Characterization and Optimization of Commit Processing Performance in Distributed Database Systems*

Jayant Haritsa † Krithi Ramamritham ‡ Ramesh Gupta †

† Database Systems Lab, SERC
Indian Institute of Science
Bangalore 560012, India

‡ Dept. of Computer Science
Univ. of Massachusetts
Amherst 01003, U.S.A.

Abstract

A significant body of literature is available on distributed transaction commit protocols. Surprisingly, however, the relative merits of these protocols have not been sufficiently studied with respect to their *quantitative* impact on transaction processing performance. Also, even though several optimizations have been suggested to improve the performance of the ubiquitous Two-Phase Commit (2PC) protocol, none have addressed the fact that under 2PC the data updated by a transaction during its data processing phase remains unavailable to other transactions during its commit processing phase and, worse, there is no inherent bound on the duration of this unavailability. This paper addresses both these issues.

First, using a detailed simulation model of a distributed database system, we profile the transaction throughput performance of a representative set of commit protocols, including 2PC, Presumed Abort, Presumed Commit and 3PC. Second, we propose and evaluate a new commit protocol, **OPT**, that allows transactions to "optimistically" borrow the updated data of transactions currently in their commit phase. This borrowing is controlled to ensure that cascading aborts, usually associated with the use of dirty data, do not occur. The new protocol is easy to implement and incorporate in current systems, and can coexist, often *synergistically*, with many of the optimizations proposed earlier including current industry standard protocols such as Presumed Commit and Presumed Abort.

The experimental results show that distributed *commit* processing can have considerably more influence than distributed *data* processing on the throughput performance and that the choice of commit protocol clearly affects the magnitude of this influence. Among the protocols evaluated, the new optimistic commit protocol provides the best transaction throughput performance for a variety of workloads and system configurations. In fact, OPT's peak throughput is often close to the upper bound on achievable peak performance. Even more interestingly, when data contention is significant, integrating OPT with the non-blocking *three-phase* commit protocol provides better peak throughput performance than all of the standard *two-phase* blocking protocols evaluated in our study. Further, OPT makes efficient use of the borrowing approach and its performance is robust for practical workloads. In short, OPT is a portable, practical, high-performance, efficient and robust distributed commit protocol.

Keywords: Distributed Database, Commit Protocol, Two Phase Commit, Three Phase Commit, Performance Evaluation

*A partial and preliminary version of the results presented here appeared earlier in *Revisiting Commit Processing in Distributed Database Systems*, Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Tucson, Arizona, May 1997.

Contents

1 Introduction **3**

2 Distributed Commit Protocols **5**
2.1 Two Phase Commit Protocol . . . 6
2.2 Presumed Abort . . . 7
2.3 Presumed Commit . . . 7
2.4 Three Phase Commit . . . 9
2.5 Other Commit Protocols . . . 9
2.6 Transaction and Cohort Execution Phases . . . 9

3 Optimistic Commit Processing **10**
3.1 Aborts in OPT do not Cascade . . . 11
3.2 Integrating Prior 2PC Optimizations with OPT . . . 12
3.3 System Integration . . . 12
3.3.1 Additional Memory Requirement due to the Hash Tables . . . 13

4 Simulation Model, Metrics, and Baseline Protocols **15**
4.1 Database Model . . . 17
4.2 Workload Model . . . 17
4.3 System Model . . . 17
4.4 Transaction Execution . . . 18
4.5 Concurrency Control . . . 18
4.6 Surprise Aborts . . . 18
4.7 Logging . . . 19
4.8 Performance Metric . . . 19
4.9 Baseline Protocols . . . 19

5 Experiments and Results **20**
5.1 Experiment 1: Resource and Data Contention . . . 20
5.2 Experiment 2: Pure Data Contention . . . 23
5.3 Experiment 3: Fast Network Interface . . . 24
5.4 Experiment 4: Higher Degree of Distribution . . . 26
5.5 Experiment 5: Non-Blocking OPT . . . 28
5.6 Experiment 6: Surprise Aborts . . . 29
5.7 Experiment 7: Partial-Update Transactions . . . 31
5.8 Experiment 8: Sequential Transactions . . . 32
5.9 Other Experiments . . . 33

6 Lending Variations of OPT **33**
6.1 Borrow-Bit OPT (BB-OPT) 34
6.2 Borrow-Message OPT (BM-OPT) 34
6.3 Performance of the Variants 35

7 Other Commit Protocols and Optimizations **37**
7.1 Protocols Satisfying Traditional Atomicity 37
7.2 Integration with OPT 38
7.3 Protocols Satisfying Relaxed Atomicity through Compensations 39

8 Related Performance Work **39**

9 Conclusions **41**
9.1 Future Work 43

1 Introduction

Distributed database systems implement a transaction *commit protocol* to ensure transaction atomicity. A commit protocol guarantees the uniform commitment of distributed transaction execution, that is, it ensures that all the participating sites agree on the final outcome (commit or abort). Most importantly, this guarantee is valid even in the presence of site or network failures.

Over the last two decades, a variety of commit protocols have been proposed by database researchers (see [Koh81, Bha87, OV91] for surveys). These include the classical *two phase commit (2PC)* protocol [Gra78, LS76], its variations such as *presumed commit* and *presumed abort* [LL93, MLO86], and *three phase commit (3PC)* [Ske81]. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately in a synchronous manner. Due to this series of synchronous message and logging overheads, commit processing can result in a significant increase in transaction response times [LL93, SBCM95, SJR91]. Therefore, *the choice of commit protocol is an important design decision for distributed database systems.*

In light of the above, it seems reasonable to expect that the results of detailed studies of commit protocol performance would be available to assist distributed database system designers in making an informed choice (as is the case with, for example, distributed concurrency control [CL88, CL89, CL91, FHRT93]). Surprisingly, however, most of the earlier performance studies of commit protocols (e.g., [AC95, LL93, MLO86, SBCM95]) have been limited to comparing protocols based on the *number* of messages and the *number* of log-writes that they incur. Thorough quantitative performance evaluation with regard to *overall* transaction processing metrics such as mean response time or peak throughput has, however, received virtually no attention. This is a significant omission since transaction processing performance is usually a primary concern for database system designers.

A second, and related, issue is that even though several optimizations have been suggested to reduce commit processing overheads, none have addressed the fact that the data updated by a transaction during its data processing phase remains unavailable to other transactions during the commit processing phase. This unavailability can result in increased response times for transactions, especially since, as discussed above, commit processing can take a significant (and not inherently bounded) amount of time.

The lack of quantitative performance studies and the absence of protocols that deal with the issue of updated data being inaccessible during commit processing represent significant lacunae in the published literature. Hence we address these issues in this paper.

Contributions

Within the above context, our contributions are two-fold:

1. Using a simulator based on a detailed closed queueing model of a distributed database system, we compare the throughput performance of a representative set of previously proposed commit protocols for a variety of distributed database workloads and system configurations. Both "blocking" (two-

phase) commit protocols, and "non-blocking" (three-phase) commit protocols, are included in the scope of our study. In blocking protocols, failures can lead to transaction processing grinding to a halt, whereas nonblocking protocols try to ensure that such major disruptions do not occur – to achieve this, however, they incur additional message and logging overheads.

To isolate and quantify the effects of supporting data distribution and achieving transaction atomicity on system performance, we use two baselines in the simulations: (1) a centralized system, and (2) a system wherein data processing is distributed but commit processing is centralized.

2. We propose and evaluate a new commit protocol, called **OPT**, that, in contrast to earlier commit protocols, allows executing transactions to "optimistically" *borrow* the updated data of transactions currently in their commit processing phase. Although this implies that dirty reads are permitted, there is no danger of incurring cascading aborts [BHG87] since the borrowing is done in a controlled manner. The protocol is easy to implement and to incorporate in current systems, and can be integrated, often *synergistically*, with most other optimizations proposed earlier.

Overall, our goal is to quantitatively investigate the implications, on the system throughput, of the protocol choice made for achieving transaction atomicity in a distributed database system. Accordingly, our focus is on the impact of the *mechanisms* required during *normal* (no-failure) operation to provide for recoverability, rather than on the post-failure recovery *process* itself.

Results

Salient observations from our simulation experiments include:

- Distributed *commit* processing can have considerably more effect than distributed *data* processing on the system performance.

- Among the commit protocols evaluated, the new OPT protocol provided the best overall performance in our experiments, performing considerably better than the classical protocols.[1] In fact, OPT's peak throughput was often close to that obtained with the "distributed processing, centralized commit" baseline mentioned above, which in a sense represents an upper bound on the achievable peak performance.

- Due perhaps to their increased overheads, non-blocking protocols such as three-phase commit (3PC) have not been used in real-world systems. However, our experiments show that under conditions where there is sufficient data contention in the system, a combination of OPT and 3PC provides better throughput performance than any of the standard two-phase blocking protocols. This suggests that it would be possible for distributed database systems that are operating in high contention situations and are currently using 2PC-based protocols to switch over to OPT-3PC, thereby obtaining the superior performance of OPT during normal processing and, in addition, acquiring the highly desirable non-blocking feature of 3PC. This, in essence, is a "win-win" situation.

[1] A suitably modified version of OPT exhibited similar good performance characteristics in our recent research on commit processing in distributed *real-time* database systems [GHRS96, GHR97].

- OPT's design is based on the assumption that transactions that lend their uncommitted data will almost always commit. Nevertheless, the performance of OPT is *robust* in that, even if transactions abort in the commit processing phase (due to violation of integrity constraints, software errors, etc.), OPT maintains its superior performance as long as the probability of such aborts does not exceed *fifteen percent*, a level that is much higher than what might be expected in practice.

Organization

The remainder of this paper is organized as follows: The classical distributed commit protocols evaluated in our study are described in Section 2, and the new optimistic commit protocol is presented in Section 3. The performance model is described in Section 4, and the results of the simulation experiments are highlighted in Section 5. Protocols based on variations of the OPT approach are presented and evaluated in Section 6. The relationship between OPT and other commit protocol optimizations is discussed in Section 7. Related work on performance aspects of commit protocols is reviewed in Section 8. Finally, in Section 9, we present the conclusions of our study and identify future research avenues.

2 Distributed Commit Protocols

A common model[2] of a distributed transaction is that there is one process, called the *master*, which is executed at the site where the transaction is submitted, and a set of other processes, called *cohorts*, which execute on behalf of the transaction at the various sites that are accessed by the transaction.[3] Cohorts are created by the master sending a STARTWORK message to the local transaction manager at that site. This message includes the work to be done at that site and is passed on to the cohort. Each cohort sends a WORKDONE message to the master after it has completed its assigned data processing work, and the master initiates the commit protocol (only) after it has received this message from all its cohorts. For this model, a variety of transaction commit protocols have been devised, most of which are based on the classical **two phase commit (2PC)** protocol [Gra78], described later in this section.

Before we describe the protocols, it is important to point out that, from a performance perspective, commit protocols can be compared on the following three issues:

Effect on Normal Processing: This refers to the extent to which the commit protocol affects the normal (no-failure) distributed transaction processing performance. That is, how expensive is it to provide atomicity using this protocol?

Resilience to Failures: When a failure occurs in a distributed system, ideally transaction processing in the rest of the system should not be affected during the recovery of the failed component. With most commit protocols, however, failures can lead to transaction processing grinding to a halt (as explained in Section 2.4), and they are therefore termed as "blocking protocols".

[2]An alternative model is the "peer-to-peer" environment, wherein *any* cohort in the transaction can decide to initiate a commit operation and thus become the root of the transaction commit tree [MD94, SBCM95].

[3]In the most general case, each of the cohorts may itself spawn off sub-transactions at other sites, leading to the "tree of processes" transaction structure of System R* [Lin84] – for simplicity, we only consider a two-level tree here.

To ensure that such major disruptions do not occur, efforts have been made to design "non-blocking commit protocols". These protocols, in the event of a site failure, permit transactions that had cohorts executing at the failed site to terminate at the operational sites without waiting for the failed site to recover [OV91, Ske81].[4] To achieve their functionality, however, they usually incur *additional* messages and forced-log-writes than their blocking counterparts.

In general, "two-phase" commit protocols are susceptible to blocking whereas "three-phase" commit protocols are non-blocking.

Speed of Recovery: This refers to the time required for the database to be recovered to a consistent state when the failed site comes back up after a crash. That is, how long does it take before transaction processing can commence again in a recovering site?

Of the three issues highlighted above, the design emphasis of most commit protocols has been on the first two issues (effect on normal processing and resilience to failures) since they directly affect ongoing transaction processing. In comparison, the last issue (speed of recovery) appears less critical for two reasons: First, failure durations are usually orders of magnitude larger than recovery times. Second, failures are usually rare enough that we do not expect to see a difference in *average* performance among the protocols because of one commit protocol having a faster recovery time than the other. Based on this viewpoint, this paper's focus is on the *mechanisms* required during normal (no-failure) operation to provide for recoverability, rather than on the post-failure recovery *process* itself.

In the remainder of this section, we briefly describe the 2PC protocol and a few popular variations of this protocol – complete descriptions are available in [MLO86, Ske81, SBCM95]. For ease of exposition, the following notation is used in the sequel – "SMALL CAPS FONT" for messages, "`typewriter font`" for log records, and "sans serif font" for transaction states.

2.1 Two Phase Commit Protocol

The two-phase commit protocol, as suggested by its name, operates in two phases: In the first phase, called the "voting phase", the master reaches a global decision (commit or abort) based on the local decisions of the cohorts. In the second phase, called the "decision phase", the master conveys this decision to the cohorts. For its successful execution, the protocol assumes that each cohort of a transaction is able to *provisionally* perform the actions of the transaction in such a way that they can be undone if the transaction is eventually aborted. This is usually implemented by using *logging* mechanisms such as WAL (write-ahead-logging) [Gra78], which maintain sequential histories of transaction actions in stable storage. The protocol also assumes that, if necessary, log records can be *force-written*, that is, written synchronously to stable storage.

After receiving the WORKDONE message from all the cohorts participating in the distributed execution of the transaction, the master initiates the first phase of the commit protocol by sending PREPARE (to commit) messages in parallel to all its cohorts. Each cohort that is ready to commit first force-writes a

[4] It is impossible to design commit protocols that are completely non-blocking to both site and link failures [BHG87]. However, the number of simultaneous failures that can be tolerated before blocking arises depends on the protocol design.

prepare log record to its local stable storage and then sends a YES vote to the master. At this stage, the cohort has entered a prepared state wherein it cannot unilaterally commit or abort the transaction but has to wait for the final decision from the master. On the other hand, each cohort that decides to abort force-writes an abort log record and sends a NO vote to the master. Since a NO vote acts like a veto, the cohort is permitted to unilaterally abort the transaction without waiting for the final decision from the master.

After the master receives the votes from all its cohorts, the second phase of the protocol is initiated. If all the votes are YES, the master moves to a committing state by force-writing a commit log record and sending COMMIT messages to all its cohorts. Each cohort, upon receiving the COMMIT message, moves to the committing state, force-writes a commit log record, and sends an ACK message to the master.

If the master receives even one NO vote, it moves to the aborting state by force-writing an abort log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to the aborting state, force-write an abort log record and send an ACK message to the master.

Finally, the master, after receiving ACKs from all the prepared cohorts, writes an end log record and then "forgets" the transaction (by removing from virtual memory all information associated with the transaction). The above sequence of actions is shown pictorially in Figure 1.

2.2 Presumed Abort

As described above, the 2PC protocol requires transmission of several messages and writing or force-writing of several log records. A variant of the 2PC protocol, called **presumed abort (PA)** [MLO86], tries to reduce these message and logging overheads by requiring all participants to follow, during failure recovery, an "in the no-information case, abort" rule. That is, if after coming up from a failure a site queries the master about the final outcome of a transaction and finds no information available with the master, the transaction is (correctly) assumed to have been aborted. With this assumption, it is not necessary for cohorts to send ACKs for ABORT messages from the master, or to force-write the abort record to the log. It is also not necessary for an aborting master to force-write the abort log record or to write an end log record.

In short, the PA protocol behaves identically to 2PC for committing transactions, but has reduced message and logging overheads for aborted transactions.

2.3 Presumed Commit

A variation of the presumed abort protocol is based on the observation that, in general, the number of committed transactions is much more than the number of aborted transactions. In this variation, called **presumed commit (PC)** [MLO86], the overheads are reduced for *committing* transactions, rather than aborted transactions, by requiring all participants to follow, during failure recovery, an "in the no-information case, commit" rule. In this scheme, cohorts do not send ACKs for a commit decision sent from the master, and also do not force-write the commit log record. In addition, the master does not write an end log record. On the down side, however, the master is required to force-write a collecting log record

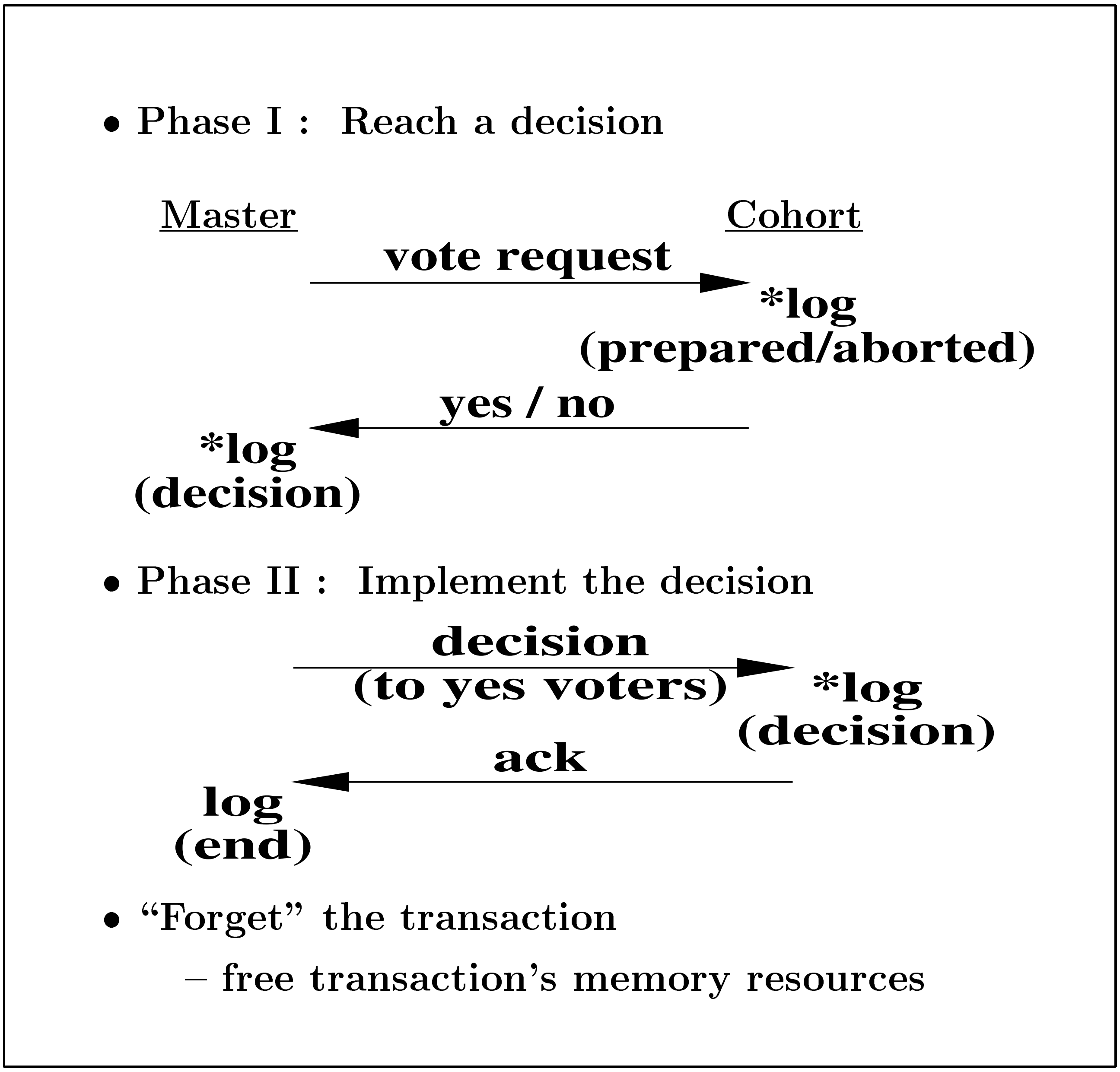

Figure 1: Two-Phase Commit Protocol (* indicates a *forced write*)

before initiating the two-phase protocol. This log record contains the names of all the cohorts involved in executing that transaction.

The above optimizations of 2PC have been implemented in a number of database products including Tandem's TMF, DEC's VAX/VMS, Transarc's Encina and USL's TUXEDO. In fact, PA is now part of the ISO-OSI and X/OPEN distributed transaction processing standards [MD94, SBCM95].

2.4 Three Phase Commit

A fundamental problem with all of the above protocols is that cohorts may become *blocked* in the event of a site failure and remain blocked until the failed site recovers. For example, if the master fails *after* initiating the protocol but *before* conveying the decision to its cohorts, these cohorts will become blocked and remain so until the master recovers and informs them of the final decision. During the blocked period, the cohorts may continue to hold system resources such as locks on data items, making these unavailable to other transactions. These transactions in turn become blocked waiting for the resources to be relinquished, resulting in "cascading blocking". It is easy to see that if the duration of the blocked period is significant, major disruption of transaction processing activity could result.

To address the blocking problem, a **three phase commit (3PC)** protocol was proposed in [Ske81]. This protocol achieves a non-blocking capability by inserting an extra phase, called the "precommit phase", in between the two phases of the 2PC protocol. In the precommit phase, a preliminary decision is reached regarding the fate of the transaction. The information made available to the participating sites as a result of this preliminary decision allows a global decision to be made despite a subsequent failure of the master site. Note, however, that the price of gaining non-blocking functionality is an increase in the communication and logging overheads since (a) there is an extra round of message exchange between the master and the cohorts, and (b) both the master and the cohorts have to force-write additional log records in the precommit phase.

2.5 Other Commit Protocols

The above-mentioned protocols are well-established and have received the most attention in the literature – we therefore concentrate on them in our study. It should be noted, however, that a variety of other protocols have also been proposed. We defer the discussion of these protocols to Section 7.

2.6 Transaction and Cohort Execution Phases

As described above, commit protocols typically operate in two or three phases. For ease of exposition, we will similarly also divide the *overall execution* of transactions and of individual cohorts into phases.

A transaction's execution is composed of two phases: the "data phase" and the "commit phase". The data phase begins with the sending of the first STARTWORK message by the master and ends when all the WORKDONE messages have been received, that is, it captures the data processing period. The commit phase begins with the sending of the PREPARE messages and ends when the master forgets the transaction, that is, it captures the commit processing period.

A cohort's execution is composed of three phases: (1) the "data phase" wherein the cohort carries out its locally assigned data processing – it begins with the receipt of the STARTWORK message from the master and ends with the sending of the WORKDONE message to the master; (2) the "commit phase" which begins with the cohort receiving the PREPARE message and ends with the last commit-related action taken by the cohort (this is a function of the commit protocol in use): (3) the "wait phase" that denotes the time period in between the "data phase" and the "commit phase", that is, the time period between sending the WORKDONE message and receiving the PREPARE message.

3 Optimistic Commit Processing

In all of the protocols described in the previous section, a cohort that reaches the prepared state can release all of its read locks.[5] However, the cohort has to retain all its *update* locks until it receives the global decision from the master – this retention is fundamentally necessary to maintain atomicity. More importantly, the lock retention interval is *not bounded* since the time duration that a cohort is in the prepared state can be arbitrarily long (for example, due to network delays). If the retention period is large, it may have a significant negative effect on performance since other transactions that wish to access this prepared data are forced to block until the commit processing is over. It is important to note that this "prepared data blocking" is *distinct* from the "decision blocking" (because of failures) that was discussed in Section 2.4. That is, in all the commit protocols, *including 3PC*, transactions can be affected by prepared data blocking. In fact, 3PC's strategy for removing decision blocking *increases* the duration of prepared data blocking. Moreover, such data blocking occurs during normal processing whereas decision blocking only occurs during failure situations.

To address the above issue of prepared data blocking, we have designed a new version of the 2PC protocol in which transactions requesting data items held by other transactions in the prepared state are allowed to access this data. That is, prepared cohorts *lend* their uncommitted data to concurrently executing transactions. The mechanics of the interactions between such "lenders" and their associated "borrowers" are captured in the following three scenarios, only one of which will occur for each lending:

1. **Lender Receives Decision Before Borrower Completes Data Processing:**
   Here, the lending cohort receives its global decision before the borrowing cohort has completed its local data processing. If the global decision is to commit, the lending cohort completes in the normal fashion. On the other hand, if the global decision is to abort, the lender is aborted in the normal fashion. In addition, the borrower is also aborted since it has utilized dirty data.

2. **Borrower Completes Data Processing Before Lender Receives Decision:**
   Here, the borrowing cohort completes its local data processing before the lending cohort has received its global decision. The borrower is now "put on the shelf", that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the (commit-related) processing that could eventually lead to its reaching the prepared state.

[5] The release can occur as soon as the PREPARE message is received and does not have to wait until the `prepare` log record is force-written to disk.

Instead, it has to wait until the lender receives its global decision. If the lender commits, the borrower is "taken off the shelf" (if it has no other "pending" lenders) and allowed to send its WORKDONE message. On the other hand, if the lender aborts, the borrower is also aborted immediately since it has utilized dirty data (as in Scenario 1 above).

3. Borrower Aborts Before Lender Receives Decision:
Here, the borrowing cohort aborts (due to a local problem or due to receiving an ABORT message from its master) before the lending cohort has received its global decision. This could happen either during the borrowing cohort's data processing or while it is waiting on the shelf (i.e., Scenario 2 above). In this situation, the borrower's updates are undone and the lending is nullified.

In summary, the above protocol allows transactions to access uncommitted data held by prepared transactions in the "optimistic" belief that this data will eventually be committed.[6] We will hereafter refer to this protocol as **OPT**.

3.1 Aborts in OPT do not Cascade

An important point to note here is that OPT's policy of using uncommitted data is generally *not* recommended in database systems since this can potentially lead to the well-known problem of *cascading aborts* [BHG87] if the transaction whose dirty data has been accessed is later aborted. However, for the OPT protocol, this problem is alleviated due to two reasons:

1. The lending transaction is typically expected to commit because (a) the lending cohort is in prepared state and cannot be aborted due to local data conflicts, and (b) the sibling cohorts are also expected to eventually vote to commit since they have survived all the data conflicts that occurred prior to their sending of the WORKDONE messages.[7] In fact, if we assume that a locking-based concurrency control mechanism such as 2PL [Esw76] is used, it is easy to verify that there is *no possibility* of sibling cohorts aborting, during the commit processing period, due to serializability considerations. Therefore, an abort vote can arise only due to *other reasons* such as violation of integrity constraints, software errors, system failure, etc. We will hereafter use the term "surprise aborts" to refer to these kinds of aborts occurring in the commit phase.

2. Even if the lending transaction does eventually abort, it only results in the abort of the immediate borrower and does not cascade beyond this point (since the borrower is not in the prepared state, the only situation in which uncommitted data can be accessed). That is, a borrower cannot *simultaneously* be a lender, as shown in Figure 2. Therefore, the abort chain is bounded and is of length one. Of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the length of each abort *chain* is limited to one. In short, OPT implements a *controlled* lending policy.

[6]Similar, but unrelated, strategies of allowing access to uncommitted data have been used earlier for developing improved *concurrency control* protocols (see, for example, [AA90, RS96]).

[7]This expectation is not valid in conjunction with concurrency control protocols that *delay* conflict resolution – for example, validation-based techniques.

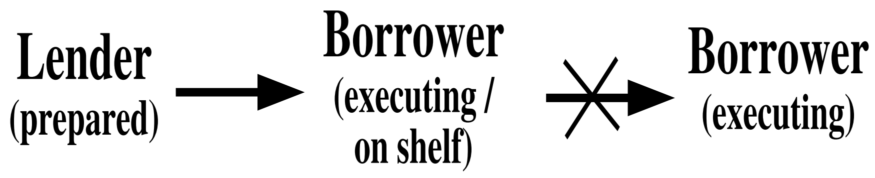

Figure 2: Aborts do not Cascade in OPT

3.2 Integrating Prior 2PC Optimizations with OPT

The optimistic premise underlying the OPT protocol was described above. In addition to this feature, other optimizations can also be incorporated in OPT, as described below:

Presumed Abort/Commit: The optimizations of Presumed Commit or Presumed Abort discussed earlier in Section 2 can also be used in conjunction with OPT to reduce the protocol overheads. We consider both options in our experiments.

Nonblocking OPT: Our description of OPT above assumed a 2PC protocol as the basis. However, the OPT approach can be applied directly to the 3PC protocol as well. We evaluate the performance of this protocol also in our experiments.

An interesting observation is that it is *especially attractive* to integrate OPT with 3PC. This is because, as mentioned earlier, 3PC in its attempt to remove decision blocking suffers an *increase* in the prepared data blocking period. Therefore, OPT can play a greater role in improving system performance by reducing the effects of this drawback.

Other Optimizations: Apart from the above, a variety of other optimizations have been proposed for the 2PC protocol – we defer their discussion to Section 7. As we will show there, many of these optimizations can be easily integrated with OPT as a useful supplement (as in the combination with PA/PC), and often the combination is *synergistic* (as in the combination with 3PC).

3.3 System Integration

We now comment on the *implementation* issues that arise with regard to incorporating the OPT protocol in a distributed database system. The important point to note here is that the required modifications are *local to each site* and do not require inter-site communication or coordination.

1. For a borrower cohort that finishes its data processing before its lenders have received their decisions from their masters, the *local transaction manager* must not send a WORKDONE message until the fate of all its lenders is determined.

2. When a lender is aborted and consequently its borrowers are also aborted, the *local transaction manager* should ensure that the actions of the borrowers are undone *first* and only then are the

updates of the associated lender undone – that is, the recovery manager should be invoked in a "borrowers first, lender next" sequence.

Note that in the event of a system crash, the log records will *naturally* be processed in the above order since the log records of lenders will always precede those of the borrowers in the sequential log and since the log is always scanned backwards during undo processing.

3. The *local lock manager* must be modified to permit borrowing of data held by prepared cohorts. The lock mode used by the borrowing cohort should become the current lock mode of the borrowed data item as far as other executing transactions are concerned.

4. The *local lock manager* must keep track of the lender-borrower relationships. This information will be needed to handle all possible outcomes of the relationship (for example, if the lender aborts, the associated borrowers must be immediately identified and also aborted), and can be maintained using the following data structures:

   Lender Hash Table (LHT): A hash table with the lender's transaction ID as the key and the IDs of the borrowers stored in the associated table entries. This hash table is for handling the situation wherein the borrower is alive until the lender receives its decision.

   Borrower Hash Table (BHT): A hash table with the borrower's transaction ID as the key and the IDs of the lenders stored in the associated table entries. This hash table is for handling the situation wherein the borrower aborts before the lender receives its decision.

   Note that in all cases updates to both tables are made, with the only situation-specific factor being whether the lender–borrower associations are initially found using the LHT or the BHT. Also note that both the tables are required to handle situations wherein a single borrower has borrowed from *multiple* lenders.

The above modifications do not appear difficult to incorporate in current database system software. In fact, some of them are *already provided* in current DBMS – for example, the high-performance industrial-strength ARIES recovery system [Moh92] implements *operation logging* to support semantically rich lock modes that permit updating of uncommitted data. Moreover, as shown later in our experiments, the performance benefits that can be derived from these changes more than compensate for the small amount of run-time overheads entailed by the above modifications and the effort needed to implement them.

In the rest of this section, we discuss the space overheads incurred by OPT because of the need for the two hash tables, LHT and BHT, and show that for practical environments these hash tables easily fit into memory at each site.

3.3.1 Additional Memory Requirement due to the Hash Tables

An important point to bear in mind in the following discussion is that even if a borrower borrows *multiple* data items from the same lender, only *one* entry is maintained in the hash tables since the subsequent processing is only dependent on the final disposition of the transactions, not the magnitude of borrowing.

We first derive an upper bound on $\mathcal{B}_{max}$, the maximum number of concurrent borrowings at each site. In this derivation, we use MPL to denote the multi-programming level at the site, C_m, the maximum cohort size (measured in number of data items), and p, the number of cohorts simultaneously in the prepared state.

The maximum number of borrowings occurs when all the $(MPL - p)$ executing cohorts borrow from all of the p prepared cohorts. The number of borrowings in this case is $(MPL - p) * p$, and it is easy to verify that this product is maximized when $p = MPL/2$. Therefore, the upper bound on the number of concurrent borrowings is $\mathcal{B}_{max} = MPL^2/4$.

The above bound is valid for the entire MPL range and, in particular, for the disjoint range pair $0 \le MPL \le 2 * C_m$ and $2 * C_m < MPL < \infty$. Note, however, that we can derive a tighter bound for the second range $(2 * C_m < MPL < \infty)$ based on the observation that no cohort can borrow more than C_m items. In this range the number of borrowings is maximized when there are exactly C_m prepared transactions and the corresponding bound is $\mathcal{B}_{max} = (MPL - C_m) * C_m$. This is because if one of the C_m prepared transactions is shifted to the executing group, the maximum number of borrowings would then be $(MPL - C_m + 1) * (C_m - 1) = (MPL - C_m) * C_m - (MPL - 2 * C_m + 1)$, and since $MPL > 2 * C_m$ the new product is lower. By induction this can be proved for any other combination of executing and prepared transactions.

Therefore, taking the above bounds in conjunction, we have

$$\mathcal{B}_{max} = \begin{array}{cc} MPL^2/4 & 0 \le MPL \le 2 * C_m \\ (MPL - C_m) * C_m & 2 * C_m < MPL < \infty \end{array}$$

With the above bounds, note that even if we assume an MPL as high as 1000 transactions per site, assume C_{max} to be as high as 32 data items, and assume that each entry in the hash tables takes 8 bytes of memory (4 for the cohort's id and 4 for the pointer to the next entry in the hash chain), the total memory requirement works out to less than 256 KB, which can be easily handled by most current systems.

In practice, the *average* number of concurrent borrowings per site, $\overline{\mathcal{B}}$, can be expected to be significantly lower than the above upper bounds, as shown by the following estimator, based on Little's Law [Tri82] and simple combinatorics:

$$\overline{\mathcal{B}} = (MPL - p) * p * c$$

where

$$p = MPL * \frac{AvgPreparedStateDuration}{AvgResponseTime}$$

is the average number of prepared cohorts, and

$$c = 1 - \frac{1}{D\begin{pmatrix} D+C-1 \\ C \end{pmatrix}} \sum_{max(0,D-C)}^{D-1} k \begin{pmatrix} D \\ k \end{pmatrix} (D-k)^{C-(D-k)}$$

is the probability of data conflict between an executing transaction and a prepared transaction. In the formula for c, C denotes the average cohort size, and $\frac{1}{D} = \frac{C * U}{N}$ where N is the database size and U is the update probability, denotes the fraction of prepared data items in the database.[8]

[8] Recall that locks are held *only* on updated items during the prepared state.

Table 1: Simulation Model Parameters

DBSize	Number of pages in the database
NumSites	Number of sites in the database
MPL	Transaction multiprogramming level / site
TransType	Transaction Type (Sequential or Parallel)
DistDegree	Degree of Distribution (number of cohorts)
CohortSize	Average cohort size (in pages)
UpdateProb	Page update probability
NumCPUs	Number of processors per site
NumDataDisks	Number of data disks per site
NumLogDisks	Number of log disks per site
PageCPU	CPU page processing time
PageDisk	Disk page access time
BufHit	Probability of finding a requested page in the buffer
MsgCPU	Message send / receive time
SurpriseAbort	Probability of surprise abort

For the database environments modeled in our experiments, the above value of $\overline{\mathcal{B}}$ was typically an order of magnitude smaller than the corresponding value of the $\mathcal{B}_{max}$ upper bound.

4 Simulation Model, Metrics, and Baseline Protocols

To evaluate the performance of the various commit protocols described in the previous sections, we developed a detailed simulator based on a closed queueing model of a distributed database system. Our simulation model is similar to the one used in [CL88] to study distributed concurrency control protocols.

The model consists of a database that is distributed, in a non-replicated manner, over N sites connected by a network. Figure 3 shows the general structure of the model. Each site has six components: a *source* which generates transactions; a *transaction manager* which models the execution behavior of the transaction; a *concurrency control manager* which implements the concurrency control algorithm; a *resource manager* which models the physical resources; a *recovery manager* which implements the details of commit protocols; and a *sink* which collects statistics on the completed transactions. In addition, a *network manager* models the behavior of the communication network.

The following subsections describe the database model, the workload generation process and the hardware resource configuration. Subsequently, we describe the execution pattern of a typical transaction and the policies adopted for concurrency control and recovery. A summary of the parameters used in the model is given in Table 1.

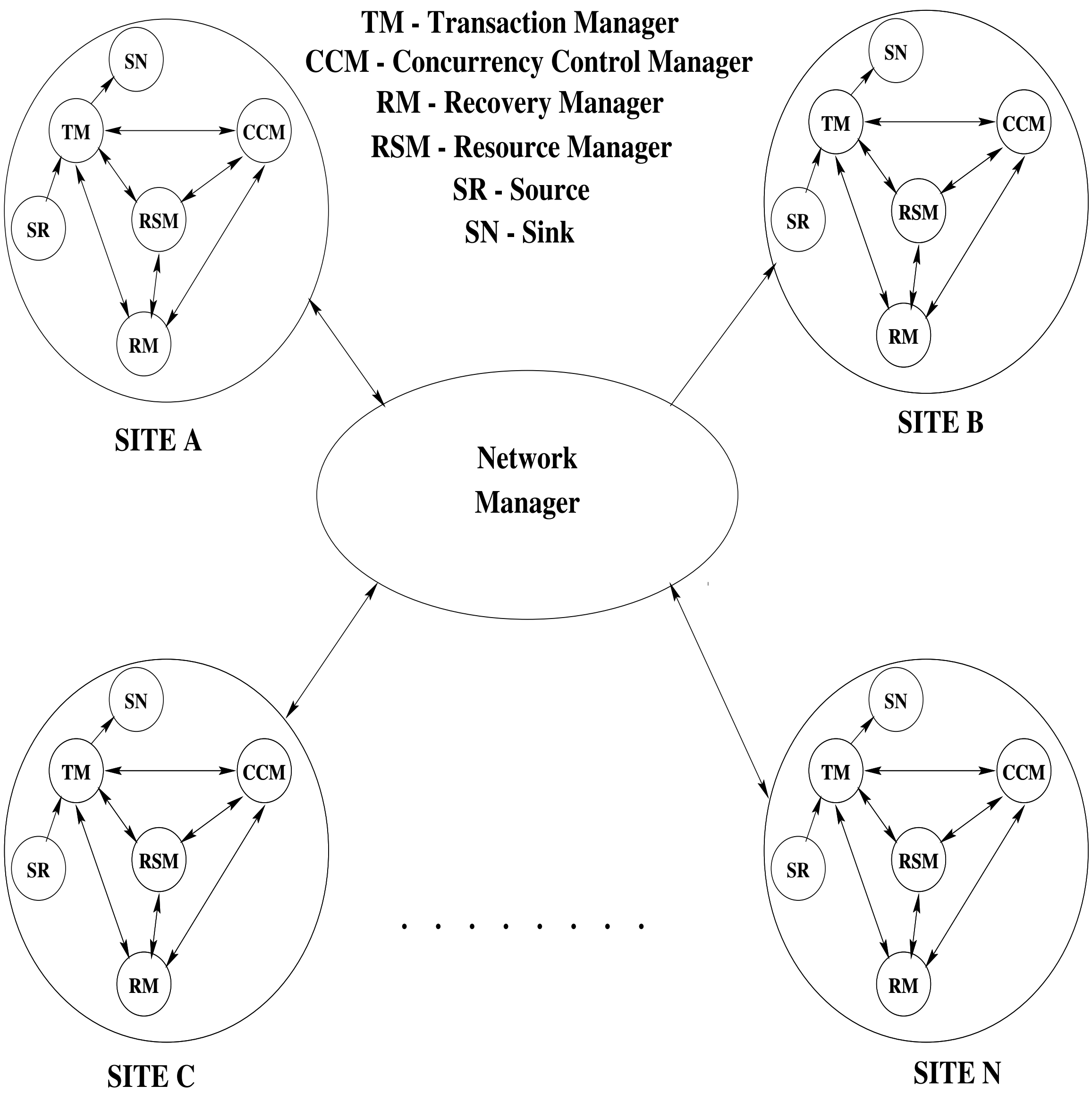

Figure 3: Distributed DBMS Model Structure

4.1 Database Model

The database is modeled as a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. Transactions make requests for data pages and concurrency control is implemented at the page level.

4.2 Workload Model

At each site, the transaction multiprogramming level is specified by the MPL parameter. Each transaction in the workload has the "single master – multiple cohort" structure described in Section 2. Transactions in a distributed system can execute in either *sequential* or *parallel* fashion. The distinction is that cohorts in a sequential transaction execute one after another, whereas cohorts in a parallel transaction are started together and execute independently until commit processing is initiated. We consider both types of transactions in our study – the parameter $TransType$ specifies whether the transaction execution is sequential or parallel.

The number of sites at which each transaction executes is specified by the $DistDegree$ parameter. The master and one cohort reside at the site where the transaction is submitted whereas the remaining $DistDegree - 1$ cohorts are set up at different sites chosen at random from the remaining $NumSites - 1$ sites. At each of the execution sites, the number of pages accessed by the transaction's cohort varies uniformly between 0.5 and 1.5 times $CohortSize$. These pages are chosen uniformly (without replacement) from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$.[9] A transaction that is aborted is restarted after a delay and makes the same data accesses as its original incarnation, that is, restarts are not "fake" [ACL87]. The duration of the delay is equal to the average transaction response time – this is the same heuristic as that used in most of the earlier transaction management studies [ACL87, CL88, CL89, CL91] – the goal of the heuristic is to ensure that the data conflict that resulted in the abort does not recur. Finally, after a transaction completes, a new one is submitted immediately at its originating site.

4.3 System Model

The physical resources at each site consist of $NumCPUs$ processors, $NumDataDisks$ data disks, and $NumLogDisks$ log disks. The data disks store the data pages while the log disks store the transaction log records. There is a single common queue for the processors whereas each of the disks has its own queue. All queues are processed in FCFS order except that at the CPUs message processing is given higher priority than data processing. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. When a transaction makes a request for accessing a data page, the data page may be found in the buffer pool, or it may have to be accessed from the disk. The $BufHit$ parameter gives the probability of finding a requested page already resident in the buffer pool.

The communication network is simply modeled as a switch that routes messages since we assume a local area network that has high bandwidth. However, the CPU overheads of message transfer, given by the

[9] A page write operation is always preceded by a read for the same page; that is, there are no "blind writes" [BHG87].

$MsgCPU$ parameter, are taken into account at both the sending and the receiving sites. This means that there are two classes of CPU requests – local data processing requests and message processing requests, and as noted above, message processing is given higher priority than data processing.

4.4 Transaction Execution

When a transaction is initiated, it is assigned the set of sites where it has to execute and the data pages that it has to access at each of these sites. The master is then started up at the originating site, forks off a local cohort and sends messages to initiate each of its cohorts at the remote participating sites.

Based on the transaction type, the cohorts execute either in parallel or in sequence. Each cohort makes a series of read and update accesses. A read access involves a concurrency control request to obtain access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. Update requests are handled similarly except for their disk I/O – the writing of the data pages takes place asynchronously after the transaction has committed.[10] We assume sufficient buffer space to allow the retention of data updates until commit time. The commit protocol is initiated when the transaction has completed its data processing.

4.5 Concurrency Control

For transaction concurrency control (CC), we use the distributed strict two-phase locking (2PL) protocol [BHG87]. Transactions, through their cohorts, set read locks on pages that they read and update locks on pages that need to be updated. All locks are held until the receipt of the PREPARE message from the master. Subsequently, the cohort releases all its read locks but retains its update locks until it receives and implements the global decision from the master. For the new protocol, OPT, however, the lock manager at each site is modified to permit borrowing of updated data items held by prepared transactions.

Since 2PL is used for concurrency control, deadlocks are a possibility, of course. In our simulation implementation, both global and local deadlock detection is immediate, that is, a deadlock is detected as soon as a lock conflict occurs and a cycle is formed. The youngest transaction in the cycle is restarted to resolve the deadlock. We do not explicitly model the overheads for detecting deadlocks or for concurrency control since (a) these costs would be similar across all the commit protocols, and (b) they are usually negligible compared to the overall cost of accessing data [CL88].

4.6 Surprise Aborts

With the 2PL CC mechanism described above, there is no possibility of serializability-induced aborts occurring in the *commit phase*, as mentioned in Section 3.1. To model other sources of commit phase aborts, i.e., surprise aborts, we allow each cohort, on receiving the PREPARE message from the master to randomly vote NO, instead of always voting YES. The probability of a NO vote is determined by the $SurpriseAbort$ parameter.

[10] Update-locks are acquired when a data page intended for modification is first read, i.e., lock upgrades are not modeled.

4.7 Logging

With regard to logging costs, we explicitly model only *forced* log writes since they are done synchronously and suspend transaction operation until their completion. The cost of each forced log write is the same as the cost of writing a data page to the disk. The overheads of flushing the transaction log records related to *data processing* (i.e., WAL [Gra78]), however, are not modeled. This is because these records are generated during the cohort's data phase and are therefore *independent* of the choice of commit protocol. We therefore do not expect their processing to affect the *relative* performance behavior of the commit protocols evaluated in our study.

4.8 Performance Metric

The primary performance metric of our experiments is *transaction throughput*, that is, the rate at which the system completes transactions.[11] Note that this is a "end-user metric", that is, it reflects the system performance as perceived by the customers of the distributed database system – in our opinion, such metrics are more relevant than the processing and message complexity-based evaluation used in earlier studies.

We also emphasize the *peak* throughput achieved by each protocol since this represents its maximum attainable performance and, by using a suitable admission control policy (for example, Half-and-Half [CKL90]), the throughput can be maintained at this level in high-performance systems. All the throughput graphs in this paper show mean values that have relative half-widths about the mean of less than 10 percent at the 90 percent confidence level, with each experiment having been run until at least 50000 transactions were processed. Only statistically significant differences are discussed here.

The simulator was instrumented to generate a number of other statistical information, including lock holding times, number of messages and force-writes, resource utilizations, transaction restarts, etc. These secondary measures help to explain the throughput behavior of the commit protocols under various workloads and system conditions.

4.9 Baseline Protocols

To help isolate and understand the effects of distribution and atomicity on throughput performance, and to serve as a basis for comparison, we have also simulated the performance behavior of two additional scenarios, CENT and DPCC, described below.

In **CENT** (Centralized), a *centralized* database system that is equivalent (in terms of overall database size and number of physical resources) to the distributed database system is modeled. Messages are obviously not required here and commit processing only requires writing a *single* decision log record (force-write if the decision is to COMMIT). Modeling this scenario helps to isolate the overall effect of distribution on throughput.

In **DPCC** (Distributed Processing, Centralized Commit), data processing is executed in the normal distributed fashion, that is, involving messages. The *commit* processing, however, is like that of a cen-

[11] Since we are using a closed queueing model, the inverse relationship between throughput and response time (as per Little's Law [Tri82]) makes either a sufficient performance metric.

Table 2: Baseline Parameter Settings

$NumSites$	8	$NumCPUs$	2
$DBSize$	8000 pages	$NumDataDisks$	3
$TransType$	Parallel	$NumLogDisks$	1
$DistDegree$	3	$PageCPU$	5 ms
$CohortSize$	6 pages	$PageDisk$	20 ms
$UpdateProb$	1.0	$BufHit$	0.1
MPL	1 – 10 per site	$MsgCPU$	5 ms
$SurpriseAbort$	0		

tralized system, requiring only the writing of the decision log record at the master. While this system is clearly artificial, modeling it helps to isolate the effect of distributed commit processing on throughput performance (as opposed to CENT which eliminates the entire effect of distributed processing).

5 Experiments and Results

Using the distributed database model described in the previous section, we conducted an extensive set of simulation experiments comparing the performance of the 2PC, PA, PC, 3PC and OPT commit protocols, which were presented in Sections 2 and 3. The simulator was written in C++SIM [LM94], an object-oriented simulation testbed, and the experiments were conducted on UltraSparc workstations running Solaris 2.5 operating system. In this section, we present the results of a variety of experiments and describe the performance profiles of the commit protocols.

5.1 Experiment 1: Resource and Data Contention

We began our performance evaluation by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. The settings of the workload parameters and system parameters for our first experiment are listed in Table 2. These settings were chosen to ensure significant levels of both resource contention (RC) and data contention (DC) in the system, thus helping to bring out the performance differences between the various commit protocols.

In this experiment, each transaction executes in a *parallel* fashion at three sites, accessing and updating an average of six pages at each site. Each site has two CPUs, three data disks and one log disk. The CPU and disk processing times are such that the system operates in an I/O-bound region. However, since it is not heavily I/O-bound, it is possible for message-related CPU costs to shift the system into a region of CPU-bound operation – this occurs, for example, in Experiment 4 where a higher degree of transaction distribution, and consequently more network activity, is modeled. Also, there are no surprise aborts in the commit phase.

For this experiment, Figure 4a presents the transaction throughput results as a function of the per-site multiprogramming level. We first observe here that for all the protocols, the throughput initially increases as the MPL is increased, and then it decreases. The initial increase is due to the fact that better performance is obtained when a site's CPUs and disks are utilized in parallel. As the multiprogramming level is increased beyond a certain level, however, the system starts thrashing due to data contention, resulting in degraded transaction throughput. This behavior is similar to those observed in earlier transaction management studies (e.g. [ACL87, CL88, CL89, CL91, FHRT93]).

Comparing the relative performance of the protocols in Figure 4a, we observe that the centralized system (CENT) performs the best, as expected, and that the peak performance of distributed processing/centralized commit (DPCC) is close to that of CENT. In contrast, there is a noticeable difference between the performance of these baseline systems and the performance of the classical commit protocols (2PC, PA, PC, 3PC) throughout the loading range. This demonstrates that distributed *commit processing* can have considerably more effect (difference between the performance of DPCC and 2PC) than distributed *data processing* (difference between the performance of CENT and DPCC) on the peak throughput performance, as will be seen in our other experiments as well.

Moving on to the relative performance of 2PC and 3PC, we observe that there is a clear difference in their performance. The difference arises from the additional message and logging overheads involved in 3PC, which are shown in Table 3 (for committing transactions).

Table 3: Protocol Overheads (DistDegree = 3)

Protocol	Execution	Commit	
	Messages	Forced-Writes	Messages
2PC	4	7	8
PA	4	7	8
PC	4	5	6
3PC	4	11	12
DPCC	4	1	0
CENT	0	1	0

Shifting our focus to PC, we observe that it performs very similarly to 2PC. This is because of the following reasons: First, while PC saves on the forced writes and acknowledgements at each cohort, it incurs an additional *collecting* forced write at the master. Second, the forced writes that PC saves were done in *parallel* at the cohorts by 2PC. Therefore, even though PC saves considerably on the overheads it fails to appreciably reduce the *overall* response time of the transaction. Thus the gain in throughput for PC over 2PC is not significant.

There is no possibility of serializability-induced aborts in the commit phase, as mentioned earlier in Section 4.5, due to using distributed strict 2PL as the CC mechanism. In the absence of any other source of aborts, as in this experiment, PA reduces to 2PC and performs *identically*. We will therefore defer further discussion of the performance of PA until Experiment 6 where we model "surprise aborts" of transactions in the commit phase.

Finally, turning our attention to the new protocol, OPT, we observe that its performance is either the same or better than that of all the standard algorithms over the full range of MPL. At low MPLs, when there is less data contention, and consequently little opportunity for borrowing, OPT is virtually identical

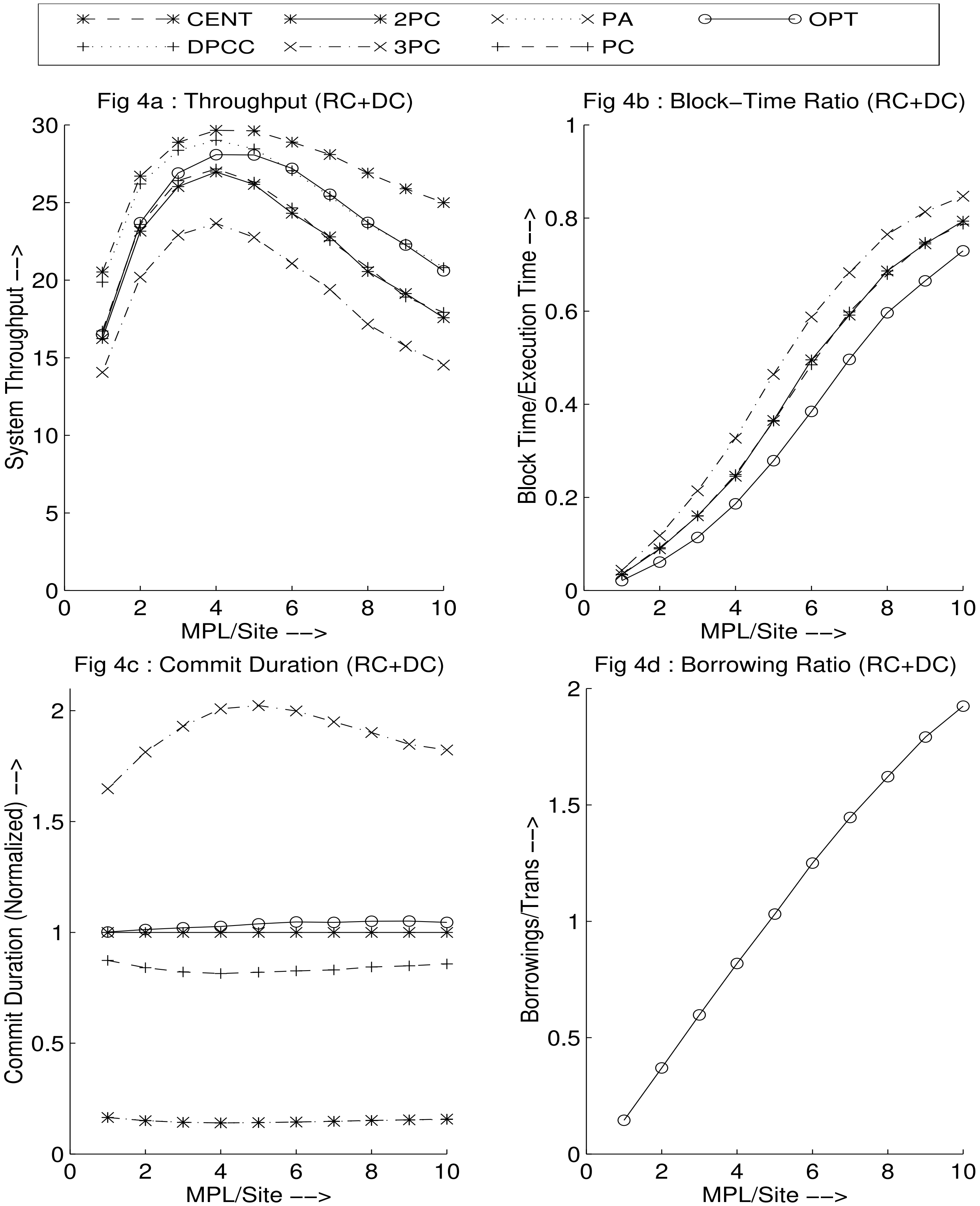

Figure 4: Baseline Experiment (RC + DC)

to 2PC and therefore performs at the same level. At higher MPLs, however, the performance of OPT is superior to that of 2PC, and in fact, becomes close to that of DPCC. The reason for this is that in OPT, the cohorts in the prepared state do not contribute to data contention thus reducing the number of blocked transactions. This is explicitly seen in Figure 4b, which presents the "block-time ratio", that is, the average ratio of a cohort's data blocking time (cumulative time during which the cohort is data-blocked) to its data phase duration (see 2.6). The graph shows that with OPT, transactions spend appreciably less time waiting for the data locks as compared to 2PC and 3PC.

In Figure 4c, the average duration of the commit phase is plotted for each protocol, *normalized* with respect to the average commit phase duration of 2PC. Here we see that, as expected, 3PC takes almost twice as much time for commit processing as 2PC, whereas PC is slightly faster than 2PC because of its reduced overheads. Further, the time taken by CENT and DPCC is negligible in comparison. An interesting observation in this picture is that OPT's commit duration is usually slightly *more* than that of 2PC. The reason for this is that OPT, by virtue of its higher concurrency, creates longer queues at the physical resources – therefore, transactions in the commit phase have to wait longer in these resource queues. The important point to note, however, is that in spite of this increase in commit duration, the decrease in *blocking duration* (which was shown in Figure 4b), is large enough to result in OPT performing much better with respect to the throughput metric (which captures the net effect of the data processing and commit processing durations).

Finally, in Figure 4d, we plot the "borrow ratio" for OPT, that is, the average number of data items (pages) borrowed per transaction. This graph clearly shows that borrowing comes into the picture at higher MPLs, resulting in improved performance for OPT.

In summary, OPT performs similar to 2PC and its variants under low data contention and much better than 2PC and its variants under high data contention.

5.2 Experiment 2: Pure Data Contention

The goal of our next experiment was to isolate the influence of *data contention* (DC) on the performance of the commit protocols. Modeling this scenario is important because while resource contention can usually be addressed by purchasing more and/or faster resources, there do not exist any equally simple mechanisms to reduce data contention. In this sense, data contention is a more "fundamental" determinant of database system performance.

For this experiment, the physical resources (CPUs and disks) were made "infinite", that is, there is no queueing for these resources [ACL87]. The other parameter values are the same as those used in Experiment 1. The results of this experiment are shown in Figures 5a through 5d. In these figures, as in the previous experiment, CENT shows the best performance and the performance of DPCC is close to that of CENT. The performance of the standard protocols relative to the baselines is, however, markedly worse than before. This is because in Experiment 1, the considerable difference in overheads between CENT and 2PC was largely submerged due to the resource and data contention in the system having a predominant effect on transaction response times. In the current experiment, however, where throughput is limited only by data contention, although transaction response times are typically smaller than under RC+DC, the commit phase here occupies a *bigger proportion* of the overall transaction response time and therefore the overheads of 2PC are felt to a greater extent. Similarly, 3PC performs significantly worse than 2PC due to its considerable extra overheads. Finally, the performance of PC remains similar to that of 2PC, for the same reasons as those given in the previous experiment.

Moving on to OPT, we observe that at very low MPLs, it behaves almost identically to 2PC since there are few opportunities for borrowing. At higher MPLs, however, OPT's performance is substantially better than that of 2PC. In fact, OPT's peak throughput is closer to that of DPCC than to 2PL. An interesting observation here is that 2PC and 3PC (as also DPCC and CENT) achieve their peak throughput at an MPL of 4, while for OPT, the corresponding MPL value is 5. This is explained by considering Figure 5b, which shows the cohort block-time ratio, where we see that this ratio is significantly lower for OPT as compared to the other protocols for a given multiprogramming level due to the complete elimination of blocking for prepared data.[12] Thus, for a given data contention level, OPT allows more concurrency in the system than the standard protocols.

In Figure 5c, the normalized (w.r.t. 2PC) values of the average commit phase durations are shown. The values are mostly as expected with 3PC being significantly slower than 2PC and PC being slightly faster than 2PC. Note, however, that in contrast to the previous experiment, the commit durations of OPT and 2PC are *identical* here. This is an outcome of the absence of resource contention which ensures that the additional concurrency generated by OPT has no negative "side-effect" on the commit phase. Therefore all the benefits of OPT's increased concurrency are derived in this environment.

Another point to note is that all the graphs are "flat" in Figure 5c. This is because commit processing only involves use of the physical resources and in the absence of resource contention, the duration of this processing becomes fixed independent of the MPL.

Finally, Figure 5d presents the transaction borrow ratio for OPT, and we observe again that borrowing increases almost linearly with MPL.

In summary, this experiment indicates that unless data contention is very low, OPT's performance can be substantially superior to that of the standard protocols.

5.3 Experiment 3: Fast Network Interface

In the previous experiments, the cost for sending and receiving messages modeled a system with a relatively slow network interface ($MsgCpu = 5ms$). We conducted another experiment wherein the network interface was faster by a factor of five, that is, $MsgCpu = 1\,ms$. The experiment was conducted for both resource-cum-data contention (RC+DC) and pure data contention (DC) scenarios.[13]

The results of this experiment under RC+DC and under pure DC are shown in Figures 6a and 6b, respectively. We observe in these figures the following:

1. The differences between the performance of CENT and the other protocols become smaller compared to the previous experiments, and in fact, for the pure DC scenario, DPCC and CENT are virtually

[12] The access delays are completely eliminated, but subsequently there may be "on-the-shelf" delays if the borrower has to wait for the lender to receive its final decision.

[13] The default parameter settings for the RC+DC and pure DC scenarios in this experiment, as well as in the following experiments, are the same as those used in Experiment 1 and Experiment 2, respectively.

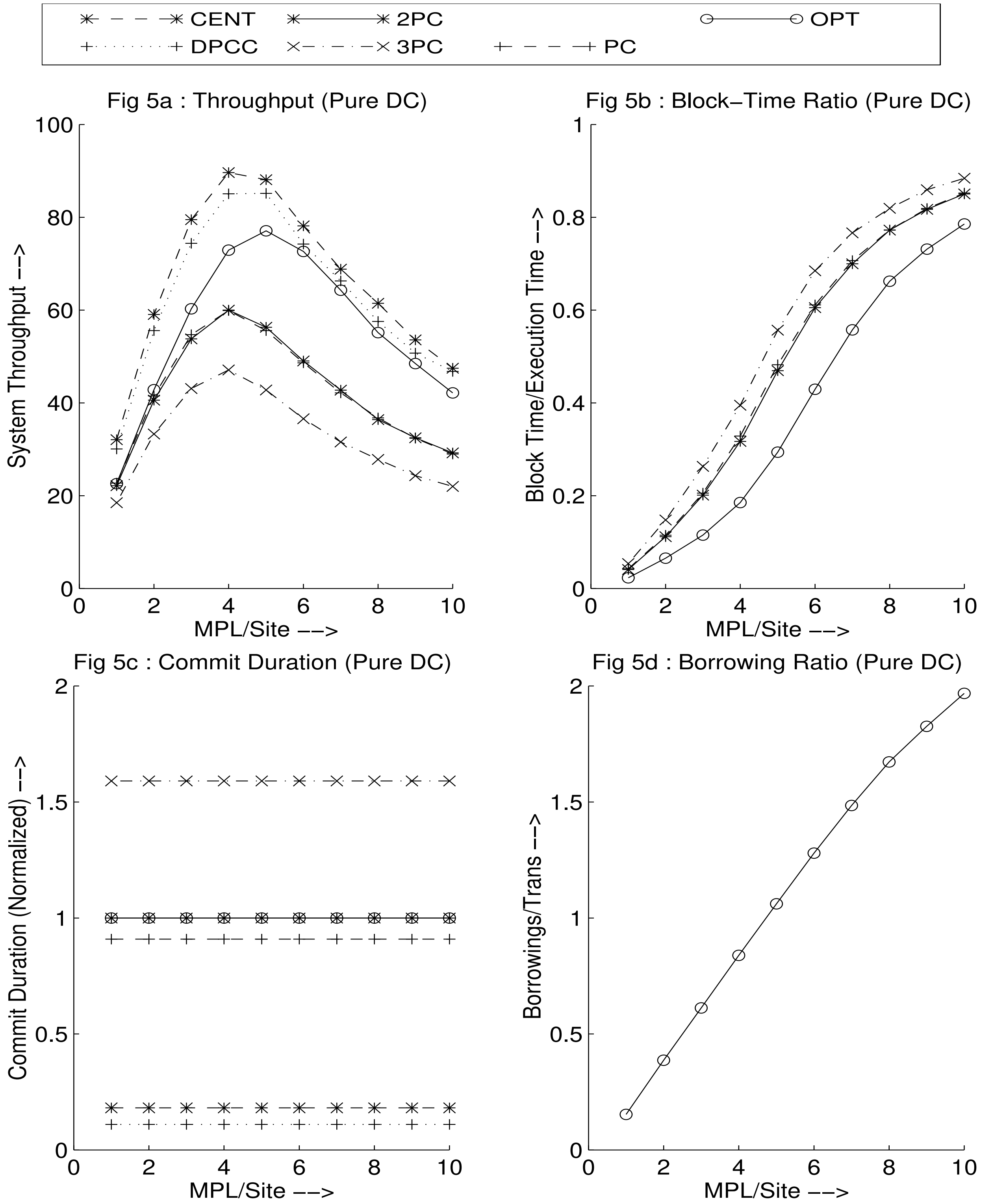

Figure 5: Pure Data Contention

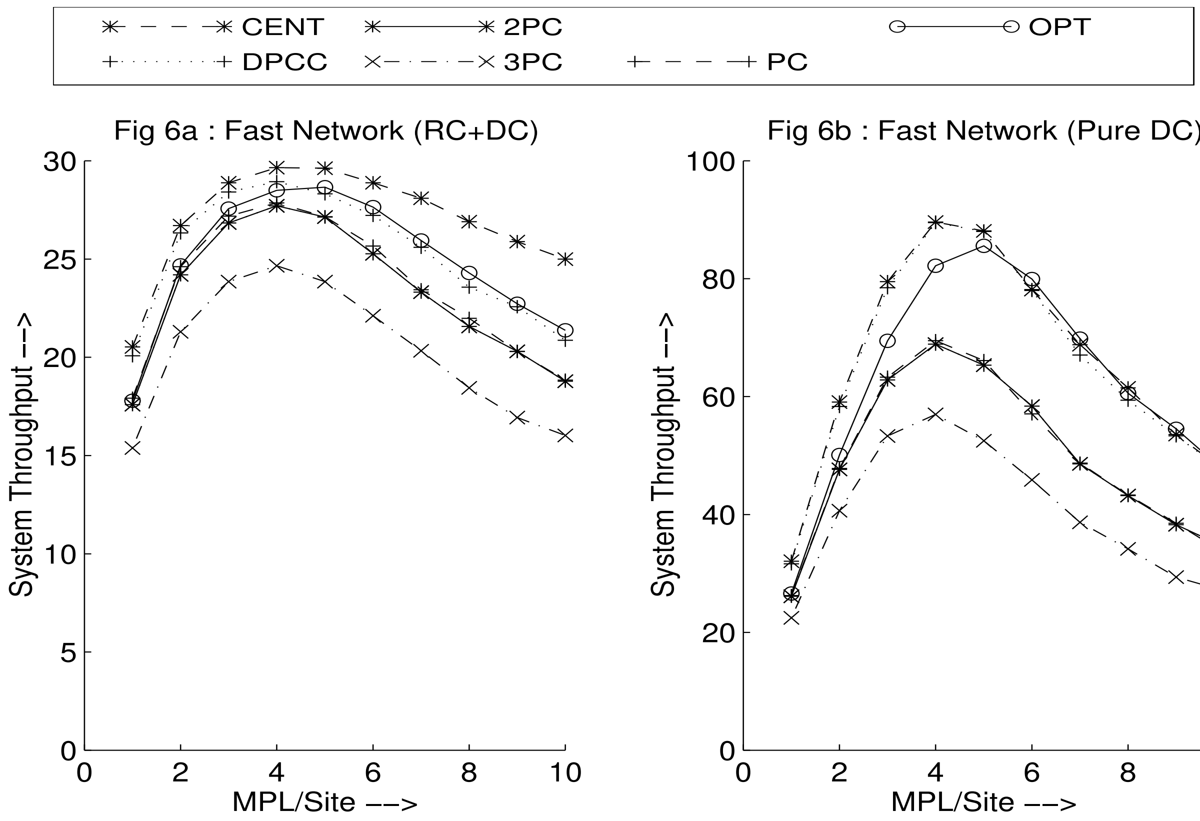

Figure 6: Fast Network

indistinguishable. This improved behavior of the protocols is only to be expected since low message costs effectively eliminate the effect of a significant fraction of the overheads involved in each protocol. Under pure DC, however, the remaining overheads of the forced-writes are significant enough to result in clear differences between the performances of DPCC and 2PC and those of 2PC and 3PC.

2. OPT's peak throughput performance is again close to that of DPCC in both the RC+DC and pure DC environments.

In summary, this experiment shows that adopting the OPT principle can be of value even with very high-speed network interfaces because faster message processing does not necessarily eliminate the *data contention* bottleneck.

5.4 Experiment 4: Higher Degree of Distribution

In the experiments described so far, each transaction executed on *three* sites. To investigate the impact of having a higher degree of distribution, we performed an experiment wherein each transaction executed at *six* sites. The *CohortSize* in this experiment was reduced from 6 pages to 3 pages in order to keep the average transaction length equal to that of the previous experiments. For this environment, the overheads of the protocols are shown in Table 4 (for committing transactions).

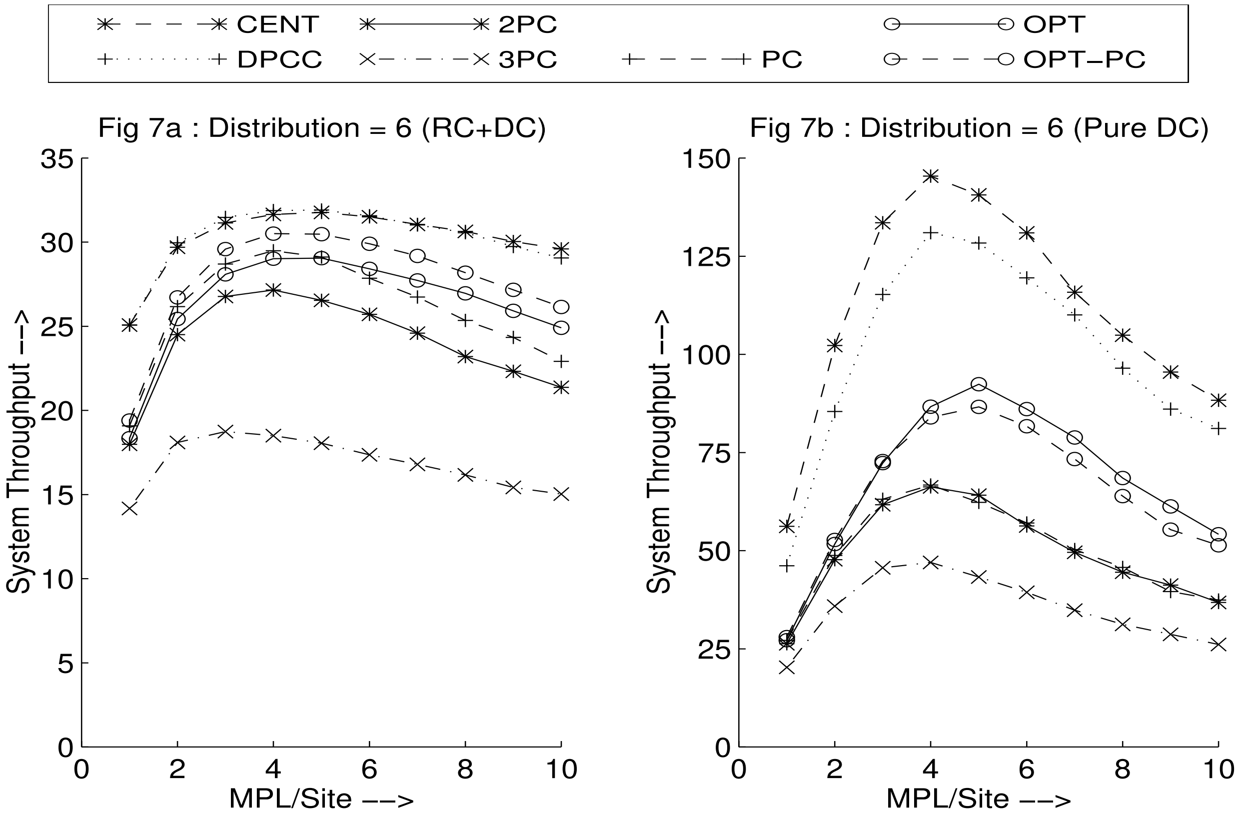

Figure 7: Higher Degree of Distribution

For this experiment, Figure 7a and 7b present the transaction throughput results under RC+DC and under pure DC, respectively. In Figure 7a, we observe that the performance of the baselines, CENT and DPCC, is virtually indistinguishable although the message processing overheads of DPCC are significantly more. The reason for this is that even with these additional overheads, the CPU is still not the bottleneck for DPCC. Moreover, these overheads are incurred in parallel at the transaction's execution sites.

For the standard protocols, however, the increase in the message overheads is sufficiently large (Table 4) that the system now operates in a heavily *CPU-bound* region. As a result, the performance differences between the baselines and these protocols are visibly more than those seen for the limited distribution workload of Experiment 1 (Figure 4a).

An interesting feature of Figure 7a is that, for the *first* time, we observe a significant difference between the performance of PC and that of 2PC. In fact, PC exhibits better performance than 2PC across the entire MPL range. The reason for this behavior is the heavily CPU-bound nature of the workload. In this environment, PC's reduced message overheads result in its performance being better than 2PC. This is further confirmed in the results of Figure 7b, where in the absence of resource contention, PC again performs almost identically to 2PC.

Turning our attention to OPT, we observe that under RC+DC (Figure 7a), it performs *slightly worse* than PC at lower MPLs (i.e., in the non-thrashing region). This is because OPT is based on 2PC and therefore incurs the same magnitude of overheads, although their effect is partially offset by its increased

Table 4: Protocol Overheads (DistDegree = 6)

Protocol	Execution	Commit	
	Messages	Forced-Writes	Messages
2PC	10	13	20
PA	10	13	20
PC	10	8	15
3PC	10	20	30
DPCC	10	1	0
CENT	0	1	0

concurrency.

These results may seem to suggest that OPT may not be the best approach for workloads of the type considered in this experiment. Note, however, that we can now bring into play the ability of OPT to "peacefully and usefully coexist" with other optimizations by combining OPT and PC to form an **OPT-PC** protocol. The performance of this protocol is also shown in Figure 7a, wherein it provides the best overall performance by deriving the benefits of both the OPT and PC optimizations.

Under pure data contention (Figure 7b), note that the performance difference between CENT and DPCC is more as compared to earlier experiments. This is because the average transaction response time is reduced due to the heightened degree of distribution. As a result, message delays now form a significant fraction of the response time. We also observe in Figure 7b that the difference between the performance of DPCC and that of 2PC is now very large (the peak throughput of DPCC is almost *twice* that of 2PC). This again clearly demonstrates that distributed commit processing can affect performance to a significantly greater degree than distributed data processing.

As mentioned earlier, PC performs almost the same as 2PC in Figure 7b. The performance of OPT continues to be superior to 2PC, but in contrast to what was seen in Figure 7a, we now observe that the performance of OPT-PC is no better than that of OPT at low MPLs and slightly worse at higher MPLs. This is because the effect of OPT-PC is to increase the duration of the wait phase of cohorts (by introducing the `collecting` forced write log record), and at the same time to reduce their commit phase duration (by reducing their forced writes), thus decreasing the "commit proportion" of the overall response time and diminishing the utility of the optimistic feature.

In summary, these results indicate that the OPT approach can be usefully utilized even in non-conducive environments by combining it with the classical protocol that works best for that environment.

5.5 Experiment 5: Non-Blocking OPT

In the previous experiments, we observed that OPT, which is based on 2PC, performed significantly better than the standard protocols. This motivated us to evaluate the effect of incorporating the same optimizations in the **3PC** protocol. We refer to this protocol as OPT-3PC and for this experiment, Figures 8a and 8b present the throughput results under RC+DC and pure DC, respectively. In Figure 8a we observe that the performance of OPT-3PC is similar to that of 3PC at lower MPLs. However, at

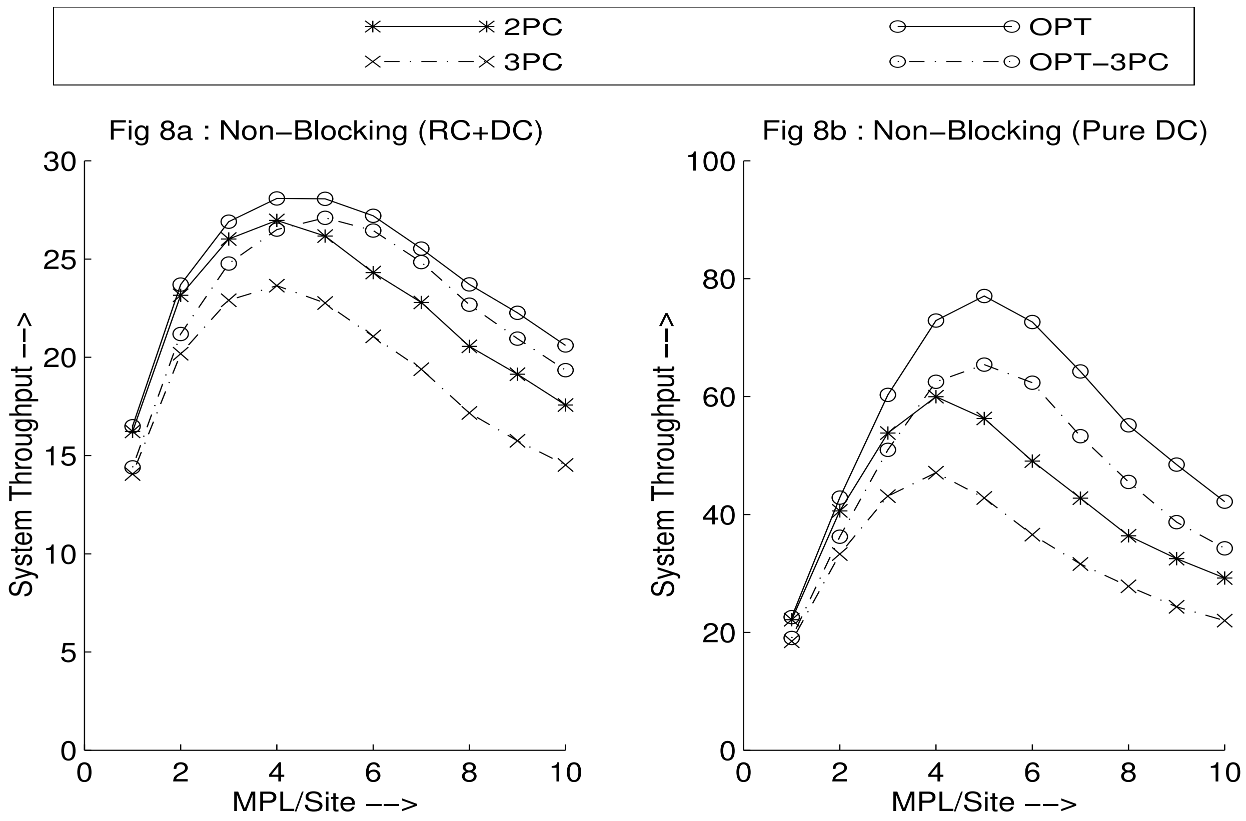

Figure 8: Non-blocking OPT

higher MPLs, OPT-3PC not only performs better than 3PC, but also achieves a peak throughput that is comparable to that of 2PC. These observations show up more vividly in Figure 8b, where the peak throughput of OPT-3PC actually visibly *surpasses* that of 2PC.

An important point to note here is that the optimistic feature has potentially more impact in the 3PC context than in the 2PC context. This is because the duration of the prepared state is significantly longer in 3PC (due to the extra phase), thereby increasing the benefits of borrowing.

In summary, these results indicate that by using the OPT approach, we can *simultaneously* obtain both the highly desirable non-blocking functionality and better peak throughput performance than the classical blocking protocols. This, in essence, is a "win-win" situation.

5.6 Experiment 6: Surprise Aborts

In all of the experiments discussed earlier, the *SurpriseAbort* probability parameter was set to zero. It is for this reason that no difference was possible in the relative performance of PA and 2PC. We also conducted an experiment, described below, to model environments where surprise aborts do arise in the commit phase due to violation of integrity constraints, software errors, system failure, etc., and evaluated their impact on protocol performance. The important point to note here is that these environments are fundamentally biased against OPT since its underlying assumption that transactions will usually commit may not hold under surprise aborts.

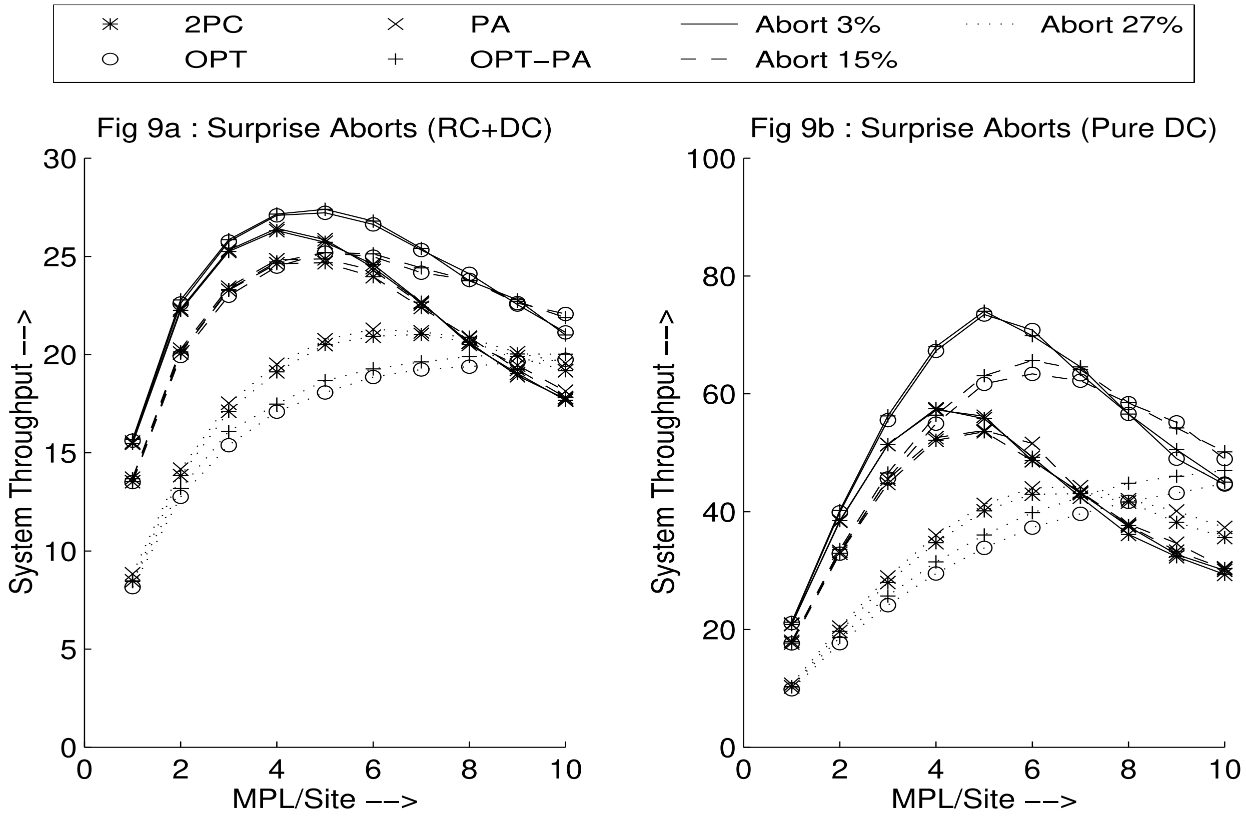

Figure 9: Surprise Aborts

In this experiment, we considered three different cases, corresponding to *SurpriseAbort* settings of 1 percent, 5 percent and 10 percent, respectively. Since each transaction is composed of three cohorts, these cohort abort probabilities translate to overall *transaction* abort probabilities of approximately 3 percent, 15 percent and 27 percent, respectively. Note that the latter two cases model abort probabilities that appear artificially high as compared to what might be expected in practice. Modeling these high abort levels, however, helped us determine the extent to which OPT was *robust*.

The results of this experiment under RC+DC and under pure DC are presented in Figures 9a and 9b, respectively. We observe here that OPT's peak throughput performance is comparable to that of 2PC even when 15 percent of the transactions are aborting and it is only when the abort level is in excess of this value that we begin to see an appreciable difference in performance. At these high abort levels, the optimism of borrowing transactions proves to be misplaced since they are often aborted due to the abort of their lenders.

We also observe in these figures that, in spite of having a large fraction of aborts, PA which is designed to do well in the abort case, shows only marginal improvement over 2PC. This is because although PA saves on the forced writes and acknowledgements, these savings are small in comparison to the *total* overheads of commit processing. For example, in the 27 percent transaction abort probability case, 2PC incurs about 8.8 forced writes and 2.5 acknowledgements per committed transaction, whereas the corresponding values for PA are 7.7 and 2, respectively. When the system is not heavily resource-bound, these savings do not affect the throughput performance significantly. To investigate this further, we conducted the same experiment with a higher degree of distribution ($DistDegree = 6$), which results in a heavily CPU-bound environment, as in Experiment 4. In this situation, we found that the savings by PA were sufficient to make it perform clearly better than 2PC.

The important point to note here is that OPT can *also* derive these performance benefits of PA by combining with it to form OPT-PA – the performance of this combined algorithm is also presented in Figures 9a and 9b and shows the expected outcome.

Another interesting feature in Figures 9a and 9b is that, at high MPLs, the performance of the OPT and 2PC protocols in a system with higher probability of surprise aborts is better than their performance in a system with lower probability of aborts, that is, there is a *performance crossover*. For example, for a surprise abort percentage of 3 in the pure DC case, the crossover occurs at an MPL of 7 for OPT and OPT-PA. The reason for the crossover is the following: In our model, as in earlier transaction management studies (e.g. [ACL87]), aborted transactions are delayed before restarting with the delay period set equal to the average response time. This delay effectively becomes a crude way of controlling the data contention in the system. At high MPLs, this results in the system with higher transaction abort probabilities having less data contention than the system with lower transaction abort probabilities and therefore performing better, resulting in the crossover. Note, however, that the *peak* throughput is significantly smaller for the system with higher transaction abort probabilities than the one with lower abort probabilities.

In summary, these results indicate that the OPT approach is robust in that it provides performance gains over the entire practical range of surprise abort frequencies.

5.7 Experiment 7: Partial-Update Transactions

In the previous experiments, all transactions were complete updaters, that is, every page accessed by the transaction was also updated. We conducted another experiment where each page accessed by the transaction was updated with a probability of only one-half (by setting $UpdateProb = 0.5$). Reducing the update probability reduces the contention level in the system in two ways: First, the obvious reduction due to sharing of read locks by concurrently executing cohorts. Second, and specific to the distributed commit environment, is the reduction due to cohorts releasing the read locks on receiving the PREPARE message from the master.

For this partial-update experiment, Figures 10a and 10b present the transaction throughput results for the RC+DC and pure DC environments, respectively. These results show similar qualitative behavior to those observed in the corresponding experiments for complete update transactions discussed earlier. Quantitatively, however, the performance difference between OPT and 2PC is decreased (as compared to Experiment 1)for the RC+DC environment. The reason for this is the following: Due to the decrease in data contention, the utilization of the physical resources increases due to the increased concurrency and they become the primary performance bottleneck. Both these factors (the data contention decrease and the physical resource bottleneck) result in less opportunities for optimistic lending. This fact was confirmed by measuring the borrow ratio – its value for OPT was significantly less (approximately halved) for this experiment as compared to its value for the corresponding complete update experiment (Experiment 1

Note, however, that for the pure DC environment (Figure 10b), the quantitative performance of OPT is again similar to that seen in the complete update environment, since lending opportunities are maximized in this environment.

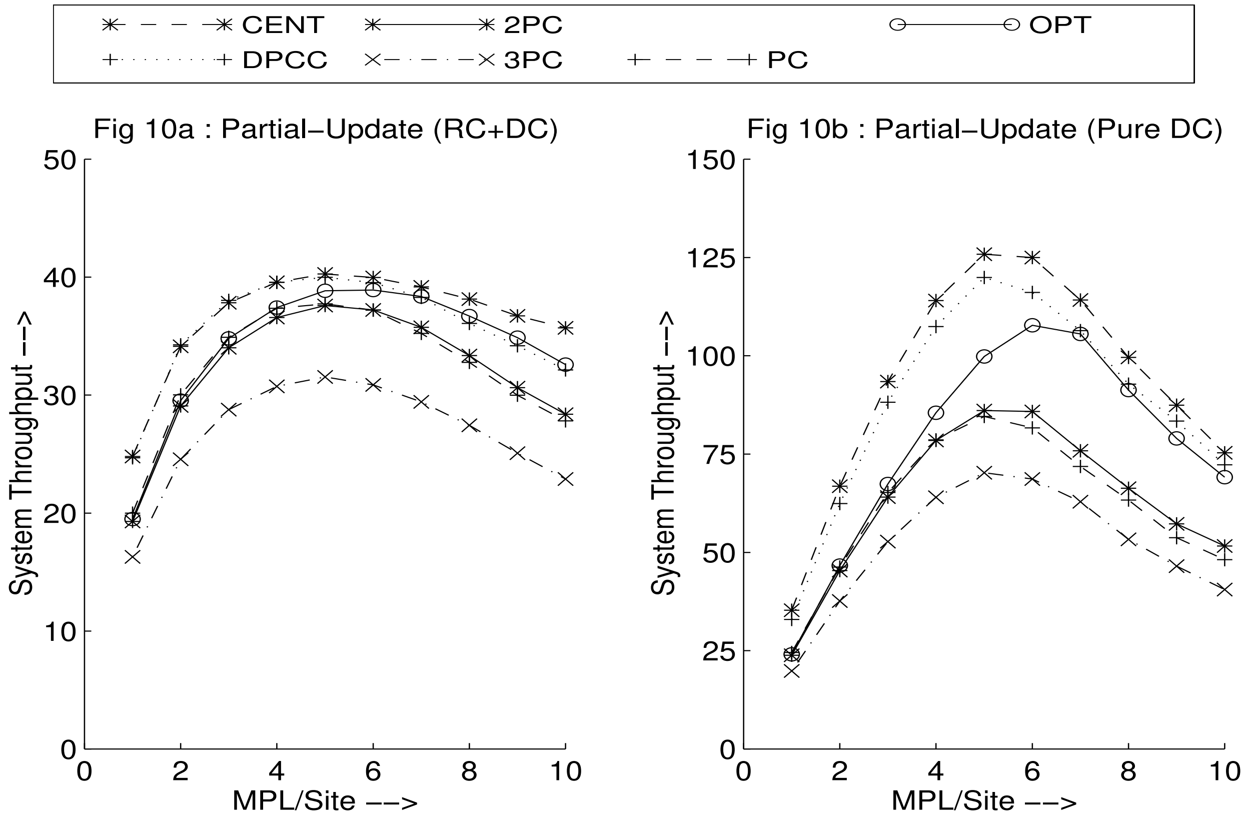

Figure 10: Partial-Update Transactions

5.8 Experiment 8: Sequential Transactions

In all of the previous experiments, the cohorts of each transaction executed in parallel. We also conducted similar experiments for transaction workloads with *sequential* cohort execution and we report on those results here.

An interesting feature that we noticed in the sequential transaction experiments is that the performance differences between the protocols *decreased* as compared to carrying out the same experiment with parallel transactions. This is because the duration of the data phase is more for sequential transactions as compared to their parallel counterparts. The duration of the commit phase, however, is *the same* for both types of transactions. Hence, the commit proportion is significantly smaller in the case of sequential transactions (this effect was confirmed by measuring these values in our experiments), resulting in OPT having lesser impact on the throughput even though it continues to provide better throughput than 2PC and its classical variants.

The detailed results for all the sequential experiments are available in [Gup97] – for illustrative purposes, we present here the results for the baseline RC+DC and pure DC situations in Figures 11a and 11b, respectively.

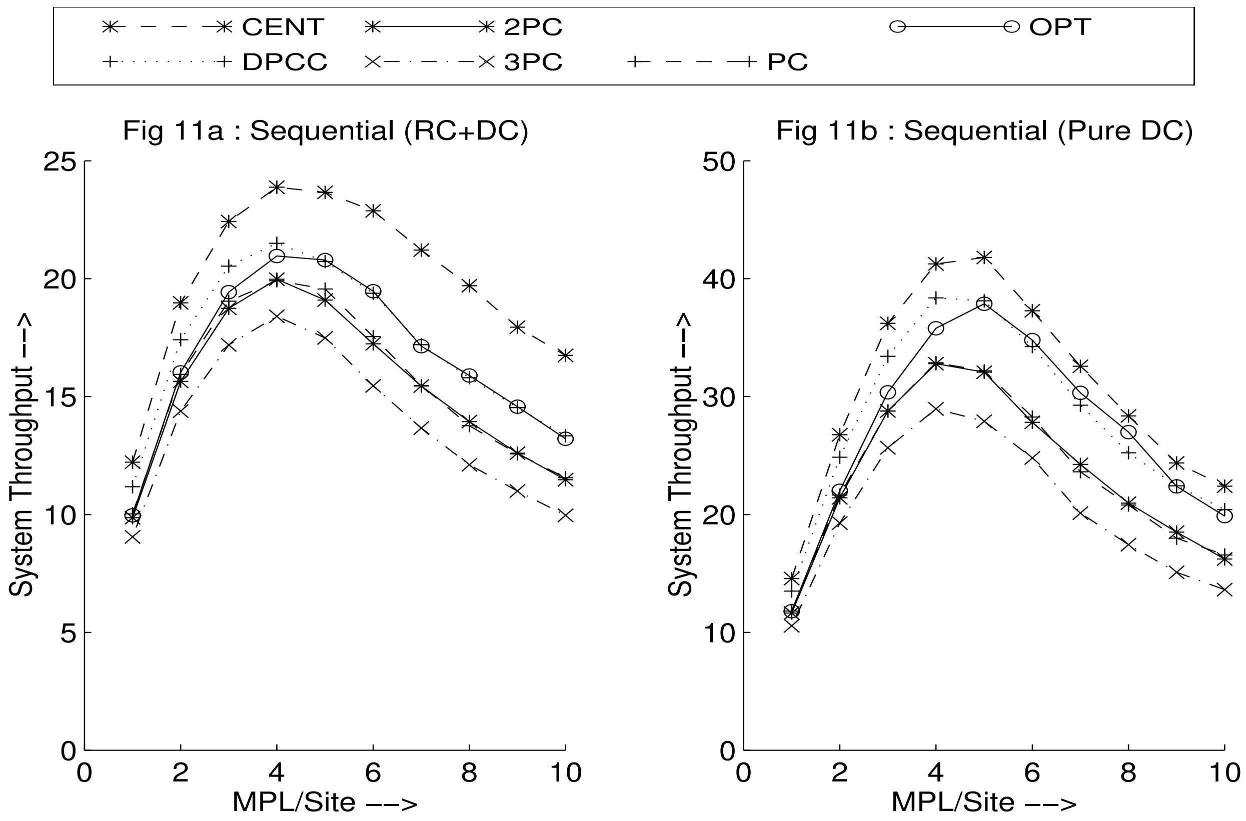

Figure 11: Sequential Transactions

5.9 Other Experiments

We conducted several other experiments to explore various regions of the workload space. These include workloads with different database sizes, non-zero think times, etc. The relative performance of the protocols in these additional experiments remained qualitatively similar to that seen in the experiments described here (see [Gup97] for the details).

6 Lending Variations of OPT

In the previous section, we quantitatively demonstrated that the OPT approach is conducive to good performance. We now move on to considering possible variations, in the design of OPT's lending mechanism, that may have the potential to increase the performance of OPT. Two such variants, Borrow-Bit OPT and Borrow-Message OPT, are presented and evaluated in the remainder of this section.

Recall that in basic OPT, a borrower that completes its work is placed "on the shelf" until the associated lender receives its decision. Only after that (and if the lender commits) the borrower sends its WORKDONE message. Suppose, instead, that a borrower sends its WORKDONE message as soon as its work is done. If the master then initiates the first phase of the commit protocol, the borrower can be made to wait to respond to the the PREPARE message until the lender receives its decision (assuming, of course, that the lender has not received the decision in the meantime). An advantage of this approach is that the transaction response time is reduced for the borrower because should the lender commit, it can immediately respond to the master's PREPARE message – in contrast, in basic OPT, it would send the WORKDONE message only at this time.

The above optimization is particularly relevant for *sequential* transactions since each on-the-shelf wait during data processing results in an increase of the overall response time. By allowing the transaction to proceed with the execution of the next cohort in the sequence, the total execution time is reduced. For parallel transactions, however, there is a possibility of overlap between the on-the-shelf wait of the borrower cohort and the data phase of sibling cohorts (with longer data processing times). This partially negates the advantages accruing from a cohort (that finishes early) sending its WORKDONE message even before its lending transaction commits.

A problem, on the other hand, is that unless further steps are taken, a transaction that borrows at one site can become a lender at another site, leading to an abort chain whose *length is greater than one*. This happens when the sibling cohorts (of the borrower) become lenders at their sites after responding to the master's PREPARE message. If the original lender aborts, it will result in cascaded aborts.

The BB-OPT and BM-OPT protocols, discussed next, implement the above optimization but are designed to avoid the cascading abort problem.

6.1 Borrow-Bit OPT (BB-OPT)

In this protocol, after finishing its data processing work, a borrower cohort (whose lender has not yet committed) sends its WORKDONE message to the master with an additional "borrow bit" set to true. The master passes this bit on to the sibling cohorts with the PREPARE message. Cohorts receiving this bit do not lend their data, thereby ensuring that borrower transactions cannot simultaneously be lenders (the condition necessary to ensure the absence of cascading aborts).

The advantage of the BB-OPT approach is that it reduces the borrower transaction's response time as discussed above and is easily implementable. It only adds one more bit of information to the WORKDONE and PREPARE messages but no new messages are required. A disadvantage, however, is that it prevents the borrower transaction from subsequently ever becoming a lender, *even after the borrowing is over*. Yet another disadvantage, particularly relevant for sequential transactions, is that in the event that the lender aborts, the continued execution of the sibling cohorts turns out to be a waste of system resources.

6.2 Borrow-Message OPT (BM-OPT)

In this protocol, the above-mentioned potential drawback of BB-OPT is addressed in the following manner: a borrower cohort that had set the borrow bit sends a BORROW-OVER message to the master if the lender subsequently commits. This message to the master effectively invalidates the borrower bit sent earlier.[14]

A special situation in the above protocol is wherein the borrower cohort receives the PREPARE message from the master *before* it sends the BORROW-OVER message. In this situation, the master will have *already* sent the borrower-bit to the sibling cohorts, thereby turning off their lending. To turn lending on again

[14] If multiple cohorts concurrently send the borrow bit, the master waits till it receives the BORROW-OVER message from all such borrowers before turning lending on again.

after receiving the BORROW-OVER message, it will have to send a LEND-START message to *all* these sibling cohorts.

Though incorporating the LEND-START message may appear to be a good addition to the protocol, we do not expect this approach to be useful in general for the following reasons: First, it involves much more message complexity (BORROW-OVER involves only a single message from borrower to master, whereas LEND-START must be from the master to *all* the sibling cohorts). Second, it may be fruitless if the sibling cohorts have already received the final decision by the time the LEND-START message reaches them, or even if they receive it shortly afterwards, since the lending duration becomes very small. There is a good chance of this happening since it takes two message delays to turn lending on again (BORROW-OVER plus LEND-START) – exactly the same number that it takes for the sibling cohorts to receive their decision (YES vote plus the COMMIT decision message). Therefore, it appears that only in the case where cohorts, for some reason, take a long time to vote, that the optimization may have the desired effect.

Based on the above discussion, we feel that a more effective solution in general would be for the borrower cohort to *not send* the BORROW-OVER message if it has already received the PREPARE message. A related issue arising out of the distributed nature of execution is the situation wherein the BORROW-OVER and PREPARE messages "cross" each other on the network – in this situation, the master ignores the BORROW-OVER message. It is this approach that we have implemented for BM-OPT in our simulation experiments described below.

Overall, note that while the BM-OPT scheme potentially increases the lending opportunities, it is at the cost of increasing the message processing overhead in the system. Moreover, the implementation becomes more complex since inter-site communication and coordination now become necessary. Also, the protocol ceases to be a fully "hand-shaking" protocol since the BORROW-OVER message may now be received asynchronously, thereby necessitating significant changes to the transaction manager program.

6.3 Performance of the Variants

We have implemented and evaluated the performance of the BB-OPT and BM-OPT variants for all the workloads discussed earlier in this study. In the interest of space, we present here only the results for the baseline RC+DC and pure DC environments with parallel transaction execution.

The throughput performance of BB-OPT, BM-OPT and the basic OPT protocol are shown in Figures 12a and 12b for the RC+DC and pure DC environments, respectively. We observe here that for the RC+DC environment, the performance of both variants is fairly close to that of basic OPT. More interestingly, the performance of basic OPT is *better* than or the same as that of the variants over the entire loading range. Moving on to the pure DC environment (Figure 12b), we find that perceptible performance differences between the protocols arise at higher MPLs. Again, basic OPT performs the best, followed by BM-OPT and BB-OPT.

The above results can be understood by considering Figures 12c and 12d, which capture the borrow ratio of all the protocols for the RC+DC and pure DC environments, respectively. From these figures, the reason for the variants performing worse than basic OPT becomes clear – they lend less than basic OPT at higher MPLs.

The reason for reduced lending in BB-OPT is that it prevents borrowers from subsequently becoming

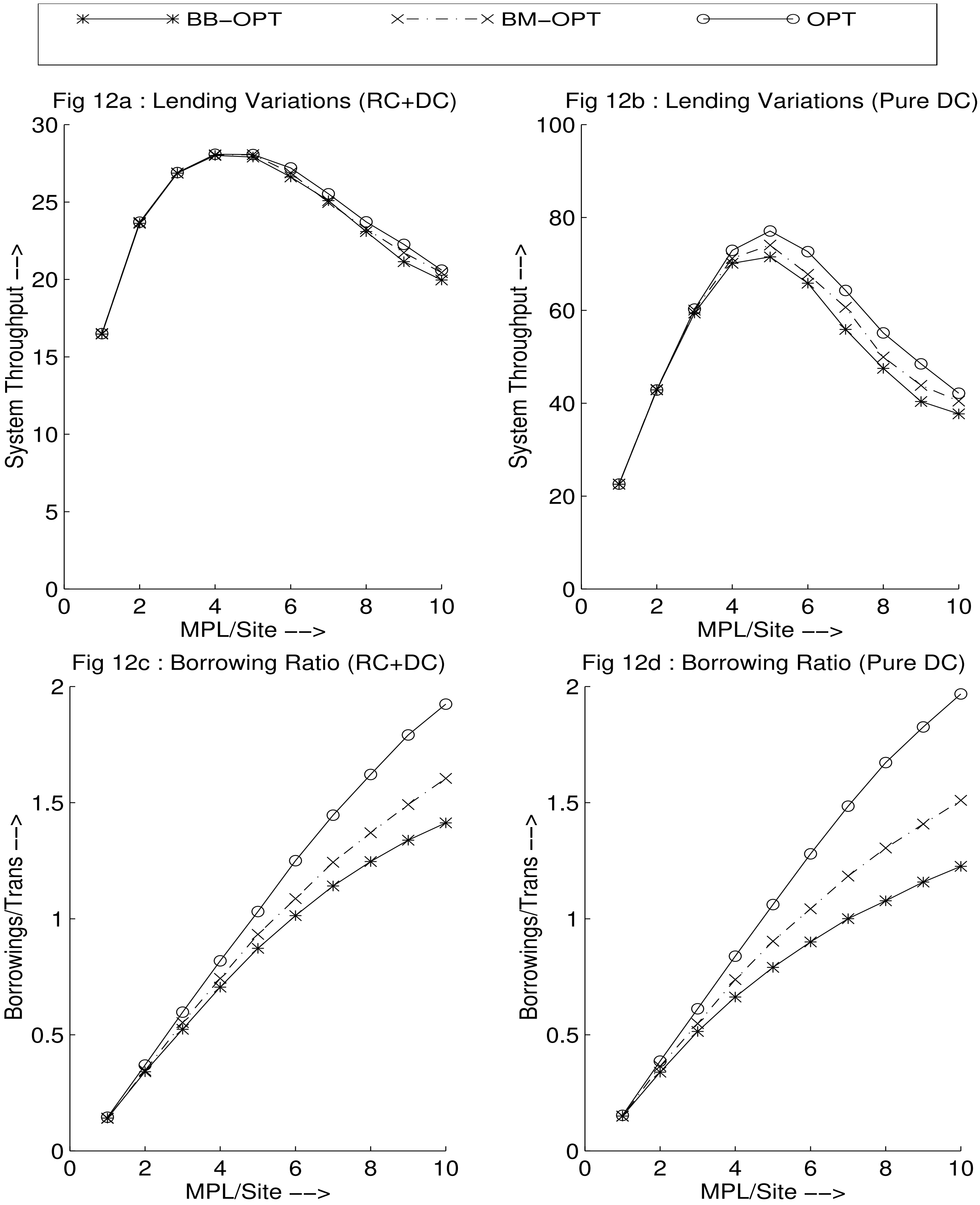

Figure 12: Lending Variations of OPT

lenders, as mentioned above. This becomes a handicap at higher MPLs where data contention is significant and there is considerable scope for lending. On the other hand, BM-OPT, by virtue of its BORROW-OVER message, partially addresses the reduced lending problem of BB-OPT and therefore has a higher borrow ratio than BB-OPT. However, the improvement is not sufficient to match basic OPT's borrow ratio because often either (a) the cohort receives the PREPARE message before its borrowing period is over, or (b) due to the message transmission delays, the master receives the BORROW-OVER message *after* it has sent the PREPARE message. Note that in both these cases, the PREPARE message has the "no-lending" bit set since the master thinks that the borrowing period is not yet over.

In summary, the above experiment shows (as was also confirmed in our remaining experiments with the variants) that trying to "fine-tune" OPT's lending policy may turn out be counter-productive. That is, the basic design itself is quite efficient in deriving most of the benefit available from the lending approach.

As mentioned earlier in this section, the BB-OPT and BM-OPT optimizations appear to be particularly suited for *sequential* transactions. The performance results for this environment (not shown here) did exhibit the expected effect in that the performance of both BB-OPT and BM-OPT improved to come very close to that of basic OPT, but never actually surpassing it. This supports our above conclusion.

7 Other Commit Protocols and Optimizations

A variety of commit protocols have also been proposed in the literature in addition to those evaluated in our study (2PC,PA,PC,3PC). These include *linear 2PC* [Gra78], *distributed 2PC* [OV91], *Unsolicited Vote (UV)* [Sto79], *Early Prepare (EP)* and *Coordinator Log (CL)* [SC90, SC93] protocols. More recently, the *Implicit Yes Vote (IYV)* protocol [AC95] and the *two-phase abort (2PA)* protocol [BC96] have been proposed for distributed database systems that are expected to be connected by extremely high speed (gigabits per second) networks. We give a brief description of these other protocols below and also discuss how (some of) these can be integrated with OPT. We also consider commit protocols that have been proposed with a relaxed notion of atomicity in mind.

7.1 Protocols Satisfying Traditional Atomicity

The *linear 2PC* [Gra78] protocol (also known as *nested 2PC*) reduces the number of the message-exchanges in 2PC by ordering all the cohorts in a *linear chain* for communication purposes. This, however, is done at the cost of concurrency in the commit processing, resulting in larger commit processing times. In *distributed 2PC* [OV91], each cohort *broadcasts* its vote to all other cohorts. This means that each cohort can independently arrive at the final decision without needing to have this information communicated by the master. The tradeoff here is between the reduction in commit processing time and the increase in message overheads due to the cohort vote broadcasts.

The protocol used in the Distributed Ingres database system is called *Unsolicited Vote (UV)* [Sto79]. Here, cohorts enter the prepared state at the time of sending the WORKDONE message itself. Thus, the WORKDONE message acts as a *implicit* YES vote eliminating the need for sending the PREPARE messages by the master. However, the cohorts remain in the prepared state for a longer period than in 2PC, thus making them more vulnerable to failure-blocking. Also, the read locks, which can usually be released when a cohort enters the prepared state, cannot be released in UV because the master may later send more work to the cohort forcing it back into the processing state from the prepared state. Thus, releasing the read locks would compromise the serializability semantics of the transaction. Moreover, a cohort in UV has to force-write a prepared log record *each time* it responds with the WORKDONE message.

In [SC90, SC93], a combination of the UV and the PC protocols, called *Early Prepare (EP)*, is proposed. They also propose a *Coordinator Log (CL)* protocol for client/server environments wherein cohorts *piggyback* the log records on the WORKDONE messages to the master, instead of force-writing them locally. That is, the distributed log associated with each transaction is effectively converted into a centralized log located at the site where the master is resident. A difficulty with this approach is that ensuring WAL and the rolling back of aborted transactions becomes more complex and cumbersome since these functionalities have to be implemented over the network at remote sites [AC95].

As mentioned earlier in Section 2.3, the PC protocol introduces an additional force-write (the collecting log record) and therefore incurs more overheads as compared to the PA protocol for both aborted transactions as well as read-only transactions. Mechanisms for minimizing the logging activity of PC and making its overheads comparable to that of PA have been proposed in [LL93] and [ACL97].

The recently proposed *Implicit Yes Vote (IYV)* protocol [AC95, AC96] and *two-phase abort (2PA)* protocols [BC95, BC96] are designed to exploit the high data transfer rates of gigabit networks. In a gigabit network, the number of messages exchanged, and not the size of the messages, is the primary concern. So these protocols attempt to minimize the number of messages exchanged in the commit processing by including more information in each message.

Apart from PA and PC, a variety of other optimizations have been proposed for the 2PC protocol. A comprehensive description and analysis of such optimizations is presented in [SBCM95]. While some of them are similar to those mentioned above, additional ones include *Read-Only* (one phase commit for read-only transactions), *Long Locks* (cohorts piggyback their commit acknowledgments onto subsequent messages to reduce network traffic), *Shared Logs* (cohorts that execute at the same site as their master share the same log and therefore do not need to force-write their log records) and *Group Commit* (forced writes are batched together to save on disk I/O).

7.2 Integration with OPT

As discussed earlier in Section 3.2 and as demonstrated by the experimental results of Section 5, optimizations such as PA/PC can be easily integrated with OPT as a useful supplement, or even *synergistically* as in the combination with 3PC. Among the additional optimizations discussed above in this section, OPT is especially attractive to integrate with protocols such as *Group Commit* and *linear 2PC*, since they *extend*, like 3PC, the period during which data is held in the prepared state, thereby allowing OPT to play a greater role in improving system performance. Most of the logging-minimized optimizations of PC can also be integrated with OPT in a straightforward manner, as also optimizations such as *Read Only*, *Long Locks*, and *Shared Logs*.

However, we *do not* recommend combining OPT with protocols wherein entering the prepared state does not automatically imply work completion. This is because the design of these protocols is fundamentally at odds with the motivating assumption behind OPT, namely, that a transaction entering the prepared state is highly likely to commit. The *Unsolicited Vote*, *Early Prepare* and *IYV* protocols are examples of this class since they do not guarantee that a cohort which has unilaterally entered the prepared state will not later be forced back into a (data) processing state. Combining OPT with such "multi-data-phase" protocols can result in performance *degradation* due to cascading aborts, long "on-the-shelf"-times for borrowers, deadlocks involving the lender and the borrower, etc., as explained below.

First, a lender that has been forced back to the processing state can be aborted (for example, due to becoming involved in a deadlock) causing the borrower also to abort. Therefore, surprise aborts are no longer the only reason for lender aborts. This means that OPT's assumption that sibling cohorts will almost always vote to commit may not hold true to the same extent as described in Section 3.1. Second, a lender that re-enters the executing state may later become a borrower, thus giving rise to a "borrow-chain" of *arbitrary* length. In case the lender cohort at the root of this chain aborts, all the cohorts in the chain have to be aborted – that is, cascading aborts result. Third, a lender is usually expected to quickly complete its commit processing since it is only waiting for the decision from its master. However, if a lender is forced back to the processing state, its associated borrowers will have to wait on-the-shelf while the lender completes its newly assigned work. This could give rise to long on-the-shelf delays for the borrowers and, since the borrowers are holding resources during the delay, could have a serious negative impact on performance. Finally, deadlocks between the lender and the borrower can occur – for example, if the borrower is waiting for the lender to finish while the lender needs to access a data item currently held by the borrower. Identifying and handling such deadlocks will impose extra overheads on the system.

7.3 Protocols Satisfying Relaxed Atomicity through Compensations

A radically different approach to improving commit protocol performance is explored in [SLKS92, Yoo94].[15] Here, individual sites are allowed to *unilaterally* commit – the idea is that unilateral commitment results in improved response times. If it is later found that the decision is not consistent globally, "compensation" transactions are used to rectify the errors.

While the compensation-based approach certainly appears to have the potential to improve transaction response times, yet there are quite a few practical difficulties. First, the standard notion of transaction atomicity is not supported – instead, a "relaxed" notion of atomicity [KLS90, LKS91a, LKS91b] is provided. Second, the design of a compensating transaction is an application-specific task since it is based on the application semantics. Third, the compensation transactions need to be designed in advance so that they can be executed as soon as errors are detected – this means that the transaction workload must be fully characterized *a priori*. Finally, "real actions" [Gra81] such as firing a weapon or dispensing cash may not be compensatable at all [LKS91a].

8 Related Performance Work

As mentioned in the Introduction, virtually all the earlier performance studies of commit protocols have been limited to comparing protocols based on the *number* of messages and the *number* of force-writes that

[15] These papers consider *real-time* database environments but their approach is also applicable to conventional DBMS.

they incur. We are aware of only two exceptions in this regard: (1) In [Coo82], the expected number of sites that are *decision-blocked* after a failure was determined through a probabilistic analysis. It compared a variety of two-phase and three-phase protocols, including 2PC, Linear 2PC and 3PC, with regard to this metric. (2) In [LAA94], end-user metrics similar to those evaluated in our study were considered and evaluated using a simulation model. It compared the performance of commit protocols in a client/server type of database environment. As explained below, the scope and methodology of both these studies is considerably different from ours.

The first difference with Cooper's work [Coo82] arises from our focus being prepared data blocking, not decision blocking. The former is of relevance to this paper because of our interest in studying the performance effects of commit protocols on normal transaction processing. As explained in 2, effects on normal processing are of primary importance since failures are (expected to be) rare. Secondly, we explicitly model both message and logging overheads, whereas their analysis incorporated *only* message overheads. Finally, we include the industry-standard PA and PC protocols in our evaluation, whereas their work predated these protocols.

Turning to [LAA94], recall that we model a *fully distributed* database environment wherein each site can host both transaction masters and transaction cohorts. In contrast, a client/server database type of environment is considered in [LAA94], where the master of a transaction resides at the client site and the cohorts of the transaction are executed at the server sites. Apart from this major difference in the system model, there are many other significant differences:

1. An *open* system is modeled wherein the input process is independent of the system execution process. Most studies of database systems (including ours) have, however, usually employed a *closed* queueing model since it is more representative of real-world environments.

2. Only blocking commit protocols are considered in their study, whereas we have examined both blocking and non-blocking protocols.

   For blocking protocols, they model the effects of site failures and the overheads of recovery processing. However, as discussed in Section 2, failures are typically rare enough that we do not expect to see a difference in *average* performance among the protocols because of one commit protocol having a better failure recovery mechanism than the other. In fact, their results indicate that even with exaggerated failure rates, the effect on the relative performance of the protocols is almost negligible.

3. We present and evaluate a new high-performance protocol (OPT) that is easy to implement and incorporate in current systems, whereas their study is restricted to classical protocols.

4. Finally, the utility of the performance results presented in the paper is unclear since the descriptions of the protocols do not appear to strictly adhere to their original published versions. For example, it is mentioned that in 2PC a cohort does not update the modified data until it receives the COMMIT message from the master whereas in PC the cohort updates the data when it votes YES. In our understanding of commit protocols, there is no relationship between the timing of data updates and specific protocols – in fact, no mention of data updates is made in [MLO86], the paper in which the PA and PC protocols were first proposed. Another example of a difference in protocol description is

that no mention is made about the `collecting` log record in the PC protocol and it is stated that the overheads of PC and 2PC are the same for aborting transactions, when in fact, due to the additional `collecting` log record, the overheads of PC are more than that of 2PC. Similar differences exist in the descriptions of the other protocols as well (see [Gup97] for details).

9 Conclusions

In this paper, we have quantitatively investigated the performance implications of supporting distributed transaction atomicity. Using a detailed simulation model of a distributed database system, we evaluated the throughput performance of a variety of standard commit protocols, including 2PC, PA, PC and 3PC, apart from two baseline protocols, CENT and DPCC. We also developed and evaluated a new commit protocol, OPT, that was designed specifically to reduce the blocking arising out of locks held on prepared data. To the best of our knowledge, these are the first quantitative results on commit processing in a fully-distributed DBMS with respect to end-user performance metrics.

Our experiments demonstrated the following:

1. Distributed commit processing can have considerably more effect than distributed data processing on system performance. This highlights the need for developing high-performance commit protocols.

2. The PA and PC variants of 2PC, which reduce protocol overheads, have been incorporated in a number of database products and transaction processing standards. In our experiments, however, we found that these protocols provide tangible benefits over 2PC only in a few restricted situations. PC performed well only when the degree of distribution was high – in current applications, however, the degree of distribution is usually quite low [SBCM95]. PA, on the other hand, performed only marginally better than 2PC even when the probability of surprise aborts was close to thirty percent – in practice, surprise aborts occur only occasionally.

   We should note that the above conclusion is limited to the update-oriented transaction workloads considered here – PA and PC have additional optimizations for fully or partially *read-only* transactions [MLO86] that may have a significant impact in workloads having a large fraction of these kinds of transactions.

3. Use of the new protocol, OPT, either by itself, or in conjunction with other standard optimizations, leads to significantly improved performance over the standard algorithms for all the workloads and system configurations considered in this paper. Its good performance is attained primarily due to its reduction of the blocking arising out of locks held on prepared data. A feature of the OPT protocol design is that it limits the abort chain to one, thereby preventing cascading aborts. This is achieved by allowing only prepared transactions to lend their data and ensuring that borrowers cannot enter the prepared state until their borrowing is concluded.

   The power of the optimistic access feature is substantial enough that OPT's peak throughput performance was often close to that obtained with the DPCC baseline, which represents an upper bound on the achievable peak performance.

4. An attractive feature of OPT is that it can be integrated, often synergistically, with most other optimizations proposed earlier. In particular, in the few experiments wherein PC or PA performed noticeably better than 2PC, using OPT-PC or OPT-PA resulted in the best performance. The only exception wherein OPT does not appear to combine well is with protocols that allow cohorts to enter the prepared state unilaterally and therefore run the risk of having to revert to a processing state in case the master subsequently sends additional work to the cohort (for example, the Unsolicited Vote protocol of Distributed Ingres [Sto79]).

5. OPT's design is based on the assumption that transactions that lend their uncommitted data will almost always commit. However, OPT is reasonably robust in that it maintains its superior performance until the probability of surprise transaction aborts in the commit phase exceeds as high a value as fifteen percent. Beyond this level, its performance becomes progressively worse than that of the classical protocols.

6. Under conditions where there is sufficient data contention in the system, a combination of OPT and 3PC provides better throughput performance than the standard 2PC-based blocking protocols. This suggests that it would be possible for distributed database systems that are operating in high contention situations and currently using 2PC-based protocols, to switch over to OPT-3PC, thereby obtaining the superior performance of OPT during normal processing and, in addition, acquiring the highly desirable non-blocking feature of 3PC.

While the above experimental results demonstrated the utility of OPT, we also showed how OPT could be implemented without requiring inter-site communication or coordination and with only minimal simple changes to the database managers at each site.

In summary, we suggest that distributed database systems currently using the 2PC, PA or PC commit protocols may find it beneficial to switch over to using the corresponding OPT algorithm, that is, OPT, OPT-PA or OPT-PC. Specifically,

- if blocking (due to failures) is acceptable, then OPT is preferable under all conditions except two:
    1. under large degree of distribution and substantial resource constraints – OPT-PC is preferable.
    2. when the percentage of surprise aborts is high – OPT-PA is preferable.
- if having non-blocking functionality is desirable, but 3PC has not been used due to its excessive overheads, then OPT-3PC appears to be an attractive choice since it provides a peak throughput that exceeds those of the standard 2PC protocols.

Viewed in toto, OPT is a portable (can be used with many other optimizations), practical (easy to implement), high-performance (substantially increases throughput performance), efficient (makes good use of the lending approach), and robust (can handle practical levels of surprise aborts) distributed commit protocol.

9.1 Future Work

While we hope that the results described above would serve to help distributed DBMS designers to make more informed choices, there are a number of ways by which this work could be extended to cover a wider range of environments: First, only a two-level transaction tree was considered in our work. The effects of the commit protocols on the performance of a system with "deep-transactions" (tree of processes architecture) need to be evaluated. Second, our study focussed on transaction workloads where all cohorts were either complete or partial updaters – it would be useful to include the effects of read-only transactions. Third, the workload was homogeneous in that the transaction inputs at all the sites had the same characteristics. Modeling heterogeneous workloads would be of considerable utility in capturing system dynamics. Fourth, it would be instructive to model failures and evaluate what failure rates and durations would result in the non-blocking protocols delivering performance similar to that of the blocking protocols. Finally, it would be interesting to implement OPT in a real database system and quantitatively measure its performance benefits "in the field".

Acknowledgements

This work was supported in part by research grants from the Dept. of Science and Technology, Govt. of India, and from the National Science Foundation of the United States under grant IRI - -9619588.

References

[AA90] D. Agrawal and A. Abbadi, "Locks with Constrained Sharing", *Proc. of 9th Symp. on Principles of Database Systems (PODS)*, April 1990.

[AC95] Y. Al-Houmaily and P. Chrysanthis, "Two-Phase Commit in Gigabit-Networked Distributed Databases", *Proc. of 8th Intl. Conf. on Parallel and Distributed Computing Systems*, September 1995.

[AC96] Y. Al-Houmaily and P. Chrysanthis, "The Implicit-Yes Vote Commit Protocol with Delegation of Commitment", *Proc. of 9th Intl. Conf. on Parallel and Distributed Computing Systems*, September 1996.

[ACL87] R. Agrawal, M. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Trans. on Database Systems*, 12(4), 1987.

[ACL97] Y. Al-Houmaily, P. Chrysanthis and S. Levitan, "An Argument in Favor of the Presumed Commit Protocol", *Proc. of 13th IEEE Intl. Conf. on Data Engineering*, April 1997.

[Bha87] B. Bhargava, (editor), *Concurrency and Reliability in Distributed Database Systems*, Van Nostrand Reinhold, 1987.

[BC95] S. Banerjee and P. Chrysanthis, "Data Sharing and Recovery in Gigabit-Networked Databases", *Proc. of 4th Intl. Conf. on Computer Communications and Networks*, September 1995.

[BC96] S. Banerjee and P. Chrysanthis, "A Fast and Robust Failure Recovery Scheme for Shared-Nothing Gigabit-Networked Databases", *Proc. of 9th Intl. Conf. on Parallel and Distributed Computing Systems*, September 1996.

[BHG87] P. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[CKL90] M. Carey, S. Krishnamurthi and M. Livny, "Load Control for Locking: The 'Half-and-Half' Approach", *Proc. of 9th Symp. on Principles of Database Systems*, April 1990.

[CL88] M. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication", *Proc. of 14th Intl. Conf. on Very Large Data Bases*, August 1988.

[CL89] M. Carey and M. Livny, "Parallelism and Concurrency Control Performance in Distributed Database Machines", *Proc. of ACM SIGMOD Conf.*, June 1989.

[CL91] M. Carey and M. Livny, "Conflict Detection Tradeoffs for Replicated Data", *ACM Trans. on Database Systems*, 16(4), 1991.

[Coo82] E. Cooper, "Analysis of Distributed Commit Protocols", *Proc. of ACM Sigmod Conf.*, June 1982.

[Esw76] K. Eswaran et al, "The Notions of Consistency and Predicate Locks in a Database Systems", *Comm. of ACM*, 19(11), 1976.

[FHRT93] P. Franaszek, J. Haritsa, J. Robinson and A. Thomasian, "Distributed Concurrency Control based on Limited Wait-Depth", *IEEE Trans. on Parallel and Distributed Systems*, November 1993.

[Gra78] J. Gray, "Notes on Database Operating Systems", *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, 60, 1978

[Gra81] J. Gray, "The Transaction Concept: Virtues and Limitations", *Proc. of 7th Intl. Conf. on Very Large Data Bases*, 1981.

[Gup97] R. Gupta, "Commit Processing in Distributed On-Line and Real-Time Transaction Processing Systems", *M.Sc.(Engg.) Thesis*, SERC, Indian Institute of Science, March 1997.

[GHRS96] R. Gupta, J. Haritsa, K. Ramamritham and S. Seshadri, "Commit Processing in Distributed Real-Time Database Systems", *Proc. of 17th IEEE Real-Time Systems Symposium*, December 1996.

[GHR97] R. Gupta, J. Haritsa and K. Ramamritham, "More Optimism about Real-Time Distributed Commit Processing", *Proc. of 18th IEEE Real-Time Systems Symposium*, December 1997.

[Koh81] W. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", *ACM Computing Surveys*, 13(2), June 1981.

[KLS90] H. Korth, E. Levy and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *Proc. of 16th VLDB Conf.*, August 1990.

[Lin84] B. Lindsay et al, "Computation and Communication in R*: A Distributed Database Manager", *ACM Trans. on Computer Systems*, 2(1), 1984.

[LAA94] M. Liu, D. Agrawal and A. El Abbadi, "The Performance of Two-Phase Commit Protocols in the Presence of Site Failures", *Proc. of 24th Intl. Symp. on Fault-Tolerant Computing*, June 1994.

[LKS91a] E. Levy, H. Korth and A. Silberschatz, "An optimistic commit protocol for distributed transaction management", *Proc. of ACM SIGMOD Conf.*, May 1991.

[LKS91b] E. Levy, H. Korth and A. Silberschatz, "A theory of relaxed atomicity", *Proc. of ACM Symp. on Principles of Distributed Computing*, August 1991.

[LL93] B. Lampson and D. Lomet, “A New Presumed Commit Optimization for Two Phase Commit”, *Proc. of 19th Intl. Conf. on Very Large Data Bases*, August 1993.

[LM94] M. C. Little and D. L. McCue, *C++SIM User's Guide Public Release 1.5*, Dept. of Computing Science, Univ. of Newcastle upon Tyne, U.K., 1994.

[LS76] B. Lampson and H. Sturgis, “Crash Recovery in a Distributed Data Storage System”, *Tech. Report*, Xerox Palo Alto Research Center, 1976.

[Moh92] C. Mohan et al, “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging”, *ACM Trans. on Database Systems*, 17(1), 1992.

[MD94] C. Mohan and D. Dievendorff, “Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queuing” *Data Engineering Bulletin*, 17(1), 1994.

[MLO86] C. Mohan, B. Lindsay and R. Obermarck, “Transaction Management in the R^* Distributed Database Management System”, *ACM Trans. on Database Systems*, 11(4), 1986.

[OV91] M. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, 1991.

[RS96] R. Rastogi and A. Silberschatz, “On Commutativity Relationship for Atomicity and Serializability in Database Systems”, *Proc. of IEEE Intl. Conf. on Data Engineering*, 1996.

[SBCM95] G. Samaras, K. Britton, A. Citron and C. Mohan, “Two-Phase Commit Optimizations in a Commercial Distributed Environment”, *Journal of Distributed and Parallel Databases*, 3, 1995 (also in *Proc. of 9th Intl. Conf. on Data Engineering (ICDE)*, 1993).

[SC90] J. Stamos and F. Cristian, “A Low-Cost Atomic Commit Protocol”, *Proc. of 9th Symp. on Reliable Distributed Systems*, October 1990.

[SC93] J. Stamos and F. Cristian, “Coordinator Log Transaction Execution Protocol”, *Journal of Distributed and Parallel Databases*, 1(4), 1993.

[SJR91] P. Spiro, A. Joshi and T. Rengarajan, “Designing an Optimized Transaction Commit Protocol”, *Digital Technical Journal*, 3(1), 1991.

[SLKS92] N. Soparkar, E. Levy, H. Korth and A. Silberschatz, “Adaptive Commitment for Real-Time Distributed Transactions”, *TR-92-15, CS, Univ. of Texas (Austin)*, 1992.

[Ske81] D. Skeen, “Nonblocking Commit Protocols”, *Proc. of ACM SIGMOD Conf.*, June 1981.

[Sto79] M. Stonebraker, “Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES”, *IEEE Trans. on Software Engg.*, 5(3), 1979.

[Tri82] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall, 1982.

[Yoo94] Y. Yoon, “Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems”, *Ph.D. Thesis*, Korea Adv. Inst. of Science and Technology, May 1994.