Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication

by
Michael J. Carey
Miron Livny

Computer Sciences Technical Report #758
March 1988

Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication

Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication

Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

Many concurrency control algorithms have been proposed for use in distributed database systems. Despite the large number of available algorithms, and the fact that distributed database systems are becoming a commercial reality, distributed concurrency control performance tradeoffs are still not well understood. In this paper we attempt to shed light on some of the important issues by studying the performance of four representative algorithms — distributed 2PL, wound-wait, basic timestamp ordering, and a distributed optimistic algorithm — using a detailed simulation model of a distributed DBMS. We examine the performance of these algorithms for various levels of contention, "distributedness" of the workload, and data replication. The results should prove useful to designers of future distributed database systems.

1. INTRODUCTION

For the past decade, distributed databases have attracted a great deal of attention in the database research community. Data distribution and replication offer opportunities for improving performance through parallel query execution and load balancing as well as increasing the availability of data. In fact, these opportunities have played a significant role in driving the design of the current generation of database machines [Tera83, Chei86, DeWi86, Fush86]. Distribution and replication are not a panacea, however; they aggravate the problems of concurrency control and crash recovery. In order to reap the potential performance benefits, the cost of maintaining data consistency must be kept at an acceptable level in spite of the added complexity of the environment. In the concurrency control area, this challenge has led to the development of a large number of concurrency control algorithm proposals. This paper addresses some of the important performance issues related to these algorithms.

Most distributed concurrency control algorithms fall into one of three basic classes: *locking* algorithms [Mena78, Rose78, Gray79, Lind79, Ston79], *timestamp* algorithms, [Thom79, Bern80b, Reed83], and *optimistic* (or certification) algorithms [Bada79, Schl81, Ceri82, Sinh85]. Bernstein, Goodman, and Hadzilacos survey many of

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

the proposed algorithms and describe how further algorithms may be synthesized by combining basic mechanisms from the locking and timestamp classes [Bern81, Bern87]. A number of recent papers in the concurrency control area have focused on the design of algorithms that provide improved availability (e.g., [Eage83, Bern84, Herl87, Jajo87]). El Abbadi and Toueg provide a framework for reasoning about such algorithms and survey some of the recent work in [ElAb88]. Lastly, approaches to extending concurrency control mechanisms to use various kinds of semantic information have also been of interest recently (e.g., [Garc83, Schw84, Weih85]).

Given the many proposed distributed concurrency control algorithms, a number of researchers have undertaken studies of their performance. For example, the behavior of various distributed locking algorithms has been investigated in [Garc79, Ries79a, Ries79b, Lin82, Oszu85], where algorithms with varying degrees of centralization of locking and deadlock detection have been studied and compared with one another. A qualitative study addressing performance issues for a number of distributed locking and timestamp algorithms was presented in [Bern80a]. The performance of locking was compared with that of basic timestamp ordering in [Gall82], with basic and multiversion timestamp ordering in [Lin83], and with an optimistic algorithm in [Kohl85] (where prototype implementations were measured). Several alternative schemes for handling or preventing deadlock in distributed locking algorithms were studied in [Balt82].

While the distributed concurrency control performance studies to date have been informative, a number of important questions remain unanswered. These include:

(1) How do the performance characteristics of the various basic algorithm classes compare under alternative assumptions about the nature of the database, the workload, and the computational environment?

(2) How does the distributed nature of transactions affect the behavior of the various classes of concurrency control algorithms?

(3) How much of a performance penalty must be incurred for synchronization and updates when data is replicated for availability or query performance reasons?

The first of these questions remains unanswered due to shortcomings of past studies that have examined multiple algorithm classes. The most comprehensive of these studies, [Lin83] and [Balt82], suffer from unrealistic modeling assumptions. In particular, contention for physical resources such as CPUs and disks was not captured in their models. Recent work has shown that neglecting to model resources can drastically change the conclusions reached [Agra87]. In [Gall82], the model of resource contention was artificial and the study assumed fully replicated data, extremely small transactions, and a very coarse concurrency control granularity. The performance results reported in [Kohl85] were obtained using a relatively lightly loaded two-site testbed system. The second question above remains open since a number of previous studies have modeled transactions as executing at a single site, making remote data access requests as needed (e.g., [Balt82, Gall82, Lin83]); few studies have carefully considered distributed transaction structures. Finally, the third question remains open since previous studies have commonly assumed either no replication (as in [Lin83, Balt82]) or full replication (as in [Gall82]), and their simplified models of transaction execution have often ignored important related overheads such as that of the commit protocol.

In this paper, we report on the first phase of a study aimed at addressing the questions raised above. The study employs a performance evaluation framework based on a fairly detailed model of a distributed DBMS. The design goal for the framework was to provide a facility for experimenting with and evaluating alternative transaction management algorithms on a common basis. The framework captures the main elements of a distributed database system: physical resources for storing and accessing the data, e.g., disks, CPUs, and communications channels; the distributed nature of transactions, including their access behavior and the coordination of their distributed execution; and the database itself, including the way that data is distributed and allocated to sites. The design of the performance framework was influenced heavily by previous results on the importance of realistic concurrency control modeling assumptions, especially with respect to system resources [Agra87]. Given the framework, we then proceed to examine the performance impact of varying the system load, the degree to which transactions are distributed, and the level of data replication on the performance of a representative set of distributed concurrency control algorithms. While we address only a subset of the open questions, we feel that our results constitute an important step towards understanding distributed concurrency control performance issues.

We examine four concurrency control algorithms in this study, including two locking algorithms, a timestamp algorithm, and an optimistic algorithm. The algorithms considered span a wide range of characteristics in terms of how conflicts are detected and resolved. Section 2 describes our choice of concurrency control algorithms. We use a simulator based on a closed queuing model of a distributed database system for our performance studies. The structure and characteristics of our model are described in Section 3. Section 4 presents our initial performance experiments and the associated results. Finally, Section 5 summarizes the main conclusions of this study and raises questions that we plan to address in the future.

2. DISTRIBUTED CONCURRENCY CONTROL ALGORITHMS

For this study we have chosen to examine four algorithms that we consider to be representative of the basic design space for distributed concurrency control mechanisms. We summarize the salient aspects of these four algorithms in this section. In order to do so, however, we must first explain the structure that we will assume for distributed transactions.

2.1. The Structure of Distributed Transactions

Figure 1 depicts a general distributed transaction in terms of the processes involved in its execution. Each transaction has a *master* process (M) that runs at its site of origination. The master process in turn sets up a collection of *cohort* processes (C_i) to perform the actual processing involved in running the transaction. Since virtually all query processing strategies for distributed database systems involve accessing data at the site(s) where it resides, rather than accessing it remotely, there is at least one such cohort for each site where data is accessed by the transaction. We will examine several query execution patterns; whether there is more than one cohort per site, and whether cohorts execute sequentially or in parallel, will depend on the query execution model of interest. We will clarify this point further in describing the workload model in Section 3. For the moment, simply note that similar transaction structures arise in R* [Lind84], Distributed INGRES [Ston79], and Gamma [DeWi86]. These systems differ, however, in the degree of parallelism involved in query execution.

In general, data may be replicated, in which case each cohort that updates any data items is assumed to have one or more *update* (U_{ij}) processes associated with it at other sites. In particular, a cohort will have an update pro-

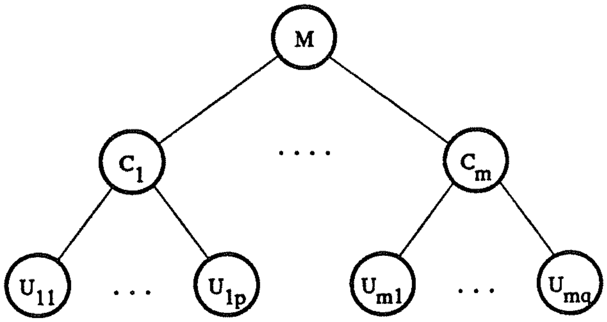

Figure 1: Distributed transaction structure.

cess at each remote site that stores a copy of the data items that it updates. It communicates with its update processes for concurrency control purposes, and it also sends each of them a copy of the relevant updates during the first phase of the commit protocol described below.

In this study, we will assume the use of a centralized two-phase commit protocol [Gray79, Lind79], with the master acting as the commit coordinator. Assuming no replication, things work as follows [Gray79]: When a cohort finishes executing its portion of a query, its sends an "execution complete" message to the master. When the master has received such a message from each cohort involved in the query, it will initiate the commit protocol by sending "prepare to commit" messages to all sites. Assuming that a cohort wishes to commit, it sends a "prepared" message back to the master, and the master will send "commit" messages to each cohort after receiving prepared messages from all cohorts. The protocol ends with the master receiving "committed" messages from each of the cohorts. If any cohort is unable to commit, it will return a "cannot commit" message instead of a "prepared" message in the first phase, causing the master to send "abort" instead of "commit" messages in the second phase of the protocol.

When update processes are present, the commit protocol becomes a nested two-phase commit protocol as described in [Gray79, Lind79]: Messages flow between the master and the cohorts, and the cohorts in turn interact with their updaters. That is, each cohort sends "prepare to commit" messages to its updaters after receiving such a message from the master, and it gathers the responses from its cohorts before sending a "prepared" message back to the master; phase two of the protocol is similarly modified. Copies of updated data items are carried in the "prepare to commit" messages sent from cohorts to updaters. Again, this is reminiscent of the "tree of processes" transaction structure of R* [Lind84].

Given this background on the assumed distributed transaction structure, we can now describe the four concurrency control algorithms that will be examined in the remainder of the paper.

2.2. Distributed Two-Phase Locking (2PL)

This algorithm is the distributed "read any, write all" two-phase locking algorithm described in [Gray79, Lind79]. Transactions set read locks on items that they read, and they convert their read locks to write locks on items that need to be updated. To read an item, it suffices to set a read lock on any copy of the item, so the local copy is locked; to update an item, write locks are required on all copies. Write locks are obtained as the transaction executes, with the transaction blocking on a write request until all of the copies of the item to be updated have been successfully locked. All locks are held until the transaction has successfully committed or aborted.

Deadlock is a possibility, of course, and we will handle it via a variant of the centralized detection (or "Snoop") scheme of Distributed INGRES [Ston79]. The scheme employed here is as follows: Local deadlocks are checked for any time a transaction blocks, and are resolved when necessary by restarting the transaction with the most recent initial startup time among those involved in the deadlock cycle. (A cohort is restarted by aborting it locally and sending an "abort" message to its master, which in turn notifies all of the processes involved in the transaction.) Global deadlock detection is handled by a "Snoop" process, which periodically requests waits-for information from all sites and then checks for and resolves any global deadlocks (using the same victim selection criteria as for local deadlocks). Unlike Distributed INGRES, we do not associate the "Snoop" responsibility with any particular site. Instead, each site takes a turn being the "Snoop" site and then hands this task over to the next site. The "Snoop" responsibility thus rotates among the sites in a round-robin fashion, ensuring that no one site will become a bottleneck due to global deadlock detection costs.

2.3. Wound-Wait (WW)

This is the distributed Wound-Wait locking algorithm of [Rose78], again with the "read any, write all" rule. It differs from 2PL in its handling of the deadlock problem: Rather than maintaining waits-for information and then checking for local and global deadlocks, deadlock is prevented via the use of timestamps. Each transaction is numbered according to its initial startup time, and younger transactions are prevented from making older ones wait. If an older transaction requests a lock, and if the request would lead to the older transaction waiting for a younger transaction, the younger transaction is "wounded" — it is restarted unless it is already in the second phase of its commit protocol (in which case the "wound" is not fatal, and is simply ignored). Younger transactions are permitted to wait for older transactions, however. The possibility of deadlocks is eliminated because any cycle of waiting transactions would have to include at least one instance where an older transaction is waiting for a younger one which is blocked as well, and this is impossible under the algorithm.

2.4. Basic Timestamp Ordering (BTO)

This is the basic timestamp ordering algorithm of [Bern80b, Bern81]. Like Wound-Wait, it uses transaction startup timestamps to prevent possible deadlocks, but it does so somewhat differently. Rather than using a locking approach, BTO associates timestamps with all recently accessed data items and requires that conflicting data accesses by transactions be performed in timestamp order. Transactions that attempt to perform out-of-order accesses are restarted. More specifically, each recently accessed data item has a read timestamp, which is the most recent timestamp among its readers, and a write timestamp, which is the timestamp of the most recent writer. When a read request is received for an item, it is permitted if the timestamp of the requester exceeds the item's write timestamp. When a write request is received, it is permitted if the requester's timestamp exceeds the read timestamp of the item; in the event that the timestamp of the requester is less than the write timestamp of the item, the update is simply ignored (by the Thomas write rule [Bern81]).

For replicated data, the "read any, write all" approach is again employed, so a read request may be sent to any copy while a write request must be sent to (and approved by) all copies. Integration of the algorithm with two-phase commit is accomplished as follows [Bern81]: Granted writes for a given data item are queued in timestamp order without blocking the writers until they are ready to commit, at which point their writes are dequeued and processed in order. Writers therefore keep their updates in a private workspace until commit time. Accepted read requests for such a pending write must be queued as well, blocking the readers, as readers cannot be permitted to proceed until the update becomes visible. Effectively, a write request locks out any subsequent read requests with later timestamps until the corresponding write actually takes place, which occurs when the updating transaction commits and its writes are dequeued and processed.

2.5. Distributed Certification (OPT)

This is the distributed, timestamp-based, optimistic concurrency control algorithm from [Sinh85][1], which operates by exchanging certification information during the commit protocol. For each data item, a read timestamp and a write timestamp are maintained. Transactions may read and update data items freely, storing any updates into a local workspace until commit time. For each read, the transaction must remember the version identifier (i.e., write timestamp) associated with the item when it was read. Then, when all of the transaction's cohorts have completed their work, and have reported back to the master, the transaction is assigned a globally unique timestamp. This timestamp is sent to each cohort in the "prepare to commit" message, and it is used to locally certify all of its reads

[1] Actually, two such algorithms are proposed in [Sinh85]. We chose their first algorithm for this study, as it is the simpler of the two algorithms.

and writes as follows: A read request is certified if (i) the version that was read is still the current version of the item, and (ii) no write with a newer timestamp has already been locally certified. A write request is certified if (i) no later reads have been certified and subsequently committed, and (ii) no later reads have been locally certified already. The term "later" refers to timestamp time here, so these conditions are checked using the timestamp given to the transaction when it started the commit protocol.

To handle replicated data, the scheme requires updaters to participate in the certification process. Updaters simply certify the set of writes that they receive at commit time, and again the necessary communication can be accomplished by passing information in the messages of the commit protocol. Failure of the certification test by any cohort or updater is handled in OPT by having that process send a "cannot commit" reply in response to the "prepare to commit" message, causing the transaction to be restarted.

2.6. Some Observations

The four protocols that we have selected span the three major algorithm classes, and they represent a fairly wide range of conflict detection and resolution methods and times. 2PL prevents conflicts as they occur using locking, resolving global deadlocks using a centralized deadlock detection scheme. WW is similar, except that it prevents deadlocks using timestamps and restarts rather than checking for deadlocks and incurring the associated message costs. BTO uses timestamps to order transactions a priori, restarting transactions when conflicting, out-of-order accesses occur; read requests must occasionally block when they request data from pending, uncommitted updates. Finally, OPT always uses restarts to handle conflicts, checking for problems only when a transaction is ready to commit. 2PL, WW, and BTO all involve sending write access requests between a cohort and its updaters when a write request is received at the cohort site; in contrast, OPT defers communication between cohorts and updaters until commit time, piggybacking its concurrency control information on the messages of the commit protocol.

3. MODELING A DISTRIBUTED DBMS

As mentioned in Section 1, we have developed a single, uniform, distributed DBMS model for studying a variety of concurrency control algorithms and performance tradeoffs. Figure 2 shows the general structure of the model. Each site in the model has four components: a *source*, which generates transactions and also maintains transaction-level performance information for the site, a *transaction manager*, which models the execution behavior

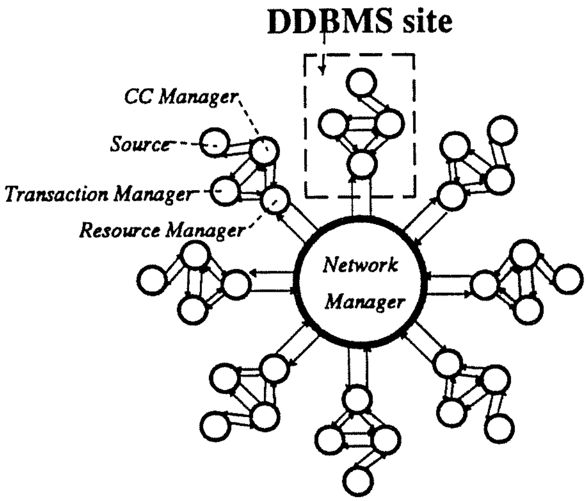

Figure 2: Distributed DBMS Model Structure.

of transactions, a *concurrency control manager*, which implements the details of a particular concurrency control algorithm; and a *resource manager*, which models the CPU and I/O resources of the site. In addition to these per-site components, the model also has a *network manager*, which models the behavior of the communications network. Figure 3 presents a slightly more detailed view of these components and their key interactions. The component interfaces were designed to support modularity, making it easy to replace one component (e.g., the concurrency control manager) without affecting the others. We describe each component in turn in this section, preceded by a discussion of how the database itself is modeled.

3.1. The Database Model

We model a distributed database as a collection of *files*. A file can be used to represent a relation, or it can represent a partition of a relation in a system where relations are partitioned across multiple sites (as in Gamma [DeWi86]). Files are assumed to be the unit of replication, so a given file can have from one copy (no replication) up to as many copies as there are sites in the system (full replication). Table 1 summarizes the parameters of the database model, which include the number of sites and files in the database and the sizes of the files. As indicated in the table, files are modeled at the page level. The mapping of files to sites is specified via the parameter *FileLocations*, a boolean array in which $FileLocations_{ij}$ is true if a copy of file i resides at site j.

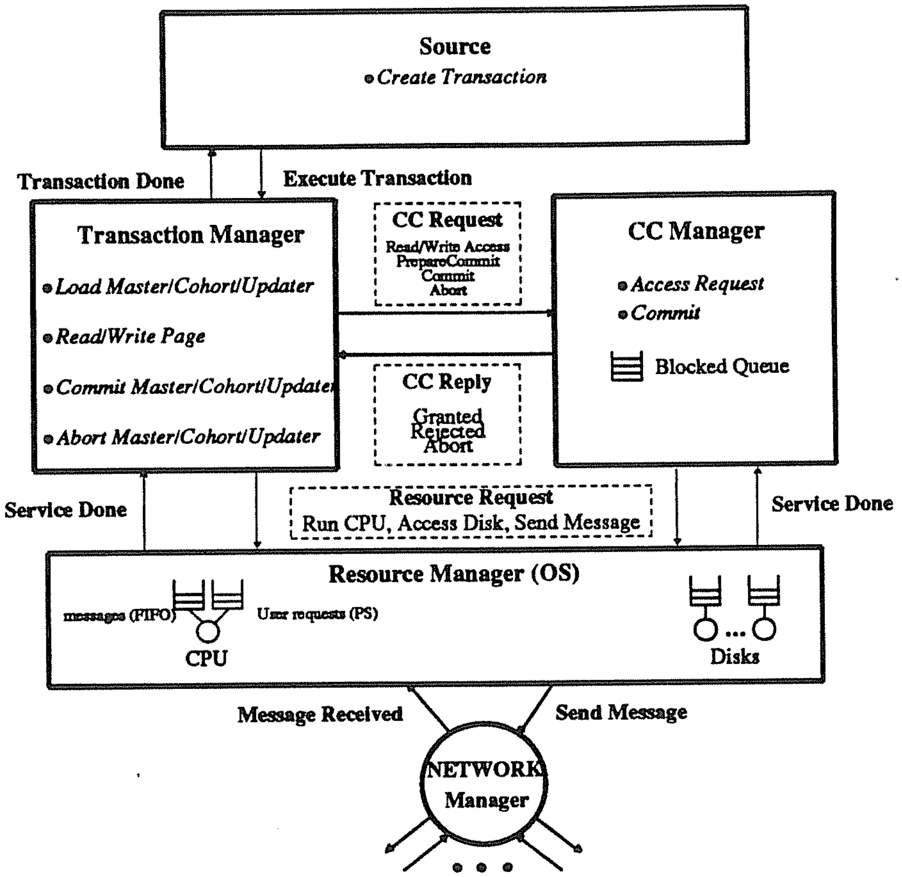

Figure 3: A Closer Look at the Model.

Parameter	Meaning
NumSites	Number of sites in the database
NumFiles	Number of files in the database
FileSize$_i$	Number of pages in file *i*
FileLocations$_{ij}$	Placement of files at sites

Table 1: Database Model Parameters.

3.2. The Source

The source is the component responsible for generating the workload for a site. The workload model used by the source characterizes transactions in terms of the files that they access and the number of pages that they access and update in each file. Table 2 summarizes the key parameters of the workload model for a site; each site has its own set of values for these parameters. The *NumTerminals* parameter specifies the number of terminals per site, and the *ThinkTime* parameter is the mean of an exponentially distributed thinking period between the completion of one transaction and the submission of the next one at a terminal. *NumClasses* specifies the number of transaction classes for the site.

The *ClassProb* parameter specifies the probability that a newly generated transaction will be of a given class. The remaining per-class parameters characterize transactions of the class as follows: *ExecPattern* specifies the execution pattern, either sequential or parallel, for transactions. (More will be said on this when the transaction model is described.) *FileCount* is the number of files accessed, and $FileProb_i$ gives the probability distribution (or relative file weights) for choosing the actual files that the transaction will access. The next two parameters determine the file-dependent access characteristics for transactions of the class, including the average number of pages read and the probability that an accessed page will be updated. The last parameter specifies the average amount of CPU time required for transactions of the class to process a page of data when reading or writing it. The actual number of pages accessed ranges uniformly between half and twice the average, and the page CPU time is exponentially distributed.

3.3. The Transaction Manager

Each transaction in the workload has the general structure described in Section 2.1, with a master process, a number of cohorts, and possibly a number of updaters. As described earlier, the master resides at the site where the transaction was submitted. Each cohort makes a sequence of read and write requests to one or more files that are stored at its site; a transaction has one cohort at each site where it needs to access data. Cohorts communicate with

Parameter	Meaning
	Per-Site Parameters
NumTerminals	Number of terminals per site
ThinkTime	Think time for the terminals
NumClasses	Number of transaction classes
	Per-Class Parameters
ClassProb	Probability of this class
ExecPattern	Sequential or parallel execution
FileCount	Number of files accessed
$FileProb_i$	Access probability for file *i*
$NumPages_i$	Average number of file *i* pages read
$WriteProb_i$	Write probability for file *i* pages
PageCPU	CPU time for processing a page of data

Table 2: Workload Model Parameters for a Site.

their updaters when remote write access permission is needed, and the updaters then make the required write requests for local copies of the data on behalf of their cohorts. A transaction can execute in either a sequential or parallel fashion, depending on the execution pattern of the transaction class. Cohorts in a sequential transaction execute one after another[2], whereas cohorts in a parallel transaction are started together and execute independently until commit time. A sequential transaction might be thought of as representing a series of steps in a relational query. A parallel transaction might be thought of as modeling the kind of parallel execution that is seen in systems like Gamma [DeWi86] or the Teradata database machine [Tera83].

The transaction manager is responsible for accepting transactions from the source and modeling their execution. To choose the execution sites for a transaction's cohorts, the decision rule is: If a file is present at the originating site, use the copy there; otherwise, choose uniformly from among the sites that have remote copies of the file. If the file is replicated, the transaction manager will initiate updaters at sites of other copies when the cohort accessing the file first needs to interact with them for concurrency control reasons. The transaction manager also models the details of the transaction commit and abort protocols.

To understand how transaction execution is modeled, let us follow a typical transaction from beginning to end. When a transaction is initiated, the set of files and data items that it will access are chosen by the source. The master is then loaded at the originating site, and it sends "load cohort" messages to initiate each of its cohorts. Each cohort makes a series of read and write accesses. A read access involves a concurrency control request to get access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. Write requests are the same except for the disk I/O; the I/O activity for writes takes places asynchronously after the transaction has committed.[3] A concurrency control request for a read or write access is always granted in the case of the OPT algorithm, but this is not the case for the other algorithms. When a concurrency control request cannot be granted immediately, due to conflicts or remote write requests, the cohort will wait until the request is granted by the concurrency control manager. If the cohort must be restarted, the concurrency control manager notifies the transaction manager, which then invokes the abort protocol. Once the transaction manager has finished aborting the transaction, it delays the master for a period of time before letting it attempt to rerun the transaction; as in [Agra87], we

[2] In this paper, sequential transactions will have just one cohort, a cohort that accesses a collection of files residing at a single site. The model is capable of handling the more general case, however.

[3] We assume sufficient buffer space to allow the retention of updates until commit time, and we also assume the use of a log-based recovery scheme where only log pages must be forced prior to commit. We do not model logging, as we assume it is not the bottleneck.

use one average transaction response time (as observed at the master site in this case) for the length of this period.

3.4. The Resource Manager

The resource manager can be viewed as a model of the operating system for a site; it manages the physical resources of the site, including its CPU and its disks. The resource manager provides CPU and I/O service to the transaction manager and concurrency control manager, and it also provides message-sending services (which involve using the CPU resource). The transaction manager uses CPU and I/O resources for reading and writing disk pages, and it also sends messages. The concurrency control manager uses the CPU resource for processing requests, and it too sends messages.

The parameters of the resource manager are summarized in Table 3. Each site has *NumDisks* disks plus one CPU. The CPU service discipline is first-come, first-served (FIFO) for message service and processor sharing for all other services, with message processing being higher priority. Each of the disks has its own queue, which it serves in a FIFO manner; the resource manager assigns a disk to serve a new request randomly, with all disks being equally probable, so our I/O model assumes that the files stored at a site are evenly balanced across the disks. Disk access times for the disks are uniform over the range [*MinDiskTime* , *MaxDiskTime*]. Disk writes are given priority over disk reads (to ensure that the system keeps up with the demand for asynchronously writing updated pages back to disk after the updater has committed). The parameter *InitWriteCPU* models the CPU overhead associated with initiating a disk write for an updated page. Finally, *MsgCPUTime* captures the cost of protocol processing for sending or receiving a message.

Parameter	**Meaning**
NumDisks	Number of disks per site
MinDiskTime	Minimum disk access time
MaxDiskTime	Maximum disk access time
InitWriteCPU	Time to initiate a disk write
MsgCPUTime	Message send or receive time

Table 3: Resource Manager Parameters.

3.5. The Network Manager

The network manager encapsulates the model of the communications network. Our network model is currently quite simplistic, acting just as a switch for routing messages from site to site. This is because our experiments assume a local area network, where the actual time on the wire for messages is neglible, although we do take the CPU overhead for message processing into account at both the sending and receiving sites. This cost assumption has become fairly common in the analysis of locally distributed systems, as it has been found to provide reasonably accurate performance results despite its simplicity [Lazo86]. Of course, given that the characteristics of the network are isolated in this module, it would be a simple matter to replace our current model with a more sophisticated one in the future.

3.6. The Concurrency Control Manager

The concurrency control manager captures the semantics of a given concurrency control algorithm, and is the only module that must be changed from algorithm to algorithm. As was illustrated in Figure 3, it is responsible for handling concurrency control requests made by the transaction manager, including read and write access requests, requests to get permission to commit a transaction, and several types of of master and cohort management requests to initialize and terminate master and cohort processes. We have implemented a total of five concurrency control managers, including four for the concurrency control algorithms described in Section 2 and one that we will refer to as NONE. NONE has a message-passing structure identical to the locking and timestamp algorithms, but it grants all requests; it will provide useful performance bound information for the other algorithms, as will be seen shortly.

The concurrency control manager has a variable number of parameters. One parameter, *CCReqCPU*, specifies the amount of CPU time required to process a read or write access request; this parameter is present for all of our algorithms. Each algorithm then has zero or more additional parameters. Of the algorithms studied in this paper, only 2PL uses another parameter. Its second parameter is *DetectionInterval*, which determines the amount of time that a site should wait, after becoming the next "Snoop" site, before gathering global waits-for information and performing global deadlock detection.

4. EXPERIMENTS AND RESULTS

In this section, we present our initial performance results for the four concurrency control algorithms of Section 2 under various assumptions about data replication, CPU cost for sending and receiving messages, transaction locality, and sequential versus parallel execution. The simulator used to obtain these results was written in the DeNet simulation language [Livn88], which allowed us to preserve the modular structure of our model when implementing it. We describe the performance experiments and results following a discussion of the performance metrics of interest and the parameter settings used.

4.1. Metrics and Parameter Settings

The primary performance metric employed in this paper is the throughput (transaction completion rate) of the system.[4] Several additional metrics are used to aid in the analysis of the experimental results. One is the *restart ratio*, giving the average number of times that a transaction has to restart per commit, computed by dividing the number of transaction restarts by the number of commits. We also examine the *message ratio*, computed similarly, which gives the average number of messages per commit. Finally, in our last experiment, we employ ratios of response times to illustrate the costs and benefits of parallel execution. The response time is computed there as the completion time of the transaction's master process minus the time when it was initiated at a terminal.

Table 4 gives the values of the key simulation parameters in our experiments. We consider a database which is distributed over 8 sites. The database contains 24 files, organized into 8 groups of 3 files, where each file contains 800 pages of data. There are 50 terminals per site, and the mean terminal think time is varied from 0 to 5 seconds in each experiment in order to vary system load. In terms of the workload, a transaction accesses 3 files; it reads an average of 6 pages of each file and updates each page with a probability of 1/4. Thus, each transaction involves an average of 18 reads and 4.5 writes. This transaction size was chosen as being relatively small, as transactions tend to be in transaction processing environments, but not so small as to be unrealistic [Lind87]. The corresponding file sizes were selected so as to provide an interesting level of data contention. Finally, it takes transactions an average of 8 milliseconds of CPU time to process each page read or written. More information regarding transaction classes and data placement will be provided in the description of each experiment.

Continuing through the parameters in Table 4, each site has two disks, and each disk has an average access time of 20 milliseconds. Initiating a disk write for an updated page takes 2 milliseconds of CPU time, and the mean CPU time for message protocol processing on each end is varied from 1 to 10 milliseconds. The concurrency con-

[4] Since we are using a closed queueing model, the inverse relationship between throughput and response time makes either a sufficient performance metric.

Parameter	Setting
NumSites	8 sites
NumFiles	24 files (8 groups of 3)
$\mathit{FileSize}_i$	800 pages per file
NumTerminals	50 terminals per site
ThinkTime	0-5 seconds
FileCount	3 files (1 group)
$\mathit{FileProb}_i$	1/3 for each of 3 files
$\mathit{NumPages}_i$	6 pages per file
$\mathit{WriteProb}_i$	1/4
PageCPU	8 milliseconds
NumDisks	2 disks per site
MinDiskTime	10 milliseconds
MaxDiskTime	30 milliseconds
InitWriteCPU	2 milliseconds
MsgCPUTime	1, 4, and 10 milliseconds
CCReqCPU	negligible (0)
DetectionInterval	1 second

Table 4: Simulation Parameter Settings.

trol CPU overhead is assumed to be negligible, for all algorithms, compared to the 8 millisecond CPU time for page processing. Finally, the interval between the end of one global deadlock check and the beginning of the next one is 1 second in 2PL.

The I/O and CPU cost parameter values for the experiments reported here were chosen so that, messages aside, the system will operate in an I/O-bound region. In particular, when the disks are fully utilized, only about 80% of the CPU capacity of the system is utilized. However, since the workload is not heavily I/O-bound, we will see that it is possible for message-related CPU costs to shift the system into a region of CPU-bound operation. Such a shift changes the performance profile of the system. We have run a number of experiments with a larger page CPU time as well, where the system is CPU-bound regardless of communication activity. Space limitations prevent us from including those results in detail, but we will comment on them throughout this section. Lastly, our workload consists only of update-oriented transactions. While we recognize that replication can lead to performance advantages for read-intensive workloads by reducing dependence on remote data and providing an opportunity for load balancing [Care85, Care86a], we wish to focus our attention here on the cost issues related to concurrency control.

4.2. Experiment 1: Algorithms and Replication

The purpose of this experiment is to investigate the performance of the four algorithms as the system load varies, and to see how performance is impacted by data replication. In this experiment, each group of three files is placed at a site as follows: There are eight sites, $S_i, 1 \le i \le 8$, and eight groups of files, $G_i, 1 \le i \le 8$. In the one copy case, the three files F_{i1}, F_{i2}, F_{i3} comprising group G_i are stored at site S_i. When we consider two copies of each file, the files in group G_i are stored both at site S_i and site $S_{(i \bmod 8)+1}$. In the three copy case, an additional copy is stored at site $S_{(i \bmod 8)+2}$. Transactions execute sequentially in this experiment. Furthermore, they execute locally: Transactions originating at site S_i access the files in group G_i, not needing to touch remote data except to update other copies. Thus, the one copy case examined here is basically a centralized concurrency control situation, except that global deadlock checking is taking place in 2PL. In the case of replicated data, the distributed nature of the system is used only to improve availability. We assume efficient communications software, using a value of 1 millisecond for *MsgCPUTime* in this experiment.

Figure 4 presents the transaction throughput results for the one copy case. Since think time is used to vary the load, the system becomes more heavily loaded going from right to left[5] along the curves. As expected, the results indicate that the throughput for each algorithm initially increases as the system load is increased, and then it decreases. The increase is due to the fact that better performance is obtained when a site's CPU and disks are utilized in parallel; throughput then degrades for all four of the concurrency control algorithms due to transaction restarts caused by data contention. These trends are natural for a centralized DBMS [Care84, Care86b, Agra87]. The NONE curve indicates how the system would perform if no concurrency control conflicts were to occur, increasing at first and then leveling off without degrading due to restarts. Among the concurrency control algorithms studied, Figure 4 indicates that 2PL provides the best performance, followed by BTO and WW (which are virtually indistinguishable), followed by OPT.

To understand the relative throughput ordering of the algorithms, Figure 5 presents their restart ratios. The results are easily explained based on these results. 2PL has the lowest restart ratio by far, and consequently performs the best. BTO and WW have higher restart ratios, providing the next best throughput results. OPT has the highest restart ratio, and thus has the lowest throughput among the algorithms. Since OPT restarts transactions at

[5] Note that load increases in the *opposite direction* here than if the number of terminals or multiprogramming level was being varied. The most heavily loaded operating region is where the think time is zero.

2PL □——□ WW ○- - - -○ BTO +- - - -+ OPT ×——× NONE △--------△

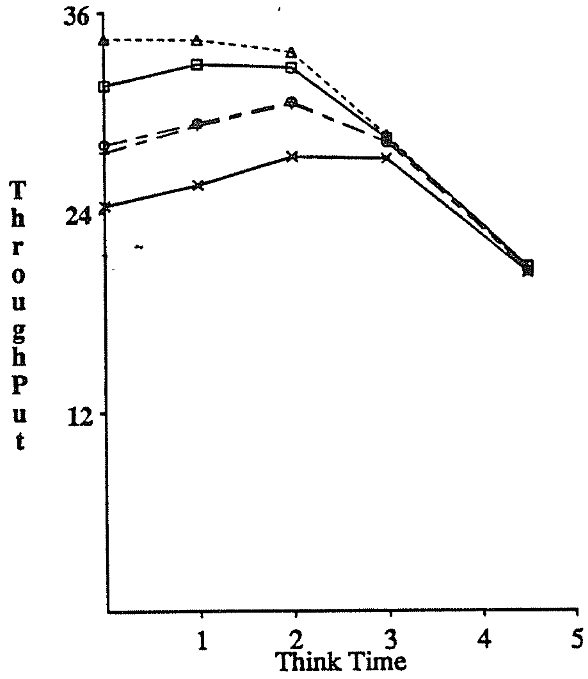

Figure 4: Throughput, 1 Copy.
(100% Local, MsgCPUTime = 1 ms)

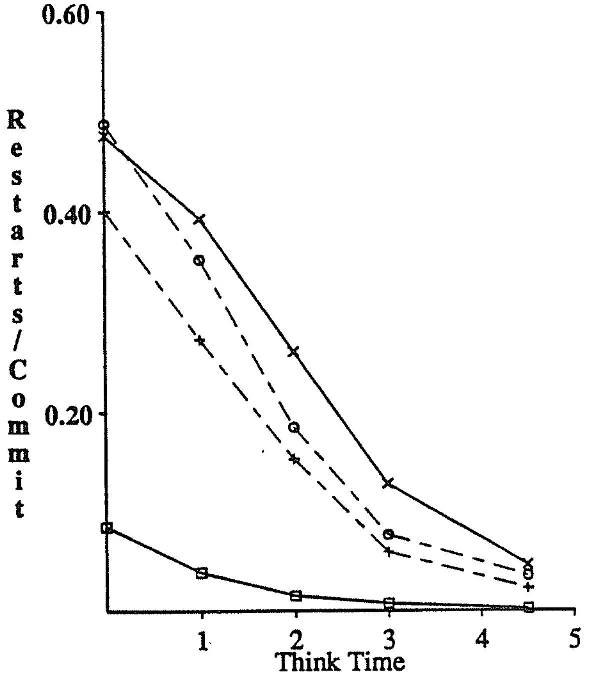

Figure 5: Restarts per Commit, 1 Copy.
(100% Local, MsgCPUTime = 1 ms)

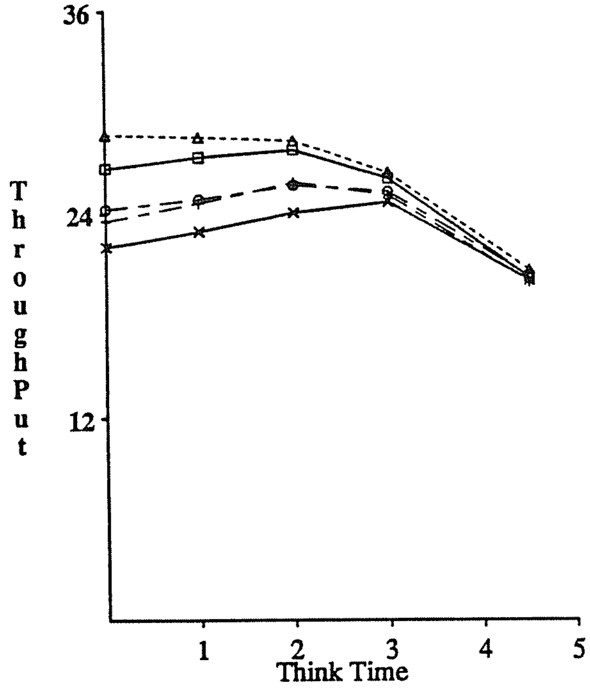

Figure 6: Throughput, 2 Copies.
(100% Local, MsgCPUTime = 1 ms)

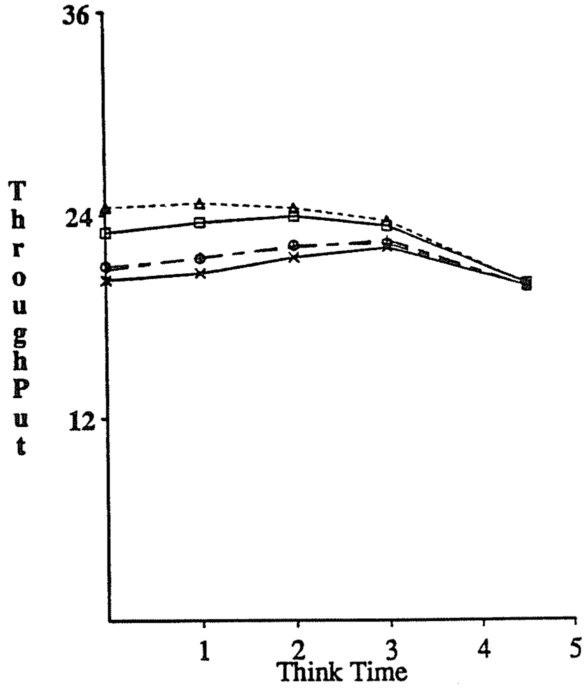

Figure 7: Throughput, 3 Copies.
(100% Local, MsgCPUTime = 1 ms)

commit time, rather than earlier as in BTO and WW, it does not take a very big difference in the restart ratios to cause the significantly lower throughput seen under high loads for OPT. The reason that BTO and WW perform alike despite having different restart ratios is similar — while WW has a higher ratio of restarts to commits, it always selects a younger transaction to restart, making its individual restarts less costly than those of BTO. These results indicate the importance of restart ratios as a performance determinant.

Figures 6 and 7 present the throughput results for the two and three copy cases, respectively. Increasing the number of copies increases both the amount of I/O involved in updating the database and the level of synchronization-related message traffic required in all but the OPT algorithm. As a result, three trends are evident in the figures: First, increasing replication leads to decreased performance due to the additional update work. This is particularly significant given the I/O-bound nature of our workload, as increasing the number of copies strains the bottleneck resource. Due to the low message CPU time here, the system remains I/O-bound even in the three copy case. Second, the differences between algorithms decrease as the level of replication is increased. The explanation for this is again restart-related: Successfully completing a transaction in the presence of replication involves all the work of the one copy case, plus the additional work of updating remote copies of data. Since remote updates occur only after a successful commit, the relative cost of a restart decreases as the number of copies increases. This is because the amount of effort wasted becomes a smaller fraction of the transaction's total required effort. Third, the performance of OPT suffers a bit less than that of the other algorithms. This is due to the fact that the presence of copies implies inter-site concurrency control messages for each write in 2PL, WW, and BTO, whereas no additional messages are needed in OPT. This last point will become much more evident in Experiment 2.

Our CPU-bound versions of these experiments produced the same relative ordering of the algorithms, but the performance differences and trends were somewhat different. The separation between the algorithms was greater in the CPU-bound version of the one copy case, as CPU is a more critical resource than I/O — that is, one CPU can be a more stringent bottleneck than two disks. Thus, the performance impact of restarts was greater here. In addition, in the two and three copy cases, the differences between algorithm performance did not shrink to the same extent. This is due to the fact that end-of-transaction updates have less impact on CPU than on I/O, and CPU was the bottleneck resource. OPT again suffered the least due to replication, an effect more evident here since additional messages imply additional CPU cost.

4.3. Experiment 2: Message Cost Considerations

This experiment examines the impact of message cost on the performance of the algorithms. The data layout, workload, and transaction execution pattern used here are identical to those of Experiment 1. However, instead of using a value of 1 millisecond for *MsgCPUTime*, we use values of 4 milliseconds and 10 milliseconds in this experiment. We remind the reader that this parameter determines the CPU time to send or receive a message, meaning that the 4 and 10 millisecond values place 8 or 20 millisecond lower bounds on message transfers; the latter time represents the upper end of the message cost spectrum. We do not present one copy results here, as the increased message overhead only affects performance when remote updates are involved (since transactions execute at their site of origin).

Figure 8 presents the throughput results obtained by repeating the two copy case from Experiment 1 with *MsgCPUTime* = 4 milliseconds. The only messages involved in the one copy case occur for global deadlock detection in 2PL, so the results that we obtained in the one copy case were really no different than those of Figure 4. However, the results in Figure 8 are quite different than those of Figure 6. The performance of each of the algorithms is worse in Figure 8 because of the additional message cost. However, OPT suffers the least from the additional cost due to its use of commit protocol messages; thus, we find that OPT actually does a bit better than WW and BTO here, and the difference between OPT and 2PL is less dramatic. Figure 9 shows the average number of messages per completed transaction, making it clear that OPT requires significantly fewer messages. Looking deeper, when we examined the resource utilization levels in this case, we discovered that 2PL, BTO, WW, and NONE all become CPU-bound here due to the CPU cost associated with their message activity; OPT, on the other hand, remains I/O-bound. Thus, not only does OPT require fewer messages, but messages have a lesser performance impact for OPT since CPU is not the performance bottleneck.

Figures 10 and 11 present the corresponding results for this message cost in the three copy case. The trends that began in Figures 8 and 9 are more pronounced here. Again, all algorithms suffer performance-wise as compared with Experiment 1 due to the increased message cost, and again OPT suffers the least. In fact, because of the large number of messages required by 2PL, BTO, and WW to interact with remote copy sites, OPT outperforms the other algorithms here. This is especially clear since OPT outperforms NONE in this case — recall that NONE communicates with copy sites on each write access, like DD, WW, and BTO, but with write permission always being granted. The implication is that the performance loss in OPT due to end-of-transaction restarts is more than

2PL □——□ WW ○- - - -○ BTO +- - - -+ OPT ×——× NONE △---------△

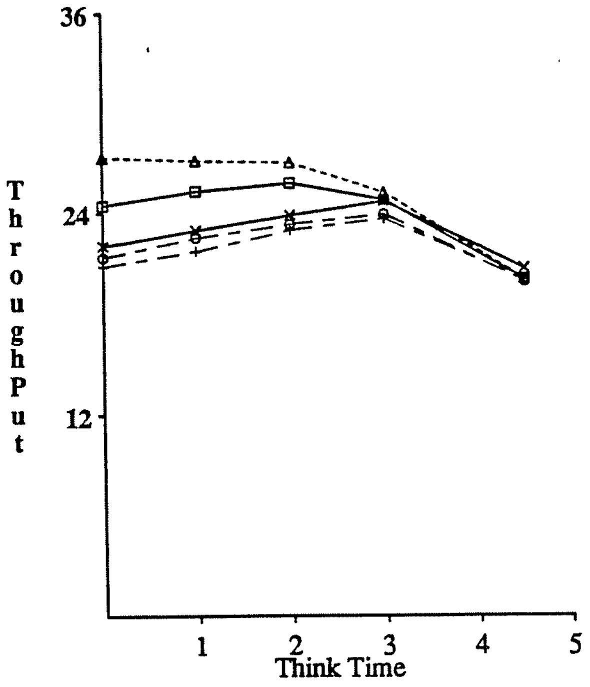

Figure 8: Throughput, 2 Copies.
(100% Local, MsgCPUTime = 4 ms)

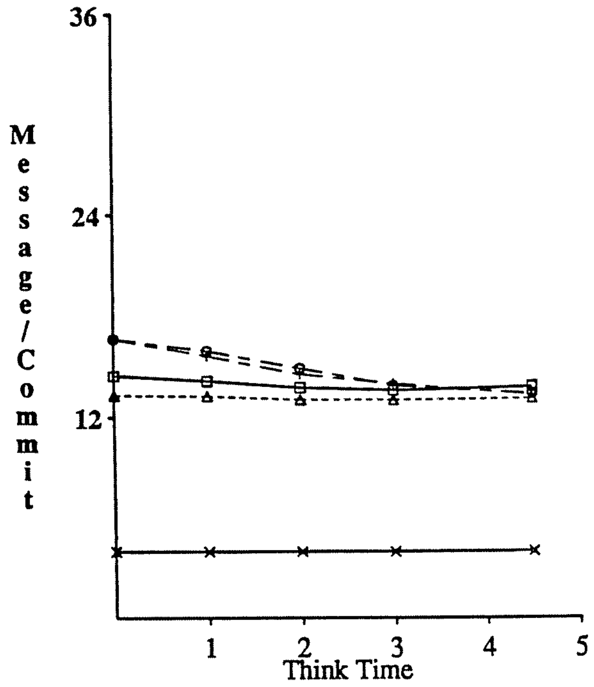

Figure 9: Messages per Commit, 2 Copies.
(100% Local, MsgCPUTime = 4 ms)

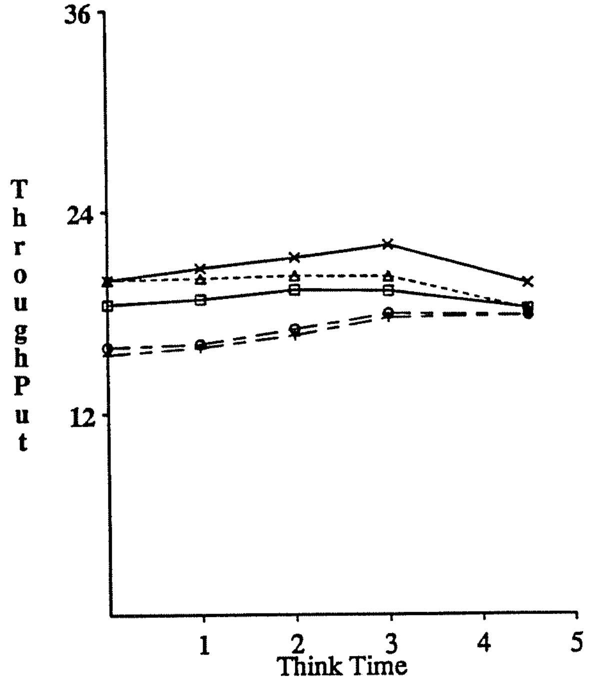

Figure 10: Throughput, 3 Copies.
(100% Local, MsgCPUTime = 4 ms)

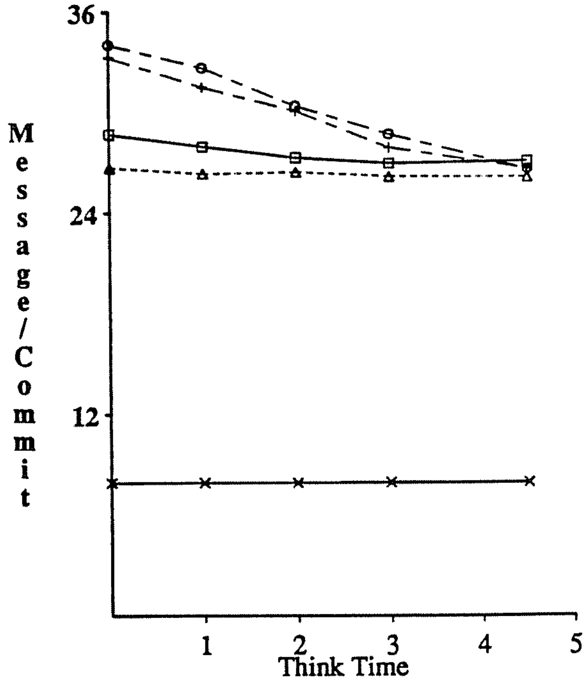

Figure 11: Messages per Commit, 3 Copies.
(100% Local, MsgCPUTime = 4 ms)

compensated for by the message savings in this case, where messages are moderately expensive and three copies of each data file exist. This is aided by the fact that restarts become less serious for OPT in this case: Since remote updates are only performed after a successful commit, these are not done (and thus not redone) when OPT restarts a transaction. However, all of the CPU-related message activity required to obtain remote access permission in the other algorithms must be redone in the event of an abort, and these algorithms are CPU-bound due to the high message CPU cost. These additional messages are visible, especially for BTO and WW, at the low think times in Figures 9 and 11.

Figures 12 and 13 present the throughput results for the two and three copy cases with a *MsgCPUTime* of 10 milliseconds. The message ratio results are not affected by the message cost, so we refer the reader to Figures 9 and 11 for this data. The shift in results is similar to that observed in previous curves, except that they are heavily amplified here due to the even higher cost associated with message processing. While all algorithms suffer some performance loss due to message overhead, OPT suffers the least by far. OPT outperforms the other algorithms significantly here in both the two and three copy cases due to its minimal communication requirements.

4.4. Experiment 3: Nonlocal Data Access

This experiment considers a situation where a transaction may access non-local data. The data layout and transaction execution pattern used here are the same as in Experiments 1 and 2, and all of the files needed by a given transaction still reside on a single site. Here, however, we set the workload parameters in such a way that, in the one copy case, a given transaction has a 70% chance of using local data and a 30% chance of needing to use non-local data instead.[6] In the latter case, the file group accessed by the transaction is chosen randomly from among the seven remote groups, with each being equally likely. We consider only the 4 millisecond *MsgCPUTime* setting here, and examine both the two and three copy cases. While we also reran the 1 millisecond *MsgCPUTime* experiments with this nonlocal data access pattern, the results were virtually identical to the purely local case; the message cost associated with remote execution added very little to the overall transaction path length in this case.

Figures 14 and 15 present the throughput results for the two copy case and the three copy case, respectively. Comparing these curves to the strictly local execution cases of Figures 8 and 10, we find the results to be similar

[6] The probability of a given transaction requiring non-local access drops somewhat with replication, as one or two file groups that were non-local in the one copy case will now be replicated at this site as well.

2PL □——□ WW ○- - - -○ BTO +- - - -+ OPT ×——× NONE △--------△

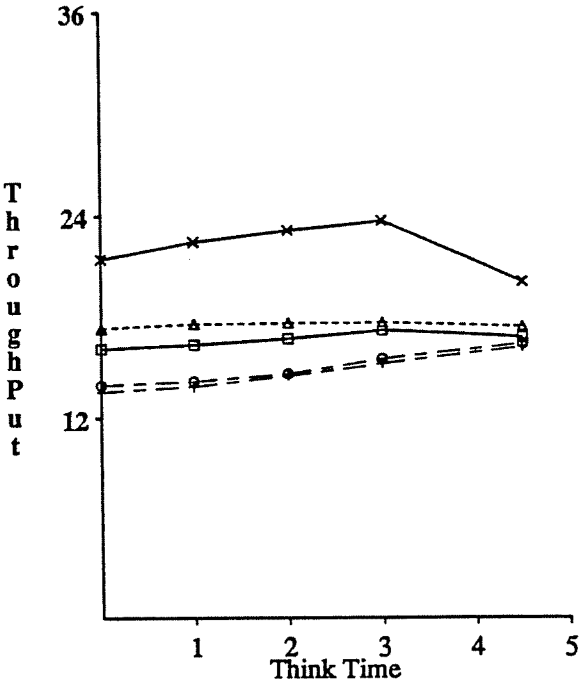

Figure 12: Throughput, 2 Copies.
(100% Local, MsgCPUTime = 10 ms)

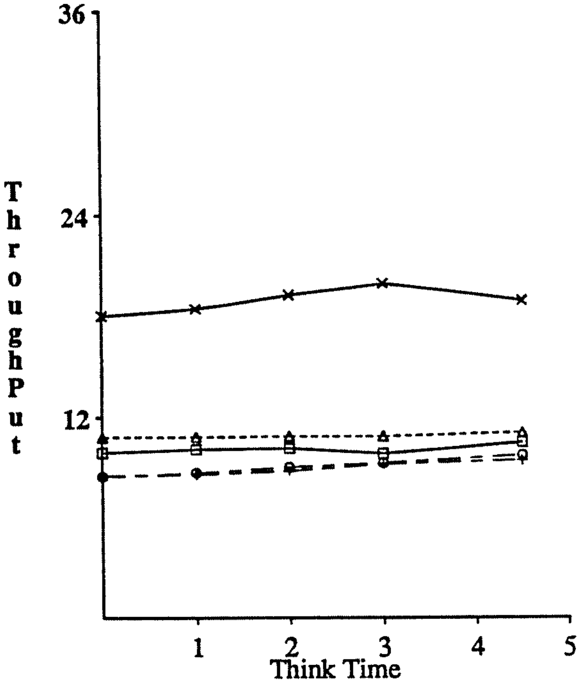

Figure 13: Throughput, 3 Copies.
(100% Local, MsgCPUTime = 10 ms)

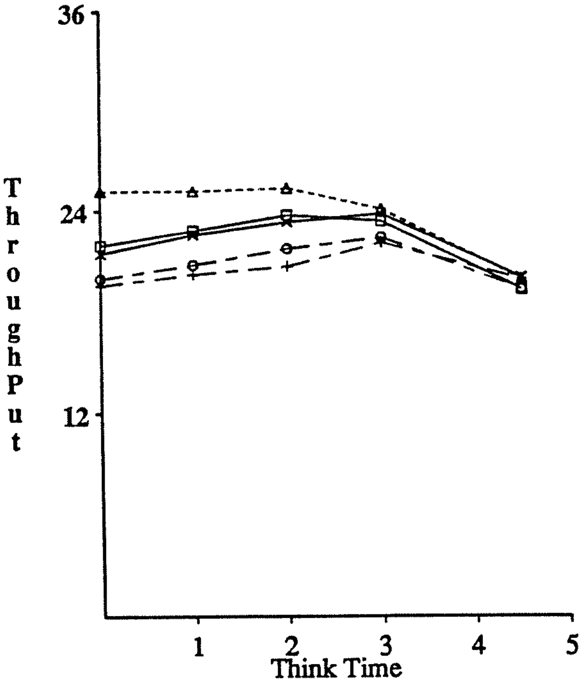

Figure 14: Throughput, 2 Copies.
(70% Local, MsgCPUTime = 4 ms)

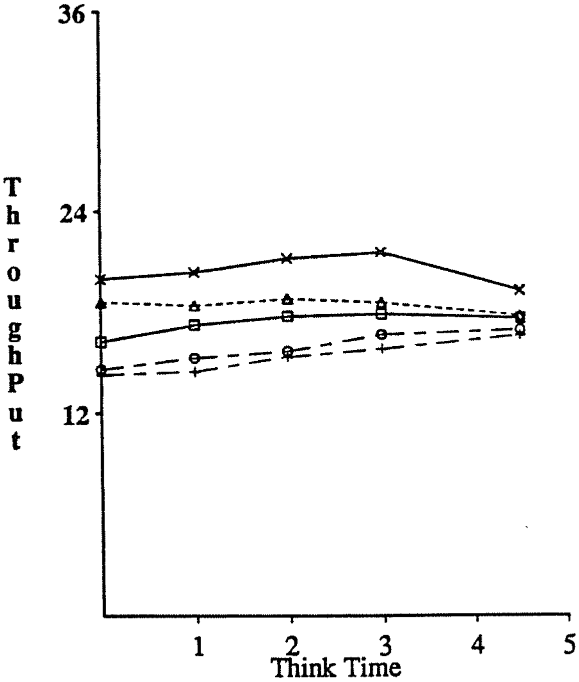

Figure 15: Throughput, 3 Copies.
(70% Local, MsgCPUTime = 4 ms)

except in the relationship of OPT to the other algorithms in Figure 14. In Figure 8, OPT performs a bit better than WW and BTO, but it is still noticeably worse than 2PL in its performance. In Figure 14, however, OPT and 2PL actually perform comparably. This is because, with a 4 millisecond message cost and two copies of data, all algorithms except OPT end up being CPU-bound in this case; their performance thus worsens as a result of the additional messages associated with remote cohort execution. The relative performance of 2PL compared to NONE is also a bit worse here, as additional message overhead causes transactions to hold locks somewhat longer. As before, OPT exhibits the best performance in Figure 15. Synchronizing three copies of data is very expensive for the other algorithms with this message cost.

4.5. Experiment 4: Parallel Execution

The purpose of this experiment is to investigate performance under a parallel transaction execution pattern. In this case, the data layout is different, and a bit more complex. We consider only the one copy case here: The three files F_{i1}, F_{i2}, F_{i3} comprising group G_i are stored at sites S_i, $S_{(i \bmod 8)+1}$, and $S_{(i \bmod 8)+2}$, respectively. The workload is arranged so that transactions originating at site S_i access group G_i, as always, but in this case the implication is that the transaction will have three cohorts, two of which will execute at remote sites. These three cohorts will execute in parallel with one another. We examine results for the one copy case for the 1 and 4 millisecond *MsgCPUTime* settings here.

Figure 16 presents the throughput results for the 1 millisecond case, and Figure 17 compares the performance results for parallel execution with those obtained for serial execution in Experiment 1. Specifically, Figure 17 shows the percentage improvement in response time[7] in the parallel case as compared to the serial case. As is evident in the figure, parallelism provides significant performance gains — of more than 50% — when the system is lightly loaded. At high loads, however, parallel execution actually leads to a slight degradation in response time. Figures 18 and 19 present the same information for the case where *MsgCPUTime* = 4 milliseconds. Given this four-fold message cost increase, parallelism is seen to be significantly less attractive. The gains under light loads have decreased quite a bit, and the performance penalty associated with parallel execution versus serial execution at high loads has increased dramatically. For both message costs, the performance degradation can be attributed in part to the additional overhead involved in initiating several cohorts and coordinating their commit. However, this

[7] Response time is the interesting metric here because parallelism is employed in database machines to improve query response times.

2PL □——□ WW ○– – –○ BTO +– – –+ OPT ×——× NONE △·········△

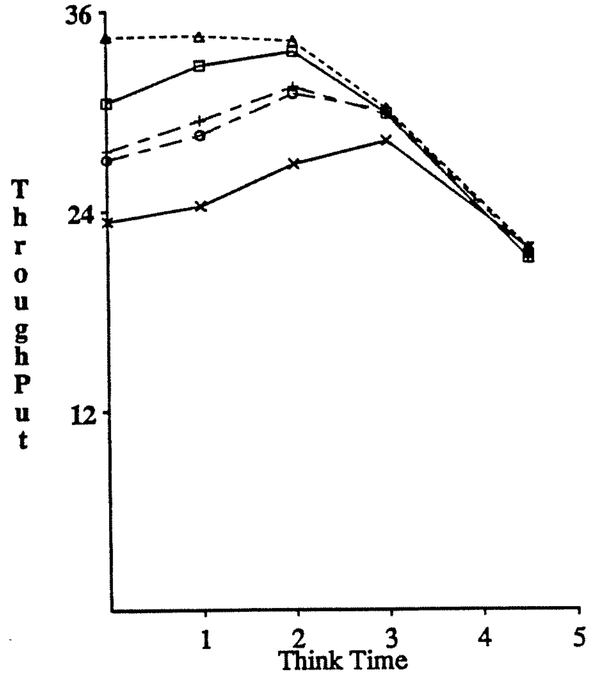

Figure 16: Throughput, 1 Copy (Distributed).
(Parallel Execution, MsgCPUTime = 1 ms)

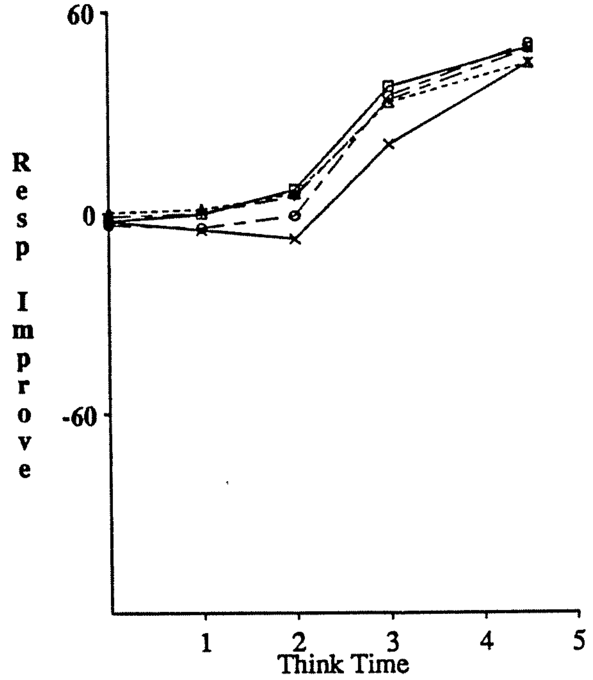

Figure 17: Improvement, 1 Copy (Distributed).
(Parallel Execution, MsgCPUTime = 1 ms)

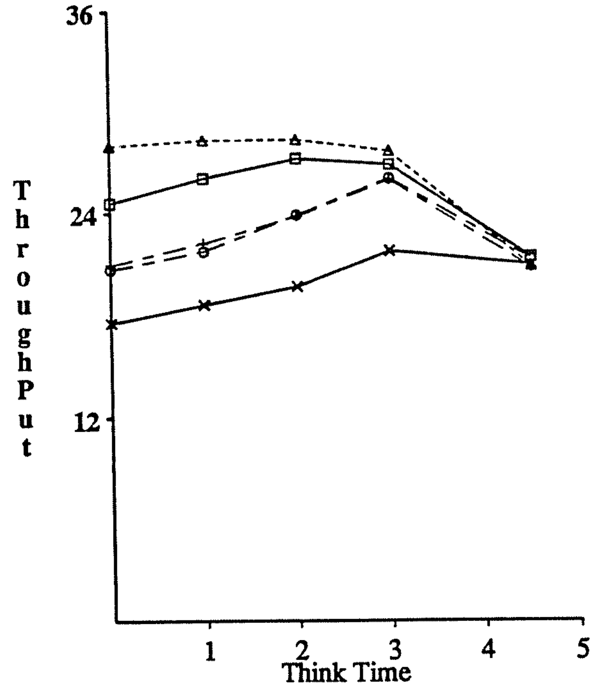

Figure 18: Throughput, 1 Copy (Distributed).
(Parallel Execution, MsgCPUTime = 4 ms)

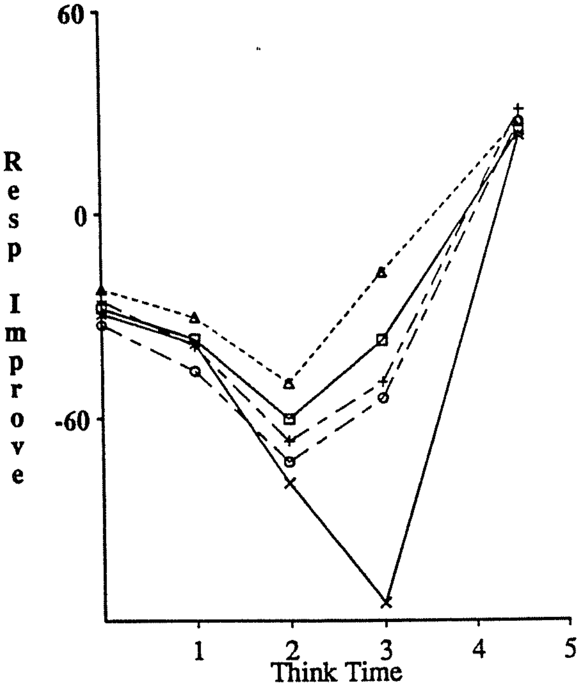

Figure 19: Improvement, 1 Copy (Distributed).
(Parallel Execution, MsgCPUTime = 4 ms)

is not a complete explanation; the distributed nature of transactions also turns out to have an interesting impact on algorithm behavior.

In the case of OPT, an examination of the restart ratios underlying these curves revealed that the frequency of restarts is higher here than for serial execution. We believe that this is due to the following scenerio: In OPT, given a pair of transactions that each have multiple cohorts, a problem can arise if the cohorts concurrently attempt to read and then update common data — the cohorts can end up being locally certified in opposite orders at their different sites. If this happens, both transactions will end up being restarted, as they each see the other as potentially invalidating their readset. This is not a problem in the local case, as the transactions will certify in one order or the other (in a critical section) in this case, leading to only one of them being restarted. Further, this problem is exacerbated when the message cost is increased, since this cost determines the period of time needed for certification and it is during this period when transactions are vunerable to the problem. This hypothesis was borne out by an examination of the restart ratios. OPT was found to have a very significant increase in its restart ratio in going from the 1 millisecond case to the 4 millisecond case. Moreover, the additional messages involved in running the transaction commit protocol in the parallel case are enough to move OPT from I/O-bound to CPU-bound operation in the 4 millisecond case. This makes its restarts relatively more expensive than in the serial case.

2PL is also negatively affected, at least to some extent, by parallelism under high loads. In particular, a comparison of Figure 16 to Figure 4 reveals that 2PL's throughput drops off more significantly under high loads in the parallel case. This is due to the fact that, if one cohort of a transaction has difficulty obtaining locks while other cohorts of the same transaction are successful, the others will eventually become blocked as well — waiting for the commit protocol to begin, while holding all of their locks. As a result, the average number of locked items is higher in the case of the parallel execution pattern, causing more blocking (and also deadlocks) and leading to a drop in the utilization of the bottleneck resource under heavy loads. This effect was indeed visible in the resource utilizations in the parallel case, which fell off for 2PL at the 0 and 1 second think time points. We also instrumented our 2PL implementation to keep track of the number of locked items, and we found the average number of locked items to be 35-40% higher in the parallel case than in the sequential case with a think time of zero. WW suffers from a related phenomenon, but only marginally so because it uses a mix of blocking and restarts to handle conflicting requests.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have tried to shed light on distributed concurrency control performance tradeoffs by studying the performance of four representative algorithms — distributed 2PL, wound-wait, basic timestamp ordering, and a distributed optimistic algorithm — using a common performance framework. We examined the performance of these algorithms under various degrees of contention, data replication, and workload "distributedness."

In terms of the relative performance of the algorithms, we found that 2PL and OPT dominated BTO and WW. When the cost of sending and receving messages was low, 2PL was the superior performer due to its avoidance of transaction restarts. However, when the message cost was high and data was replicated, OPT was seen to outperform 2PL due to its ability to exchange the necessary synchronization information using only the messages of the two-phase commit protocol. In such cases, for our workload, the work lost due to restarts was more than compensated for by the savings due to avoiding costly messages. The combination of these results suggests that "optimistic locking," where transactions lock remote copies of data only as they enter into the commit protocol (at the risk of end-of-transaction deadlocks), might actually be the best performer in replicated databases where messages are costly. We intend to investigate this conjecture in the future by adding such an algorithm to our current repetoire. Finally, BTO and WW performed almost indistinguishably in our initial experiments.

In terms of replication, increasing the number of copies had the expected effect on performance due to update costs. However, we found that increasing the number of copies of data did not change the relative ordering of the algorithms at the lowest or highest message costs, where (respectively) 2PL and OPT dominated the other algorithms. When the message cost was such that increasing the number of copies moved the system from an I/O-bound situation into a CPU-bound one, changing the number of copies was sufficient to move OPT into the performance leadership position.

Turning to distribution and parallelism, we examined two cases in this paper: One where transactions executed serially, but sometimes nonlocally, and the other where transactions executed in parallel at several sites. In the nonlocal case, minor differences were observed as compared to the case of strictly local execution. In the parallel case, we observed some interesting behavior. First, our results indicated that, as one would expect, parallelism is only beneficial under light loads, especially if messages are expensive. We also observed some algorithm-related phenomena. Multiple cohorts were found to increase the likelihood of restarts for OPT, particularly under higher message costs. This was attributed to the lack of a single critical section, because it is possible for concurrently certifying transactions to restart one another. In 2PL, an increase in waiting due to lock contention was observed in the case of parallel execution. We intend to study these effects more fully in the future, with much more highly parallel workloads, in order to understand the kinds of concurrency control problems that may arise in highly parallel database machines.

In addition to these performance-related observations, we found that our distributed transaction processing testbed provides an effective environment for debugging and experimenting with distributed concurrency control and commit protocols. In the future, we intend to use it to investigate the points raised above, and we also plan to study the performance-availability tradeoffs raised by crash-resilient concurrency control algorithms.

REFERENCES

[Agra87] Agrawal, R., Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Systems* 12, 4, Dec. 1987.

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases," *Proc. of the COMPSAC '79 Conf.*, Chicago, IL, Nov. 1979.

[Balt82] Balter, R., Berard, P., and Decitre, P., "Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management," *Proc. of the 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Aug. 1982.

[Bern80a] Bernstein, P., and Goodman, N., *Fundamental Algorithms for Concurrency Control in Distributed Database Systems*, Tech. Rep., Computer Corp. of America, Cambridge, MA, 1980.

[Bern80b] Bernstein, P., and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," *Proc. of the 6th Int'l. Conf. on Very Large Data Bases*, Oct. 1980.

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Comp. Surveys* 13, 2, June 1981.

[Bern84] Bernstein, P., and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Trans. on Database Systems* 9, 4, Dec. 1984.

[Eern87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proc. of the 10th Int'l. Conf. on Very Large Data Bases*, Singapore, Aug. 1984.

[Care85] Carey, M., Livny, M., and Lu, H., "Dynamic Task Allocation in a Distributed Database System," *Proc. of the 5th Int'l. Conf. on Distributed Computing Systems*, Denver, Colorado, May 1985.

[Care86a] Carey, M., and Lu, H., "Load Balancing in a Locally Distributed Database System," *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Washington, DC, May 1986.

[Care86b] Carey, M., and Muhanna, W., "The Performance of Multiversion Concurrency Control Algorithms," *ACM Trans. on Computer Systems* 4, 4, Nov. 1986.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proc. of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Feb. 1982.

[Chei86] Cheiney, J., et al, "A Reliable Parallel Backend Using Multiattribute Clustering and Select-Join Operator," *Proc. of the 12th Int'l. Conf. on Very Large Data Bases*, Aug. 1986.

[DeWi86] DeWitt, D., et al, "GAMMA — A High Performance Backend Database Machine," *Proc. of the 12th Int'l. Conf. on Very Large Data Bases*, Aug. 1986.

[Eage83] Eager, D., and Sevcik, K., "Achieving Robustness in Distributed Database Systems," *ACM Trans. on Database Systems* 8, 3, Sept. 1983.

[ElAb88] El Abbadi, A., and Toueg, S., "The Group Paradigm for Concurrency Control Protocols," *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Chicago, IL, June 1988.

[Fush86] Fushimi, S., et al, "An Overview of the System Software of a Parallel Relational Database Machine GRACE," *Proc. of the 12th Int'l. Conf. on Very Large Data Bases*, Aug. 1986.

[Gall82] Galler, B., *Concurrency Control Performance Issues*, Ph.D. Thesis, Computer Science Department, Univ. of Toronto, Sept. 1982.

[Garc79] Garcia-Molina, H., *Performance of Update Algorithms for Replicated Data in a Distributed Database*, Ph.D. Thesis, Computer Science Dept., Stanford Univ., June 1979.

[Garc83] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database System," *ACM Trans. on Database Systems* 8, 2, June 1983.

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Herl87] Herlihy, M, "Dynamic Quorum Adjustments for Partitioned Data," *ACM Trans. on Database Systems* 12, 2, June 1987.

[Jajo87] Jajodia, S., and Mutchler, D., "Dynamic Voting," *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, San Francisco, CA, May 1987.

[Kohl85] Kohler, W., and Jenq, B., *Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed*, Tech. Rep. No. CS-85-133, Dept. of Elec. and Computer Eng., Univ. of Massachusetts, Amherst, 1985.

[Lazo86] Lazowska, E., et al, "File Access Performance of Diskless Workstations," *ACM Trans. on Computer Systems* 4, 3, Aug. 1986.

[Lin82] Lin, W., and Nolte, J., "Performance of Two Phase Locking," *Proc. of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks, Feb. 1982.*

[Lin83] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking," *Proc. of the 9th Int'l. Conf. on Very Large Data Bases*, Florence, Italy, Nov. 1983.

[Lind79] Lindsay, B., et al, *Notes on Distributed Databases*, Rep. No. RJ2571, IBM San Jose Research Laboratory, 1979.

[Lind84] Lindsay, B., et al, "Computation and Communication in R*," *ACM Trans. on Computer Systems* 2, 1, Feb. 1984.

[Lind87] Lindsay, B., et al, personal communication.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1988.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases," *Proc. of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, Aug. 1978.

[Oszu85] Oszu, M., "Modeling and Analysis of Distributed Database Concurrency Control Algorithms Using an Extended Petri Net Formalism," *IEEE Trans. on Software Eng.* SE-11, 10, Oct. 1985.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data," *ACM Trans. on Computer Systems* 1, 1, Feb. 1983.

[Ries79a] Ries, D., *The Effects of Concurrency Control on Database Management System Performance*, Ph.D. Thesis, Dept. of Elec. Eng. and Computer Science, Univ. of California, Berkeley, 1979.

[Ries79b] Ries, D., "The Effects of Concurrency Control on the Performance of a Distributed Data Management System," *Proc. of the 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, Aug. 1979.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. on Database Systems* 3, 2, June 1978.

[Schl81] Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. of the 7th Int'l. Conf. on Very Large Data Bases*, Cannes, France, Sept. 1981.

[Schw84] Schwarz, P., and Spector, A., "Synchronizing Shared Abstract Data Types," *ACM Trans. on Computer Systems* 2, 3, Nov. 1984.

[Sinh85] Sinha, M., et al, "Timestamp Based Certification Schemes for Transactions in Distributed Database Systems," *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Austin, TX, May 1985.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. on Software Eng.* SE-5, 3, May 1979.

[Tera83] *Teradata DBC/1012 Data Base Computer Concepts & Facilities*, Teradata Corp. Document No. C02-0001-00, 1983.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Systems* 4, 2, June 1979.

[Weih85] Weihl, W., and Liskov, B., "Implementing Resilient, Atomic Data Types," *ACM Trans. on Prog. Lang. and Systems* 7, 1, April 1985.