Conflict Detection Tradeoffs for Replicated Data

by
Michael J. Carey
Miron Livny

Computer Sciences Technical Report #826
March 1989

Conflict Detection Tradeoffs for Replicated Data

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).

Conflict Detection Tradeoffs for Replicated Data

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 examine some of these tradeoffs by using a detailed model of a distributed DBMS to study a set of representative algorithms, including several derivatives of the two-phase locking, timestamp ordering, and optimistic approaches to distributed concurrency control. In particular, we examine the performance of these algorithms as a function of data contention for various levels of data replication and "distributedness" of accesses to replicated data. The results provide some interesting insights into how the tradeoffs between early and late conflict detection vary as a function of message cost, and should prove useful to distributed database system designers.

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 enhancing performance by improving data locality, enabling parallel query execution and load balancing, and 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 (e.g., [Tera83, DeWi86]). 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 a large number of distributed 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, Ston79, Trai82], *timestamp* algorithms, [Thom79, Bern80b, Reed83], and *optimistic* (or

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).

certification) algorithms [Bada79, Schl81, Ceri82, Sinh85]. Bernstein and Goodman review many of the proposed algorithms and describe how additional algorithms may be synthesized by combining basic mechanisms from the locking and timestamp classes [Bern81].

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 was investigated in [Garc79, Ries79, Lin82, Oszu85, Noe87], where algorithms with different degrees of centralization of locking and various approaches to deadlock handling have been studied and compared with one another. Several distributed timestamp-based algorithms were examined in [Li87]. 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 optimistic algorithms in [Bhar82, Kohl85]. 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 data accesses 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. In [Bhar82], a central site wound-wait variant was compared with a distributed optimistic algorithm, message costs were high, and tran-

saction restart costs were biased by buffering assumptions. The results of [Kohl85] were obtained using a lightly loaded two-site testbed system, and were strongly influenced by the fact that both data and log records were stored on the same disk. The second question above remains open since most 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.[1] 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 assumptions, especially with respect to system resources [Agra87]. Given the framework, we then proceed to examine the performance of a representative set of distributed concurrency control algorithms as a function of data contention for various levels of data replication and "distributedness" of accesses to replicated data. While we address only a subset of the open questions, our results provide useful insights regarding how and when conflicts should be detected in a distributed DBMS and how the associated tradeoffs are influenced by data replication and message cost.

We examine five concurrency control algorithms in this study, including three locking variants, a timestamp-based 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 performance

[1] Preliminary results of this study were presented in [Care88].

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 five algorithms that we consider to be representative of the basic design space for distributed concurrency control mechanisms. In terms of their treatment of replicated data, each algorithm that we consider is from the "read one copy, write all copies" family.[2] Four of the algorithms that we consider here are drawn directly from the distributed database literature. The fifth is essentially a hybrid of several of the others; we conjectured at the end of [Care88] that such an algorithm would perform well. Together, the five algorithms span the three major algorithm classes, and they represent a wide range of conflict detection times and resolution methods. We summarize their salient features in this section, describing their relative advantages and disadvantages. First, however, we describe the process structure that we will be assuming 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 or *coordinator* process (M) that runs at the site where the transaction originated. The coordinator 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

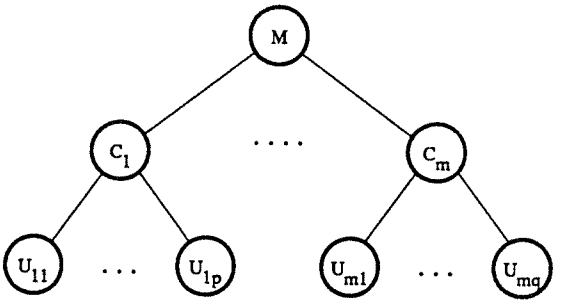

Figure 1: Distributed transaction structure.

[2] We focus on the performance of this family of algorithms because alternatives like quorum-based schemes have the major drawback of turning read operations into multi-site operations, even for local data [Bern87]. Also, [Eage83], [Bern84], and [ElAb85] all behave like "read one, write all" algorithms in the absence of failures.

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 support a variety of query execution patterns; the number of cohorts and whether cohorts execute sequentially or in parallel depends on the query execution model of interest. We will clarify this point further in describing the workload model in Section 3. For now, 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 a replicated data item is assumed to have one or more *remote update* (U_{ij}) processes associated with it at other sites. In particular, a cohort will have a remote update process at every other site that stores a copy of a data item that it updates. It communicates with its remote update processes for concurrency control purposes, and it also sends them copies of the relevant updates during the first phase of the commit protocol described below.

In this study, we assume the use of a centralized two-phase commit protocol [Gray79], with the coordinator process controlling the protocol. This same protocol is used in conjunction with each of the concurrency control algorithms examined. Assuming no replication, the protocol works as follows: When a cohort finishes executing its portion of a query, its sends an "execution complete" message to the coordinator. When the coordinator has received such a message from every cohort, it will initiate the commit protocol by sending "prepare to commit" messages to all cohorts. Assuming that a cohort wishes to commit, it sends a "prepared" message back to the coordinator. The coordinator will send "commit" messages to each cohort after receiving a "prepared" message from every cohort. The protocol ends with the coordinator 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 coordinator to send "abort" instead of "commit" messages in the second phase of the protocol. Earlier aborts are handled similarly: An aborted cohort will report to the coordinator, which then instructs all of the cohorts to abort the transaction; each aborted cohort reports back to the coordinator once the abort procedure has been completed.

When remote update processes are present, the commit protocol becomes a nested two-phase commit protocol as described in [Gray79]: Messages flow between the coordinator and the cohorts, and the cohorts in turn interact with their remote updaters. That is, each cohort sends "prepare to commit" messages to its remote updaters after receiving such a message from the coordinator, and it gathers the responses from its remote updaters before sending

a "prepared" message back to the coordinator; phase two of the protocol is similarly modified. Again, this is reminiscent of the "tree of processes" transaction structure of R* [Lind84]. Copies of updated data items are carried in the "prepare to commit" messages sent from cohorts to remote updaters.

2.2. Distributed Two-Phase Locking (2PL)

The first algorithm is the distributed two-phase locking algorithm described in [Gray79]. Transactions set read locks on items that they read, converting 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 committed or aborted.

Deadlock is possible, 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 whenever a transaction blocks. When a local deadlock is detected, it is resolved by aborting the transaction with the most recent initial startup time among those involved in the deadlock cycle. 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" process 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)

The second algorithm is the distributed wound-wait locking algorithm of [Rose78]. It differs from 2PL only in its handling of the deadlock problem: Rather than maintaining waits-for information and then checking for local and global deadlocks, deadlocks are prevented via the use of timestamps. Each transaction is numbered according to its initial startup time[3], and younger transactions are prevented from making older ones wait. When a transaction

[3] Lamport clocks can be used to ensure the global uniqueness of such timestamps in a distributed system [Lamp78].

requests a lock, and the lock request is blocked due to a younger transaction, the younger transaction is "wounded" — it is aborted 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 can 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 waiting as well, and this is prevented by the algorithm.

2.4. Basic Timestamp Ordering (BTO)

The third algorithm is the basic timestamp ordering algorithm of [Bern80b]. Like wound-wait, it employs transaction startup timestamps, but it uses them 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 aborted. 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 [Bern80b]).

With replicated data, a read request is processed using the local copy of the requested data item, while a write request must be approved at all copies before the transaction proceeds. Integration of the algorithm with two-phase commit is accomplished as described in [Bern80b]: Writers keep their updates in a private workspace until commit time. 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. 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)

The fourth algorithm is the distributed, timestamp-based, optimistic concurrency control algorithm from [Sinh85][4], 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 coordinator, the transaction is assigned a unique timestamp. This timestamp is sent to each cohort in the "prepare to commit" message, and it is used to locally certify all of the cohort's reads 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 the timestamp ordering here, so these conditions are checked using the timestamp given to the transaction when it started the commit protocol. Local certification is performed in a critical section.

To handle replicated data, the algorithm requires remote updaters to participate in certification. Updaters simply certify the set of writes that they receive at commit time, and the necessary communication can again be accomplished by passing information in the messages of the commit protocol. Failure of the certification test by any cohort or remote 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 aborted. Comparing OPT with the previous algorithms, notice that 2PL, WW, and BTO send write access requests from a cohort to its remote updaters whenever a cohort makes a write request for replicated data. Thus, copy-related conflicts are detected when they occur, but at the expense of communicating with all copy sites in order to process each write request. In contrast, OPT defers all communication between cohorts and remote updaters until commit time, avoiding these additional messages; of course, conflicts will not be detected until end-of-transaction as a result.

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

2.6. Distributed 'Optimistic' Two-Phase Locking (O2PL)

The fifth algorithm, O2PL, can be thought of as a more 'optimistic' version of 2PL. O2PL handles read requests and deadlock detection in the same way that 2PL does; in fact, O2PL and 2PL are absolutely identical in the absence of replication. However, O2PL handles replicated data like OPT does. When a cohort updates a replicated data item, it requests a write lock immediately on the local copy of the item, but it defers requesting write locks on any of the remote copies until the beginning of the commit phase is reached. Thus, as in OPT, communication with remote copy sites can be accomplished by simply passing information in the messages of the commit protocol. In particular, the "prepare to commit" request sent by a cohort to its remote updaters includes a list of items to be updated, and each remote updater must obtain write locks on these items before it can act on the prepare request itself.

Since O2PL waits until end-of-transaction to obtain write locks on copies, both blocking and deadlocks are possible rather late in the execution of a transaction (just like aborts in OPT). For example, if two transactions at different sites both try to read and then update different copies of a common data item, an end-of-transaction global deadlock will occur, causing one of the transactions to be aborted eventually by the "Snoop" process. In order to speed up the detection and resolution of such deadlocks, our O2PL algorithm requires updaters to request special *copy locks* rather than normal write locks. Copy locks are just like write locks in terms of their compatibility with themselves and with other lock types, but they enable the lock manager to know when a lock is being requested on a copy of a replicated data item. Then, when the lock manager sees a copy lock request waiting behind a write lock request, or vice versa, it can immediately detect this as being an indication of an impending global deadlock[5] and immediately abort the younger of the two transactions. This is what the "Snoop" would do eventually, but this way the global deadlock becomes a problem that can be detected and resolved earler (and hence more cheaply).

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. The general structure of the model is presented in Figure 2. Each site in the model has four components: a *source*, which generates transactions and also

[5] Note that the requester of a copy lock at one site must already hold a write lock on another copy; conversely, the holder of a write lock on one copy must eventually acquire copy locks on all other copies.

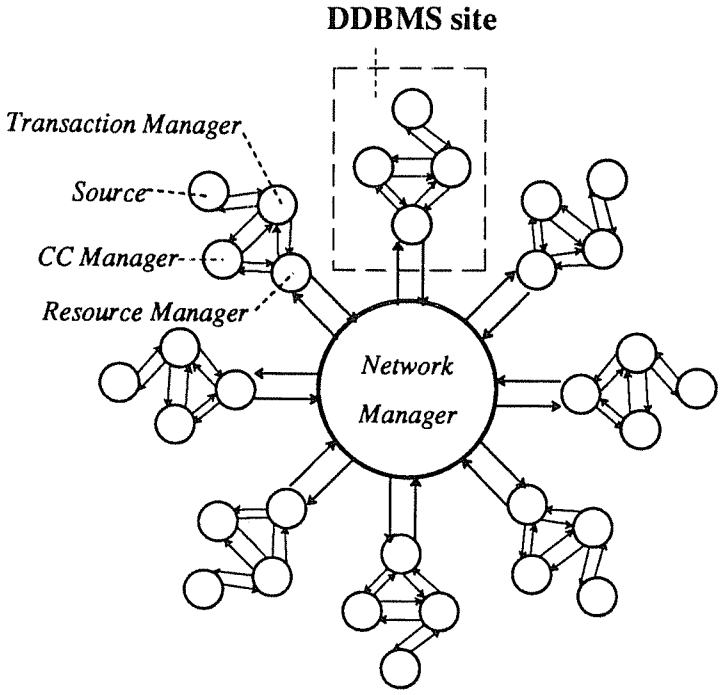

Figure 2: Distributed DBMS Model Structure.

maintains transaction-level performance information for the site, a *transaction manager*, which models the execution behavior 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 an entire 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 data 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*,

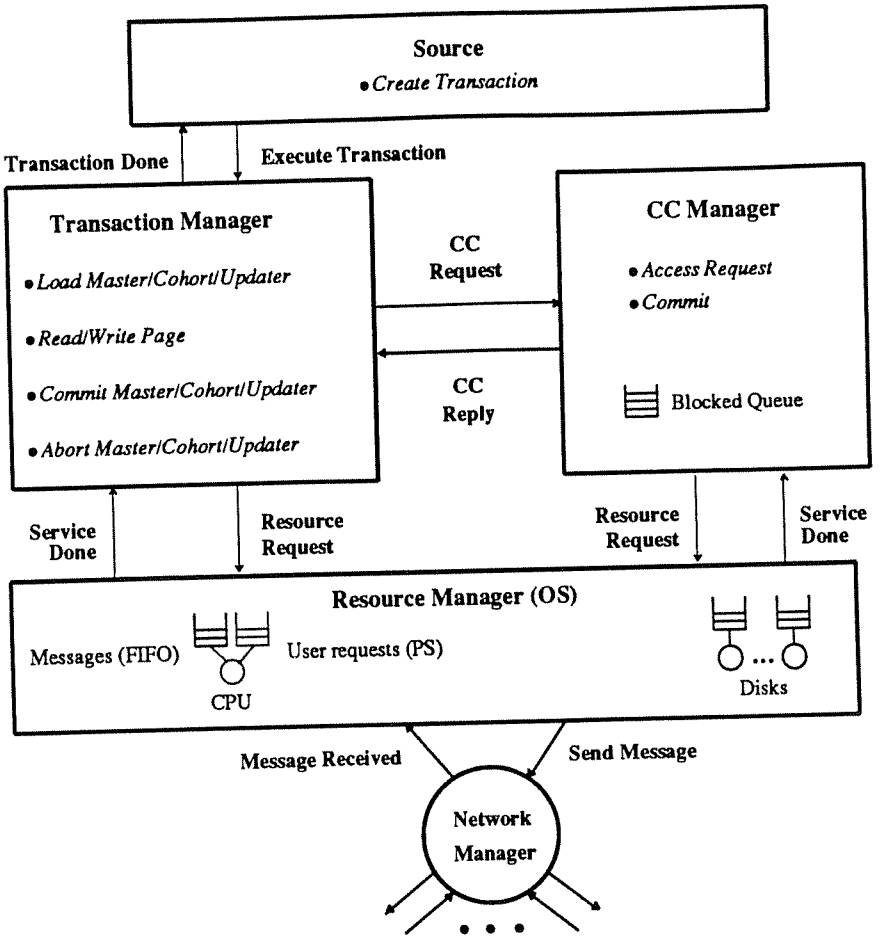

Figure 3: A Closer Look at the Model.

a boolean array in which $FileLocations_{ij}$ is true if a copy of file i resides at site j.

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

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.

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 think time between the completion of one transaction and the submission of the next one at a terminal. *NumClasses* gives the number of transaction classes for the site.

The *ClassFrac* parameter specifies the fraction of the site's terminals that generate transactions 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 about this shortly.) *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 coordinator process, a number of cohorts, and possibly a number of remote updaters. As described earlier, the coordinator resides at the site where the transaction originated. Each cohort makes a sequence of read and write requests to one or more files that are stored locally; a transaction has one cohort at each site where it needs to access data. Cohorts com-

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	
ClassFrac	Fraction of terminals 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.

municate with their remote updaters when remote write access permission is needed for replicated data, and the remote 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, 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 query execution that is seen in systems like Gamma [DeWi86] or the Teradata database machine [Tera83]. In this paper, sequential transactions will have just one cohort, which accesses a collection of files residing at a single site, and we will not consider parallel transactions. The model is capable of handling the more general case, however [Care88, Care89].

The transaction manager is responsible for accepting transactions from the source and modeling their execution. To choose the execution site(s) 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 remote 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 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 coordinator 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.[6] 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 aborted, the concurrency control manager

[6] 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.

notifies the transaction manager, which then invokes the abort protocol. Once the transaction manager has finished aborting the transaction, it delays the coordinator for a period of time before letting it restart and rerun the transaction; as in [Agra87], we use one average transaction response time (as observed at the coordinator 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 a site's operating system and resources; it manages the physical resources of the site, including its CPU and its disks. The resource manager provides CPU, I/O, and message-sending services; sending and receiving messages involves use of the CPU resource. The transaction manager uses the CPU and I/O services for reading and writing disk pages. The concurrency control manager uses the CPU service for processing concurrency control requests. Both the transaction manager and concurrency control manager use the message-sending services provided by the resource manager for their communication needs.

The parameters of the resource manager are summarized in Table 3. Each site has one CPU and *NumDisks* disks. The CPU service discipline is first-come, first-served (FIFO) for message service and processor sharing for all other services; message processing preempts other CPU services. 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. We believe that the main cost of sending a message in a local area network is the CPU processing cost at 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 model that captures the behavior of a wide area network, for example.

3.6. The Concurrency Control Manager

The concurrency control manager captures the semantics of a given concurrency control algorithm, and it 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 coordinator and cohort management requests to initialize and terminate coordinator and cohort processes. We have implemented a total of five concurrency control managers based on the algorithms described in Section 2.

The concurrency control manager has an algorithm-dependent number of parameters. All algorithms have the parameter *CCReqCPU*, which specifies the amount of CPU time required to process a read or write access request. 2PL and O2PL have one additional parameter, *DetectionInterval*. This parameter determines the amount of time that a site should wait, after becoming the "Snoop" site, before gathering global waits-for information and performing global deadlock detection.

4. EXPERIMENTS AND RESULTS

In this section, we present our performance results for the five concurrency control algorithms of interest under various assumptions about data replication, the nature of accesses to replicated data, and the CPU cost for sending and receiving messages. 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.[7] Several additional metrics are used to aid in the analysis of the experimental results. One is the *message ratio*, which is the average number of messages sent per committed transaction. This metric is computed by dividing the total number of messages sent by the number of transaction commits. We also examine the average number of times per commit that a transaction has to be restarted due to an abort, which we call the *restart ratio*. This is computed by dividing the number of transaction aborts by the number of transaction commits. Finally, in addition to examining these metrics for each of the five concurrency control algorithms, we will also present results for two "algorithms" referred to as NO_DC_{EC} (no data contention, early communication) and NO_DC_{LC} (no data contention, late communication). NO_DC_{EC} processes read and write requests for replicated data in a manner identical to that of 2PL, WW, and BTO, but it grants all requests immediately. Similarly, NO_DC_{LC} processes read and write requests just like OPT and O2PL do, again granting them right away. Put differently, NO_DC_{EC} and NO_DC_{LC} capture the message-related behavior of the two sets of algorithms, but without data contention, thereby providing useful performance bound information. We will present NO_DC_{EC} results in all graphs, and we will give NO_DC_{LC} results where they differ significantly from NO_DC_{EC} (i.e., with multiple copies of data and significant message CPU costs).

Table 4 gives the values of the key simulation parameters in our experiments. We consider a database that is distributed over 8 sites. The database consists of 8 files, each containing 2400 pages of data. There are 50 terminals per site, and the mean terminal think time is varied over a range from 0 to 10 seconds in order to vary the load on the system. In terms of the workload, a transaction accesses data from a single file; it reads an average of 18 pages of the file and updates an average of one-fourth of these pages. Thus, the size of the average transaction is 22.5 (18 reads and 4.5 writes). This transaction size was chosen as being relatively small, as transactions tend to be in transaction processing environments. The corresponding database size and the number of terminals were selected so as to provide an interesting level of data contention. Finally, it takes a transaction an average of 8 milliseconds of CPU

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

time to process each page that it reads or writes. The degree of file replication and the choice of the particular file to be accessed by a given transaction varies from experiment to experiment; we will discuss these choices shortly.

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 control CPU overhead is assumed to be negligible, for all algorithms, compared to the 8 millisecond CPU time for page processing. Lastly, the global deadlock detection interval for 2PL and O2PL is 1 second.[8]

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 also ran a subset of the experiments with a larger page CPU time, making the system CPU-bound regardless of communication activity, but this produced only quantitative (and not qualitative) differences in the results [Care88]. We will comment only very briefly on these CPU-bound

Parameter	Setting
NumSites	8 sites
NumFiles	8 files
$FileSize_i$	2400 pages per file
NumTerminals	50 terminals per site
ThinkTime	0-10 seconds
FileCount	1 file
$FileProb_i$	1.0
$NumPages_i$	18 pages
$WriteProb_i$	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.

[8] With the parameter settings in Table 4, the system can generate up to 800 lock requests per second, so each global deadlock check is thus amortized over hundreds of lock requests.

results here. Finally, 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 [Care86], we wish to focus our attention here on the cost issues related to concurrency control.

In the remainder of this section, we report on the results of a series of six experiments. We vary three main parameters across the experiments — the CPU cost associated with sending and receiving messages, the number of copies per file, and the number of *active* copies per file (described below). We vary these parameters as follows:

Message Cost: The message CPU cost is varied as shown in Table 4. We performed experiments with the *MsgCPUTime* parameter set to 1, 4, and then 10 milliseconds per message send or receive operation. We refer to these subsequently as the low, medium, and high message cost settings.

File Replication: In each experiment, files are placed at sites as follows: There are eight sites, $S_i, 1 \leq i \leq 8$, and eight files, $F_i, 1 \leq i \leq 8$. In the one copy case, file F_i is stored at site S_i. When we consider two copies of each file, file F_i is stored both at site S_i and site $S_{(i+1) \bmod 8}$. In the three copy case, an additional copy of the file is stored at site $S_{(i+2) \bmod 8}$. Finally, in the case of full replication, which we include in the study for completeness, each file is stored at all sites.

Number of Active Copies: In all of the experiments reported here, transactions perform all of their reads locally. Thus, they only interact with remote sites if they involve updates to replicated files. In the workload that we call the *one-copy-active* workload, transactions that originate at site S_i always access file F_i. When data is not replicated, then, this is basically a centralized concurrency control situation. When data is replicated, the distributed nature of the system is used only to improve availability. Furthermore, with this workload, although checking still takes place, there are no multi-site conflicts (e.g., global deadlocks in 2PL or O2PL, or remote aborts in OPT); this is because all conflicts involving F_i are detected and resolved by the cohort running at site S_i, so F_i essentially serves as a primary copy for the file. In the other workload used in this study, called the *all-copies-active* workload, every copy of a replicated file will be accessed. In this workload, if the number of copies of each file is k, transactions that originate at site S_i choose any one of the files stored at S_i to operate on; that is, a transaction at S_i will select one of the files $F_{(i-j) \bmod 8}$, $0 \leq j \leq k-1$ to access. Here, terminals at each site are statically partitioned so that each of the site's file copies will be accessed by transactions from one-k^{th} of the site's terminals, meaning that each copy of a file is presented with a statistically equivalent workload. In the *all-copies-active* workload, multi-site

concurrency control conflicts do arise. The *one-copy-active* and *all-copies-active* workloads are used here because they represent the two extremes from the range of copy access behavior that may occur in actual systems.

4.2. Experiment 1: Low Message Cost, One Copy Active

The purpose of this experiment is to investigate the performance of the five concurrency control algorithms as the system load varies, and to see how performance is impacted by different levels of data replication. We assume efficient communications software in this experiment, using a value of 1 millisecond for *MsgCPUTime*, and we focus our attention on the one-copy-active workload. We vary the number of copies per file from one (no replication) to eight (full replication), as described above. We view the one, two, and three copy cases as being in some sense reasonable; the eight-copy case is included for completeness, in order to demonstrate the limiting behavior of the system as the number of copies is increased.

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[9] 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 simultaneously; throughput then degrades for all five of the concurrency control algorithms due to data contention. At the peak for each algorithm, the disk utilization of the system is nearly 100%. These trends are natural for a centralized DBMS [Care84, Agra87]. The NO_DC_{EC} 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 five concurrency control algorithms studied here, Figure 4 indicates that 2PL and O2PL provide the best performance, followed by WW and BTO (which perform quite similarly), followed by OPT. 2PL and O2PL perform identically in Figure 4; this is because they differ only in their handling of replicated data, and data is not replicated here.

To understand the relative throughput ordering of the algorithms, Figure 5 presents their restart ratios. The results are easily explained based on these ratios. 2PL and O2PL have the lowest restart ratios by far, and consequently perform the best. WW and BTO have higher restart ratios, providing the next best throughput results. Note

[9] 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.

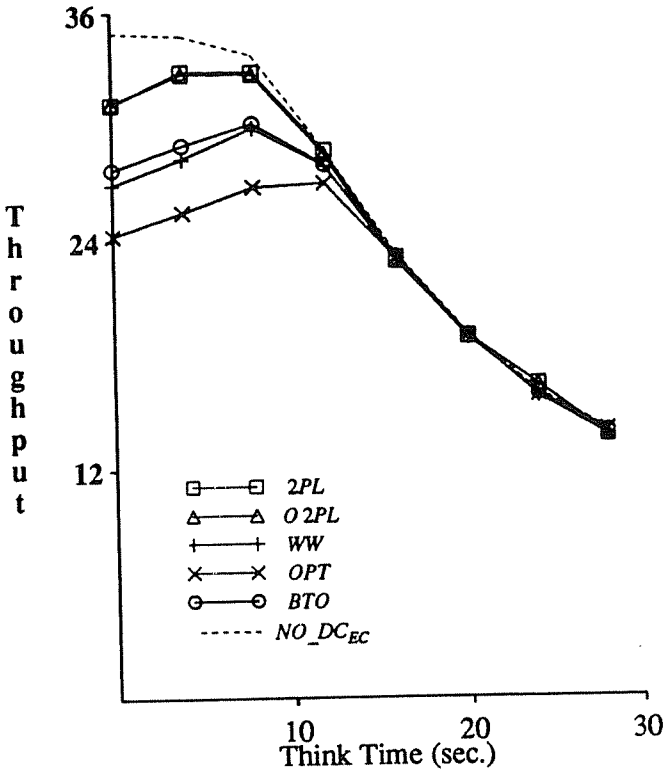

Figure 4: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 1 copy)

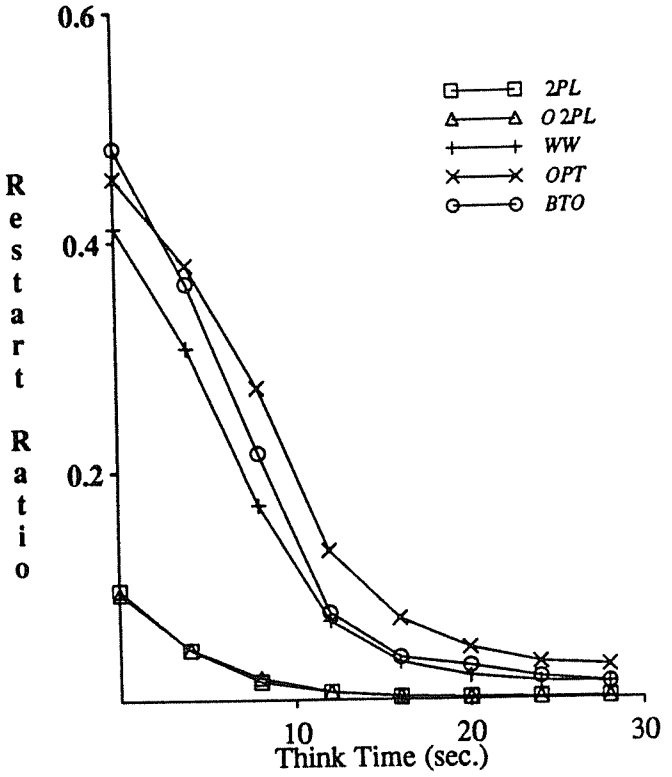

Figure 5: Restart Ratio.
(MsgCPUTime = 1 ms, 1 copy)

that BTO and WW perform nearly alike despite having different restart ratios. Although WW has a higher ratio of restarts to commits, it always selects a younger transaction to abort, making its individual restarts less costly than those of BTO. OPT has the highest restart ratio except when the terminal think time is zero. In addition, OPT aborts transactions at commit time, rather than earlier as in WW and BTO. As a result, OPT has the lowest throughput among the algorithms, and it does not take a very big difference in the restart ratios to cause OPT to have significantly lower throughput under high loads. These results indicate the importance of restart ratios and the age of aborted transactions as a performance determinant.

Figures 6-8 present the throughput results for the two, three, and eight copy cases, respectively. (We do not present the associated restart ratios, as they are quite similar to those shown for the one copy case.) 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. As a result, several trends are evident in the figures: First,

increasing replication leads to decreased performance due to the additional update work; this is seen most dramatically in the fully-replicated case. The performance decrease due to replication 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 despite the CPU cost for sending and receiving messages even in the eight copy case. The second trend is that 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.[10] This is because the amount of effort wasted becomes a smaller fraction of the transaction's total required effort. Finally, Figures 7 and 8 show O2PL having a very slight performance advantage over 2PL in the three and eight copy cases. This is strictly due to the fact that O2PL requires fewer messages, and thus has a marginally lower CPU overhead, as we will see (much more clearly) in Experiment 3 and in subsequent experiments. Notice also that there is no "price" associated with using O2PL over 2PL here because, as noted earlier, all conflicts are resolved locally (i.e., before the commit phase, and before remote updaters are even initiated in O2PL) under the one-copy-active workload.

As discussed in [Care88], we also performed versions of the one, two, and three copy experiments starting with a CPU-bound system. 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 multiple copy cases, the differences between algorithm performance did not shrink to the same extent. This is because end-of-transaction updates have less impact on CPU than on I/O, and CPU was the bottleneck. Lastly, we found that 2PL, WW, and BTO suffered a bit more as the level of replication was increased, as the presence of copies implies a round-trip, inter-site concurrency control message for each write in these three algorithms. This is the same effect that produced the slight O2PL performance advantage mentioned above, but it was a bit more evident in this experiment since messages involve CPU cost (and the system was CPU-bound).

[10] This is because our model assumes that the buffer pool is large enough that updates need not be propagated to disk until after a transaction commits.

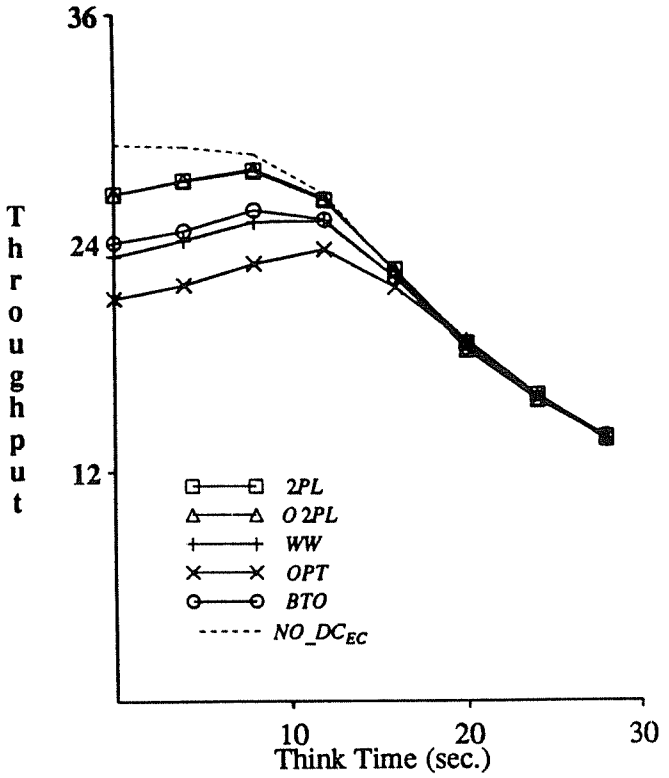

Figure 6: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 2 copies, ONE active)

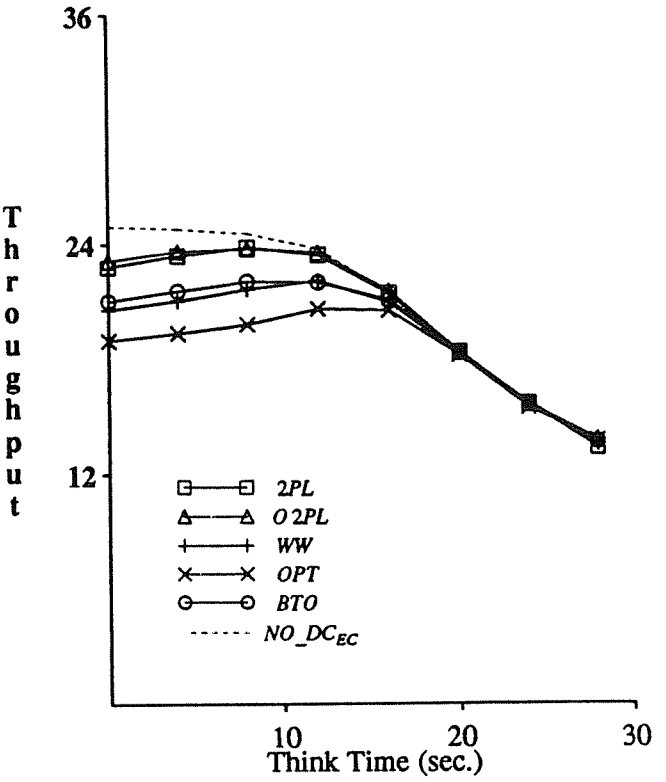

Figure 7: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 3 copies, ONE active)

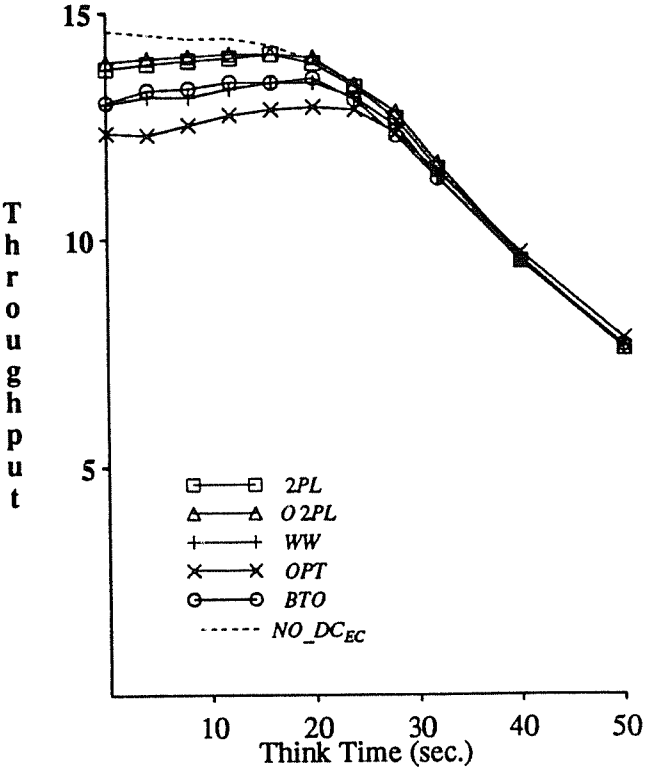

Figure 8: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 8 copies, ONE active)

4.3. Experiment 2: Low Message Cost, All Copies Active

The purpose of this experiment is to examine the impact of replicated data access on concurrency control algorithm performance. With the exception of the workload parameters, the parameter settings used for Experiment 2 are the same as those of the previous experiment. Thus, this experiment amounts to Experiment 1 repeated for the all-copies-active workload with two, three, and eight copies per file. Figures 9-11 present the throughput results for these different replication levels under the all-copies-active workload. With the exception of O2PL, the performance results shown here are very similar to those observed in Experiment 1. Here, rather than having a slight performance advantage over 2PL, O2PL actually performs a little bit worse than 2PL. The other algorithms seem to be insensitive, or at least nearly so, to the change in the nature of the workload.

The reason for O2PL's performance shift under the all-copies-active workload can be understood by considering the impact of this workload on how conflicts are detected and resolved in O2PL. In the one-copy-active workload, since all of the transactions that access a given file do so on the same site, all conflicts for that file occur on a single site. It is a certainty that, once a transaction succeeds in obtaining all of its read and write locks on this copy of the file, it will succeed in acquiring the necessary remote write locks later. Also, the one-copy-active workload is free of distributed deadlocks. In the all-copies-active case, however, this is not true. Here, it is possible in O2PL for each copy site to have an update transaction obtain locks locally, discovering the existence of competing update transaction(s) at other sites only after successfully executing locally. Competing update transactions are detected early under the one-copy-active workload, being handled by a mix of blocking and deadlock-induced aborts. Under the all-copies-active workload, however, such conflicting update transactions will discover one another much later, at a point where the only means for resolving the conflict is to abort one of the conflicting transactions. Figure 12 illustrates these differences by showing the restart ratios for the various algorithms in the two copy case; note that O2PL has a significantly higher restart ratio than 2PL.

The insensitivity of the other algorithms to the number of active copies is easily explained. 2PL does not suffer like O2PL because it does not wait until end-of-transaction to discover remote update conflicts; it discovers them earlier since write locks are obtained on both local and remote copies before the local write proceeds.[11] Thus,

[11] The observant reader may have noted that 2PL does perform just a little bit worse when the think time is zero in the all-copies-active case than it did in the one-copy-active case. This is because some local deadlocks in the one-copy-active workload are distributed deadlocks here, being resolved somewhat later due to message delays and the periodic nature of distributed deadlock detection.

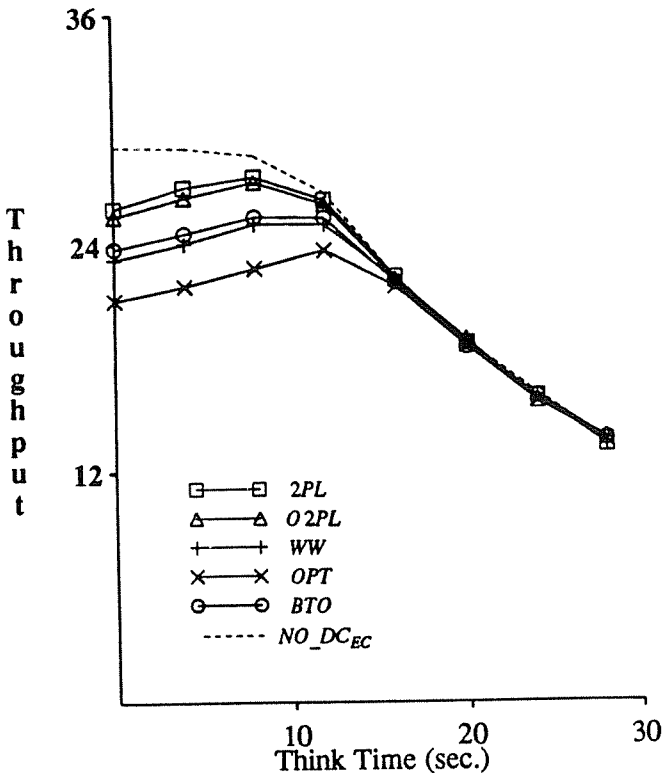

Figure 9: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 2 copies, ALL active)

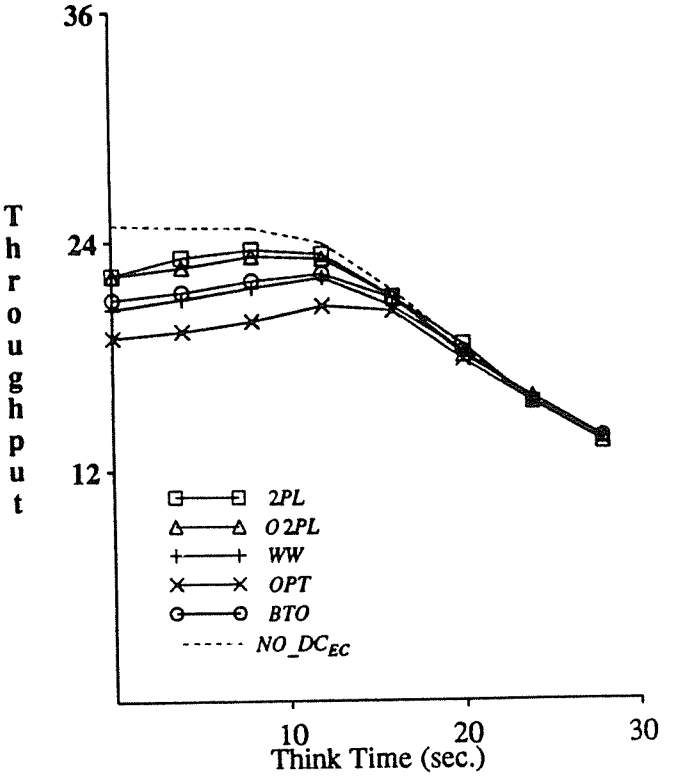

Figure 10: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 3 copies, ALL active)

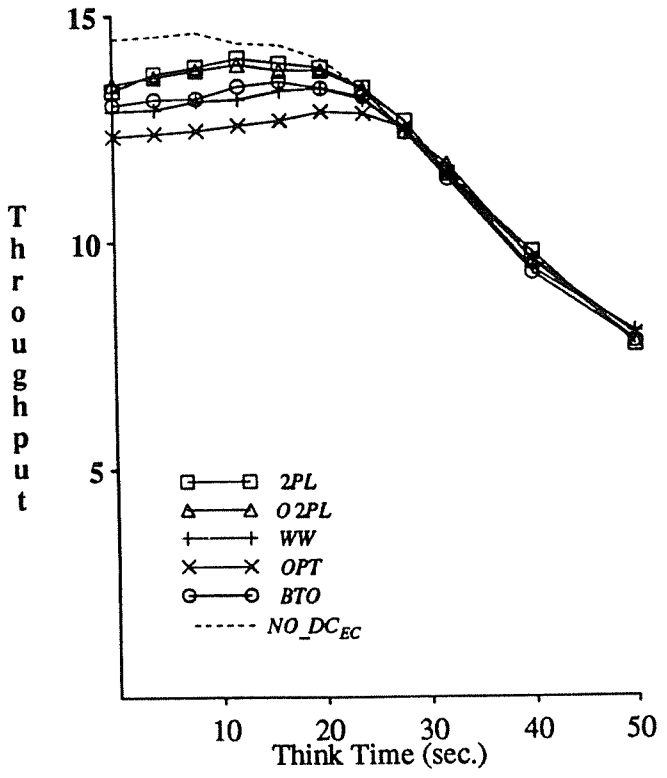

Figure 11: Throughput (transactions/sec.).
(MsgCPUTime = 1 ms, 8 copies, ALL active)

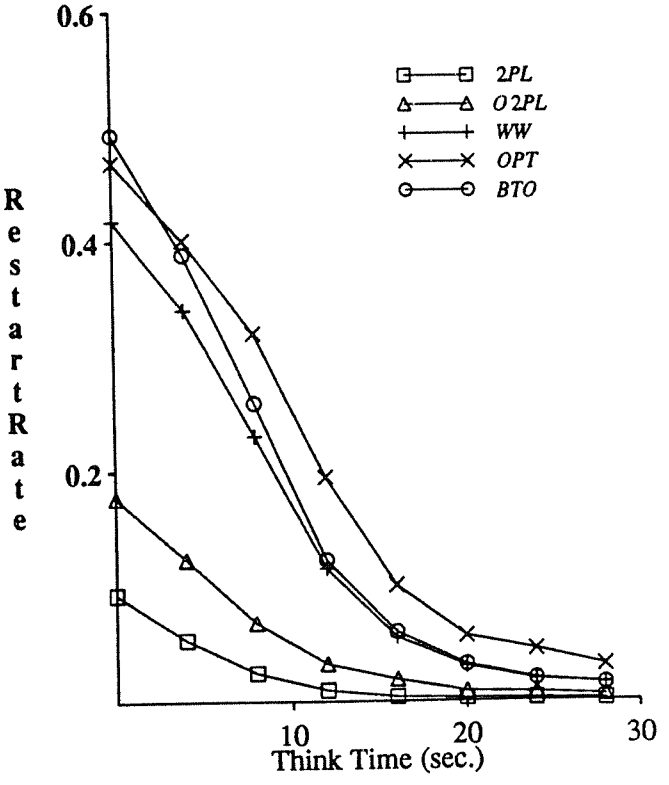

Figure 12: Restart Ratio.
(MsgCPUTime = 1 ms, 2 copies, ALL active)

write lock information on all copies is kept similar (within the limits imposed by message delays), and the chances of a deadlock occurring due to a lock upgrade conflict on one data item are minimal. WW and BTO also communicate with remote copy sites on a per-write basis, and they both use local information (transaction timestamps) to resolve conflicts as they occur; they are also therefore unaffected by the number of active copies. OPT is insensitive to the number of active copies because it only checks for conflicts at end-of-transaction, and it matters little whether the conflicts occur at one copy site or multiple copy sites (since all copy sites are involved in the validation process anyway).

4.4. Experiment 3: Medium Message Cost, One Copy Active

This experiment examines the impact of an increase in message cost on performance. The data layout and workload used here are identical to those of Experiment 1. However, instead of using the low *MsgCPUTime* setting of 1 millisecond here, we use the medium value of 4 milliseconds. We remind the reader that this parameter determines the CPU time to send or receive a message, meaning that the 4 millisecond value places an 8 millisecond lower bound on the message transfer time. 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).[12] As in Experiment 1, the one-copy-active workload is used as the basis for this experiment.

Figure 13 presents the throughput results obtained by repeating the two copy case from Experiment 1 with *MsgCPUTime* = 4 milliseconds. The results in this figure are quite different than those of Figure 6. The performance of each of the algorithms is worse here because of the additional message cost. However, OPT and O2PL suffer the least from the additional cost due to their use of commit protocol messages to handle copy-related update requests; the difference between the NO_DC_{EC} and NO_DC_{LC} curves shows the extent to which per-write messages affect performance. Thus, O2PL now performs significantly better than 2PL because the latter suffers much more from the increased message CPU cost. (In fact, O2PL even outperforms NO_DC_{EC} at some think times for this reason here.) OPT performs a bit better than BTO, providing the same level of performance as WW, and the difference between OPT and 2PL is now somewhat less dramatic.

[12] Only 2PL and O2PL have any message increase, due to deadlock detection, but this was not sufficient to significantly alter their performance here. In fact, the performance impact of these messages was negligible even for the highest of the three message CPU costs that we considered in our experiments.

Figure 14 shows the average number of messages per completed transaction, making it clear that O2PL and OPT require significantly fewer messages that the other algorithms. Looking deeper, when we examined the resource utilization levels in this case, we found that 2PL, BTO, WW, and NO_DC_{EC} are all CPU-bound here due to the CPU cost associated with their level of message activity; O2PL, OPT, and NO_DC_{LC}, however, remain I/O-bound. Thus, not only do O2PL and OPT require fewer messages, but messages have a lesser performance impact for these two algorithms because the CPU is not their bottleneck. O2PL retains its performance advantage over OPT here due to its much lower reliance on restarts to resolve conflicts. Note that the number of messages per commit for O2PL is slightly larger than the number for OPT. This is because aborts in OPT usually occur due to failure to certify a cohort, and thus involve no messages, whereas a significant number of the end-of-transaction aborts in O2PL occur after updaters have been initiated.[13] This is also why there is no noticeable difference between the number of messages per commit for OPT and NO_DC_{LC}. Finally, the fixed message overhead due to periodic deadlock detection in 2PL and O2PL is reflected at high think times in Figure 14. The number of messages per commit increases there because the same number of deadlock detection messages is amortized over fewer and fewer commits.

Figures 15 and 16 present the corresponding results for the medium message CPU cost in the three copy case. The trends that began in Figure 13 are more pronounced here. Again, all algorithms suffer performance-wise as compared with Experiment 1 due to the increased message cost. O2PL and OPT suffer the least, and O2PL provides the best performance. However, due to the large number of messages required by 2PL, BTO, and WW to interact with remote copy sites on a per-write basis, OPT actually outperforms all three of these algorithms here. This message cost explanation is supported dramatically by the fact that OPT also outperforms NO_DC_{EC}, which 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 compensated for by the message savings in this case, as 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 (and also for O2PL) in this case: Since remote copy updates are only performed after a successful commit, they are not done (and thus not undone) when OPT restarts a transaction. However, all of the CPU-related message activity required to obtain

[13] This will be explained in more detail in Section 4.7.

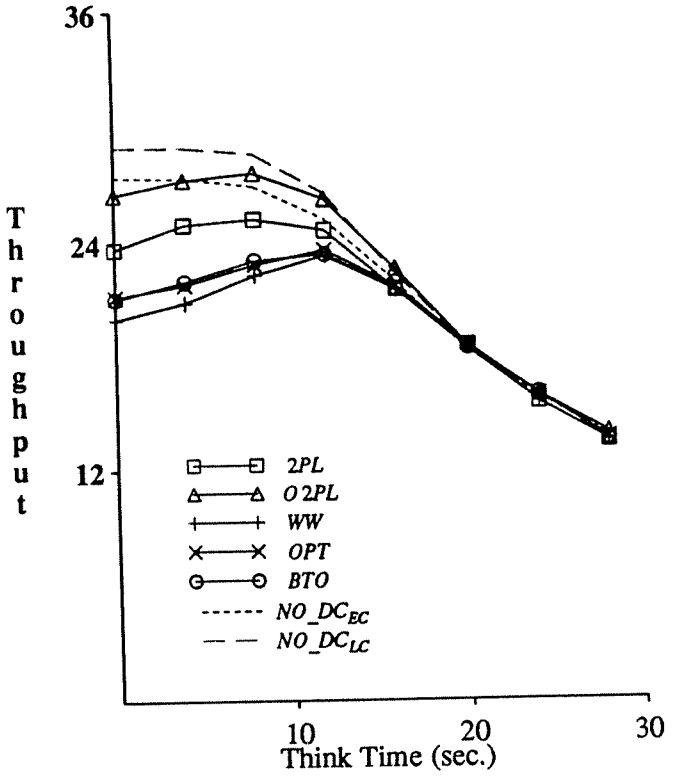

Figure 13: Throughput (transactions/sec.).
(MsgCPUTime = 4 ms, 2 copies, ONE active)

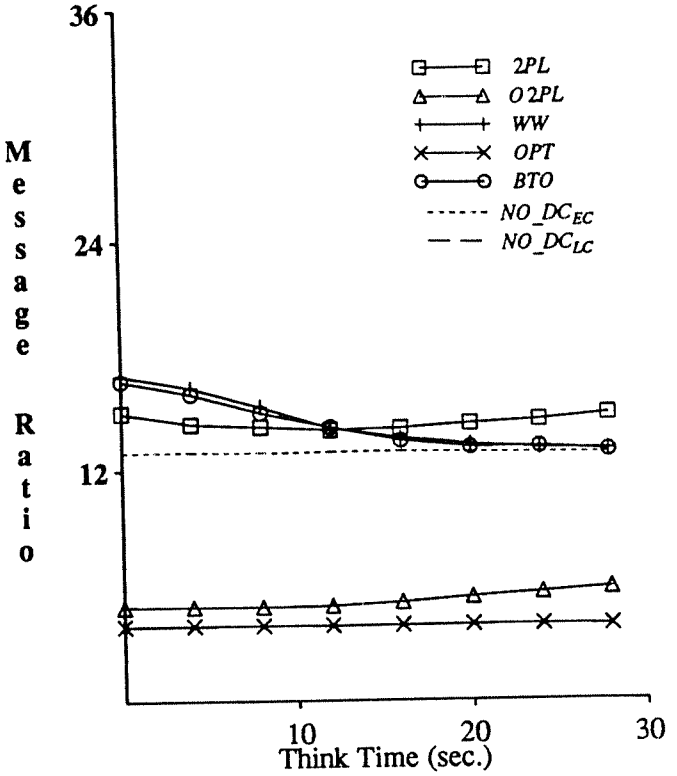

Figure 14: Message Ratio.
(MsgCPUTime = 4 ms, 2 copies, ONE active)

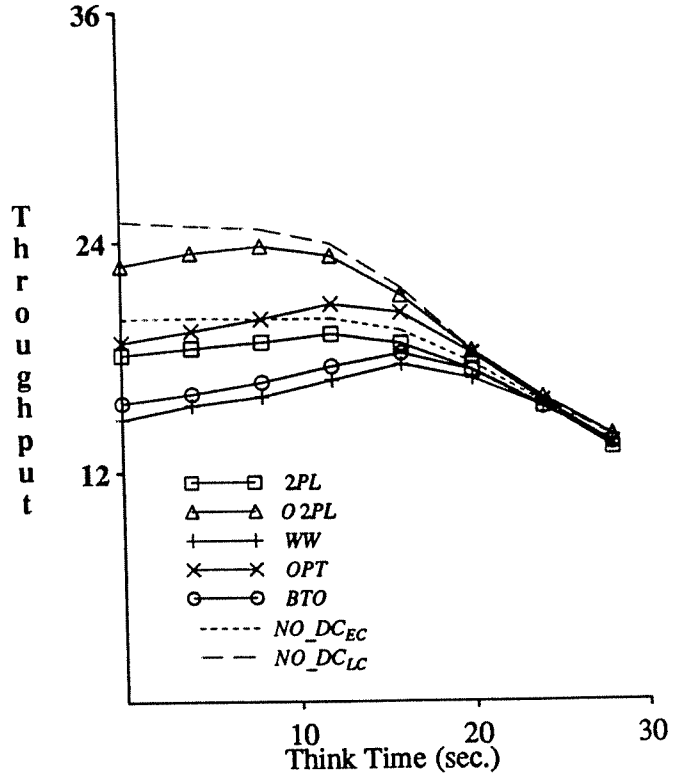

Figure 15: Throughput (transactions/sec.).
(MsgCPUTime = 4 ms, 3 copies, ONE active)

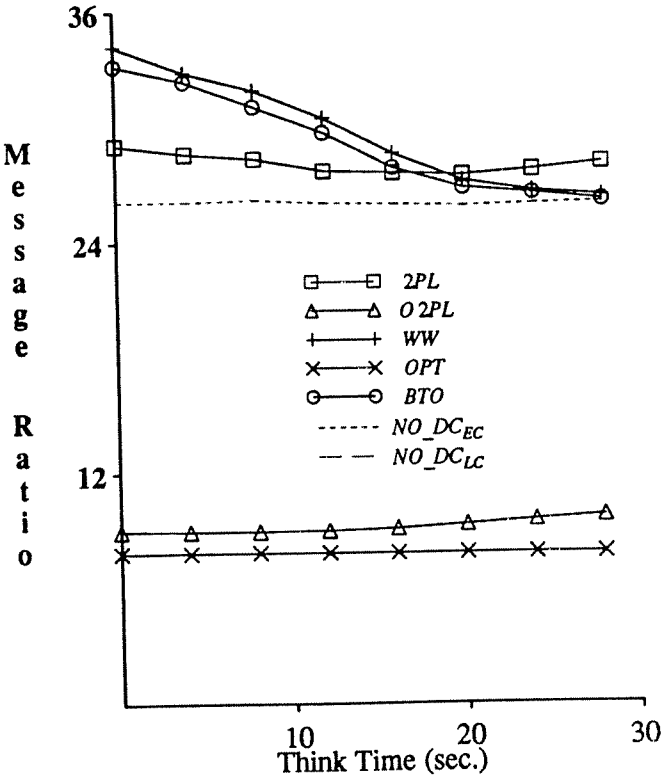

Figure 16: Message Ratio.
(MsgCPUTime = 4 ms, 3 copies, ONE active)

remote write 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 at the low think times in Figure 16, especially for BTO and WW.

Figures 17 and 18 present the throughput and message results obtained for the fully-replicated (eight copy) case. The shifts in the results are similar to those observed in the previous cases, except that they are heavily amplified here due to the large number of remote copy sites that each transaction must interact with in order to update a file. Among the subset of the algorithms that communicate with remote copies on a per-write basis, 2PL is still the best performer, followed by WW and BTO. However, all three of these algorithms perform much worse here than O2PL and OPT, which piggyback their remote copy communications on the messages of the commit protocol. Between the latter two algorithms, O2PL performs better. Again, this is due to its more judicious use of restarts relative to OPT.

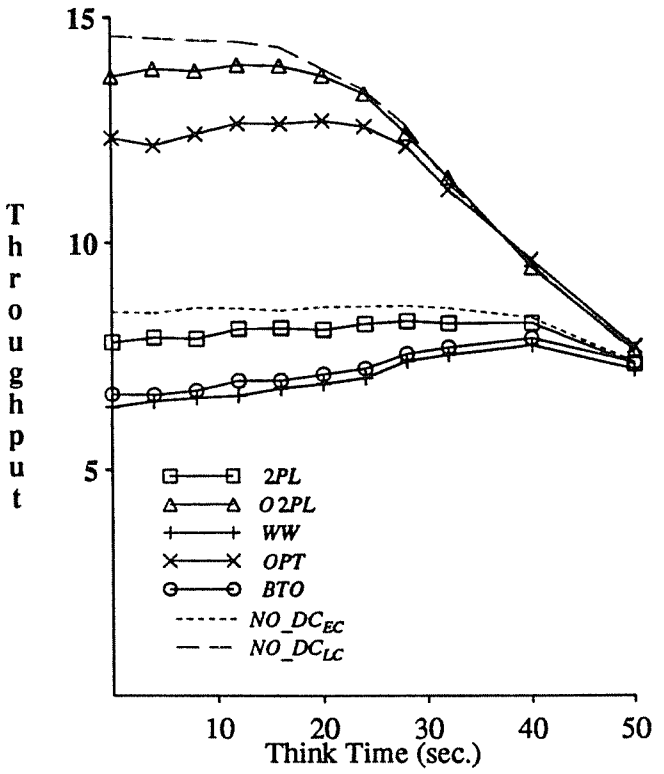

Figure 17: Throughput (transactions/sec.).
(MsgCPUTime = 4 ms, 8 copies, ONE active)

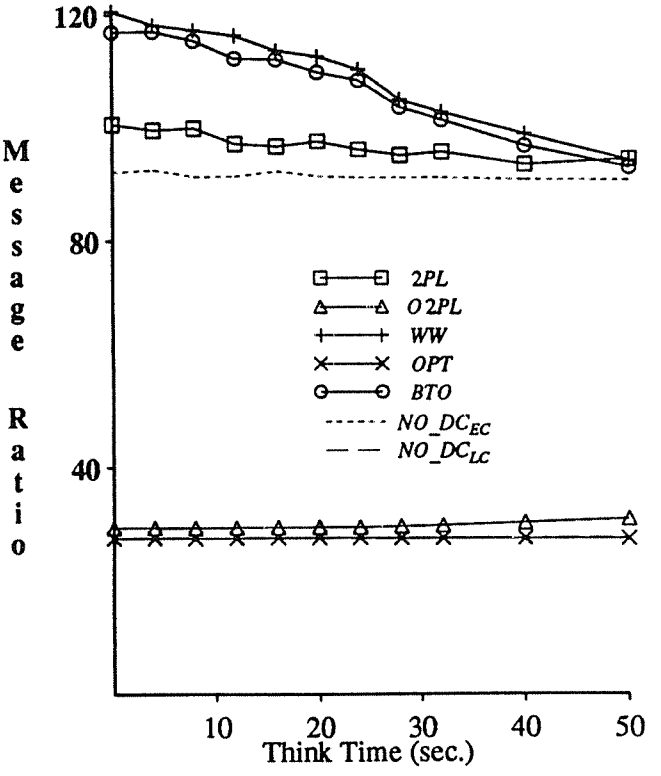

Figure 18: Message Ratio.
(MsgCPUTime = 4 ms, 8 copies, ONE active)

4.5. Experiment 4: Medium Message Cost, All Copies Active

This experiment repeats Experiment 3, with the medium message cost, but for the all-copies-active workload. Figures 19-21 show the throughput results obtained for the two, three, and eight copy cases, respectively. As in Experiment 2, with the notable exception of O2PL, the performance results obtained in this experiment are quite similar to those obtained in its one-copy-active counterpart. While the performance of the other algorithms is basically the same as in Experiment 3, O2PL performs somewhat worse. It suffers here for the reason described in Experiment 2 — when the number of active copies is increased, conflicts between competing updaters that execute at different copy sites go undetected until end-of-transaction, leading to aborts at that time. Unlike Experiment 2, however, this does not lead to O2PL being outperformed by 2PL. Here, due to the higher message cost incurred by 2PL, O2PL remains the superior performer. Thus, O2PL outperforms 2PL under the all-copies-active workload for the medium message cost setting; it just does not outperform 2PL by quite as much as it did under the one-copy-active workload of Experiment 3. Also, Figure 21 indicates that the performance of 2PL is quite close to that of NO_DC_{EC} in the eight copy case; this is because, when the system is heavily CPU-bound, blocking has less of a negative impact on the utilization of the CPU (and hence on throughput).

4.6. Experiment 5: High Message Cost, One Copy Active

This experiment examines the impact of a further increase in message cost on the performance of the algorithms. The data layout and workload used here are again identical to those of Experiment 1. However, instead of using the medium *MsgCPUTime* setting of 4 milliseconds, we assume a 10 millisecond time to send or receive a message. The one-copy-active workload is used for this experiment. Figures 22-24 present the throughput results for the various levels of data replication. Here, we see the trends that arose under the medium message cost taken to the extreme: For each of the replication levels, O2PL outperforms OPT, OPT outperforms 2PL, and 2PL outperforms WW and BTO. In particular, as shown in Figure 22, O2PL provides about 20% more throughput than OPT under high loads in the two copy case, and it outperforms 2PL by over 50% in this case. In the three copy case, shown in Figure 23, O2PL performs approximately 15% better than OPT and 100% better than 2PL. In the fully-replicated case, the combination of expensive messages and many copy sites causes the algorithms to merge into two classes of more or less indistinguishable performers — the per-write (or early) communicators and the end-of-transaction (or late) communicators. The latter outperform the former by about 200% in this very extreme case, where throughput is determined largely by the capacity of the CPU.

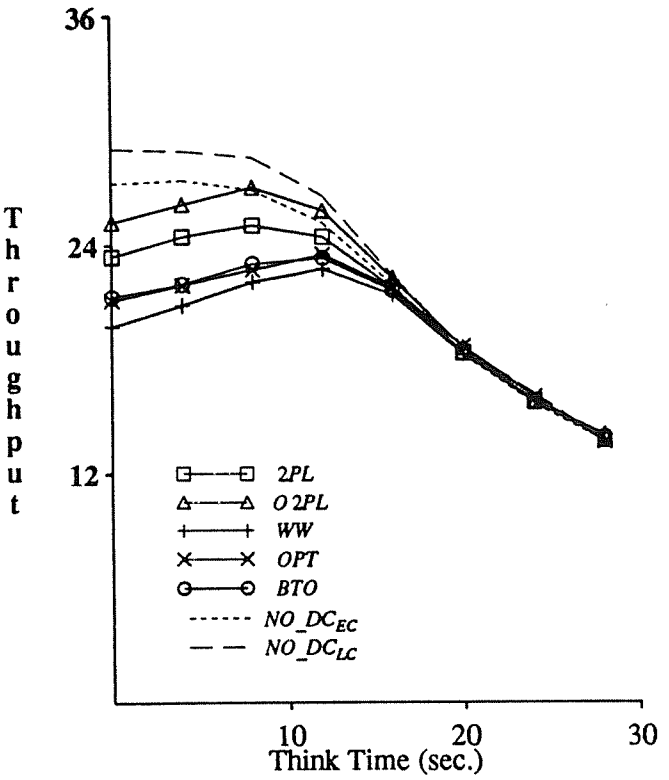

Figure 19: Throughput (transactions/sec.).
(MsgCPUTime = 4 ms, 2 copies, ALL active)

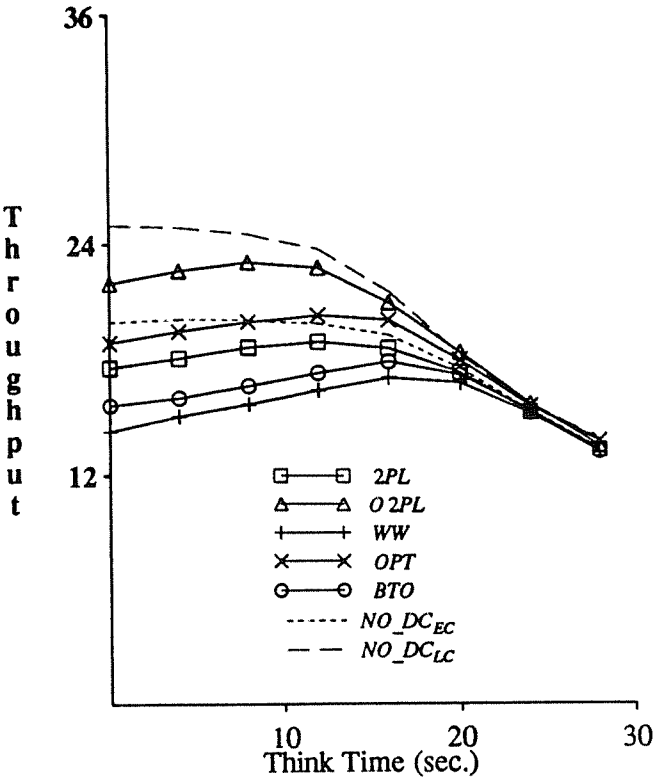

Figure 20: Throughput (transactions/sec.).
(MsgCPUTime = 4 ms, 3 copies, ALL active)

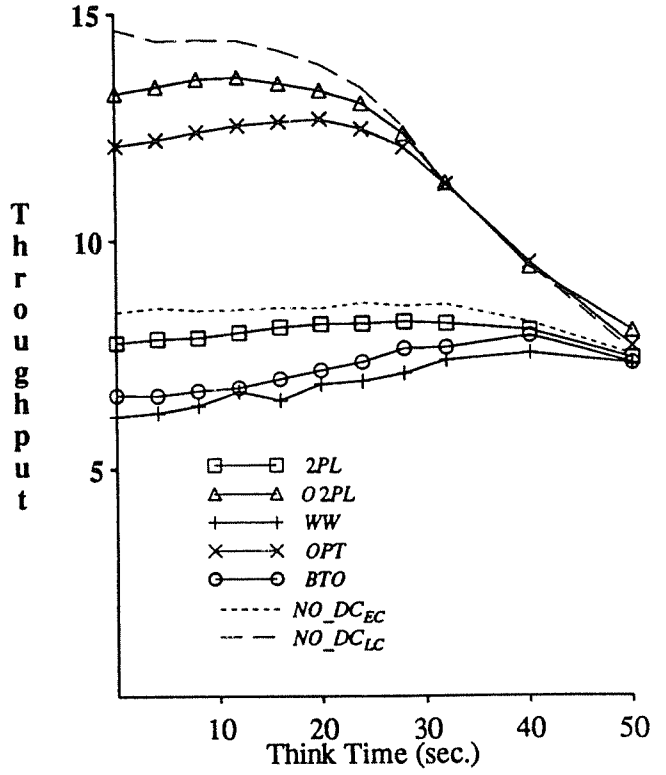

Figure 21: Throughput (transactions/sec.).
(MsgCPUTime = 4 ms, 8 copies, ALL active)

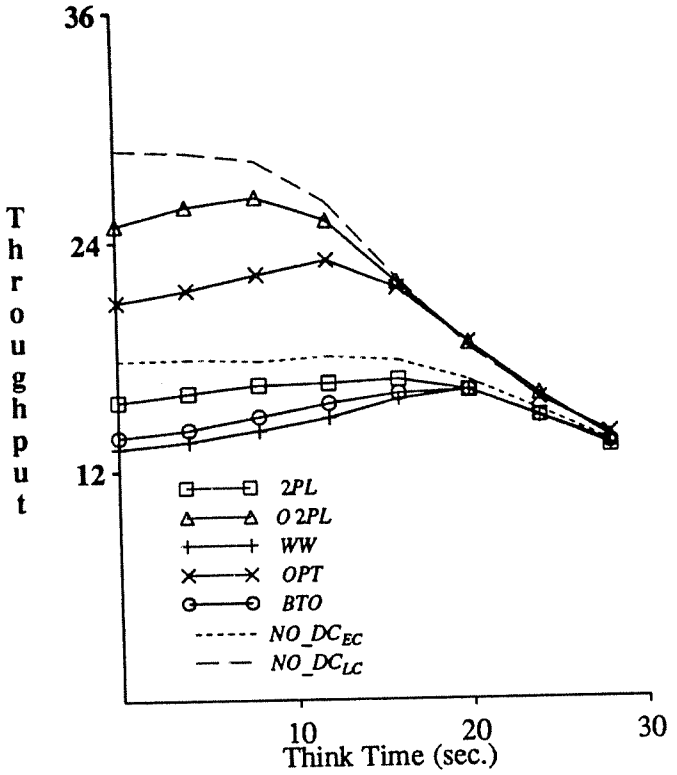

Figure 22: Throughput (transactions/sec.).
(MsgCPUTime = 10 ms, 2 copies, ONE active)

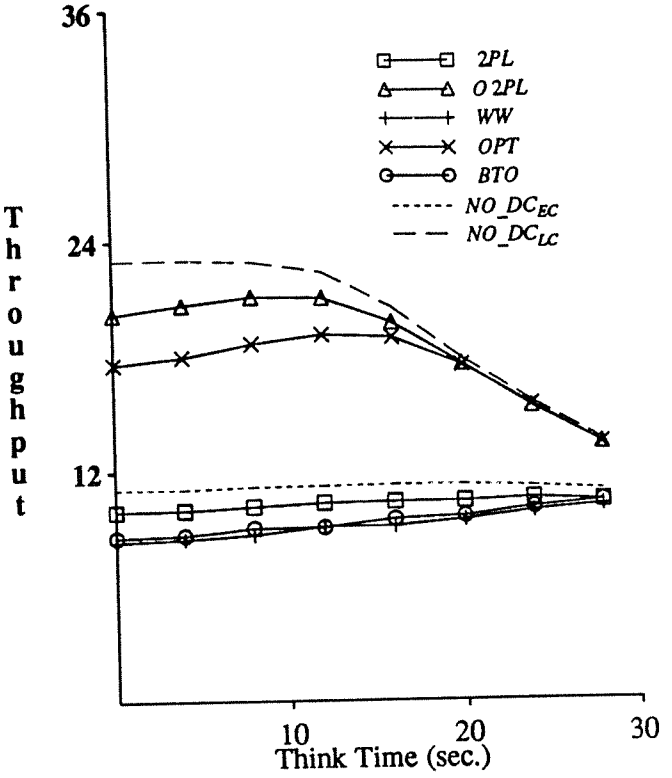

Figure 23: Throughput (transactions/sec.).
(MsgCPUTime = 10 ms, 3 copies, ONE active)

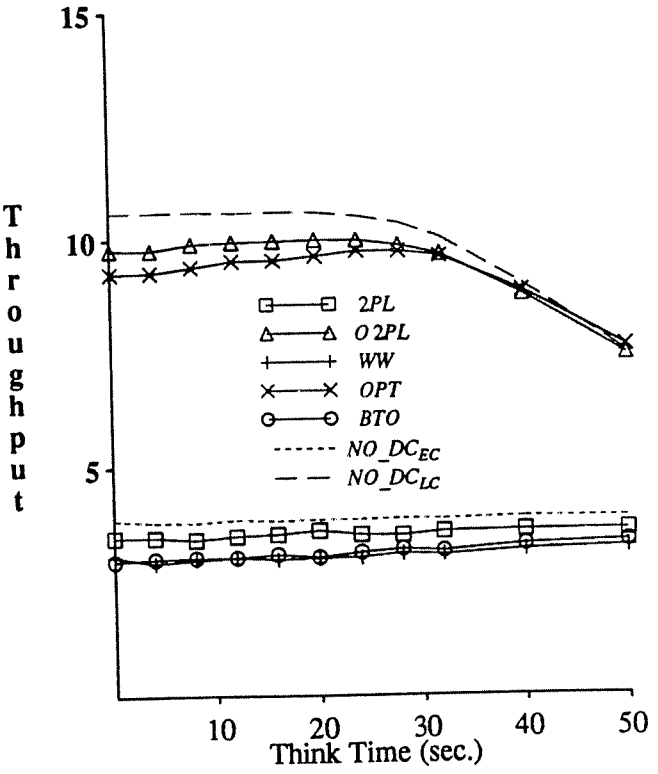

Figure 24: Throughput (transactions/sec.).
(MsgCPUTime = 10 ms, 8 copies, ONE active)

4.7. Experiment 6: High Message Cost, All Copies Active

This final experiment repeats Experiment 5 for the all-copies-active workload. The results are given in Figures 25-27. As one would expect from what we have already observed, the results are quite similar to those of Experiment 5 with the exception of O2PL. O2PL suffers visibly due to end-of-transaction aborts that occur when competing updaters execute simultaneously at different copy sites. However, it is still the clear performance leader in the two and three copy cases.

In the most extreme case, with eight active copies, O2PL actually performs slightly worse than OPT. In investigating why this is so, we found that it is essentially due to a fairness-related problem with OPT. Suppose that two transactions T_1 and T_2 are both attempting to read and update copies of a data item X, and suppose that T_1 enters its commit protocol first. In OPT, T_1 will commit (assuming no other conflicts), and its update will be recorded at all copies of X. Later, T_2 will be aborted because of its conflict with T_1. Also, note that this conflict will be discovered at T_2's cohort site, meaning that T_2 will abort without initiating remote updaters. In O2PL, however, the way that this conflict is dealt with depends on the relative age of T_1 and T_2. If T_1 is older, T_1 will abort T_2 due to a conflict at the copy of X being used by T_2's cohort; otherwise, T_1 itself will abort there. In the latter case, O2PL will be aborting a transaction after it has initiated its remote updaters, making the restart more costly (especially in communication terms) than in OPT. We discovered that modifying O2PL to favor committing transactions (like OPT does) can solve this problem, making O2PL uniformly superior to OPT here as well as in all our other experiments. However, this would not be a desirable change in general, as it would give O2PL the bias that OPT has against large transactions: In OPT, and in such a modified version of O2PL, small transactions can cause large transactions to starve in workloads that consist of both small and large transactions.

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 five representative algorithms using a common performance evaluation framework. The algorithms studied were distributed 2PL (2PL), wound-wait (WW), basic timestamp ordering (BTO), a distributed optimistic algorithm (OPT), and an 'optimistic' variant of 2PL (O2PL) that defers making remote lock requests until commit time. We examined the performance of these algorithms under various degrees of contention, data replication, message costliness, and "distributedness" of replica usage.

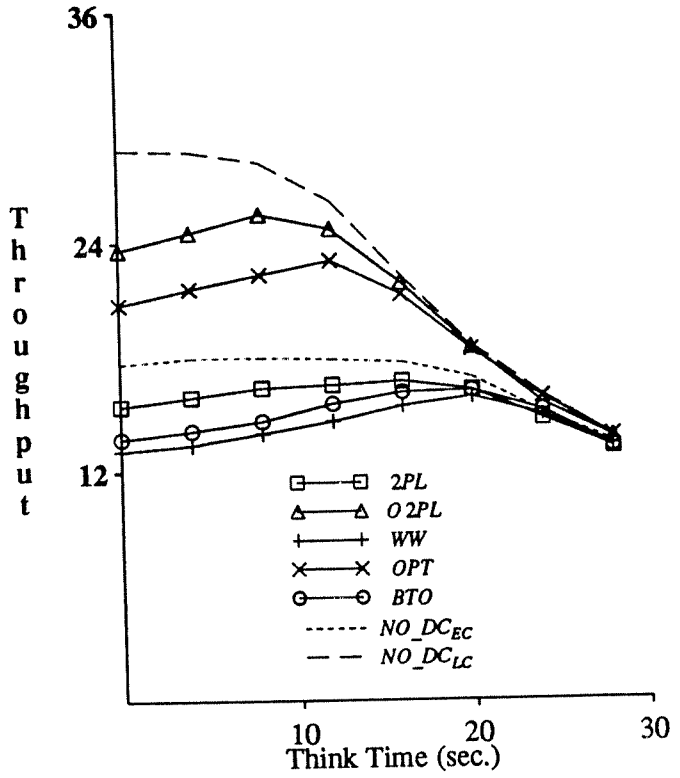

Figure 25: Throughput (transactions/sec.).
(MsgCPUTime = 10 ms, 2 copies, ALL active)

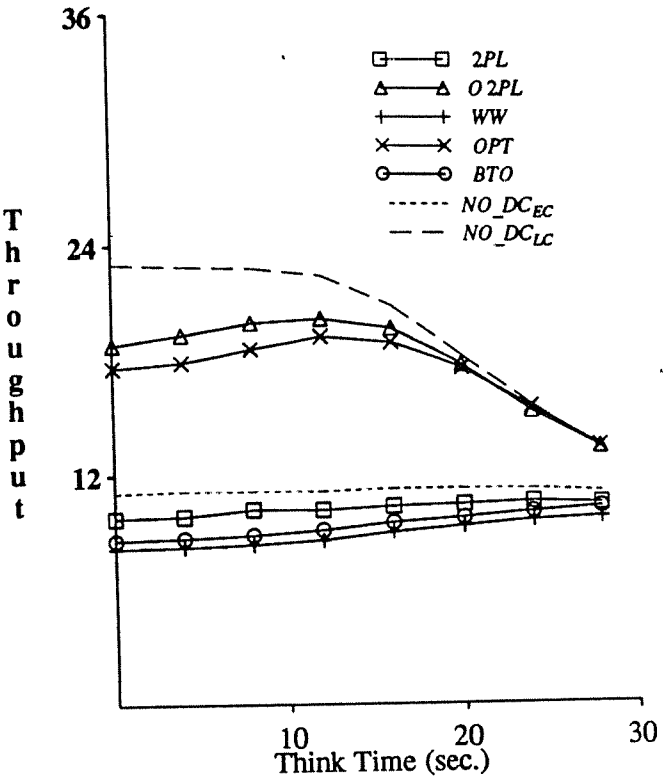

Figure 26: Throughput (transactions/sec.).
(MsgCPUTime = 10 ms, 3 copies, ALL active)

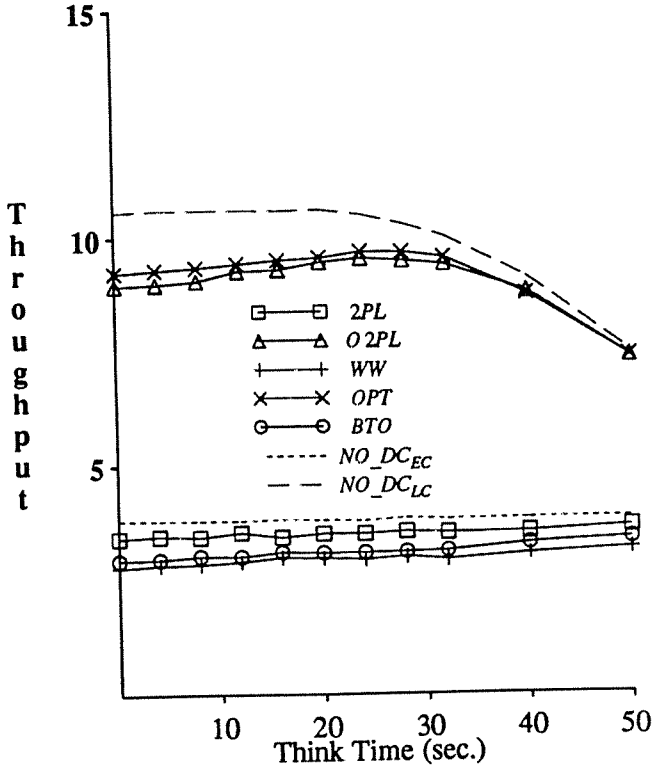

Figure 27: Throughput (transactions/sec.).
(MsgCPUTime = 10 ms, 8 copies, ALL active)

In terms of the relative performance of the algorithms, we found that 2PL, O2PL, and OPT dominated BTO and WW. When the cost of sending and receiving messages was low, 2PL was the superior performer due to its avoidance of transaction aborts. However, when the message cost was medium to 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 aborts was more than compensated for by the savings due to avoiding costly messages. However, in such cases, OPT was in turn outperformed by O2PL. O2PL was the best performer for replicated data under the higher message costs because it has the message characteristics of OPT together with 2PL's preference for blocking over aborts. Finally, BTO and WW performed fairly similarly in the experiments presented here.

In terms of replication, increasing the number of copies had the expected negative 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 O2PL 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, doing so was sufficient to move O2PL into the performance leadership position. In terms of replicated data accesses, we considered two workloads — a workload where one copy of each file was active, and a workload where all copies of each file were equally active; these two workloads delimit a range of possible access behaviors. O2PL was the only algorithm that displayed any significant sensitivity to this change in the workload, as moving from one active copy to having all copies active changes the point at which some update conflicts are detected in O2PL. 2PL performed a bit better than O2PL in the low message cost, all-copies-active experiments with replicated data. However, these differences were not all that large, and O2PL significantly outperformed 2PL when the message cost was medium to high. Our conclusion is that O2PL is quite robust, and it should perhaps be viewed as "2PL done right" with respect to dealing with replicated data, particularly in light of the cost of sending and receiving messages in a typical distributed system today.

To summarize, the two algorithms based on two-phase locking with deadlock detection (2PL and O2PL) dominated the other algorithms in our experiments. In the absence of replication, the two algorithms are identical. With replicated data, when multiple copies of a file were in use by transactions, we found that the algorithm of choice depended on the cost of inter-site communications. When the cost to send and receive messages was low, 2PL was slightly preferable because it resolves conflicts early and thus avoids possible end-of-transaction aborts. However,

when messages were more expensive, O2PL was clearly preferable. This is because it retains the advantages of 2PL for local accesses, while it takes a more optimistic (and significantly less expensive) approach to handling conflicts that arise on remote copies. Thus, an interesting tradeoff exists between the cost and timing of conflict detection in a distributed DBMS, a tradeoff that depends primarily on the efficiency of the communications subsystem.

In terms of future work, several interesting questions remain. First, we focused our attention on replication-related tradeoffs here, and did not explore the impact of intra-transaction parallelism (e.g., the kind found in parallel database machines). We have recently been investigating the question of how parallelism affects the performance of the distributed concurrency control algorithms examined here [Care89]. Second, we did not consider failures, or algorithms specifically designed to tolerate them, in this study. In the future we plan to study both the performance and the availability characteristics of failure-tolerant replica control schemes using our model.

ACKNOWLEDGEMENTS

The authors would like to thank Bruce Lindsay of IBM Almaden Research Center for encouraging us to study O2PL and for comments and criticisms that helped us to improve the presentation.

REFERENCES

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

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases," *Proc. 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. 1st ACM SIGACT-SIGOPS Symp. on Principles of Dist. Comp.*, 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. 6th VLDB Conf.*, Mexico City, Mexico, 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 in Replicated Distributed Databases," *ACM Trans. on Database Sys.* 8, 4, Dec. 1984.

[Bern87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., 1987.

[Bhar82] Bhargava, B., "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking," *Proc. 3rd Int'l. Conf. on Dist. Comp. Sys.*, Miami, FL, October 1982.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proc. 10th VLDB Conf.*, Singapore, Aug. 1984.

[Care86] Carey, M., and Lu, H., "Load Balancing in a Locally Distributed Database System," *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1986.

[Care88] Carey, M., and Livny, M., "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Care89] Carey, M., and Livny, M., "Parallelism and Concurrency Control Performance in Distributed Database Machines," *Proc. ACM SIGMOD Conf.*, Portland, OR, June 1989, to appear.

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

[DeWi86] DeWitt, D., et al, "GAMMA — A High Performance Backend Database Machine," *Proc. 12th VLDB Conf.*, Kyoto, Japan, Aug. 1986.

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

[ElAb85] El Abbadi, A., Skeen, D., and Cristian, F., "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," *Proc. 4th ACM Symp. on Principles of Database Sys.*, Portland, OR, March 1985.

[Gall82] Galler, B., *Concurrency Control Performance Issues*, Ph.D. Thesis, Comp. Sci. Dept., Univ. of Toronto, Sept. 1982.

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

[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.

[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 Comp. Eng., Univ. of Massachusetts, Amherst, 1985.

[Lamp78] Lamport, L., "Time, Clocks, and Ordering of Events in a Distributed System," *Comm. ACM* 21, 7, July 1978.

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

[Li87] Li, V., "Performance Model of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases," *IEEE Trans. on Comp.* C-36, 9, Sept. 1987.

[Lin82] Lin, W., and Nolte, J., "Performance of Two Phase Locking," *Proc. 6th Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks, Feb. 1982.*

[Lin83] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking," *Proc. 9th VLDB Conf.*, Florence, Italy, Nov. 1983.

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

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

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

[Noe87] Noe, J., and Wagner, D., "Measured Performance of Time Interval Concurrency Control Techniques," *Proc. 13th VLDB Conf.*, Brighton, England, Sept. 1987.

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

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

[Ries79] Ries, D., "The Effects of Concurrency Control on the Performance of a Distributed Data Management System," *Proc. 4th Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks*, Aug. 1979.

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

[Schl81] Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. 7th VLDB Conf.*, Cannes, France, Sept. 1981.

[Sinh85] Sinha, M., et al, "Timestamp Based Certification Schemes for Transactions in Distributed Database Systems," *Proc. ACM SIGMOD Conf.*, Austin, TX, May 1985.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. on Softw. 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 Sys.* 4, 2, June 1979.

[Trai82] Traiger, I., et al, "Transactions and Consistency in Distributed Database Systems," *ACM Trans. on Database Sys.* 7, 3, Sept. 1982.