Network-Aided Concurrency Control in Distributed Databases

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Rashmi Srinivasa

January 2002

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy (Computer Science)

Rashmi Srinivasa

This dissertation has been read and approved by the Examining Committee:

Paul F. Reynolds, Jr. (Thesis Advisor)

Sang H. Son (Committee Chairman)

James C. French

Jack W. Davidson

Ronald D. Williams

Accepted for the School of Engineering and Applied Science:

Dean Richard W. Miksad
School of Engineering and Applied Science

January 2002

Abstract

Concurrency control is an integral part of a database system. Devising a concurrency control technique that has a low lost opportunity cost and a low restart cost is a hard problem. The interconnection network in a distributed database system can act as a powerful coordination mechanism by providing certain useful properties. We identify several such useful network properties, and present a new family of concurrency control techniques that are built on top of these properties. Network-aided concurrency control techniques use network properties to keep the lost opportunity cost and restart cost low. Our thesis is that network properties can be exploited to achieve efficient concurrency control of transactions.

We also contribute to the evaluation of concurrency control techniques, both analytically and through simulation. Traditional analytical models fail when trying to model certain kinds of workloads. We present a new analytical modelling technique that overcomes the limitations of traditional models. We also show that current perception of the relative merits of concurrency control techniques is flawed. Timestamp ordering concurrency control techniques have been perceived as poor performers until now. We show that two timestamp ordering techniques — a traditional technique and a network-aided technique — perform better than the most popular concurrency control technique for a wide range of conditions.

Acknowledgements

Several people are responsible for making the writing of this dissertation such an enjoyable experience. My husband, Anand Natrajan, looked on my lightning leaps from confidence to self-doubt to elation with equanimity and perhaps a little amusement. Not only did he encourage and cheer me on throughout the process, but he also gave me useful technical comments and criticism. He celebrated my little triumphs by buying me cappuccinos, and consoled me during setbacks by buying me cappuccinos. (I had a good thing going there until he wised up.)

My parents, K. Srinivasa and Parvathi Srinivasa, have always been an important influence in my life. They brought me up to be a confident and well-adjusted individual, and always believed in my ability to finish the things I set out to do. I appreciate their constant support and love. My parents-in-law, S. Natrajan and Shanta Natrajan, and brother-in-law, Chintu, have also wished me well throughout this endeavour, and have celebrated my successes heartily.

My committee is responsible for the thoroughness of this dissertation. I thank my advisors, Prof. Paul Reynolds and Dr. Craig Williams, for their comments and their attention to detail. I am also grateful to Prof. Jim French for his advice and his reassurance. It was a pleasure collaborating with him on various projects, and I learnt a lot in the process. Prof. Sang Son was also a pleasure to work with, and I appreciated his technical expertise as well as his infectious enthusiasm for research. My colleagues, especially Dave Coppit and Gabe Ferrer, were very helpful with their incisive questions and comments.

A number of friendships enriched my experience here. Aruna Viswadoss and Ravi Vancheeswaran helped in so many ways, and made my stay in Virginia very enjoyable. I have Prakash Vachaspati and Venkat Pallasana to thank for many hours of hiking, coffee, games and conversation. Karine Boulle, Anh Nguyen-Tuong, Glenn Wasson, Dave and Dorothy Coppit and Joy and George Matthews have been wonderful friends, and I have spent many agreeable days in their company. I am also appreciative of my many friends in various cities and countries who brightened my days with their letters, cards and phone calls, and wished me well from across the world. My friends from volleyball and dance provided the all-important diversions that allowed me to go back to my work refreshed and ready.

Finally, Virginia is the most beautiful place I've lived in, and I will never forget its mountains, lakes, forests and farms, that constantly renewed my spirits.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrency control in a database system is the activity of coordinating the actions of transactions that operate in parallel, access shared data, and potentially interfere with one another [BeHG87]. It is desirable that this coordination be efficient. There are two costs associated with concurrency control: 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. Restart 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. Devising a concurrency control technique that has both a low lost opportunity cost and a low restart cost is a hard problem.

We present a new family of concurrency control techniques that use the interconnection network in a distributed database system as an aid to concurrency control. The network can act as a powerful coordination mechanism by providing certain useful properties in the form of communication guarantees. Network-aided concurrency control techniques use such properties to keep the lost opportunity cost and restart cost low. Our thesis is that network properties can be exploited to achieve efficient concurrency control of transactions.

The evaluation of different concurrency control techniques is an important task, and traditional techniques have been evaluated both analytically and through simulation. Traditional analytical models fail when trying to model certain kinds of workloads. We present a new analytical modelling approach that solves this problem. Timestamp ordering concurrency control techniques have been perceived as poor performers until now. We show that two timestamp ordering techniques — a traditional technique and a network-aided technique — perform better than the most popular concurrency control technique for a wide range of conditions.

In section 1.1, we discuss the evaluation of concurrency control techniques, and describe our contributions in this area. In section 1.2, we describe our contributions in network-aided concurrency control. In section 1.3, we list the assumptions and limitations of this work. In section 1.4, we provide an outline of the thesis.

1.1 Evaluation of Concurrency Control Techniques

Traditionally, concurrency control techniques have been classified into four categories — locking, timestamp ordering, optimistic and hybrid. Current databases use dynamic two-phase locking (2PL) and its variants almost exclusively [Date00, BeHG87, GrRe92].

Concurrency control techniques have been evaluated both by analytical modelling and through simulations. We use both these methods in order to evaluate our network-aided concurrency control techniques against traditional techniques. We chose to simulate our techniques rather than implement them on a prototype system, because a simulation allows us to consider a wide variety of systems, and prevents our results from being confined to the characteristics of any single prototype. As is typical in simulations of concurrency control techniques, we used synthetic workloads rather than traces, because these synthetic workloads are easily available and fairly standardized, and are based on actual transaction workloads.

Careful modelling is important in order to be able to evaluate concurrency control techniques under different system parameters and workloads. A serious limitation of traditional analytical models for lock-based techniques is that they do not accurately model performance at high data contention levels. As the demand for high transaction throughput increases, the degree of transaction concurrency increases, and it becomes more and more important to study high data contention scenarios. Traditional analytical models assume that the queue length at any server never increases beyond two. We show that this assumption leads to an imprecise prediction of performance at high data contention levels, and causes traditional models to incorrectly predict good performance when the system has already become unstable. Moreover, traditional models consider only a restricted form of a distributed database. An important contribution of our work is a new analytical modelling technique for database concurrency control, that models a fully distributed database, and that allows arbitrary queue lengths. We use this technique to model a traditional concurrency control technique and a network-aided concurrency control technique. Our analytical models continue to predict performance accurately, even under high data contention.

Traditionally, timestamp ordering concurrency control techniques have been largely ignored by the database community because of their reportedly poor performance. We show the surprising result that timestamp ordering is a serious competitor to the most popular concurrency control method — dynamic 2PL. As hardware capabilities and workload specifications change, the effects of balancing different trade-offs change, and timestamp ordering becomes a viable alternative. We show that a traditional timestamp ordering technique — BTO — performs better than 2PL for a wide range of conditions. An important contribution of our work is the motivation of a re-evaluation of the merits of timestamp ordering concurrency control in distributed databases.

1.2 Network-Aided Concurrency Control

The network can act as a powerful coordination mechanism by providing certain properties that are useful to concurrency control of distributed transactions. Efficient and scalable concurrency control techniques can be built on top of a network that provides these useful properties at a low cost. The network or communication subsystem can help with concurrency control in several ways including ordering, propagating useful information that allows faster progress, taking over some of the tasks of the servers, and reducing wasted processing. We have identified several such network properties — total ordering, predictability, extended predictability, pruning and caching. We discuss the utility of these properties to concurrency control, the feasibility of their implementation and the kinds of systems where the properties would be useful. We describe network-aided concurrency control techniques based on these useful network properties. We examine two of the properties — total ordering and predictability — in detail, and discuss their utility to concurrency control.

We define a network-aided concurrency control technique — ORDER — that is based on the network property of total ordering. We detail the algorithms that must be executed by the various modules of a distributed database system in order to implement ORDER. We study ORDER both analytically and through simulation, and demonstrate that it performs better than traditional concurrency control techniques for a wide range of conditions. We study the performance effects of various parameters including ordering cost, and demonstrate that ORDER's advantage disappears only when network latency is high and ordering is implemented inefficiently.

We define another network-aided concurrency control technique — PREDICT — that is based on the network property of predictability. We describe the algorithms required to implement PREDICT. We study the behaviour of PREDICT, and demonstrate that it performs better than traditional concurrency control techniques under a variety of workloads and system parameters. PREDICT falls under the category of timestamp ordering concurrency control, and the study of PREDICT bolsters our observation that timestamp ordering concurrency control deserves a new look. We present a set of variants of the PREDICT technique based on reasonable assumptions about network characteristics in a distributed database system. We demonstrate that PREDICT achieves a good balance between lost opportunity cost and restart cost.

Our work provides valuable insight into the choice of concurrency control technique for a given system. It is clear from our studies that dynamic 2PL is a poor choice because its performance degrades in the presence of high data contention, at lighter loads than other techniques. ORDER and PREDICT are both very good choices when message latencies are low, because they keep both queue lengths and restart behaviour down. When message latencies are high, PREDICT and BTO are good choices, because they keep queue lengths lower than 2PL, and do not suffer from the latency penalty of ORDER. Both ORDER and PREDICT require predeclaration of accesses. If predeclaration is not possible or too expensive, BTO is a good choice, because it performs better than or as well as 2PL in all cases except when data contention and message latencies are both low.

1.3 Assumptions and Limitations

We are considering networks that provide certain communication guarantees. The effect of system failures on the performance of such networks is not the focus of this work. We also do not investigate how recovery techniques can benefit from these communication guarantees. We show that our network-aided concurrency control techniques interface with the standard recovery protocol in a straightforward manner. We also show that the impact of recovery overhead on systems that use network-aided techniques and on systems that use traditional concurrency control techniques is similar. However, an investigation of the precise quantitative effect of different modes of failure on performance is beyond the scope of this dissertation.

There are networks that are highly available due to redundant hardware like duplicate routers, dual power supplies, dual CPUs, disk mirroring and multiple network interface cards. Examples of such network systems or components include CytaNet, Bay Networks' BLN and BCN, and Cisco and HP's HA Server-to-Switch Foundation Configuration. It is reasonable to assume the feasibility of databases that operate on such a highly-available network, or alternatively, on a network contained in a small area like a building. Such databases would benefit from network-based concurrency control techniques as they stand, without the need for significant exploration of recovery algorithms.

In our performance studies, we assume that there is no replication of data on multiple servers. Our network-aided concurrency control algorithms work correctly even with replication. However, an analysis of the performance effects of replication is outside the scope of my investigation.

We have assumed serializability to be the correctness criterion for this work. Serializability is the most-widely used correctness criterion in databases. In future work, it would be interesting to study network-aided techniques that guarantee weak consistency semantics instead of serializability, in an attempt to improve performance.

1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we provide background and terminology, and survey related work in the fields of database concurrency control and network properties. In Chapter 3, we discuss limitations of traditional analytical models, and present our new analytical modelling technique. We apply our technique to the traditional dynamic 2PL concurrency control technique, and show that our model is superior to traditional models. In Chapter 4, we show that it is time to re-evaluate the merits of timestamp ordering concurrency control. We demonstrate conditions under which a traditional timestamp ordering technique outperforms the popular dynamic 2PL technique. In Chapter 5, we present network-aided concurrency control. We identify several network properties that are useful to concurrency control, and describe concurrency control techniques based on these properties. We also discuss the implementation of these properties, and discuss the kinds of systems where the properties would be useful. In Chapter 6, we define and study a network-aided concurrency control technique based on the network property of total ordering, and show that it performs better than traditional concurrency control techniques for a wide range of conditions. In Chapter 7, we define and study another network-aided concurrency control technique based on the network property of predictability, and show the conditions under which it is superior to traditional techniques. In Chapter 8, we address the issue of recovery. We show that network-aided techniques are compatible with a well-known recovery protocol, and discuss the effects of failures on different techniques. In Chapter 9, we summarize the results, contributions and limitations of our work, and suggest avenues for future research.

Chapter 2

Background and Related Work

In this chapter, we describe background, terminology and related work in the fields of database concurrency control and network properties. In section 2.1, we describe our system model, including transactions, correctness of execution and distributed database architecture. In section 2.2, we discuss concurrency control, the classification of concurrency control techniques and existing performance studies of these techniques. In section 2.3, we discuss network properties, covering research in the areas of ordering properties and active networks. We summarize in section 2.4.

2.1 System Model

A database consists of a set of *data items*, each of which has a value. A *transaction* is a logical unit of work, typically involving several database operations [Date00]. Each transaction, if executed alone on an initially consistent database, will terminate, produce correct results, and leave the database consistent. For correctness, transactions must execute atomically, meaning that each transaction accesses shared data without interfering with other transactions, and a transaction either has no effect at all, or all of its effects are permanent. The above properties of a transaction constitute the ACID properties [Date00]. *Operations* are commands used by a transaction in order to execute. Operation types are `read`, `write`, `predeclare`, `commit` and `abort`. A `read` on data item `x` causes the database to return the value stored in `x`. A `write` of value `val` on data item `x` causes the value of `x` to be replaced with `val`. A `predeclare` on data item `x` warns the database that the transaction intends to write to `x` in the future. A `commit` tells the database that the transaction has terminated normally, and all of its effects should be made permanent. An `abort` tells the database that the transaction has terminated abnormally, and all of its effects should be obliterated. An example of a transaction is as follows:

```
transaction T1 {
  v1 = read (x);
  v2 = read (y);
  if (v1 < 100) {
    abort;
  }
  else {
    write (x, v1-100);
    write (y, v2+100);
    commit;
  }
}
```

As seen in the above example, a transaction may abort if it decides not to proceed. This type of abort is called a *unilateral abort*. In contrast to a unilateral abort, a transaction may be aborted by the database management system (DBMS) if committing the transaction will lead to incorrect execution. We assume that reading and writing data items in the database is the only way in which two transactions can communicate with each other. The set of data items that are read by a transaction is called the *readset* of the transaction, and the set of data items that are written to by a transaction is called the *writeset* of the transaction.

2.1.1 Correctness

When transactions execute concurrently, their operations may be interleaved. One way to avoid interference problems is to disallow transaction operations from being interleaved. An execution is *serial* if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. However, a serial execution allows no concurrency, making poor use of system resources. A more efficient way of avoiding interference problems is to control the ordering of conflicting operations only. Two operations are said to conflict if they both operate on the same data item, and at least one of them is a `write`. Controlling the ordering of conflicting operations yields executions that have the same effect as serial executions. Such executions are called *serializable* executions. An execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions.

In order to ensure correctness in the presence of failures, an execution must also be *recoverable*. An execution is recoverable if, for every transaction T that commits, T's commit follows the commit of every transaction from which T read. Aborting transactions may trigger further abortions, a phenomenon called *cascading abort*. Enforcing recoverability does not remove the possibility of cascading aborts. A DBMS avoids cascading aborts if it ensures that every transaction reads only those values that were written by committed transactions. *Strict* executions avoid cascading aborts and are recoverable. A DBMS that ensures strict execution delays both `read`s and `write`s for `x` until all transactions that have previously written `x` are committed or aborted.

Serializability is the most popular correctness criterion, but there are other weaker forms of correctness or *isolation levels*. The ANSI-SQL92 standard defines four isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE [ANSI92]. The levels differ according to the kinds of inconsistencies they allow.

2.1.2 Distributed Database Architecture

A distributed database management system (DDBMS) involves a collection of sites interconnected by a network. Each site runs one or more of the following software modules: a transaction manager (TM), a data manager (DM) and a concurrency control scheduler (or simply *scheduler*). In the client-server model, a site can function as a client, a server or both. A *client* runs only the TM module, and a *server* runs only the DM and scheduler modules. Each server stores a portion of the database. Each data item may be stored at any server or redundantly at several servers. A centralized database management system, in contrast, stores all its data items on a single server, and has a single DM managing all the data. Figure 1 shows the system architecture for the client-server model. Users interact with the DDBMS by executing transactions, which are on-line queries or application programs. TMs supervise interactions between transactions and the database. The TM at the site where the transaction originates is called the *initiating TM*. The initiating TM receives operations issued by a transaction, and forwards them to the appropriate schedulers. The goal of a scheduler is to order operations so that the resulting

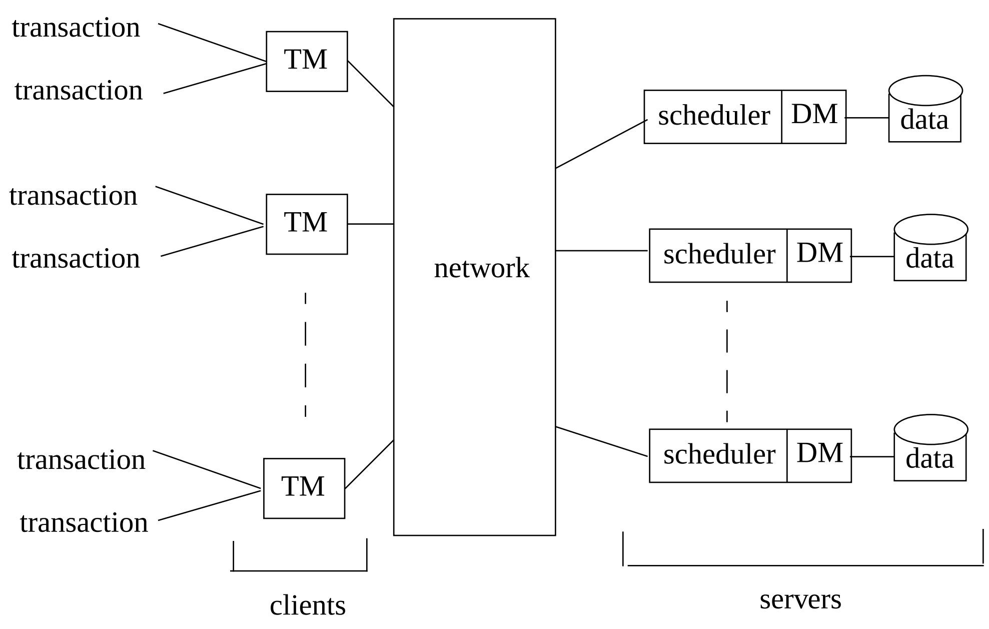


FIGURE 1: System architecture

execution is correct. DMs manage the actual database by executing operations, and are responsible for recovery from failures. Transactions communicate with TMs, TMs communicate with schedulers and DMs, and DMs manage data.

2.2 Concurrency Control

Concurrency control in a database is the activity of coordinating the actions of transactions that operate in parallel, access shared data, and therefore potentially interfere with one another [BeHG87]. Serializability is the most widely used criterion for correctness of a DDBMS. A transaction is an *atomic action*. An atomic action is a group of operations that must be executed as a whole, without interference from other operations [OwGr76]. Atomic actions can be *structured* or *flat*. A *structured* atomic action has internal dependences among the variables accessed by the operations in the atomic action. A *flat* atomic action has no internal dependences. Concurrency control combined with a recovery protocol guarantees the ACID properties even in the presence of failures.

Several concurrency control techniques have been proposed, based on different types of schedulers. A scheduler can be *conservative* or *aggressive* or a combination of the two. A *conservative* scheduler delays operations until it is certain that there will be no conflicting operation. Some of this delay is unnecessary because there may be no conflicting operation that is due to arrive. Therefore, conservative techniques suffer from a lost opportunity cost. Examples of conservative techniques are conservative timestamp ordering [BeGo81] and two-phase locking (2PL) [EGLT76].

An *aggressive* scheduler reduces unnecessary delay by scheduling operations immediately. However, it runs the risk of having to reject operations later, thereby causing the issuing transaction to abort and restart. Examples of aggressive techniques are basic timestamp ordering (BTO) [ShMi77a, ShMi77b] and optimistic concurrency control (OPT or OCC) [Bada79, Casa79, BaHR80].

Designing a concurrency control technique that keeps the lost opportunity cost as well as the restart cost low is a hard problem. Traditionally, concurrency control techniques have been classified into four categories — locking, timestamp ordering, optimistic and hybrid. Surveys of the different techniques are available [Thom98a, YWLS94].

2.2.1 Locking

In basic two-phase locking (2PL) [EGLT76], a transaction must own a *read lock* on data item `x` before reading `x`, and must own a *write lock* on `x` before writing `x`. Read locks conflict with write locks on the same data item, and write locks conflict with other write locks on the same data item. Read locks are implicitly requested by the TM by sending `read`s, and write locks are implicitly requested by the TM by sending `write`s. Write locks are implicitly released by `commit`s, but in order to release read locks, special lock release operations are required. Every transaction obtains locks in a two-phase manner. During the *growing* phase, the transaction obtains locks without releasing any locks. During the *shrinking* phase, the transaction releases locks without obtaining any locks. A basic 2PL scheduler follows the following three rules.

1. When the 2PL scheduler receives a lock request, it tests whether the requested lock conflicts with another lock that is already set. If so, it queues the lock request. If not, it responds to the lock request by setting the lock.
2. Once the 2PL scheduler has set a lock on a data item, it cannot release the lock until the DM has completed processing of the lock's corresponding operation.
3. Once the 2PL scheduler has released a lock for a transaction, it may not subsequently obtain any locks for the same transaction.

A basic 2PL scheduler requires a strategy to prevent, avoid or detect-and-break deadlocks. Various strategies are waits-for-graphs [Holt72, KiCo74], preordering and predeclaration of locks [BeGo81], timestamp-priority-based restarts [RoSL78] and many others. Variations on the basic 2PL method include primary copy 2PL [Ston79], voting 2PL [Thom79], multiversion 2PL [BeHG87, StRo81], centralized 2PL [AlDa76, Garc79], asymmetric running priority [FrRo85], symmetric running priority [FrRT92], wait-depth-limited locking (WDL) [FrRT92], dynamic locking with no waiting [RyTh90b], asymmetric cautious waiting [HsZh92], Wound-Wait [RoSL78], Wait-Die [RoSL78], local wait-depth control (LWDC) [WaLK98] and adaptive callback locking [CaFZ94]. Other variations that make restrictive assumptions about transaction-specification and correctness are weaker consistency semantics [Date95, GrRe92], decomposition into subtasks [HaRo93], ordered sharing [AgEL94], altruistic locking [SaGS94], proclamations [JaSh92], increment/decrement locks [GrRe92], sagas and compensations [GaSa87, GrRe92], commutative operations [Weih88], and other semantic methods [SkZd89, Ozsu94, RaCh96].

Dynamic 2PL and *static 2PL* are two variants of basic 2PL. In dynamic 2PL, a transaction obtains a lock only when it needs to access the corresponding data item. In static 2PL, a transaction predeclares and obtains all the locks it may need before it begins any computation. Current databases use dynamic 2PL and its variants almost exclusively [Date95, BeHG87, GrRe92]. Almost all implementations of 2PL enforce strict execution, which requires the scheduler to release all of a transaction's read locks after the transaction terminates (when the scheduler receives the transaction's `commit` or `abort`), and all of the transaction's write locks after the DM has processed the transaction's `commit` or `abort`. 2PL involves the overhead of extra messages needed to acknowledge lock sets and to release locks, and a mechanism to prevent or detect-and-break deadlocks.

2.2.2 Timestamp Ordering

In timestamp ordering methods, the TM assigns a unique timestamp to each transaction it executes, and attaches the transaction's timestamp to every operation issued by the transaction. A timestamp ordering (TO) scheduler orders conflicting operations according to their timestamps. The aggressive form of TO-based concurrency control is basic timestamp ordering (BTO) [ShMi77a, ShMi77b]. A BTO scheduler executes an operation immediately if it is possible to do so. The scheduler rejects an operation if it has already executed a conflicting operation with a later timestamp. When an operation is rejected, the transaction that issued it must abort and restart with a later timestamp. If a BTO scheduler receives operations in an order widely different from their timestamp order, it may reject too many operations, thereby causing too many transactions to abort. Starvation of transactions can occur in a BTO system because of continuous conflicts and aborts. Variations on the basic algorithm include Thomas Write Rule [Thom79] and multiversion TO [Reed78].

A TO technique that avoids restarts is conservative timestamp ordering (CTO) [BeGo80]. A CTO scheduler delays an operation until it is certain that no conflicting operations with a lower timestamp will arrive. For every TM, the scheduler keeps track of the minimum timestamp of a `read` that it has received from that TM but not yet executed, and also the minimum timestamp of a `write` that it has received from that TM but not yet executed. The scheduler executes a `read` if it has received a `write` with a greater timestamp from every other TM, and buffers the `read` otherwise. The CTO scheduler executes a `write` if it has received a `read` or a `write` with a greater timestamp from every other TM, and buffers the `write` otherwise. To keep computation progressing, TMs periodically send timestamped `null` operations to every scheduler. This all-to-all message sending is slow and unscalable. The scheduler processes *all* operations in timestamp order, not just *conflicting* ones, thus enforcing an overly-restrictive ordering. A variant on the basic algorithm uses transaction classes [BGRP78, BeSR80].

In a multiversion timestamp-ordering (MVTO) system [Reed78], DMs maintain multiple versions of data items. Maintaining multiple versions may not add much to the cost of concurrency control, because the versions may be needed by the recovery algorithm [BeHG87]. An MVTO scheduler keeps track of the timestamps of various versions of data items, and executes a `read(x)` by telling the DM which version of `x` to use. The scheduler rejects a `write` if a `read` with a later timestamp has already read an earlier version of the data item. On the other hand, if the `write` can be executed without making a previous `read` execution invalid, the scheduler and the DM execute the `write` and create a new version.

2.2.3 Optimistic

Optimistic (OPT) schedulers schedule each operation as soon as it is received. When a transaction is about to commit, all involved OPT schedulers check whether committing the transaction will violate serializability. If serializability will be violated, the transaction is aborted. Such schedulers are also called certifiers [Bada79, Casa79, BaHR80]. A variation on OPT is adaptive optimistic concurrency control [AGLM95].

The aggressive behaviour of OPT schedulers may lead to a high number of aborts and restarts. Moreover, a transaction is aborted at a very late stage, when it has completed all its computation, thus resulting in a large amount of wasted processing. On the other hand, the amount of unnecessary waiting involved in the OPT technique is lower than that in conservative methods like 2PL or CTO.

2.2.4 Hybrid

Bernstein and Goodman [BeGo81] enumerate several concurrency control methods that combine 2PL and TO. A distributed optimistic 2PL scheduler [CaLi91] executes transactions optimistically, but if a transaction is aborted, the scheduler uses 2PL to execute the transaction a second time. Other hybrid approaches are optimistic with dummy locks [HaDo91], hybrid optimistic concurrency control [YuDi92] and broadcast optimistic concurrency control [YuDi92].

2.2.5 Performance of Concurrency Control Techniques

The performance of concurrency control techniques has been studied analytically as well as experimentally. Previous performance studies have concentrated on centralized servers and low data contention scenarios. We first discuss concurrency control performance in centralized databases, and then discuss performance studies of concurrency control techniques in distributed databases.

2.2.5.1 Centralized Database Concurrency Control

The *thrashing* behaviour of dynamic 2PL is well known, and it causes the throughput to start dropping after a certain multiprogramming level [Thom93]. Thrashing occurs because the addition of one transaction to the system causes more than one transaction to become blocked. Approximate mean value analysis has been used to analyze the performance of 2PL, OPT and several hybrid concurrency control schemes in centralized databases [TaGS85, YuDi92, YuDi93, YuDL93, RyTh90a, RyTh90b, Thom93, Thom98a, ThRy91]. These analyses cannot predict peak throughput in systems where the level of transaction concurrency is high [Thom98a]. Dynamic 2PL performs better than its variants when processing capacity is low. Variants like WDL perform better than dynamic 2PL when processing capacity is high [Thom98a], but at the cost of additional processing.

Timestamp ordering methods are more appropriate for distributed databases, and are known to have a poor performance otherwise [Thom98a]. 2PL has been shown to outperform BTO in a centralized database system [RyTh90b]. Another study has compared 2PL to BTO and MVTO in a system with a single server connected to one or more clients over a network [LiNo83], and reports that 2PL is superior to the two other techniques. Optimistic concurrency control methods are susceptible to abort according to a quadratic effect, that is, the probability of abort increases as the square of transaction size [FrRT92].

2.2.5.2 Distributed Database Concurrency Control

Concurrency control techniques in distributed databases have been analytically studied, but these analytical studies make restrictive assumptions [Gray96, CDIY90, CiDY92]. Existing analytical models for 2PL in a distributed database [JeKT88, ShWo97] do not permit simultaneous processing of a transaction at multiple sites. Basic timestamp ordering and multiversion timestamp ordering techniques have also been analyzed [Li87, ReTH96, Sing91a, Sing91b].

BTO has been reported to perform worse than 2PL in a distributed database, in a low data contention scenario [CaLi91]. The non-replicated case in the study assumes that all of a transaction’s accesses are local. The replicated cases assume that the primary copies of all data items accessed by a transaction are located at a single site. There have been no prior studies of the performance of BTO in a truly distributed database environment where a transaction can access data items at multiple arbitrary sites. Performance studies of BTO and 2PL under high levels of data contention are also unavailable.

OPT performs better than BTO except when the cost of message-sending is very low [CaLi91]. The hybrid method ODL outperforms 2PL in high-conflict situations, but the two methods perform similarly in low-conflict situations [HaDo91]. LWDC has been shown to outperform WDL and 2PL, and the percentage improvement in peak throughput of LWDC over 2PL is approximately 30% [WaLK98]. There is no consensus on the best concurrency control technique for distributed databases, and dynamic 2PL is the technique that is the most widely-used.

2.3 Network Properties

Ordering properties like total ordering and causal ordering have been explored in the distributed computing literature. In addition, active networks research has explored the possibility of performing various tasks in network routers in order to help distributed applications.

2.3.1 Ordering Properties

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]. An isotach network has been simulated in software [Rege97] and implemented as a small hardware prototype [LaMy00]. Isis allows multicasting to process groups with various ordering guarantees [BiJo86]. The Totem system does total ordering between clusters of workstations [MMAB96]. The Transis system also provides ordering protocols for process group systems [DoMa96]. Psync is a distributed computing system that provides ordered and atomic multicast protocols [MiPS91]. The Amoeba operating system supports a subsystem in which message delivery is atomic and totally ordered [Mull90, ReST89, KaTa91]. Vector clocks were used for ordering in the Harp replicated file system [LLSG92, Lisk91]. In the Highly Available System (HAS), protocols were proposed for achieving totally ordered delivery guarantees to groups [CASD85].

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. Neiger and Toueg [NeTo93] propose a communication primitive called a *publication* that makes all recipients deliver a message at the same logical time called the *publication time*. Database concurrency control is proposed as a possible application, where processors *publish* their lock-requests and lock-releases. Schiper and Raynal [ScRa96] propose a concurrency control technique based on totally-ordered-multicast and group communication [BiJo86]. They refer to Totem [MMAB96] as a system that provides an implementation for the property of total ordering of messages across groups. They do not address the issue of transactions that are structured atomic actions, or transactions that issue unilateral aborts. Neither Neiger and Toueg nor Schiper and Raynal address performance.

2.3.2 Active Networks

Active networks are an approach to network architecture in which the switches of the network perform customized computations on the messages flowing through them [TSSW97]. This processing can be customized on a per-user or per-application basis. Caching, fusion, fission, delegation and filtering have been proposed as suitable tasks for active networks [LeWG98]. A reliable multicasting protocol called Active Reliable Multicast (ARM), has used the idea of active networks [LeGT98]. ARM is a loss-recovery scheme for reliable multicast over the Internet. ARM routers suppress duplicate negative acknowledgements, share the load of retransmission and use partial multicasting to limit the scope of retransmitted data. To our knowledge, active networks have not been used in a DBMS setting.

2.4 Summary

Concurrency control is an integral part of a database management system. Several concurrency control techniques exist in the literature, and they are traditionally classified into lock-based, timestamp ordering, optimistic and hybrid. Dynamic 2PL and its variants are used almost exclusively by current databases.

Ordering properties like total ordering and causal ordering have been explored in the distributed computing literature. In addition, active networks research has explored the possibility of performing various tasks in network routers in order to help distributed applications. To our knowledge, our work is the first detailed exploration of the utility of network properties in database concurrency control.

Chapter 3

Analytical Modelling

Careful modelling is important in order to be able to evaluate concurrency control techniques under different system parameters and workloads. Several analytical models have been proposed for various concurrency control techniques. A serious limitation of traditional analytical models for lock-based techniques is that they do not accurately model performance at high data contention levels. As the demand for high transaction throughput increases, the degree of transaction concurrency increases, and it becomes more and more important to study high data contention scenarios. We present a new analytical modelling technique that continues to predict performance accurately even at high data contention levels [SrWR01a]. In addition, unlike previous work, we model a fully distributed database, where a transaction can be executed at multiple sites, and transaction actions can be processed concurrently at the different sites. We apply our modelling technique to the 2PL concurrency control technique, and demonstrate through validation against a simulation, that our model yields more accurate predictions than traditional analytical models for 2PL. In a later chapter (Chapter 6), we apply our analytical modelling technique to another concurrency control technique as well.

In section 3.1, we describe traditional analytical models and discuss their limitations. In section 3.2, we present a new analytical modelling technique for concurrency control, that addresses these limitations. In section 3.3, we use our modelling technique to model the 2PL concurrency control technique. In section 3.4, we validate our analytical model of 2PL against a simulation, and demonstrate its superiority over traditional 2PL models. In section 3.5, we discuss the effects of the simplifying assumptions we have made in our analytical model of 2PL.

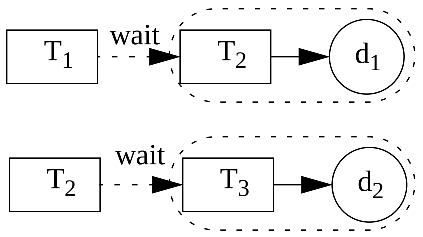
3.1 Traditional Analytical Models

Most analytical studies of concurrency control techniques have focussed on performance in centralized databases. An approximate mean value analysis method is used by Tay et al. to analyze the performance of 2PL in a centralized database system [TaGS85]. A similar approximate mean value analysis method is used by Yu et al. to analyze 2PL, OPT and several hybrid concurrency control techniques in centralized databases [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 by Thomasian and Ryu in order to analyze 2PL and OPT in a centralized database system and estimate the lock contention level at which dynamic 2PL starts thrashing [RyTh90a, RyTh90b, Thom93, Thom98a, ThRy91].

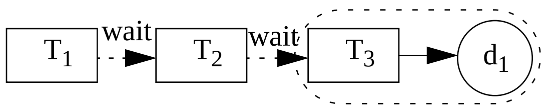
Concurrency control in a distributed database has been modelled analytically, but with several restrictions. Gray uses approximate mean value analysis to estimate wait probability and deadlock rate in a fully-replicated distributed database, but assumes that replicate updates are performed sequentially [Gray96]. Approximate mean value analysis is used by Ciciani et al. to estimate the probability of conflict and response time of transactions using the OPT concurrency control technique in a restricted form of distributed database [CDIY90, CiDY92]. The restriction is that remote transactions execute entirely at a central site that replicates all data. In another restricted form of distributed database, transactions of one class use 2PL and execute entirely at the local primary-copy site; transactions of the other class use OPT and access data that is at a single known remote site [CiDY90]. Analytical models for 2PL in a distributed database are presented by Jenq et al. and Sheikh et al., but the models do not permit concurrent processing of a transaction at multiple sites [JeKT88, ShWo97].

3.1.1 Limitations of Traditional Models

A major limitation of traditional analyses of lock-based techniques — both in centralized and distributed databases — is that they cannot predict peak throughput in systems where the level of transaction concurrency is high [Thom98a]. Transaction operations queue up in order to access a data item. Traditional models assume that the queue length at any data item is never more than two (one operation holding the data item and another operation waiting to access the data item) [TaGS85, CiDY90, CiDY92, YuDi93, YuDL93, Thom93]. With this assumption, certain data conflict scenarios are ignored. More specifically, the effect of transaction T_1 waiting at data item d_1 for transaction T_2 while transaction T_2 is waiting at a different data item d_2 for transaction T_3, *is* captured; but the effect of T_1 waiting at d_1 for T_2, while T_2 is waiting at the same data item d_1 for T_3, *is not* captured (Figure 2). By modelling queue lengths of only 1 or 2, traditional models consider only two situations — a transaction encounters no conflicts; or the transaction encounters exactly one conflict at one or more data items. This assumption makes traditional models simple to analyze, and is adequate to predict performance well for the low data contention scenarios that traditional studies have considered. However, since queue lengths increase as data contention increases, traditional analytical models are inadequate to model performance at high data contention levels.



(a) Queue length ≤ 2
Captured by traditional models



(b) Queue length > 2
Not captured by traditional models

FIGURE 2: Limitation of traditional analytical models

Another limitation of the analyses for distributed databases is that they do not model a fully distributed database. Some models assume that any transaction is executed in its entirety at a single site, which can be local or remote [CDIY90, CiDY90, CiDY92]. Other models allow a transaction to be executed at multiple sites, but assume that transaction actions are executed *sequentially* at the different sites [JeKT88, Gray96, ShWo97]. Both of these assumptions about distributed databases allow traditional models to analyze response time in a simple manner by disregarding the overlapping of transaction tasks at different sites. However, these assumptions lead to overestimation of transaction response time by disallowing the parallel processing of a transaction at multiple sites. In the following section, we present our analytical modelling technique, and describe how it avoids the above limitations.

3.2 A New Analytical Modelling Technique

Our modelling technique is similar to previous ones [YuDi92, CDIY90, TaGS85], in that we combine a data contention model and a hardware resource contention model, and use an iterative approach to calculate mean transaction response time. The iterative approach captures the following dependency. The probability of data conflict for any transaction depends on the concurrency control method and the transaction response time, which in turn depends on the data conflict probability. For example, if the data conflict probability increases, transaction response time increases due to additional waits to access a data item. A longer transaction response time leads to a longer data-holding time, and hence to a higher data conflict probability.

A major improvement made by our technique over previous ones is that our technique captures the high data contention scenario accurately. Previous models have assumed that the queue length at any data item is never greater than two. Our technique, on the other hand, allows arbitrary queue lengths at data items. As data contention increases, queue lengths at data items increase, and performance degrades rapidly due to long waits to access data. Traditional models are unable to predict this degradation in performance caused by long queues. By allowing arbitrary queue lengths, our modelling technique is able to predict the load at which this performance degradation occurs. By assuming that the queue length encountered by a transaction is less than or equal to two, traditional models consider only two possibilities — a transaction encounters no conflicts, and its data contention wait time is zero; or the transaction encounters exactly one conflict at one or more data items, and its data contention wait time is the mean remaining data-holding time. However, in a real system, a transaction can encounter different data contention wait times at different data items, some of these wait times possibly overlapping. First, we make the simplifying assumption that response time is dominated by the wait time of the operation that encounters the maximum queue length. Then, we do a probabilistic analysis of queue lengths, considering situations where this maximum queue length is 1, 2, 3 and so on. We combine the data contention wait time implied by each of these situations, in order to achieve a realistic estimate of transaction response time. The capability of handling arbitrary queue lengths makes our model applicable to a wider range of systems, rather than being restricted to systems that exhibit low data contention.

For the fully distributed case, our technique also improves upon previous ones in modelling the hardware resource contention component. We model a fully distributed database, where a transaction can be executed at multiple sites, and transaction tasks can be processed concurrently at different sites. Traditional models assume that different tasks are processed sequentially at the servers. We model a more realistic scenario where the tasks of a transaction overlap at multiple servers. This improvement prevents our model from underestimating the hardware resource component of response time.

3.2.1 Assumptions

We assume an open system model with Poisson transaction arrivals. 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.

Accesses to data items are assumed to be distributed uniformly over the entire database. With a non-uniform distribution, we would get higher data contention at lower loads. A non-uniform distribution is explored in later chapters. We assume that access contention events for a transaction are independent. We assume strict and serializable executions, and that there are no unilateral aborts. We assume that all transactions are of the same size, and that transaction operations arrive at each data item as a Poisson process. We explore the effect of variable-size transactions in section 3.5. We assume that the operation that spends the maximum amount of time waiting in a queue at the scheduler is the one whose response arrives the latest at the initiating TM.

Number of clients	C	Number of servers	S
Number of data items in database	D	Number of accesses by a 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}
Scheduler 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 modelling parameters

Table 1 shows the parameters that we use in our analysis. Transactions arrive at each of the C clients at rate λ. Each transaction makes K accesses, and the accesses are uniformly distributed over the entire database of D data items. The database is partitioned among S servers, with no replication, so that each data item is managed by only one server. The MIPS ratings of a client and a server are M_C and M_S respectively. The network latency is distributed exponentially with mean D_{NW}, and there is an overhead of I_{NW} instructions at a site for every message sent or received. A transaction takes I_{INPL} instructions for initial processing, and I_{COMP} instructions for computation after every `read`. A database access from stable storage takes D_{IO} seconds, and there is an I/O overhead of I_{IO} instructions at the server per access. The TM and scheduler impose overheads of I_{TM} and I_{CC} respectively on each operation or access.

In the following section, we apply our analytical modelling technique to analyze the 2PL concurrency control technique.

3.3 Analytical Model of 2PL

We present an analytical model of 2PL in a fully distributed database. Since executions are strict and serializable, a transaction holds all of its locks until the end of the transaction. We make the following assumptions in order to simplify the analysis. We assume that all lock requests are made at the start of the transaction, in effect requiring predeclaration of all data item accesses. In section 3.5, we show that this assumption does not significantly affect our model's ability to predict performance accurately. We assume that the number of deadlocks and restarts is zero. In section 3.5, we show that this assumption does not significantly affect the accuracy of our model. We also assume that all locks are exclusive. This assumption is relaxed in section 3.3.4.

We model data contention and hardware resource contention separately, and then iterate over the two components in order to obtain total response time. 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 operation of the transaction waiting to access a data item, and R_{EXEC} is the execution time of the transaction excluding this longest wait time. R_{EXEC} models hardware resource contention, and is estimated using queueing models, while R_{CONT} models data contention.

3.3.1 Data Contention

Let the mean data-holding time of a data item be T_H. Since we assume that all accesses 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 data item. We will compute R' in the next section. Let T_R be the mean remaining time that an operation will have to wait for the current data item holder to release the data item. With the assumption that all lock requests are made at the start of the transaction, T_R is equal to $R' / 2$ [YDRI85].

We assume that the K access contention events for a single transaction are independent of one another. The probability of contention for an access request is the data item utilization ρ.

$$\begin{aligned}\rho &= \text{(arrival rate of operations at a data item) / (service rate of operations at the data item)}\\ &= \text{(arrival rate of operations at a data item)} / (1 / T_H)\end{aligned}$$

Since there are C clients, each generating transactions of size K at rate λ, and accesses are uniformly distributed among the D data items, the arrival rate of lock requests at a data item is $\lambda * C * K / D$. We assume that lock requests arriving at a data item form a Poisson process with mean $\lambda * C * K / D$. Therefore,

$$\begin{aligned}\rho &= (\lambda * C * K / D) / (1 / T_H)\\ &= \lambda * K * T_H * C / D \quad \text{.........} \quad (1)\end{aligned}$$

In the following discussion, we use the term *queue length* at a data item to mean the number of operations waiting for that data item, plus the number of operations holding the data item. At any data item, the probability that the queue length < x is equal to $(1 - \rho^x)$.

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

In general,

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

Each of the K operations of a transaction has to wait before the data item is free and can be accessed by the transaction. Our aim is to find the longest amount of time that one of these operations will have to wait. This longest wait time corresponds to the longest queue length among the queues at the K data items. Let the probability that a transaction will not have to wait for any of its K accesses be P_0. Let the probability that the longest queue length = 1 be P_1, the probability that the longest queue length = 2 be P_2, and so on.

P_0 = Prob (queue length < 1 at all K data items)

= $(1 - \rho^1)^K$

P_1 = Prob (queue length < 2 at all K data items) - Prob (queue length < 1 at all K data items)

= $(1 - \rho^2)^K - (1 - \rho)^K$

In general,

P_x = Prob (queue length < x+1 at all K data items) - Prob (queue length < x at all K data items)

= $(1 - \rho^{x+1})^K - (1 - \rho^x)^K$

When $(P_0 + P_1 + P_2 + ... + P_n)$ is close enough to 1, then the data contention wait for a transaction is approximately

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

3.3.2 Hardware Resource Contention

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

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

where I_{INPL} accounts for the initial processing; the three $K*(I_{TM}+I_{NW})$ terms account for the TM and network overhead on lock requests, lock responses and `commit`s or `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, let the total number of instructions executed at the servers be I_S.

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

where $3*K*(I_{NW}+I_{CC})$ accounts for the network and scheduler overhead of processing lock requests, responses and `commit`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 access requests to the servers, the servers process the access requests and send read responses, and the client performs transaction computation. A `read` request results in a database access at the server, while a `write` request does not, because a transaction must send a `commit` before its `write`s will be written to the database. Therefore, we assume that the response time in the first phase is dominated by the response time of the `read` requests. We also assume that the first phase response time is dominated by the access request *l* that has to wait for the maximum amount of time on a scheduler 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 access 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} can be divided into a data contention component (R_{CONT}) and a hardware contention component R'_{PHASE1}.

$$R'_{PHASE1} = R_{PHASE1} - R_{CONT}$$

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

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. 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

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

$R_{CL2} = R_{CL} * K*(I_{TM}+I_{NW})/I_c$ (sending of `commit`s and lock-releases); and

$D_{NW}*(1/1 + 1/2 + ... + 1/W)$ = network traversal time of the slowest of the W `commit`s, where $W = (1-P_R)K$.

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.

3.3.3 Total Response Time

The hardware contention component R_{EXEC} is computed as described in the previous section. The data contention component R_{CONT} depends on the mean data-holding time T_H and the mean remaining data-holding time T_R. Recall that T_H is approximately equal to R', and $T_R = R' / 2$, where R' is the portion of the response time of a transaction when the transaction is holding at least one data item.

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

$= R - [R_{CL1} + D_{NW} + R_{SV}*(I_{NW}+I_{CC}+I_{IO})/I_S + D_{IO}]$ ....(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}, and execute the following steps.

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 $\rho \geq 1.0$, 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 $R = R_{EXEC} + R_{CONT}$.

3.3.4 Modelling Shared Locks

Allowing shared locks in addition to exclusive locks changes the behaviour of the lock-request queues at the schedulers. 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 $P_R m$, 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. If $P_R \leq 0.5$, 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 $P_R \leq 0.5$.

Therefore, equation 2 is replaced with

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

3.4 Validation

We designed a simulation in order to do detailed performance studies of dynamic 2PL and other concurrency control techniques. In this section, we describe our simulation model in terms of the system, workloads and algorithms simulated.

Our system model consists of a number of clients and servers, with given MIPS ratings, interconnected by a network. We model message latency with an average value and a coefficient of variance, modelling the distribution as Erlang, exponential or hyperexponential, depending on whether the coefficient of variance is <1, =1 or >1. Most previous studies have assumed an exponential distribution for message latency. However, latency variance can have an effect on the amount of data conflict, and it is, therefore, important to study different variances. We assume that the network is not a bottleneck at the load levels that we are modelling. We also take into account the CPU overhead to send or receive a message. We model the client overheads for the initial processing of each transaction, transaction computation after every read response and TM processing per operation. We also model the server overheads for scheduler processing per operation and I/O overhead per data item access. As in previous work, we assume that non-volatile random access stable storage is available, and sufficient I/O bandwidth is available to enable modelling the I/O server as an infinite server with a load-independent service time [Thom98b]. We believe that this assumption is reasonable for today's systems, where I/O time is improved using multiple disks and disk controllers [GSSS01], or fast non-volatile storage that is periodically flushed to disk.

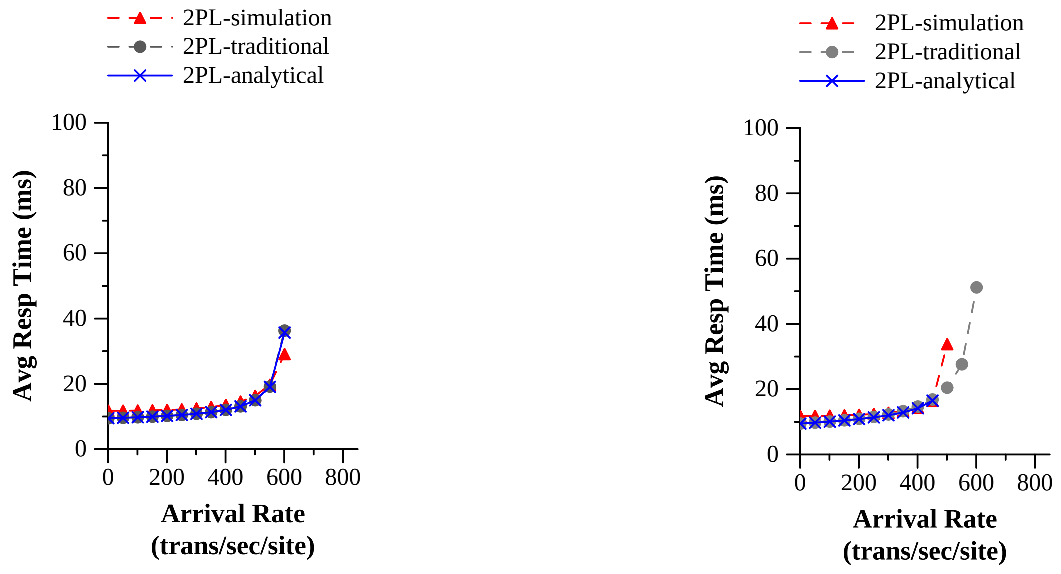
We model two different classes of workloads — a *high data contention* scenario modelled by a small database consisting of 4000 data items, and a *low data contention* scenario modelled by a larger database consisting of 32000 data items. With the same transaction load, a smaller database results in a higher degree of data contention, because the same number of transactions are competing for a smaller set of data items. We use a b-c pattern of hot spot access, where a fraction b of the accesses goes to a fraction c of the database (*hot spots*), and b > c [TaGS85]. We set the values of b and c such that high-contention data items or hot spots are accessed approximately ten times as frequently as regular data items [Thom93]. The database is partitioned equally among all the servers, with no replication, so that each data item is managed by only one server. We discuss replication in a later chapter (section 9.2). We consider an open system model with Poisson transaction arrivals at each client. We model variable-size transactions, and the size of each transaction (the number of data items it accesses) is drawn from a uniform distribution with the given bounds. We model different transaction compositions by varying the probability of a transaction's access being a `read`.

The concurrency control algorithms that we simulated are dynamic 2PL, BTO, a best-case algorithm that we call Zero-Cost (ZC), and the network-aided concurrency control techniques — ORDER and PREDICT. We describe the 2PL simulation now, and the other simulations in later chapters. Servers in our dynamic 2PL simulation do local as well as global deadlock detection using waits-for graphs. The cost of deadlock detection is set to zero, as in previous studies [Thom98a]. A zero cost for deadlock detection also removes any dependence of our results on the particular method used to handle deadlocks. With a non-zero cost for deadlock detection, the performance of dynamic 2PL would be worse. Locks can be shared or exclusive, with no upgrade from shared to exclusive. When a transaction is aborted, a new transaction is started in order to simulate a restart.

Our baseline parameters and their values are standard ones culled from recent hardware trends and performance studies of concurrency control techniques as reported in the literature, and are listed below. In addition to standard values, we experimented with faster processors (MIPS ratings) and longer transactions, in order to model current systems and workloads realistically.

- Number of clients = 8 (default), 16, 24, 32
- Number of servers = 8
- MIPS rating of each client = 200, 400 (default), 800
- MIPS rating of each server = 800
- Average message latency = 20 µs (default for µs range), 80µs, 200µs, 500µs, 1.5ms (default for ms range), 5ms, 10ms
- Coefficient of variance of message latency = 0.5, 1 (default), 2, 3
- Network overhead at client/server per message = 5K instructions
- Client's initial processing of a transaction = 100K instructions
- Client's computation per `read` = 20K instructions
- Transaction Manager overhead at client per operation = 1K instructions
- Overhead at scheduler per data item access = 1K instructions
- I/O overhead at server per data item access = 5K instructions
- I/O delay per data item access = 4 ms
- Database size = 32000 (low data contention), 4000 (high data contention)
- Number of high-contention data items (hot spots) = 1/20th of database size
- High-contention access probability = 0.33
- Mean arrival rate at each client = 1-801 transactions/second (Poisson), 151 default
- Transaction size: 8-24 (default), 16-26, 24-34, 32-42, 40-50: uniform distribution
- Probability of a transaction's access being a `read` = 0, 0.25, 0.5, 0.75 (default), 1
- Time between access requests = 300µs
- Local deadlock detection = every 10 operations arriving at scheduler
- Global deadlock detection = 100 times a second
- Deadlock detection cost = 0

We ran the simulation, our analytical model and a traditional analytical model of 2PL for different arrival rates of transactions and for different database sizes, and measured the average transaction response time. We used only the data contention component of the traditional model [TaGS85, CiDY90, CiDY92], and combined it with our hardware resource contention component, in order to get the traditional model. The aim was to discover the effect of the assumption made by traditional models that queue lengths are never greater than two. Figure 3 shows the results for the low data contention scenario and the high data contention scenario, for the default message latency of 20μs (exponential distribution). We used no hot spots in the simulation, in order to simulate the same scenario as the analytical model. Applying the traditional method of modelling hot spots resulted in an underestimation of data contention in the 4000-item database. Extending the analytical model to handle hot spots is left to future work. In later chapters, we use our simulations to study performance in the presence of hot spots.



(a) Database size = 32000 (low data contention)

(b) Database size = 4000 (high data contention)

FIGURE 3: Analytical versus simulation results for 2PL

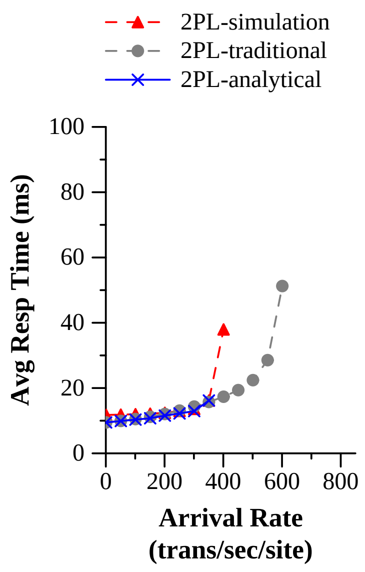
The method of independent replications was used to obtain an accurate estimate of transaction response time from the simulation. The values given in the graphs for simulation results are the midpoints of a confidence interval which is 1% of the sample mean (on each side) at a 90% confidence level. One extra point at a wider confidence interval was plotted for each curve, in order to show the performance degradation after the knee. This extra point represents an unstable region of performance, and the reader should not attach importance to the exact value of response time at this point.

As transaction arrival rate increases, response time increases slowly until a *knee*, after which response time rises steeply, and the system becomes unstable. When data contention is sufficiently low, the performance degradation at the knee is mainly due to hardware resource contention: the increased transactions compete for the limited CPU resources available. At high data contention levels, the knee occurs at a lower transaction arrival rate. As more and more transactions are introduced into the system, the data conflict probability increases, causing longer queue wait times at the schedulers and longer response times. The data conflict probability is higher in the 4000-item database because the same number of operations are competing for access to a smaller set of data items. The performance degradation at the knee of the curve in the 4000-item database is mainly due to data contention.

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. The traditional analytical model of 2PL predicts performance well in the 32000-item database, where hardware resource contention is the dominant factor affecting performance. However, the traditional model is unable to predict the degradation in performance in the

4000-item database, where data contention is the dominant factor affecting performance. The traditional model incorrectly predicts that the performance of the system is good until the point where hardware resource contention becomes high enough to degrade performance. However, the performance of 2PL degrades earlier, because of the effect of high data contention. By modelling queue lengths correctly, our model is able to predict this earlier degradation in performance due to high data contention.

The inaccuracy in prediction of the traditional model increases as data contention increases. Figure 4 shows the analytical modelling results for the high data contention when the number of clients is sixteen, which is twice the default number of clients. Increasing the number of clients increases data contention, because there are more transactions competing for the same set of data items. The traditional model still assumes that performance stays stable until a high transaction load, due to its assumption that queue lengths are less than or equal to 2. However, the high data contention causes longer queue lengths, and performance degrades at a lower transaction load, and our new analytical model is able to capture this effect accurately.



Number of clients = 16, High data contention

FIGURE 4: Increasing data contention

Our analytical model makes some simplifying assumptions that are not made in the simulation. The close agreement between the results of the analytical model and simulation requires further investigation. The next section shows that the effect of the assumptions is insignificant.

3.5 Effects of Assumptions

There are several differences between the analytical model and the simulation of 2PL. One difference is that locks are acquired statically in the analytical model of 2PL, thus using predeclaration, while the simulation assumes that locks are acquired on demand as in true dynamic 2PL. We ran the simulation with and without the assumption of predeclaration, and compared the results (Figure 5). The graphs show that the assumption of predeclaration has very little effect on the performance of 2PL. Therefore, the assumption of predeclaration does not significantly affect our model's ability to predict the performance of 2PL.

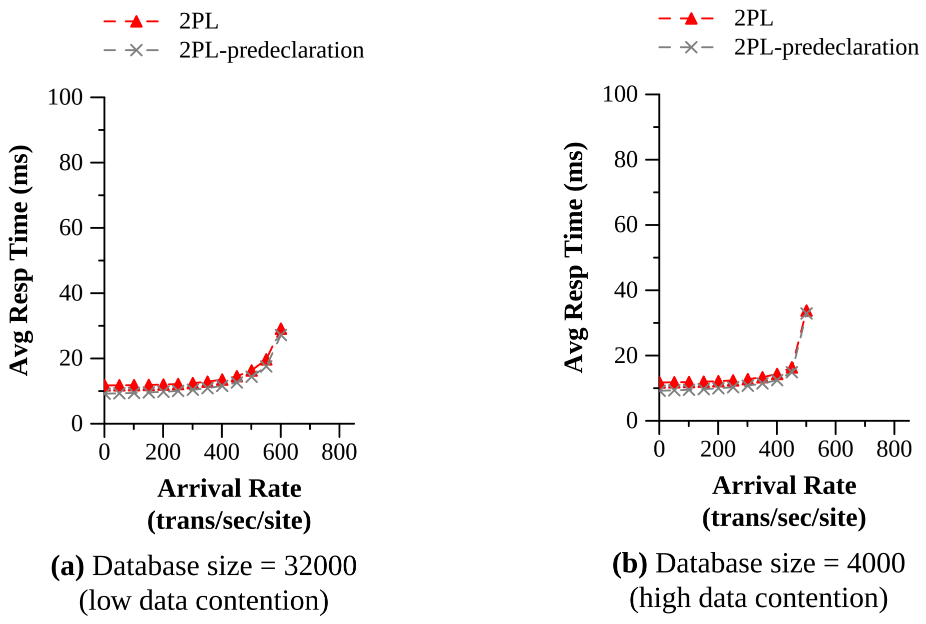
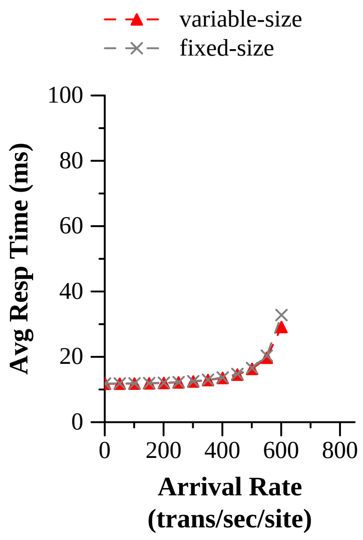
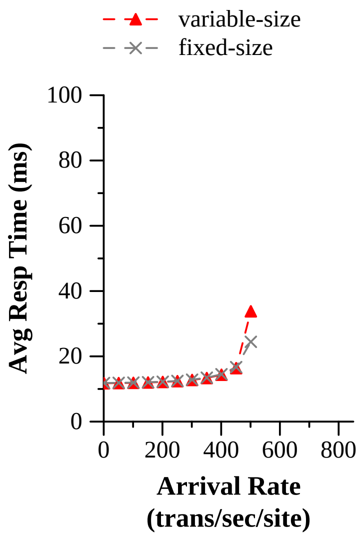


FIGURE 5: Static versus dynamic acquisition of locks in 2PL

Another difference between the analytical model and the simulation is that the analytical model assumes fixed-size transactions, while the simulation uses variable-size transactions. We ran the simulation with fixed-size transactions (of size 16), and then with variable-size transactions (of size 8-24, uniformly distributed). Figure 6 shows the results from this test. The assumption of fixed-size transactions also does not significantly affect our model's accuracy in predicting the performance of 2PL.



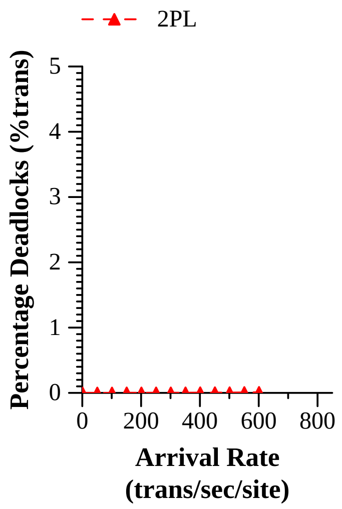
(a) Database size = 32000
(low data contention)



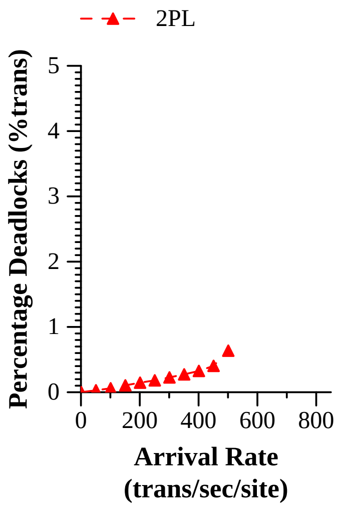
(b) Database size = 4000
(high data contention)

FIGURE 6: Fixed versus variable size transactions in 2PL

While the 2PL simulation allows for deadlocks, our analytical model assumes that the number of deadlocks is zero. We measured the percentage of deadlocks obtained in the simulation runs (Figure 7). The percentage of deadlocks is less than 0.5% even in the high



(a) Database size = 32000
(low data contention)



(b) Database size = 4000
(high data contention)

FIGURE 7: Percentage of deadlocks in 2PL

data contention case, showing that this last assumption is also a reasonable one. The analytical model tracks the simulation well, despite the simplifying assumptions made in the analysis.

3.6 Summary

Traditional analytical models for concurrency control are inadequate, in that they do not model performance accurately when data contention is high. These models can mislead database designers into believing that a DBMS will have acceptable performance under conditions that would actually cause the system to become unstable. Another limitation of traditional analyses for distributed databases is that they do not model a fully distributed database. We have presented a new analytical modelling technique that addresses these limitations. We have applied our modelling technique to the 2PL concurrency control technique, and validated the results against a simulation. We have shown that the ability to model arbitrary queue lengths allows our model of 2PL to predict performance accurately, even under high data contention. We have studied the effects of the assumptions made in our analytical model, and shown that they do not significantly affect the model’s ability to predict performance.

Chapter 4

A New Look at Timestamp Ordering

Dynamic 2PL is the most popular concurrency control technique, and current database designers use it almost exclusively [BeHG87, GrRe92, Date95]. Popular conception has been that timestamp ordering techniques like Basic Timestamp Ordering (BTO) perform poorly as compared to dynamic 2PL. In previous studies that have concentrated on centralized servers and low data contention scenarios, BTO has been shown to perform poorly as compared to 2PL, owing to BTO's high restart rate which causes it to reach a performance limit imposed by available hardware resources. In light of today's faster processors and changing workload characteristics [Fran99], it is time to reconsider BTO. We show the surprising result that the performance of BTO is significantly better than that of dynamic 2PL for a wide range of conditions, especially when there is significant data contention [SrWR01b]. 2PL outperforms BTO only when both data contention and message latency are low. With increasing throughput demands [Gray99], the high data contention case becomes important, and BTO becomes an attractive choice for concurrency control. In a later chapter (Chapter 7), we present a new timestamp ordering technique (PREDICT), that also outperforms dynamic 2PL under a wide range of conditions. These results motivate a reevaluation of the merits of timestamp ordering concurrency control in distributed databases.

In section 4.1, we describe BTO, and review previous studies of its performance. In section 4.2, we present a qualitative comparison of BTO and 2PL. In section 4.3, we describe our experimental set-up and testbeds. In section 4.4, we discuss results comparing the performance of BTO and 2PL. We conclude in section 4.5.

4.1 Basic Timestamp Ordering

In timestamp ordering concurrency control, the TM assigns a unique timestamp to each transaction, where the timestamps are values drawn from a totally ordered domain. In a distributed system, the total ordering of timestamps is usually guaranteed by the following method. Each TM is assigned a unique number — its site identifier, for example. Each TM maintains a local counter (which could be the local clock), and assigns the value of the counter to each transaction it generates, ensuring that each transaction gets a unique counter value. A timestamp is now the ordered pair consisting of the counter value followed by the TM's unique number. Timestamps are, therefore, totally ordered, first by their counter value and then, in case of ties, by their unique TM numbers.

The TM attaches a transaction's timestamp to each `read` and `write` operation issued by the transaction. A timestamp ordering scheduler ensures that transaction operations are executed in order according to their timestamps. BTO is an aggressive timestamp ordering technique. A BTO scheduler delivers transaction operations to the DM in timestamp order, rejecting any operation that arrives too late (after a conflicting operation with a higher timestamp has already been delivered to the DM). A rejected operation results in the transaction being aborted and restarted with a new and larger timestamp.

We assume that timestamps are stored in high-speed memory, and that the probability of loss of timestamps is low. Since timestamp management does not involve disk access, timestamp management is fast. Storing timestamps in main memory involves the risk of loss of timestamp information in case of system failures. However, such failures do not occur frequently. Moreover, the timestamps in main memory can be periodically saved to stable storage in an asynchronous manner, thereby reducing the risk of lost timestamps.

Previous performance studies have concentrated on centralized servers and low data contention scenarios. 2PL has been shown to outperform BTO in a centralized database system [RyTh90b]. This result can be explained as follows. If there are multiple servers, timestamps are useful in order to make the servers execute transactions in the same order. However, in the case of a single server, it has been found that it is better to let the serialization order be determined by the *arrival* of operations rather than by a predetermined ordering. Moreover, dynamic 2PL's deadlock detection costs are low in a centralized database, because the server does not have to communicate with other servers in order to detect deadlocks. Another study has compared 2PL to BTO in a system with a single server connected to one or more clients over a network [LiNo83], and reports that 2PL outperforms BTO.

BTO has been reported to perform worse than 2PL in a distributed database, in a low data contention scenario [CaLi91]. The non-replicated case in the study assumes that all of a transaction's accesses are local. The replicated cases assume that the primary copies of all data items accessed by a transaction are located at a single site. There have been no prior studies of the performance of BTO in a truly distributed database environment where a transaction can access data items at multiple sites. Performance studies of BTO and 2PL under high levels of data contention are also unavailable. Larger transactions are becoming important in today's workloads, making the high data contention scenario an important one [Fran99]. Moreover, Gray envisions increasing throughput demands, as C2C (computer-to-computer) transactions become more common [Gray99]. Higher throughput demands can lead to higher degrees of concurrency, leading to high data contention. Even as databases get larger, hot spots or frequently-accessed sets of data items still exist, and the high data contention scenario remains important. In later sections, we study the performance of BTO and 2PL in a distributed database environment, under both low and high levels of data contention, and show that BTO outperforms dynamic 2PL under a wide range of conditions.

4.2 Qualitative Comparison of BTO to Dynamic 2PL

An important difference between BTO and dynamic 2PL is that in 2PL, a lock is acquired as a result of a `read` operation, whereas in BTO, a `read` does not result in lock acquisition. A BTO scheduler delivers a `read` to the DM as soon as the `read` reaches the head of the queue. A 2PL scheduler must hold a read lock for transaction `t` until commit time in order to ensure that no other transaction invalidates data items that transaction `t` has read, before `t` commits. A BTO scheduler does not need a read lock because the `read`s and `write`s of a transaction carry the same timestamp, and this timestamp determines the ordering of the transaction with respect to other transactions. In both BTO and 2PL, `write` operations have the same effect of making the data item unavailable until the `writes` are committed. In 2PL, a write lock is held until the corresponding `commit` arrives. In BTO, when a `write` reaches the head of the queue in the scheduler, it is executed by the DM only after the corresponding `commit` arrives. On commit, a client in 2PL must send lock releases to the servers in order to release the read locks. BTO does not incur this cost since it does not use read locks.

2PL and BTO also differ with respect to deadlocks. Deadlocks cannot occur in BTO, because transaction operations are executed in timestamp order, and transaction timestamps are drawn from a totally ordered domain. Therefore, deadlock detection is unnecessary in BTO, while servers in 2PL incur the cost of either preventing deadlocks, or of detecting and breaking them. In a distributed database system, deadlock detection involves both local and global deadlock detection, the latter involving communication among multiple servers.

Both 2PL and BTO have a restart cost, but restarts in 2PL are caused by the need to break deadlocks, while restarts in BTO are caused by conflicting operations arriving out of timestamp order. Previous studies have shown that deadlocks in 2PL are rare [Thom98a],

	Dynamic 2PL	BTO
Predeclaration required?	No	No
Writes make data unavailable?	Yes	Yes
Reads make data unavailable?	Yes	No
Read lock releases required?	Yes	No
Deadlocks	Yes	No
Restarts	Yes (to resolve deadlocks)	Yes (operations arriving out of timestamp order)
Additional costs	Deadlock-detection cost	Timestamp management

TABLE 2: Dynamic 2PL and BTO

but restarts in BTO occur with significant frequency [CaLi91]. The overall effect of the above differences on transaction processing performance is explored in section 4.4. Table 2 summarizes the differences between dynamic 2PL and BTO.

4.3 Experimental Set-up and Testbeds

We studied the relative performance of BTO and dynamic 2PL through a set of simulations. We described our simulation model in the previous chapter in section 3.4. The concurrency control algorithms that we simulated include BTO, dynamic 2PL and a best-case algorithm that we call Zero-Cost (ZC). ZC assumes that there are no data conflicts and no concurrency control overhead. In ZC, `read`s and `write`s are sent by the client to the servers, and are delivered to the DM immediately, without being ordered at the scheduler. There are no locks, and consequently no lock-set or lock-release messages. ZC allows us to isolate the effects of hardware resource contention from data contention and concurrency control costs. Note that ZC guarantees a correct execution only in the absence of data conflicts. Servers in our dynamic 2PL simulation do local as well as global deadlock detection using waits-for graphs. The cost of deadlock detection is set to zero, as in previous studies [Thom98a]. With a non-zero cost for deadlock detection, the performance of dynamic 2PL would only be worse, strengthening our conclusion that BTO outperforms 2PL for a wide range of conditions. A zero cost for deadlock detection also removes any dependence of our results on the particular method used to handle deadlocks. Locks can be shared or exclusive, with no upgrade from shared to exclusive. When a transaction is aborted in any scheme, a new transaction is started in order to simulate a restart. Our baseline parameters and their values were listed in section 3.4.

4.4 Performance Evaluation

We ran extensive tests on our set of simulations and compared the performance of BTO and dynamic 2PL for a wide range of system parameters and workloads. Our tests were broadly divided into two sets, based on the workload class simulated — the high data contention scenario (database size = 4000) and the low data contention scenario (database size = 32000). For each of these workload classes, we studied the effect of varying several parameters: transaction arrival rate, message latency, latency variance, processor speeds, number of clients, transaction size and transaction composition. We first discuss the high data contention workload experiments.

4.4.1 High Data Contention Scenario

Figure 8 shows the effect of varying the arrival rate of transactions on average transaction response time in ZC, 2PL and BTO, for the default millisecond message latency of 1.5ms. As transaction arrival rate increases, response time increases for all three techniques, reaching a *knee* and then rising steeply as performance degrades and the system goes into an unstable region. The knee for ZC is entirely due to hardware resource contention. The increased number of transactions compete for limited CPU resources, causing a performance degradation when the available resources are insufficient to sustain the transaction load. For 2PL and BTO, the knee occurs at a lower arrival rate than for ZC,

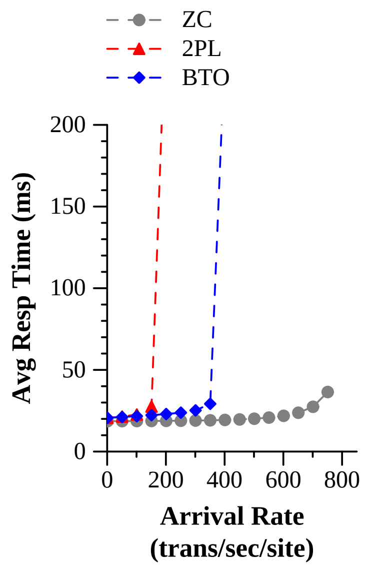


FIGURE 8: Transaction arrival rate (high data contention)

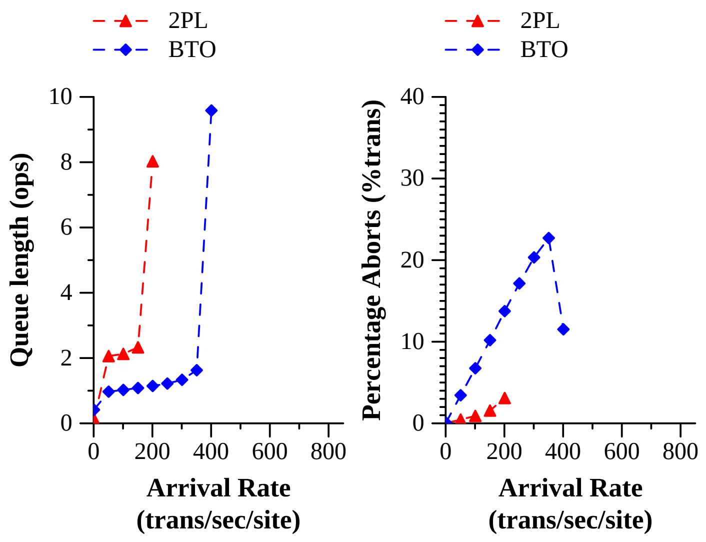
because transactions in 2PL and BTO encounter data contention in addition to hardware resource contention.

The method of independent replications was used to obtain an accurate estimate of transaction response time. The values in the graphs are the midpoints of a confidence interval which is less than 5% of the sample mean (on each side) at a 90% confidence level. One extra point at a wider confidence interval was plotted for each curve, in order to show the performance degradation after the knee. This extra point represents an unstable region of performance, and the reader should not attach importance to the exact value of response time at this point.

Two important factors that affect the performance of concurrency control techniques are queue lengths at the schedulers, and restart behaviour. As queue lengths increase, operations have to wait longer. The increased operation wait time increases transaction response time. The increased transaction response time increases data-holding time, which causes the queues to get longer. This iterative build-up of data contention is an important cause of performance degradation in both 2PL and BTO. The amount of restart behaviour is the other important factor that affects performance. A large number of restarts puts pressure on the hardware resources available, and degrades performance when the available resources are insufficient to sustain the load. As processor speeds increase, performance becomes less sensitive to restarts, and it becomes necessary to reconsider restart-oriented techniques like BTO which have been perceived as poor performers until now.

BTO significantly outperforms 2PL for the high data contention workload class. BTO’s sustainable transaction load is of order of twice that of 2PL. The reason for this surprising result is found through an examination of the queue lengths at the schedulers. Figure 9a shows the average queue length encountered by a transaction operation at the scheduler, for dynamic 2PL and BTO. Recall that `reads` do not lock data items in BTO. Consequently, the data-holding time in BTO is lower than that in 2PL, keeping queues shorter at a BTO scheduler. As a result of the shorter queues, the iterative build-up of data contention is less severe in BTO, allowing BTO to significantly outperform 2PL.

Restarts increase steadily with transaction arrival rate in BTO, whereas in 2PL, the increase in restart rate is more gradual. At today's processor speeds, restarts have to increase to a significant amount before they degrade performance due to hardware resource contention. Therefore, the performance degradation due to hardware resource contention in BTO is delayed because of the high speed of modern processors. On the other hand, the iterative build-up of data contention in 2PL is high, and causes the performance of 2PL to degrade earlier than that of BTO.



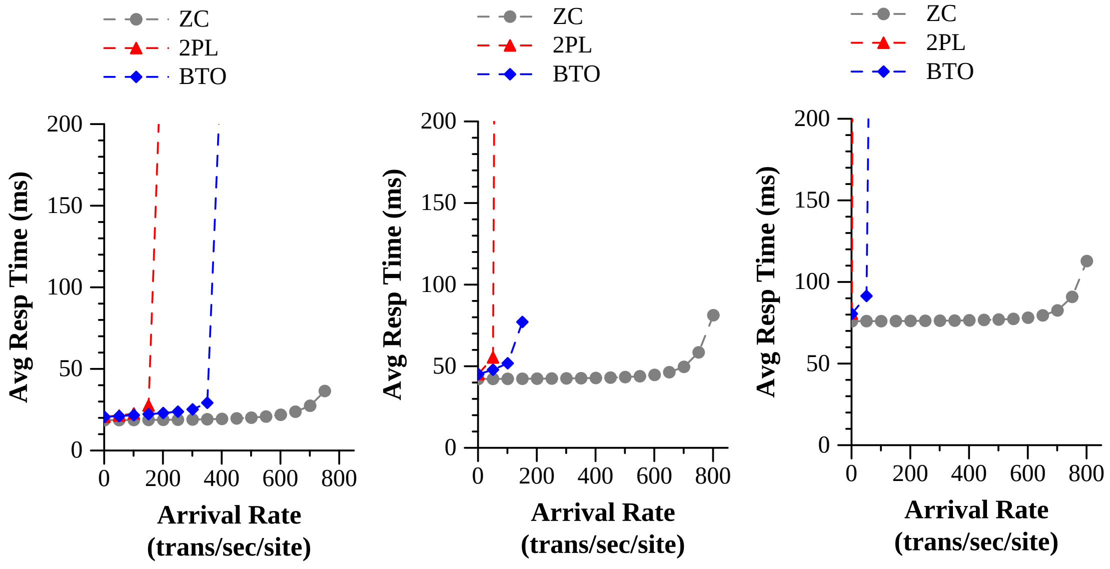
(a) Average queue length **(b)** Percentage restarts

FIGURE 9: Queue lengths and restarts (high data contention)

The hardware resource contention in BTO is affected by the rising queue lengths, as evidenced by the reversal in the steep rise of restarts (Figure 9b). The reason for this reversal is that the average queue length increases, increasing operation wait time on the scheduler queues. The effect of the increased operation wait time is to allow late-arriving operations to be inserted into the queue in timestamp order. This effect reduces the number of rejected operations and, therefore, the number of restarts in BTO.

4.4.1.1 Message Latency

Figure 10 shows the effect of different message latencies in the millisecond range on ZC, 2PL and BTO. As the message latency increases, operations take longer to arrive, and



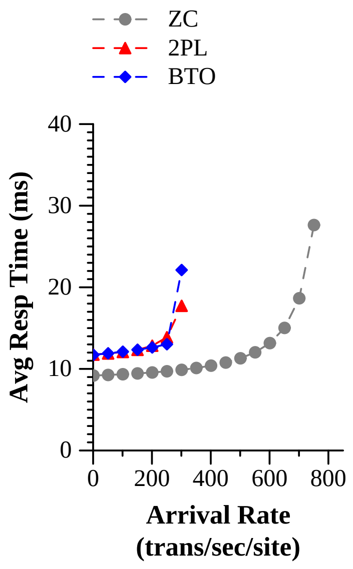
(a) Avg. latency = 1.5ms **(b)** Avg. latency = 5ms **(c)** Avg. latency = 10ms

FIGURE 10: Message latency (milliseconds)
(high data contention)

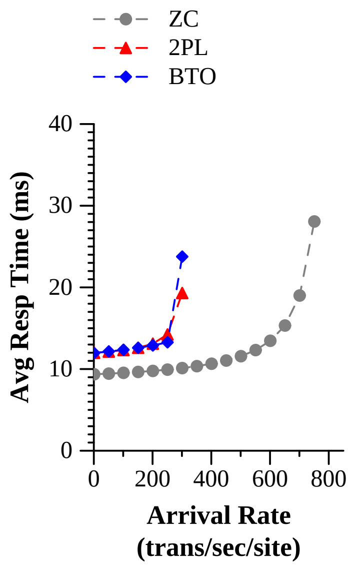
the data-holding times increase for all three techniques, causing response time to increase. Therefore, the average response time is higher at higher message latencies, for all three techniques. Moreover, in 2PL and BTO, a higher message latency causes a higher data-holding time, which in turn causes longer queues and a higher probability of data conflict. As described earlier, an iterative build-up of data contention occurs. Therefore, as the message latency increases, the performance knee occurs at lower arrival rates for 2PL and BTO due to data contention. Since queues at a 2PL scheduler are longer than those at a BTO scheduler, the iterative build-up of data contention is worse for 2PL. Therefore, the knee of the 2PL performance curve occurs at a lower load than the knee of BTO. This allows BTO to outperform 2PL at all three message latencies. However, both techniques perform poorly at high latencies. At the highest latency of 10ms, the performance of dynamic 2PL is so poor that it cannot even sustain an arrival rate of 51 transactions per second.

We also studied performance for message latencies in the microsecond range. The graphs in Figure 11 a, b, c and d show the results of these experiments. With lower message latencies, the iterative build-up of data contention is less severe, and the performance difference between BTO and 2PL decreases.

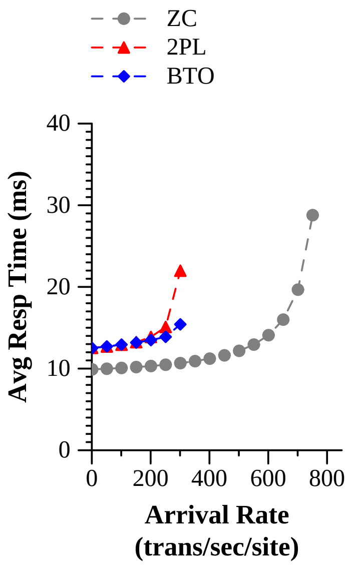
At the highest latency of 500μs, BTO outperforms 2PL, but its sustainable transaction load is only about an order of



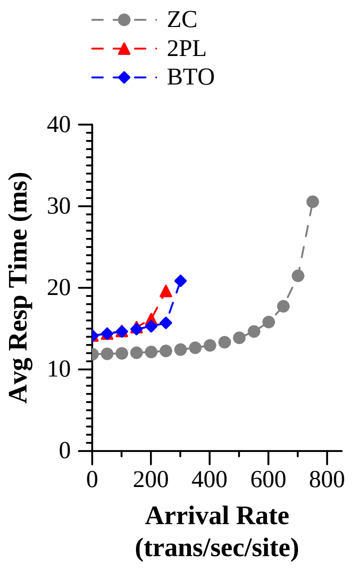
(a) Avg. latency = 20μs



(b) Avg. latency = 80μs



(c) Avg. latency = 200μs



(d) Avg. latency = 500μs

FIGURE 11: Message latency (microseconds)
(high data contention)

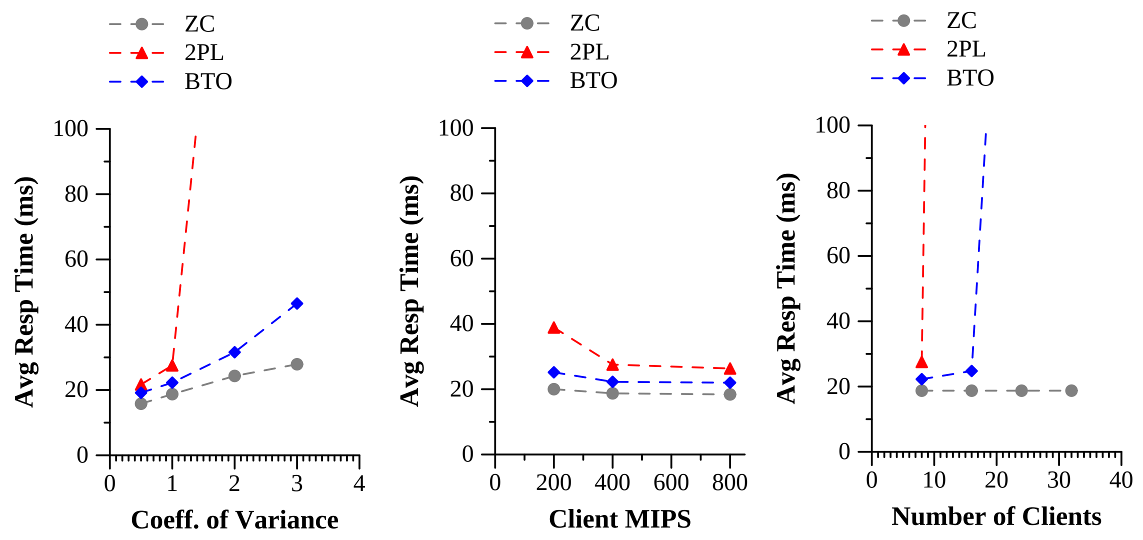
0 to 50% higher than that of 2PL. At the lower latencies, the performance curves of BTO and 2PL are similar.

4.4.1.2 Latency Variance

The previous experiments assumed a coefficient of variance of 1, which translates to an exponential distribution for message latency. In the following experiment, we studied the effects of a hyperexponential distribution (coefficient of variance > 1) and an Erlang distribution (coefficient of variance < 1) for message latency. Figure 12a shows the performance of 2PL and BTO for different variance values. As the variance increases, the size of the time window within which a transaction operation may arrive increases, increasing the probability of data conflict. This increase in data conflict is manifested as an increased queue length in the 2PL scheduler. The increase in queue length in BTO is less for two reasons: `read`s don't hold data items; and conflicting operations that arrive too late are aborted rather than queued. Therefore, 2PL degrades faster than BTO as latency variance increases.

4.4.1.3 Processor Speed

Hardware resource contention affects performance. As processor speeds increase, resource contention decreases for all three techniques. At high processors speeds, hardware resource contention becomes low enough that performance differences among the different techniques are determined primarily by data contention. Figure 12b shows the effect of different client processor speeds on the performance of ZC, 2PL and BTO. A reduction in processor speed from 400 MIPS to 200 MIPS degrades the performance of 2PL more than that of BTO, because the increased hardware resource contention increases



(a) Latency variance **(b)** Processor speed **(c)** Number of clients

FIGURE 12: Latency variance, processor speed and number of clients
(high data contention)

transaction response time. The increased response time causes longer data-holding times and, therefore, a higher probability of data conflict. Once again, the iterative build-up of data contention is more severe in 2PL than in BTO due to the difference in queue lengths. An increase in processor speed from 400 to 800 MIPS did not significantly improve performance, and therefore, we concluded that we had explored the question of the benefit of increased processor performance sufficiently.

4.4.1.4 Number of Clients

Figure 12c 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. One effect of the increased data conflict is that queue lengths in the 2PL schedulers increase, causing an iterative build-up of data contention. In addition, the probability of deadlock in 2PL increases, and the percentage of aborts consequently increases.

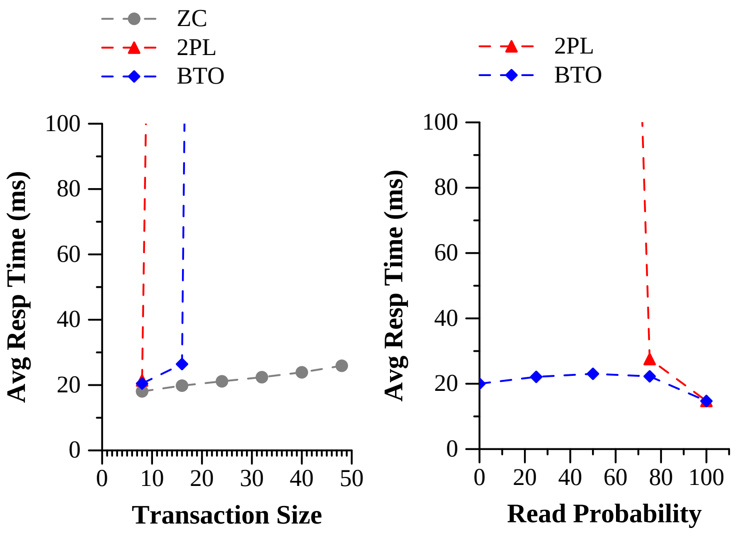
The increased data conflict also affects BTO, and this effect is manifested in an increased percentage of aborts. Once again, the lower queue lengths in BTO allow it to outperform 2PL. Increasing the number of clients also increases hardware resource contention at the servers, by increasing the load on the servers. However, the ZC curve shows that this effect is negligible.

4.4.1.5 Transaction Size

Figure 13a 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-holding time and, therefore, data conflict. The increased data conflict increases the probability of deadlock in 2PL, and the percentage of restarts in BTO. As before, the lower queue lengths in BTO allow it to outperform 2PL.

4.4.1.6 Transaction Composition

Figure 13b shows the effect of varying the read probability while transaction size is held constant. More `reads` implies that the queue length and data-holding time decrease, because read locks are shared (in 2PL) and `reads` don't hold data items



(a) Transaction size **(b)** Read probability

FIGURE 13: Transaction size and composition (high data contention)

(in BTO). When the read probability is 100%, there are no `write`s, and therefore the disk access time of the second phase is absent for all the techniques. Therefore, the response time is low when the probability of read is 100%.

As read probability decreases, there are fewer `read`s and more `write`s in the system, increasing data conflict. The increase in data conflict causes an increased queue length in 2PL. The increase in queue length in BTO is less because conflicting operations that arrive too late are aborted rather than queued. At this default load of 151 transactions per second, the aborts in BTO are not high enough to cause a degradation in performance. However, 2PL is unable to sustain the load for a read probability of less than 75%.

4.4.2 Low Data Contention Scenario

We now discuss the low data contention workload experiments. Figure 14 shows the effect of varying the arrival rate of transactions. The performance knee of BTO occurs at a lighter transaction load than the knee for 2PL. The performance advantage that BTO had over 2PL in the high data contention workload class is not apparent in this low data contention scenario. 2PL actually performs better than BTO. However, the improvement in sustainable transaction load is only of the order of 10% over that of BTO.

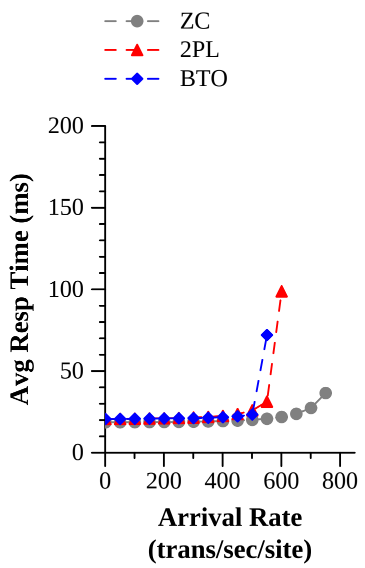


FIGURE 14: Transaction arrival rate (low data contention)

Figure 15 plots the queue lengths and percentage of restarts for the two techniques. While the queue lengths in BTO are still lower than those in 2PL, queue lengths in both techniques stay low, unlike the trend in the high data contention scenario. Consequently, data contention is no longer the overriding effect in determining performance. In this low data contention scenario, hardware resource contention becomes an important factor. The hardware resource contention is higher in BTO than in 2PL, because of BTO's high percentage of restarts (Figure 15b). As arrival rate increases, more and more operations arrive out of order in BTO and are rejected. The large number of restarts combined with the high resource contention at the client due to this high load degrades the performance of BTO. The percentage of restarts in 2PL is much lower than that in BTO, since deadlocks are not very frequent. Recall from section 4.4.1 that a secondary effect of increased queue lengths reversed the rise of restarts in BTO. This effect is not present in the low data contention case, because queue lengths are not high enough to result in this effect.

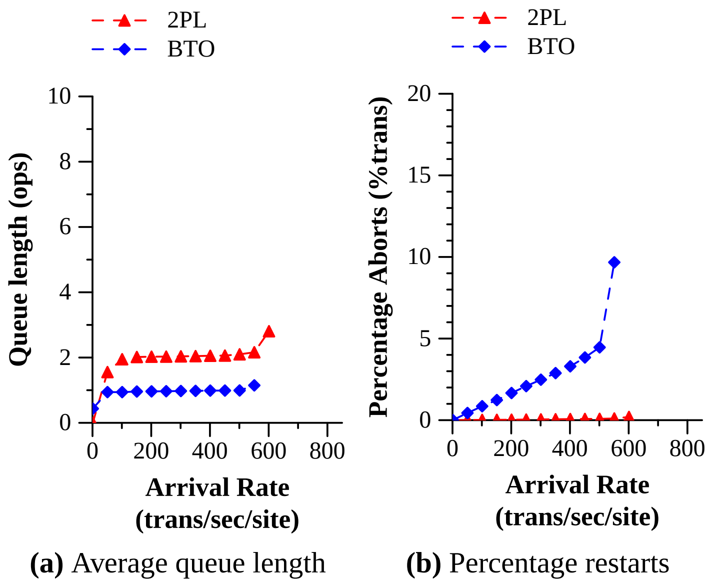
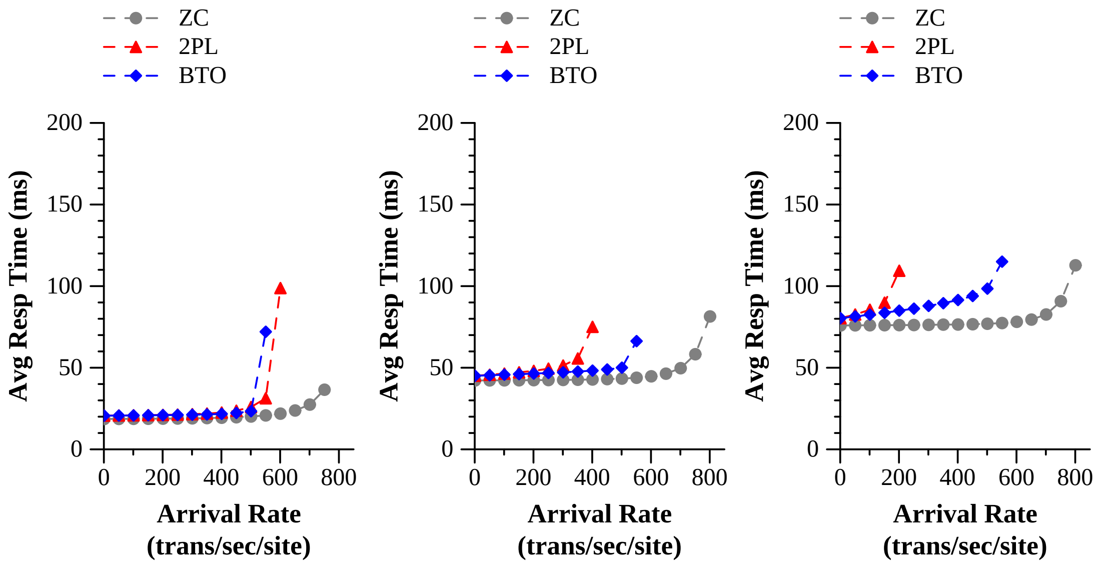


FIGURE 15: Queue lengths and restarts (low data contention)

4.4.2.1 Message Latency

We explored the effect of message latency on performance in this low data contention scenario. Recall from the high data contention workload experiments (section 4.4.1.1) that

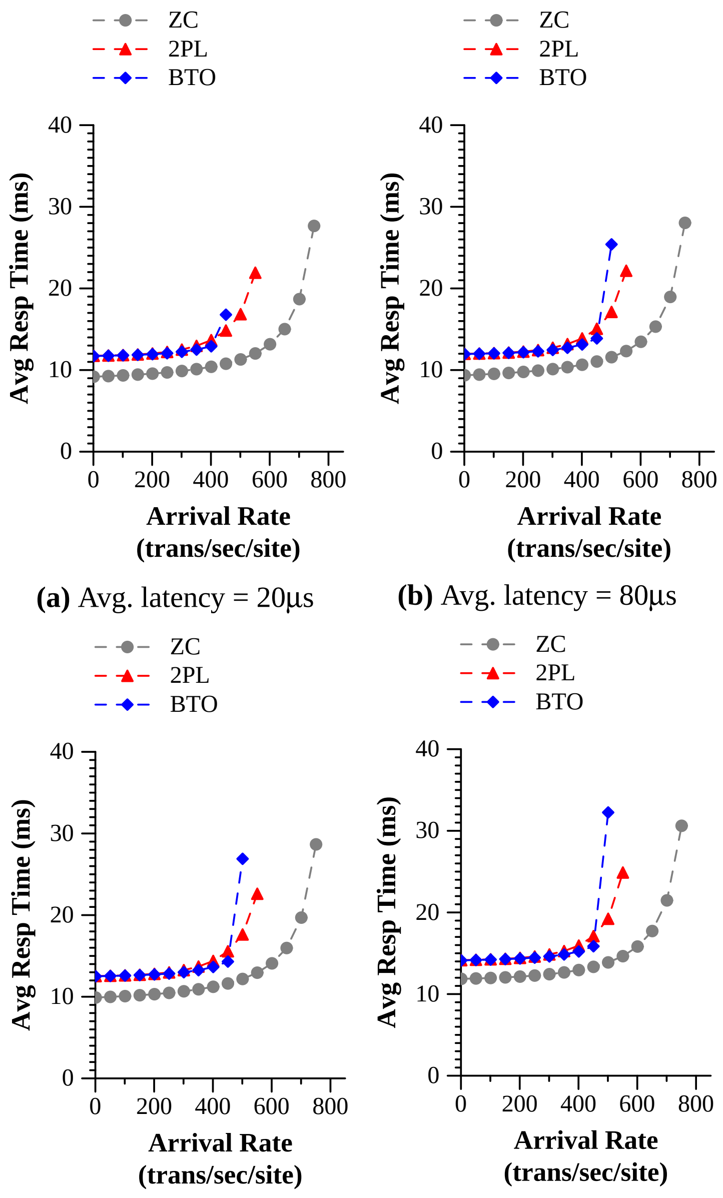


(a) Avg. latency = 1.5ms **(b)** Avg. latency = 5ms **(c)** Avg. latency = 10ms

FIGURE 16: Message latency (milliseconds)
(low data contention)

increasing the message latency caused increased data contention which caused the performance of 2PL to degrade faster than that of BTO. Low message latencies were the only cases in which data contention was low enough to let 2PL perform as well as BTO. The trends are similar in this low data contention scenario, except that 2PL gains an advantage over BTO at certain message latencies.

Figure 16 shows the effect of different latencies in the millisecond range. BTO performs slightly worse than 2PL for the 1.5ms latency, but outperforms 2PL at the higher latencies. This is the same trend that was seen in the latency experiments in section 4.4.1.1. As message latency increases, the iterative build-up of data contention in 2PL becomes severe, and its performance degrades quickly. BTO is able to outperform 2PL because of its short queues. At the highest latency, BTO’s sustainable transaction load is of an order of 200% better than that of 2PL.



(a) Avg. latency = 20μs **(b)** Avg. latency = 80μs

(c) Avg. latency = 200μs **(d)** Avg. latency = 500μs

FIGURE 17: Message latency (microseconds)
(low data contention)

The graphs in Figure 17 a, b, c and d show the results of the experiments for message latencies in the microsecond range. This time, 2PL slightly outperforms BTO for all the latency values. The degree of data contention is so low for this workload class at these low message latencies that hardware resource contention begins to contribute significantly to the performance difference between BTO and 2PL. Since restarts in BTO are more frequent than in 2PL, BTO suffers from high hardware resource contention, and the performance of BTO degrades at a lower load than that of 2PL. 2PL's sustainable transaction load is of an order of 10% to 25% better than that of BTO, at these latencies.

4.4.2.2 Other Parameters

We studied the effect of other parameters such as latency variance, processor speed, number of clients, transaction size and transaction composition. The results were similar to the ones presented in the high data contention scenario (section 4.4.1).

4.5 Conclusion

Popular conception has been that timestamp ordering techniques perform poorly as compared to 2PL. We have shown the surprising result that the performance of BTO is better than that of 2PL for a wide range of conditions. BTO outperforms 2PL under high data contention for all cases except when message latency is very low. When message latency is low, the performance of 2PL is similar to that of BTO. Under low data contention, BTO performs better than 2PL when message latencies are high. However, the performance of 2PL is slightly better than that of BTO when message latency is low. Table 3 summarizes the performance differences between 2PL and BTO. The symbol > translates to "performs better than", and the symbol = translates to "performs as well as."

The iterative build-up of data contention is an important factor determining performance, and is more severe in 2PL than in BTO, due to longer queue lengths. Hardware resource contention is also an important factor determining performance, and is

Conditions	Relative Performance
High data contention workload, low message latencies	BTO ≥ 2PL
High data contention workload, high message latencies	BTO > 2PL
Low data contention workload, low message latencies	2PL > BTO
Low data contention workload, high message latencies	BTO > 2PL

TABLE 3: Performance comparison of dynamic 2PL and BTO

more severe in BTO than in 2PL, due to a high percentage of restarts. Small increases in data contention lead to significant performance effects, because of the iterative build-up phenomenon. Increasing factors such as message latency, transaction size, number of clients and probability of write increases data contention, and causes the performance of 2PL to degrade rapidly. When data contention is low, hardware resource contention begins to affect performance differences more, and BTO can perform worse than 2PL for some cases. We have shown a wide range of conditions for which BTO performs better than 2PL. The high data contention scenario is an important workload class, and we have shown that BTO is better than 2PL for this workload class. 2PL performs better than BTO only when the workload and system parameters are chosen to keep the data contention extremely low. These results motivate the reevaluation of timestamp ordering techniques for concurrency control.

Chapter 5

Network Properties

The network can act as a powerful coordination mechanism by providing certain properties that are useful to concurrency control of distributed transactions. A network that provides these useful properties at a low cost can be used as the basis for efficient and scalable concurrency control. We have identified several such properties — total ordering, predictability, extended predictability, pruning and caching. In this chapter, we describe these useful properties, and discuss how they can help with concurrency control. For each property, we discuss the feasibility of its implementation and the kinds of systems where the property would be useful. In later chapters (Chapter 6 and Chapter 7), we examine two of these properties — total ordering and predictability — in detail, presenting concurrency control techniques based on them and evaluating the performance of these techniques.

5.1 Total Ordering

Total ordering guarantees that messages are delivered in the same order at all the destinations of that message. 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 such as the isotach network [ReWW97, Will93], that provides total ordering at a low cost can be used as the basis for efficient concurrency control of transactions.

An example of a concurrency control technique that uses total ordering works as follows. The technique uses predeclaration of all accesses. A client starts the execution of a transaction by constructing an atomic action composed of all the `read`s and `predeclare`s of the transaction. The network delivers atomic actions in a total order with respect to one another. Each server executes conflicting operations respecting the total order among the atomic actions. An atomic action effectively reserves a consistent time slice across servers. Clients then execute the transactions by examining `read` responses, performing computation, deciding on a commit or abort, and sending the commit or abort decision to the servers. These messages do not have to be delivered with the total ordering constraint. The total ordering of the atomic actions representing the transactions guarantees serializability.

It is feasible to implement the total ordering property. An example of a network that provides total ordering at a low cost is the isotach network. Other systems that implement total ordering include Isis [BiJo86], Totem [MMAB96], Transis [DoMa96] and publications [NeTo93]. There is a cost to total ordering, which can be manifested as extra messages, extra information piggybacked on existing messages, an extra round of communication between the sender and the destinations, or a restriction that only one sender can send a totally ordered multicast at any time. The total ordering property and a concurrency control technique based on the property are studied in detail in Chapter 6.

5.2 Predictability

Predictability is the ability of the sender of a message to predict the time at which the message will have arrived at its destinations, where *time* is either real-time or logical time in a time frame valid for the system. This ability can be used to give a server an estimate of how safe it is to execute an operation. The server can use this estimate to make intelligent trade-offs between idle wait time and the cost of dealing with a misordering of transactions. In other words, the server has some control over balancing lost opportunity cost against restart cost. If the server manages to keep the two costs low, concurrency control can be very efficient.

A concurrency control technique based on the property of predictability works as follows. The technique requires a transaction to predeclare all of its accesses. A client starts the execution of a transaction by constructing an atomic action composed of all the `read`s and `predeclare`s of the transaction. The network assigns a unique timestamp `t` to this atomic action and stamps all messages of the atomic action with this timestamp. The network guarantees that the messages constituting the atomic action will all arrive at their destinations at or before time `t`. A server can be conservative and execute the operations at their scheduled times. Alternatively, the server can be aggressive and execute operations before their scheduled times, aborting transactions if a misordering is discovered. The potential amount of restart behaviour can be estimated and controlled by the difference between the timestamp of an operation and the time at which the operation is executed by the server, possibly combined with information about the frequency of access of the data item being accessed. Since all servers execute conflicting operations in order by the timestamps of the operations, serializability is achieved.

It is feasible to implement the property of predictability in both logical time and real-time. An example of a network that provides logical-time predictability at a low cost is the isotach network. In an isotach system, a processor has the ability to control the logical time at which the messages that it sends are delivered, by controlling the logical time at which it sends messages. Thus an isotach network implements the property of predictability. Many systems provide an imperfect form of real-time predictability, by guaranteeing that a large percentage of messages are delivered within a certain amount of time. As we will show in Chapter 7, this imperfect form of predictability can be utilized by a concurrency control technique. Since `read` responses and commit and abort messages do not have to be delivered with the predictability constraint, quality of service techniques can be used to guarantee predictable delivery for a limited class of messages. The predictability property and a concurrency control technique based on the property are studied in detail in Chapter 7.

5.3 Extended Predictability

Extended predictability is the ability of the sender of a message to predict the time before which a response to the message will arrive, where time is either real-time or logical time in a time frame valid for the system. This ability allows a sender to infer the occurrence of certain events by the passage of time, without actually receiving a message informing the sender of the event. Extended predictability also allows a client to keep the data-holding time bounded, as shown in the following concurrency control algorithm based on extended predictability.

A client starts the execution of a transaction by sending the `read`s of the transaction to the appropriate servers. On receiving responses to the reads, the transaction decides to either commit or abort. In the case of an abort, the client does nothing. Otherwise, the client constructs an atomic action from the `read`s and the `write`s of the transaction. For convenience of discussion, we call these `read`s `reread`s. The client stamps all operations in the atomic action with the current time and its site identifier. Each server maintains two values — maxRTS and maxWTS — for every data item that it stores. These values are the maximum timestamps of the `read` and `write` operations that the server has executed. The server discards a `write` if the timestamp of the `write` is less than maxRTS or maxWTS for that data item. The server discards a `reread` if the timestamp of the `reread` is less than maxWTS for that data item, or if the queue is not empty. Otherwise, the server responds to `reread`s and acknowledges the `write`s immediately. Due to the property of extended predictability, any `reread` responses and `write` acknowledgements must arrive at the client by a predetermined amount of time. There are two possible scenarios:

1. `Reread` responses and `write` acknowledgements arrive in time, and the values of the `reread` responses are the same as the values returned by the initial reads. The client commits the transaction.
2. `Reread` responses and `write` acknowledgements do not arrive in time, or the values of the `reread` responses are different from the values returned by the initial `read`s. The client aborts the transaction.

Since all servers execute conflicting operations in order by the timestamps of the operations, serializability is achieved.

It is feasible to implement the property of extended predictability in both logical time and real-time. Recall that the isotach network provides logical-time predictability at a low cost. Extending the logical time system to host level implements the property of extended predictability [LaMy00]. Estimates of maximum message latency and end-system message-handling costs can be used to provide an imperfect form of real-time extended predictability. This imperfection in the extended predictability guarantee does not affect the accuracy of the above concurrency control algorithm, but may affect performance by increasing the restart behaviour. Since initial `read`s, `read` responses, `commit`s and `abort`s do not have to be delivered with any predictability constraint, quality of service techniques can be used to guarantee predictable delivery for a limited class of messages.

5.4 Pruning

Pruning cuts message paths short by discarding unnecessary messages before they are delivered to their destinations. Network routers can prune messages by storing control information like timestamps and rejecting operations early in their paths. This property saves network bandwidth, and also reduces the load at the servers and the wait time at the clients. Pruning has been used in other contexts. Time-to-live counters are used in IP in order to prune the paths of datagrams.

An example of a concurrency control technique that uses pruning is as follows. The algorithm works like BTO, with the following differences. The server periodically updates designated routers with control information in the form of the maxRTS and maxWTS values of data items. A router stores this control information for a few data items frequently accessed by clients that are connected to it. When a client sends an operation to a server, the router intercepts the operation and rejects it if its timestamp is less than maxWTS (or maxRTS for a write) for the data item. Otherwise, the router passes the operation up to the server. If the overhead of updating routers with control information is low enough, pruning saves network bandwidth by pruning the paths of some of the messages that are going to be rejected by the server. In addition, the early reject reduces the wait time at clients. Finally, the network reduces the load on the servers by taking over some of the servers' tasks. Other concurrency control techniques like OCC using timestamps and dynamic 2PL with timestamp-based deadlock prevention (Wound-Wait) can also benefit from pruning.

The above discussion assumes that there is sufficient processing power and storage space at the network routers or switches, and that the added complexity at the routers is not enough to offset the performance gains. Pruning is useful when network bandwidth is at a premium, and when the clients are geographically distant from the servers, making communication between the servers and the clients expensive.

5.5 Caching

The network can cache and/or prefetch data and associated control information. This property reduces the wait time at clients and the load at servers, and saves network bandwidth. If the network can do caching, it can service clients' requests instead of the server.

A concurrency control technique that uses caching works as follows. The scheme is similar to OPT, with the following differences. A router caches the values and version numbers of some data items as `read` responses traverse the network. When a client requests a `read` on a cached data item, the router responds to the `read` with the value and version number of the cached data item. When a transaction is ready to commit, it sends a certification request to the servers. Each server checks the read and write sets of currently active transactions. If the transaction has read a version that is not the last committed version of the data item, or if the transaction has written a version that is lower than the last committed version of the data item, then the transaction is aborted. Otherwise, all involved servers enter a two-phase commit protocol and try to certify the transaction. The caching property helps to reduce the wait time for `read`s at the clients, and reduces the usage of network bandwidth since the reads do not travel all the way to the servers. Finally, the network reduces the load on the servers by taking over some of the servers' tasks.

The above discussion assumes that there is sufficient processing power and storage space at the network routers or switches or network interfaces. Caching is useful when network bandwidth is at a premium, and when the clients are geographically distant from the servers, making communication between the servers and the clients expensive.

5.6 Conclusion

There are several network properties that are useful to concurrency control. If the network can provide such a property at a low cost, efficient concurrency control techniques can be built on top of it. We have presented five useful network properties — total ordering, predictability, extended predictability, pruning and caching. We have discussed how each of these properties can help with concurrency control, and presented concurrency control techniques built on top of these properties. In the following chapters, we will explore exploiting two of these properties — total ordering and predictability — in detail.

Chapter 6

Total Ordering

Total ordering is a property that guarantees that messages are delivered in the same order at all destinations. This ordering guarantee can be exploited by distributed databases to ensure that transactions are viewed in the same order at all servers. A network that provides total ordering at a low cost can be used as the basis for efficient concurrency control of transactions. We present a new concurrency control technique — ORDER — that uses the interconnection network in a distributed database as an aid to concurrency control [SrWR01a]. We analyze the performance of ORDER using both an analytical model and a simulation. We use our new analytical modelling technique (Chapter 3) in order to model ORDER. The results from our analytical model agree closely with the results from our simulation. Our analytical model of ORDER continues to predict performance accurately even in the high data contention scenario. We demonstrate that ORDER outperforms traditional concurrency control techniques like dynamic 2PL and BTO for a wide range of conditions. We study the performance effects of various parameters like message latency, ordering cost, transaction size, transaction composition, number of clients and processor speed. ORDER’s advantage disappears only when network latency is high and ordering is implemented inefficiently.

In section 6.1, we discuss different ways to implement total ordering. In section 6.2, we discuss previous work on concurrency control based on total ordering. In section 6.3, we describe ORDER and its implementation. In section 6.4, we present a qualitative comparison between ORDER and traditional methods like dynamic 2PL and BTO. In section 6.5, we present our analytical model of ORDER. In section 6.6, we evaluate the performance of ORDER under different system parameters and workloads. In section 6.7, we validate our analytical model against the simulation. We conclude in section 6.8.

6.1 Total Ordering Implementations

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. In one possible implementation, the logical time taken by a message to travel between two sites is equal to the logical distance (number of hops) between the two sites. 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. The system can achieve total ordering of messages if the destination SIUs reorder received messages according to pulse, pid and rank. 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 on an isotach network [Will93].

There are several other systems that implement the total ordering property. Isis allows multicasting to process groups with various ordering guarantees [BiJo86]. The Totem system does total ordering between clusters of workstations [MMAB96]. The Transis system also provides ordering protocols for process group systems [DoMa96]. Psync is a distributed computing system that provides ordered and atomic multicast protocols [MiPS91]. The Amoeba operating system supports a subsystem in which message delivery is atomic and totally ordered [Mull90, ReST89, KaTa91]. Vector clocks were used for ordering in the Harp replicated file system [LLSG92, Lisk91]. In the Highly Available System (HAS), protocols were proposed for achieving totally ordered delivery guarantees to groups [CASD85].

6.2 Concurrency Control Based on Total 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. An early version of Isis uses a similar algorithm in order to guarantee total ordering [BiJo87].

A concurrency control technique based on totally-ordered-multicast and group communication [BiJo86] has been proposed [ScRa96]. 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 authors refer to Totem [MMAB96] as a system that could provide an implementation for the property of total ordering of messages across groups. The proposed system does not address the issue of transactions that have dependencies among operations, or transactions that issue unilateral aborts. Finally, the paper does not address performance.

6.3 Ordering Network Aided Concurrency Control

The total ordering guarantee can be exploited by a distributed database to ensure that transactions are viewed in the same order at all destinations. The ORDER concurrency control technique is based on the total ordering property.

Given a fast and scalable implementation of the total ordering guarantee in the network, we can build an efficient concurrency control technique that uses this guarantee. 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). We define *latency penalty ratio* to be the ratio between the average message latency in the ordering network and the average message latency in a conventional network. By using a ratio, we eliminate any dependence of our results on the actual value of average message latency. As discussed in section 6.1, the isotach network is an efficient way of providing the total ordering guarantee. On a prototype isotach system optimized for large messages, the latency penalty ratio is 2.31 to 1.43 (for large messages) [LaMy00]. The second component of ordering cost — inherent ordering delay — is insignificant on the prototype, but could be significant if the variance in network latency is high. If the latency penalty ratio and inherent ordering delay are not too high, ORDER can potentially outperform a conventional concurrency control technique like dynamic 2PL.

The working of an ORDER system is presented below. The algorithm requires transactions to predeclare their accesses. The implications of this requirement are listed in section 6.4. A TM starts the execution of a transaction by issuing all the `reads` and `predeclare`s of the transaction marked as a single atomic action. The network delivers the operations comprising the atomic action in a total order with respect to other atomic actions, using some technique for implementing total ordering, such as the isotach technique described in section 6.1.

Each server buffers `read`s and `predeclare`s in queues corresponding to the data item 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, but 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 scheduler finds the corresponding `predeclare` on its queue and replaces the `predeclare` with a committed `write`. A committed `write` (`cwrite`) is a write that has been committed by the issuing transaction. If the committed `write` is now at the head of the queue, the scheduler forwards committed `write`s and `read`s from the head of the queue to the DM to be executed. If the committed `write` is not at the head of the queue, the scheduler uses the value of the committed `write` to send `read` responses to all the `read`s that are immediately behind the committed `write` on the queue, and then deletes the `read`s from the queue. The scheduler follows a similar procedure on receipt of an `abort`. The scheduler finds the corresponding `predeclare` on a queue and deletes the `predeclare`. If the deletion of the predeclare leaves committed `write`s and/or `read`s on the queue that are now ready to be executed, the scheduler executes the committed `write`s and `read`s. A transaction is complete when all of its operations have been either committed or aborted. The algorithms executed by various modules in an ORDER system are as follows.

```
TM Algorithm:
event: receive op from transaction;
  assert: op is a read/predeclare/write/commit/abort;
  if op == read/predeclare
    mark op with atomic action identifier;
    send op to network for ordered delivery;
  elsif op == write store op locally;
  elsif op == commit/abort
    look up stored writes for the transaction;
    generate a commit/abort operation for each write;
    send operations to network for unordered delivery;
event: receive op from network;
  assert: op is a read-response;
  forward op to transaction;
Scheduler Algorithm:
event: receive op for data item x from network;
  assert: op is a read/predeclare/commit/abort;
  assert: reads and predeclares arrive totally ordered;
  let queue = operation queue for data item x;
  if op == read
    if queue is empty send op to DM;
    elsif operation at tail(queue) is a cwrite
      send read-response with value from the cwrite;
    else append op to queue;
  elsif op == predeclare
    append op to queue;
  elsif op == abort
    find corresponding predeclare on queue;
    delete the predeclare;
    if predeclare was at head(queue)
      while operation at head(queue) == read/cwrite
        send operation to DM;
    elsif operation ahead of predeclare was a cwrite
      use the value of the cwrite to send read-responses to
```

```
          any reads now immediately behind the cwrite;
          delete the reads for which read-responses were sent;
    else if op == commit
        find corresponding predeclare on queue;
        replace predeclare with a cwrite;
        if predeclare was at head(queue)
            while operation at head(queue) == read/cwrite
                send operation to DM;
        else
            while operation behind newly-created cwrite is a read
                send read-response with value from cwrite;
                delete the read;
event: receive op from DM;
    assert: op is a read-response;
    send op to network for unordered delivery;
```

The network or communication subsystem provides two kinds of services — ordered delivery and unordered (conventional) delivery. As an example of how ORDER uses a specific total ordering implementation, consider how the isotach network implementation described in section 6.1 provides the ordered delivery service. On receipt of the last `read/predeclare` in a transaction (atomic action) from the TM, the SIU connected to the client assigns an isotach logical timestamp to the atomic action, where the timestamp consists of a pulse and a pid. The pulse is chosen to be equal to the maximum logical distance between the client and any of the servers that store any part of the writeset of the transaction. It is guaranteed that each operation in the atomic action will be received at its destination SIU before the receipt of the token ending the pulse. The pid is simply the identifier of the client that initiated the transaction. After having assigned a timestamp, the SIU marks every `read` and `predeclare` in the atomic action with the timestamp. An optional field (rank) may be added to an operation's timestamp. The rank field can be used if the application wants to order operations within the transaction, but is not needed if the network guarantees point-to-point FIFO (first-in-first-out) delivery. The token managers and SIUs in the isotach network exchange tokens and maintain isotach logical time, delivering operations in a total order by their timestamp. Recall that the total order is determined by the pulse, with ties being broken first by pid and then by rank.

6.4 Qualitative Comparison of ORDER to 2PL and BTO

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. An ORDER scheduler delivers a `read` to the DM as soon as the `read` reaches the head of the queue, and sometimes responds to a `read` early by using a committed `write` that is ahead of it. 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. In both ORDER and 2PL, `write` operations have the same effect of making the data item unavailable until the `write`s are committed. In 2PL, a `write` lock is held until the corresponding `commit` arrives. In ORDER, when a `write` reaches the head of the queue in the scheduler, it is executed by the DM only after the corresponding `commit` arrives. In 2PL, the servers send explicit acknowledgements to the client in response to `write`s. The client in ORDER needs no such notification because `write`s are never rejected.

2PL and ORDER also differ with respect to deadlocks. Deadlocks cannot occur in ORDER, because transaction operations are executed in a total order. Therefore, deadlock detection is unnecessary in ORDER, while servers in 2PL incur the cost of either preventing deadlocks, or of detecting and breaking them. In a distributed database system, deadlock detection involves both local and global deadlock detection, the latter involving communication among multiple servers. When a deadlock occurs in 2PL, a transaction must be aborted and restarted, a cost never incurred in ORDER. Previous studies have shown that deadlocks in 2PL are rare [Thom98a].

ORDER differs from BTO in several ways. In BTO, the servers have to send explicit acknowledgements to the client in response to `write`s. The client in ORDER needs no such notification. An ORDER system does not suffer from restarts, while conflicting operations arriving out of timestamp order can cause restarts in BTO. Previous studies have shown that restarts in BTO are significant [CaLi91].

ORDER incurs two costs that both 2PL and BTO do not — the latency penalty and the inherent ordering delay. Moreover, in ORDER, a transaction is forced to predeclare all of its accesses. 2PL and BTO do not require predeclaration, and access requests are made on demand. Predeclaration allows access requests to be made early, but may result in a longer data-holding time. In section 3.5, we showed that predeclaration had no significant effect on the performance of 2PL. In section 6.6.1.1, we show a scenario in which predeclaration imposes a penalty on the performance of BTO due to a longer data-holding time. The requirement of predeclaration also imposes a penalty on transactions for which the access set is not known in advance. In such situations, a transaction may be forced to declare its access set conservatively (bigger than it actually is) in order to ensure safe execution. In our performance studies, we assume that the access set for a transaction is known. Table 4 summarizes the differences among dynamic 2PL, BTO and ORDER.

In order to reveal the influence of the various trade-offs on overall performance, we studied the performance of 2PL, BTO and ORDER under different workloads and system parameters. The results are presented in the following two sections.

	2PL	BTO	ORDER
Predeclaration required?	No	No	Yes
Writes make data unavailable?	Yes	Yes	Yes
Reads make data unavailable?	Yes	No	No
Read lock releases required?	Yes	No	No
Write acks required?	Yes	Yes	No
Deadlocks	Yes	No	No
Restarts	Yes (break deadlocks)	Yes (out-of-order arrival)	No
Additional costs	Deadlock-detection	Timestamp management	Latency penalty, inherent ordering delay
Assumptions about network?	None	None	Total ordering algorithm

TABLE 4: Dynamic 2PL, BTO and ORDER

6.5 Analytical Model

We use the analytical modelling technique presented in Chapter 3 in order to model ORDER. Recall that we combine a data contention model and a hardware resource contention model, and use an iterative approach to calculate mean transaction response time. The assumptions and parameters for our analytical modelling technique were described in Chapter 3. In addition, for the ORDER model, we assume that a `read` is executed only after it reaches the head of the scheduler queue and is forwarded to the DM. We also ignore the inherent ordering delay in the analysis of ORDER. We do not make these simplifying assumptions in our simulation.

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, waiting to access a data item.

6.5.1 Data Contention

Recall that the mean data-holding time is T_H. In the ORDER algorithm, the data-holding time of a `read` is zero, because a `read` is immediately executed when it reaches the head of the scheduler queue. On the other hand, a `predeclare` has a data-holding time of R', because it holds the data item from the start of the transaction until the corresponding `commit` arrives. Therefore, the data item utilization in the ORDER system differs from the 2PL lock utilization.

ρ = (arrival rate of predeclares at a data item) / (service rate of predeclares at the data item)

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

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

The data contention is derived in the same way as in 3.3.1, except that the term *queue length at a data item* now refers to the number of `predeclare`s in the queue for that data item. R_{CONT} is still described by equation 2 (section 3.3.1).

6.5.2 Hardware Resource Contention

In the ORDER algorithm, the number of instructions to be executed at the client differs from that in 2PL for 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 releases 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` responses; $K*(1-P_R)*I_{TM}$ 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 releases. 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 scheduler overhead of processing `read`s and `predeclare`s; $K*P_R*(I_{NW}+I_{CC})$ accounts for the network and scheduler overhead of processing `read` responses; $K*(1-P_R)*(I_{NW}+I_{CC})$ accounts for the network and scheduler 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 3.3.2, except that the R_{CL2} component of R_{PHASE2} changes to include the local storage of `write`s and to exclude the sending of `read` lock releases:

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

The average network latency D_{NW} in an ORDER system will be higher than that 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.

6.5.3 Total Response Time

The total response time is calculated using an iterative procedure, exactly as described for 2PL in section 3.3.3, using equation 1′ instead of equation 1.

6.6 Performance Evaluation

We simulated the ORDER system and studied its performance. ORDER uses a special network that provides the service of ordered delivery for some messages. We model this network with a latency distribution, the average for which is obtained by multiplying the conventional average message latency by the latency penalty ratio. The communication subsystem delays the delivery of a message `m` until messages that are ordered before `m` have been received and delivered. In the ORDER simulation, a `read` can be satisfied by a committed `write` in front of it on the scheduler 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 the latency penalty ratio. This allows us to study the effect of different ordering costs on the performance of ORDER. Transactions are of variable size. The baseline parameters are the same as described in section 3.4. In both the analytical model and the simulation runs, we use a default latency penalty ratio of 2, which falls in the range of values obtained from the isotach prototype (section 6.3).

We ran the simulation for different arrival rates of transactions and for different database sizes, and measured the average transaction response time. We studied both the low data contention scenario and the high data contention scenario. We also studied the effect of various parameters on performance. The method of independent replications was used to obtain an accurate estimate of transaction response time. The values given in the graphs are the midpoints of a confidence interval which is less than 5% of the sample mean (on each side) at a 90% confidence level. One extra point at a wider confidence interval was plotted for each curve, in order to show the performance degradation after the knee. We first discuss the high data contention workload experiments.

6.6.1 High Data Contention Scenario

We first discuss performance under high message latencies. ORDER performs poorly under high latencies, but as we show later in this section, ORDER outperforms both 2PL and BTO under low message latencies. Figure 18 shows the effect of varying the arrival rate of transactions, at the default message latency of 1.5ms. As transaction arrival rate increases, response time increases for all the techniques, reaching a *knee* and then rising steeply as performance degrades and the system

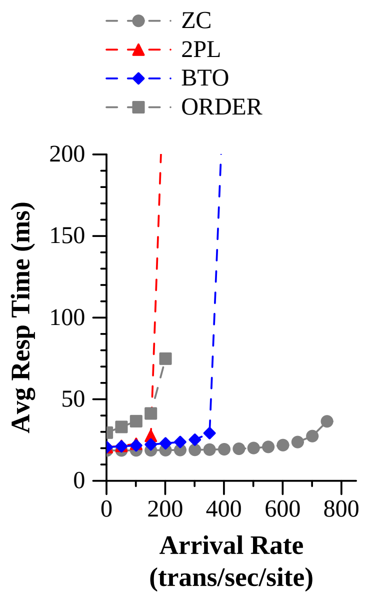


FIGURE 18: Transaction arrival rate (high data contention)

goes into an unstable region. As described in Chapter 4, the knee for ZC is entirely due to hardware resource contention, while the knees for the other techniques occur at lower arrival rates due to data contention.

Recall that the two important factors that affect performance of concurrency control techniques are queue lengths at the schedulers, and restart behaviour. Queues are longer in 2PL than in BTO, and the resulting iterative build-up of data contention degrades the performance of 2PL faster than that of BTO. Restarts are higher in BTO than in 2PL, and the resulting high resource contention degrades BTO's performance when the available resources are insufficient to sustain the load. ORDER involves no restarts, and therefore, its restart cost is zero. ORDER is similar to BTO in that `reads` do not lock data items. This should keep queue lengths in ORDER low. However, ORDER has an additional cost due to its latency penalty. At the default latency penalty ratio of two, messages in ORDER take twice as long to reach their destinations as in BTO and 2PL. A longer message traversal time causes longer data-holding times, in turn causing a longer response time. This is the cause for ORDER's high average response time. Moreover, the longer data-holding time causes longer queues and a higher probability of data conflict. This leads to an iterative build-up of data contention in ORDER. Figure 19 shows the average queue length encountered by a transaction operation at the scheduler, for ORDER, dynamic 2PL and BTO. The queue lengths in ORDER start rising sharply at the same load as that of 2PL, which is lower than the load at

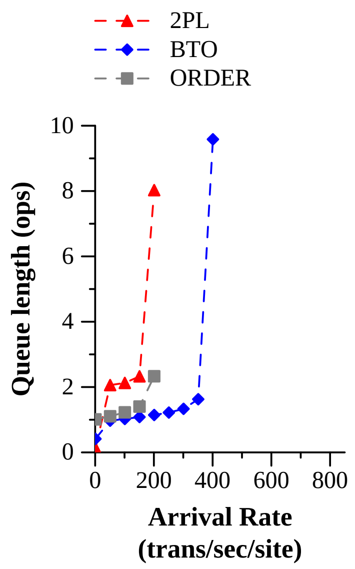


FIGURE 19: Queue length (high data contention)

which BTO peaks. Therefore, the knee of ORDER's response time curve occurs at the same load as that of 2PL's. For the default message latency used in this experiment, the performance of ORDER is worse than both BTO and 2PL in terms of average response time. The following section explores different ordering costs and message latencies, and shows that the performance of ORDER improves at lower ordering costs and lower message latencies.

6.6.1.1 Ordering Cost and Message Latency

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 ratios — 1, 2, 3 and 4 — on ORDER. Recall that the latency penalty ratio of the isotach prototype is 1.43-2.31. The higher the latency penalty ratio, the longer operations take to arrive, the longer the data-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 data-holding times. Figure 20 shows the effect of different ordering costs, for the default message latency. The ORDER technique is represented by four different curves, each representing a different latency penalty ratio. The *order-2* curve

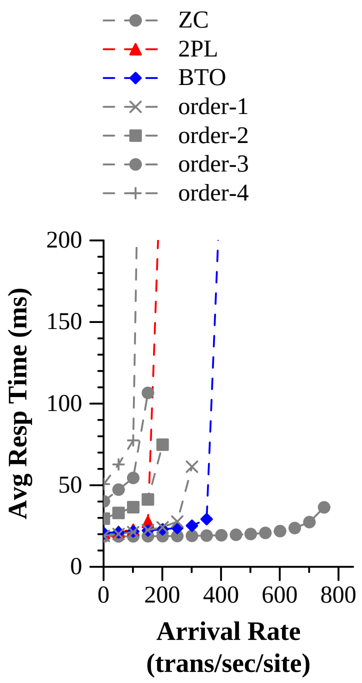


FIGURE 20: Ordering costs
Message latency = 1.5ms
(high data contention)

uses a latency penalty ratio of 2, and corresponds to the *ORDER* curves presented in earlier graphs.

The lower latency penalty ratio of order-1 leads to its performing better than order-2, but its performance is still worse than that of BTO. A latency penalty ratio of 1 corresponds to an ordering network with no penalty on message latency. BTO performs better than order-1 because transactions in BTO do not predeclare their accesses, and access data items only when needed. This keeps queue lengths in BTO lower than those in order-1. Figure 21 shows the effect of predeclaration on the performance of BTO. For this experiment, we ran the BTO simulation without predeclaration (normal case) and then with predeclaration, and compared the resulting transaction response times. The graph shows that predeclaration imposes a penalty on BTO, bringing BTO's performance knee down to that of order-1. This result was initially surprising because we had expected BTO-with-predeclaration to do worse than order-1, due to BTO's restart cost. The two reasons for this surprising result are that predeclaration reduces the percentage of restarts in BTO, and that performance in this high data contention with high message latency case is dominated by data contention. When transactions in BTO predeclare their accesses, the probability of operations arriving out of timestamp order is lower, reducing the amount of restart behaviour. Moreover, the amount of restart behaviour in BTO is not high enough to significantly degrade performance at the load where performance degrades due to data contention.

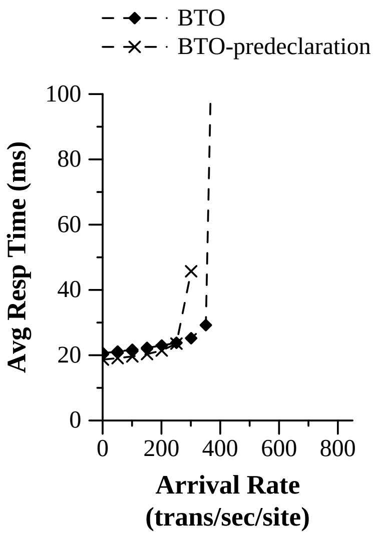
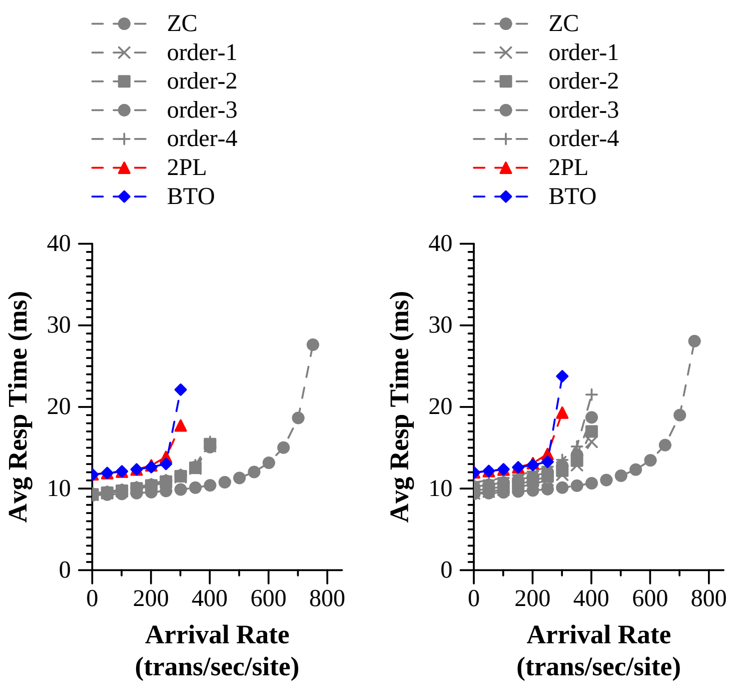


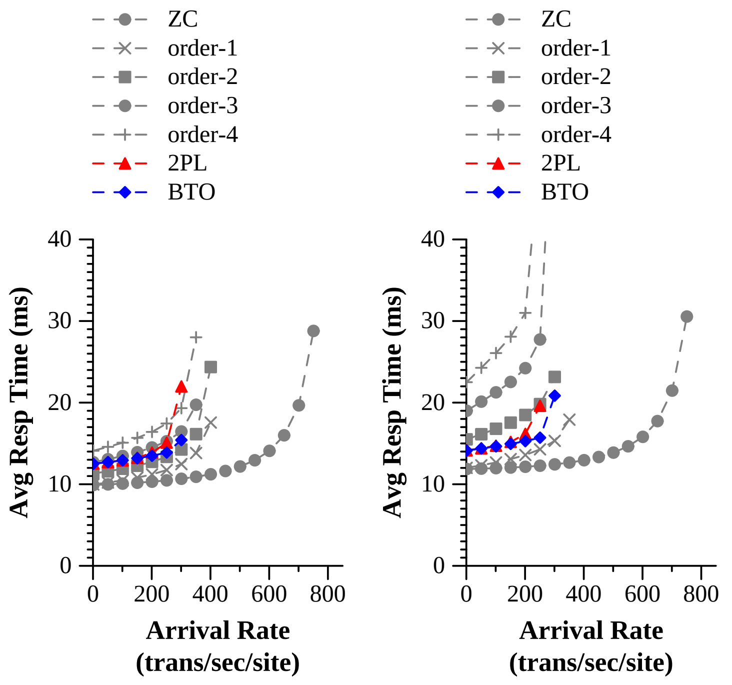
FIGURE 21: Predeclaration
Message latency = 1.5ms
(high data contention)

Figure 22 shows the effect of message latencies in the microsecond range. At low latencies of 20µs and 80µs, all of the ORDER variants perform better than both 2PL and BTO. ORDER's sustainable transaction load is of an order of 40% better than that of 2PL and BTO. The effect of varying the latency penalty ratio is more apparent in the graphs for higher network latencies.

At a latency of 500µs, order-1, order-2 and order-3 remain stable up to a higher load than 2PL does, but the performance of order-4 degrades at the same load as that of 2PL. Comparing ORDER to BTO, order-1 remains stable up to a higher load than BTO at a latency of 500µs. However, the performance of order-2 and order-3 degrades at the same load as that



(a) Avg. latency = 20µs **(b)** Avg. latency = 80µs



(c) Avg. latency = 200µs **(d)** Avg. latency = 500µs

FIGURE 22: Message latency (microseconds)
(high data contention)

of BTO, and the performance of order-4 degrades earlier than BTO. Moreover, at a latency of 500µs, the average response times of order-3 and order-4 are significantly higher than those of BTO and 2PL in the stable region.

At low message latencies, the latency penalty for ORDER is lower, and this allows ORDER to perform better than 2PL and BTO. Queue lengths in ORDER stay lower than in 2PL. In addition, the absence of restarts in ORDER gives ORDER an advantage over BTO. This effect can be seen in Figure 23, which shows the average queue lengths and restart behaviour of ORDER, 2PL and BTO for the lowest message latency (20µs) case.

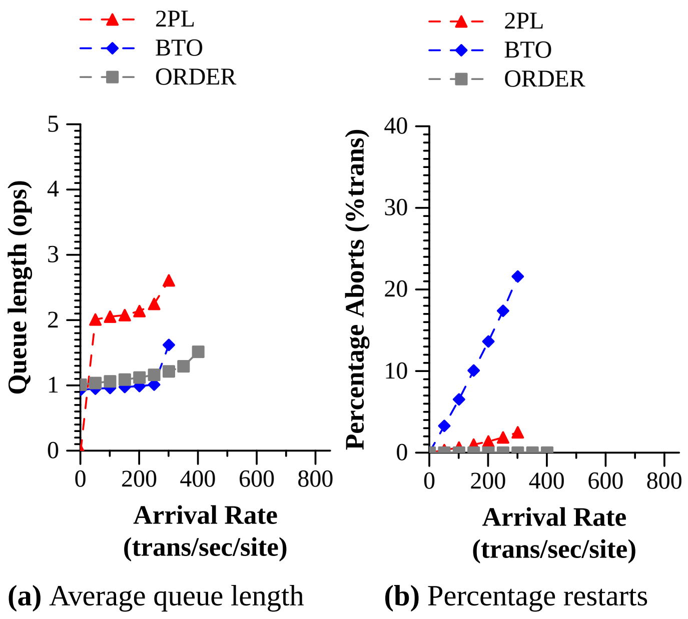


FIGURE 23: Queue lengths and restarts
(high data contention)

In summary, ORDER outperforms 2PL and BTO at low message latencies. When message latencies are high, the network must provide the total ordering guarantee at a low latency penalty ratio, in order for ORDER to outperform 2PL and BTO. ORDER is an attractive technique for high message latencies only when the latency penalty ratio is very low. However, ORDER becomes a good choice for concurrency control when message latencies are low. In the following experiments, we use the default low message latency of 20µs, and explore the effects of other parameters on performance.

6.6.1.2 Processor Speed

Hardware resource contention is one of the factors that affect performance. As processor speeds increase, resource contention decreases for all three techniques. At high processors speeds, hardware resource contention becomes low enough so that performance differences among the different techniques are determined primarily by data contention. Figure 24 shows the effect of different client processor speeds on the performance of ORDER, 2PL and BTO. A reduction in processor speed from 400 MIPS to 200 MIPS degrades the performance of ORDER, 2PL and BTO, because the increased hardware resource contention increases transaction response time. The hardware contention in BTO is more than that in ORDER and 2PL, because BTO suffers from a high percentage of restarts. This effect dominates performance, and BTO performs poorly at low processor speeds. In addition, the increased response time caused by hardware resource contention contributed to an iterative build-up of data contention. Since queue lengths are higher in 2PL than in ORDER, this data contention effect causes 2PL to perform worse than ORDER when processor speed goes down.

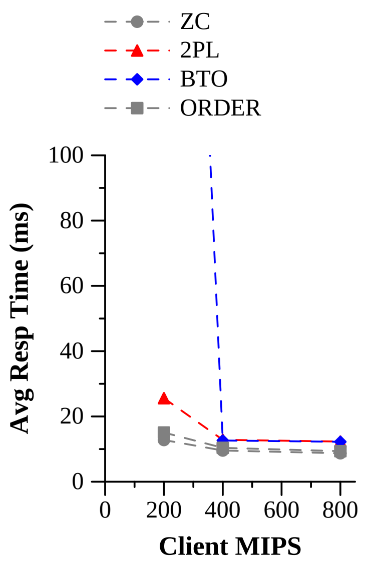


FIGURE 24: Processor speed (high data contention)

6.6.1.3 Number of Clients

Figure 25 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. This effect of an increased number of active transactions can also be obtained by increasing the transaction arrival rate at each client.

However, increasing the number of clients allows us to increase the amount of data conflict without increasing hardware resource contention at the clients.

One effect of increased data conflict is that queue lengths at the schedulers increase, causing an iterative build-up of data contention. Queues in ORDER and BTO are shorter than those in 2PL because `reads` do not lock data items in these two techniques. This allows both ORDER and BTO to outperform 2PL. Predeclaration causes queue lengths in ORDER to be higher than those in BTO, and therefore, ORDER is affected more by the increased load of clients. The increased load does cause an increase in BTO's restart behaviour, increasing resource contention. However, the data contention effect is dominant in this experiment.

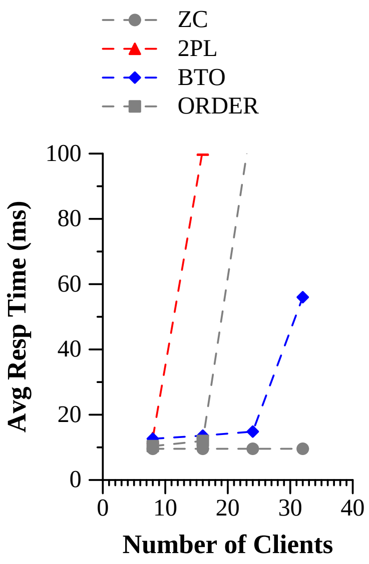


FIGURE 25: Number of clients (high data contention)

6.6.1.4 Transaction Size

Larger transactions are becoming important in today's workloads, making it important to explore transactions with long lifetimes. [Fran99]. Figure 26 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-holding time and, therefore, data conflict. The increased data conflict increases the probability of

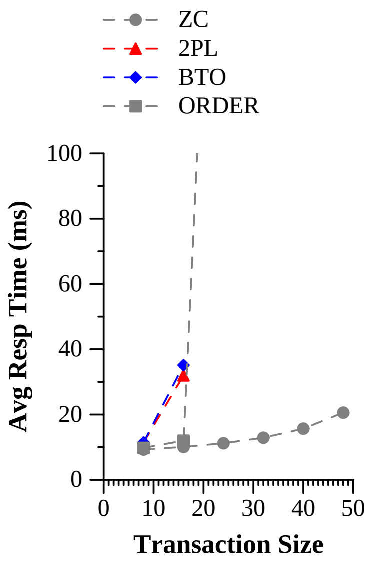


FIGURE 26: Transaction size (high data contention)

deadlock in 2PL, and the percentage of restarts in BTO. As before, the lower queue lengths in ORDER and BTO allow them to outperform 2PL. Moreover, the absence of aborts in ORDER allows it to outperform BTO.

6.6.1.5 Transaction Composition

Figure 27 shows the effect of varying the read probability while transaction size is held constant. When the probability of `read` is 100%, there are no `write`s, and therefore the disk access time of the second phase goes away for all the techniques. Therefore, the response time is low when the probability of read is 100%. As the probability of read decreases, there are fewer `read`s and more `write`s in the system, increasing data conflict. This increase in data conflict is manifested as an increased queue length at the schedulers. Since queue lengths are longer in 2PL than in ORDER and BTO, 2PL is affected more by the increased data conflict.

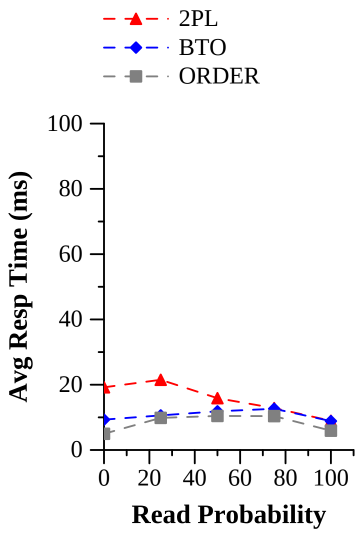


FIGURE 27: Read probability (high data contention)

At the lowest read probability of zero, all transactions are 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 and BTO offer no such advantage because the client must wait for acknowledgements of the `write`s, before committing. Another advantage of write-only transactions (for all the techniques) is that they perform no computation. The response time for ORDER is the lowest of the three techniques when the read probability is zero.

6.6.2 Low Data Contention Scenario

We now analyze results from the low data contention workload experiments. The default low message latency of 20µs is used in the following experiments. Figure 28 shows the effect of varying the arrival rate. ORDER continues to outperform both 2PL and BTO. The queue lengths in ORDER and BTO stay lower than in 2PL because `reads` do not lock objects. In addition, the absence of aborts in ORDER lets it perform better than BTO.

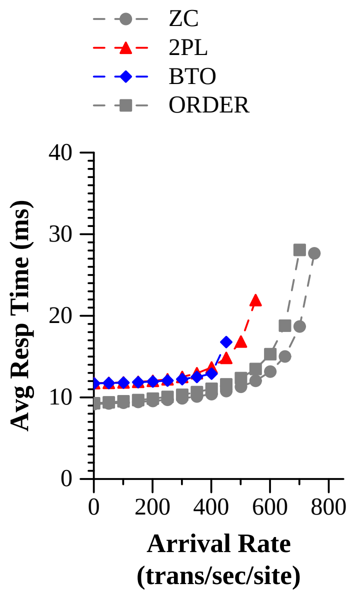
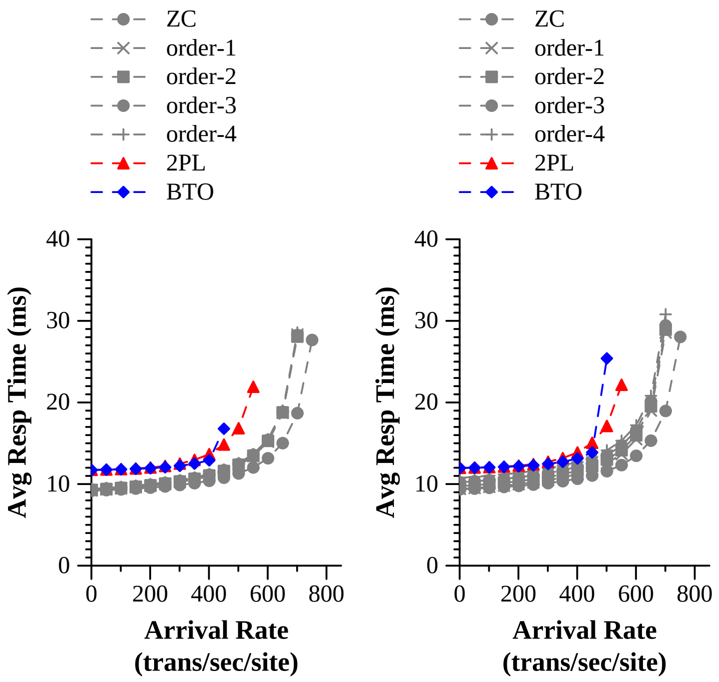


FIGURE 28: Transaction arrival rate (low data contention)

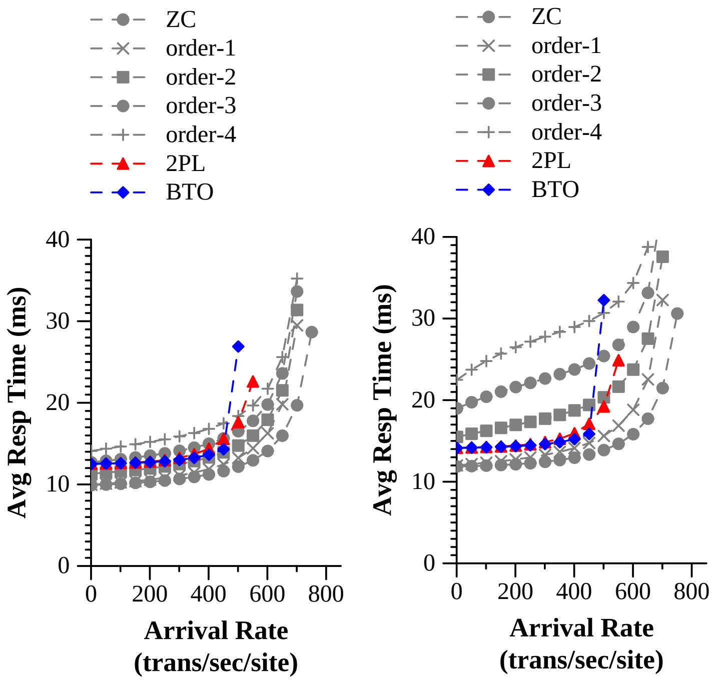
6.6.2.1 Message Latency

We explored the effect of message latency on performance in this low data contention scenario. Figure 29 shows the results of the experiments for message latencies in the microsecond range. All of the order variants remain stable up to heavier loads than 2PL and BTO at all message latencies.



(a) Avg. latency = 20µs **(b)** Avg. latency = 80µs
FIGURE 29: Message latency (microseconds) (low data contention)

At the largest message latency of 500μs, the average response time of order-3 and order-4 are significantly higher than those of 2PL and BTO in the stable region. ORDER performs well when message latency is low. If message latency is high, ordering must be implemented very efficiently (low latency penalty ratio) for ORDER to outperform 2PL and BTO in terms of average response time as well as performance knee.



(c) Avg. latency = 200μs **(d)** Avg. latency = 500μs

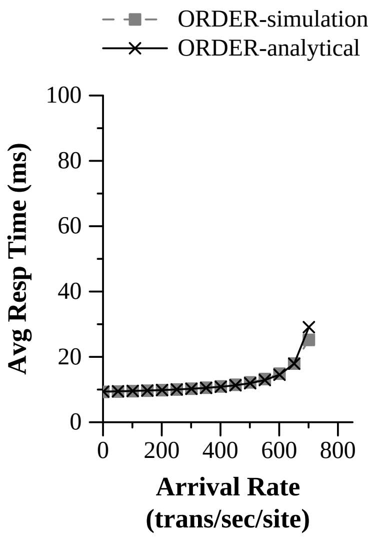
FIGURE 29: Message latency (microseconds) (low data contention)

6.6.2.2 Other Parameters

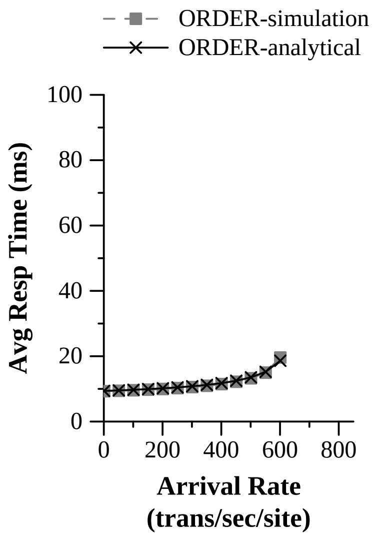
We studied the effect of other parameters like processor speed, number of clients, transaction size and transaction composition. The results were similar to the ones we presented in the high data contention scenario (section 6.6.1).

6.7 Validation

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 30 shows the results for two database sizes: 32000 and 4000 data items. Recall that the database size = 32000 case models low data contention, and the database size = 4000 case models high data contention. In this experiment, no hot spots



(a) Database size = 32000
(low data contention)



(b) Database size = 4000
(high data contention)

FIGURE 30: Analytical versus simulation results for ORDER

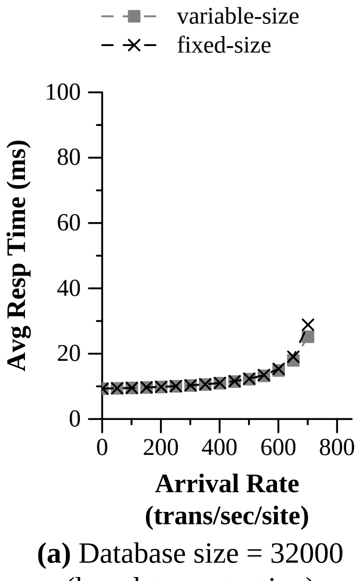
were used in the simulation, in order to simulate the same scenario as the analytical model.

As the transaction arrival rate increases, response time increases slowly until a knee, after which response time rises steeply, and the system becomes unstable. As more and more transactions are introduced into the system, the data conflict probability increases, causing longer queue wait times at the schedulers and longer response times. 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. The remarkably close agreement between the analytical and simulation results despite several differences in assumptions led us to perform further experiments to explore the effect of these assumptions. The next section shows that the assumptions do not significantly affect our model's accuracy in predicting the performance of ORDER.

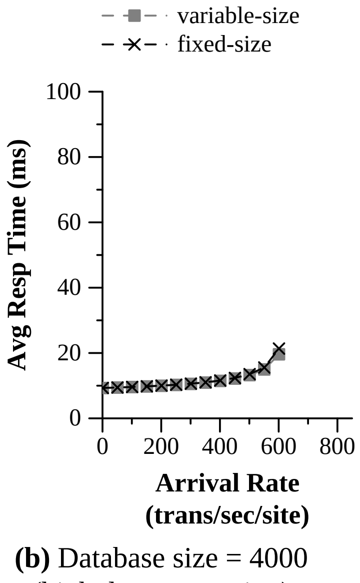
6.7.1 Effect of Assumptions

The analytical model makes three simplifying assumptions, which the simulation does not — neglecting the inherent ordering delay, assuming fixed-size instead of variable-size transactions, and ignoring early execution of `reads`. Recall 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. Our experiments show that the average inherent ordering delay in our simulation contributes only about 0.1% of the total transaction response time.Therefore, neglecting the inherent ordering delay in our analytical model does not significantly affect the agreement between the analytical model and the simulation.

The analytical model assumes fixed-size transactions, while the simulation uses



(a) Database size = 32000
(low data contention)

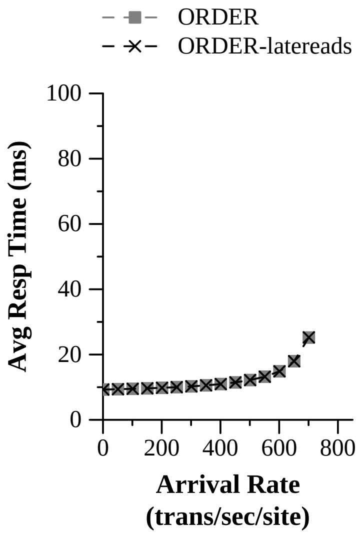


(b) Database size = 4000
(high data contention)

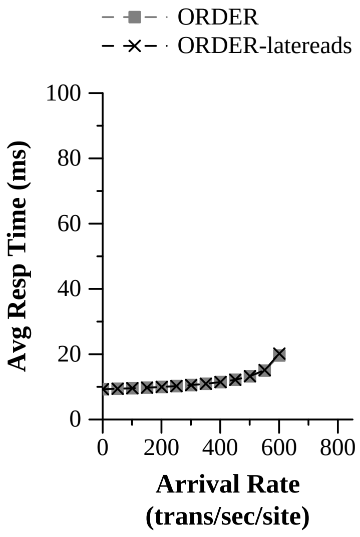
FIGURE 31: Fixed versus variable size transactions in ORDER

variable-size transactions. We ran the simulation with fixed-size transactions (of size 16), and then with variable-size transactions (of size 8-24, uniformly distributed). Figure 31 shows the results from this test. The assumption of fixed-size transactions does not significantly affect our model's accuracy in predicting the performance of ORDER.

The last difference between the analytical model and the simulation is that in the simulation, a `read` can be satisfied by a committed `write` in front of it on the scheduler queue, instead of having to wait to get to the head of the queue. We ran the simulation with this early execution of `read`s (normal case), and then without it (*late reads*). Figure 32 shows the results from this test. The assumption of late `read`s also does not significantly affect our model's accuracy in predicting the performance of ORDER.



(a) Database size = 32000 (low data contention)



(b) Database size = 4000 (high data contention)

FIGURE 32: Late reads in ORDER

6.8 Conclusion

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 concurrency control. We have also presented an analytical model for ORDER, and validated it against a simulation. Our analytical model continues to predict performance accurately even at high data contention levels, owing to the more accurate modelling of queue lengths we perform. The ability to model ORDER accurately is a significant success of our analytical modelling approach.

ORDER outperforms both dynamic 2PL and BTO when network latency is low. ORDER's advantage disappears only when network latency is high and ordering is implemented inefficiently. The performance of the isotach prototype implies that ordering can be implemented efficiently. ORDER is a good candidate for high as well as low data contention workloads, as long as message latencies are low. At high latencies, the latency penalty ratio of ORDER must be very low (less than 2) in order for it to perform well.

Conditions	Relative Performance
High data contention workload, low message latencies	ORDER > BTO ≥ 2PL
High data contention workload, high message latencies	BTO > 2PL > ORDER*
Low data contention workload, low message latencies	ORDER > 2PL > BTO
Low data contention workload, high message latencies	BTO > 2PL > ORDER*

TABLE 5: Performance comparison of 2PL, BTO and ORDER

* ORDER can perform well at high latencies if the latency penalty ratio is low.

In all the techniques studied, 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. In BTO, aborts can increase to a point where they significantly increase hardware resource contention and degrade performance. ORDER retains most of BTO's advantage of low queue lengths. In addition, the absence of aborts in ORDER can allow ORDER to outperform BTO for a wide range of conditions, as long as the latency penalty ratio is not too high. Table 5 summarizes the performance differences among the different techniques.

Chapter 7

Predictability

Predictability is the ability of the sender of a message to predict the time at which the message will have arrived at its destinations, where *time* is either real-time or logical time in a time frame valid for the system. This property can be exploited by a distributed database in order to schedule transactions in a way that ensures serializable execution and a low number of restarts. A network that provides predictable message delivery can be used as the basis for efficient concurrency control of transactions. We present a new concurrency control technique — PREDICT — that uses the interconnection network in a distributed database as an aid to concurrency control [SrWR01b]. A predictable network can give the concurrency control scheduler an estimate of the amount of risk involved in processing a transaction at a certain time. The ability to estimate risk allows a scheduler to make intelligent decisions in order to keep concurrency costs low.

We analyze the performance of PREDICT using a simulation, and demonstrate that PREDICT outperforms traditional concurrency control techniques like dynamic 2PL and BTO for a wide range of conditions. We study the performance effects of various parameters like message latency and variance, degree of aggressiveness, processor speed, transaction size, transaction composition and number of clients. PREDICT outperforms dynamic 2PL under all conditions. PREDICT also outperforms BTO under low data contention. Under high data contention, the relative performance of BTO as compared to PREDICT varies, depending on the message latency. We present a set of variants of the PREDICT technique based on reasonable assumptions about network characteristics in a distributed database system. We demonstrate that PREDICT achieves a good balance between lost opportunity cost and restart cost.

In section 7.1, we discuss predictability and how to implement it. In section 7.2, we describe predictability-aided concurrency control. In section 7.3, we present a qualitative comparison between PREDICT and the traditional techniques of dynamic 2PL and BTO. In section 7.4, we evaluate the performance of PREDICT under different system parameters and workloads. We conclude in section 7.5.

7.1 Predictable Message Delivery

While perfect predictability in real-time may be difficult to achieve, most networks can guarantee imperfect predictability. Studies of network transmission delays have shown that a majority (80-95%) of messages are delivered within a bound of about 1.5-2.5 times the average message latency, while the remaining messages can take unpredictably large amounts of time to reach their destinations [Cris89a, WaKe99]. We show that a distributed database can make use of even this imperfect predictability to achieve a low restart cost.

Other systems have made the assumption of predictable message delivery for other purposes. An optimistic concurrency control technique that prunes validation information based on loosely synchronized clocks has been proposed [AGLM95]. The technique uses backward validation, and assumes that some synchronization mechanism like Network Time Protocol [Mill88] guarantees bounded clock skews and message transmission delays. Some work on atomic broadcast uses the assumption of bounded transmission delays [Cris89b]. As we will show in the following section, the predictable delivery guarantee is required only for a subset of all messages, which implies that quality-of-service techniques can be used in order to provide predictable delivery for this limited class of messages.

Predictability can also be implemented in logical time. A network that provides predictability at a low cost is the *isotach network* [ReWW97, Will93] that was described in Chapter 6. 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.

7.2 Predictability-Aided Concurrency Control

With predictable message delivery, when a client issues the operations of a transaction, it can predict the time `t` by which the operations will be received by all the destinations. With this knowledge, a conservative site can schedule the transaction to be executed at a time that is equal to or later than `t`. The transaction can be executed at the scheduled time at all destination servers, because the network guarantees that the operations of the transaction will be received before that scheduled time. A server can break ties between transactions scheduled for the same time by using a predetermined ordering of clients. Thus transactions are executed in the same order at all servers, guaranteeing serializability.

An efficient implementation of the predictability property in the network is a useful building block in the construction of efficient concurrency control techniques for transactions.

We propose a new timestamp ordering concurrency control technique called PREDICT, that provides a mechanism to control the amount of restart behaviour. The PREDICT technique works as follows. For clarity of exposition, we first explain the algorithm assuming that the network provides perfect predictability, and then describe why the algorithm works even under imperfect predictability. The algorithm requires transactions to predeclare their accesses. 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 assigns a timestamp `t` to this atomic action, where `t` is the time by which all the operations will have reached their destinations. Upon receipt of an operation `o`, the destination server has two options in processing `o`:

1. The conservative option is to wait until time `t` to process `o`. If all servers process a transaction's operations at the same time `t`, then a total order of transactions is achieved.
2. The aggressive option is to process `o` before time `t`. The server then rejects conflicting operations that arrive out of timestamp order. Rejected operations cause transaction restarts. The amount of restart behaviour can be estimated and controlled by the difference between `t` and the processing time. The control over restart behaviour might be adaptive, based on past behaviour of the system.

The *aggressive limit* is the maximum time interval a server will allow between the timestamp and the processing time of any operation. In other words, when the server receives an operation with timestamp `t` at time t_r, the server processes the operation immediately if ($t-t_r$) is less than or equal to the aggressive limit. Otherwise, the server waits until the time (`t`-aggressive limit) and then processes the operation. The aggressive limit can vary from zero to no_limit. If the aggressive limit is zero, then the server always processes an operation at its timestamp, and the operations in each transaction are being processed in a total order. PREDICT then reduces to the ORDER concurrency control technique (Chapter 6). If the aggressive limit is set to no_limit, the server always processes operations immediately upon receipt. The more the aggressive limit, the higher the probability of abort. In addition, if the variance in network latency is high, then there is a high probability of messages arriving out of order, and therefore a high probability of abort. If the number of aborts is sufficiently low, PREDICT can outperform a conventional concurrency control technique like 2PL.

At a server, the scheduler can either accept or reject an operation. The scheduler buffers accepted `read`s and `predeclare`s in queues corresponding to the data item accessed, and maintains two values — maxRTS and maxWTS — for every data item that is stored at that server. These values are the maximum timestamps of the `read` and `predeclare` operations that the server has accepted. When the scheduler processes a `read` with timestamp `t`, the scheduler rejects the `read` if `t` < maxWTS for that data item. When the scheduler processes a `predeclare` with timestamp `t`, the scheduler rejects the `predeclare` if `t` < maxRTS or if `t` < maxWTS for that data item. If the scheduler accepts an operation, it appends the operation to the tail of the appropriate queue, sending an explicit acknowledgement to the client if the operation is a `predeclare`. (Read responses double as acknowledgements for the `read`s.) The scheduler immediately executes a `read` by sending it to the DM, if the `read` is at the head of its 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 `predeclare`s and has issued all of its `write`s, the transaction issues `commit`s. On the other hand, if the client receives a reject, or if it decides to perform a unilateral abort, the client sends out `abort`s for every `predeclare` that it issued. The TM sends `commit`s (carrying the values of the corresponding `write`s) and `abort`s as regular messages rather than timestamped messages. The network does not have to guarantee predictable message delivery for these `commit`s or `abort`s. On receipt of a `commit`, the destination scheduler finds the corresponding `predeclare` on its queue and replaces the `predeclare` with a committed `write`. A committed `write` is a write that has been committed by the issuing transaction. If the committed `write` is now at the head of the queue, the scheduler forwards committed `write`s and `read`s from the head of the queue to the DM to be executed. If the committed `write` is not at the head of the queue, the scheduler uses the value of the committed `write` to send `read` responses to all the `read`s that are immediately behind the committed `write` on the queue, and then deletes the `read`s from the queue. The scheduler follows a similar procedure on receipt of an `abort`. The scheduler finds the corresponding `predeclare` on a queue and deletes the `predeclare`. If the deletion of the predeclare leaves committed `write`s and/or `read`s on the queue that are now ready to be executed, the scheduler executes the committed `write`s and `read`s. A transaction is complete when all of its operations have been either committed or aborted. Since operations are executed in timestamp order, and timestamps are drawn from a totally ordered domain, serializability is guaranteed.

As noted before, the network may provide imperfect predictability, in that operations may not always arrive before their timestamp. The above algorithm guarantees correct execution even with imperfect predictability. The estimated worst-case latency, possibly combined with maximum clock skews, are used in order to assign a timestamp to a transaction. An operation that arrives late simply has a higher probability of rejection. However, imperfect predictability does affect performance. The better the network at predicting message delivery time, the lower the restart cost. The predictable delivery guarantee is required only for `read`s and `predeclare`s, and not for other messages. This implies that quality-of-service techniques can be used in order to provide predictable delivery for this limited class of messages.

7.3 Qualitative Comparison

PREDICT differs from 2PL in all the ways in which BTO differs from 2PL (section 4.2). In addition, PREDICT differs from 2PL and BTO in the following ways. For good performance, PREDICT assumes that the network provides some measure of predictability in message delivery, while 2PL and BTO make no such assumption. Moreover, a transaction in PREDICT is forced to predeclare all of its accesses. 2PL and BTO do not require predeclaration, and access requests are made on demand. Predeclaration allows access requests to be made early, but may result in a longer data-holding time. An important difference between PREDICT and BTO is that a server in PREDICT can delay the processing of an operation in order to decrease the probability of abort. Finally, a scheduler in PREDICT immediately rejects operations that were delivered

	2PL	BTO	ORDER	PREDICT
Predeclaration required?	No	No	Yes	Yes
Writes make data unavailable?	Yes	Yes	Yes	Yes
Reads make data unavailable?	Yes	No	No	No
Read lock releases required?	Yes	No	No	No
Write acks required?	Yes	Yes	No	Yes
Deadlocks	Yes	No	No	No
Restarts	Yes (break deadlocks)	Yes (out-of-order arrival)	No	Yes (out-of-order arrival)
Additional costs	Deadlock-detection	Timestamp management	Latency penalty, inherent ordering delay	Delayed processing cost, timestamp management
Assumptions about network?	None	None	Total ordering algorithm	Predictability (for good performance)
Ability to tune performance	No	No	No	Yes

TABLE 6: Dynamic 2PL, BTO, ORDER and PREDICT

out of order. A BTO scheduler, on the other hand, accepts operations that were delivered out of order but that can still be executed in timestamp order. A variant of PREDICT is possible in which the scheduler demonstrates the above *late reject* behaviour similar to BTO, and we study the performance of this variant in section 7.4.1.1.* Note that rejecting operations increases restart cost, but may improve data item availability by keeping

* The late-reject variant of PREDICT with the aggressive_limit set to no_limit is equivalent to a variant of BTO that predeclares all accesses.

queues short. Wait-depth locking (WDL) is a variant of 2PL that restricts queue lengths at the expense of restarts [FrRo93]. A comparative study of WDL with timestamp ordering methods is not explored here. As mentioned in section 7.2, PREDICT reduces to the ORDER technique when the aggressive limit is zero, and when perfect predictability is implemented. By allowing imperfect predictability, PREDICT incurs a restart cost, which ORDER does not. On the other hand, PREDICT can be implemented on a network providing real-time predictability based on bounded transmission delays. Such an implementation of PREDICT has an advantage over ORDER, in that it eliminates the need for a special ordering algorithm in the network, and the cost of the ordering algorithm. Table 6 summarizes the differences among dynamic 2PL, BTO, ORDER and PREDICT.

7.4 Performance Evaluation

We simulated the PREDICT system and studied its performance. Our analytical modelling technique (Chapter 3) cannot be directly applied to PREDICT, because the model does not account for transaction aborts and restarts. Extending the model to handle restarts involves a careful analysis of the interaction between hardware resource contention caused by aborts and data contention, and has not been explored here. The baseline parameters are the same as described in section 3.4. PREDICT uses a network that provides imperfect predictability, in that the clients know the 90th percentile of the network's delay distribution and use it as the estimated worst-case message latency. In other words, 90% of all messages arrive within the estimated worst-case time, and 10% arrive later. We model message latency as an exponential distribution; therefore, the estimated worst-case message latency (90th percentile of the distribution) is approximately

2.303 times the average latency. We model the servers' degree of aggressiveness by the parameter *aggressiveness*, which is measured as a percentage of the worst-case message latency. Recall that the aggressive limit is the maximum time interval a server will allow between the timestamp and the processing time of any operation. The aggressive limit is calculated as (aggressiveness * estimated worst-case message latency). Therefore, the aggressive limit can vary from zero to the estimated worst-case message latency. The default aggressiveness is 50%.

We ran the simulation for different arrival rates of transactions and for different database sizes, and measured the average transaction response time. We studied both the low data contention scenario and the high data contention scenario. We also studied the effect of various parameters on performance. The method of independent replications was used to obtain an accurate estimate of transaction response time. The values given in the graphs are the midpoints of a confidence interval which is less than 5% of the sample mean (on each side) at a 90% confidence level. One extra point at a wider confidence interval was plotted per curve in order to show the performance degradation after the knee.

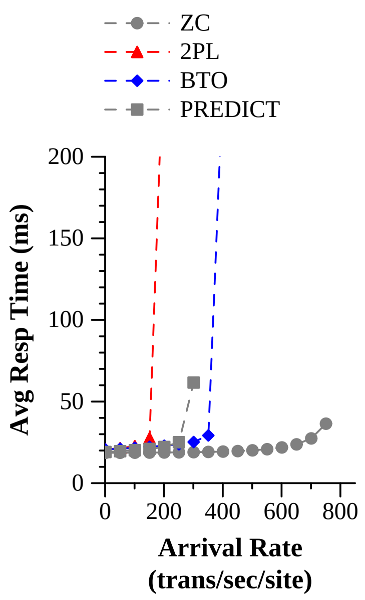


FIGURE 33: Transaction arrival rate (high data contention)

7.4.1 High Data Contention Scenario

Figure 33 shows the effect of varying the arrival rate of transactions, on average transaction response time in ZC, PREDICT, 2PL and BTO. As transaction arrival rate increases, response time increases for all four techniques, reaching a knee and then rising steeply as performance degrades and the system goes into an unstable region. The sustainable transaction load for PREDICT is of an order of 60-70% better than that of 2PL. However, BTO achieves a sustainable transaction load that is about 40% higher than that of ORDER.

Recall that the two important factors that affect performance of concurrency control techniques are queue lengths at the schedulers, and restart behaviour. Figure 34a shows the average queue length encountered by a transaction operation at the scheduler. Since `reads` do not lock objects in PREDICT, queues are longer in 2PL than in PREDICT. The resulting iterative build-up of data contention degrades the performance of 2PL faster than that of PREDICT. Figure 34b shows the percentage of restarts for PREDICT, 2PL and BTO. The number of restarts is higher in PREDICT than in 2PL, but is not high enough to cause significant deterioration in the performance of PREDICT.

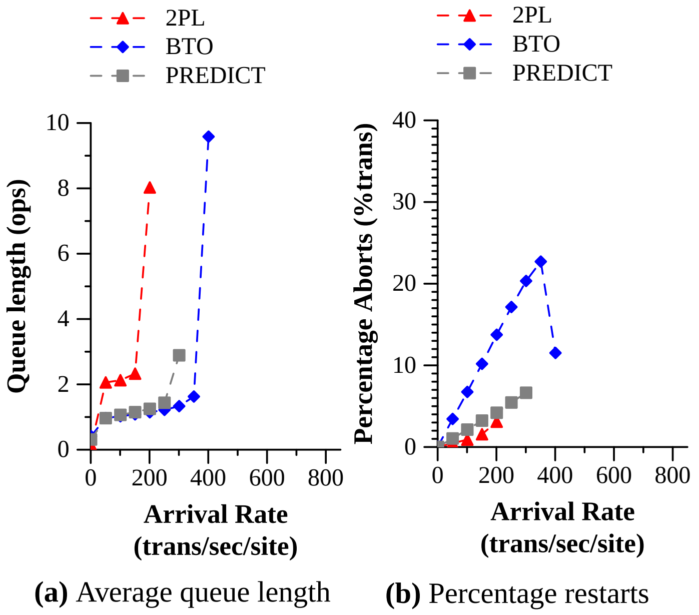


FIGURE 34: Queue lengths and restarts
(high data contention)

Since PREDICT predeclares all of its accesses, the probability of operations arriving out of timestamp order is lower in PREDICT than in BTO. Moreover, since the PREDICT algorithm uses an aggressiveness of 50%, it delays the processing of operations, thereby reducing the probability of the scheduler seeing operations out of order. Thus, PREDICT uses predeclaration of accesses and delayed processing of operations in order to keep the percentage of aborts lower than that in BTO (Figure 34b). However, predeclaration causes a longer data-holding time, because access requests are made early. Moreover, delayed processing of operations increases the lost opportunity cost, which increases transaction response time, which can increase data-holding time. In this high data contention scenario where queue lengths already tend to be high because of the small database size as well as high message latency, the increased data-holding time of PREDICT results in longer queues than those in BTO (Figure 34a). This effect dominates the performance difference between BTO and PREDICT, and allows BTO to outperform PREDICT in this scenario.

Recall that another difference between PREDICT and BTO is that PREDICT rejects operations early, while BTO follows the late reject policy. Figure 35 shows the effect of late rejects on the performance of PREDICT. For this experiment, we ran the PREDICT simulation with immediate rejects (normal case) and then with late rejects, and compared the resulting transaction response times. There is no significant difference in the performance of PREDICT under the two policies for the given scenario.

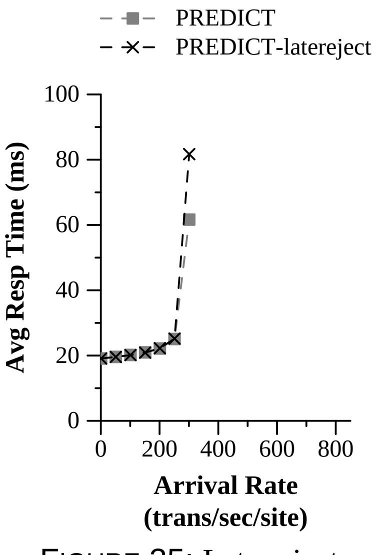


FIGURE 35: Late rejects
Message latency = 1.5ms
(high data contention)

7.4.1.1 Message Latency

We explored the effect of different message latencies in the millisecond and microsecond range. Figure 36 shows the effect of different latencies in the millisecond range. At an average message latency of 1.5ms, PREDICT performs better than 2PL.

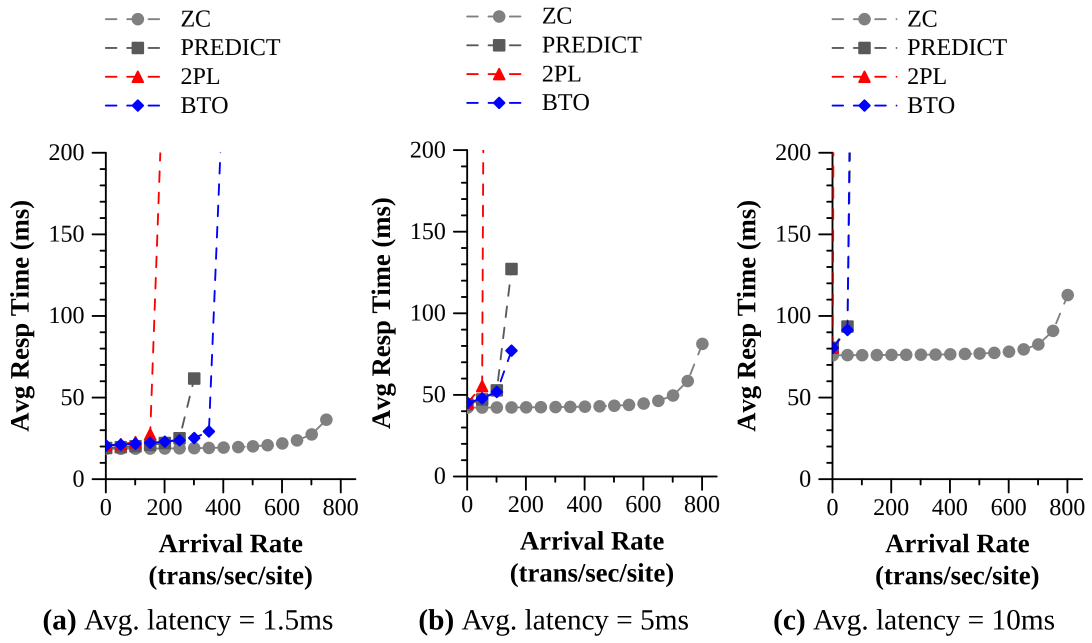


FIGURE 36: Message latency (milliseconds)
(high data contention)

Increasing the message latency increases data contention, and causes the knee to occur at a lower arrival rate. At higher latencies, a small increase in the transaction load causes a large increase in data contention, and the performance of all the techniques deteriorates. For high latencies, the knees of the PREDICT and BTO curves occur at similar loads.

Figure 37 shows the effect of message latencies in the microsecond range. PREDICT outperforms both 2PL and BTO at all the latencies studied in this experiment, achieving a sustainable transaction load that is of an order of 40% higher than that of 2PL and BTO.

At these microsecond latencies, the iterative build-up of data contention is less severe, and the data-holding time no longer dominates the performance difference between BTO and PREDICT. In this scenario, PREDICT's strategy of reducing aborts through predeclaration and delayed processing of operations pays off. Since PREDICT has fewer restarts than BTO, PREDICT is able to outperform BTO in all cases.

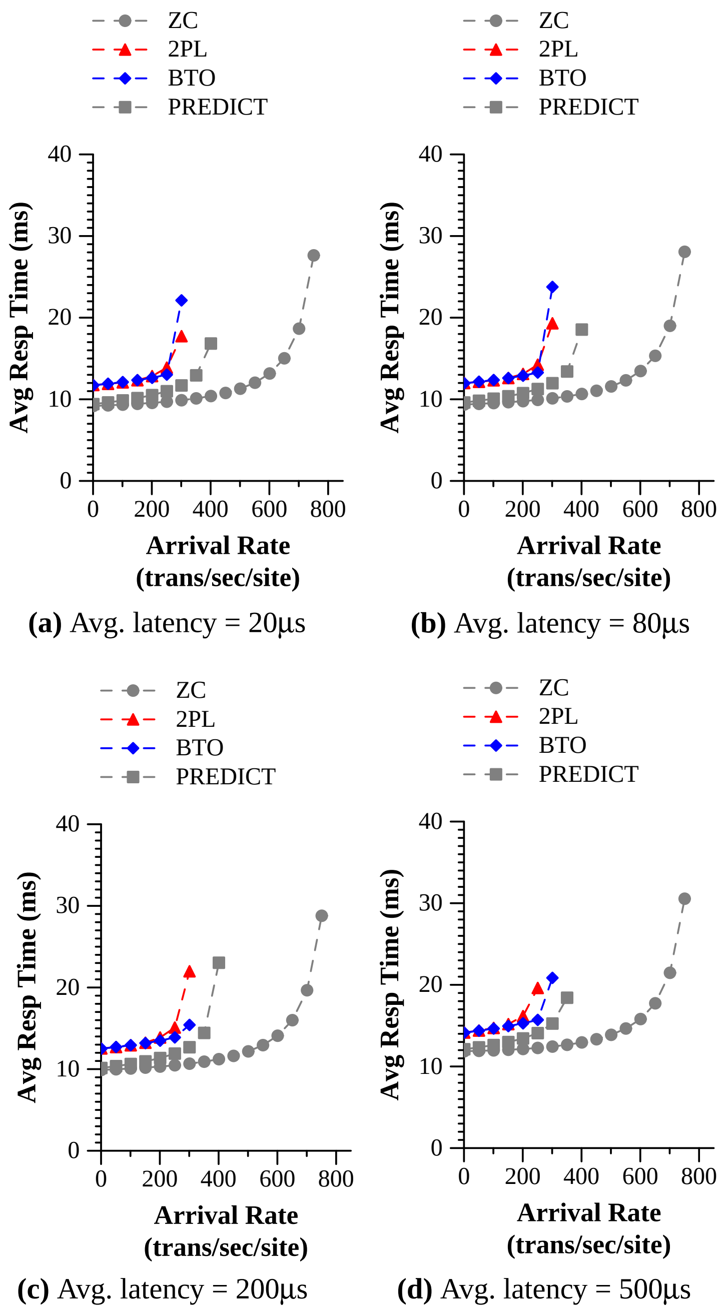


FIGURE 37: Message latency (microseconds) (high data contention)

To summarize the message latency experiments, PREDICT outperforms 2PL in all cases. In addition, PREDICT outperforms BTO when message latencies are low, but performs as well as or worse than BTO when message latencies are high. While both PREDICT and BTO keep scheduler queues shorter than 2PL (by not using read locks, and by aborting rather than queueing some operations), PREDICT also manages to keep the restart cost low. This ability lets PREDICT outperform BTO in situations where data contention is not severe. However, in order to keep the restart cost low, PREDICT pays the price of longer data-holding times than BTO, which can cause it to perform worse than BTO when data contention is severe.

7.4.1.2 Latency Variance

In the following experiment, we studied the effects of a hyperexponential distribution (coefficient of variance > 1) and an Erlang distribution (coefficient of variance < 1) for message latency. Figure 38 shows the performance of PREDICT, 2PL and BTO for different variance values. As the variance increases, the time window within which a transaction operation may arrive increases, increasing the probability of data conflict. This increase in data conflict is manifested as an increased queue length in the 2PL scheduler. The increase in queue length in both PREDICT and BTO is less for two reasons: `read`s don't hold data items; and conflicting operations that arrive out of order are aborted rather than queued. Another effect of the increased variance is to increase the percentage of aborts in BTO and PREDICT. However, in this high data contention scenario, the effect of the queue lengths dominates, and 2PL degrades faster than both PREDICT and BTO as latency variance increases.

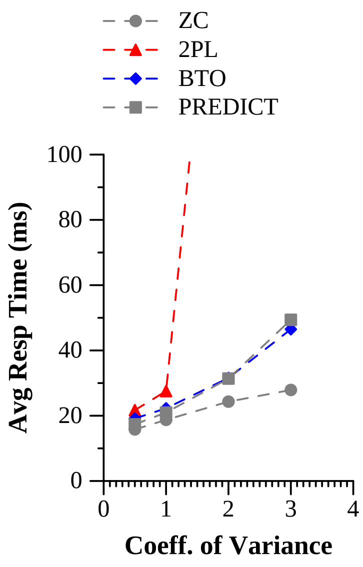


FIGURE 38: Latency variance (high data contention)

7.4.1.3 Processor Speed, Transaction Size and Transaction Composition

Figure 39 shows the effect of different client processor speeds, transaction sizes and read probabilities on the performance of PREDICT, 2PL and BTO. The effect of these

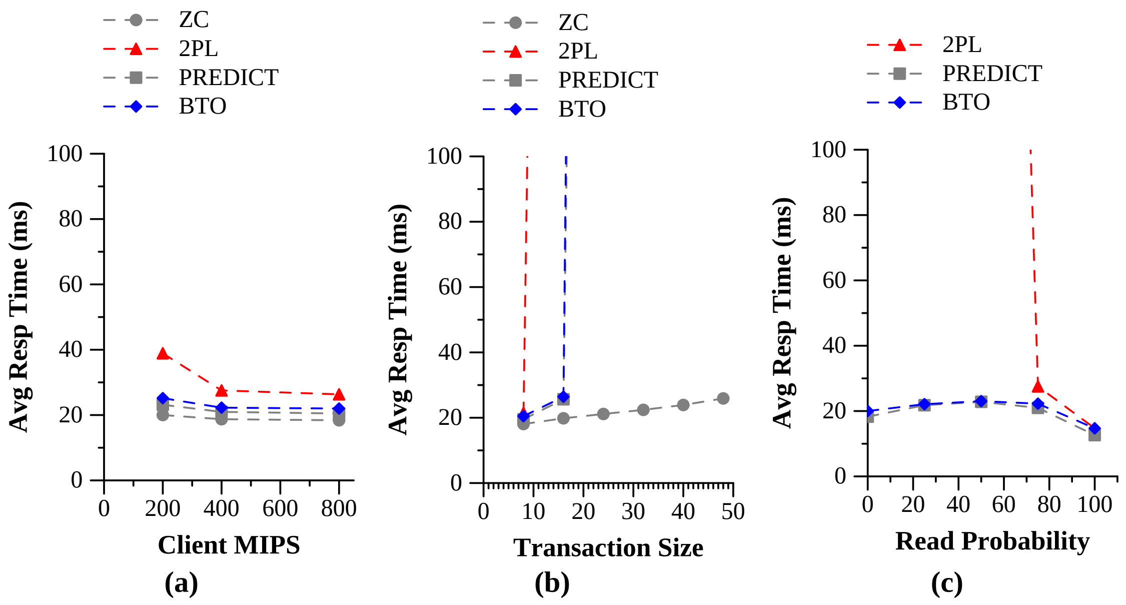


FIGURE 39: Processor speed, transaction size and transaction composition (high data contention)

parameters on PREDICT is similar to the effect on BTO. As parameters change adversely, the iterative build-up of data contention is more severe in 2PL than in PREDICT or BTO, since 2PL has longer queues than PREDICT or BTO. Recall that PREDICT and BTO keep scheduler queues short by not using read locks, and by aborting rather than queueing some operations. The data contention effect is dominant in these experiments, and 2PL is affected more than PREDICT and BTO.

7.4.1.4 Number of Clients

Figure 40 shows the effect of varying the number of clients submitting transactions. As the number of clients increases, the increased data conflict causes an iterative build-up of data contention. Once again, the shorter queues in PREDICT and BTO allow them to outperform 2PL. Predeclaration and delayed processing cause the data-holding time in PREDICT to be higher than that in BTO, and therefore, PREDICT is affected more by the increased load of clients than BTO.

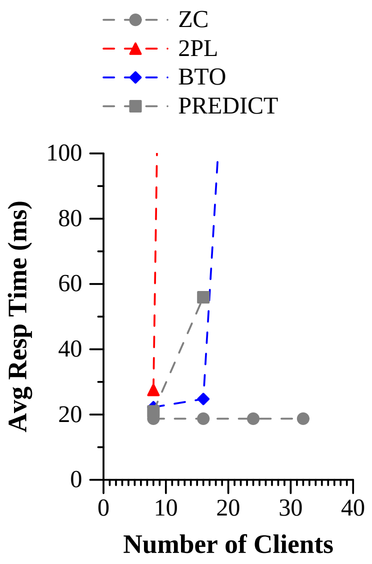
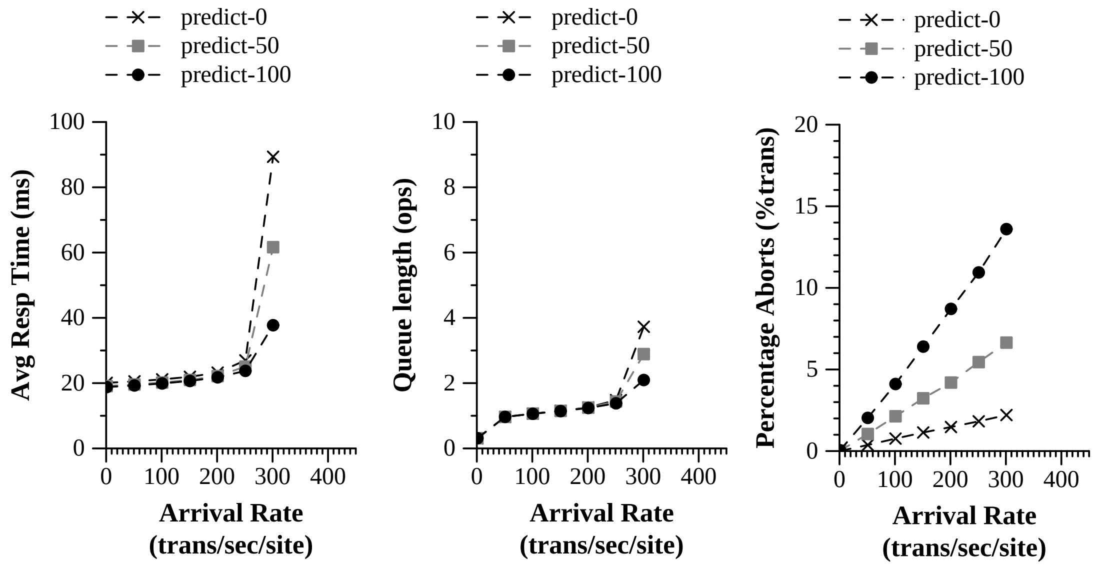


FIGURE 40: Number of clients (high data contention)

7.4.1.5 Degree of aggressiveness

In order to discover how aggressive the servers should be in processing operations, we studied the effect of three different values — 0%, 50% and 100% — for aggressiveness. These values span the range from completely conservative to completely aggressive, with a middle value that tries to achieve a balance between the two extremes. Recall that an aggressiveness of 0% corresponds to total ordering in a system providing perfect predictability. The higher the aggressiveness, the greater the probability of late-arriving operations, and the higher the probability of abort for PREDICT. On the other hand, the lower the aggressiveness, the longer the server waits before processing operations, and some of this waiting may be unnecessary if there is little data conflict. In other words, reducing the aggressiveness increases the lost opportunity cost but reduces the restart cost. Increasing the aggressiveness increases the restart cost but reduces the lost opportunity cost. Figure 41 shows the results from the aggressiveness experiments. The predict-50 curve in the graphs represents an aggressiveness of 50%, and corresponds to the PREDICT curves presented in earlier graphs. The percentage of restarts increases as the degree of aggressiveness goes up (Figure 41c), increasing hardware resource contention. The predict-0 and predict-50 variants achieve a knee that is similar to that of predict-100 (Figure 41a), but with far fewer transactions having to be restarted. The average response time in the stable region is slightly higher for predict-0 as compared to predict-100, since predict-0 suffers from a higher lost opportunity cost by delaying the processing of an operation until the estimated worst-case arrival time of conflicting operations. The predict-50 variant lies between the other two variants in terms of both lost opportunity cost and restart cost. For this default case, all three variants achieve similar performance in terms of response time. Later in this section, we show a scenario in which the performance of the system differs depending on the variant that is used.



(a) Response time **(b)** Queue length **(c)** Restarts

FIGURE 41: Aggressiveness
(high data contention)

Recall that the previous experiments used a default exponential distribution for message latency. It is possible for the message latency to have a distribution with a slowly-decreasing right tail, e.g., the lognormal distribution [Pate00]. A slowly-decreasing tail implies that the estimated worst-case latency must be higher, which results in a higher lost opportunity cost for all PREDICT variants except the most aggressive variant predict-100. The slowly-decreasing tail also implies that there is a larger time window during which operations can arrive out of order, which can increase the percentage of aborts for all the PREDICT variants.

Figure 42 shows the results when the message latency is lognormal with parameters μ=0.75 and σ=0.75. For this scenario, the lost opportunity cost in predict-0 and predict-50 is high enough to cause performance to degrade at a lighter load than that of predict-100. The higher processing delay imposed by predict-0 and predict-50 leads to a longer data-holding time, which causes an iterative increase in data contention. This data contention effect dominates the difference in performance among the variants, causing predict-0 and predict-50 to perform worse than predict-100. The increased hardware resource contention in predict-100 does not significantly degrade its performance until after predict-0 and predict-50 have succumbed to high data contention.

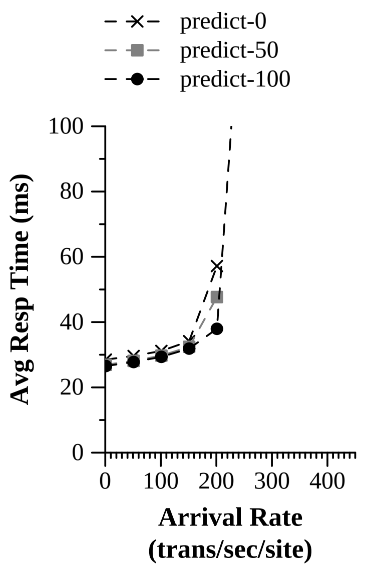


FIGURE 42: Lognormal distribution (high data contention)

In the above scenario, it is beneficial to be aggressive. However, this is not necessarily true for all situations. In general, it is possible to tune the aggressiveness parameter in PREDICT in an attempt to optimize performance for a given set of system parameters and workloads. The choice of the degree of aggressiveness can also be made adaptive, based on the performance of the system in the recent past. Adaptively varying the degree of aggressiveness will be useful if the optimal value is inexpensive to compute, and if the optimal value does not vary very frequently during the course of execution. Finally, the degree of aggressiveness can be modified based on some measure of the popularity of a data item. For instance, operations on hot spots can be processed less aggressively than operations on regular data items, because conflicts involving hot spots are more likely than conflicts involving regular data items. A quantitative study of the above optimizations is outside the scope of my investigation.

7.4.2 Low Data Contention Scenario

We now analyze the results from the low data contention workload experiments. Figure 43 shows the effect of varying the arrival rate of transactions. PREDICT outperforms both 2PL and BTO, achieving a sustainable transaction load of an order of 10-20% better than that of the other two techniques. Recall from section 4.4.2 that hardware resource contention plays the dominant role in determining relative performance in this low data contention scenario. BTO succumbs to high hardware resource contention caused by a high percentage of restarts. On the other hand, PREDICT keeps its restart behaviour low by using predeclaration and by delaying operations before processing them.

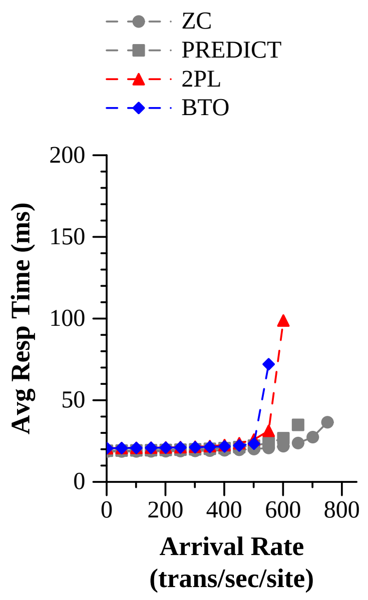


FIGURE 43: Transaction arrival rate (low data contention)

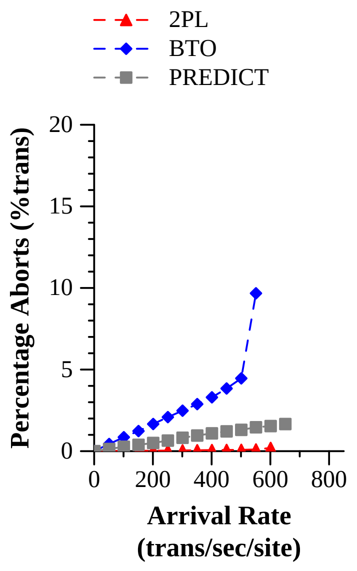
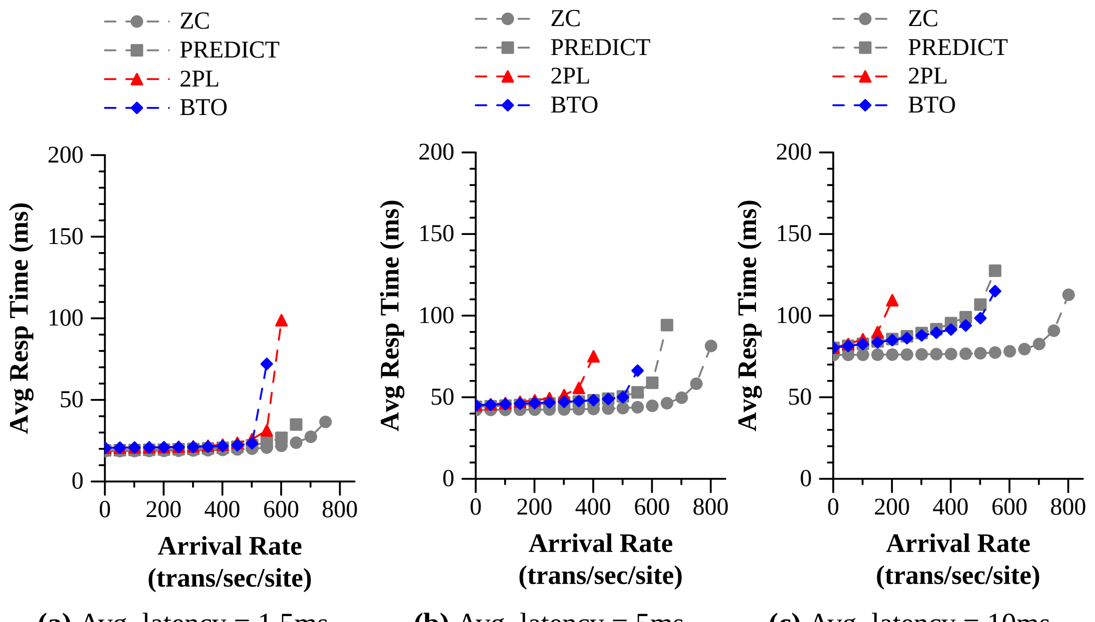


FIGURE 44: Restarts (low data contention)

This difference in the number of restarts allows PREDICT to outperform BTO in the low data contention scenario. Figure 44 shows the restart behaviour of the different techniques. In summary, while both BTO and PREDICT reduce the lost opportunity cost by keeping operation wait time low, PREDICT also manages to keep the restart cost low. This allows PREDICT to outperform 2PL under both levels of data contention.

7.4.2.1Message Latency

Figure 45 shows the effect of different message latencies in the millisecond range. PREDICT outperforms 2PL in all cases. The performance of PREDICT is better than or



(a) Avg. latency = 1.5ms **(b)** Avg. latency = 5ms **(c)** Avg. latency = 10ms

FIGURE 45: Message latency (milliseconds) (low data contention)

similar to BTO at these latencies. As message latency increases, data contention begins to play a larger role in determining performance. The longer data-holding time of PREDICT due to predeclaration and delayed processing of operations reduces PREDICT's advantage over BTO. Therefore, the performance of PREDICT approaches that of BTO as message latency increases.

7.4.2.2 Other Parameters

We studied the effect of other parameters like latency variance, processor speed, number of clients, transaction size and transaction composition. The results were similar to the ones we presented in the high data contention scenario (section 7.4.1).

7.5 Conclusion

We have presented a new timestamp ordering concurrency control technique — PREDICT — that performs well under different levels of data contention. PREDICT retains most of BTO's advantage of shorter queues. In addition, PREDICT uses predeclaration and the delaying of operation processing in order to keep restart behaviour low. PREDICT outperforms 2PL under all of the conditions we studied. PREDICT also performs as well as or better than BTO in several cases. We have presented a set of variants of the PREDICT technique based on reasonable assumptions about network characteristics in a distributed database system. We have demonstrated through simulation that PREDICT achieves a good balance between lost opportunity cost and restart cost, and outperforms popular traditional concurrency control techniques for a wide range of conditions. An adaptive form of PREDICT is possible in which the degree of aggressiveness is varied dynamically based on the behaviour of the system in the recent past. The aggressiveness can also be varied based on some measure of the popularity of the data item being accessed. A detailed study of such systems is left to future work.

Table 7 summarizes the performance differences among all the techniques we have studied.

Conditions	Relative Performance
High data contention workload, low message latencies	(PREDICT = ORDER) > BTO ≥ 2PL
High data contention workload, high message latencies	BTO ≥ PREDICT > 2PL > ORDER*
Low data contention workload, low message latencies	ORDER > PREDICT > 2PL > BTO
Low data contention workload, high message latencies	PREDICT > BTO > 2PL > ORDER*

TABLE 7: Performance comparison of 2PL, BTO, ORDER and PREDICT

* ORDER can perform well at high latencies if the latency penalty ratio is low.

Chapter 8

Recovery from Failures

In this chapter, we explore the issue of processing transactions in a fault-tolerant manner. We do not propose any new recovery algorithms, but show that the ORDER and PREDICT network-aided concurrency control techniques interface with the two-phase commit (2PC) protocol in a straightforward manner. We also show that recovery considerations do not affect the correctness of our results. We show that the impact of recovery overhead on systems that use 2PL, ORDER, PREDICT and BTO is similar. We begin with a discussion of different types of failures in section 8.1. We discuss atomic commitment protocols in section 8.2. In section 8.3, we describe how a standard two-phase commit recovery protocol is applied to 2PL, BTO, ORDER and PREDICT, and study the effect of recovery overhead on these techniques. We discuss the effect of failures on these four techniques in section 8.4.

8.1 Failures

Several types of failures are possible in a database system. A *transaction failure* occurs when a transaction aborts, and is handled by the scheduler, as discussed in the previous chapters. At any given site, there can be two types of failures — *system failures* and *media failures* [BeHG87]. A system failure refers to main memory loss or corruption due to a power failure or an operating system failure. Media failures refer to damaged disks or other stable storage, and can be dealt with using techniques that are similar to the techniques that are used to handle system failures. We concentrate on system failures in this discussion. We do not consider incorrectly-programmed transactions or data entry errors. We assume that transactions halt, and that the execution of any single transaction in isolation preserves database consistency in the absence of failures.

In a distributed system, in addition to system failures at individual sites (*site failures*), there can be *communication failures*. We assume that sites exhibit fail-stop behaviour, which means that when a site fails, processing stops abruptly, and the site never performs incorrect actions. By using redundancy in hardware and software, one can build systems that approximate fail-stop behaviour. When a site recovers from a failure, it executes a *recovery procedure*. Communication failures include corrupted messages, lost messages and network partitioning. Message corruption can be handled by error-detection codes and message retransmission. Message loss can be handled by retransmission and rerouting. In network partitioning, the operational sites are divided into two or more components, and communication between any two components is impossible. Our ability to avoid network partitioning is limited, although the probability of partitioning can be reduced by designing a highly-connected network.

We assume that all failures are detected. Both site failures and communication failures manifest themselves as the inability of sites to exchange messages, and can be detected through timeouts. The underestimation of the timeout period can lead to *timeout failures*, which fall under the category of communication failures. In section 8.2, we will discuss atomic commitment protocols that handle site failures and communication failures.

8.2 Atomic Commitment

An atomic commitment protocol (ACP) is an algorithm which results in all the participants either committing or aborting a transaction, and is essential to maintaining the ACID properties. The basic protocol involves a *coordinator* and a set of *participants*. In order to make a `COMMIT` or `ABORT` decision, the coordinator solicits votes from the participants, the participants vote `Yes` or `No`, and the coordinator makes its decision based on the votes. Formally, an ACP must enforce the following five rules [BeHG87]:

- AC1: All participants that reach a decision reach the same decision.
- AC2: A participant cannot reverse its vote after voting `Yes` or `No` for a `COMMIT`.
- AC3: The `COMMIT` decision can be reached only if all the participants voted `Yes`.
- AC4: If there are no failures and all participants voted `Yes`, the decision is `COMMIT`.
- AC5: At any point, if all failures are repaired, and no new failures occur for sufficiently long, then all participants will reach a decision.

The period between the time a participant voted `Yes` and the time it has received sufficient information to know what the decision will be, is called the *uncertainty period*. During the uncertainty period, a participant does not know whether the decision will be a `COMMIT` or an `ABORT`, and cannot unilaterally decide on an `ABORT`.

8.2.1 Two-Phase Commit

Two-phase commit (2PC) is the simplest and most popular ACP [Gray78, LaSt76]. The algorithm works as follows:

1. The coordinator sends a `vote_request` to all the participants.
2. On receiving a `vote_request`, a participant responds with its vote, which can be a `Yes` or a `No`. The participant aborts if its vote is a `No`.
3. If all the votes were `Yes`, the coordinator sends `COMMIT`s to all the participants that voted `Yes`. Otherwise, the coordinator sends `ABORT`s to all the participants that voted `Yes`.
4. A participant waits for a `COMMIT` or an `ABORT` and decides accordingly.

A participant's uncertainty period starts after step 2, and ends in step 4 when it receives a `COMMIT` or `ABORT`. If the participant times out while waiting for the `vote_request`, the participant unilaterally decides `ABORT`. If the coordinator times out while waiting for votes, the coordinator decides `ABORT`, and sends `ABORT` to all the participants that voted `Yes`. If the participant times out while waiting for the `COMMIT` or `ABORT`, the participant executes a *termination protocol*. The simplest termination protocol is to block until communication with the coordinator is restored, and then retrieve the `COMMIT` or `ABORT` decision. A cooperative termination protocol can result in less blocking than the simple termination protocol, and works as follows. Assume that the coordinator appends the list of participants to its `vote_request`. The participant sends a `decision_request` to every other participant. A participant sends a `COMMIT` or `ABORT` if it knows the decision, does nothing if it is uncertain, and sends an `ABORT` if it has not voted yet. The cooperative termination protocol can still block if all the participants are uncertain and the coordinator has failed.

In order to be able to recover correctly from a failure, the coordinator and the participants must record every `Yes`, `No`, `COMMIT` or `ABORT` message on stable storage before sending it. When a participant is recovering from a failure, it executes the following recovery procedure. The participant decides `ABORT` if it had not voted yet. If the decision is recorded on stable storage, the participant decides accordingly. The participant executes the termination protocol if it is uncertain. 2PC is resilient to site failures and all communication failures. It needs three rounds of messages when there are no failures, and five rounds of messages when there are any number of failures [BeHG87].

8.2.2 Other ACPs

Two variants of 2PC that attempt to improve the efficiency of the protocol are decentralized 2PC [Skee82] and linear 2PC [Gray78, RoSL78]. Three-Phase Commit protocols have also been proposed, in an attempt to reduce the frequency of blocking [Skee82]. In most practical applications, 2PC blocking happens rarely. Consequently, almost all systems that we know of employ some version of 2PC.

8.3 Recovery Overhead in Transaction Processing

In this section, we will describe how a standard 2PC recovery protocol fits into the transaction processing architecture. Recall that the scheduler forwards `read`s and committed `write`s to the DM in an order that produces a serializable execution of transactions. The DM is responsible for ensuring correct execution in the presence of failures. Every site has access to a *log*, which is stable storage that stores information about the history of execution, and enables recovery from failures. At the client, the TM maintains a log, and at the server, the DM maintains a log. The log is stable storage that is logically (and possibly physically) separate from the disk that records the actual database. In this discussion, we assume that the granularity of data items supported by the DM is identical to that supported by the stable storage. We also assume that the DM follows the *redo* policy rather the *undo* policy. A *redo* policy implies that the effects of only committed transactions are written to disk, and in order to recover from a failure, the DM has to redo the effects of uncommitted transactions. An *undo* policy implies that the effects of uncommitted as well as committed transactions are stored on disk, and in order to recover from a failure, the DM must undo the effects of uncommitted transactions. The redo assumption can be replaced with an undo assumption without affecting the conclusions of the following discussion. The operating system sends a `restart` to the DM upon recovery from a system failure. A `restart` causes the DM to initiate a recovery procedure.

We now describe how a standard 2PC recovery protocol interfaces with the 2PL, ORDER, PREDICT and BTO concurrency control algorithms. In all of these systems, a process at the initiating TM acts as the coordinator of the 2PC protocol, and processes at the DMs at the servers in the write set act as the participants. Note that it is not necessary to have a separate process at each site for each transaction that is participating in the 2PC protocol. A site may have one or more processes that are responsible for executing the 2PC protocol for all transactions communicating with that site. In the following discussion, we will omit the word “process” and simply use the terms TM and DM.

8.3.1 Recovery Overhead in 2PL

Recall that in a 2PL system, when a transaction is ready to commit, the initiating TM sends lock releases corresponding to every `read` issued by the transaction, and `commit`s corresponding to the `write`s. The lock releases result in the read locks being released by the scheduler, while the `commit` results in committed `write`s being forwarded by the scheduler to the DM, and in the write locks being released. In order to enable recovery, in addition to forwarding committed `write`s to the DM, the scheduler also forwards a `vote_request` to the DM. Moreover, the write locks are not released immediately, but are held until the end of the 2PC protocol. On receiving a committed `write`, the DM records it on the log. On receiving a `vote_request`, the DM waits until it has recorded all the committed `write`s of the transaction on the log, and then votes `Yes`. In order to vote `Yes`, the DM records its `Yes` vote on the log and then sends a `Yes` to the initiating TM. The initiating TM waits until it has received votes from all the servers that store part of the transaction's write set (or timed out), and then makes its decision of `COMMIT` or `ABORT`. The initiating TM records this decision on its log, and then sends the decision to all the participants. On receiving a `COMMIT`, the DM writes the `COMMIT` decision to the log, releases the write lock, and copies the committed `write` values of the transaction from the log to the disk.

Therefore, the read-lock-holding time remains the same even with the overhead of the recovery protocol, but the write-lock-holding time increases by the amount of time it takes to complete the 2PC protocol. The increased lock-holding time lowers data availability, and has the iterative degradation effect on performance discussed in earlier chapters.

8.3.2 Recovery Overhead in ORDER and PREDICT

Recall that in both ORDER and PREDICT, when a transaction is ready to commit, the initiating TM sends `commit`s to every server that stores part of the transaction's write set. On receiving a `commit`, the scheduler replaces the `predeclare`s of the transaction with committed `write`s, forwards these committed `write`s to the DM when they get to the head of the queue, and then deletes them from the queues. In order to enable recovery, the scheduler responds to a `commit` by forwarding a `vote_request` to the DM. Moreover, the committed `write`s are not deleted from the queues after being forwarded to the DM, but are retained on the queues until the end of the 2PC protocol. On receiving a committed `write`, the DM records it on the log. On receiving a `vote_request`, the DM waits until it has recorded all the committed `write`s of the transaction on the log, and then votes `Yes`. The DM and the initiating TM execute the rest of the 2PC protocol as in the 2PL case described in section 8.3.1. On receiving a `COMMIT`, the DM writes the `COMMIT` decision to the log, allows the committed `write`s to be deleted from the scheduler queues, and copies the committed `write` values of the transaction from the log to the disk.

The retention of committed `write`s on the scheduler queues until the end of the 2PC protocol results in an increased data-holding time, reducing data item availability. The increased data-holding time has an iterative effect on degradation of performance.

8.3.3 Recovery Overhead in BTO

The recovery protocol works in the same way for BTO as for ORDER and PREDICT, except that the scheduler replaces `write`s with committed `write`s, on receiving a `commit`. The DMs and the initiating TM follow the same protocol as in section 8.3.2 in order to execute a 2PC. As in ORDER and PREDICT, the retention of committed `write`s on the scheduler queues until the end of the 2PC protocol results in an increased data-holding time, reducing data item availability. The increased data-holding time has an iterative effect on degradation of performance. In addition, a longer data-holding time implies that there is a greater probability that late-arriving operations will be inserted into the scheduler queues in timestamp order, instead of being rejected. Therefore, the number of aborts in BTO may decrease because of the recovery overhead. This effect is not present in ORDER and PREDICT, because ORDER is a completely conservative scheme without aborts, and PREDICT rejects out-of-order operations as soon as they are delivered.

In summary, the overhead of the recovery protocol increases the data-holding time for the write set data items in all four schemes in exactly the same manner. This implies that the knee of the performance curve will occur at a lower transaction load for all the schemes, when recovery overhead is considered. In addition, BTO may experience a reduction in the number of aborts due to the recovery overhead. In future work, it would be interesting to determine the quantitative effect of this reduction in the number of aborts on BTO's performance.

8.4 Effect of Failures

As discussed in section 8.1, failures in a distributed system can be classified into site failures and communication failures. We now discuss the effects of these failures on systems that use 2PL, BTO, PREDICT and ORDER.

8.4.1 Effect of Failures on 2PL

A site can crash while holding onto read locks or write locks, stalling the progress of other transactions. Site crashes can be detected through timeouts, and are handled in the 2PC protocol as described in section 8.2.1.

Lost messages have to be retransmitted. As message retransmissions increase, lock-holding times can increase, slowing down transactions. Network partitions cause timeouts in the same way that site crashes do, and transactions can be blocked until the partition is repaired. However, it may be possible for some transactions to make progress in individual partitions.

8.4.2 Effect of Failures on BTO and PREDICT

A site can crash before sending out `commit`s or `abort`s. This means that `write`s or `predeclare`s can reach the head of the scheduler queues, and be retained there while the initiating site is down. This retention of `write`s or `predeclare`s on the scheduler queues due to site crashes has the same effect as the holding of write locks in 2PL, stalling the progress of other transactions. Note that there is no effect in BTO and PREDICT corresponding to that of read locks in 2PL, because BTO and PREDICT do not use read locks.

Lost messages have to be retransmitted. As message retransmissions increase, data-holding times can increase, slowing down transactions. In addition, retransmitted messages may arrive out of timestamp order, causing the frequency of aborts to increase in BTO and PREDICT. Message retransmissions should be taken into account in the estimation of the worst-case transmission delay used by PREDICT.

Network partitions cause timeouts in the same way that site crashes do, and transactions can be blocked until the partition is repaired. It may be possible for some transactions to make progress in individual partitions. When the partition is repaired, waiting transactions can have their operations rejected because they arrived too late, and higher-timestamped operations have already been executed. Therefore, partition recovery can involve a large number of aborted transactions.

8.4.3 Effect of Failures on ORDER

A site can crash before sending out `commit`s or `abort`s. This means that `predeclare`s can get to the head of the scheduler queues, and be retained there while the initiating site is down. This retention of `predeclare`s on the scheduler queues due to site crashes has the same effect as the holding of write locks in 2PL, stalling the progress of other transactions. Note that there is no effect in ORDER corresponding to that of read locks in 2PL, because ORDER does not use read locks.

Lost messages have to be retransmitted. As message retransmissions increase, data-holding times can increase, slowing down transactions. Network partitions cause timeouts in the same way that site crashes do, and transactions can be blocked until the partition is repaired.

Communication failures may affect the total ordering guarantee provided by the network, resulting in messages being delivered out of order. We assume that a destination can detect ordering violations, for example, through timestamps. For the isotach-based ORDER system, this means that the network has to be able to deal with lost tokens. Lost tokens can be handled through timeouts. When a destination receives an operation in a manner that violates the total ordering guarantee, the destination votes `No` in the 2PC protocol in order to abort the transaction and ensure correct execution.

In case of network partitioning, it may be possible for some transactions to make progress in individual partitions. If individual partition progress is permitted, partition recovery may involve aborts in order to maintain the total ordering of transactions.

8.5 Conclusion

Distributed systems are subject to different types of failure, and a transaction processing system needs a recovery algorithm in order to ensure correct execution. Two-phase commit (2PC) is the most popular recovery protocol. The ORDER and PREDICT network-aided concurrency control techniques interface with 2PC in a straightforward manner. The impact of recovery overhead on systems that use 2PL, ORDER, PREDICT and BTO is similar. In addition, recovery overhead may reduce the amount of restart behaviour in BTO, and a performance study of this effect has not been explored here.

Site failures as well as communication failures can be handled by all four systems, using the 2PC recovery protocol. The impact of failures is felt in the form of increased data-holding times in all four schemes, especially in 2PL and ORDER, which are conservative techniques. The effect of increased data-holding time is especially severe in 2PL because it can affect **read** data items as well as written data items. Another effect of the failures is an increase in the restart behaviour, and this effect is seen in BTO and PREDICT in the same manner. An investigation of the precise quantitative effect of different modes of failure on performance is beyond the scope of this dissertation.

Chapter 9

Conclusion

Before now, the world would have turned to dynamic 2PL as the technique for concurrency control in a distributed database system. As a result of our work, timestamp ordering techniques and ordered networks must now be given serious consideration. As a result of our work, the analytic tools for evaluating choices *accurately* is now possible. Of course, much remains to be explored.

9.1 Results and Contributions

We have demonstrated that network properties can be exploited to achieve efficient concurrency control of transactions. The network can act as a powerful coordination mechanism by providing certain useful properties in the form of communication guarantees. Network-aided concurrency control techniques use such properties to keep the lost opportunity cost and restart cost low. We have presented a new family of concurrency control techniques that use the interconnection network in a distributed database system as an aid to concurrency control.

There are several network properties that are useful to concurrency control. If the network can provide such a property at a low cost, efficient concurrency control techniques can be built on top of it. We have identified five useful network properties — total ordering, predictability, extended predictability, pruning and caching. We have discussed how each of these properties can help with concurrency control, and presented concurrency control techniques built on top of these properties. We have explored two of these properties — total ordering and predictability — in detail.

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 concurrency control. We have studied ORDER analytically as well as through a simulation. ORDER outperforms both dynamic 2PL and BTO when network latency is low. ORDER's advantage disappears only when network latency is high and ordering is implemented inefficiently. The performance of the isotach prototype implies that ordering can be implemented efficiently. ORDER is a good candidate for high data contention as well as low data contention workloads, as long as message latencies are low. At high latencies, the latency penalty ratio of ORDER must be very low in order for it to perform well.

We have presented a new timestamp ordering concurrency control technique — PREDICT — that performs well under different levels of data contention. PREDICT retains most of BTO's advantage of short queues, and outperforms 2PL under all the conditions that we studied. In addition, PREDICT uses the two policies of predeclaration and the delaying of operation processing in order to keep restart behaviour low, consequently performing better than BTO in most cases. However, these policies also result in a longer data-holding time, which affects performance when data contention and message latency are already high. Therefore, PREDICT performs worse than BTO when data contention and message latency are both high. We have presented a set of variants of the PREDICT technique based on reasonable assumptions about network characteristics in a distributed database system. We have demonstrated through simulation that PREDICT achieves a good balance between lost opportunity cost and restart cost, and outperforms popular traditional concurrency control techniques for a wide range of conditions. Table 8 (repeated from Chapter 7) summarizes the relative performance of the different concurrency control techniques we have studied.

Conditions	Relative Performance
High data contention workload, low message latencies	(PREDICT = ORDER) > BTO ≥ 2PL
High data contention workload, high message latencies	BTO ≥ PREDICT > 2PL > ORDER*
Low data contention workload, low message latencies	ORDER > PREDICT > 2PL > BTO
Low data contention workload, high message latencies	PREDICT > BTO > 2PL > ORDER*

TABLE 8: Concurrency control performance comparison

* ORDER can perform well at high latencies if the latency penalty ratio is low.

We have contributed to the evaluation of concurrency control techniques. Traditional analytical models for concurrency control are inadequate, in that they do not model performance accurately when data contention is high. Another limitation of traditional analyses for distributed databases is that they do not model a fully distributed database. We have presented a new analytical modelling technique that addresses these limitations by accurately predicting performance under different levels of data contention, and by modelling a fully distributed database. We have applied our modelling technique to 2PL and ORDER, and validated the results against simulations that we have described in detail. We have shown that the ability to model arbitrary queue lengths enables our models to predict performance accurately, even under high data contention.

Popular conception has been that timestamp ordering techniques perform poorly as compared to 2PL. We have shown the surprising result that the performance of BTO is better than that of 2PL for a wide range of conditions. BTO outperforms 2PL for all cases except when data contention and message latency are both low. In addition, a timestamp ordering technique based on network properties — PREDICT — outperforms 2PL under all the conditions we studied. An important contribution of our work is the motivation of a re-evaluation of the merits of timestamp ordering concurrency control in distributed databases.

The iterative build-up of data contention is an important factor determining performance, and is more severe in 2PL than in ORDER, PREDICT or BTO, due to longer queue lengths in 2PL. Hardware resource contention is also an important factor determining performance, and is more severe in BTO and PREDICT than in 2PL and ORDER, due to a high percentage of restarts. Small increases in data contention lead to significant performance effects, because of the iterative build-up phenomenon. Increasing factors like message latency, transaction size, number of clients and probability of write increases data contention, and causes performance to degrade. When data contention is low, hardware resource contention begins to affect performance differences more.

The low queue lengths in ORDER and PREDICT allow them to outperform 2PL for a wide range of conditions. While BTO also maintains low queue lengths, its performance is sensitive to the amount of restart behaviour under some conditions. The absence of restarts in ORDER gives ORDER an advantage over BTO for situations where hardware resource contention plays an important role in determining performance. On the other hand, the latency penalty incurred by ORDER gives ORDER a disadvantage when message latencies are high. The two policies of predeclaration and delayed processing of operations help keep restarts low in PREDICT, allowing it to outperform BTO under a range of conditions. However, when data contention and message latencies are both high, these same policies prove to be a disadvantage to PREDICT, by increasing data-holding time to levels that degrade performance. The ability to control the amount of restart behaviour is an important asset to PREDICT, and allows a database designer to tune performance based on the workload characteristics and system parameters.

Our work provides valuable insight into the choice of concurrency control technique for a given system. It is clear from our studies that dynamic 2PL is a poor choice because it suffers from high data contention at lighter loads than other techniques. ORDER and PREDICT are both very good choices when message latencies are low, because they keep both queue lengths and restart behaviour down. When message latencies are high, PREDICT and BTO are good choices, because they keep queue lengths lower than 2PL, and do not suffer from the latency penalty of ORDER. Both ORDER and PREDICT require predeclaration of accesses. If predeclaration is not possible or too expensive, BTO is a good choice, because it performs better than or as well as 2PL in all cases except when data contention and message latencies are both low.

Distributed systems are subject to different types of failure, and two-phase commit (2PC) is the most popular recovery protocol. We have shown that ORDER and PREDICT interface with 2PC in a straightforward manner, by recording committed `write`s on a log, and writing them to disk after the TM and DMs participate in the 2PC protocol. The impact of recovery overhead on systems that use 2PL, ORDER, PREDICT and BTO is similar. In addition, recovery overhead may reduce the amount of restart behaviour in BTO, and a performance study of this effect is outside the scope of my investigation. Site failures as well as communication failures can be handled by all four systems, using the 2PC recovery protocol. The impact of failures occurs in the form of increased data-holding times in all four schemes, especially in 2PL and ORDER. The effect of increased data-holding time is especially severe in 2PL because it can affect **read** data items as well as written data items. Another effect of the failures is an increase in the restart behaviour, and this effect is reflected in BTO and PREDICT in the same manner. An investigation of the precise quantitative effect of different modes of failure on performance has not been explored here.

9.2 Limitations and Future Work

The network-aided concurrency control techniques of ORDER and PREDICT require a transaction to predeclare all accesses. As we have seen earlier, predeclaration may be impossible or too expensive for some applications. Concurrency control techniques based on the other network properties of extended predictability, pruning and caching do not require predeclaration. Detailed performance evaluations of these techniques would widen the applicability of network-aided concurrency control to a more general set of applications, by including applications for which predeclaration is not possible. Another avenue for future research is the exploration of other network properties that are useful to concurrency control.

Our performance studies are based on the assumption that there is no replication. Our concurrency control techniques work with replicated data as well, as long as `predeclare`s and `commit`s for a data item are sent to all servers that store that data item. However, a quantitative study of the relative performance of various techniques under the assumption of replicated data deserves a further look.

We have assumed serializability to be the correctness criterion for this work. It would be interesting to study network-aided techniques that guarantee weak consistency semantics. A property that may be useful for weak consistency semantics is *causal ordering*, which guarantees that messages are delivered according to a causal order (which is a partial order) at all destinations. For instance, causal ordering may be useful when it is legal for two concurrent `write`s to be viewed in different orders by two destinations. In this case, data availability of the system would be improved over the case where all operations are totally ordered. Causal ordering can be implemented using vector clocks [BiSS91, LLSG92, ScES89].

Our analytical modelling technique has been applied to 2PL and ORDER, but not to restart-oriented techniques like PREDICT and BTO. Adding the ability to model restart-oriented techniques would be a useful extension to our analytical modelling technique. Moreover, extending the analytical technique to model hot spots is another possible improvement.

It may be possible to apply network-aided techniques to special types of databases like real-time databases. An exploration of network properties that are useful to such systems is an avenue for future work.

Finally, it would be interesting to study the adaptive form of PREDICT, and to explore the effect of wait-depth-limiting rules on timestamp ordering techniques.

References

AGEL94 Agrawal D., El Abbadi A. and Lang A. E., The Performance of Protocols Based on Locks with Ordered Sharing, IEEE Transactions on Knowledge and Data Engineering 6/5, Oct 1994.

AGLM95 Adya A., Gruber R., Liskov B. and Maheshwari U., Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks, Proceedings of the ACM SIGMOD International Conference on the Management of Data, May 1995.

ALDA76 Alsberg P. A. and Day J. D., A Principle for Resilient Sharing of Distributed Resources, Proceedings of the 2nd International Conference on Software Engineering, Oct 1976.

ANSI92 ANSI X3.135-1992, American National Standard for Information Systems — Database Language — SQL, Nov 1992.

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 Engineering, 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.
BIJO87	Birman K. P. and Joseph T. A., Reliable Communication in the Presence of Failures, ACM Transactions on Computer Systems 5/1, Feb 1987, pp 47-76.
BISS91	Birman K. P., Schiper A. and Stephenson P., Lightweight Causal and Atomic Group Multicast, ACM Transactions on Computer Systems 9/3, Aug 1991.
CAFZ94	Carey M., Franklin M. and Zaharioudakis M., Fine Grained Sharing in a Page-Server OODBMS, Proceedings of the 1994 ACM SIGMOD, May 1994.
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.

CASD85 Cristian F., Aghili H., Strong R. and Dolev D., Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement, Proceedings of the 15th International Symposium on Fault-Tolerant Computing, 1985, pp 200-206.

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

CIDY90 Ciciani B., Dias D. and Yu P., Analysis of Replication in Distributed Database Systems, IEEE Transactions on Knowledge and Data Engineering 2/2, Jun 1990, pp 247-261.

CIDY92 Ciciani B., Dias D. and Yu P., Analysis of Concurrency-Coherency Control Protocols for Distributed Transaction Processing Systems with Regional Locality, IEEE Transactions on Software Engineering 18/10, Oct 1992, pp 889-914.

CRIS89A Cristian F., Probabilistic Clock Synchronization, Distributed Computing 3, 1989, pp 146-158.

CRIS89B Cristian F.: Synchronous Atomic Broadcast for Redundant Broadcast Channels, IBM Research Report RJ 7203, Dec 1989.

DATE95 Date C. J., An Introduction to Database Systems, Sixth Edition, Addison-Wesley, 1995.

DATE00 Date C. J., An Introduction to Database Systems, Seventh Edition, Addison-Wesley, 2000.

DOMA96 Dolev D. and Malkhi D., The Transis Approach to High Availability Cluster Communication, Communications of the ACM 39/4, Apr 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.

FRAN99 Franklin M. (Associate Professor, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley), Personal Communication, Nov 1999.

FRRO85 Franaszek P. A. and Robinson J. T., Limitations of Concurrency in Transaction Processing, ACM Transactions on Database Systems 10/1, Mar 1985, pp 1-28.

FRRO93 Franaszek P. A. and Robinson J. T., Distributed Concurrency Control Based on Limited Wait Depth, IEEE Transactions on Parallel and Distributed Systems 4/11, Nov 1993, pp 1246-1264.

FRRT92 Franaszek P. A., Robinson J. T. and Thomasian A., Concurrency Control for High Contention Environments, ACM Transactions on Database Systems 17/2, Jun 1992.

GARC79 Garcia-Molina H., Performance of Update Algorithms for Replicated Data in a Distributed Database, PhD dissertation, Computer Science Department, Stanford University, Jun 1979.

GASA87 Garcia-Molina H. and Salem K., Sagas, Proceedings of the 1987 SIGMOD Conference on Management of Data, May 1987.

GRAY78 Gray J. N., Notes on Database Operating Systems, Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60, Springer-Verlag, Berlin, 1978, pp 393-481.

GRAY96 Gray J., The Dangers of Replication and a Solution, ACM SIGMOD Conference, 1996, pp 173-182.

GRAY99 Gray J. N., How High is High Performance Transaction Processing?, Presentation at the High Performance Transaction Processing Workshop (HPTS99), Asilomar, California, 26-29th Sep 1999.

GRRE92 Gray J. N. and Reuter A., Transaction Processing: Concepts and Facilities, Morgan-Kaufmann, 1992.

GSSS01	Greenwald R., Stern J. and Stackowiak R., Oracle Essentials: Oracle9i, Oracle8i & Oracle8 (2nd Edition), O'Reilly & Associates, ISBN 0596001797, Jun 2001.
HADO91	Halici U. and Dogac A., An Optimistic Locking Technique For Concurrency Control in Distributed Databases, Transactions on Software Engineering 17/7, Jul 1991.
HARO93	Haerder T. and Rothermel K., Concurrency Control Issues in Nested Transactions, International Journal of Very Large Databases 2, 1993.
HOLT72	Holt R. C., Some Deadlock Properties of Computer Systems, ACM Computing Surveys 4/3, Dec 1972.
HSZH92	Hsu M. and Zhang B., Performance Evaluation of Cautious Waiting, ACM Transactions on Database Systems 17/3, Sep 1992, pp 477-512.
JASH92	Jagadish H. V. and Shmuelli O., A Proclamation Based Model for Cooperating Transactions, Proceedings of the 18th International Conference on Very Large Databases, 1992.
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.
KATA91	Kaashoek M. F. and Tanenbaum A., Group Communication in the Amoeba Distributed Operating System, Proceedings of the 11th International Conference on Distributed Computing Systems, May 1991, pp 222-230.
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.
LAMP78	Lamport L., Time, Clocks and Ordering of Events in a Distributed System, Communications of the ACM 21/7, Jul 1978.
LAMY00	Lack M. N. and Myers, P., The Isotach Messaging Layer: Ironman Design, Technical Report CS-2000-17, Department of Computer Science, University of Virginia, May 2000.

LAST76	Lampson B. and Sturgis H., Crash Recovery in a Distributed Data Storage System, Technical Report, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, California, 1976.
LAVE83	Lavenberg S. (Ed.), Computer Performance Modeling Handbook, Academic Press, Orlando, Florida, 1983.
LEGT98	Lehman L., Garland S. and Tennenhouse D., Active Reliable Multicast, Proceedings of IEEE INFOCOM98, San Francisco, California, Mar 1998.
LEWG98	Legedza U., Wetherall D. and Guttag G., Improving the Performance of Distributed Applications Using Active Networks, Proceedings of IEEE INFOCOM98, San Francisco, California, Mar 1998.
LI87	Li V., Performance Models of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases, IEEE Transactions on Computers 36/9, Sep 1987.
LINO83	Lin and Nolte, Basic Timestamp, Multiple Version Timestamp and Two-Phase Locking, Proceedings of the 9th VLDB Conference, Nov 1983.
LISK91	Liskov B. et al, Replication in the Harp File System, Proceedings of the 13th ACM Symposium on Operating Systems Principles, Oct 1991, pp 226-238.
LLSG92	Ladin R., Liskov B., Shrira L. and Ghemawat S., Providing Availability Using Lazy Replication, ACM Transactions on Computer Systems 10/4, Nov 1992, pp 360-391.
MILL88	Mills D. L.: Network Time Protocol: Specification and Implementation, DARPA-Internet Report RFC 1059, DARPA, Jul 1988.
MIPS91	Mishra S., Peterson L. L. and Schlichting R. D., A Membership Protocol Based on Partial Order, Proceedings of the IEEE International Working Conference on Dependable Computing for Critical Applications, Feb 1991, pp 137-145.
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.

MULL90 Mullender S. J. et al, Amoeba — A Distributed Operating System for the 1990s, IEEE Computer 23/5, May 1990, pp 44-53.

NETO93 Neiger G. and Toueg S., Simulating synchronized clocks and common knowledge in distributed systems, Journal of the ACM 40/2, Apr 1993.

OWGR76 Owicki S. and Gries D., An Axiomatic Proof Technique for Parallel Programs I, Acta Informatica 6, 1976.

OZSU94 Ozsu M. T., Transaction Models and Transaction Management in Object-Oriented Database Management Systems, Springer-Verlag, 1994.

PATE00 Patek S. (Assistant Professor, Dept. of Systems Engineering, University of Virginia), Personal Communication, Oct 2000.

RACH96 Ramamithram K. and Chrisanthis P. K., Advances in Concurrency Control and Transaction Processing, IEEE Computer Society Press, 1996.

REED78 Reed D. P., Naming and Synchronization in a Decentralized Computer System, PhD dissertation, Department of Electrical Engineering, Massachusetts Institute of Technology, Sep 1978.

REGE97 Regehr J., An Isotach Implementation for Myrinet, Technical Report CS-97-12, Department of Computer Science, University of Virginia, May 1997.

REST89 van Renesse R., van Staveren H. and Tanenbaum A., The Performance of the Amoeba Distributed Operating System, Software — Practice and Experience 19/3, March 1989, pp 223-234.

RETH96 Ren J., Takahashi Y. and Hasegawa T., Analysis of Impact of Network Delay on Multiversion Conservative Timestamp Algorithms in DDBS, Performance Evaluation 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. and Thomasian A., Analysis of Database Performance with Dynamic Locking, Journal of the ACM 37/3, Jul 1990, pp 491-523.

RyTh90b Ryu I. and Thomasian A., Performance Analysis of Dynamic Locking with the No-Waiting Policy, IEEE Transactions on Software Engineering 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.

ScES89 Schiper A., Eggli J. and Sandoz A., A New Algorithm to Implement Causal Ordering, Proceedings of the 3rd International Workshop on Distributed Algorithms, Berlin, 1989, pp 219-232.

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 17th International Conference on Distributed Computing Systems, May 1997.

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

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

Skee82 Skeen D., Nonblocking Commit Protocols, Proceedings of the ACM SIGMOD Conference on the Management of Data, Orlando, Florida, Jun 1982, pp 133-147.

SKZD89 Skarra A. H. and Zdonik S. B., Concurrency Control and Object-Oriented Databases, Object-Oriented Concepts, Databases and Applications, ACM Press, 1989.

SRWR01A Srinivasa R., Williams C. and Reynolds P. F., Distributed Transaction Processing on an Ordering Network, Technical Report CS-2001-08, Department of Computer Science, University of Virginia, Feb 2001.

SRWR01B Srinivasa R., Williams C. and Reynolds P. F., A New Look at Timestamp Ordering Concurrency Control, 12th International Conference on Database and Expert Systems Applications (DEXA 2001), Munich, Sep 2001.

STON79 Stonebraker M., Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES, IEEE Transactions on Software Engineering, SE-5/3, May 1979.

STRO81 Stearns R. E. and Rosenkrantz D. J., Distributed Database Concurrency Controls Using Before-Values, Proceedings of the ACM SIGMOD Conference on the Management of Data, 1981.

TAGS85 Tay Y., Goodman N. and Suri R., Locking Performance in Centralized Databases, ACM Transactions on Database Systems 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 Transactions on Database Systems 18/4, Dec 1993, pp 579-625.

THOM98A Thomasian A., Concurrency Control: Methods, Performance, and Analysis, ACM Computing Surveys 30/1, Mar 1998, pp 70-119.

THOM98B Thomasian A., Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing, IEEE Transactions on Knowledge and Data Engineering 10/1, Jan/Feb 1998, pp 173-189.

THRY91 Thomasian A., Ryu I., Performance Analysis of Two-Phase Locking, IEEE Transactions on Software Engineering 17/5, May 1991, pp 386-401.

TSSW97 Tennenhouse D., Smith J., Sincoskie D., Wetherall D. and Minden G., A Survey of Active Network Research, IEEE Communications Magazine, 35/1, Jan 1997, pp 80-86.

WAKE99 Wang J., Keshav S.: Efficient and Accurate Ethernet Simulation, Proceedings of the 24th Conference on Local Computer Networks (LCN ‘99), Oct. 1999.

WALK98 Wang J., Li J. and Kameda H., Distributed Concurrency Control with Local Wait-Depth Policy, IEICE Transactions on Information and Systems E81-D/6, Jun 1998, pp 513-520.

WEIH88 Weihl W. E., Commutativity Based Concurrency Control for Abstract Data Types, IEEE Transactions on Computers 37/12, Dec 1988.

WILL93 Williams C., Concurrency Control in Asynchronous Computations, PhD Dissertation, Department of Computer Science, University of Virginia, 1993.

YDRI85 Yu P., Dias D., Robinson J., Iyer B. and Cornell D., Modelling of Centralized Concurrency Control in a Multi-System Environment, Performance Evaluation Review 13/2 (Proceedings of the 1985 ACM SIGMETRICS), 1985, pp 183-191.

YUDI92 Yu P. and Dias D., Analysis of Hybrid Concurrency Control Schemes for a High Data Contention Environment, IEEE Transactions on Software Engineering 18/2, Feb 1992, pp 118-129.

YUDI93 Yu P. and Dias D., Performance Analysis of Concurrency Control Using Locking with Deferred Blocking, IEEE Transactions on Software Engineering 19/10, Oct 1993.

YUDL93 Yu P., Dias D. and Lavenberg S., On the Analytical Modelling of Database Concurrency Control, Journal of the ACM 40/4, Sep 1993, pp 831-872.

YWLS94 Yu P., Wu K., Lin K. and Son S., On Real-Time Databases: Concurrency Control and Scheduling, Proceedings of the IEEE 82/1, Jan 1994.