeHG87]. There are two costs associated with CC: lost opportunity cost and restart cost. The former cost is a significant factor in conservative methods, which involve waiting to ensure that there will be no conflict or interference. Some of this waiting may be unnecessary, constituting a lost opportunity cost. The latter cost is significant in aggressive methods which optimistically execute transactions, based on the assumption that there will be no conflict. If a conflict does arise, some transactions must be aborted and restarted, thus incurring a restart cost. A scalable CC technique that has both a low lost opportunity cost and a low restart cost is a much-sought-after goal. Dynamic two-phase locking (2PL) is the CC technique that current databases use almost exclusively [Date95, GrRe92, BeHG87]. A dynamic 2PL system thrashes at high data contention levels, restricting performance to levels inconsistent with available resources.

This paper makes two contributions: a new CC technique — ORDER, and a new analytical model for 2PL. ORDER uses the interconnection network in a distributed database as an aid to

concurrency control. The network can be a powerful coordination mechanism if it provides the property of total ordering at a low cost. We compare the performance of 2PL and ORDER using both an analytical model and a simulation, and demonstrate that ORDER outperforms dynamic 2PL for a wide range of workloads. Unlike previously-proposed models for 2PL, our analytical model continues to predict performance accurately even at high data contention levels. We also study the performance effects of various parameters like communication delay, transaction size, transaction composition and number of sites.

We consider a distributed database management system with a collection of sites interconnected by a network [BeHG87]. Each site runs one or more of the following software modules: a transaction manager (TM), a data manager (DM) and a concurrency control scheduler (CCS). A *client* runs only the TM module, and a *server* runs only the DM and CCS modules. TMs supervise interactions between users and the DDBMS, CCSs coordinate transactions, and DMs manage the actual database. The network is assumed to be perfectly reliable and point-to-point FIFO. Figure 1 shows the system architecture. The database is a collection of data items or *objects*, and each object is managed by a single DM. Users interact with the DDBMS by executing transactions, which are on-line queries or application programs. Transactions communicate with TMs, TMs communicate with CCSs and DMs, and DMs manage data. In order to execute a transaction, a client issues *read*, *predeclare*, *write*, *commit*, *lock-release* and *abort* operations. A server responds with *read-response* and *lock-set* operations.

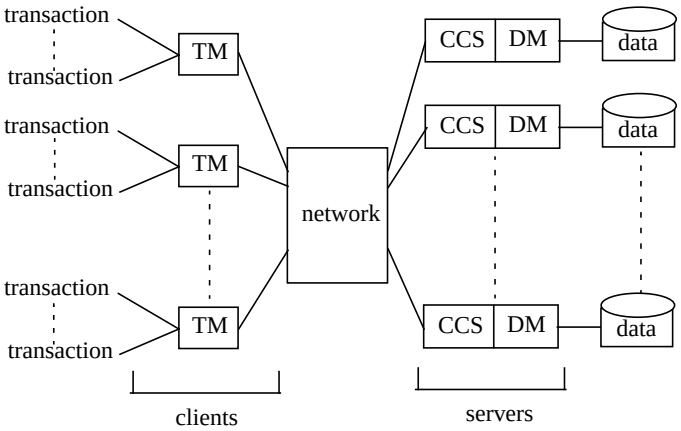


Figure 1: System Architecture

2 Related Work

2.1 Concurrency Control Techniques

Traditionally, CC techniques have been classified into four categories (locking, timestamp-ordering, optimistic and hybrid). In two-phase locking (2PL) [EGLT76], a transaction obtains locks in a growing phase and releases locks in a shrinking phase. In the static version of 2PL, a transaction obtains all of its locks in the beginning, while in dynamic 2PL, a transaction obtains a lock only when it needs to access the corresponding object. A 2PL scheduler requires a strategy to prevent, avoid or detect-and-break deadlocks. Some strategies are waits-for-graphs [Holt72, KiCo74], preordering and predeclaration of locks [BeGo81] and timestamp-priority-based restarts [RoSL78]. Variations on the basic 2PL technique include primary copy 2PL [Ston79], voting 2PL [Thom79], multiversion 2PL [BeHG87, StRo81], centralized 2PL [AlDa76, Garc79], altruistic locking [SaGS94] and increment/decrement locks [GrRe92].

In timestamp-ordering techniques, a unique timestamp is assigned to each transaction, and conflicting transaction operations are ordered according to their timestamps. Different TO-based CC techniques are basic timestamp ordering (BTO) [ShMi77a, ShMi77b], conservative timestamp ordering (CTO) [BeGo80] and multiversion timestamp-ordering (MVTO) [Reed78]. Variations on the basic TO techniques include Thomas Write Rule [Thom79] and transaction classes [BGRP78, BeSR80].

Optimistic (OPT) schedulers schedule each operation as soon as it is received. When a transaction is ready to commit, all involved OPT schedulers check whether committing the

transaction will violate serializability, and abort the transaction if necessary. Such schedulers are also called certifiers [Bada79, Casa79, BaHR80].

Hybrid CC techniques include 2PL-TO combinations [BeGo81], distributed optimistic 2PL (O2PL) [CaLi91], optimistic with dummy locks (ODL) [HaDo91], hybrid optimistic CC and broadcast optimistic CC [YuDi92], and dynamic locking with no waiting (DLNW) [RyTh90b]. Current databases use dynamic 2PL and its variants almost exclusively [Date95, GrRe92, BeHG87].

2.2 Analytical Models

There have been numerous analytical studies of the performance of CC schemes. An approximate mean value analysis method is used in [TaGS85] to analyze the performance of 2PL in a centralized database system. A similar approximate mean value analysis method is used to analyze 2PL, OPT and several hybrid concurrency control schemes in centralized databases in [YuDi92, YuDi93, YuDL93]. These analyses combine a data contention model with a conventional queueing model for hardware resource contention [Lave83] and iterate between the two models. Approximate mean value analysis and the iterative method are also used in [RyTh90a, RyTh90b, Thom93, Thom98, ThRy91] in order to analyze 2PL and OPT in a centralized database system and estimate the lock contention level at which dynamic 2PL starts thrashing. Gray uses approximate mean value analysis to estimate wait probability and deadlock rate in a fully-replicated distributed database, but assumes that replicate updates are performed sequentially [Gray96]. In [CDIY90, CiDY92], approximate mean value analysis is used to estimate the probability of conflict and response time of transactions using the OPT CC technique in a restricted form of distributed database: remote transactions execute entirely at a central site that replicates all data. In yet another form of distributed database, class 1 transactions use 2PL and execute entirely at the local primary-copy site; class 2 transactions use OPT and access data that is at a single known remote site [CiDY90]. Analytical models for 2PL in a distributed database are presented in [JeKT88, ShWo97], but the models do not permit simultaneous processing of a transaction at multiple sites. Basic timestamp ordering and multiversion timestamp ordering techniques have been analyzed in [Li87, ReTH96, Sing91a, Sing91b].

2.3 Concurrency Control based on Ordering

Database concurrency control based on totally-ordered-multicast has been proposed [NeTo93, ScRa96], but never implemented or simulated, and nothing is known about its performance. In [NeTo93], a communication primitive called a *publication* makes all recipients deliver a message at the same logical time called the *publication time*. A publication can be implemented in the following way: the sender broadcasts a message and solicits proposed publication times from all processors. The sender then picks the maximum proposed time and broadcasts it to everyone. Database concurrency control is proposed as a possible application, where processors *publish* their lock-requests and lock-releases. Since the lock-requests and lock-releases are totally ordered, there can be no deadlock.

In [ScRa96], a concurrency control technique based on totally-ordered-multicast and group communication [BiJo86] is proposed. In the proposed system, all sites at which a data item is replicated are members of the *group* corresponding to that data item. A transaction sends operations in a totally-ordered-multicast to all groups corresponding to the data items accessed. The total ordering of operations across groups ensures serializability of transactions. The proposed system does not address the issue of transactions that have dependencies among operations, or transactions that issue unilateral aborts. The paper does not provide an implementation for the property of total ordering of messages across groups, but refers to Totem [MMAB96] as a system that provides such a property. Finally, the paper does not address performance.

3 Ordering Network Aided CC

The *total ordering* property guarantees that messages are delivered in the same order at all destinations. This ordering guarantee can be exploited by a distributed database to ensure that transactions are viewed in the same order at all destinations. A network that provides total ordering at a low cost can be used as the basis for efficient concurrency control of transactions.

3.1 ORDER Algorithm

The ORDER CC technique is based on the total ordering property. A network that provides total ordering at a low cost is the *isotach network* [ReWW97, Will93]. An isotach network maintains *isotach logical time*, an extension of Lamport's logical time [Lamp78]. Isotach times are assigned to *send* and *deliver* events associated with a message, and are lexicographically ordered n-tuples of integers, of which the first component is called the *pulse*. The other components are *pid* and *rank*, and act as tie-breakers among events occurring in the same pulse. The *pid* is the identifier of the site that issued the message, and *rank = r* if the message is the r^{th} message issued by that site. In an isotach system, a site can control the logical time at which the messages that it sends are delivered, by controlling the logical time at which it sends messages.

An implementation of an isotach network is as follows. Every network switch has a *token manager* attached to one of its ports, and every host has a *switch interface unit* (SIU) connecting it to the nearest switch. Isotach logical time is implemented through the exchange of special messages called *tokens* by neighbouring token managers and SIUs. A token indicates that the sender has advanced its local logical clock. The system can achieve total ordering of messages if the destination SIUs reorder received messages according to *pulse, pid* and *rank*. The isotach network guarantees that a message is received at its destination SIU before the receipt of the token ending the pulse in which the message should be delivered. Token managers are critical to the scalability of an isotach network. Without them, every SIU would have to exchange tokens with every other SIU, which is clearly impractical for large networks. An isotach network has been simulated in software [Rege97] and implemented as a small hardware prototype [LaMy00]. Williams presents a method of executing atomic actions in an ordered and sequentially-consistent manner [Will93]. Other methods for guaranteeing total ordering include ISIS/Horus [BiJo86], Totem [MMAB96], Transis [DoMa96] and *publications* [NeTo93].

Given a fast and scalable implementation of the total ordering guarantee in the network, we can build an efficient CC technique. The cost of using an ordering mechanism can be divided into two components: the *latency penalty* (the additional delay before a message is received due to the ordering mechanism) and the *inherent ordering delay* (the additional delay before a received message becomes deliverable due to the need to wait for logically preceding messages). On a prototype isotach system optimized for large messages, the latency penalty is a factor of 2.31 to 1.43 (for large messages) times that of a conventional network [LaMy00]. The inherent ordering delay is insignificant on the prototype, but could be significant if the variance of network latency is high. If the latency penalty and inherent ordering delay are low, ORDER can potentially outperform a conventional CC technique like 2PL.

An ORDER system based on an isotach network, and derived from the isotach isochron scheduling technique [Will93] is presented below. A TM starts the execution of a transaction by issuing all the *read*s and *predeclare*s of the transaction as a single atomic action. The network delivers the atomic action in a total order using the isotach technique for implementing total ordering described above. Figure 2 shows the algorithms executed by various modules in an ORDER system.

```
TM Algorithm:
event: receive op from transaction;
   assert: op is a read/predeclare/write/commit/abort;
   if op == read/predeclare
      if op == last read/predeclare of transaction
         mark op as last_of_atomic_action;
```

```
      send op to network;
   else if op == write store op locally;
   else if op == commit/abort
      look up stored writes for the transaction;
      generate a commit/abort operation for each write;
      send commit/abort operations to network;
event: receive op from network;
   assert: op is a read-response;
   forward op to transaction;
```

Network Algorithm:

```
event: receive op from TM;
   assert: op is a read/predeclare/commit/abort;
   if op == read/predeclare
      if op is not marked last_of_atomic_action
         save op along with other operations from that transaction;
      else
         collect all saved operations from that transaction;
         assign a timestamp to the atomic action;
         add timestamp to all operations in the atomic action;
         deliver operations (in order) to destination CCS modules;
   else if op == commit/abort
      deliver operations (without ordering) to destination CCS modules;
event: receive op from CCS;
   assert: op is a read-response;
   deliver op (without ordering) to TM module;
```

CCS Algorithm:

```
event: receive op for object x from network;
   assert: op is a read/predeclare/commit/abort;
   assert: reads and predeclares arrive totally ordered;
   let queue = operation queue for object x;
   if op == read
      if queue is empty send op to DM;
      else if operation at tail(queue) is a commit
         send read-response with commit value;
      else append op to queue;
   else if op == predeclare
      append op to queue;
   else if op == abort
      let pos = position of corresponding predeclare on queue;
      delete the predeclare at pos;
      if pos == head(queue)
         while operation at head(queue) is a read/commit
            send operation at head(queue) to DM;
      else
         let src_op = first commit/predeclare ahead of pos;
         if src_op == commit
            while operation at pos(queue) == read
               send read-response with value of src_op;
               delete operation at pos(queue);
   else if op == commit
      let pos = position of corresponding predeclare on queue;
      replace predeclare with op;
      if pos == head(queue)
         while operation at head(queue) is a read/commit
            send operation at head(queue) to DM;
      else
         pos = pos + 1;
         while operation at pos(queue) is a read
            send read-response with value of op;
```

```
          delete operation at pos(queue);
event: receive op from DM; assert: op is a read-response;
   send op to network;
```

Figure 2: The ORDER System Algorithms

Each server buffers *read*s and *predeclare*s in queues corresponding to the object accessed, and immediately executes a *read* if it is at the head of a queue. As a transaction receives *read* responses, it issues *write*s corresponding to the previously-issued *predeclare*s. The TM does not send these *write*s across the network to the destinations, and stores them locally instead. When a transaction has received responses to all of its *read*s and has issued all of its *write*s, it sends out *commit*s (or *abort*s if it decides to perform a unilateral abort). The TM sends *commit*s (carrying the values of the corresponding *write*s) and *abort*s as regular messages rather than totally-ordered messages. The network delivers these *commit*s or *abort*s as quickly as it can, without ordering them with respect to other messages. On receipt of a *commit*, the destination finds the corresponding *predeclare* on its queue, replaces the *predeclare* with the *commit*, and executes any *commit*s and *read*s that are ready to be executed. On receipt of an *abort*, a destination finds the corresponding *predeclare* on a queue, deletes the *predeclare*, and executes any *commit*s and *read*s that are ready to be executed. A transaction is complete when all of its operations have been either committed or aborted.

3.2 Qualitative Comparison Between ORDER and Dynamic 2PL

An important difference between ORDER and dynamic 2PL is that in 2PL, a *read* lock is acquired as a result of a *read* operation and is held until the commit has been received by the server, whereas in ORDER, a *read* operation does not result in lock acquisition. 2PL and ORDER also differ with respect to deadlocks. Since deadlock cannot occur in ORDER, deadlock detection is unnecessary, while servers in 2PL incur deadlock detection costs. Moreover, when a deadlock occurs in 2PL, a transaction must be aborted and restarted, a cost never incurred in ORDER. In 2PL, the servers have to send explicit *lock-set* operations to the client in response to *write*s. The client in ORDER needs no such notification. On commit, a client in 2PL must send *lock-releases* to the servers in order to release the *read* locks. ORDER does not incur this cost since it does not use *read* locks.

ORDER incurs two that 2PL does not — the latency penalty and the inherent ordering delay. Moreover, a transaction is forced to predeclare all of its accesses and acquire all its *write* locks in the beginning. 2PL does not require predeclaration, and locks are acquired on demand. In order to reveal the influence of the above trade-offs on overall performance, we designed an analytical model and a simulation, and studied the performance of 2PL and ORDER under different workloads and system parameters.

4 Analytical Model

We present an analytical model of 2PL and ORDER in a fully distributed database. Our model uses an iterative method similar to some previous ones [YuDi92, CDIY90, TaGS85]. However, our model captures the high-data-contention scenario accurately. Our model combines a data contention model and a queueing model of hardware contention, and yields an expression for mean transaction response time. The probability of data conflict for any transaction depends on the CC technique and the transaction response time, which in turn depends on the conflict probability. For example, if lock contention probability increases, transaction response time increases due to additional lock waits. A longer transaction response time leads to a longer lock holding time, and hence to a higher lock contention probability. Our iterative approach captures the above dependency.

4.1 Assumptions

We assume an open system model with Poisson transaction arrivals. We assume that transaction operations arrive at each object as a Poisson process, and that access contention events for a transaction are independent. We model the network latency from an exponential distribution, and assume sufficient I/O bandwidth to enable modelling the I/O server as an infinite server with a load-independent service time. We assume that all transactions are of the same size, and that there are no inherent aborts. Finally, we assume that accesses to objects are distributed uniformly among all objects. With a non-uniform distribution, we would get higher contention at lower loads.

In order to make the analysis tractable, we make a few simplifying assumptions about 2PL. All locks are assumed to be requested at the start of the transaction and held until the end of the transaction, as in static 2PL. We assume that deadlocks and restarts don't occur, and that all locks are exclusive. The latter assumption is relaxed in a later section. For both 2PL and ORDER, we assume that the operation or lock request that spends the maximum amount of time waiting in a queue at the CCS module is the one whose response arrives the latest at the client that initiated the transaction. In the analytical model for ORDER, we assume that a *read* is satisfied only after it reaches the head of the CCS queue and is processed by the DM. Finally, we ignore the inherent ordering delay in the analysis of ORDER. We do not make these simplifying assumptions in our simulation. Table 1 shows the parameters we use in our model.

Number of clients	C	Number of servers	S
Number of objects in database	D	Number of accesses by transaction	K
Probability of read	P_R	Client MIPS	M_C
Server MIPS	M_S	Initial processing (instructions)	I_{INPL}
Computation per *read* (instructions)	I_{COMP}	TM overhead per operation (instructions)	I_{TM}
Network overhead per message (instructions)	I_{NW}	I/O overhead per access (instructions)	I_{IO}
CCS overhead per access (instructions)	I_{CC}	I/O delay per access (seconds)	D_{IO}
Avg. network latency per message (seconds)	D_{NW}	Mean transaction arrival rate per client (tps)	λ

Table 1: Analytical Model Parameters

4.2 Analytical Model of 2PL

The total response time of a transaction is modelled as: $R = R_{EXEC} + R_{CONT}$, where R_{CONT} is the longest time spent by any lock request of the transaction waiting for locks, and R_{EXEC} is the execution time of the transaction excluding this longest lock-wait-time. R_{EXEC} models hardware resource contention, and is estimated using queueing models, while R_{CONT} models data contention.

4.2.1 Data Contention R_{CONT}

Transactions arrive at each of the C clients at rate λ. Each transaction makes K lock requests, and the requests are uniformly distributed over the entire database of D objects. We assume lock requests arriving at an object form a Poisson process with mean $\lambda*C*K/D$, and all locks are exclusive. The K lock contention events for a single transaction are independent of one another.

Let the mean lock-holding time of an object be T_H. Since we assume that all locks are requested at the start of the transaction, T_H is approximately equal to R', where R' is the portion of the response time of a transaction when the transaction is holding at least one lock. We will compute R' in the next section. Let T_R be the mean remaining time that a lock request will have to wait for the current lock-holder to release the lock. $T_R = R' / 2$ [YDRI85].

The probability of contention for a lock request is the lock utilization ρ.

ρ = (arrival rate of lock requests at an object) / (service rate of lock requests at that object)
= $(\lambda * C * K / D) / (1 / T_H) = \lambda * K * T_H * C / D$ ..... (1)

In the following discussion, we use the term *queue length at an object* to mean the number of lock requests waiting to be processed at that object, plus the number of lock requests being processed at that object. At any object, the probability that the queue length < x is equal to $(1 - \rho^x)$. Let the probability that a transaction will not have to wait for any of its K locks be P_0.

Probability (queue length < 1 at all K objects) = $(1 - \rho^1)^K = P_0$

Probability (queue length < x at all K objects) = $(1 - \rho^x)^K$

Each of the K lock requests of a transaction has to wait before the lock is free and can be granted to the transaction. Our aim is to find the longest amount of time that one of these lock requests will have to wait. This longest wait time corresponds to the longest queue length among the queues at the K objects. Let the probability (longest queue length = 1) be P_1, the probability (longest queue length = 2) be P_2, and so on.

P_1 = Prob (queue length < 2 at all K objects) - Prob (queue length < 1 at all K objects)
= $(1 - \rho^2)^K - (1 - \rho)^K$

P_x = Prob (queue length < x+1 at all K objects) - Prob (queue length < x at all K objects)
= $(1 - \rho^{x+1})^K - (1 - \rho^x)^K$

Let $(P_0 + P_1 + P_2 + ... + P_n)$ be approximately equal to 1. Then the data contention wait for a transaction is approximately

$R_{CONT} = \{P_0 * 0\} + \{P_1 * T_R\} + \{P_2 * (T_R + 1 * T_H)\} + ... + \{P_n * (T_R + (n-1) * T_H)\}$ ..... (2)

Previous models have assumed that the queue length at any object is never greater than one. With this assumption, the effect of transaction t_1 waiting at object o_1 for transaction t_2 while transaction t_2 is waiting at a different object o_2 for transaction t_3, **is** captured; but the effect of t_1 waiting at o_1 for t_2, while t_2 is waiting at the same object o_1 for t_3, **is not** captured. Since queue lengths increase as data contention increases, traditional analytical models are inadequate to model R_{CONT} at high data contention levels. Our analysis, on the other hand, is able to capture response time at high data contention levels.

4.2.2 Hardware Resource Contention R_{EXEC}

We first compute the utilizations at the client and the servers. In order to execute a transaction, the number of instructions to be executed at the client is

$I_c = I_{INPL} + K*(I_{TM}+I_{NW}) + K*(I_{TM}+I_{NW}) + K*P_R*I_{COMP} + K*(I_{TM}+I_{NW})$

where I_{INPL} accounts for the initial processing; the three $K*(I_{TM}+I_{NW})$ terms account for the Transaction Manager and network overhead on lock requests, lock responses and *commit*s/*abort*s respectively; and $K*P_R*I_{COMP}$ accounts for the computation in response to the *read*s.

Arrival rate at the client = λ transactions per second

Service rate at the client = $\mu_c = (M_C / I_C) * 10^6$ transactions per second

Client Utilization $\rho_c = \lambda / \mu_c = (\lambda * I_C) / (M_C * 10^6)$

Processing time at the client = $R_{CL} = (1/\mu_c) / (1-\rho_c)$ seconds

In order to execute a transaction, the total number of instructions executed at the servers is

$I_s = 3*K*(I_{NW}+I_{CC}) + K*I_{IO}$

where $3*K*(I_{NW}+I_{CC})$ accounts for the network and CCS module overhead of processing lock requests, responses and *commit*s/*abort*s; and $K*I_{IO}$ accounts for the I/O overhead of database *read*s and *write*s.

Arrival rate at a server = $\lambda * C / S$ transactions per second

Service rate at a server = $\mu_s = (M_S / I_S) * 10^6$ transactions per second

Server Utilization $\rho_s = (\lambda * C / S) / \mu_s = (\lambda * C / S * I_S) / (M_S * 10^6)$

Processing time at a server to do a transaction's worth of work = $R_{SV} = (1/\mu_s) / (1-\rho_s)$

There are two phases in the execution of a transaction. In the first phase, the client sends lock requests to the servers, the servers process the lock requests and send lock responses, and the client

performs transaction computation. A *read* lock request results in a database access at the server, while a *write* lock request does not. Therefore, we assume that the response time in the first phase is dominated by the response time of the *read* lock requests. We also assume that the first phase response time is dominated by the lock request *l* that has to wait for the maximum amount of time on a CCS module queue. The response time for the transaction in this first phase is given by

R_{PHASE1}= R_{CL1} + (network traversal time of *l* + server-and-disk processing time of *l* + network traversal time of *l*'s response + client processing time of *l*'s response) + R_{COMP}; where

R_{CL1} = R_{CL} * [I_{INPL} + K*(I_{TM}+I_{NW})]/I_c (initial processing and sending of lock requests); and

R_{COMP} = R_{CL} * (K*P_R*I_{COMP})/I_c (transaction computation in response to *read*s).

R_{PHASE1}= R_{CL1} + D_{NW} + R_{SV}*(I_{NW}+I_{CC})/I_s + R_{CONT} + R_{SV}*I_{IO}/I_s + D_{IO} + R_{SV}*(I_{CC}+I_{NW})/I_s + D_{NW} + R_{CL} * (I_{NW}+I_{TM})/I_c + R_{COMP}

R_{PHASE1} has a data contention component (R_{CONT}) and a hardware contention component R'_{PHASE1}.

R'_{PHASE1}=R_{CL1} + D_{NW} + R_{SV}*(I_{NW}+I_{CC})/I_s + R_{SV}*I_{IO}/I_s + D_{IO} + R_{SV}*(I_{CC}+I_{NW})/I_s + D_{NW} + R_{CL} * (I_{NW}+I_{TM})/I_c + R_{COMP}

In the second phase of execution of a transaction, the client sends *commit*s to the servers. A committed *write* results in a database access at the server, while a committed *read* does not. We assume that the response time in the second phase is dominated by the response time of the committed *write*s. The response time for the transaction in the second phase is given by

R_{PHASE2}= R_{CL2} + D_{NW}*(1/1 + 1/2 + ... + 1/K) + R_{SV}*(I_{NW}+I_{CC})/I_s + D_{IO} + R_{SV}*I_{IO}/I_s; where

R_{CL2} = R_{CL} * K*(I_{TM}+I_{NW})/I_c (sending of *commit*s and *lock-release*s), and

D_{NW}*(1/1 + 1/2 + ... + 1/K) = network traversal time of the slowest of the K operations.

The hardware component of the response time of a transaction is R_{EXEC} = R'_{PHASE1} + R_{PHASE2}, and can be computed directly from the parameters to the analytical model.

4.2.3 Total Response Time R

The hardware contention component R_{EXEC} is computed as described in the previous section. The data contention component R_{CONT} depends on the mean lock-holding time T_H and the mean remaining lock-holding time T_R. Recall that T_H is approximately equal to R', where R' is the portion of the response time of a transaction when the transaction is holding at least one lock.

R' = R - [portion of response time when transaction is holding no locks]

= R - [R_{CL1} + D_{NW} + R_{SV}*(I_{NW}+I_{CC}+I_{IO})/I_S + D_{IO}] ..... (3)

We use an iterative model in order to compute mean transaction response time R. We start with an initial value of zero for R_{CONT}.

Step 1: R = R_{EXEC} + R_{CONT}

Step 2: Compute R' according to equation 3.

Step 3: T_H = R'

Step 4: T_R = R' /2

Step 5: Compute ρ according to equation 1. If ρ >= 1.0, stop; the system is unstable.

Step 6: Compute R_{CONT} according to equation 2.

Step 7: If R_{CONT} has not changed by a significant amount, stop; else go to Step 1.

The iterative process continues until the computation of R_{CONT} has converged; that is, until the difference between successive-iteration values of R_{CONT} is very small. The mean transaction response time is then computed as R = R_{EXEC} + R_{CONT}.

4.2.4 Modelling Shared and Exclusive Locks

Allowing shared locks in addition to exclusive locks changes the behaviour of the lock-request queues at the CCS modules in the servers. Since multiple transactions can hold shared locks (*read* locks) simultaneously, a lock-request queue behaves as if contiguous *read*s are compressed into a single *read*. Consequently, *effective* queue lengths at the servers can be smaller than actual queue lengths. If the actual queue length is *m*, the expected number of *read*s on the queue is P_Rm, and the

expected number of *write*s is $(1-P_R)m$. In order to make our model tractable, we make the following simplifying assumption. If $P_R > 0.5$, then the contiguous *read* sequences are uniformly spaced among the *write*s. And if $P_R <= 0.5$, then the *write*s are uniformly spaced among the *read*s. With this assumption, if the actual queue length is *m*, the corresponding effective queue length is

$Q_m' = 2*(1-P_R)*m$, if $P_R > 0.5$;

$Q_m' = m$, if $P_R <= 0.5$.

Therefore, equation 2 is replaced with

$$R_{CONT}=\{P_0*0\} + \{P_1*T_R\} + \{P_2*(T_R+(Q_2'-1)*T_H)\} + \{P_3*(T_R+(Q_3'-1)*T_H)\} + ... + \{P_n*(T_R+(Q_n'-1)*T_H)\} \quad (2')$$

4.3 Analytical Model of ORDER

As in the 2PL model, the total response time of a transaction is modelled as the sum of a hardware resource contention component R_{EXEC} and a data contention component R_{CONT}. R_{CONT} is the longest time spent by any operation of the transaction in the CCS module, waiting to be processed.

4.3.1 Data Contention R_{CONT}

We assume that operations arriving at an object form a Poisson process with mean $\lambda*C*K/D$. Let the mean "object-holding time" be T_H. The object-holding time of an object is the amount of time for which an operation effectively "locks" the object, disallowing access by other operations. In the ORDER algorithm, the object-holding time of a *read* is zero, because a *read* is immediately executed when it reaches the head of the CCS queue. On the other hand, a *predeclare* has an object-holding time of R', because it holds the object from the start of the transaction until the corresponding committed *write* arrives. The object utilization in the ORDER system differs from the 2PL lock utilization (equation 1) as follows.

ρ = (arrival rate of predeclares at an object) / (service rate of predeclares at that object)

$= (\lambda * C * K * (1 - P_R) / D) / (1 / T_H) = \lambda * K * (1 - P_R) * T_H * C / D$ ..... (1')

The data contention is derived in the same way as in §4.2.1, except that the term *queue length at an object* now refers to the number of predeclares at that object. R_{CONT} is still described by equation 2.

4.3.2 Hardware Resource Contention R_{EXEC}

In the ORDER algorithm, the number of instructions to be executed at the client differs from that in 2PL, because of three reasons: the server does not send *lock-set* messages to the client; *write*s are stored locally; and the client does not send *read lock-release*s to the servers. In order to execute a transaction, the number of instructions to be executed at the client in an ORDER system is

$$I_c = I_{INPL} + K*(I_{TM}+I_{NW}) + K*P_R*(I_{TM}+I_{NW}) + K*(1-P_R)*I_{TM} + K*P_R*I_{COMP} + K*(1-P_R)*(I_{TM}+I_{NW})$$

where I_{INPL} accounts for the initial processing; $K*(I_{TM}+I_{NW})$ accounts for the TM and network overhead on *read*s and *predeclare*s; $K*P_R*(I_{TM}+I_{NW})$ accounts for the TM and network overhead on *read-response*s; $K*(1-P_R)*I_{TM}$ accounts for the TM overhead on *write*s; $K*P_R*I_{COMP}$ accounts for the computation in response to the *read*s; and $K*(1-P_R)*(I_{TM}+I_{NW})$ accounts for the TM and network overhead on *commit*s.

In contrast to a 2PL system, the servers in an ORDER system do not send *lock-set* messages, nor do they receive and process *read lock-release*s. In order to execute a transaction, the total number of instructions executed at the servers is

$$I_s = K*(I_{NW}+I_{CC}) + K*P_R*(I_{NW}+I_{CC}) + K*(1-P_R)*(I_{NW}+I_{CC}) + K*I_{IO}$$

where $K*(I_{NW}+I_{CC})$ accounts for the network and CCS module overhead of processing *read*s and *predeclare*s; $K*P_R*(I_{NW}+I_{CC})$ accounts for the network and CCS module overhead of processing *read-response*s; $K*(1-P_R)*(I_{NW}+I_{CC})$ accounts for the network and CCS module overhead of

processing *commit*s; and $K*I_{IO}$ accounts for the I/O overhead in performing the database *read*s and *write*s.

R_{PHASE1} and R_{PHASE2} are computed in the same manner as in §4.2.2, except that the R_{CL2} component of R_{PHASE2} changes to include the local storage of *write*s and exclude the sending of *read lock-release*s:

$R_{CL2} = R_{CL} * \{K*(1-P_R)*I_{TM} + K*(1-P_R)*(I_{TM}+I_{NW})/I_c\}$ (storage of *write*s and sending of *commit*s)

The average network latency D_{NW} in an ORDER system will probably be higher than the D_{NW} of the conventional network used by 2PL, because the network in an ORDER system is doing extra work in order to deliver messages in order. The severity of this latency penalty affects the difference in performance between 2PL and ORDER.

4.4 Analytical Model of Best Case

We use the No Data Contention (NDC) algorithm as the best case algorithm that assumes a zero ordering cost and no data conflicts. NDC allows us to isolate the effects of hardware contention from data contention and ordering cost. Note that NDC guarantees a correct execution only in the absence of data contention. The model of NDC is the same as that of ORDER, except that the data contention component of response time (R_{CONT}) is always zero.

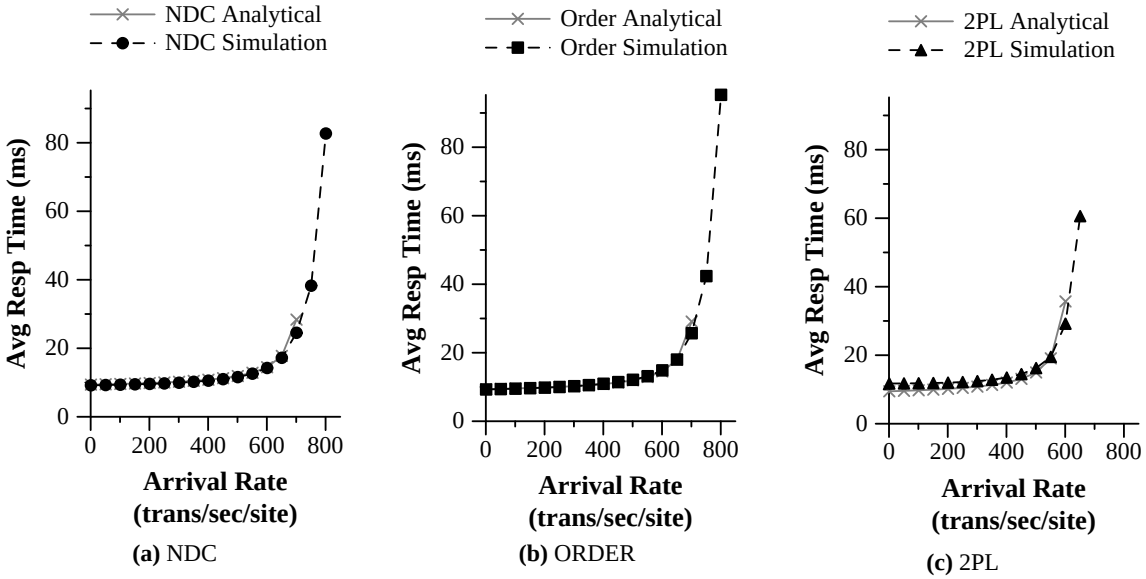
5 Simulation

We simulated the NDC, ORDER and dynamic 2PL systems and performed detailed performance studies. Servers in the 2PL simulation detect deadlocks using waits-for graphs, but the cost for deadlock detection is set to zero. When a transaction is aborted in order to resolve deadlock, a new transaction is started in order to simulate a restart. Locks can be shared or exclusive, with no upgrade from shared to exclusive. In the ORDER simulation, a *read* can be satisfied by a committed *write* in front of it on the CCS queue. Note that the inherent ordering delay (which was ignored in the analytical model) comes into effect in the ORDER simulation, because the network can delay messages so that they are delivered according to a total order. We model the latency penalty by multiplying the conventional network's average latency by a *latency penalty factor*. Transactions are of variable size. The baseline parameters that we used are standard ones culled from performance studies of CC techniques in the literature.

- Client's initial processing of a transaction (I_{INPL}) = 100K instructions
- Client's computation per *read* (I_{COMP}) = 20K instructions
- Transaction Manager overhead at client per operation (I_{TM}) = 1K instructions
- Network overhead at client/server per message (I_{NW}) = 5K instructions
- I/O overhead at server per object access (I_{IO}) = 5K instructions
- Overhead at CC module at server per object access (I_{CC}) = 1K instructions
- Number of clients = 8
- Number of servers = 8
- Database size = 32000 or 4000 objects
- Number of high-contention objects (hot spots) = $1/20^{th}$ of database size
- High-contention access probability = 0.33
- Size of a transaction = 8-24 object accesses (uniform distribution)
- Probability of a transaction's access being a *read* = 0.75
- Commit probability of a transaction = 1
- MIPS rating of each client = 400
- MIPS rating of each server = 800
- Average communication delay per message (D_{NW}) = $20 * 10^{-6}$ seconds
- Latency penalty factor (for ORDER) = 2
- I/O delay per object access (D_{IO}) = $4 * 10^{-3}$ seconds
- Mean arrival rate of transactions at each client (λ) = 201 transactions per second
- Percentage of read-only transactions = 0

- Percentage of write-only transactions = 0
- Local deadlock detection period (for 2PL) = every 10 operations arriving at CCS module
- Global deadlock detection (for 2PL) = 100 times a second
- Deadlock detection cost (for 2PL) = 0

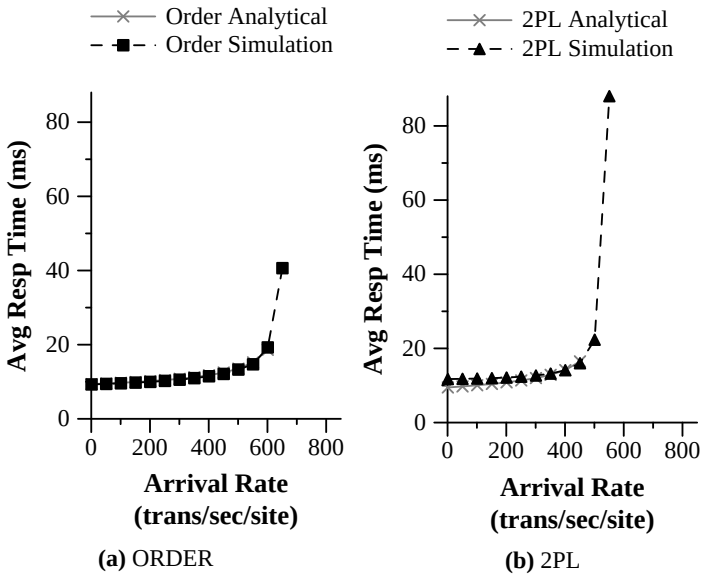
We ran the simulation and the analytical model for different arrival rates of transactions and for different database sizes, and measured the average transaction response time. Figure 3 and Figure 4 show the results for NDC, ORDER and 2PL, for two database sizes: 32000 and 4000 objects. The smaller the database the higher the degree of data contention, because the same number of transactions are contending for a smaller number of objects. Henceforth in this paper, the database size = 32000 case will be referred to as the low data contention scenario, and the database size = 4000 case will be referred to as the high data contention scenario. Since data contention has no effect on NDC, there is only one graph for NDC.



(a) NDC **(b)** ORDER **(c)** 2PL

Figure 3: Database size = 32000 objects (low data contention)

In this experiment, the latency penalty factor for ORDER was set to 2, and no hot spots were used in the simulation or in the analytical model. Each point in the graphs is an average over 6 independent runs. For all three algorithms, as transaction arrival rate increases, response time increases slowly until a *knee* when it shoots up to a high value. In the NDC case, the performance degradation at the knee is due to hardware resource contention: the increased transactions compete for the limited CPU resources available. At high data contention levels, the knees of the ORDER and 2PL curves occur at a lower transaction arrival rate.



(a) ORDER **(b)** 2PL

Figure 4: Database size = 4000 objects (high data contention)

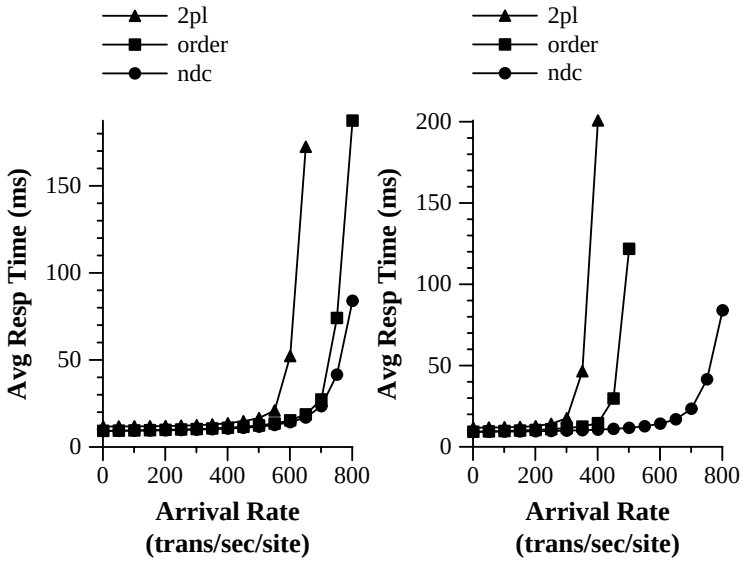
As more and more transactions are introduced into the system, the data conflict probability increases, causing longer CCS queue wait times and longer response times. The data conflict probability is higher in the 4000-object database because the same number of operations are competing for access to a smaller set of objects. The graphs show that our analytical model tracks the simulation results very well, and predicts the knee of the curve accurately even at high data contention levels. As noted, the analytical model makes several simplifying assumptions not made in the simulation. The close agreement between the results of the analytical model and simulation indicates that the effect of these assumptions is either insignificant or self-cancelling. Future work will explore the effect of each assumption.

6 Performance Evaluation

We ran extensive experiments on our simulations, and studied performance for various workloads and system parameters. We used a b-c pattern of hot spot access, where a fraction *b* of object accesses go to a fraction *c* of the database (*hot spots*), and *b* > *c* [TaGS85].

6.1 Transaction arrival rate

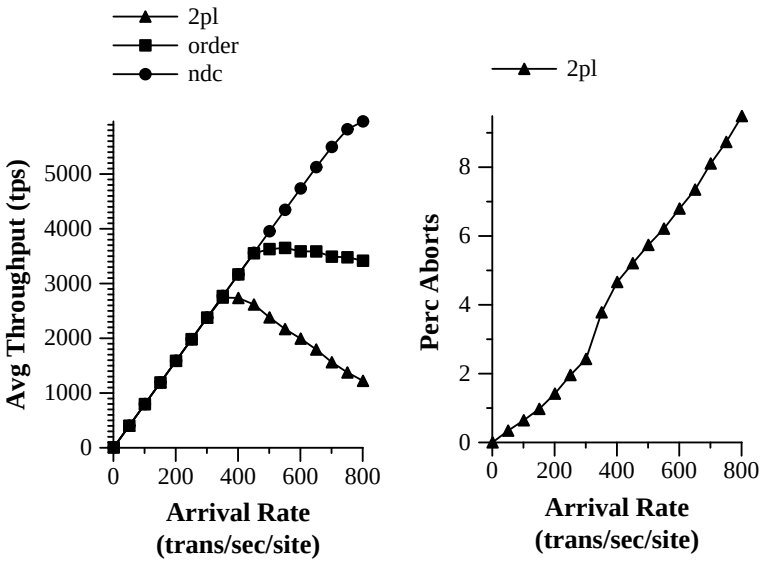
Figure 5 shows the effect of varying the arrival rate of transactions. As the arrival rate of transactions increases, the response time increases for all three schemes, hitting a *knee* and then going rapidly up as the system goes into an unstable region. The response time graphs show that the knee of the curve occurs at a lower arrival rate for 2PL than for ORDER. The sharp increase in the response time of NDC is due to hardware resource contention. For ORDER and 2PL, the sharp degradation in performance occurs at a lower arrival rate than for NDC because of data contention.



(a) Low data contention **(b)** High data contention

Figure 5: Effect of transaction arrival rate

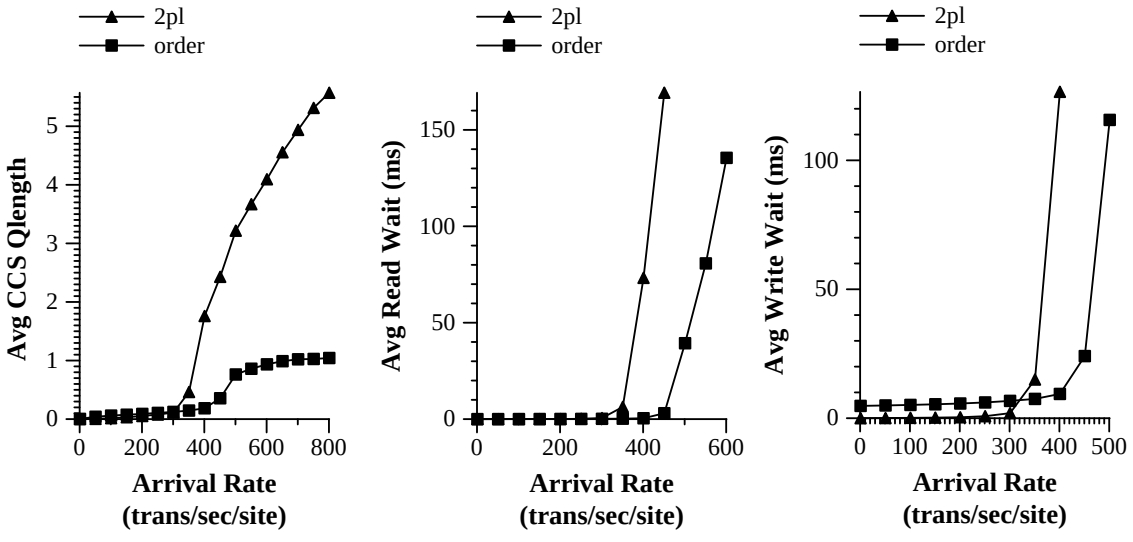
Figure 6 shows the throughput and percentage aborts curves for the database size = 4000 case. The throughput graph shows that the peak throughput is reached at the knee arrival rate, after which the throughput levels off and drops as the system becomes unstable. Aborts increase steadily in the 2PL system as transaction arrival rate increases, since increased response time means longer lock-holding times, which in turn means a higher probability for deadlock.



(a) Throughput **(b)** Percentage aborts

Figure 6: Throughput and aborts (high data contention)

Figure 7 shows that modelling queue lengths accurately is important for predicting performance under high contention. The graphs plot average queue length at the CCS modules, average *read* wait and average *write* wait for the high data contention case. At the knee arrival rate, average queue length increases sharply, causing *read* and *write/predeclare* operations to wait longer, increasing response time. The increased response time in turn causes the mean lock-hold time to increase, causing operations to wait longer, further increasing response time. The average queue length of 2PL increases beyond one when the system reaches the knee, showing that the queue length = 1 assumption made in traditional analytical models for 2PL is insufficient in order to predict the knee of the curve. The queue length in 2PL is initially lower than that in ORDER because 2PL staggers its request of locks while ORDER requests all of its locks in the beginning. However, when increased data contention sets in, the lower lock-holding time of ORDER keeps the queue length and operation wait time low.

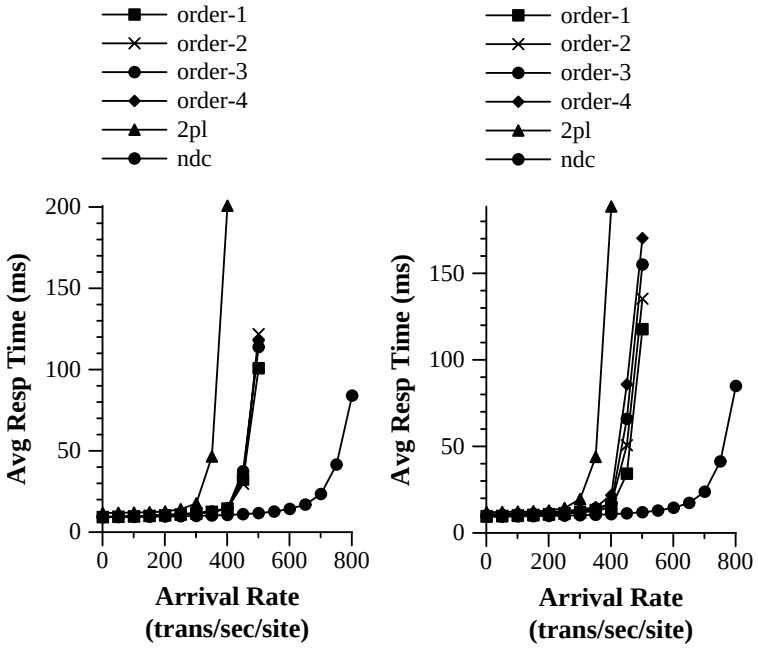


(a) Average CCS queue length **(b)** Average read wait **(c)** Average write wait

Figure 7: Data contention wait curves

6.2 Network latency and ordering cost

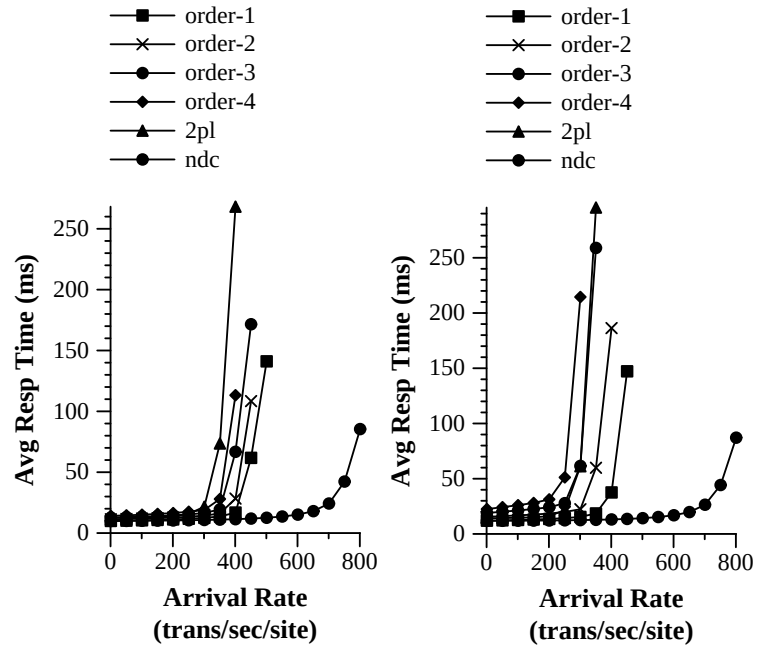
Figure 8 shows the effect of varying the average network latency and the latency penalty factor (for ORDER) in the high data contention scenario. The ORDER technique is represented by four different curves, each representing a different latency penalty factor. The *order-2* curve uses a latency penalty factor of 2, and corresponds to the *ORDER* curves presented in earlier graphs. As network latency increases, operations arrive later, causing lock-holding times and object-holding times to increase, in turn causing higher response times. The increased data contention at high network latencies causes both ORDER and 2PL to peak at lower arrival rates.



(a) Network latency = 20µs **(b)** Network latency = 80µs

Figure 8: Effect of network latency and ordering penalty.

In order to discover how efficient the network has to be in providing the property of total ordering, we studied the effect of four different latency penalty factors — 1, 2, 3 and 4 — on ORDER. Recall that the latency penalty factor of the isotach prototype is 1.43-2.31. The higher the latency penalty factor, the longer operations take to arrive, the longer the object-holding times, and the longer the response time of ORDER. In addition, in our experiments, since network latency is drawn from an exponential distribution, the higher the mean latency, the higher the variance in latency. In an ORDER system, a high latency variance means that the probability of operations arriving out of order is high, and therefore, the inherent ordering delay is high. However, the inherent ordering delay is still a smaller contributor to the overall response time than the object-



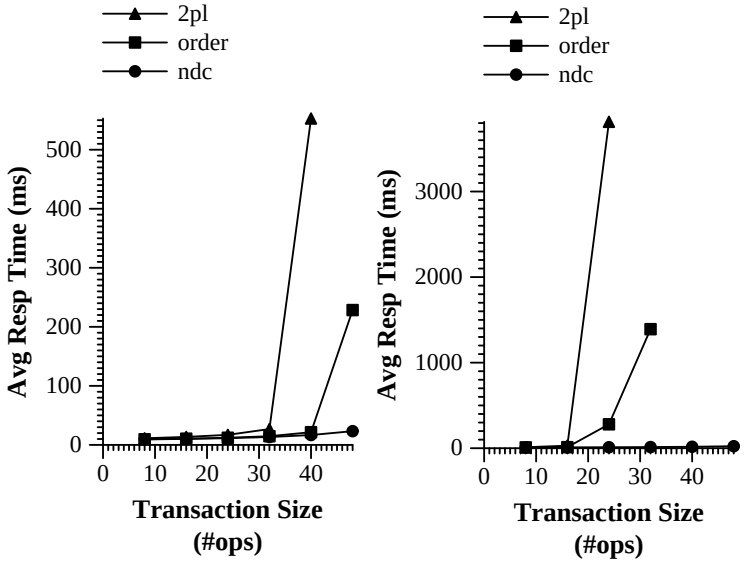
(c) Network latency = 200µs **(d)** Network latency = 500µs

Figure 8: Effect of network latency and ordering penalty.

holding times. The effect of varying the latency penalty factor becomes more apparent in the graphs for higher network latencies. At latencies of 20µs, 80µs and 200µs, all of the ORDER variants perform better than 2PL. At a latency of 500µs, order-1 and order-2 still outperform 2PL, but the performance of order-3 is similar to that of 2PL. Moreover, order-4 performs worse than 2PL. In summary, ORDER outperforms 2PL at low to moderate network latencies. When the average network latency is high (500µs), the network must provide the total ordering guarantee in a very efficient way in order to outperform 2PL.

6.3 Transaction size

Figure 9 shows the effect of varying the transaction size. As the transaction size increases, the number of operations active in the system at any time increases, thus increasing data conflict. In addition, a bigger transaction has a longer lifetime, thus increasing data conflict and the probability of deadlock in 2PL. As before, the lower lock-holding time of ORDER keeps the queue length and operation wait time of ORDER lower than in 2PL.



(a) Low data contention **(b)** High data contention

Figure 9: Effect of transaction size

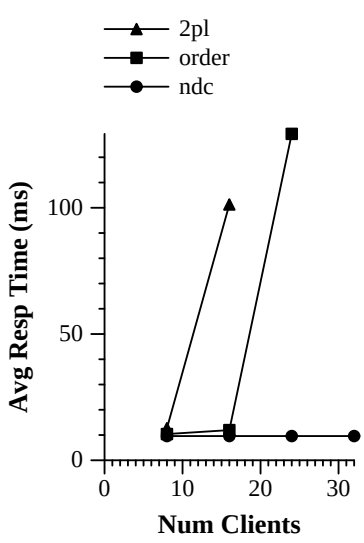


Figure 10: Number of clients (high data contention)

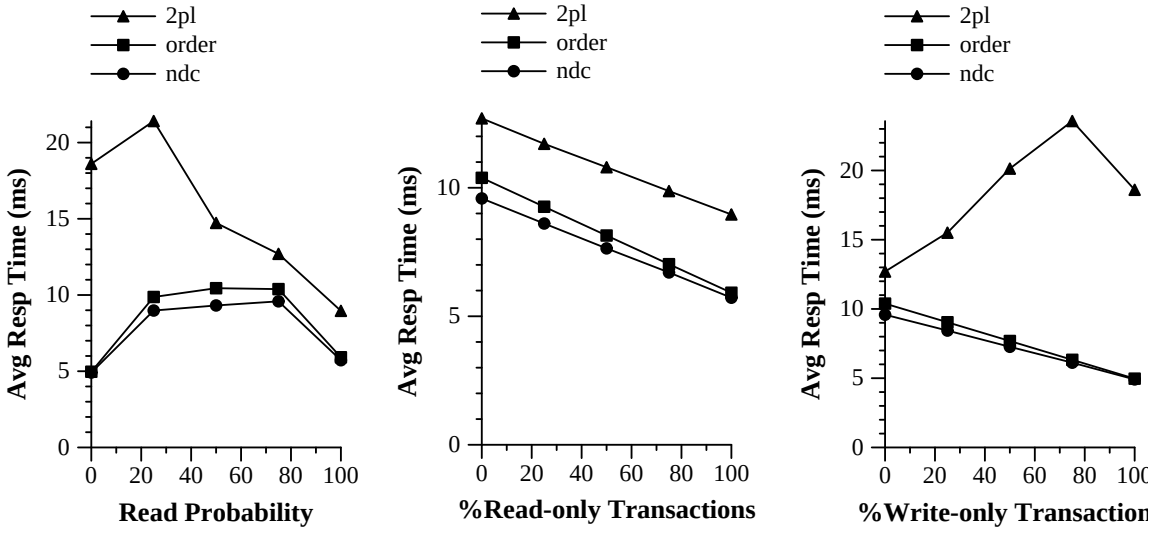
6.4 Number of clients

Figure 10 shows the effect of varying the number of clients submitting transactions. As the number of clients increases, the number of transactions active in the system increases, and data conflict increases. The probability of deadlock in 2PL increases, and the percentage of aborts consequently increases. Once again, the queue length, operation wait time and response time curves for 2PL have knees at a lower value for number of clients, as compared to ORDER.

6.5 Transaction composition

Figure 11 (a) shows the effect of varying the probability of read while transaction size is held constant. More *read*s implies more computation by the transaction, and more time before the commit decision can be made (for all three schemes). This effect dominates in the earlier portion of the curves. However, more *read*s also implies that the queue length (and lock-holding time) goes down because *read* locks are shared (2PL) and *read*s can be satisfied earlier (ORDER). When

the probability of *read* becomes 100%, there are no *write*s, and therefore the disk access time of the second phase goes away for all 3 schemes. Hence the sharp drop in response time.



(a) Read probability **(b)** Read-only transactions **(c)** Write-only transactions

Figure 11: Transaction composition (high data contention)

Figure 11 (b) shows the effect of varying the percentage of read-only transactions. For read-only transactions, the second phase involving disk writes is absent. In addition, more *read*s implies lower lock-holding time and queue lengths. On the other hand, more *read*s implies more transaction computation. But the first two effects dominate, and response time decreases as the percentage of read-only transactions increases (for all three schemes).

Figure 11 (c) shows the effect of varying the percentage of write-only transactions. In ORDER, write-only transactions have an advantage over read-write transactions because no communication from servers to client is necessary in order for the client to make its commit decision. 2PL offers no such advantage because the client must wait for lock-sets before committing. Another advantage of write-only transactions (for both schemes) is that they perform no computation. Therefore, the response time for NDC and ORDER go down as the percentage of write-only transactions increases. However, for 2PL, the increased *write*s heighten the already-serious data conflict situation, causing queue length, operation wait time, response time and percentage of aborts to go up.

7 Conclusion

The conventional 2PL CC technique causes system thrashing at high data contention levels, restricting transaction throughput to levels inconsistent with the available resources. We have presented a new concurrency control technique called ORDER, that uses a total ordering guarantee provided by the network in order to achieve efficient CC. We have presented a new analytical model for 2PL. Unlike previously-proposed 2PL models, our analytical model continues to predict performance accurately even at high data contention levels.

Our analytical model and simulation agree in predicting that ORDER outperforms dynamic 2PL under high data contention. ORDER's advantage disappears only when network latency is high and ordering is implemented inefficiently. The performance of the isotach prototype implies that ORDER is a good candidate for high contention databases. In both ORDER and 2PL, as parameters change adversely, queue lengths increase, increasing operation wait time and response time. In 2PL, aborts also increase due to deadlocks, further increasing response time. The mean lock-holding time in ORDER is lower than that in 2PL, letting ORDER sustain good performance longer. We have demonstrated that ORDER outperforms 2PL for a wide range of workloads. In

future work, we plan to study the effect of high-variance network latencies and recovery techniques on ORDER and 2PL.

8 References

AlDa76 Alsberg P. A. and Day J. D., A Principle for Resilient Sharing of Distributed Resources, Proceedings of the 2nd International Conference on Software Engg., Oct 1976.

Bada79 Badal D. Z., Correctness of Concurrency Control and Implications in Distributed Databases, Proceedings of COMPSAC 79 Conference, Nov 1979.

BaHR80 Bayer R., Heller H. and Reiser A., Parallelism and Recovery in Database Systems, ACM Transactions on Database Systems 5/2, Jun 1980.

BeGo80 Bernstein P.A. and Goodman N., Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems, Proceedings of the 6th International Conference on Very Large Databases, Oct 1980.

BeGo81 Bernstein P. and Goodman N., Concurrency Control in Distributed Database Systems, ACM Computing Surveys 13/2, Jun 1981.

BeHG87 Bernstein P. A., Hadzilacos V. and Goodman N., Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

BeSR80 Bernstein P. A., Shipman D. W. and Rothnie J. B., Concurrency Control in a System for Distributed Databases (SDD-1), ACM Transactions on Database Systems 5/1, Mar 1980.

BGRP78 Bernstein P. A., Goodman N., Rothnie J. B. and Papadimitriou C. A., The Concurrency Control Mechanism of SDD-1: a System for Distributed Databases (the fully redundant case), IEEE Transactions on Software Engg., SE-4/3, May 1978.

BiJo86 Birman K. P. and Joseph T. A., Low-Cost Management of Replicated Data in Fault-Tolerant Distributed Systems, ACM Transactions on Computer Systems 4/1, Feb 1986.

CaLi91 Carey M. and Livny, Conflict Detection Tradeoffs for Replicated Data, ACM Transactions on Database Systems 16/4, Dec 1991.

Casa79 Casanova M. A., The Concurrency Control Problem for Database Systems, PhD dissertation, Harvard University; Technical Report TR-17-79, Center for Research in Computing Technology, 1979.

CiDY90 Ciciani B., Dias D., Yu P., Analysis of Replication in Distributed Database Systems, IEEE TKDE 2/2, Jun 1990, pp 247-261.

CiDY92 Ciciani B., Dias D., Yu P., Analysis of Concurrency-Coherency Control Protocols for Distributed Transaction Processing Systems with Regional Locality, IEEE TSE 18/10, Oct '92, pp 889-914.

CDIY90 Ciciani B., Dias D., Iyer B., Yu P., A Hybrid Distributed Centralized System Structure for Transaction Processing, IEEE TSE 16/8, 1990, pp 791-806.

Date95 Date C. J., An Introduction to Database Systems, Sixth Edition, Addison-Wesley, 1995.

DoMa96 Dolev D. and Malkhi D., The Transis Approach to High Availability Cluster Communication, CACM 39/4, April 1996, pp 64-70.

EGLT76 Eswaran K. P., Gray J. N., Lorie R. A. and Traiger I. L., The Notions of Consistency and Predicate Locks in a Database Systems, Communications of the ACM 19/11, Nov 1976.

Garc79 Garcia-Molina H., Performance of Update Algorithms for Replicated Data in a Distributed Database, PhD dissertation, Computer Science Dept., Stanford University, Jun 1979.

Gray96 Gray J., The Dangers of Replication and a Solution, ACM SIGMOD Conf., 1996, pp 173-182.

GrRe92 Gray J. N. and Reuter A., Transaction Processing: Concepts and Facilities, Morgan-Kaufmann.

HaDo91 Halici U. and Dogac A., An Optimistic Locking Technique For Concurrency Control in Distributed Databases, TSE 17/7, Jul 1991.

Holt72 Holt R. C., Some Deadlock Properties of Computer Systems, ACM Computing Surveys 4/3, Dec 1972.

JeKT88 Jenq B., Kohler W. H. and Towsley D., A Queueing Network Model for a Distributed Database Testbed System, IEEE Transactions on Software Engineering 14/7, Jul 1988.

KiCo74 King P. F. and Collmeyer A. J., Database Sharing - an Efficient Method for Supporting Concurrent Processes, Proceedings of the 1974 National Computer Conference 42, 1974.

LaMy00 Lack M. N. and Myers, P., The Isotach Messaging Layer: Ironman Design, Technical Report CS-2000-17, Dept. of Computer Science, University of Virginia, May 2000.

Lamp78 Lamport L., Time, Clocks and Ordering of Events in a Distributed System, Communications of the ACM 21/7, Jul 1978.

Lave83 Lavenberg S. (Ed.), Computer Performance Modeling Handbook, Academic Press, Orlando, Florida, 1983.

Li87 Li V., Performance Models of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases, IEEE TOC 36/9, Sept 1987.

MMAB96 Moser L. E., Melliar-Smith P. M., Agarwal D. A., Budhia R. K. and Lingley-Papadopoulos C. A., Totem: a Fault-Tolerant Multicast Group Communication System, Communications of the ACM 39/4, Apr 1996.

NeTo93 Neiger G. and Toueg S., Simulating synchronized clocks and common knowledge in distributed systems, Journal of the ACM 40/2, Apr 1993.

Reed78 Reed D. P., Naming and Synchronization in a Decentralized Computer System, PhD dissertation, Dept. of Electrical Engg., MIT, Sep 1978.

Rege97 Regehr J., An Isotach Implementation for Myrinet, Technical Report CS-97-12, Dept. of Computer Science, University of Virginia, May 1997.

ReTH96 Ren J., Takahashi Y., Hasegawa T., Analysis of Impact of Network Delay on Multiversion Conservative Timestamp Algorithms in DDBS, Perf Eval 26, 1996, pp 21-50.

ReWW97 Reynolds P. F., Williams C. and Wagner R., IEEE Transactions on Parallel and Distributed Systems 8/4, Apr. 1997, pp 337-348.

RoSL78 Rosenkrantz D. J., Stearns R. E. and Lewis P. M., System Level Concurrency Control for Distributed Database Systems, ACM Transactions on Database Systems 3/2, Jun 1978.

RyTh90a Ryu I., Thomasian A., Analysis of Database Performance with Dynamic Locking, JACM 37/3, Jul 1990, pp 491-523.

RyTh90b Ryu I., Thomasian A., Performance Analysis of Dynamic Locking with the No-Waiting Policy, IEEE TSE 16/7, Jul 1990, pp 684-698.

SaGS94 Salem K., Garcia-Molina H. and Shands J., Altruistic Locking, ACM Transactions on Database Systems 19/1, Mar 1994.

ScRa96 Schiper A. and Raynal M., From group communication to transactions in distributed systems, Communications of the ACM 38/4, Apr 1996.

ShMi77a Shapiro R. M. and Millstein R. E., Reliability and Fault Recovery in Distributed Processing, Oceans 77 Conference record, vol II, 1977.

ShMi77b Shapiro R. M. and Millstein R. E., NSW Reliability Plan, Massachusetts Technical Report 7701-1411, Computer Associates, Wakefield, MA, Jun 1977.

ShWo97 Sheikh F. and Woodside M., Layered Analytic Performance Modelling of a Distributed Database System, Proceedings of the 17the International Conference on Distributed Computing Systems, May 1997.

Sing91a Singhal M., Performance Analysis of the Basic Timestamp Ordering Algorithm via Markov Modeling, Perf Eval 12, 1991.

Sing91b Singhal M., Analysis of the Probability of Transaction Abort and Throughput of Two Timestamp Ordering Algorithms for Database Systems, IEEE TKDE 3/2, Jun 1991.

StRo81 Stearns R. E. and Rosenkrantz D. J., Distributed Database Concurrency Controls Using Before-Values, Proceedings of the ACM-SIGMOD Conference on Management of Data, 1981.

Ston79 Stonebraker M., Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES, IEEE Transactions on Software Engg., SE-5/3, May 1979.

TaGS85 Tay Y., Goodman N., Suri R., Locking Performance in Centralized Databases, ACM TODS 10/4, Dec 1985, pp 415-462.

Thom79 Thomas R. H., A Solution to the Concurrency Control Problem for Multiple Copy Databases, Proceedings of the 1978 COMPCON Conference (IEEE), 1979.

Thom93 Thomasian A., Two-Phase Locking Performance and its Thrashing Behavior, ACM TODS 18/4, Dec 1993, pp 579-625.

Thom98 Thomasian A., Concurrency Control: Methods, Performance, and Analysis, ACM Computing Surveys 30/1, Mar 1998, pp 70-119.

ThRy91 Thomasian A., Ryu I., Performance Analysis of Two-Phase Locking, IEEE TSE 17/5, May 1991, pp 386-401.

Will93 Williams C., Concurrency Control in Asynchronous Computations, PhD Dissertation, Dept. of Computer Science, University of Virginia, 1993.

YuDi92 Yu P., Dias D., Analysis of Hybrid Concurrency Control Schemes for a High Data Contention Environment, IEEE TSE 18/2, Feb 1992, pp 118-129.

YuDi93 Yu P., Dias D., Performance Analysis of Concurrency Control Using Locking with Deferred Blocking, IEEE TSE 19/10, Oct 1993.

YuDL93 Yu P., Dias D., Lavenberg S., On the Analytical Modelling of Database Concurrency Control, JACM 40/4, Sept 1993, pp 831-872.

YDRI85 Yu P., Dias D., Robinson J., Iyer B., Cornell D., Modelling of Centralized Concurrency Control in a Multi-System Environment, Perf Eval Rev 13/2 (Proc 1985 ACM SIGMETRICS), pp 183-191.