Commit Processing in Distributed On-Line and Real-Time Transaction Processing Systems

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

By
Ramesh Kumar Gupta

Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

March 1997

To

My Mother

Six thousand years ago,

the Sumerians invented writing

for transaction processing

Transaction Processing: Concepts and Techniques – *Gray and Reuter*

Abstract

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 transaction 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. These include the classical *two phase commit (2PC)* protocol, its variations such as *Presumed Commit* and *Presumed Abort*, *nested 2PC*, *broadcast 2PC* and *three phase commit.* 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. Due to these costs, commit processing can result in a significant increase in transaction execution times, and therefore the choice of commit protocol becomes an important decision in the design of a distributed database system. Surprisingly, however, no systematic studies are available on the relative performance of these protocols with respect to their *quantitative* impact on transaction processing performance, rendering it difficult for designers to make an informed choice. In this thesis, we address this lacuna for two kinds of distributed database systems: (1) Distributed On-Line Transaction Processing Systems (OLTP), and (2) Distributed Real-Time Database Systems (RTDB). A special feature of our study is that we consider both *blocking* commit protocols, wherein site or network failures can lead to indefinite transaction blocking, and *non-blocking* protocols, wherein blocking is eliminated for most failure conditions but at the cost of incurring extra overheads during normal processing.

Distributed OLTP systems are vital to a variety of business applications that include banking, transportation, electronic commerce, etc. For these systems, *transaction throughput* is usually the major performance criterion. We study here the transaction throughput performance of a representative suite of previously proposed commit protocols by using a detailed simulation model of a distributed OLTP system. We also propose and evaluate a new commit protocol, called **OPT**, that allows transactions to "optimistically" borrow uncommitted data in a controlled manner. The OPT protocol is easy to implement and incorporate in current systems, and can coexist with most of the other optimizations proposed earlier. For example, OPT can be combined with current industry standard commit protocols such as Presumed Commit or Presumed Abort.

The experimental results show that distributed *commit* processing can have considerably more influence than distributed *data* processing on the throughput performance in distributed OLTP systems. Moreover, the choice of commit protocol clearly affects the magnitude of this influence. Among the protocols evaluated, the new optimistic commit protocol, OPT, provides the best transaction processing performance for a variety of workloads and system configurations. In fact, OPT's peak throughput performance is often close to that of an equivalent centralized system, and more interestingly, a non-blocking version of OPT exhibits better peak throughput performance than all of the standard blocking protocols evaluated in our study.

While OLTP systems cater to a large segment of business applications, Real-Time Database (RTDB) systems cater to applications wherein transactions have specific *timing constraints*, usually in the form of completion *deadlines*, and the goal of the system is to meet these deadlines, that is, to process transactions before their deadlines expire. Such real-time applications include aerospace and military systems, computer integrated manufacturing, stock markets, etc. Many of these applications are inherently distributed in nature. We investigate here the performance implications of supporting transaction atomicity in distributed real-time database systems. In particular, we consider applications with "firm-deadlines"—for such applications, completing a task after its deadline has expired is of no utility and may even be harmful. Therefore, transactions missing their deadlines are immediately discarded from the system without being executed to completion. The performance goal in a firm-deadline RTDB system is to minimize "deadline miss percent", that is, the percentage of transactions missing their deadlines.

Using a detailed simulation model of a distributed firm-deadline real-time database system, we profile the real-time performance of a representative suite of previously proposed commit protocols that are customized for the real-time domain. We also present and evaluate a new commit protocol, called **RT-OPT**, that is similar to OPT (the optimistic commit protocol proposed in the context of distributed OLTP systems) in its basic design but incorporates additional optimizations that cater to the special features of the real-time domain.

The results obtained from these simulation experiments are similar in flavor to those obtained for distributed OLTP systems: First, distributed *commit* processing can have considerably more influence than distributed *data* processing on the deadline miss percentage. Second, the choice of commit protocol clearly affects the magnitude of this influence. Third, the new real-time optimistic commit protocol, RT-OPT, provides the lowest deadline miss percentage among the real-time commit protocols evaluated in our study, for a variety of workloads and system configurations. Finally, a non-blocking version of RT-OPT exhibits better real-time performance than all of the standard blocking real-time commit protocols evaluated in our study.

In summary, the new optimistic commit protocols, OPT and RT-OPT, which permit controlled access to uncommitted data, provide significantly better transaction processing performance as compared to standard commit protocols proposed in the literature, for distributed OLTP systems and distributed RTDB systems, respectively. Moreover, they can provide the highly desirable non-blocking functionality at a relatively modest cost. Finally, they can be easily incorporated into existing systems.

Publications

1. Ramesh Gupta, Jayant Haritsa and Krithi Ramamritham, "Revisiting Commit Processing in Distributed Database Systems", *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997, *(to appear)*.

2. Ramesh Gupta, Jayant Haritsa, Krithi Ramamritham and S. Seshadri, "Commit Processing in Distributed Real-Time Database Systems", *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, USA, December 1996, pages 220-229.

3. Ramesh Gupta and Jayant Haritsa, "Commit Processing in Distributed Real-Time Database Systems", *Proceedings of the National Conference on Software for Real-Time Systems*, Cochin, India, January 1996, pages 195-204.

Acknowledgements

And name it gratitude, the word is poor. - George Meredith.

It is a difficult task to express so much of thankfulness in just a couple of pages! This work in its present shape would not have been possible without the constant help and support of many.

Words fail to express my gratitude to Dr. Jayant Haritsa. He has been more than an able advisor. The vastness of his knowledge and the zeal with which he keeps on updating it is simply admirable. And so is his attention to the finest details of the problem, and constant encouragement to strive for quality. During discussions he always encouraged independence of thought, reasoning and critical appraisal. Thanks for providing us with an excellent working environment and a very fine lab. And many many thanks for putting extra evenings and nights when the deadlines were staring in the face. Firm-deadlines I mean :-)

Prof. Balakrishnan has been very kind and always ready to extend his help in time of need. I could meet many deadlines only because he went out of his way to provide me with the necessary computing and other facilities. And he provided us with excellent facilities, including a lab just for the students! SERC have been the most pleasurable place to work—including net-surfing.

Discussions with Prof. Krithi Ramamritham of University of Massachusetts, Amherst have been instrumental in the progress of this work from the very beginning. My special thanks to Krithi. Thanks to Dr. Seshadri of IIT Bombay for the discussions and valuable inputs during the initial stages of the work. Thanks to Dr. Panos Chrysanthis of University of Pittsburgh for providing us a great deal of information about the latest research work on the subject, and for making available many excellent papers. Thanks to Dr. C. Mohan of IBM Almaden Research

Center for sending us many important IBM Research Reports, and to Dr. Jim Gray for finding time to answer many of my queries on transaction management.

Thanks to SERC faculty for being helpful and providing valuable comments during the work presentation. Special thanks to MJT. Office staff was very cooperative whenever I needed their help. Thanks to Rajlakshmi, Sarala, Todur, Triveni, and all other staff of SERC for their all the help.

Many thanks to Arun, Krishna, and Binto for being a nice colleague and also for critically reviewing manuscripts. Thanks to Shrividya for giving many valuable comments on the manuscripts. Thanks to Kalai, S. Balki and MBK for providing nice company during my stay here. Thanks to Saroja, Muthu, Pramod, Ramanathan, Ramanan, Madhavi, Neeharika and Abhijit for providing a nice ambient in SSL. Thanks to DSL members for bearing with me while I was writing the thesis.

My stay at the institute was made more sweeter by the efforts of some special persons. Saji, Sandeep, Uma – because of you people I never felt I was one and a half thousand miles away from home. Thanks to Sidhu, Gopi, Nandu, and FRK – the times spent with them have been just great! And they helped me at a time when I knew none in the institute. Thanks Bhaskar for many enlightening discussions. And one thing I am proud of getting here is the friendship of Manoj. He is truly a remarkable person.

Many more out there – mentioning just names sure isn't sufficient to acknowledge their help and support during my stay at the institute. Thanks a lot to all of them.

Frankly there wouldn't have been any thesis of mine without the encouragement, support and love my parents and my sisters have given me. I wish I knew some way of returning even a fraction of what they have given me.

And the last, but not the least, many thanks to Jim Gray and Andreas Reuter for writing an excellent book on the subject, and to the LaTeX team for providing a truly enjoyable way of word processing.

Contents

List of Figures

List of Tables

CHAPTER

1 Introduction

A **transaction** is the fundamental unit of processing in a database management system. Transfer of money from one account to another, reservation of train tickets, filing of tax returns, entering marks on a student's grade sheet, are all examples of transactions. The primary feature of a transaction is that it is an *atomic* unit of work that is either completed in its entirety or not done at all. The successful execution of the transaction results in the transaction being *committed*, which means that its effects on the database are permanent regardless of possible subsequent system failures. If, for any reason, it is not possible to commit the transaction, all the effects arising out of the partial execution of the transaction are removed from the database, and the transaction is said to have been *aborted*. In short, a transaction is an "all or nothing" unit of execution.

For a variety of reasons, many database applications store their data *distributed* across multiple sites that are connected by a communication network. In this environment, a single transaction may have to execute at many sites, based on the locations of the data that it needs to process. A potential problem associated with distributed transaction execution is that some sites could decide to commit the transaction while the others could decide to abort it, resulting in a violation of transaction atomicity. To address this problem, distributed database systems use a transaction **commit protocol**. A commit protocol ensures the uniform commitment of the distributed transaction, that is, it ensures that all the participating sites agree on the final outcome (commit or abort) of the transaction. Most importantly, this guarantee is valid even in the presence of site or network failures.

Over the last two decades, a variety of distributed transaction commit protocols have been proposed by database researchers. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites (where the distributed transaction executes). 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 these costs, commit processing can result in a significant increase in transaction execution times [SJR91, LL93, SBCM95], making the choice of commit protocol an important design decision for distributed database systems.

In light of the above discussion, 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. Surprisingly, however, adequate studies of the relative merits of these protocols with respect to their *quantitative* impact on transaction processing performance are not available. This is a significant lacuna since transaction processing performance is usually a primary concern for database system designers.

In this thesis, we address the above lacuna by quantitatively profiling the performance of a representative suite of distributed transaction commit protocols. Our evaluation is made in the context of two important categories of distributed database systems: (1) Distributed On-Line Transaction Processing Systems (**OLTP**), and (2) Distributed Real-Time Database Systems (**RTDB**).

Performance Issues

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 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: A commit protocol is said to be *non-blocking* if, in the event of a site failure, it permits transactions that were being processed at the failed site to terminate at the operational sites without waiting for the failed site to recover [Ske81, BHG87, ÖV91]. With blocking protocols, there is a possibility of transaction processing grinding to a halt in the presence of failures (as explained in Section 3.6). Non-blocking protocols, on the other hand, are designed to ensure that such major disruptions do not occur. To achieve their functionality, however, they usually incur additional messages and force-writes than their "blocking" counterparts.[1]

Of the three issues highlighted above, we believe, from a performance perspective, that the first two issues (effect on normal processing and resilience to failures) are of primary importance 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. With this viewpoint, we focus here on the *mechanisms* required during normal operation to provide the functionality of recovery, rather than on the recovery *process* itself.

As mentioned earlier, our performance evaluation is made in the context of two important categories of distributed database systems: (1) Distributed On-Line Transaction Processing Systems (OLTP), and (2) Distributed Real-Time Database Systems (RTDB). We now briefly overview their essential features and discuss the issues involved in commit processing in these systems.

[1]Unfortunately, it is impossible to design commit protocols that are completely immune, in the non-blocking sense, to both site and link failures [BHG87]. However, the number of simultaneous failures that can be tolerated before blocking arises varies across different commit protocols.

1.1 Distributed OLTP Systems

On-Line Transaction Processing Systems cater to database applications that need to access their data "on the fly". Examples of such applications include banking, transportation, electronic commerce, etc. The performance goal in these systems is to maximize the **transaction processing throughput** and their design is tuned towards this end. A variety of transaction commit protocols have been proposed for distributed OLTP systems by researchers from both industry and academia (see [Koh81, Bha87, ÖV91] for surveys). These include the classical *two-phase commit (2PC)* protocol [LS76, Gra78], its variations such as *presumed commit* and *presumed abort* [MLO86, LL93], *linear 2PC* [Gra78], *distributed 2PC* [ÖV91] and *three-phase commit (3PC)* [Ske81]. In general, two-phase commit protocols are susceptible to blocking whereas three-phase commit protocols are non-blocking.

Contributions

In this thesis, we quantitatively investigate the performance implications of supporting transaction atomicity in distributed OLTP systems by performing an extensive set of simulation experiments. 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.

   To isolate and quantify the effects of supporting data distribution and achieving transaction atomicity on system performance, we use two baselines in the simulations: (a) a centralized system, and (b) 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 transactions to "optimistically" borrow dirty

(uncommitted) data. Although 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 with most of the other optimizations proposed earlier. For example, OPT can be combined with current industry standard commit protocols such as Presumed Commit or Presumed Abort.

Salient observations from our simulation experiments include:

- Distributed *commit* processing for most workloads 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 transaction processing performance, doing considerably better than the classical protocols, for a variety of workloads and system configurations. In fact, OPT's peak throughput performance was often close to that obtained with the "distributed processing, centralized commit" baseline mentioned above, which in a sense represents an upper bound on throughput performance of distributed commit protocols.

- 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 2PC-based standard blocking protocols. This suggests that it would be possible for distributed database systems that are operating in high data 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.

- OPT's design is based on the assumption that transactions that lend their uncommitted data will almost always commit. We have found, however, that the performance of OPT in distributed OLTP systems is *robust* in that even if transactions abort in the commit phase (due to violation of integrity constraints, software errors, system failures, etc.), OPT maintains its superior performance unless the probability of such aborts is very high. In our experiments, OPT exhibited superior performance even when the transaction abort probability was as high as 10 percent. A *performance crossover* was observed at the transaction abort probability of 15 percent—a level that is much higher than what might be expected in practice. Beyond this level, OPT's performance becomes progressively worse than that of the classical protocols.

1.2 Distributed RTDB Systems

Real-Time Database Systems cater to database applications that impose *timing constraints* on the database system processing, usually in the form of transaction *completion deadlines.* Such applications include aerospace and military systems, computer integrated manufacturing, robotics, nuclear power plants, traffic control systems, stock markets, telephone-switching systems, and network management [Son90, Ulu94a]. The performance goal in these systems is to maximize the *number* of transactions that complete before their deadlines expire. That is, the important issue is whether or not a transaction completes before its deadline, but not how soon it completes before its deadline. Therefore, in contrast to the OLTP systems described above where the goal is to maximize transaction throughput, the emphasis in an RTDB system is on maximizing the number of transactions that meet their deadlines.

Many of the RTDB applications, especially in the areas of stock markets, communication systems and military systems, are inherently *distributed* in nature. Accordingly, there is a need to use transaction commit protocols to provide global transaction atomicity. Unfortunately, however, the standard transaction commit protocols used for distributed OLTP systems cannot be directly used in distributed RTDB systems. This is because they are not cognizant of transaction timing constraints. Therefore, the design of real-time transaction commit protocols has to be considered afresh.

For designing commit protocols for a distributed RTDB system, we need to address two major questions: First, how do we adapt the standard commit protocols into the real-time domain? Second, how do the real-time variants of the commit protocols compare in their performance? We address these questions in this thesis. In particular, we consider applications that have *firm-deadlines* [HCL92]. For such applications, completing a transaction after its deadline has expired is of no utility and in fact may be harmful. Therefore, late tasks are considered to be worthless and are immediately “killed”, that is, aborted and discarded from the system without being executed to completion. The performance goal in a firm-deadline RTDB system is to minimize “deadline miss percent”, that is, the steady-state percentage of transactions missing their deadlines.

Contributions

In this thesis, we quantitatively investigate the performance implications of supporting transaction atomicity in distributed RTDB systems by performing an extensive set of simulation experiments. Our contributions are three-fold:

1. We first precisely define the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a firm-deadline distributed real-time database system. Subsequently, we customize the conventional commit protocols (that is, the commit protocols that were proposed in the context of distributed OLTP systems) for the firm-deadline RTDB environment.

2. Using a detailed open model of a distributed firm-deadline real-time database system, we profile the real-time performance of a representative suite of commit protocols that are customized for the real-time domain. *To the best of our knowledge, these are the first simulation based quantitative results in this area.*

3. We present and evaluate a new commit protocol, called **RT-OPT**, that is similar to OPT (the optimistic commit protocol proposed in the context of distributed OLTP systems) in its basic design but incorporates additional optimizations that cater to the special features of the real-time domain.

The results obtained from these experiments are similar in flavor to those obtained for distributed OLTP systems:

- Distributed *commit* processing can have considerably more influence than distributed *data* processing on the deadline miss percentage.
- The new real-time optimistic commit protocol, RT-OPT, provides the lowest deadline miss percentage among the real-time commit protocols evaluated in our study, for a variety of workloads and system configurations.
- A non-blocking version of RT-OPT exhibits better real-time performance than all of the standard blocking real-time commit protocols evaluated in our study.

1.3 Organization

The thesis is organized in two parts: Part I presents the work in the context of a distributed OLTP system, and Part II presents this study for a distributed RTDB system. Chapters 2 through 6 constitute Part I while Part II consists of Chapters 7 through 12.

Chapter 2 presents a brief overview of OLTP systems. An introduction of transaction semantics and of the database mechanisms that support these semantics is given in this chapter. A detailed description of a variety of commit protocols is provided in Chapter 3. In this chapter, we also discuss many optimizations of the commit protocols proposed in the literature and describe in detail OPT, the newly proposed protocol. The simulation model used for evaluating the various protocols is described in Chapter 4. The results of our experiments for distributed OLTP systems are presented and discussed in Chapter 5. We conclude with a summary of these results in Chapter 6.

An overview of RTDB systems, and a review of the research in this area is presented in Chapter 7. In Chapter 8 we discuss the issues involved in adapting the conventional commit protocols into the real-time framework. We also describe the new real-time commit protocol RT-OPT in this chapter. The simulation model for the distributed RTDB system is described in Chapter 9, and the results of the experiments are presented in

Chapter 10. We discuss some further optimizations applicable to RT-OPT and their tradeoffs in Chapter 11. We conclude with a summary of these results in Chapter 12.

Finally, in Chapter 13, we provide an overall summary of the work presented in this thesis and identify future research avenues.

PART I

On-Line Transaction Processing Systems

CHAPTER

2

OLTP Systems: An Overview

In todays *information revolution* age, fast access to information and its efficient management are the key to the success of many businesses. The amount of the data involved is usually *huge*—running into gigabytes and terabytes. The Data Base Management Systems (DBMS) play an important role in managing the information for business requirements. DBMSs manage huge volumes of data efficiently for a large variety applications. The business applications today are not the old-styled *batch* applications—businesses have been moving their data processing activities *on-line* for over three decades now. Many businesses such as airline reservation systems and banks are no longer able to function when their on-line computer systems are down. These on-line systems involve thousands of terminals and hundreds of computers, and their databases must be up-to-date and available at all times.

End users of these on-line systems interact with the system through *transactions*. A transaction is an *atomic* unit of work that is either completed in its entirety or not done at all. In other words, if the transaction is completed, its effects on the database are permanent—in-spite of the possible subsequent failures in the system—and the transaction is said to have been *committed*. If it is not possible to commit the transaction, all its effects are wiped out from the database as if the transaction was never started, and the transaction is said to have been *aborted*. It is the responsibility of the *On-Line Transaction Processing Systems (OLTP)* used in the business applications to guarantee these requirements of transaction atomicity.

For more than two decades, both industry and research community have been showing great interest in OLTP systems [Cod70, EGLT76, LS76, Gra78, Sto79, ML83, MSF83, MLO86, MBCS92, MHL+92, MN94, Moh93, WDH+82, SKPO88, FZT+92, She93, DGS+90, SJR91, WD95]. The two concepts that have revolutionized the OLTP technology are the *Relational Model* and the *Transaction Model*. Relational Model [Cod70] paved the way for the formalization of the database systems, providing a solid theoretical foundation—a very important prerequisite for the proper understanding of any system. Transaction Model [Gra78, Gra81] formalized the notions of database consistency and transaction atomicity, making database systems attractive for many businesses. In the next section, we discuss the transaction properties and the mechanisms to ensure these properties for the *centralized* systems. Then we will discuss the mechanisms that provide these transaction properties for a *distributed* system.

2.1 Transaction Model

A transaction is characterized by the following properties [EN94]:

Atomicity: A transaction is an atomic unit of work, that is, effectively either all or none of the transaction's operations are performed. If all the operations have been performed successfully, the transaction commits. If some operation of the transaction fails, the partial results of the transaction are undone and the transaction aborts. Thus, it gives an illusion that the transaction either completed successfully or was not even started.

Consistency: The consistency of a transaction simply means its correctness. An individual execution of a transaction must take the database from one consistent state to another consistent state. It is usually the responsibility of application programmer to ensure the correctness of the transaction.

Isolation: An incomplete transaction cannot make its database modifications visible to other transactions before its commitment. Violation of this property may lead to *cascading aborts* [BHG87] of the transactions.

Durability: Once a transaction has committed, the system must guarantee that the effects of the transaction will never be lost despite subsequent failures of the system.

Serializability: The concurrent execution of a set of transactions is equivalent to some serial execution of the same set of transactions. Guaranteeing serializability lets the transaction programmer write the transaction in its individuality without worrying about other transactions that may be executing concurrently.

These transaction properties, usually called as *ACIDS*, are provided by the *concurrency control manager (CCM)* and the *recovery manager (RM)* components of the transaction management system. The CCM implements a *concurrency control protocol* to provide the properties of isolation and serializability, and the RM implements a *commit protocol* to provide the properties of atomicity and durability. As mentioned above, it is usually the responsibility of the application programmer to ensure that a transaction is correct, i.e., consistency preserving.

2.1.1 Concurrency Control Protocols

The study of concurrency control protocols has received wide attention in the database research literature. Theoretical foundations of concurrency control mechanisms are discussed in [Pap79, BHG87]. Concurrency Control protocols can be broadly classified into three categories: *locking* protocols, *timestamp* based protocols, and *optimistic* protocols. Locking protocols lock the data items to prevent multiple transactions from accessing these data items concurrently. The two-phase locking (2PL) protocol and its proof of correctness is presented in [EGLT76]. A variation of the 2PL based on ordered sharing is proposed in [AA90, AA91]. A multiversion locking protocol that allows multiple versions of data in order to improve concurrency is discussed in [BC92]. Timestamp based protocols have been mainly studied in the SDD-1 system [BRGP78] (System for Distributed Databases). Optimistic approach for concurrency control is proposed in [KR81]. Optimistic concurrency control protocols (also called as *validation* or *certification* protocols) are characterized by three phases of the transaction: (1) a read phase, where transactions read values from the database but make the updates to the private copies of the data items, (2) a validation phase, where the transaction is validated to ensure that it does not violate the serializability, and (3) a write phase, where the private copy updates of the transaction are written to the database if the transaction had passed the validation test. If the transaction fails the validation test, the private copy updates of the transaction are discarded and the transaction is restarted. However, there are some practical difficulties related with the implementation of optimistic concurrency control protocols that are discussed in [Moh92, Hae84].

Deadlocks

The use of the two-phase locking protocol for concurrency control can result in a set of transactions forming a deadlock [KS91]. There is a vast literature on deadlocks spanning operating systems, distributed systems and database systems. A preliminary analysis of the probability of deadlocks in database systems is presented in [GHOK81], where they have shown that the probability of deadlocks is very small in a database system that is not thrashing.

Performance

There are many papers available that discuss the performance of the various concurrency control protocols. In particular, two-phase locking is prone to thrashing and this phenomenon is discussed and analyzed in [TGS85, FRT92, FR85, Tho93]. A load control scheme based on a "half-and-half" rule to avoid thrashing is proposed and evaluated in [CKL90]. Half-and-half rule is based on the hypothesis that the system starts thrashing when more than *50 percent* of the transactions in the system are waiting for the data locks held by other transactions. A simulation study of the performance of various concurrency protocols is given in [ACL87]. The performance of the ordered-shared locking protocol mentioned above is discussed in [AAL91, AAL94].

2.1.2 Recovery Protocols

The recovery manager provides the semantics of atomicity and durability to the transactions. *All* of the effects of the committed transactions and *none* of the effects of the aborted transactions must be visible to the database. This must be guaranteed even in the event of failures. After the system recovers from a failure, it has to be brought to a consistent state—a state where all the effects of the transactions that had committed before the system crash, and none of the effects of the transactions that were either aborted before the crash or executing at the time of the crash, are visible to the system.

The updates of some transactions that were executing at the time of crash could have reached the disk (stable storage) before the crash occurred. Atomicity requires these effects to be *undone* when the system recovers from the failure, because these transactions were not committed by the time crash occurred. Also, the updates of some transactions that had committed just before the crash might not have reached the disk—these were lost with the volatile storage as it was wiped out in the crash. These actions have to be *redone* after the system recovers from the crash to provide the durability of the committed transactions. Therefore, the system must have enough information on the stable storage before the crash occurs in order to be capable of carrying out these undo/redo operations. This simply means that the system must have the necessary information on the stable storage *at all times*—the crashes do not tell about their intentions beforehand!

Write-Ahead Logging and Commit Point

In order to ensure that the system has the undo/redo information (information required to undo/redo the actions of a transaction) in the stable storage *at all times*, the recovery manager uses a technique called *write-ahead logging (WAL)*. Before a data item is modified by a transaction, a log record is generated having sufficient information to undo or redo the effect of the modification. The system ensures that the log record, and not the updated data item, is the first to reach the stable storage. This is ensured by using *Log Sequence Numbers (LSN)* [GR93]. An efficient technique called ARIES that uses WAL and LSN mechanisms is described in [MHL$^+$92]. Thus, the WAL and the LSN mechanisms ensure that there is always sufficient information on the stable storage to undo/redo the effects of a transaction. Usually, these undo/redo log records are *idempotent*—their repeated application to a transaction is same as applying them only once. This becomes important, for example, when a site fails during the recovery process itself. When the site comes up, the recovery process might re-apply some of the redo/undo records that had been applied previously.

After a transaction has completed its execution and is ready to commit, a `COMMIT` log record is generated. This log record is *forced* to the stable storage, that is, written in a synchronous fashion (the transaction waits until this record reaches the disk). The LSN mechanism mentioned above ensures that all the WAL records belonging to the transaction reach the stable storage before this commit log record. Thus, when the commit log record reaches the disk, there is sufficient information on the disk to redo the effects of the transaction in case a failure occurs. The point at which the commit log record reaches the disk is called the `commit point` and the transaction is said to be *committed* at this point.

We see that the writing of the log record to the disk can be done in two ways: (1) synchronously, where the execution of the transaction is suspended until the writing of the record is over, and (2) asynchronously, where the execution of the transaction continues normally without waiting for the writing operation to be complete. Hereafter we will refer to the latter as simply a *write* and the former as a *force-write*. Note that the force-writes have a larger performance penalty than writes because they increase the response time of a transaction.

Recovery and Buffer Management

The choice of recovery mechanisms to be used when a site recovers from a failure enforces certain constraints on buffer management, directing buffer manager to force certain pages to the disk at particular times, and not allowing it to write certain pages to the disk at some other times. This issue is discussed in elaborate detail in [HR83, GR93]. A page modified by a transaction that has not yet committed is a *dirty* page until either the commit or the abort of the transaction is completed. Based on how the buffer manager handles the dirty and clean pages the buffer management policies can be classified as follows:

Steal/No-Steal Policy :

If the buffer manager is *not* required to distinguish the dirty pages from the clean pages when deciding which page to remove from the bufferpool, a dirty page can be written (stolen) from the buffer to the disk. This policy is called as *steal* policy. As the disk might contain dirty pages, the log must contain sufficient information for undoing the transaction effects if required. On the other hand, if the buffer manager uses a *no-steal* policy, the dirty pages are not allowed to be written to the disk and are retained in the bufferpool until the outcome of the transaction is determined. In this case, because the disk never contains the dirty pages, the log need not contain any information for the undo purposes. In case the transaction aborts, these buffer pages are simply marked as invalid.[1] However, the no-steal policy has a few disadvantages: it enforces the lock granularity to be page which is a bad idea, and it needs a prohibitively large bufferpool for long (batch) transactions.

Force/No-Force Policy :

If the buffer manager force-writes the pages modified by a transaction when the transaction is ready to commit, the policy is called as *force* policy. On the other hand if the buffer manager does not force-write the modified pages when the transaction commits, but replaces them to the disk only when required by the buffer replacement policy, the policy is called as *no-force* policy. The force policy has the advantage that the log need not contain any information for the redo purposes while such information is mandatory if the no-force policy is used. However, the force policy may require a frequently accessed page to be force-written to the disk many more times than the no-force policy.

[1]This policy, however, requires logging of undo information for on-line undo purposes (transaction aborts during normal operation). This undo log information need not be written to the disk. To avoid undo logging completely, a page modified by a committed transaction must be written to the disk before another transaction can dirty it [Gra96a].

In general, the steal/no-force combination seems the appropriate choice for practical systems [GR93].

The other mechanisms proposed in the literature for recovery in centralized systems include shadow paging [LS76], and the differential files [SL76]. However, due to some inherent problems in these techniques, commercial systems usually avoid them and instead prefer to implement the logging mechanisms [GR93]. A study of the performance issues that arise in the integration of various concurrency control protocols with the recovery protocols is given in [AD85].

2.2 Distributed OLTP Systems

Many of the OLTP applications (for example, airline reservation systems, banks) are distributed in nature. The data in these applications is distributed across multiple sites. Therefore, a transaction in these systems may need to access the data from many sites. The cohorts on behalf of the transaction are initiated at all the sites where it need to access the data. Because of this distributed nature of the transaction, many complications arise in the concurrency control and recovery algorithms. For example, distributed deadlocks may occur which may be hard to detect. Then, to maintain the atomicity of the transaction, uniform commitment of all the cohorts of the transaction has to be ensured. Moreover, any of the sites may fail, and we would still like the remaining sites to function properly without being affected by the failed site. Also, the database now may have multiple copies of the data items residing at different sites. The concurrency control algorithm must guarantee the consistency among these copies, and the recovery algorithm must ensure that the copy of the data at a site is brought up-to-date when the site that had crashed recovers from the failure.

2.2.1 Distributed Concurrency Control

A significant body of research literature is available on the distributed concurrency control protocols. For example, [BHG87] is a comprehensive book on concurrency control and recovery algorithms that covers most of the important issues in the distributed database systems. [Gra78] gives a definition of a distributed wait-for-graph for the detection of deadlocks, and proposes a centralized protocol to detect the global deadlocks. A detailed study of deadlock detection algorithms for distributed database systems is presented in [Kna87]. Timestamp based concurrency control protocols for SDD-1 are discussed in [BRGP78]. [Sto79] describes the concurrency control protocol used in distributed INGRES. A locking protocol based on limited wait-depth is proposed and evaluated in [FHRT93]. A detailed discussion on the issues in the distributed database systems is given in [CP84] and [ÖV91]. A collection of some important papers on distributed concurrency control is presented in [Bha87]. The performance studies of various distributed concurrency control algorithms have been reported in numerous papers, for example, [MN82, CL88, CL89, CL91, FHRT93].

2.2.2 Distributed Commit Protocols

To guarantee the uniform commitment of a distributed transaction, the recovery manager employs a *transaction commit protocol*. A commit protocol ensures that all the cohorts of the transaction 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 to ensure transaction atomicity (see [Koh81, Bha87, ÖV91] for surveys). These include the classical *two-phase commit (2PC)* protocol [LS76, Gra78], its variations such as *presumed commit* and *presumed abort* [MLO86, SBCM95, LL93], *linear 2PC* [Gra78], *distributed 2PC* [ÖV91] 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 executes. In addition, several log records are generated, some of which have to be force-written to the disk. Due to these costs, commit processing can result in a significant increase in transaction execution times [SJR91, LL93, SBCM95], making the choice of commit protocol an important design decision for distributed database systems.

2.3 Our Work

In light of the above discussion, it seems reasonable to expect that the results of detailed performance studies of commit protocols would be available to assist distributed database system designers in making an informed choice (as we noted above for the distributed concurrency control protocols). Surprisingly, however, most of the earlier performance studies of commit protocols (for example, [MLO86, LL93, AHC95, SBCM95]) have been limited to comparing protocols based on the *number* of messages and the *number* of force-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 very little attention. This is a significant lacuna since transaction processing performance is usually a primary concern for database system designers. Hence we address this issue in this dissertation.

As mentioned in the Introduction, in this part of the thesis, we quantitatively investigate the performance implications of supporting transaction atomicity in distributed OLTP systems. 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. To isolate and quantify the effects of supporting data distribution and achieving transaction atomicity on system performance, we use two baselines in the simulations: (a) a centralized system, and (b) 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 transactions to "optimistically" borrow dirty (uncommitted) data. Although 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 with most of the other optimizations proposed earlier. For example, OPT can be combined with current industry standard commit protocols such as **presumed commit** or **presumed abort**.

In the next chapter, we will discuss in detail the two-phase commit protocol, its variations (presumed abort and presumed commit protocols), and the three-phase commit protocol. We will then discuss the proposed protocol OPT and its various features.

CHAPTER

3 Distributed Commit Protocols

A distributed transaction executes at multiple sites. To ensure the atomicity of the distributed transaction, all cohorts of the transaction must reach a uniform decision (commit or abort). This guarantee is provided by the *transaction commit protocols*. In this chapter, we will discuss the classical two-phase commit protocol, its variations such as presumed abort and presumed commit protocols, and the three-phase commit protocol. Then, we will describe a new protocol, called OPT, that allows the transactions to access uncommitted data—in a controlled fashion—in order to improve the performance of the system.

3.1 Distributed Transaction Execution

A common model of distributed transaction execution is shown in Figure 3.1. At the site where the transaction is submitted, a *master* process is created to coordinate the execution of the transaction. At a site where the transaction needs to access data, a *cohort* process on behalf of the transaction is created. Usually, there is only one cohort on behalf of the transaction at each such site. The master sends the STARTWORK message to a cohort when some data at the site of the cohort is required to be accessed.[1] Depending on the

[1]Actually, the master sends the STARTWORK message to the transaction manager of the site. The transaction manager creates a cohort on behalf of the master, if one is not already existing. As these implementation details are not important for the understanding of the concepts presented here, we will ignore them for the sake of simplicity.

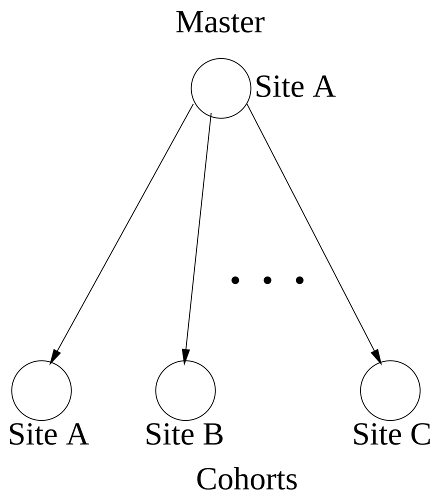


Figure 3.1: Distributed Transaction Execution

transaction architecture, the master may send the STARTWORK messages to multiple cohorts without waiting for their responses, that is, the cohorts of the transaction will execute in a *parallel* fashion. Or the master may send the subsequent STARTWORK message only after the previous cohort has completed the work assigned to it, that is, the cohorts of the transaction execute one after the other in a *sequential* fashion. A cohort, after successfully executing the master's request, sends a WORKDONE message to the master. After receiving the WORKDONE message, the master may send more work to the same cohort, or may decide to send the next piece of work to another cohort of the transaction.

The master may decide to conclude the transaction when all the work assigned to the transaction is completed. Or the decision may be necessitated at the instance of the user submitting the transaction. In any case, the decision to commit or abort the transaction must be uniform—the master and all cohorts must agree on a common decision. It might seem that the fact that a cohort had sent a WORKDONE message means that the cohort was willing to commit the transaction. The problems here are these: First, the sending of the WORKDONE message does not enforce any binding on the cohort to agree on the decision. Due to reasons such as concurrency control, performance, etc. the cohort could still be aborted even after the WORKDONE message was sent. Second, there can be failures—communication links may fail, or some of the sites hosting the cohorts of the transaction may fail. Therefore, a commit protocol is needed to ensure that all cohorts and the master reach a uniform decision—a decision that will be binding on all sites even if a failure occurs.

A variety of transaction commit protocols have been devised, most of which are based on the classical **two-phase commit (2PC)** protocol [Gra78]. The two-phase commit protocol is discussed in considerable detail in the next section.

A Digression about Failures

Two points are worth mentioning before we start the description of the 2PC protocol: First, how does a site know that a communication link or a remote site has failed? And second, how are failures handled? Generally, a timeout mechanism is used to address the first question. If the response to a message is not received from the remote site within the timeout period, the local site assumes that either the communication link to the remote site, or the remote site itself has failed.[2]. For the second question, when a site recovers from a failure, it is handed over to the recovery manager of the site. The recovery manager scans the log, and if it finds a `COMMIT` log record for a transaction, it knows that the transaction had committed before the failure occurred. In the same way, if it finds an `ABORT` log record, it knows that the transaction was aborted before the failure occurred. In these cases, the recovery manager ensures that the effects of the committed transactions are redone and the effects of the aborted transactions are undone as was explained in Section 2.1.2. The only problem is for the transactions about which neither a `COMMIT` nor an `ABORT` log record is found. The actions that recovery process need to take in this case depend on the commit protocol being used—we will discuss them with the description of the various commit protocols.

[2]Both site failures and the communication link failures manifest themselves in the loss of communication between the local and remote site. There is no way for the local site to tell whether it is a communication link failure, or the remote site has failed [BHG87].

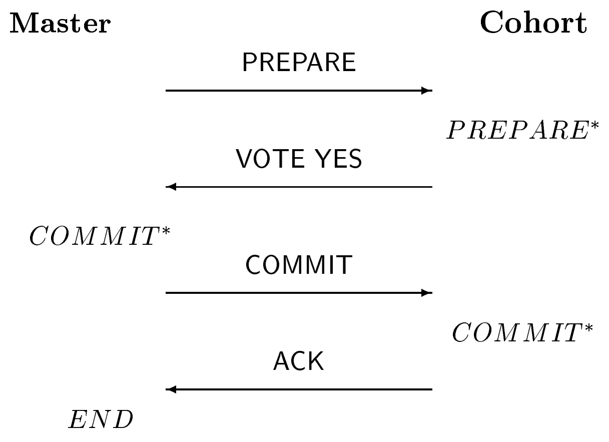


Figure 3.2: Two-Phase Commit Protocol for Committing Transactions

3.2 Two-Phase Commit Protocol

The two-phase commit protocol operates in two phases—as the name suggests. In the first phase, the master reaches a global decision (*commit* or *abort*) based on the local decisions of the cohorts. In the second phase, this decision is conveyed to the cohorts and implemented.

For sake of clarity, we first describe the protocol without considering failures. The first phase starts when the master decides to conclude the transaction. At this point, the master sends PREPARE messages in parallel to all its cohorts. The sending of the PREPARE message is an indication to the cohorts that the master would like to commit the transaction. On receiving the PREPARE message, a cohort willing to commit the transaction force-writes a PREPARE log record to its stable storage, and sends a YES vote to the master (as shown in Figure 3.2, the names on the arrows in the figure indicate the messages and the names in italics indicate the log records, an * with the name indicating that the log record is force-written). At this point, the cohort is said to be in 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, a cohort that decides to abort the transaction writes (note, not force-writes[3]) an ABORT log record and sends a NO vote

[3]Many research papers [LSG+79, MLO86, SC90, AHC95] assume that the ABORT log record mentioned

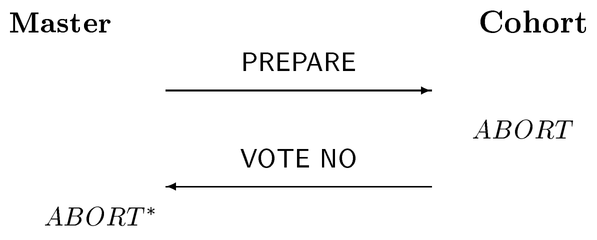


Figure 3.3: Two-Phase Commit when Cohort Votes NO

to the master (as shown in Figure 3.3). 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 the cohorts, it initiates the second phase of the protocol. If all the votes are YES, it moves to a committing state by force-writing a `COMMIT` log record and sending COMMIT messages to all the cohorts. Each cohort after receiving a COMMIT message moves to the committing state, force-writes a `COMMIT` log record, and sends an ACK message to the master. The master writes an `END` log record after receiving the ACKs from all the cohorts (Figure 3.2), and then forgets the transaction, that is, the control information about the transaction is removed from memory at the master's site.

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. The master writes and `END` log record after receiving the ACKs from those cohorts that were sent the ABORT message, and then forgets the transaction (Figure 3.4).

If a timeout occurs while the master is waiting for the vote from a cohort, it assumes

here is *force-written*. However, in 2PC, a record is force-written either to make a state persistent or to avoid the happening of the `no information` case (discussed in Section 3.2.2). As will be clear from the description in Section 3.2.3, *not* force-writing of this record does not lead to the occurrence of `no information` case, and persistence of the aborting state here is not required because the recovery process will abort such cohorts and the WAL mechanisms will take care of the undo aspects of aborting the transaction [Gra96b].

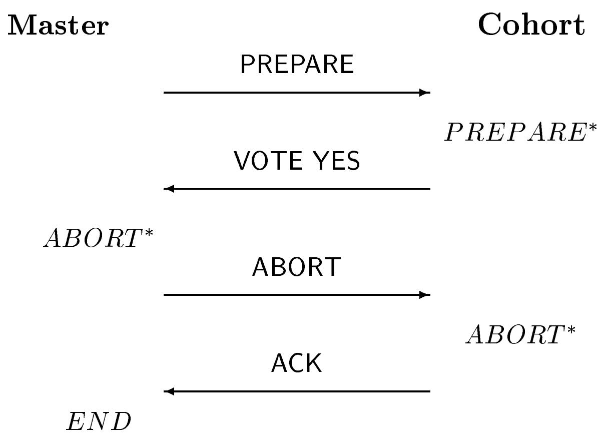


Figure 3.4: Two-Phase Commit when Cohort Votes YES but Master Decides Abort

a NO vote and aborts the transaction as described above (note that the ABORT message is sent to this cohort also to ensure that it is notified about the final decision about the transaction). If a cohort's site notices before receiving the PREPARE message from the master that the master's site has failed, the cohort is simply aborted. However, if the master fails after the cohort has dispatched the YES vote, the cohort will periodically try to contact the master until a decision from the master is received. In the same fashion, if master does not receive an ACK for the decision it has sent to the cohort, it will periodically re-send the decision to the cohort until it finally receives the ACK.

3.2.1 State Transitions

The master and the cohorts visit various states during the lifetime of a transaction. Let us assume, without loss of generality, that both the master and the cohorts are in an executing state before the two-phase commit protocol is initiated by the master. From this point, a cohort can move to either aborting state or prepared state depending upon whether it votes YES or NO to the master's request (as shown in Figure 3.5, the names on the transition arcs in the figure indicate the log records, an * with the name indicating that the log record is force-written). If it decides to vote YES, it force-writes a PREPARE log record before sending the message. The force-writing of the PREPARE record takes the

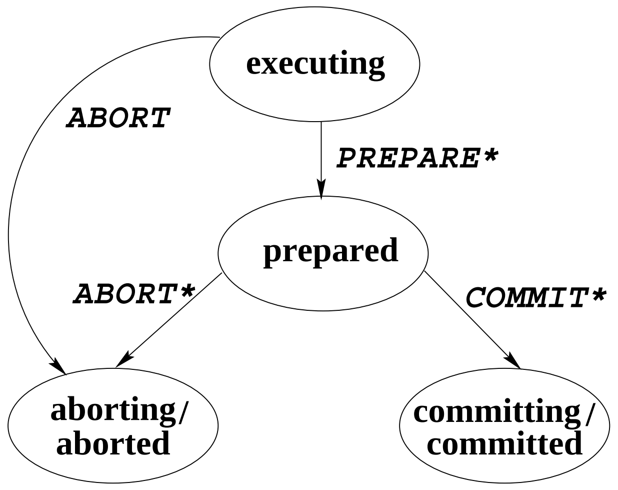


Figure 3.5: State Transitions in 2PC for the Cohort

cohort to the prepared state. Once a cohort is in prepared state, it can no longer *unilaterally* decide to move to the aborting state—it has to wait for the master's decision of commit or abort, based on which it will move to aborting or committing state by force-writing the appropriate log record.

The master makes the decision (commit or abort) about the fate of the transaction. Its state transitions are shown in Figure 3.6. It can move to the aborting state from the executing state without even sending PREPARE messages to the cohorts (for example, if the transaction is aborted by the user). It moves to the prepared state by sending the PREPARE messages to the cohorts. Note that this state transition does not require writing of any log record. From the prepared state, the master moves to the committing state (by force-writing a `COMMIT` log record) only if it gets all YES votes from the cohorts. Even if one NO vote is received, the master moves to the aborting state (by force-writing an `ABORT` log record). From these committing or aborting states, the master moves to the committed or the aborted states, respectively, by writing an `END` log record. The `END` log record is written only after the master has received all ACKs from the cohorts that were sent the decision. Note that the cohorts do not write any more log records (that is, no `END` record) after moving to committing or aborting states, therefore, for the cohorts, these states are equivalent to committed or aborted states, respectively.

Some of the states are *volatile*, for example, the executing state. The information

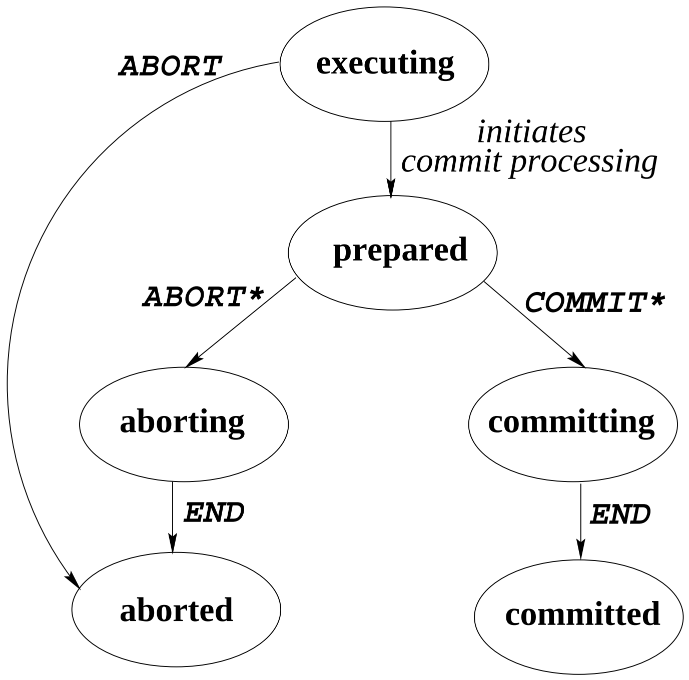


Figure 3.6: State Transitions in 2PC for the Master

about these states will not survive a system crash, and when the failed site recovers the transaction cannot resume these states. Some states are *persistent*, that is, they survive the system crash. This is because the log records before moving to these states are force-written. When the failed site recovers, the transaction can be brought back to these states by using the information available in the log. These states are prepared, committing, and aborting for the cohort, and committing and aborting for the master. The persistence of these states is required so that the recovery process can take a correct decision to ensure the atomicity of the transaction (recall from Section 3.1 that a failed site is handed over to the recovery process once the failure has been repaired). Finally, the durable states (committed, aborted) are the final states for a transaction. The durable states need not survive the crash but it must be possible to reach the same durable state of a transaction when a site recovers from the failure. This is fundamentally necessary to guarantee the durability of the transactions.

3.2.2 Forgetting the Transaction

When the master or a cohort of a transaction is created at a site, some data structures are allocated and control information is generated for the transaction. It is desirable that the transaction manager at the site hosting the master or the cohort (or both) *forgets* the transaction at the earliest possible time. A transaction manager forgets the transaction by releasing the data structures allocated to the transaction and by removing the control information about the transaction from memory.

Once a transaction manager forgets the transaction, it is no more in a position to answer queries about the transaction. Any further queries about the transaction will always get a `no information` response from the transaction manager. Therefore, a transaction manager forgets the transaction only when it is sure that no more queries about the transaction will be asked. Note that the `no information` response does not mean that the transaction manager has no information at all about the transaction. It simply means that no such information is available in the memory—there may be information in the log on the stable storage, but it will be too inconvenient for the transaction manager to find the information from the log in the absence of any control information about the transaction. Of course, if a crash occurs the log will be scanned while recovering from the crash and the information found there will be used to undo/redo the operations of the transactions.

The transaction manager forgets the transaction by writing an `END` log record (not forced). For a cohort, the `END` log record is merged with the `COMMIT` or the `ABORT` record itself, because the cohort can forget the transaction at the time of writing the `COMMIT` or the `ABORT` record. Thus there is no need for the cohort to write a separate `END` log record. The master, however, writes the `END` log record only when it has received the ACKs from all the cohorts that were sent the decision message. This is because if an ACK from a cohort has not been received, there is a chance that the cohort will query the master in near future about the outcome of the transaction.

3.2.3 Recovery from Failures

Let us now consider site and communication link failures. As mentioned earlier (Section 3.1), when a site recovers from a crash, it is handed over to the recovery manager. Recovery manager performs redo or undo operations for the transactions for which it finds `COMMIT` or `ABORT` records, respectively, in the log. If an `END` record is also found in the log for such a transaction, the recovery manager forgets the transaction. It was mentioned above that the `END` log record for a cohort is merged with the `COMMIT` or `ABORT` record. Therefore, if the recovery process finds a `PREPARE` record and a `COMMIT` or `ABORT` record for the transaction, it knows that the site was hosting a cohort of the transaction; performs the appropriate redo/undo operations; and forgets the transaction. For the master, the `END` record will be found only if all the ACKs from the cohorts were received and the log record had reached the stable storage before the crash occurred—the `END` record is not force-written to the disk. Therefore, if a `COMMIT` or `ABORT` record is found but the `PREPARE` record and the `END` record are not found in the log, this implies that: (a) the site was acting as a master for the transaction, (b) the decision about the transaction had been taken, and (c) some cohorts might not have yet received the decision. Therefore, the recovery manager periodically sends the decision message to the cohorts (the identities of the cohorts may be found in the decision record itself) until it receives ACKs from them. After receiving all ACKs, the recovery manager writes an `END` record in the log, and forgets the transaction.

For the transactions for which no log record indicating the commit processing (`PREPARE, COMMIT, ABORT`) is found, the recovery manager does not need to know whether the site was acting as the master or a cohort. It assumes that all such processes were in the executing state, and aborts them (by undoing their operations in case the undo records put by the WAL mechanism are found in the log). Note, however, that the cohorts may actually have been in the aborting state at the time of crash. But this does not affect the correctness because the recovery manager still moves the cohorts into the aborting state, and the undo operations anyway are *idempotent*.

The only case remaining is that the recovery manager finds a `PREPARE` record in the log, but finds no further COMMIT or ABORT record. This indicates that the site was hosting a cohort and the crash occurred after the cohort had sent a YES vote but before it could receive the decision from the master. On recovery from a failure, such cohorts cannot take a unilateral decision about the fate of the transaction. Therefore, the recovery manager queries the master about the outcome of the transaction (assuming that the master is up). Obviously, the master would not have received an ACK from such a cohort, therefore, the master would have retained the information about the transaction, and will reply accordingly. The recovery manager at the cohort's site will take the appropriate action based on the master's reply.

Failure-Blocking of the Cohorts

Now, consider a situation where master has dispatched the PREPARE messages but has failed before it could dispatch the decision to the cohorts. All those cohorts that had voted YES will be waiting in the prepared state for the master's decision. We call such cohorts as failure-blocked cohorts because they are waiting for the decision from the master that has failed. This is in contrast with the data-blocked cohorts where the cohorts are blocked waiting for the data items held by other cohorts. The failure-blocked cohorts will remain so until the master recovers from the failure. Moreover, these cohorts will continue to hold system resources such as locks on data items, making these unavailable to other transactions, which in turn may become data-blocked waiting for the data locks to be relinquished, i.e., "cascading blocking" results. It is easy to see that if the down-time for the master is long, it may result in major disruption of transaction processing activity in the distributed system.

No Information Case

We see till now that 2PC eliminates the possibility of the no information case which was discussed in Section 3.2.2. That is, the master site does not forget a transaction unless it is sure that no cohort will be asking it about the outcome of the transaction. Now, consider a case where the master fails after sending the PREPARE messages. A cohort on receiving the PREPARE message goes into the prepared state and sends a YES vote to the failed master. On recovery from failure, the recovery process at the master does not find any information pertaining to the commit processing of the transaction and simply aborts it and forgets it, deallocates all resources that were allocated to the transaction, and removes all information pertaining to the transaction from the memory. The cohort, on the other hand, queries the master about the outcome of the transaction. As the master has forgotten the transaction, it has no information about the transaction! Fortunately, such a case arises only in the situation discussed here. It is clear that in such a situation, the master had aborted the transaction. Therefore, a cohort on getting a `no information` response from the master can *correctly assume* that the transaction was aborted. The other solution to this situation could have been to force-write a log record at the master before sending the PREPARE message. But, the extra force-write may have an adverse impact on the performance of the commit protocol.

3.2.4 Summary of Two-Phase Commit Protocol

2PC guarantees the uniform commitment of the distributed transaction by means of messages and logging. In the absence of failures, the protocol is straightforward in that a commit decision is reached if all cohorts are ready to commit, otherwise an abort decision is taken. To handle the failures, 2PC ensures that the sufficient information is force-written on the stable storage to reach a consistent global decision about the transaction. A cohort may query the master to find out the outcome of the transaction. The 2PC ensures that the master will have the information available to answer such queries because otherwise it could lead to non-uniform commitment of the transaction (this situation, however, does occur in one special case—called the `no information` case—where the master does not have any information about the transaction to answer a cohort's query, as discussed earlier).

The message and the logging overheads in 2PC are listed in Tables 3.1 and 3.2, respectively. For each committing transaction the master sends two messages to each cohort and writes two log records one of which is forced. Each cohort sends two messages to the

Table 3.1: Message Overheads for 2PC

	Master to Cohort	Cohort to Master
For Committing Transactions	2	2
Cohort Votes No	1	1
Cohort Votes YES but Master Decides Abort	2	2

Table 3.2: Logging Overheads for 2PC

	Master		Cohort	
	Total Log Records	Force-writes	Total Log Records	Force-writes
For Committing Transactions	2	1	2	2
Cohort Votes No	2	1	1	0
Cohort Votes YES but Master Decides Abort	2	1	2	2

master and writes two log records both of which are forced. For an aborting transaction (a transaction that aborts in the commit processing phase, for example, if master receives a NO vote), the master sends two messages to each cohort that votes YES and one message to each cohort that votes NO, and writes two log records one of which is forced. Each cohort that sends a NO vote sends only one message (the NO vote), and writes only one log record which is not forced. On the other hand, each cohort that votes YES sends two messages and writes two log records both of which are forced.

In summary, the protocol guarantees atomicity and durability of the transactions despite the occurrence of site or communication link failures. The protocol, however, can lead to `failure-blocking` of the cohorts in case of failures.

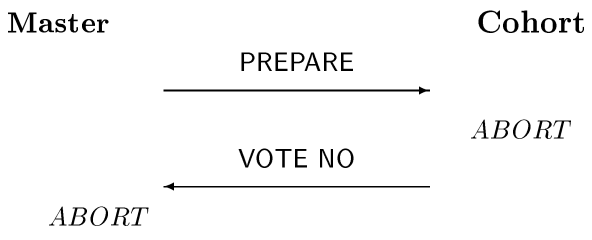


Figure 3.7: Presumed Abort when Cohort Votes NO

3.3 Presumed Abort Protocol

As described in the previous section, the 2PC protocol requires transmission of several messages and force-writing of several log records. A variant of the 2PC protocol, called **presumed abort (PA)** [MLO86], attempts to reduce these overheads by requiring all cohorts to follow a "in the `no information` case, abort" rule. We saw in the previous section that when the `no information` case occurs, it can be *correctly* assumed that the transaction was aborted. PA attempts to save on a few messages and log writes in such a way that there will be many more situations when the `no information` case will occur. But the protocol guarantees that in all such situations, the transaction can be correctly assumed to have been aborted. In other words, 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 the cohorts to either send acknowledgments for the ABORT messages from the master or to force-write the `ABORT` log record. It is also not necessary for the master to force-write the `ABORT` log record or to write an `END` log record after the `ABORT`. Figure 3.7 shows the execution of PA for a cohort that votes NO, and Figure 3.8 shows the execution of PA when the cohort votes YES but the master decides to abort the transaction (some other cohort could have voted NO).

When a site recovers from a failure, the only case it will query the master about the outcome of the transaction is that it finds a `PREPARE` record in the log. If the master had decided the commit, then it cannot forget the transaction without receiving the ACK from the cohort. Therefore, in such a case, master will respond with the COMMIT message. On

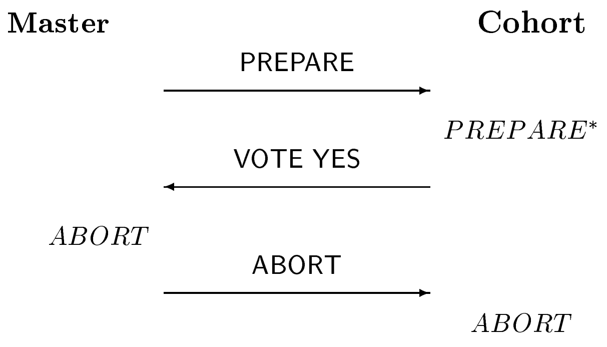


Figure 3.8: Presumed Abort when Cohort Votes YES but Master Decides Abort

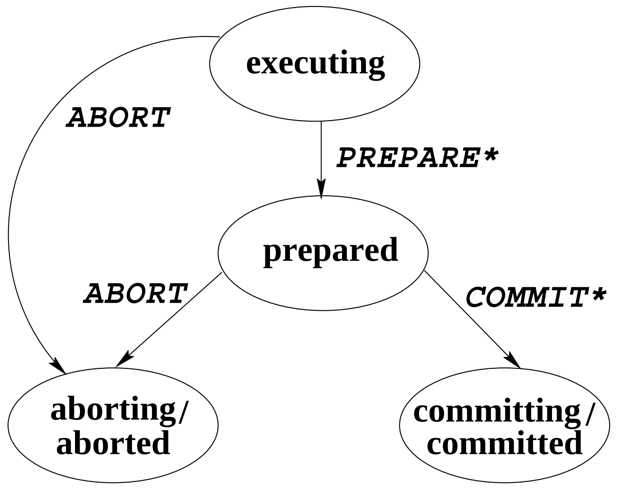


Figure 3.9: State Transitions in PA for the Cohort

the other hand, if the decision was abort, the master would have forgotten the transaction without waiting for the ACK from the cohort. Therefore, on getting a `no information` response, the cohort can correctly assume that the transaction was aborted. In all other cases, when the recovery process does not find any commit processing record in the log, it will simply (and correctly) abort the master/cohort as described in Section 3.2.3. We will not discuss about the committed transactions as they are handled in exactly the same way as 2PC. The state transition diagrams for PA are shown in Figures 3.9 and 3.10.

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

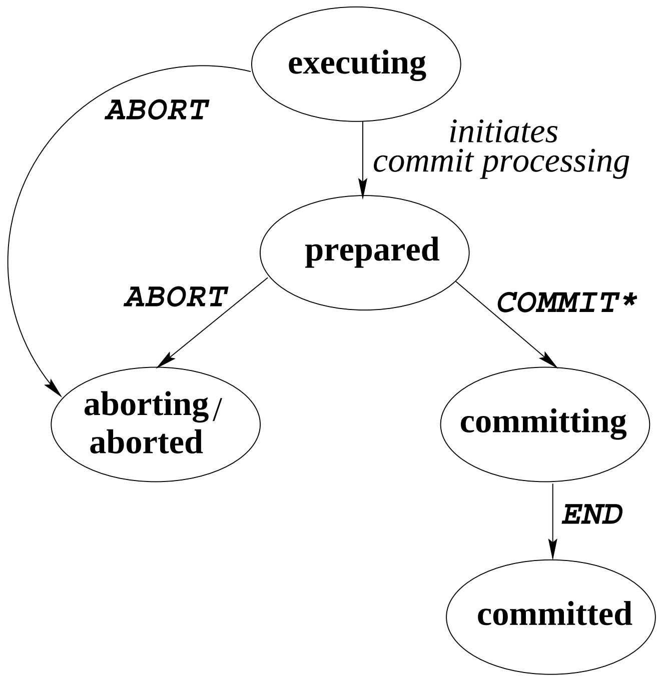


Figure 3.10: State Transitions in PA for the Master

3.4 Presumed Commit Protocol

In general, the number of committed transactions is much more than the number of aborted transactions. Based on this observation, a variation of presumed abort is proposed that attempts to reduce the messages and logging overheads for *committing* transactions rather than *aborting* transactions. This variation, called **presumed commit (PC)** [MLO86], requires all cohorts to follow a "in the `no information` case, commit" rule. In this scheme, cohorts do not send acknowledgments for the COMMIT global decision, and do not force-write the `COMMIT` log record. In addition, the master does not write an `END` log record (master does force-write the `COMMIT` log record). Instead, the cohorts send acknowledgements for the ABORT decision, and `ABORT` log records are force-written. The result is that when the `no information` case arises, the cohort can correctly assume that the transaction was committed.

However, consider a situation where the master fails after sending the PREPARE messages. When the master recovers from the failure, the recovery process does not find any commit processing record for the transaction in its log. Therefore, the recovery process

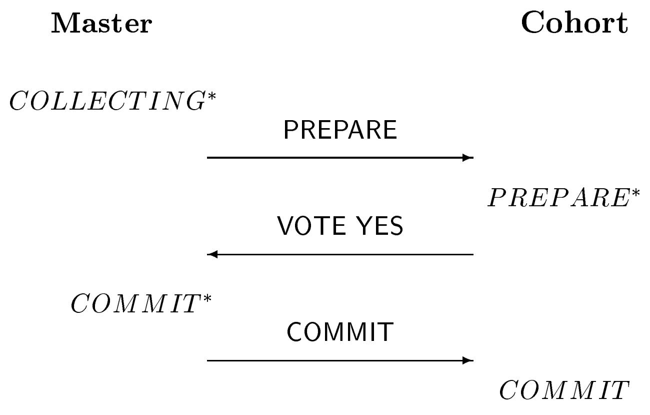


Figure 3.11: Presumed Commit for Committing Transactions

simply aborts, and *forgets* the transaction. Now, a cohort that has entered the prepared on receiving the PREPARE message queries the master about the outcome of the transaction. As the master's site has already forgotten the transaction, the response will be a `no information` case. By following the "presumed commit" rule, the cohort will decide the commit. However, the transaction had been aborted by the recovery process at the master's site. Thus, it leads to inconsistent decision about the transaction. In order to solve this problem, the master is required to force-write a `COLLECTING` log record before initiating the commit protocol. This log record contains the names of all the cohorts involved in executing that transaction. Figure 3.11 shows the execution of PC for committing transactions. The state transition diagrams for PC are shown in Figures 3.12 and 3.13.

In summary, presumed commit reduces the message and logging overheads for the committing transactions, but increases the overheads of aborting transactions. Except for the extra `COLLECTING` record, the abort processing in PC is same as in 2PC.

Significance of PA and PC

The PA and PC optimizations of 2PC have been implemented in a number of database products. These protocols were developed and implemented in the $R^\star$ project at IBM Almaden Research Center. PA has been incorporated in many research prototypes and

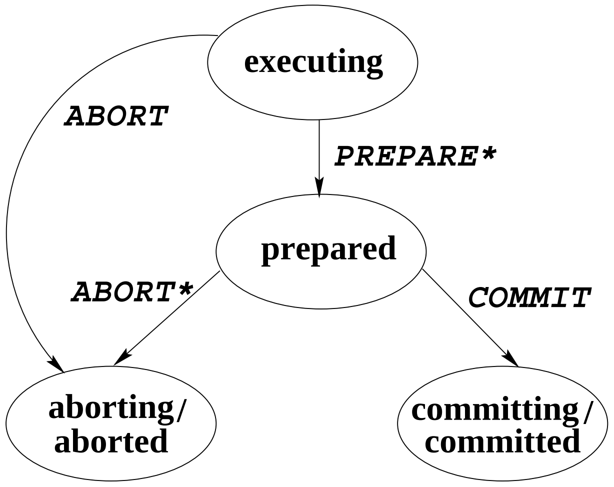


Figure 3.12: State Transitions in PC for the Cohort

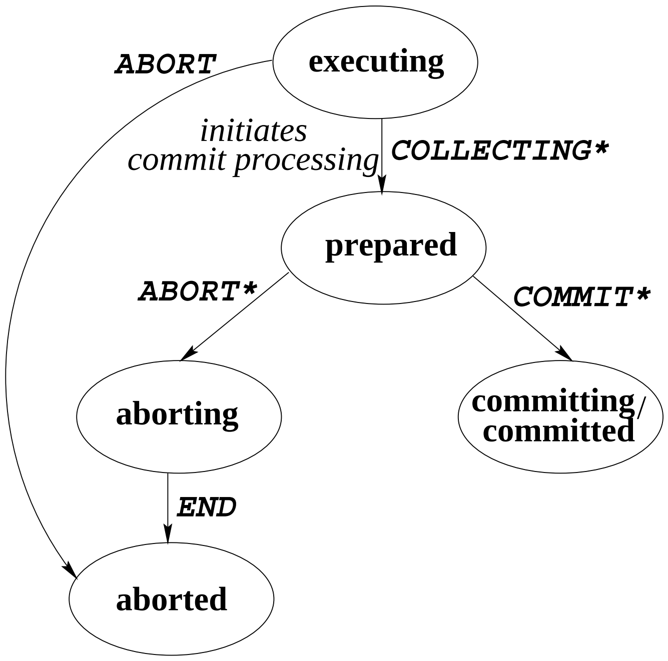


Figure 3.13: State Transitions in PC for the Master

commercial products, for example, Tandem's TMF, DEC's VAX/VMS, Transarc's Encina, CMU's Camelot, Unix System Laboratories' TUXEDO, and IBM Almaden Research Center's QuickSilver in addition to $R^\star$ [MD94]. A variation of PA, called *generalized PA (GPA)* has been implemented in SNA LU 6.2—IBM's network architecture for *peer-to-peer* inter-program communication [MBCS92, IBM93]. GPA has also been implemented in DB2 [Moh93]. PA is, in fact, now part of the ISO-OSI and X/OPEN distributed transaction processing standards [MD94, SBCM95].

3.5 Other Variations of Two-Phase Commit Protocol

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, as mentioned before, that there are a number of other variations of the 2PC that have been proposed in literature. The *Nested 2PC* [Gra78] (also called as *Linear 2PC*) reduces the number of the message-exchange in 2PC by linearly ordering all cohorts. This, however, is done at the cost of concurrency in the commit processing resulting in larger commit processing times for the transactions. In *distributed 2PC* [ÖV91], each cohort broadcasts its vote to all other cohorts, thus trading the commit processing time against the number of messages. In *Unsolicited Vote (UV)* [Sto79] protocol of distributed INGRES, the cohorts enter the prepared state at the time of sending the WORKDONE message itself. Thus, the WORKDONE message acts as a 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 send more work to the cohort forcing it back into the executing state from the prepared state. Thus, releasing the read locks would violate the two-phase locking thereby compromising the serializability semantics of the transaction. Moreover, a cohort in *UV* has to force-write a log record each time it responds with the WORKDONE message. In [SC90, SC93], a combination of the *UV* protocol and the *PC* protocol, called *Early Prepare (EP)* protocol, is proposed. They also propose a *Coordinator Log (CL)* protocol for the client/server type of environments, where the cohorts piggyback the log records on the WORKDONE messages to the master, instead of force-writing them locally. [SJR91] discusses the *Group Commit* protocol, where the log records are forced for a group of transactions instead of each individual transaction. The tradeoffs between the size of the transaction group and the response time of the transactions are discussed in this paper.

As mentioned earlier in Section 3.4, the *PC* protocol introduces an additional force-write and thus incurs more overheads as compared to the *PA* protocol. Mechanisms that attempt to reduce the cost of *PC* are proposed in [LL93] and [AHCL97] by minimizing its logging activity. Very recently, the *Implicit Yes Vote (IYV)* protocol [AHC95, AHC96] and the *two-phase abort (2PA)* protocol [BC95, BC96] have been proposed for future distributed database systems that are expected to be connected by extremely high speed (gigabit-per-second) networks. These protocols exploit the extremely high data transfer rates of the gigabit network and attempts to minimize the number of messages exchanged in the commit processing by having more information in a single message. In a gigabit network, usually the number of messages exchanged and not the size of the message, is the primary concern.

In a general case, a cohort in 2PC can also spawn off further sub-transactions at other sites, to which it will function as an intermediate master. This leads to a tree structure of the transaction. In such cases, there will be a root process which will act as the master, the leaf processes which will act as the cohorts, and the intermediate processes which will act as the master for their children processes, and as the cohorts for their parent processes. Generalizations of many 2PC based protocols for such tree of processes environments are discussed in [ML83, MSF83, MLO86, SBCM93, SBCM95].

Another variation of 2PC is the *Presumed Nothing (PN)* protocol which is a part of IBM's SNA LU 6.2 architecture [IBM93], and has been implemented in IBM's commercial products (for example, CICS/MVS and VM/ESA [SBCM95]). *PN* is a more general protocol than 2PC, and was designed for *peer-to-peer* environments. In a peer-to-peer environment, any of the participating sites can initiate the commit protocol to conclude the transaction (if more than one site initiates the protocol, the transaction is aborted). The correct functioning of the protocol in such an environment requires a log record to be forced before initiating the protocol (similar to the `COLLECTING` log record required in *PC*).

3.6 Three-Phase Commit Protocol

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 (`failure-blocking` was also discussed in Section 3.2.3). 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, which in turn become blocked waiting for the resources to be relinquished, i.e., "cascading blocking" results. It is easy to see that, if the blocked period is long, it may result in major disruption of transaction processing activity.

To address the `failure-blocking` problem, a **three-phase commit (3PC)** protocol was proposed in [Ske81]. This protocol achieves a non-blocking[4] 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. Note, however, that the price of gaining non-blocking functionality is an increase in the communication overheads since there is an extra round of message exchange between the master and the cohorts. In addition, both the master and the cohorts have to force-write additional log records in the precommit phase.

[4]As mentioned earlier, it is impossible to design commit protocols that are completely immune, in the non-blocking sense, to both site and link failures. Even the so called non-blocking protocols are prone to blocking under certain (perhaps highly unlikely) failure conditions.

3.7 Optimistic Commit Processing

In virtually all of the protocols described in the previous sections, a cohort that reaches the prepared state can release all of its read locks.[5] However, it 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 `data-blocking` is completely *orthogonal* to the `failure-blocking` (because of failures) as was discussed in Section 3.2.3. That is, in all commit protocols, including 3PC, transactions may become `data-blocked` waiting for (prepared) data to be unlocked. Moreover, such `data-blocking` occurs during normal processing whereas `failure-blocking` occurs only 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* uncommitted data to concurrently executing transactions (Figure 3.14). In this context, two situations may arise:

Lender Receives Decision First: Here, the lending cohort (i.e., the prepared cohort) receives its global decision before the borrowing cohort has completed its local execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. If the global decision is to abort, then the lender is aborted in the normal fashion; in addition, the borrower is also aborted since it has utilized inconsistent data.

Borrower Completes Execution First: Here, the borrowing cohort completes its ex-

[5]Except in *UV* and *UV-based* protocols, wherein releasing the read locks violates the two-phase locking protocol as mentioned in Section 3.5.

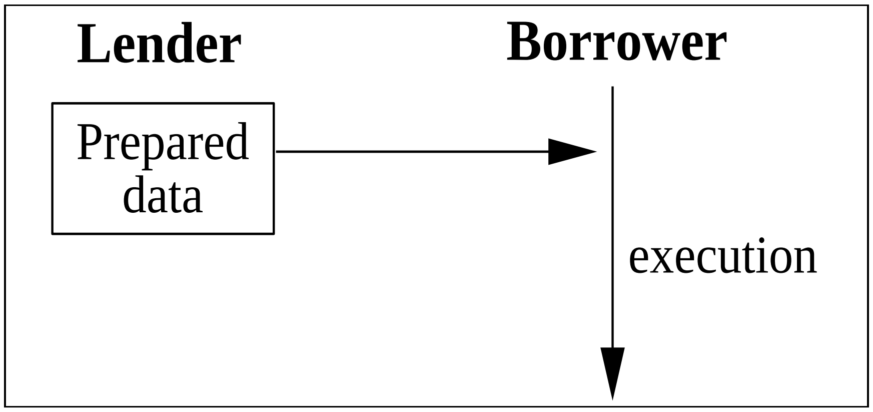


Figure 3.14: Optimistic Commit Protocol

ecution 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 processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until the lender receives its global decision. If the lender commits, then the borrower is "taken off the shelf" and allowed to send its WORKDONE message. However, if the lender aborts, then the borrower is also aborted immediately since it has utilized inconsistent data.

In summary, the protocol allows transactions to access uncommitted data held by prepared transactions in the "optimistic" belief that this data will eventually be committed. We will hereafter refer to this protocol as **OPT**. Note that the basic structure of OPT is very similar to that of 2PC. Further, as the borrowing is a local (intra-site) phenomenon, the additional overheads of OPT are negligible.

3.7.1 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

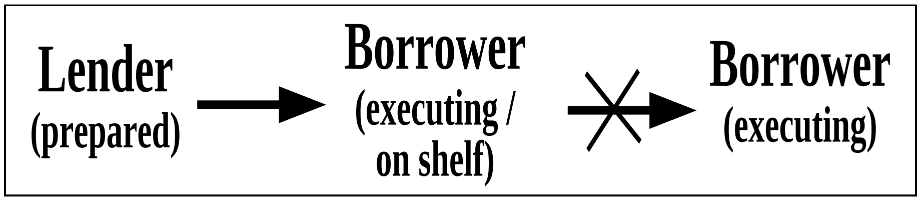


Figure 3.15: "Only a lender or a borrower be"

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 their data conflicts that occurred prior to the initiation of the commit protocol. In fact, if we assume that a lock based concurrency control mechanism such as, for example, 2PL [EGLT76], 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 this type of aborts.

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. This is because the borrower is not in the prepared state—the only situation in which uncommitted data can be lent. In other words, a borrower cannot simultaneously be a lender (Figure 3.15). In short, 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).

3.7.2 Integrating Prior 2PC Optimization with OPT

Apart from the optimistic data access described above, the following features can also be included in the OPT protocol:

Presumed Abort/Commit : The PA and PC optimizations discussed earlier for 2PC (Section 3.3 and 3.4) can also be used in conjunction with OPT to reduce the protocol overheads. We consider both options in our experiments.

Non-blocking 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.

Other Optimizations : Apart from the PA and PC optimizations and the optimizations discussed in Section 3.5, we have also considered the following optimizations: *Read-Only* (one-phase commit for the read-only transactions), *Long Locks* (cohorts piggyback their commit acknowledgements onto subsequent messages). *Unsolicited Vote*, *Coordinator Log*, *Linear 2PC*, and *Group Commit* have been discussed in Section 3.5.

OPT is especially attractive to integrate with protocols such as *3PC*, *Group Commit* and *Linear 2PC*, since they *extend* the period during which data is held in the prepared state. However, when combined with protocols such as *Unsolicited Vote* and *IYV*, OPT can lead to cascading aborts, long "on-the-shelf"-times for borrowers, deadlocks involving the lender and the borrower, etc. This is because *Unsolicited Vote* and *IYV* protocols do not guarantee that a cohort which has unilaterally entered the prepared state will not be forced back later into an executing state. A lender that has been forced back to the executing state can be aborted (for example, due to becoming involved in a deadlock) causing the borrower also to abort. Moreover, a lender that re-enters the executing state may later become a borrower, thus giving rise to a "borrow-chain" of *long* 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. Also, a lender is usually expected to complete its commit processing fast because it is only waiting for the decision from its master. However, if a lender is forced back to the execution state, the borrowers will have to wait on-the-shelf until the lender first completes its newly assigned work and then completes its commit processing. This will give rise to long on-the-shelf delays for the borrowers. And then there is a possibility of a deadlock involving both the lender and the borrower—the borrower will be waiting for the lender to finish while the lender might need to access a data item currently held by the borrower. Dealing with such deadlocks might prove a difficult task.

Barring these exceptions of *Unsolicited Vote* and *IYV*, virtually all of the above optimizations can be integrated with an OPT implementation to produce *enhanced* performance. In particular, OPT is attractive with protocols such as *Group Commit*, *3PC*, and *linear 2PC* since they extend the period during which data is held in the prepared state.

3.7.3 System Integration

We now comment on the implementation issues related to the OPT protocol:

1. The lock manager at each site must be modified to permit borrowing of data held by prepared cohorts.

2. The lock manager must keep track of the cohorts that have borrowed prepared data so that, if the lender aborts, the borrowers can also be aborted.

Lender 1	Lender 2	...
Borrower 1	Borrower 2	...

Figure 3.16: Lender-Borrower List

One mechanism to accomplish this could be by maintaining a list where each element consists of a lender-borrower pair (Figure 3.16). Whenever a data item is lent to some executing cohort, the transaction ID pair of the lender and the borrower is entered into this list. (The lock held on the data item is updated as requested by the borrower). If the lender commits, the lender-borrower entry can be safely removed from the list. On the other hand, if the lender aborts the lender-borrower entry is removed from the list and in addition the borrower cohort is also aborted (which in turn will release the lock on the borrowed data item).

3. For a borrower cohort that finishes execution before its lenders have received their global decision, the local transaction manager must not send a *WORKDONE* message until the fate of its lenders is determined.

The above modifications do not appear difficult to incorporate in current database system software. Moreover, as shown later in our experiments, the performance benefits that can be derived from these changes suggest that it is worthwhile to make the effort of implementing them.

3.8 Performance Studies of Commit Protocols

As mentioned earlier, most of the performance studies of commit protocols have been limited to comparing protocols based on the *number* of messages and the *number* of force-writes that they incur. Very little work has been done to thoroughly evaluate the quantitative performance of the commit protocols with regard to *overall* transaction processing metrics such as mean response time or peak throughput. The only work that we are aware of is that of [LAA94a] where such a performance study is attempted. However, the scope and methodology of their study is considerably different from ours as explained below:

1. Using a simulation model, the authors have made a performance study of the *2PC*, *PA*, *PC*, and *Early Prepare* protocols, in the context of a client/server database type of environment, where the master of a transaction resides at the client site and the cohorts of the transaction are executed at the server sites. We model a fully distributed database environment, where each site can host any or both of the master and the cohorts of a transaction.

2. Non-blocking commit protocols are not considered in their study. In our study, we have examined both blocking and non-blocking commit protocols.

3. The descriptions of the protocols do not strictly adhere to their original published versions. For example, in the description of the *PC* protocol, it is mentioned that in *2PC*, a cohort does not update the modified data until it receives COMMIT message from the master while in *PC*, the cohort updates the data when it votes YES. There is no relationship between data updates at cohort and the *PC* protocol—*PC* is concerned about how to handle `no information` case (which would only occur after a failure). In fact, no mention of data updates is made in [MLO86] (the *PA* and *PC* protocols were proposed in this paper). Moreover, in [LAA94a] no mention is made about the `COLLECTING` log record which needs to be forced in the *PC* protocol. Also, it is mentioned that the overheads of *PC* and *2PC* are same for aborting transactions—actually, *PC* requires the additional `COLLECTING` log record to be force-written making the overheads of *PC* to be more than *2PC*. Similar differences exist in the descriptions of the *PA* and *Early Prepare* protocols.

4. In the detailed version of the paper [LAA94b], an `ABORT` log record is forced for *2PC* even if the transaction aborts before initiating the commit processing. This results in the performance of *PA* being superior to *2PC* even in the case when there are no aborts except those due to serializability reasons (which occur before the commit processing is initiated). We have shown in our work, that in such situations, *PA* behaves identically to *2PC*.

5. They model the effects of failures but as discussed in Introduction, we feel, from a performance perspective, that the effect of the commit protocols on normal processing and resilience of the protocols to failures are of primary importance since they directly affect ongoing transaction processing. Moreover, with current technology failures are rare. In addition, the durations of failures are usually large. Therefore, we have exclusively focussed on the effect of the commit protocols on normal processing.

6. An *open* system is modeled in their study. An open system is susceptible to *instability*—the queues will start building once the system gets overloaded. It is not clear how stability of the system is ensured. In addition, they have used the transaction throughput as a metric (apart from response time). In an open system, the throughput will always be equal to the arrival rate unless the system is *unstable*.

7. In addition, we present and evaluate a new high-performance protocol (OPT) that, in contrast to earlier commit protocols, allows transactions to "optimistically" borrow (dirty) uncommitted data. Although 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 with most of the other optimizations proposed earlier.

Apart from the work mentioned above, we are not aware of any other quantitative performance study of the commit protocols. Thus, we believe that a systematic performance study of the various commit protocols would be of benefit to distributed database system designers—we address this issue in this work. In addition, we also study the performance implications of supporting the non-blocking functionality. We also present and evaluate a new high-performance protocol (OPT) that is easy to implement and incorporate in current systems, and can be combined with a variety of other commit protocols including the *PA*, *PC*, and three-phase commit protocols.

C H A P T E R

4

OLTP Simulation Model

To evaluate the performance of the various commit protocols described in the previous chapter, we developed a detailed simulator based on a closed queueing model of a distributed database system. Our simulation model is very similar to the distributed database model presented in [CL88], and consists of a database that is distributed, in a non-replicated manner, over N sites connected by a network. Figure 4.1 shows the general structure of the model. Each site in the model 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 communications network.

The following sections describe the database model, the workload generation process, the physical resource configuration, and the execution pattern of a typical transaction. A summary of the parameters used in the model are given in Table 4.1.

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

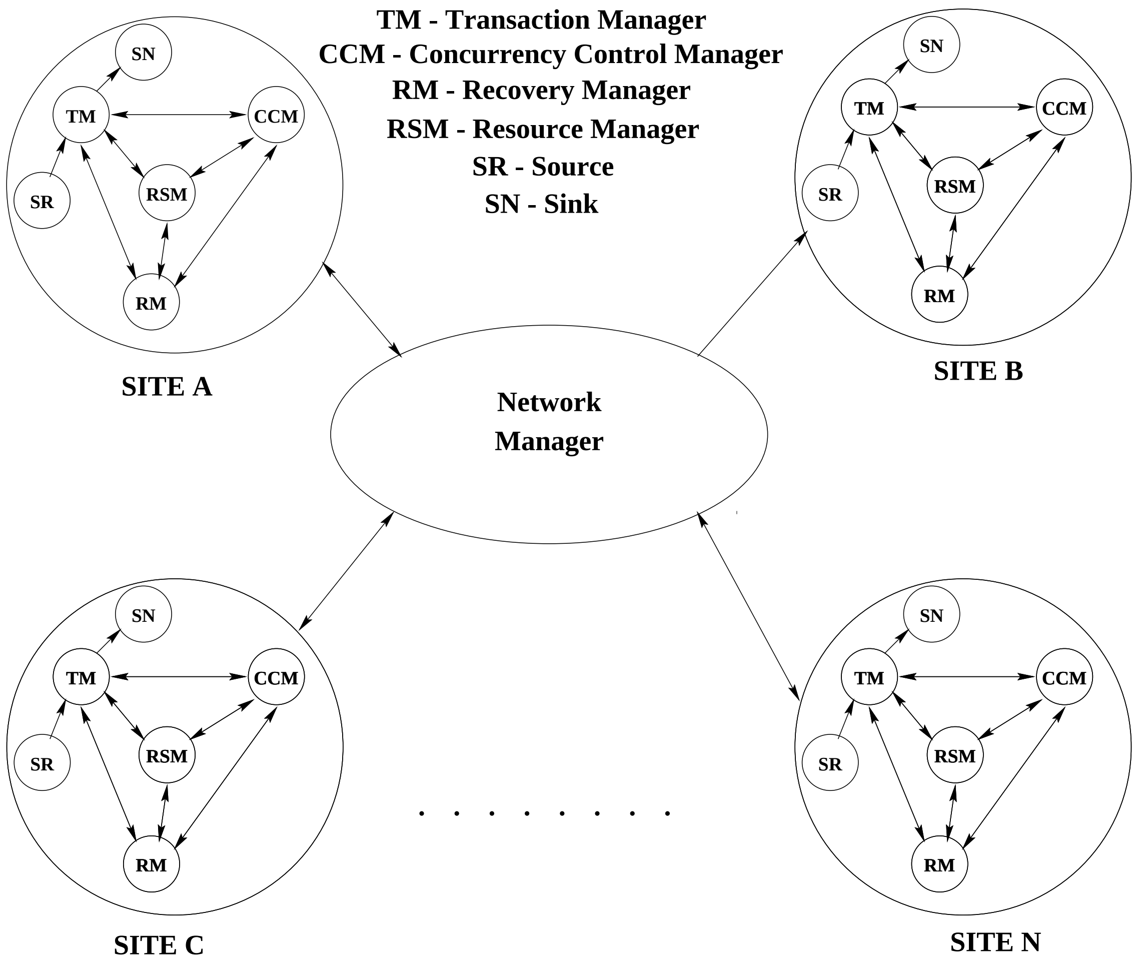


Figure 4.1: Distributed DBMS Model Structure

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 3.1. 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 randomly from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$. A transaction that is aborted is restarted after a delay and makes the same data accesses as its original incarnation, i.e., restarts are not *fake* [ACL87]. The length of the delay is equal to the average transaction response time—this is the same heuristic as that used in most studies of centralized and distributed concurrency control [ACL87, CL88, CL89, CL91]. Incorporating the delay prevents two possibilities: (a) formation of the same deadlock immediately (in the absence of the delay, the restarted transaction immediately acquires the lock on the data item it was holding before being restarted), and (b) formation of *livelocks*, i.e., a situation in which the transactions are repeatedly involved in deadlocks, virtually making no progress. Even though the work is being done, all of it is wasted due to the transaction restarts, and the system is not progressing at all causing a livelock situation. Finally, after a transaction completes, a new one is submitted immediately at its originating site, that is, there is no think time.

4.3 Execution Model

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

Table 4.1: Simulation Model Parameters

$NumSites$	Number of sites in the database
$DBSize$	Number of pages 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
$MsgCPU$	Message send / receive time
$BufHit$	Probability of finding a requested page in the buffer

up at the originating site, forks off a local cohort and sends messages to initiate each of its cohorts at the remote participating sites. 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. For example, a series of steps in a relational query can be modeled as a sequential transaction while a parallel query execution that is seen in systems such as Gamma [DGS+90] can be modeled as a parallel transaction. We consider both types of transactions in our study—the parameter $TransType$ specifies whether the transaction execution is sequential or parallel.

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. We assume sufficient buffer space to allow the retention of data updates until commit time. When a transaction intends to update a page, the page is read prior to making the update, i.e., there are no *blind* *writes* [BHG87]. Update-locks are acquired when a data page intended for modification is first read, i.e., lock upgradation is not modeled. The commit protocol is initiated when the transaction has completed its data processing.

4.4 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 an FCFS order except that message processing is given higher priority than data processing at the CPUs. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. A data page request can be satisfied by the buffer manager, or the page may have to be accessed from the disk (Section 4.6).

4.5 Network

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 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. The CPU overheads for message transfers are captured by the $MsgCPU$ parameter.

4.6 Buffer Management

When a transaction makes a request for accessing a data page, the data page may be found in the bufferpool, or it may have to be accessed from the disk. In our model, a requested data page is found in the bufferpool with a probability represented by the $BufHit$

parameter. Thus, the data page has to be accessed from the disk with a probability of $1 - BufHit$. We do not model the explicit buffer management policies—modeling them would certainly result in different absolute performance numbers, but we do not expect them to significantly alter the *relative* performance behavior of the commit protocols.

4.7 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, a 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.

A point to note here is that, as mentioned in Section 3.7.1, with this CC mechanism, there is no possibility of serializability-induced aborts occurring in the *commit processing stage*.

Deadlocks

Since we are using 2PL 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 choice of a victim in resolving a deadlock is made based on transaction timestamps—the youngest transaction in the cycle is restarted to resolve the deadlock. When a transaction is restarted, it retains the timestamp that it was assigned when it first entered the system. As mentioned earlier, the restarted transaction makes the same data accesses as its original incarnation.

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].

4.8 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 execution log records (i.e., WAL) are not modeled since we do not expect them to affect the relative performance behavior of the commit protocols discussed in our study.

CHAPTER

5

OLTP Experiments and Results

Using the distributed OLTP system model described in the previous chapter, we conducted an extensive set of simulation experiments comparing the performance of the various commit protocols discussed in Chapter 3. The simulator was written in C++SIM [SIM94], an object-oriented simulation testbed, and the experiments were conducted on SPARC-20/Solaris 2.4 and ULTRASPARC/Solaris 2.5 workstations. In this chapter, we present the results of a variety of experiments and discuss the performance profiles of the commit protocols.

5.1 Performance Metric

The primary performance metric is *transaction throughput*, that is, the rate at which the system completes transactions.[1] We also emphasize the *peak* throughput that is achievable by each protocol since this represents the maximum attainable performance and by using a suitable admission control policy (for example, Half-and-Half [CKL90]), the throughput in a heavily loaded system can be maintained at this level. All the throughput graphs in this chapter 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 by the system. Only statistically significant

[1]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.

differences are discussed here.

The simulator was instrumented to generate a host of other statistical information, including resource utilizations, transaction restarts, messages, force-writes, cohort blocking factor, cohort block-execution ratio, page conflict probability, transaction commit-execution ratio, transaction borrow ratio (for OPT), etc. These secondary measures help to explain the throughput behavior of the commit protocols under various workloads and system conditions.

5.2 Comparative 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 for two additional scenarios. These scenarios are:

Centralized System :

Here, we simulate the performance that would be achieved in an equivalent *centralized* database system—a multiprocessor system having the same number of CPUs and disks, and the same amount of data, as that of the distributed system. Note that all CPUs (and the disks) in the centralized system form a single pool, resulting in better utilization due to load-balancing. The data items in the centralized system are partitioned in the same fashion as in the distributed system in order to have the similar conflict probability patterns of the transactions in both the systems. In a centralized system, messages are obviously not required and commit processing only requires force-writing a *single* `COMMIT` log record. Modeling this scenario helps to isolate the overall effect of distribution on throughput performance.

Distributed Processing, Centralized Commit :

Here, data processing is executed in the normal distributed fashion, that is, involving messages. The *commit* processing, however, is like that of a centralized system, requiring only the force-writing of the `COMMIT` log record at the master. While this

Table 5.1: Baseline Parameter Values

NumSites	8	*NumCPUs*	2
DBSize	8000 pages	*NumDataDisks*	3
TransType	Sequential	*NumLogDisks*	1
DistDegree	3	*PageCPU*	5 ms
CohortSize	6 pages	*PageDisk*	20 ms
UpdateProb	1.0	*MsgCPU*	5 ms
MPL	1 - 10 per site	*BufHit*	0.1

system is clearly artificial, modeling it helps to isolate the effect of distributed commit processing on throughput performance (as opposed to the centralized scenario which eliminates the entire effect of distributed processing).

In the following experiments, we will refer to the performance achievable under the above two scenarios as **CENT** and **DPCC**, respectively.

5.3 Experiment 1: Baseline Experiment

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 the baseline experiment are listed in Table 5.1. These values 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, without having to simulate very high transaction multiprogramming levels.

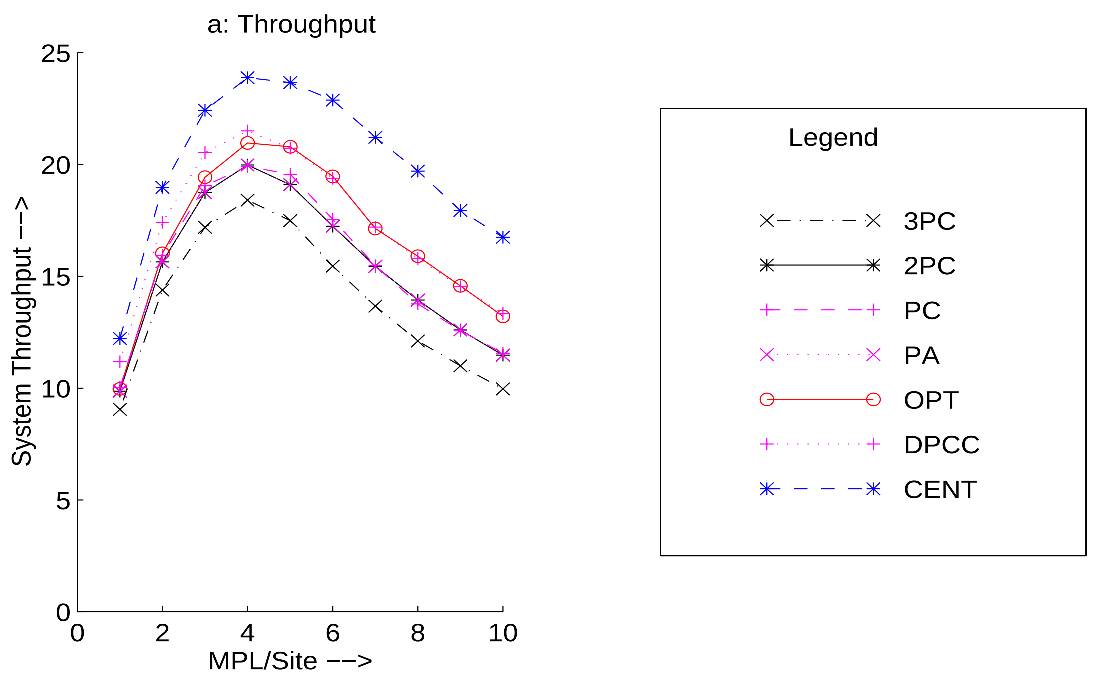
In this experiment, each transaction executes in a *sequential* fashion at three sites, accessing and updating an average of six pages at each site. Each site has two CPUs, three data disks and one log disk. The CPU and disk processing times are such that the system operates in an I/O-bound region. However, 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 5 where a higher degree of transaction distribution, and consequently more network activity, is modeled.

For this experiment, Figure 5.1a 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.

Comparing the relative performance of the protocols in Figure 5.1a, we first observe that the centralized system (CENT) performs the best, as expected, and that the performance of distributed processing/centralized commit (DPCC) is worse than that of CENT. This degradation in performance of DPCC is due to the extra overheads of DPCC over CENT. The performance of the classical commit protocols (2PC, PA, PC, 3PC) is significantly worse than that of CENT throughout the loading range. This clearly shows the impact of distribution on the throughput performance of the system. Note that distributed *commit* processing (difference between the performance of DPCC and 2PC) and distributed *data* processing (difference between the performance of CENT and DPCC) both contribute substantially to this degradation in performance. In this experiment, the performance impact of distributed data processing is more than that of distributed commit processing. We will see in other experiments, however, that usually the performance impact of distributed commit processing is more than that of distributed data processing.

Moving on to the relative performance of 2PC and 3PC, we observe that there is a significant difference in their performance. The difference arises from the additional message and logging overheads involved in 3PC, which are shown in Table 5.2 (for committing transactions).

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 force-writes and acknowledgements at each cohort, it incurs an additional `COLLECTING` force-write at the master. Second, the force-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



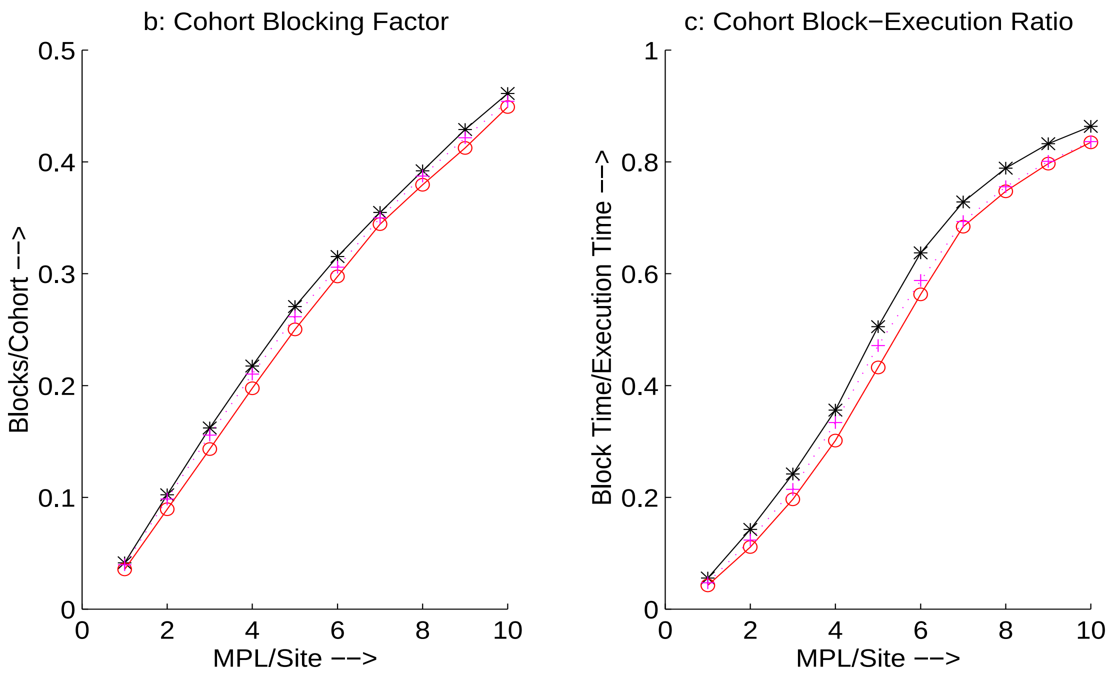


Figure 5.1: (a–c) Baseline Experiment (Sequential, RC+DC)

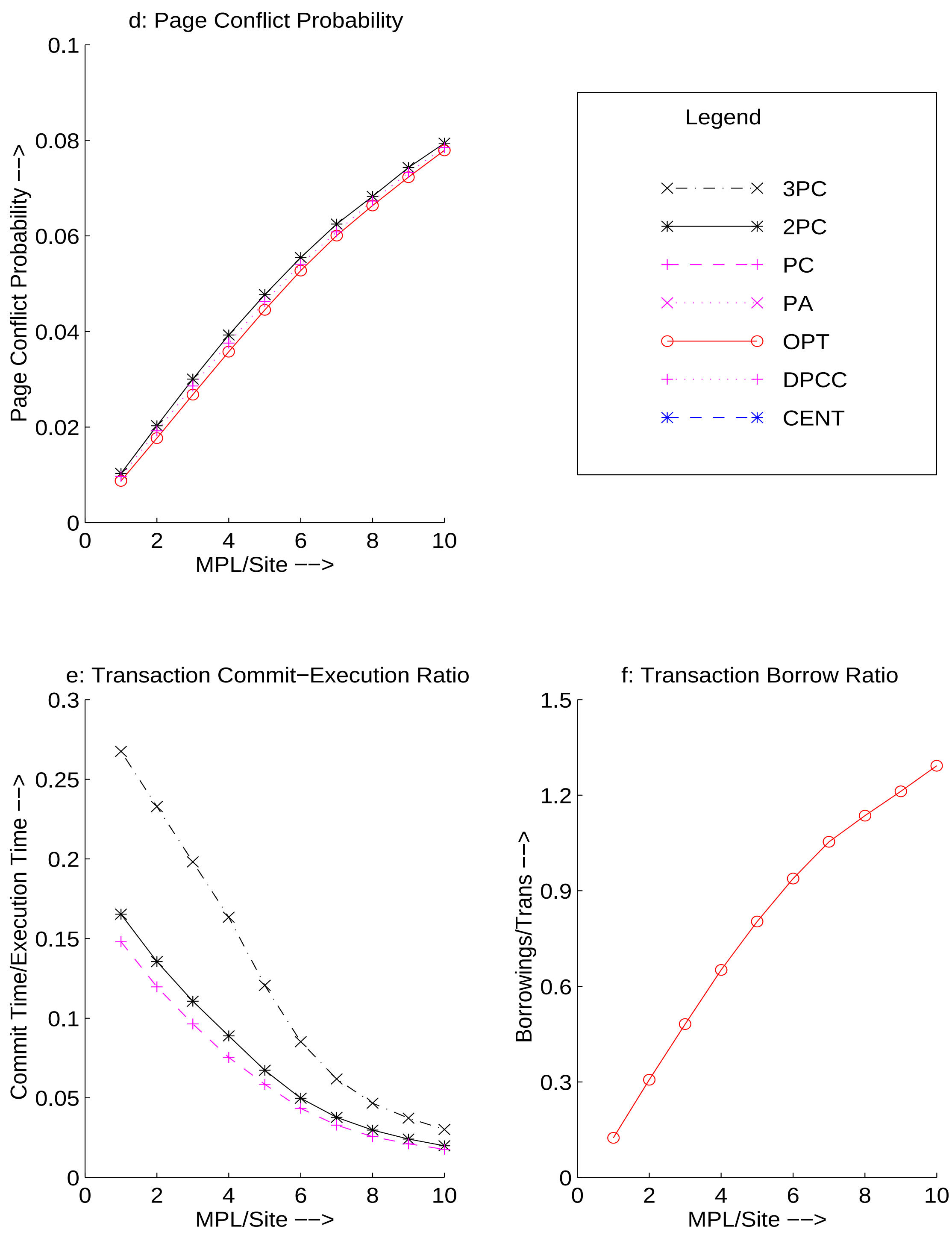


Figure 5.1: (d–f) Baseline Experiment (Sequential, RC+DC)

Table 5.2: Protocol Overheads for Committing Transaction (DistDegree = 3)

Protocol	Execution Messages	Commit Force-Writes	Commit 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

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 4.7, 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 8 where we model surprise transaction aborts 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 MPLs. At low MPLs, when there is less data contention, and consequently little opportunity for borrowing, OPT is virtually identical to 2PC and therefore performs at the same level. At higher MPLs, however, the performance of OPT is significantly 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 the data contention thus reducing the number of blocked transactions in the system. This is seen in Figure 5.1b, which shows the cohort "blocking factor", that is, the average number of times a cohort blocks during its execution. Similarly, Figure 5.1c, which presents the cohort "block-execution ratio", that is, the ratio of cohort's data blocking time (cumulative time during which a cohort is `data-blocked`) to its execution time (time elapsed at the cohort between receiving the STARTWORK message and sending the WORKDONE message), shows that with OPT, transactions spend less time waiting for the data locks as compared to 2PC.

Figure 5.1d presents the "page conflict probability", that is, the probability with which a page request will be blocked, which again is less for OPT as compared to 2PC.

Figure 5.1e presents the transaction "commit-execution ratio", that is, the ratio of the *commit phase length* of the transaction to its *execution phase length*. Execution phase length of a transaction is the time elapsed between the master sending the first STARTWORK message (for the first cohort) and the subsequent PREPARE message after all the WORKDONE messages have been received from all the cohorts. Commit phase length of a transaction is the time required for commit processing, that is, the time elapsed between the master sending the PREPARE message and the master forgetting the transaction after all the necessary responses—including acknowledgements—have been received. We see from this figure that the value of commit-execution ratio is significantly higher for 3PC as compared to other protocols, which highlights the extent of the extra commit overheads of 3PC. At very high MPLs, the commit overheads become insignificant as compared to the response time for all the protocols which results in the commit-execution ratio for all protocols asymptotically approaching zero. As explained earlier, while the commit overheads for PC are less as compared to 2PC, there is no significant difference in the response time. This results in a lower value of commit-execution ratio for PC as shown in the figure. Finally, in Figure 5.1f, we graph the "borrow ratio" for OPT, that is, the average number of data items borrowed per transaction. This graph clearly shows that borrowing comes into the picture at higher MPLs, resulting in improved performance for OPT.

5.4 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. 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 5.2a–5.2f. We observe from these figures that, as in the previous experiment, CENT shows the best performance and the performance of

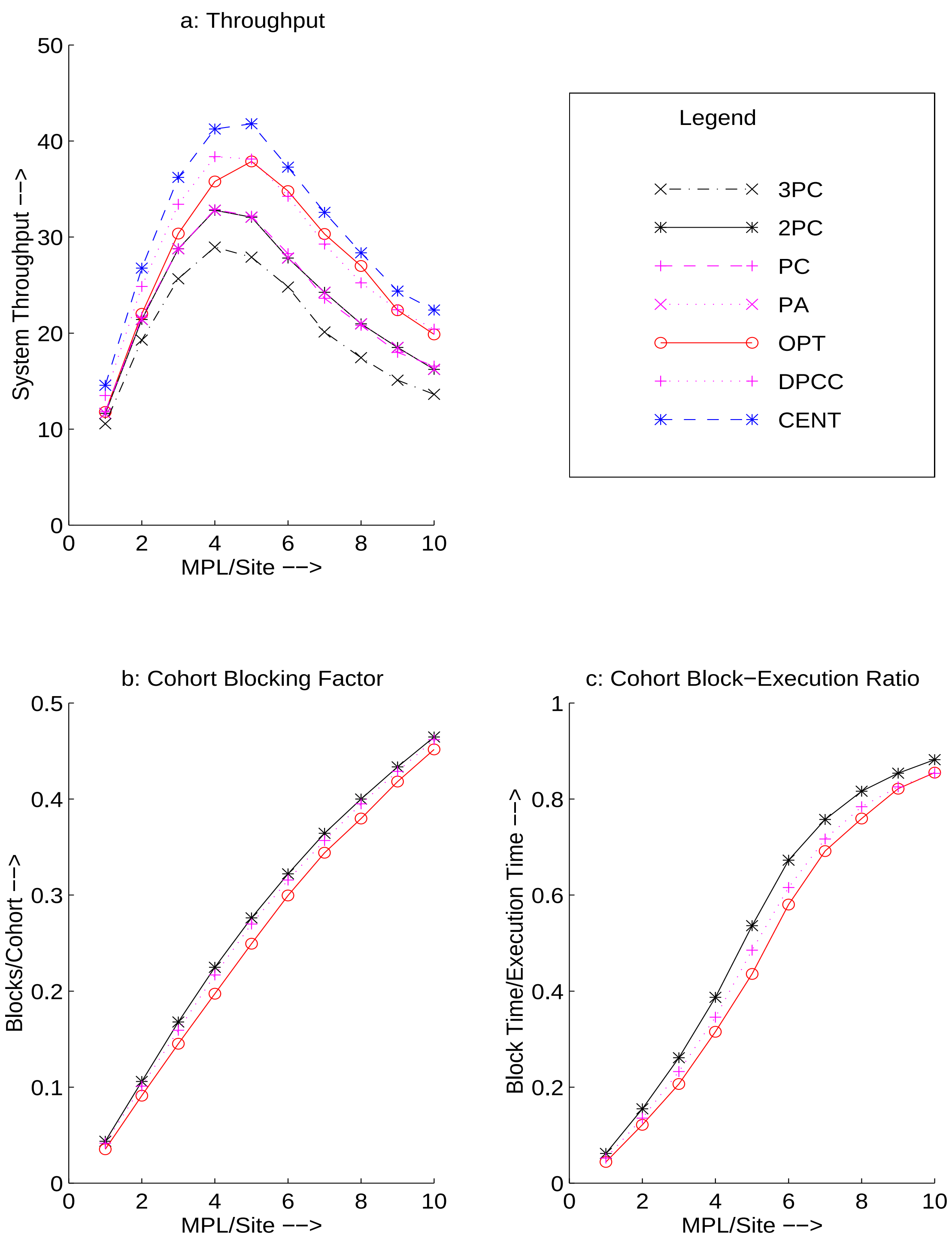


Figure 5.2: (a–c) Pure Data Contention (Sequential)

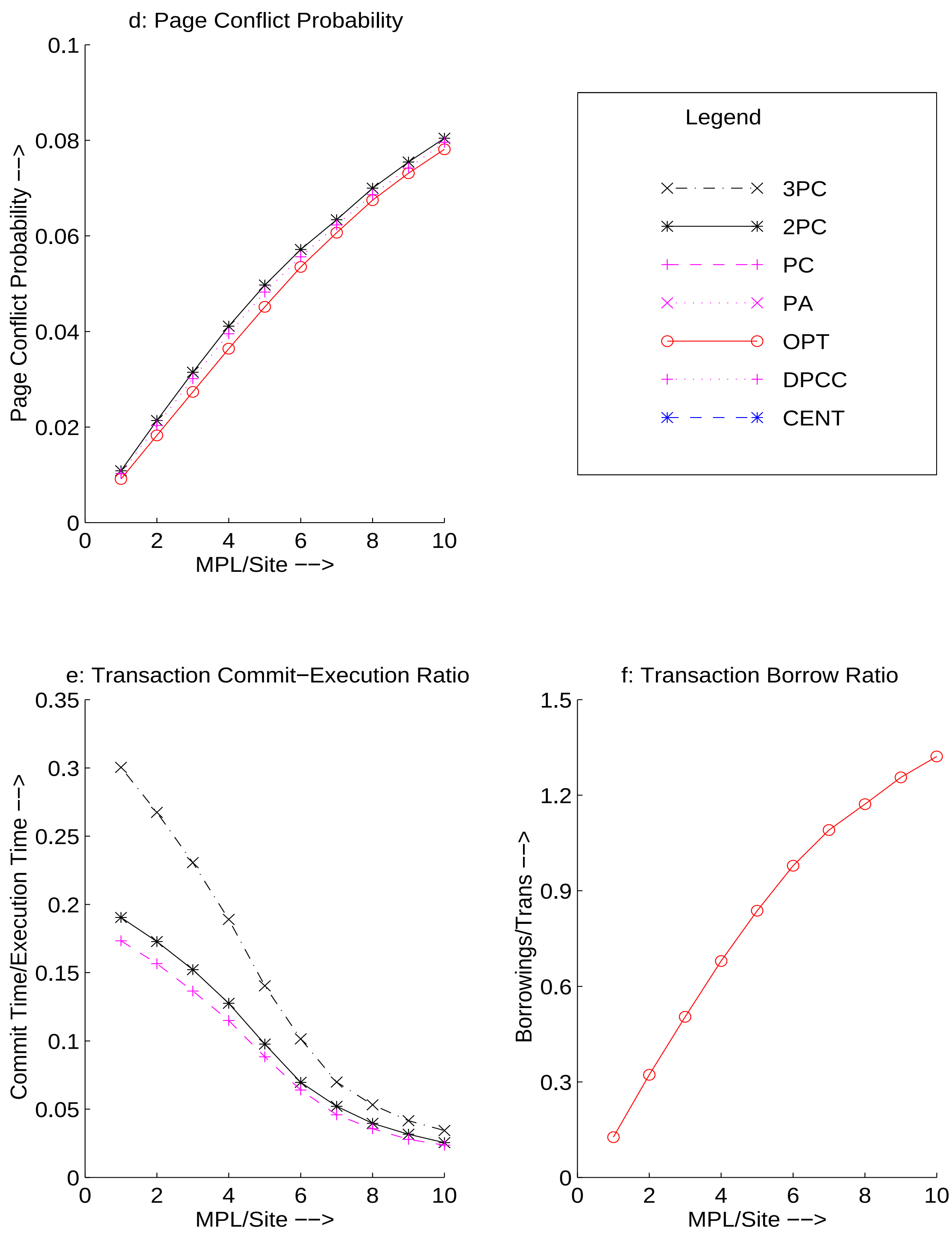


Figure 5.2: (d–f) Pure Data Contention (Sequential)

DPCC is slightly worse than that of CENT. The performance of the standard protocols relative to the baselines is, however, markedly worse than before. This is because under RC+DC (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. This is evident from Figure 5.2e where the commit-execution ratio for all protocols is higher than the corresponding values in Experiment 1 (Figure 5.1e). 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.

We also notice from Figure 5.2a that the difference between the performance of CENT and DPCC is considerably less than difference between the performance of DPCC and 2PC. This demonstrates that distributed *commit* processing (difference between the performance of DPCC and 2PC) can have considerably more effect than distributed *data* processing (difference between the performance of CENT and DPCC) on the throughput performance. In fact, similar results are observed in our other experiments as well.

Moving on to OPT, we observe that at 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. This is because in this experiment, OPT has more impact due to higher value of commit-execution ratio for 2PC as compared to Experiment 1 (compare Figure 5.2e with Figure 5.1e). In fact, OPT's peak throughput is close to that of DPCC.

In summary, this experiment indicates that under high data contention, OPT's performance can be substantially superior to that of the standard protocols. Note that while resource contention can be reduced by employing faster (possible due to technological improvements) or more resources, there exist no equally simple mechanisms to reduce *data* *contention.*

5.5 Experiment 3: Parallel Transaction Execution

In the previous experiments, we considered *sequential* transaction execution, that is, the cohorts of a transaction executed at various sites one after the other in a sequential fashion. In this experiment, we consider *parallel* transaction execution where all cohorts of a transaction execute at various sites in parallel. The goal is to investigate how parallel transaction execution affects the relative throughput performance of various protocols. We conducted this experiment for both resource-cum-data contention (RC+DC) and pure data contention (DC) scenarios. The parameter values under RC+DC and pure DC scenarios were the same as those used in Experiment 1 and Experiment 2, respectively—except *TransType*, which of course is "parallel" for this experiment. Figures 5.3a–5.3f present the results of this experiment for the RC+DC scenario while Figures 5.4a–5.4f present the results for the pure DC scenario.

From these figures we see that the observations made in the previous experiments for sequential transactions hold true, to a large extent, for parallel transactions also. The impact of distribution on the throughput performance of the system is quite significant. The performance of PC is similar to that of 2PC, and the performance of 3PC is considerably worse than that of 2PC. Finally, OPT continues to provide significantly better throughput performance than the standard protocols 2PC, PA and PC.

One interesting feature that we notice in this experiment is that the performance differences between CENT and the standard protocols (2PC,PA, PC) are *amplified* as compared to carrying out the same experiment with sequential transactions (compare Figures 5.3a and 5.4a with Figures 5.1a and 5.2a, respectively). At the same time, the relative performance differences between CENT and DPCC are decreased as compared to those observed in the sequential transaction experiments. This is because the parallel execution of the cohorts of a transaction reduces the response time of the transaction when the resources are not saturated, as in this experiment. Thus the throughput performance of both CENT and DPCC increases substantially. This increase in throughput

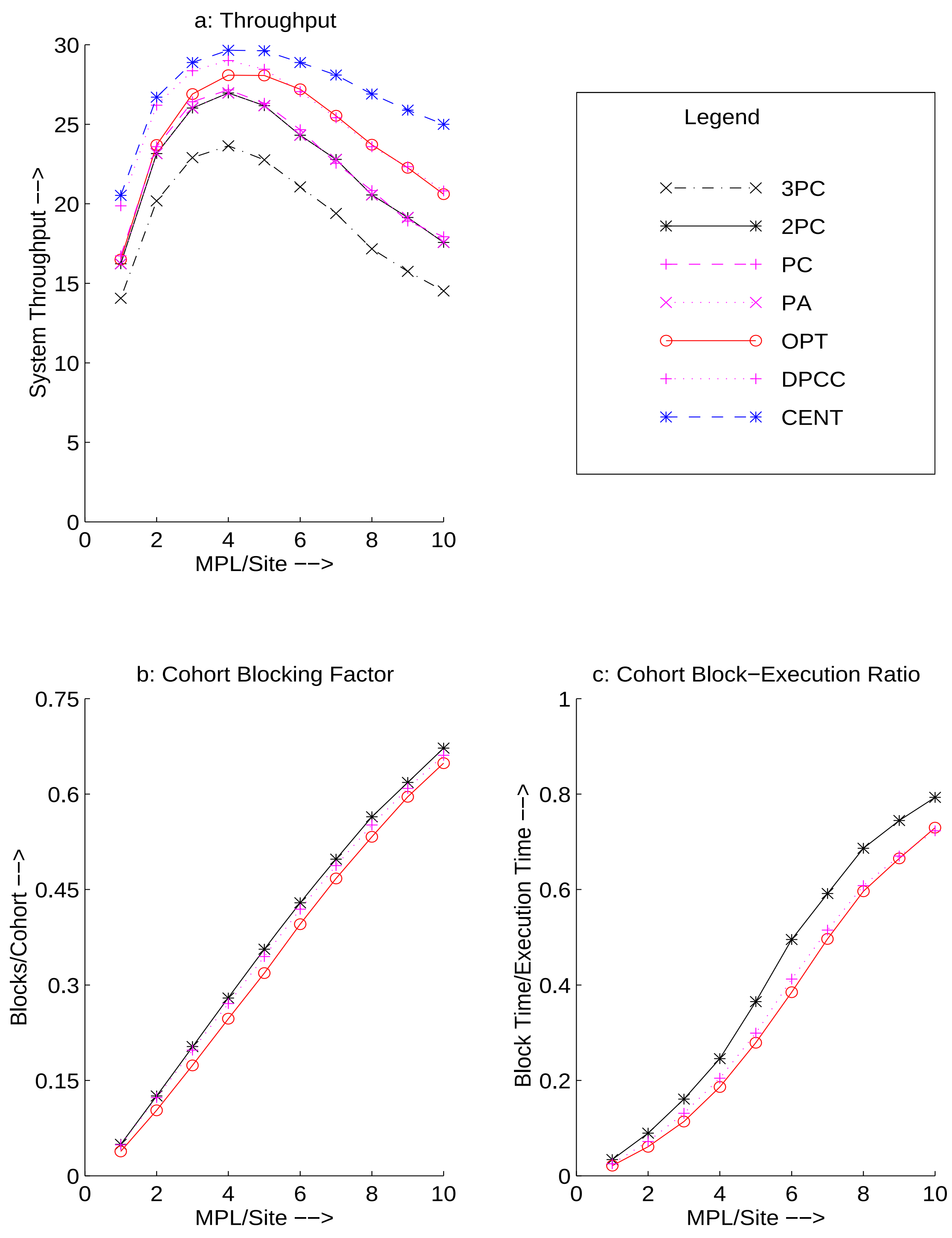


Figure 5.3: (a–c) Parallel Transactions (RC+DC)

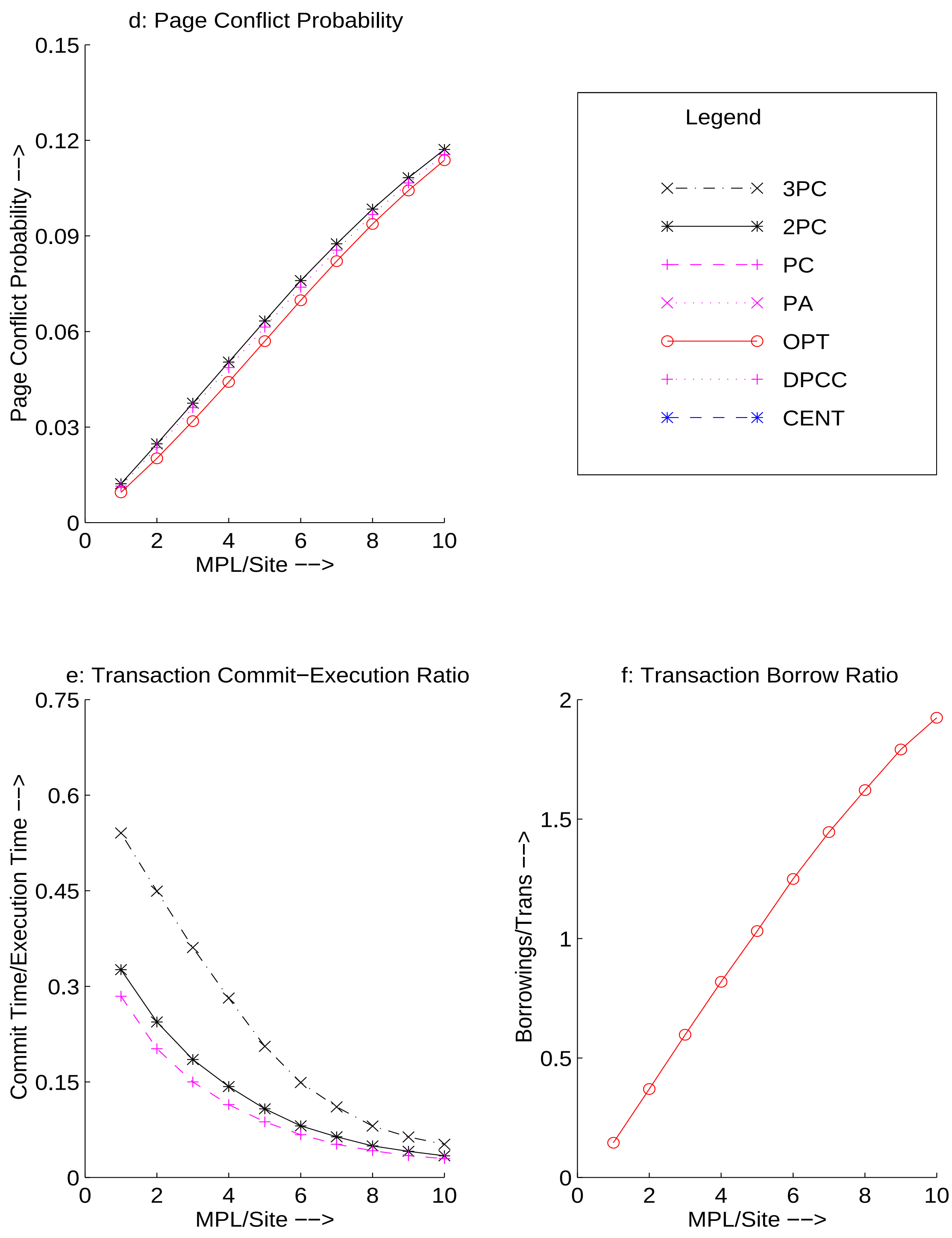


Figure 5.3: (d–f) Parallel Transactions (RC+DC)

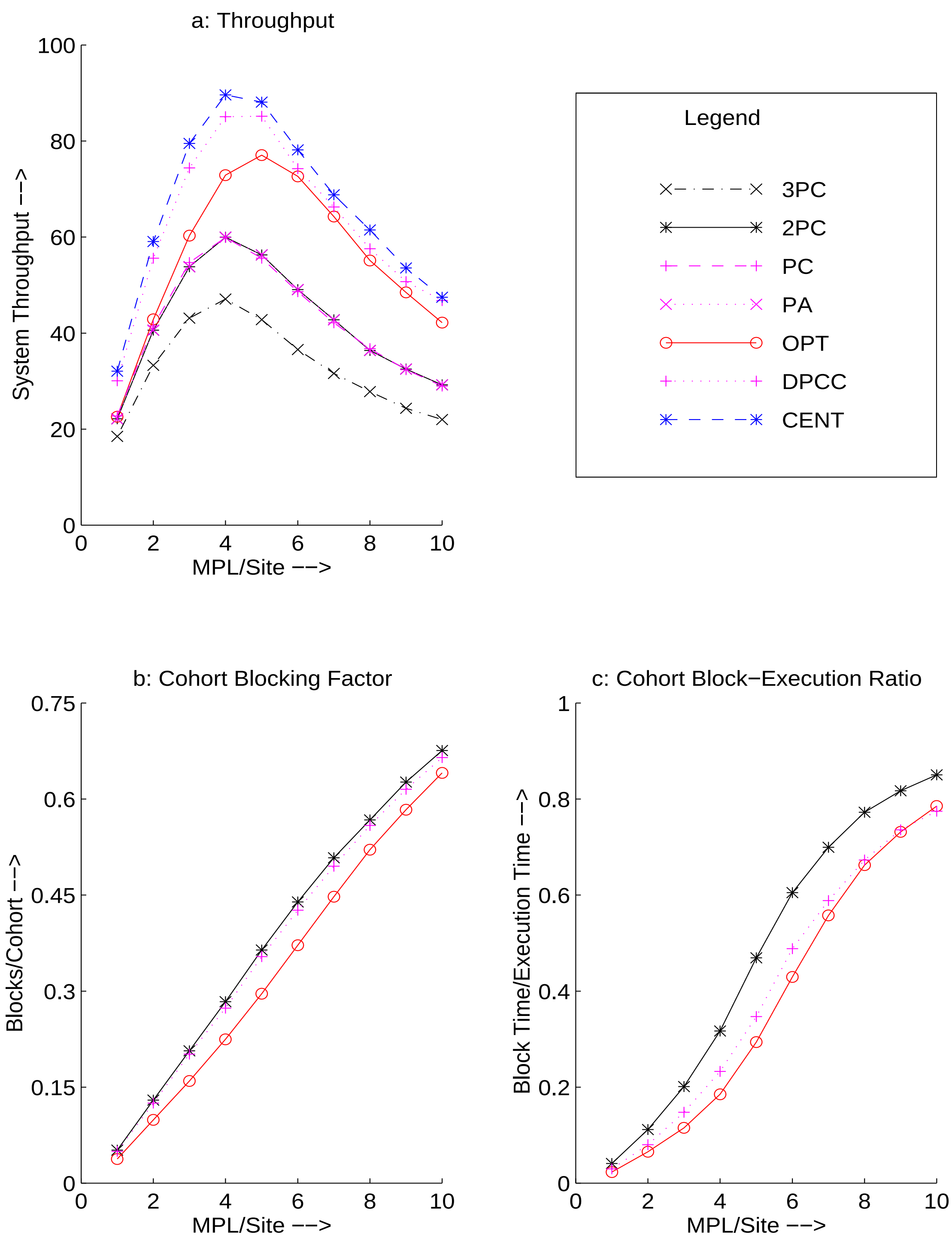


Figure 5.4: (a–c) Parallel Transactions (Pure DC)

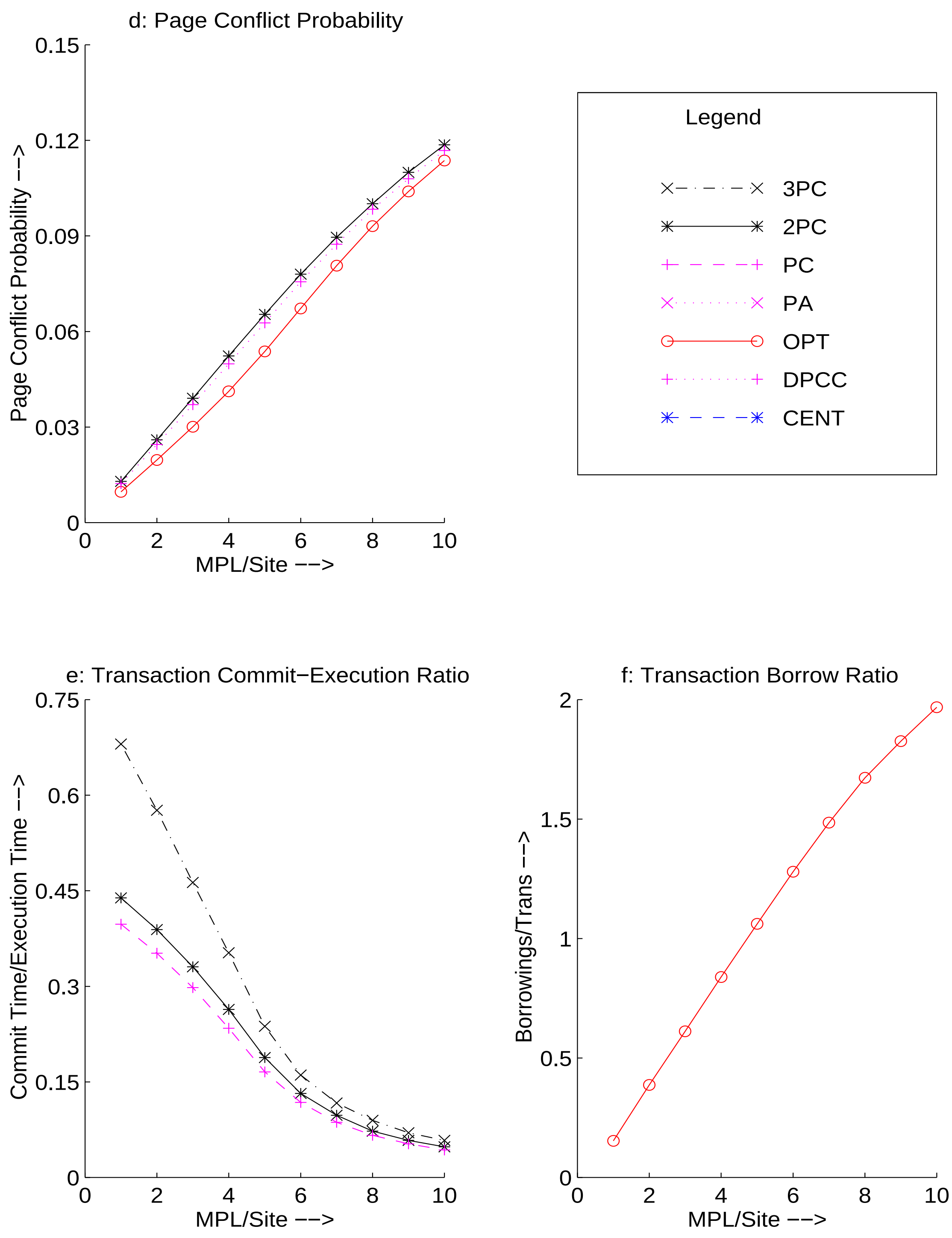


Figure 5.4: (d–f) Parallel Transactions (Pure DC)

performance is almost same for both CENT and DPCC. Also, the overheads of DPCC over CENT are not very significant, and are processed in parallel. Therefore, the performance of DPCC is close to that of CENT for the parallel case. On the other hand, for the standard protocols, the commit overheads in parallel transaction execution constitute a larger fraction of the transaction response time as compared to sequential transaction execution. While the length of the execution phase is more for sequential transactions than for parallel transactions, the length of the commit phase remains the *same* for both types of transaction executions. Hence, the impact of commit overheads on the throughput performance is more in the case of parallel transaction execution. This is evident from Figures 5.3e and 5.4e where the commit-execution ratio for the standard protocols is significantly higher than the corresponding values for the sequential transactions (Figures 5.1e and 5.2e).

We also observe from Figures 5.3a and 5.4a that the performance differences between OPT and 2PC are amplified compared to the sequential transactions. Commit overheads are more significant for parallel transactions than for sequential transactions for the reasons discussed above. Because the commit-execution ratio for parallel transactions is significantly higher than that for sequential transactions, the impact of OPT on the throughput performance is more for parallel transactions. This is also clear from Figures 5.3f and 5.4f, where OPT's borrow ratio for parallel transactions is significantly higher than that for sequential transactions (Figure 5.1f and 5.2f).

Another interesting observation from Figure 5.4a is that 2PC, PC, DPCC and CENT all achieve their peak throughput at an MPL of 4, while for OPT, the corresponding MPL value is 5. This is explained by considering Figure 5.4b, which shows the cohort blocking factor. In this figure, we see that cohort blocking factor is significantly lower for OPT than the other protocols for the same multiprogramming level due to reducing `prepared data-blocking`. Also, from Figure 5.4d, we see that the page conflict probability is significantly lower for OPT than other protocols for the same MPL. Thus for a given data contention level, OPT allows more concurrency in the system than standard protocols. Similar observations hold true for other experiments considered till now

(Figures 5.1a, 5.2a, and 5.3a), though the difference in the MPLs at which the peak throughput is achieved, is more striking in Figure 5.4a.

5.6 Experiment 4: Fast Network Interface

In the previous experiments, the cost for sending and receiving messages modeled a system with a slow network interface ($MsgCpu = 5\ ms$). We conducted another experiment wherein the network interface was faster by a factor of five, that is, $MsgCpu = 1\ ms$.

The results of this experiment under RC+DC and under pure DC are shown in Figures 5.5a and 5.5b present the results for sequential transaction execution under RC+DC and under pure DC scenarios, respectively, while Figures 5.5c and 5.5d present these results for parallel transaction execution.[2] We observe from these figures that the performance of all the protocols becomes closer to that of CENT as compared to the previous experiments, and in fact, for parallel transaction execution under pure DC, CENT and DPCC are virtually 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 force-writes are significant enough to point out clear differences between the performances of DPCC and 2PC and those of 2PC and 3PC (Figures 5.5b and 5.5d). Finally, we see that OPT's peak throughput performance is again close to that of DPCC in all the cases.

This experiment shows that adopting the OPT principle can be expected to be of value even in the gigabit networks of the future. While such technological improvements may help to reduce resource contention and provide faster processing capabilities at the network interface, there exists no equally simple mechanism to reduce *data contention*.

[2]The default parameter values under 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. Now onwards both sequential and parallel transactions are considered.

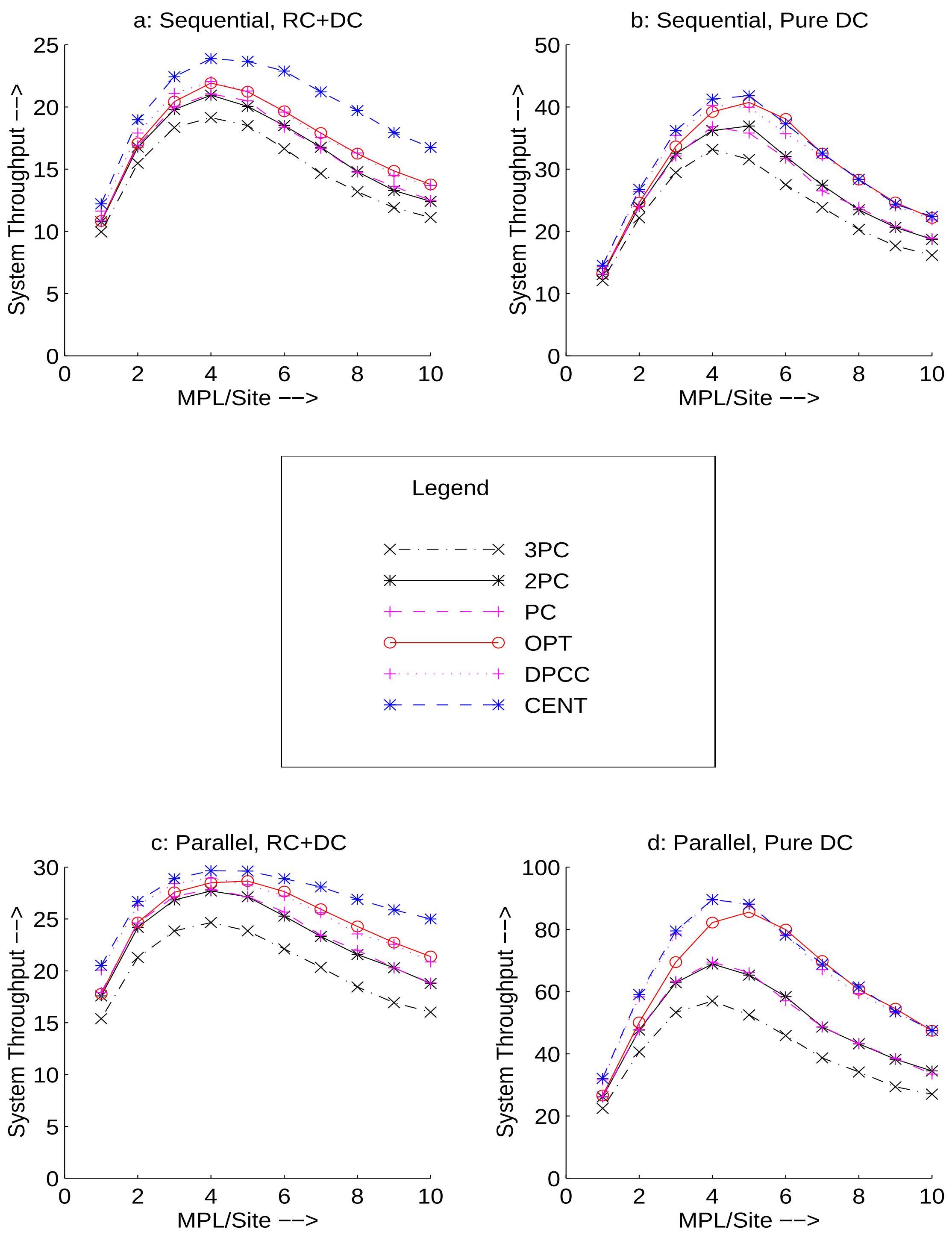


Figure 5.5: Fast Network (Throughput)

5.7 Experiment 5: 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 on *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 (for committing transactions) are shown in Table 5.3. Figures 5.6a and 5.6b present the transaction throughput results of this experiment for sequential transaction execution under RC+DC and under pure DC, respectively. Figures 5.6c and 5.6d present these results for parallel transaction execution.

For the RC+DC case (Figures 5.6a and 5.6c), we observe that the performance differences between CENT and DPCC are smaller as compared to those for the limited distribution workload experiments (Figures 5.1a and 5.3a). As explained in Section 5.2, transactions in CENT utilize the physical resources in a more efficient way (by means of load balancing) because all the physical resources in the system are available for processing a transaction in the system. This is not the case with DPCC where a cohort can utilize only those resources available at its own site. Heightened degree of distribution thus makes more resources available to a transaction, because now the transaction runs at more sites. This results in increased performance of DPCC, thus reducing the performance difference between CENT and DPCC. The increased message overheads of DPCC do not have much impact because these overheads are not sufficient to make the CPU to be the bottleneck. Moreover, for the parallel case these overheads are processed in parallel. In fact, for the parallel case, the performance difference between CENT and DPCC virtually disappears.

We also observe from Figures 5.6a and 5.6c that the performance differences between the baselines (CENT and DPCC) and the standard protocols (2PC, PA, and PC) are visibly more than those seen for the limited distribution workload experiments (Figure 5.1a and 5.3a). This is because the message overheads of the standard protocols in this experiment are sufficiently large (Table 5.3) that the system now operates in a heavily

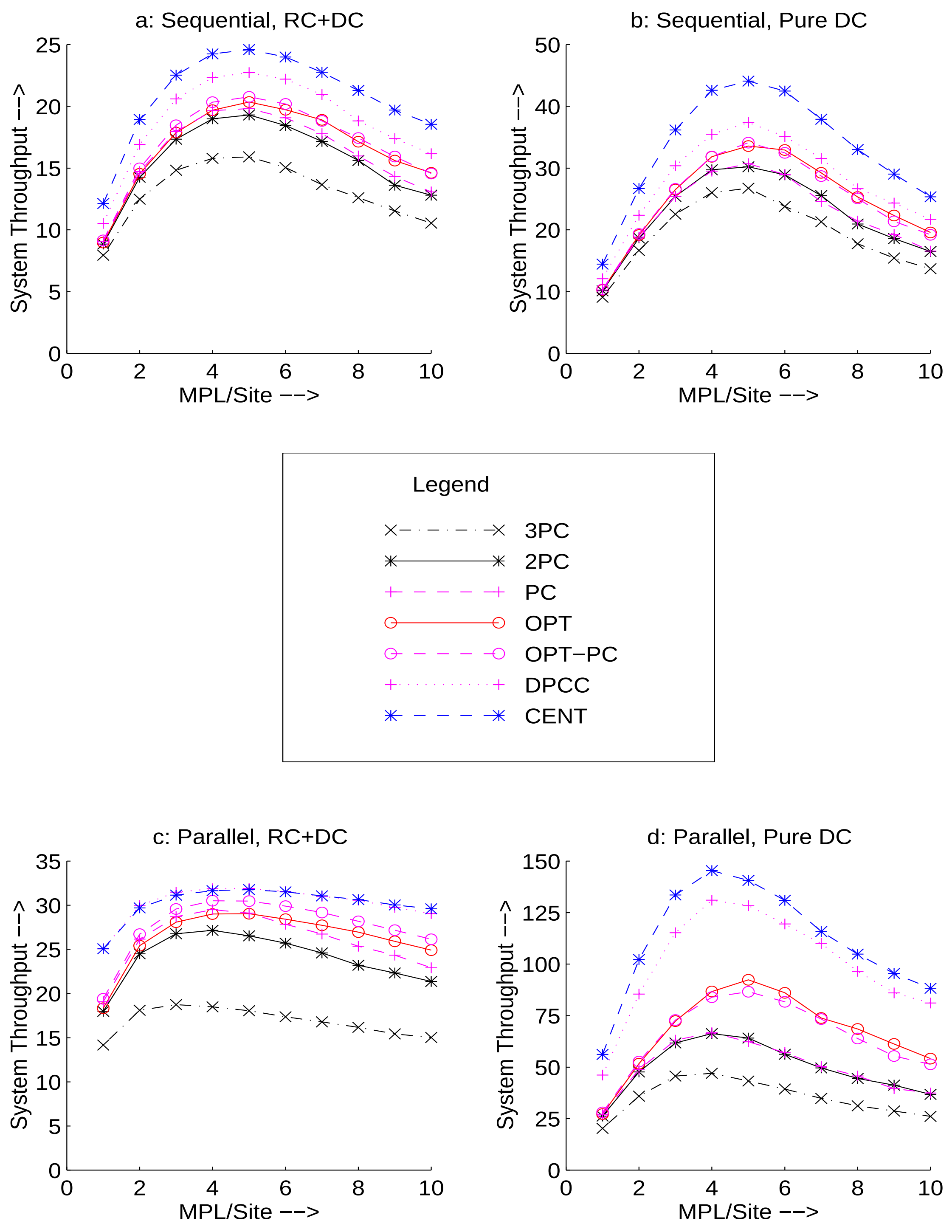


Figure 5.6: Distribution Degree = 6 (Throughput)

Table 5.3: Protocol Overheads for Committing Transaction (DistDegree = 6)

Protocol	Execution	Commit	
	Messages	Force-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

CPU-bound region. For the same reason the performance of 3PC is significantly worse than that of 2PC.

An interesting feature of Figure 5.6c 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 loading range. Similar behavior is observed for the sequential transactions also (Figure 5.6a), though to a lesser extent. The reason for this behavior is as follows: Due to the higher degree of distribution, there is a substantial increase in the message overheads of 2PC (Table 5.3), resulting in the system operating in a heavily *CPU-bound* region. This was confirmed by measuring the CPU and disk utilizations. For example, for 2PC at peak throughput about 60 percent of the CPU was utilized for message processing and the remainder for data processing. In this environment, PC's reduced overheads result in its performing better than 2PC. This is further confirmed in the results in Figures 5.6b and 5.6d, 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 (Figures 5.6a and 5.6c), the performance differences between OPT and 2PC are smaller as compared to those in the limited distribution workload experiments (Figures 5.1a and 5.3a). This is due to two factors: (i) the CPU-bound nature of the workload, and (ii) message processing has higher priority over data processing. As a result of these factors, the increase in execution phase length is much larger than the increase in commit phase length, resulting in a smaller "commit-execution ratio". Since OPT's impact is felt only during the commit phase, the decrease in this ratio causes reduced performance improvement as compared to the limited distribution workload experiments (Figures 5.1a and 5.3a). 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 Figures 5.6a and 5.6c, wherein it provides the best overall performance due to deriving the benefits of both the OPT and PC optimizations.

Under pure data contention (Figures 5.6b and 5.6d), note that the performance differences between CENT and DPCC are 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 average transaction response time. We also observe in Figure 5.6d that the difference between the performance of DPCC and that of 2PC is now very large (the peak throughput of DPCC is more than *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 Figures 5.6b and 5.6d. The performance of OPT continues to be superior to 2PC, but in contrast to what was seen for RC+DC case (Figures 5.6a and 5.6c), 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 length of the execution phase of the transaction (by introducing `COLLECTING` force-write) and at the same time to reduce the commit phase length (by saving on force-writes at cohorts), thus decreasing the commit-execution ratio and diminishing the utility of the optimistic feature.

5.8 Experiment 6: Reduced Update Probability

In the previous experiments, the update probability of a page was set to 1, that is, each page accessed by the transaction was updated. We conducted another experiment where

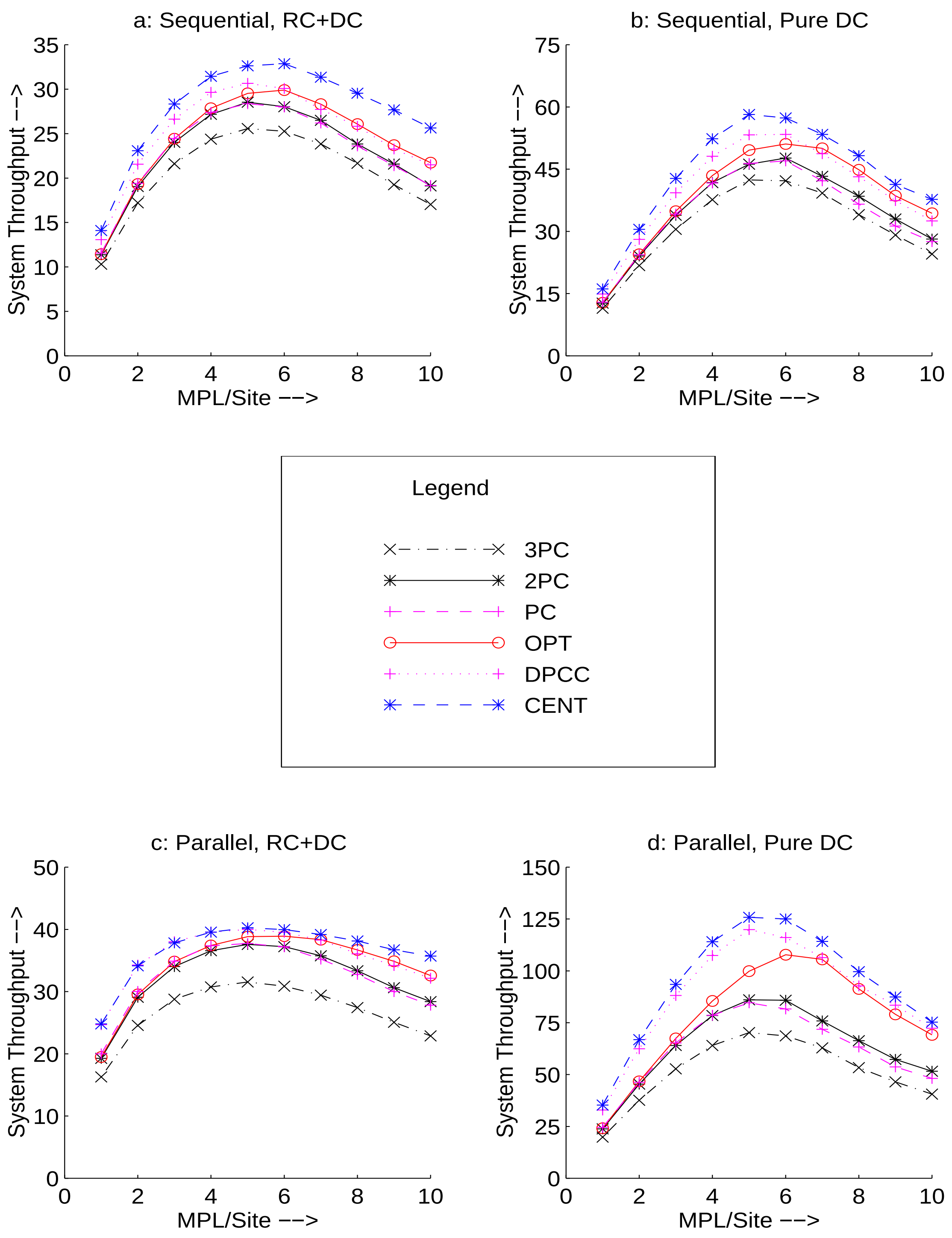


Figure 5.7: Update Probability = 0.5 (Throughput)

the value of *UpdateProb* was set to 0.5, that is, each page accessed by the transaction was updated with 0.5 probability. This affects the contention level in the system in the following two ways: (a) the read locks are shared, that is, if a cohort has already locked a page in the read mode, it can share the lock with another cohort also willing to access the page in read mode, and (b) the cohorts release the read locks before sending the votes in response to the PREPARE message from the master. This results in lower levels of data contention in the system as compared to the fully update workloads for the same multiprogramming levels.

For this experiment, Figures 5.7a and 5.7b present the transaction throughput results for sequential transaction execution, and Figures 5.7c and 5.7d for parallel transaction execution, under RC+DC and under pure DC, respectively. These results show mostly similar behavior to that observed in the corresponding experiments for fully update transactions discussed earlier. One point worth mentioning here is that now under RC+DC case, the performance difference between OPT and 2PC decreases to some extent. This is because as the data contention in the system decreases, the utilization of the physical resources increases and the physical resources become the bottleneck. Both these factors (the decrease in data contention and the physical resources being the 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 corresponding fully update transactions. However, for the pure data contention case, as physical resources are not the bottleneck, performance of OPT remains comparable to that seen for fully update transactions.

5.9 Experiment 7: 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 5.8a–5.8d present the throughput results for both sequential and parallel transactions under RC+DC and pure DC conditions. For

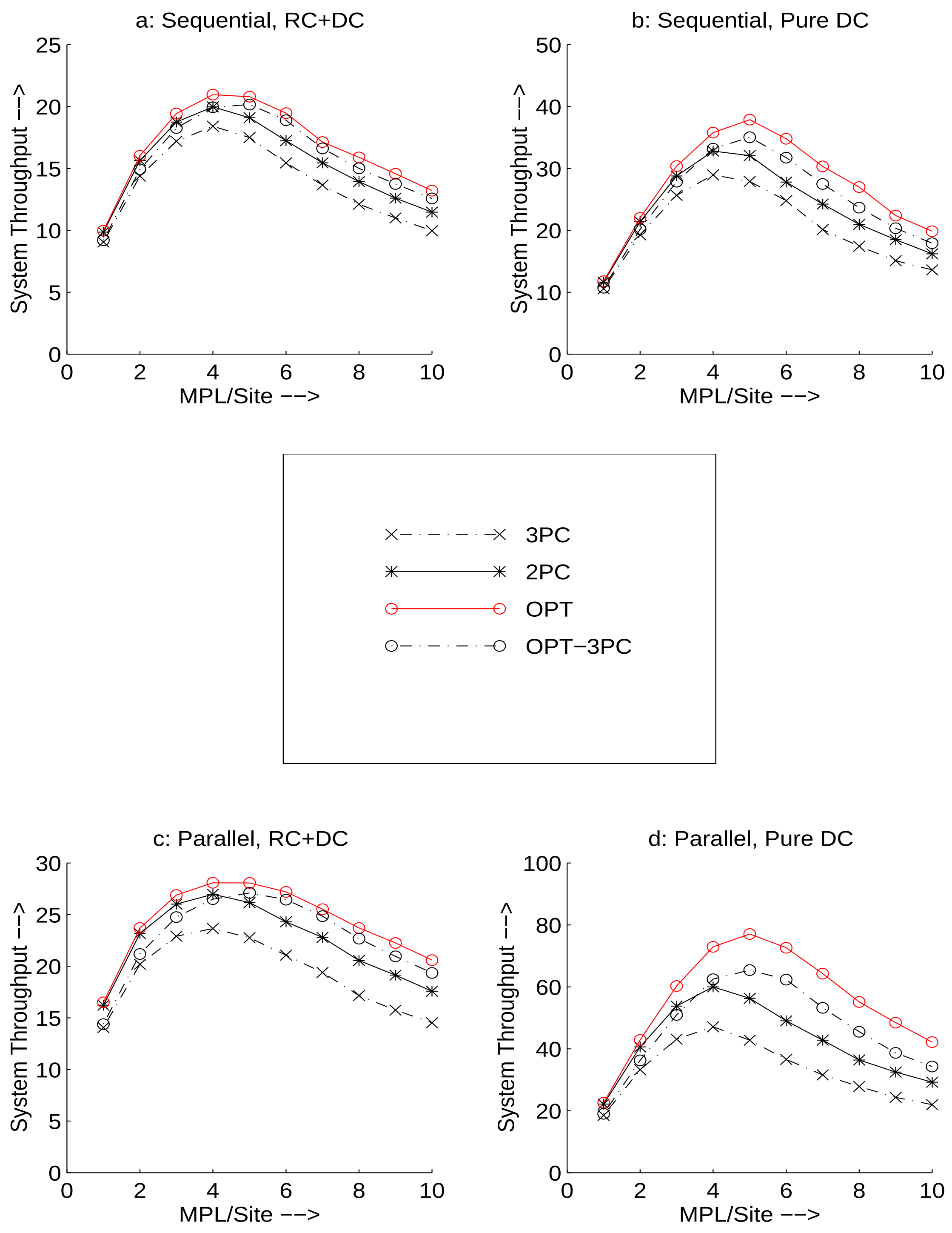


Figure 5.8: Non-Blocking OPT (Throughput)

the RC+DC case (Figures 5.8a and 5.8c), we observe that the performance of OPT-3PC is similar to that of 3PC at lower MPLs. However, at 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 under pure DC condition (Figures 5.8b and 5.8d). In both these figures, the peak throughput of OPT-3PC actually significantly *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 length of the prepared state is significantly longer in 3PC (due to the extra phase), thereby increasing the benefits of borrowing. This fact is clear from the commit-execution ratio curves in Figures 5.1e–5.4e, where the commit-execution ratio for 3PC is significantly higher than other protocols.

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.

5.10 Experiment 8: Surprise Aborts

In all of the experiments discussed earlier, there was no potential for serializability-induced aborts in the commit processing stage since the distributed strict 2PL protocol was used for concurrency control, as explained in Section 4.7. It is for this reason that no difference was possible in the relative performance of PA and 2PC. In practice, however, aborts may arise in the commit phase due to other reasons such as violation of integrity constraints, software errors, system failure, etc. We therefore conducted an experiment where such "surprise abort" situations were modeled and evaluated their impact on protocol performance. The important point to note here is that these situations are fundamentally biased against OPT since its underlying assumption that transactions will usually commit may not hold under surprise aborts.

In this experiment, each cohort, on receiving the PREPARE message from the master,

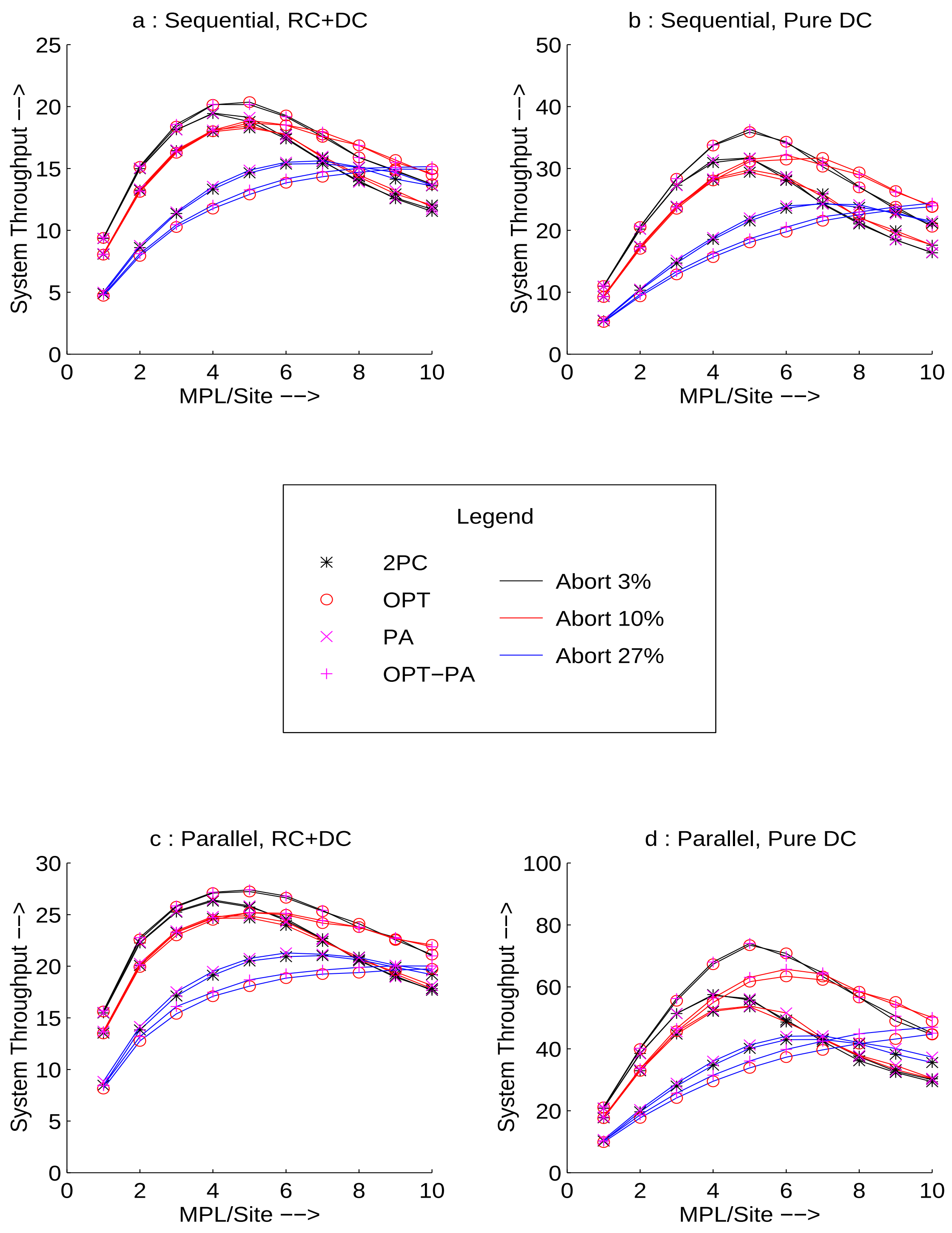


Figure 5.9: (a–d) Surprise Aborts (Throughput, $DistDegree = 3$)

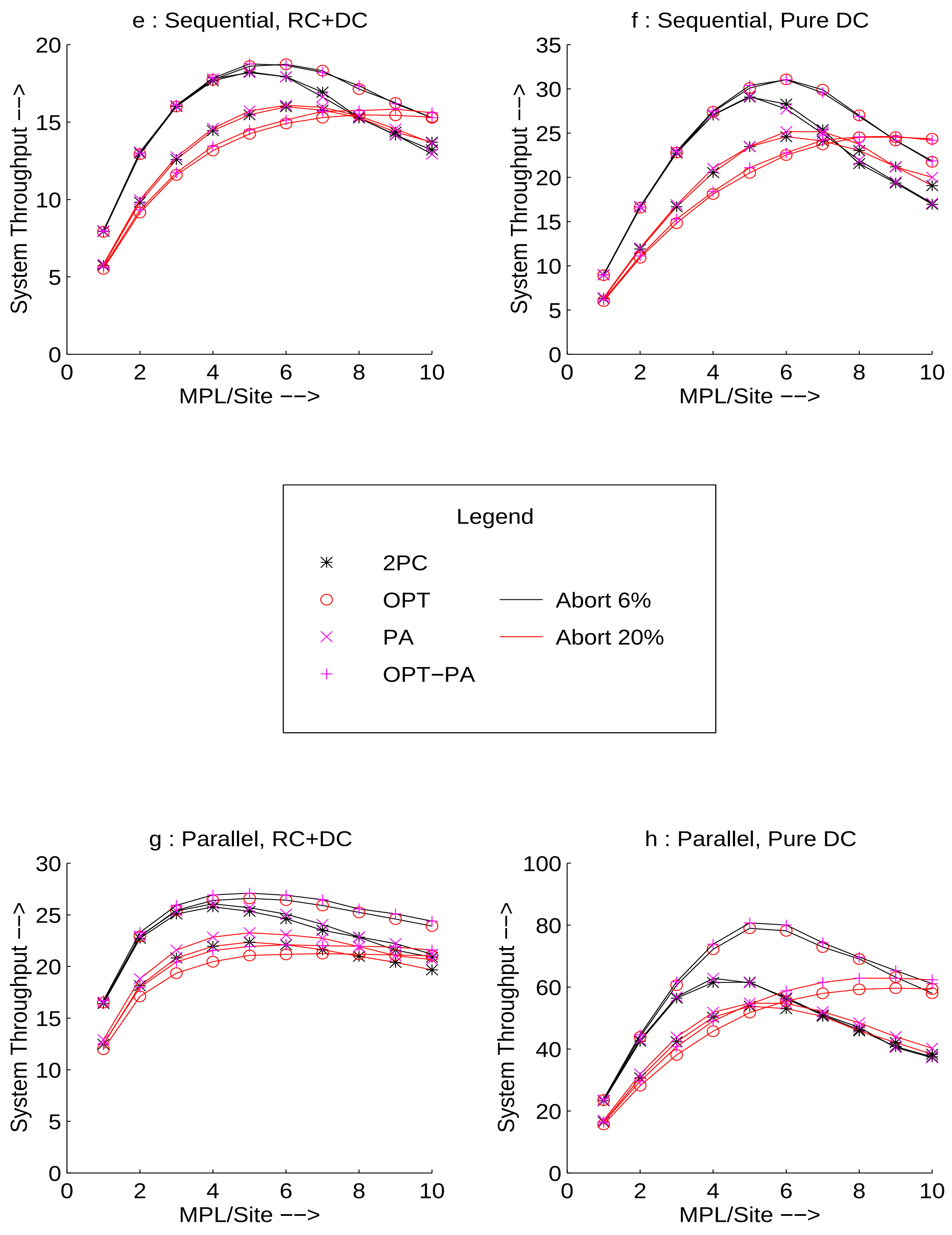


Figure 5.9: (e–h) Surprise Aborts (Throughput, $DistDegree = 6$)

instead of always voting YES, votes NO with a certain probability. We consider three different cases, where the probability of a cohort aborting arbitrarily is 1 percent, 3.5 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, 10 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*.

Figures 5.9a and 5.9b present the transaction throughput results of this experiment for sequential transaction execution under RC+DC and under pure DC, respectively. Figures 5.9c and 5.9d present these results for parallel transaction execution. We observe here that for RC+DC case (Figures 5.9a and 5.9c), OPT's peak throughput performance is comparable to that of 2PC even when 10 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. However, under pure DC, even though at lower MPLs the performance of OPT is slightly worse than that of 2PC, the peak throughput performance of OPT is comparable to that of 2PC even at the abort levels as high as 27 percent (Figures 5.9b and 5.9d).

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 force-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 force-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 5. The results of this experiment are shown in Figures 5.9e–5.9h for both sequential and parallel transactions under RC+DC and pure DC scenarios (cohort abort probabilities used are 1 percent and 3.5 percent which now translate to overall transaction abort probabilities of approximately 6 percent and 20 percent). As is clear from Figure 5.9c, the savings by PA for the 20 percent transaction abort level are 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 these figures for both limited distribution workload and high distribution workload and shows the expected outcome.

Another interesting feature in these figures is that, at high surprise abort levels and high values of MPL, there are there are two kinds of *performance crossovers*. In the first type of crossover, the performance of OPT is initially worse than that of 2PC, but as the MPL increases, it becomes better than that of 2PC (for example, in Figure 5.9d, see the 2PC and OPT curves for 27 % abort level at an MPL of 8). The reason for this behavior is the following: Initially the performance of OPT is worse than that of 2PC because at high abort levels the optimism of borrowing transactions proves to be misplaced, as explained earlier. However, at high MPLs the system is in the thrashing region for 2PC (as is evident form the declining throughput performance curve of 2PC) and its performance at the crossover point is below its peak value, while the throughput performance curve of OPT yet has to achieve its peak value (because OPT attains its peak performance at higher value of MPL as compared to 2PC, as explained earlier) and its value at the crossover point is close to its peak value. Note, however, that due to very high abort probability the *peak* throughput performance of OPT is worse than that of 2PC.

In the second type of performance crossover, we observe that as the MPL increases the performance of OPT and 2PC protocols in a system with higher probability of surprise aborts becomes better than the corresponding performance in a system with lower probability of such aborts. For example, see 2PC and OPT curves for 10 % abort level, and the corresponding curves for 27 % abort level, at an MPL of 7, in Figure 5.9d. The reason for this crossover is the following: At the crossover point, the system is in the thrashing region for curves with 10 percent abort levels, while for performance curves with the 27 percent abort levels the system is *not* in the thrashing region. This is because in our model (as in earlier transaction management studies, for example, [ACL87]) aborted transactions are delayed before restarting with the delay period being equal to the average response time. This delay effectively becomes a crude way of controlling the data contention in the system. Thus a system with higher abort levels enters the thrashing region at a higher value of MPL as compared to a system with lower abort levels. Therefore, at the crossover point the performance of 2PC (resp. OPT) curve with 10 percent abort level is decreasing and has a lower value than its peak value, while the performance of 2PC (resp. OPT) curve with 27 percent abort level is increasing and has a value that is close to its peak value. Note, however, that the *peak* throughput is significantly smaller for the curves with 27 percent abort levels than those with 10 percent abort levels.

5.11 Other Experiments

We conducted several other experiments to explore various regions of the workload space. These included workloads with small database size, workloads with inter-transaction think-time at each terminal, different sets of physical resources with different processing speeds, etc. The relative performance of the protocols in these experiments remained qualitatively similar to that seen in the experiments described so far. The performance improvement delivered by OPT was dependent on the level of data contention in the system. Here, we present the results for two such experiments: small database size, and inter-transaction think-time.

Figures 5.10a–5.10d present the throughput results for the small database size experiment for sequential and parallel transaction execution under RC+DC and pure DC scenarios. For this experiment, the value of $DBSize$ parameter is set to 2400 pages instead of 8000 pages used in earlier experiments. Note that the effect of having a smaller database size is to increase the data contention in the system. The results of this experiment are as expected. Even though the quantitative performance numbers are different,

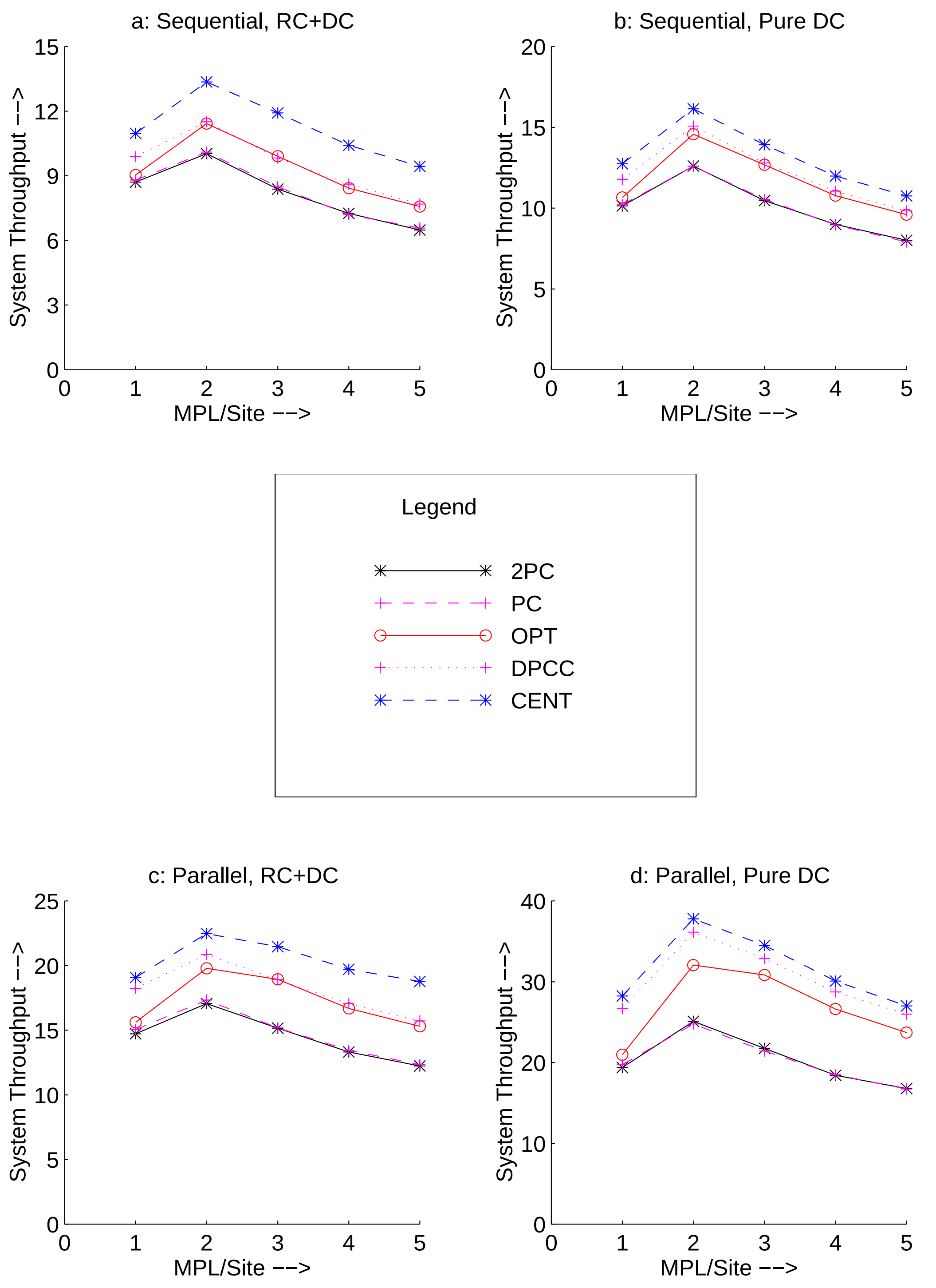


Figure 5.10: $DBSize = 2400\ Pages$ (Throughput)

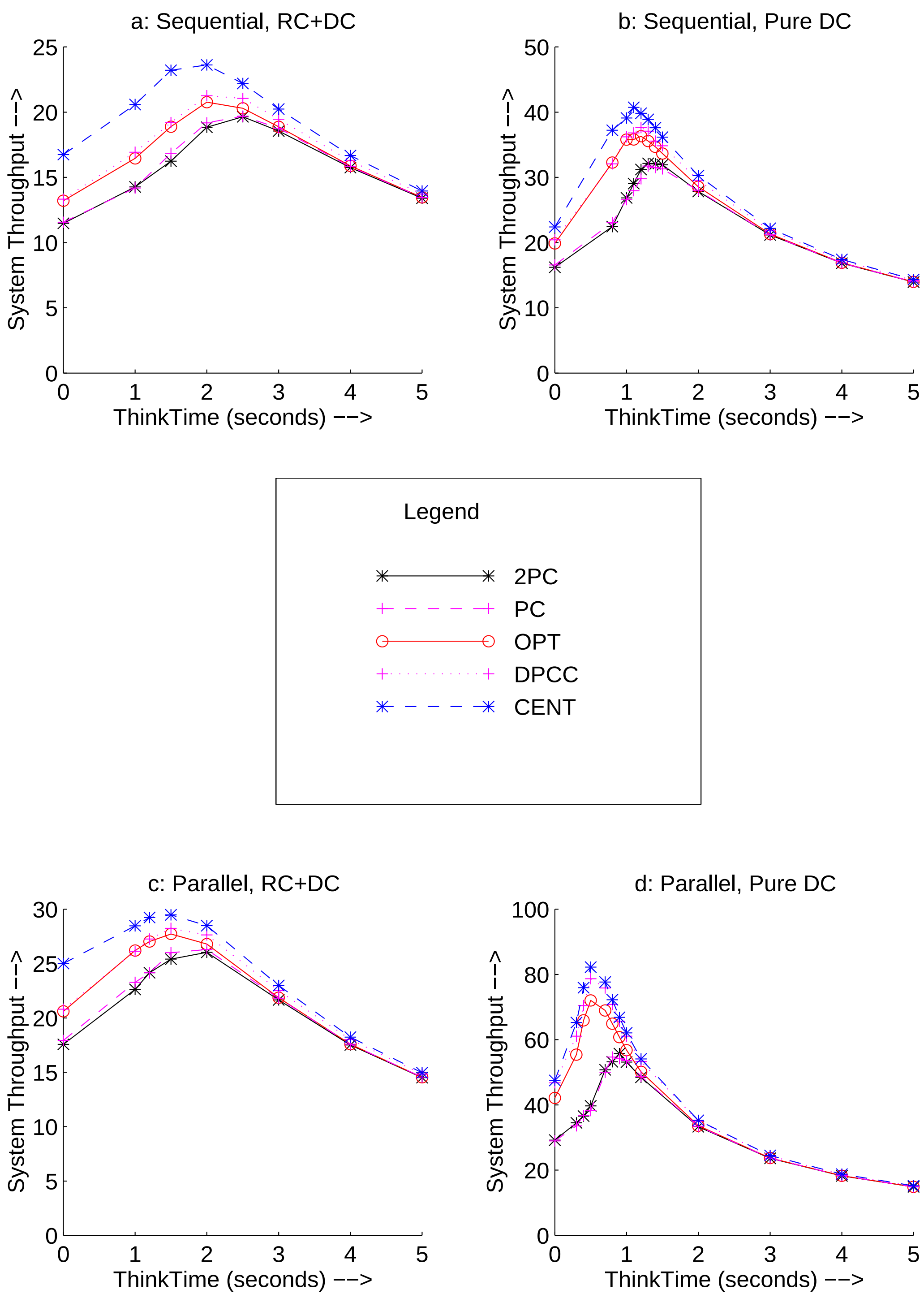


Figure 5.11: Think Time (Throughput, $MPL = 10$ per site)

the relative performance of the protocols is similar to that observed in Experiments 1–3 (Figures 5.1a–5.4a). In most cases, the performance differences between DPCC and the standard protocols (2PC and PC) are significantly higher than those between CENT and DPCC. The performance of PC is very close to that of 2PC. Finally, OPT still performs considerably better than 2PC, its peak throughput performance being close to that of DPCC.

Figures 5.11a–5.11d present the throughput results for the *inter-transaction think-time* experiment for sequential and parallel transaction execution under RC+DC and pure DC scenarios. For this experiment, a fixed value of MPL is used which is set to 10 transactions per site. The parameter $ThinkTime$—mean of exponentially distributed think time between the completion of one transaction at a terminal and the submission of the next one at the same terminal—is varied from 0 to 5 seconds. Note that lower values of ThinkTime correspond to higher load in the system while the higher values of ThinkTime indicate a lightly loaded system.

From these figures, we see that at higher values of ThinkTime, the system is lightly loaded and the throughput performance of all the protocols is almost similar. The reason for this is that there are not enough transactions in the system leaving the resources underutilized. Similar results were observed in Experiments 1–3 (Figures 5.1a–5.4a) at low multiprogramming levels. At lower values of ThinkTime, the system becomes heavily loaded, and the results are again as expected. In most cases, the peak throughput of DPCC is close to that of CENT while the standard protocols (2PC and PC) perform considerably worse than DPCC. Again, the performance of PC is very close to that of 2PC. Finally, the performance of OPT is significantly better than that of 2PC, and is close to that of DPCC. At very low values of ThinkTime, the system starts thrashing which results in a decrease in the performance of all the protocols.

CHAPTER

6 Conclusions

In this part of the thesis, we have quantitatively investigated the performance implications of supporting distributed transaction atomicity in the context of OLTP systems. 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 systematic quantitative results in this area with respect to end-user performance metrics. Our experiments demonstrated the following:

1. Distributed *commit* processing for most workloads can have considerably more effect than distributed *data* processing on the 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 transaction aborts

was close to *thirty* percent—in practice, surprise aborts are expected to occur only occasionally.

We caution the reader here 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 study. 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 done by allowing only prepared transactions to lend their data and ensuring that borrowers cannot enter the prepared state until their borrowing is terminated.

   The power of the optimistic access feature is substantial enough that OPT's throughput performance was often close to that obtained with the DPCC baseline, which represents an upper bound on achievable 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 thereby run the risk of having to revert to an executing 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 the transactions that lend their uncommitted data will almost always commit. However, OPT is fairly *robust* in that it maintains its superior performance unless the probability of surprise transaction

aborts in the commit phase is *unrealistically* high. In our experiments, OPT exhibited superior performance even when the transaction abort probability was as high as 10 percent. A *performance crossover* was observed at the transaction abort probability of 15 percent—a level much higher than what might be expected in practice. Beyond this level, OPT's performance becomes progressively worse than that of the classical protocols.

6. Under conditions where there is sufficient 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.

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. If having non-blocking functionality is important but 3PC has not been used due to its excessive overheads resulting in poor performance, then OPT-3PC appears to be an attractive choice since it provides a peak throughput that exceeds those of the standard 2PC protocols.

PART II

Real-Time Database Systems

CHAPTER

7 Overview of RTDB Systems

For the past decade, there has been a growing interest in the area of *Real-Time Database Systems (RTDB)*. The importance of research in this area has been emphasized in many workshops and conferences at the international fora in recent times [Son88, Son90, AGM92, Ulu92, Ulu94b, Bes96, BH95, BH96]. An RTDB system is an amalgam of a conventional real-time system and a database system. In a real-time system, the tasks have associated timing constraints, usually in the form of *completion deadlines*, and are scheduled in a way that they can be completed before their deadlines expire. Preserving database consistency is usually not considered in conventional real-time systems. On the other hand, timing constraints of the transactions are usually not addressed in conventional database systems. In an RTDB system, the transaction processing must not violate the database consistency, like conventional database systems. In addition, the RTDB system has to meet the timing requirements of the transactions. In other words, in order to successfully commit a transaction, the RTDB system has to satisfy the timing constraints imposed on the transaction commitment in addition to the logical consistency requirements of the transaction. The goal in an RTDB system usually is to maximize the *number* of transactions that complete before their deadlines expire, as opposed to minimizing the average transaction response time (or maximizing the average transaction throughput) which is the primary goal in a conventional database system.

7.1 Real-Time Framework

The applications in the real-time domain can be grouped into the following three categories based on how the application is impacted by the violation of the task completion deadline [HCL92]:

Hard Deadline Real-Time Applications: In these applications, the consequences of missing the deadline of even a single task could be catastrophic. Life-critical applications such as flight control systems or missile guidance systems belong to this category. Database systems for efficiently supporting hard deadline real-time applications, where *all* transaction deadlines have to be met, appear infeasible due to the large variance between the average case and the worst case execution times of a typical database transaction. The large variance is due to transactions interacting with the operating system, the I/O subsystem, and with each other in unpredictable ways. Guaranteeing completion of all transactions within their deadlines under such circumstances requires an enormous excess of resource capacity to account for the worst possible combination of concurrently executing transactions.

Soft Deadline Real-Time Applications: In these applications, the tasks do have associated deadlines, but even if a task fails to complete within the deadline, it is allowed to execute upto completion. Generally, in these systems, a "value function" assigns a value to the tasks. This value remains constant upto the deadline, but starts decreasing after the deadline. The questions to be addressed in these applications include how to identify the proper value function, which actually may be application dependent. The emphasis is on the number of tasks completing within their deadlines, as well as on the amount of *tardiness* of the tasks missing their deadlines.

Firm Deadline Real-Time Applications: These applications are different from the soft deadline applications in the sense that the tasks which miss the deadline are considered worthless (and may even be harmful if executed to completion) and are thrown out of the system immediately. The emphasis, thus, is on the *number* of tasks that complete within their deadlines.

Our interest in the RTDB systems is on the applications in the *firm deadline* real-time domain [HCL92]. We believe that understanding firm deadline RTDB systems will provide necessary insight into the RTDB technology which is necessary for addressing the more complex framework of soft deadline applications. Therefore, we have carried out our work from the perspective of a "Firm Deadline Real-Time Database System". Hereafter we will use "RTDB" to denote a firm deadline RTDB system, unless specified otherwise.

7.2 Distributed RTDB Systems

RTDB systems find applications in many areas like aerospace and military systems, computer integrated manufacturing, robotics, nuclear power plants, traffic control systems, stock market, telephone-switching systems, and network management [Son90, Ulu94a]. Many of these applications, especially in the areas of stock market, communication systems and military systems, are inherently *distributed* in nature. Unfortunately, till now, the major focus of the research in RTDB technology has been on *centralized* systems. Incorporating distributed data into the real-time database framework incurs the complexities described in Chapter 2 that are associated with transaction concurrency control and database recovery in conventional distributed database systems [BHG87, ÖV91]. While the issue of distributed real-time concurrency control has been considered to some extent(e.g. [SRL88, SK92, UB92]), comparatively little work has been done with regard to distributed real-time database recovery. In this part of the thesis, we investigate the performance implications of supporting transaction atomicity in a distributed RTDB system.

In RTDB systems, the resource scheduling and concurrency control mechanisms make use of transaction priorities in order to achieve high performance, that is, to maximize the number of transactions successfully committing before the expiry of their deadlines. The standard transaction commit protocols used in conventional database systems cannot be directly used in distributed RTDB systems because they suffer from the fact that they are not priority cognizant. Moreover, the performance considerations in conventional database systems and those in RTDB systems are different—while the average transaction throughput (or the average transaction response time) is the primary concern in conventional systems, RTDB systems strive to maximize the number of transactions that complete before their deadlines expire. For designing the commit protocols for a distributed RTDB system, we need to address two major questions: First, how do we adapt the standard commit protocols into the real-time domain? Second, how do the real-time variants of the commit protocols compare in their performance? We address these questions in this work. As mentioned earlier, we consider the "firm deadline" application framework, wherein transactions that miss their deadlines are considered to be worthless and are immediately "killed", that is, aborted and discarded from the system without being executed to completion. The performance goal in a firm deadline RTDB system is to minimize "deadline miss percent", that is, the steady-state percentage of transactions missing their deadlines. Our contributions in this regard are three-fold:

1. We first precisely define the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a firm-deadline distributed real-time database system. Subsequently, we customize the conventional commit protocols (that is, the commit protocols that were proposed in the context of distributed OLTP systems) for the firm-deadline RTDB environment.

2. Using a detailed open model of a distributed firm-deadline real-time database system, we profile the real-time performance of a representative suite of commit protocols that are customized for the real-time domain. To the best of our knowledge, these are the first quantitative results in this area.

3. We present and evaluate a new commit protocol, called **RT-OPT**, that is similar to OPT (the optimistic commit protocol proposed in the context of distributed OLTP systems) in its basic design but incorporates additional optimizations that cater to the special features of the real-time domain.

The experimental results show that data distribution has a significant influence on the real-time performance, and that the choice of commit protocol clearly affects the magnitude of this influence. Among the protocols evaluated, RT-OPT—the new real-time optimistic commit protocol—provides the best performance for a variety of workloads and system configurations.

CHAPTER

8 Real-Time Commit Processing

The conventional transaction commit protocols used in distributed OLTP systems cannot be directly used in the real-time environment, because they fail to take into consideration the real-time nature of the transactions. Therefore, the conventional commit protocols need to be modified to cater to the specific requirements of the real-time transactions. In this chapter, we discuss the issues involved in adapting the conventional commit protocols to the real-time environment. We also describe a new real-time commit protocol, called RT-OPT, that is similar to OPT (the optimistic commit protocol proposed in the context of distributed OLTP systems) in its basic design but incorporates additional optimizations that cater to the special features of the real-time domain.

8.1 Firm Deadline Distributed Transactions

As mentioned earlier, in a firm deadline real-time application framework, the transactions that miss their deadlines are considered to be worthless (and may even be harmful if executed to completion) and are immediately "killed" and discarded from the system without being executed to completion. Therefore, in an ideal distributed RTDB system, the master and all the cohorts of a transaction should agree to commit or abort before the transaction deadline expires. However, in practical distributed RTDB systems, it is impossible to provide such guarantees because of the possibility of communication link or site failures and arbitrary message delays [DLW89]. In addition, there are inherent complexities involved in achieving a global time-clock for the distributed system—synchronization of the clocks [Lam78] at various sites usually is a difficult task. Therefore, to avoid the inconsistencies in such cases, in a firm deadline RTDB system, a transaction is said to be **committed** if the master has reached the commit decision (that is, force-written the COMMIT log record to the disk) before the expiry of the deadline irrespective of whether the cohorts have received the decision by the deadline, that is, global time is viewed from the master's perspective. Note that once the master has reached the commit decision, this decision eventually will be conveyed to the cohorts even if the deadline has expired in the process. In other words, the master must reach the decision before the expiry of the deadline. If the master is unable to reach a commit decision before the deadline expires, it makes an abort decision, i.e., it immediately aborts the transaction and notifies the cohorts about the decision. Due to arbitrary message delays, a cohort may *not* receive the final decision before the deadline expires, even though the master had dispatched it before the expiry of the deadline. If a cohort eventually receives the final decision after the expiry of the deadline, all that happens is that access to data in prepared state is prevented even beyond the deadline until the decision is received by the cohort, and other transactions which would normally expect the data to be released by the deadline only experience a delay. Typically the master is responsible for returning the results of a transaction to the invoker of the transaction. From this discussion, it is clear that such results will have been generated before the deadline if a transaction commits.

8.2 Priority Inversion

The commit protocols described in Chapter 3 were designed for conventional database systems where transaction throughput or response time is the primary performance metric. In real-time database systems, however, satisfaction of transaction timing constraints is the primary goal. Therefore, the scheduling policies at the various resources (both physical and logical) in the system can be reasonably expected to be priority-driven with the priority assignment scheme being tuned to minimize the number of missed deadlines. The conventional commit protocols, however, do not take transaction priorities into account.

This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [SRL87]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable.

Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priority. At the CPU, for example, a scheduling policy such as Priority Preemptive Resume ensures the absence of priority inversion. For a non-preemptable resource such as the disk, although it is not possible to completely eliminate priority inversion due to physical device restrictions, at least it is possible to *bound* the inversion period by ensuring that the wait queue for the device is processed in priority order. Even at the network, it is easy to ensure that messages of high priority transactions are transmitted before those of other lower priority requesters. With respect to data access also, priority-based pre-emptive concurrency control algorithms such as 2PL-HP [AGM88] and OPT-WAIT [HCL90] have been developed.

Removing priority inversion in the commit protocol, however, is *not* fully feasible. This is because, once a cohort reaches the prepared state, it has to retain all its data locks until it receives the global decision from the master—this retention is fundamentally necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a prepared cohort of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a distributed real-time database system is *inherently* susceptible to priority inversion. More importantly, the priority inversion 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 inversion period is large, it may have a significant effect on performance.

8.3 Real-Time Commit Protocols

The conventional commit protocols suffer from two problems: (a) they are not deadline cognizant and hence are not capable of meeting the transaction deadline semantics as described in Section 8.1, and (b) they do not make use of the transaction priorities at the physical and logical resources, rendering them inefficient in the real-time domain. Even though the priority inversion arising due to prepared cohorts cannot be eliminated in these commit protocols, in order to be useful in the real-time environment, these commit protocols are modified to incorporate the following:

- The commit messages have the priorities of the transactions for which they are generated, so that the messages of a higher priority transaction can be given preference over the lower priority transactions at the CPU. In addition, processing of the data pages of a higher priority transaction at the CPU is given preference over the messages of lower priority transactions.

- The master ensures that a decision about the transaction is reached by the deadline.[1] If the deadline expires before the master had dispatched the PREPARE messages to the cohorts, it *unilaterally* decides to abort the transaction and sends the ABORT messages to the cohorts. On the other hand, if the deadline expires after dispatching the PREPARE messages, but before the master had reached a decision (i.e., all votes have not yet been received), the master makes an abort decision and the abort processing of the transaction is done in accordance with the protocol being used. For example, if PA is being used, the master writes the ABORT log record, sends the ABORT messages to all the cohorts, and forgets the transaction without waiting for the ACKs from the cohorts.

These modifications to the conventional commit protocols ensure that the deadline semantics of a transaction as described in Section 8.1 are met. Also, the real-time variant of the commit protocols exploit the priorities of the transactions while contending for the resources for message processing, etc. Hereafter we will not use the term "real-time variant" explicitly while referring to the real-time variants of the commit protocols. If the non-real-time (conventional) versions of the commit protocols are to be meant, we will explicitly state so.

[1]This may not be always possible in case of 3PC because in this protocol the master surrenders its right to make a *unilateral* decision after the precommit stage.

8.4 Real-Time Optimistic Commit Protocol

As discussed in Section 8.2, removing priority inversion in a commit protocol is *not* fully feasible because the lower priority cohorts that are in the prepared state cannot be preempted by the higher priority cohorts. This may have a significant effect on the system performance. To address this issue, we have designed a modified version of the 2PC protocol, wherein transactions requesting data items held by lower priority transactions in the prepared state are allowed to access this data. That is, lower priority prepared cohorts *lend* uncommitted data to the higher priority cohorts. In this context, two situations may arise:

Lender Receives Decision First: Here, the lending cohort (i.e., the prepared cohort) receives its global decision before the borrowing cohort has completed its local execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. If the global decision is to abort, then the lender is aborted in the normal fashion; in addition, the borrower is also aborted since it has utilized inconsistent data.

Borrower Completes Execution First: Here, the borrowing cohort completes its execution 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 processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until either the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, then the borrower is "taken off the shelf" and allowed to send its WORKDONE message. However, if the lender aborts, then the borrower is also aborted immediately since it has utilized inconsistent data. In the latter case (deadline of a borrower expires while waiting on the shelf), the borrower is killed in the normal manner.

In summary, the protocol allows transactions to access uncommitted data held by lower priority prepared transactions in the "optimistic" belief that this data will eventually be committed. Hereafter we will refer to this protocol as **RT-OPT**.

It was mentioned in Section 8.1 that if a cohort does not receive the decision from its master by the deadline, it will continue to hold the data in the prepared state beyond its deadline, thus preventing access to the data by other transactions. Therefore, a transaction that would normally expect the data to be released by the deadline will experience a delay. Using RT-OPT protocol solves this problem to a considerable extent as the data in the prepared state is made available to other transactions. If the cohort in the prepared state finally receives a commit decision (after the deadline has expired), only the transactions that were waiting on the shelf for the lender cohort to commit when they were aborted—perhaps because their deadlines expired—are affected (mostly these transactions otherwise also would have been killed, as their deadlines were close enough). Other borrower transactions are ignorant about the fact that the lender cohort actually received the decision only after its deadline because these transactions never went on the shelf. On the other hand, if the cohort in the prepared state finally receives an abort decision, all that happens is that the restarts of the transactions that had accessed uncommitted data is delayed, which otherwise would have occurred at the deadline expiry of the prepared cohort.

The primary motivation, as described earlier, for permitting access to uncommitted data was to reduce priority inversion. However, if we believe that lender transactions will typically commit, then this idea can be carried further to allowing *all* transactions, including low priority transactions, to borrow uncommitted data. This may further help in improving the real-time performance since the waiting period of transactions is reduced, and is therefore incorporated in RT-OPT.

Though the RT-OPT protocol is similar to the OPT protocol (described in the context of OLTP systems in Section 3.7) in its basic design, it has additional unique features that cater to the special features of the real-time environment.

8.4.1 Additional Features of RT-OPT

The following features have also been included in the RT-OPT protocol since we expect them to improve its real-time performance:

Active Abort : In the basic 2PC protocol, cohorts are *passive* in that they inform the master of their status only upon explicit request by the master, that is, if the cohort is aborted after it had dispatched the WORKDONE message, it will inform the master only on receiving the PREPARE message from the master. In a conventional distributed DBMS, after a cohort has finished its work and sent the WORKDONE message to the master, it cannot be aborted due to local data conflicts or serializability violations (assuming a lock-based concurrency control mechanism). However, in a distributed RTDB system, even such a cohort can be aborted due to higher priority conflicting transactions (provided the cohort is not already in the prepared state). Therefore, in a real-time situation, it may be better for an aborting cohort to immediately inform the master so that the abort at the other sites can be done earlier. Hence, cohorts in RT-OPT inform the master as soon as they decide to abort locally.

Silent Kill : For a transaction "kill", that is, an abort that occurs due to missing the deadline (before the master dispatched PREPARE messages), there is no need for the master to invoke the abort protocol since the cohorts of the transaction can independently realize the missing of the deadline (assuming global clock synchronization). Therefore, in RT-OPT, aborts due to deadline misses—before a cohort enters the prepared state—are done "silently" without requiring any communication between the master and the cohorts.

In addition, the PA and PC optimizations discussed earlier can be used in conjunction with RT-OPT (as was the case with OPT). Also, RT-OPT can be used with 3PC protocol to provide both the non-blocking feature of 3PC and the high performance of RT-OPT. We consider all these combinations in our experiments.

8.4.2 No Cascading Aborts in RT-OPT

Even though access to uncommitted data is allowed, aborts in RT-OPT do not cascade (as was the case with OPT in the context of OLTP systems, described in Chapter 3). The problem of cascading aborts in RT-OPT is alleviated due to the following reasons:

1. The lending cohort is in prepared state and cannot be aborted due to local data conflicts (we assume a lock-based concurrency control mechanism). Therefore, if the lending transaction finally decides to abort, it does so because of two reasons (in addition to consistency violations): (a) the master has already sent the PREPARE messages, but the deadline expires before it is able to reach a decision (Section 8.1), or (b) any of the siblings of the cohort in the prepared state votes NO. Note that when a cohort is aborted, it immediately sends an ABORT message to master (Active Abort policy), and the master, after receiving an ABORT message from a cohort never sends the PREPARE messages to other cohorts. Therefore, the situations in which a sibling of the prepared cohort could vote NO can be the following: (i) the ABORT message sent by the sibling cohort was in *transit* when master dispatched the PREPARE messages, or (ii) the PREPARE message sent by the master to the sibling cohort was in transit when the cohort was aborted. As the time during which a message is in transit is usually small compared to the transaction execution times, the above situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.

2. Even if the lender eventually does abort, it only results in the abort of the immediate borrower(s) and does not cascade beyond this point (since borrowers are not in the prepared state which is the only situation in which uncommitted data can be accessed).

In short, the abort chain for RT-OPT 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 abort chain for each borrower is bounded as described above).

8.5 Related Work on Real-Time Commit Protocols

The design of real-time commit protocols has been investigated earlier in [DLW89, SLKS92, Yoo94, GG94]. [GG94] addresses the recovery issues in the context of main-memory real-time database systems. [DLW89] describes a *timed 2PC* where the fate of a transaction (commit or abort) is guaranteed to be known to all cohorts of the transaction by the deadline, when there are no processor, communication, or clock faults. In case of faults, however, it is not possible to provide such guarantees, and an *exception state* is allowed which indicates the violation of the deadline. Now even if the master of a transaction is able to reach the decision by the deadline but it is not possible to convey the decision to all cohorts by the deadline, the transaction is killed (the master assumes the knowledge about the message delays etc.). Thus the primary concern in the paper is to ensure that all cohorts of a transaction reach the decision before the deadline expires, even at the cost of more transactions eventually being killed. In our work, we focus instead on improving the number of transactions that complete before their deadlines expire, which is of primary concern in the "firm deadline" application framework (adhering to the commitment semantics of the transactions as explained in Section 8.1). In addition, we do not assume any guarantees provided by the system for the services offered (e.g., messages, data access from the disk, etc.), unlike [DLW89] where such guarantees are fundamental to the design of the protocol.

[SLKS92, Yoo94] are based on a common theme of allowing individual sites to *unilaterally* commit—the idea is that unilateral commitment results in greater timeliness of actions. 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 timeliness, 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, LKS91b, LKS91a] 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*. Fourth, "real actions" [Gra81] such as firing a weapon or dispensing cash may not be compensatable at all [LKS91b].

From a performance viewpoint also, there are some difficulties: First, the execution of compensation transactions is itself an additional burden on the system. Second, it is not clear as to how the database system should schedule compensation transactions relative to normal transactions. Finally, no performance studies are available to evaluate the effectiveness of this approach.

Due to the above limitations of the compensation-based approach, we have in our research focused on improving the real-time performance of the *standard* mechanisms for maintaining distributed transaction atomicity *without* relaxing the transaction correctness criterion.

CHAPTER 9 RTDB Simulation Model

To evaluate the performance of the various real-time commit protocols described in the previous chapter, we developed a detailed simulation model based on an *open* queueing model of a distributed real-time database system. The simulation model is similar in many aspects to the model described in Chapter 4, and is based on a loose combination of the distributed database model described in [CL88] and the real-time database model described in [HCL92]. The model consists of a database that is distributed, in a non-replicated manner, over N sites connected by a network. Each site in the model 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 communications network.

In the following sections, we describe various components of the simulation model. Subsequently, we describe the execution pattern of a typical transaction. A summary of the parameters used in the model are given in Table 9.1.

Table 9.1: Simulation Model Parameters

$NumSites$	Number of sites in the database
$DBSize$	Number of pages in the database
$ArrivalRate$	Transaction arrival rate / site
$SlackFactor$	Slack Factor in Deadline formula
$TransType$	Transaction Type (Sequential or Parallel)
$DistDegree$	Degree of Distribution
$CohortSize$	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
$MsgCPU$	Message send / receive time
$BufHit$	Probability of finding a requested page in the buffer

9.1 Database and Workload Model

The database is modeled as a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. At each site, transactions arrive in an independent Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline. The deadline is assigned using the formula $D_T = A_T + SlackFactor * E_T$, where D_T, A_T and E_T are the deadline, arrival time and execution time, respectively, of transaction T, while $SlackFactor$ is a constant that provides control over the tightness/slackness of transaction deadlines. The execution time is the total service time at the resources that the transaction requires for its execution.[1] Note that although the workload generator knows E_T, the system does *not* have any knowledge about E_T.

Each transaction in the workload has the "single master, multiple cohort" structure described in Section 3.1. 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

[1]Since the resource time is a function of the number of messages and the number of forced-writes, which differ from one commit protocol to another, we compute the execution time assuming execution in a *centralized* system.

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 randomly from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$. A transaction that is aborted due to a data conflict is restarted immediately and makes the same data accesses as its original incarnation. Note that, unlike OLTP systems, no delay is required before restarting the transaction—the reasons for introducing the delay in OLTP systems are irrelevant in RTDB systems.

9.2 System Model

The physical resources at each site consist of $NumCPUs$ processors, $NumDataDisks$ data disks, and $NumLogDisks$ log disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. A data page request can be satisfied by the buffer manager, or the page may have to be accessed from the disk. $BufHit$ parameter gives the probability of finding a page 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 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. We do not make any distinction, however, between these different types of requests and only ensure that all requests are served in priority order. The CPU overheads for message transfers are captured by the $MsgCPU$ parameter.

9.3 Priority Assignment

As mentioned earlier, transactions in a RTDB system are typically assigned priorities in order to minimize the number of missed deadlines. In our model, all the cohorts of a transaction inherit the parent transaction's priority. Further, this priority, which is assigned at arrival time, is maintained throughout the course of the transaction's existence in the system (including the commit processing stage, if any). The processing of messages sent on behalf of a transaction also occurs at the priority of the transaction. The transaction priority assignment policy used is the widely-used *Earliest Deadline* [LL73]—transactions with earlier deadlines have higher priority than transactions with later deadlines.

9.4 Concurrency Control

The 2PL High Priority scheme [AGM88] is used for concurrency control of conflicting transactions. A concurrency control request from a high priority cohort for a data item held by a low priority cohort is satisfied by aborting the low priority cohort. If a low priority cohort requests a data item held by a high priority cohort, it is made to wait until the high priority cohort releases its lock from the data item (or allows optimistic access, as in RT-OPT). If a data item is locked by some cohort in the read mode, a cohort's request for sharing the lock is granted only if no cohort with a priority higher than that of the requesting cohort is waiting in the lock queue to access the data in an exclusive (write) mode. A cohort in the prepared state cannot be aborted due to concurrency reasons. On receipt of the PREPARE message from the master, a cohort releases all its read locks but retains its update locks until it receives and implements the global decision from the master. Due to the fact that 2PL-HP protocol is used to resolve the conflict between transactions, and that the transactions are assigned unique priorities at the arrival time, there is no possibility of the transactions forming a deadlock.

9.5 Execution Model

A transaction, at its arrival time, 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 deadline of the transaction is assigned based on the formula described in Section 9.1. The parameter $TransType$ specifies whether the transaction will execute in a sequential or parallel fashion. 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. 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. We do not explicitly model concurrency control cost parameters in our model since we assume that the overheads of performing concurrency control are small compared to data processing times. 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. 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. If the transaction's deadline expires either before this point, or before the master has written the global decision log record, the transaction is killed (the precise semantics of firm deadlines in a distributed environment are defined in Section 8.1). If the master has written the commit decision log record before the expiry of the deadline, the transaction is said to be committed.

9.6 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 execution log records (i.e., WAL) are not modeled since we do not expect them to affect the relative performance behavior of the commit protocols discussed in our study.

C H A P T E R

10

RTDB Experiments and Results

We conducted an extensive set of experiments to evaluate the performance of the various real-time commit protocols described in Chapter 8. A detailed model of a firm deadline distributed RTDB system as described in the previous chapter was used for this purpose. The simulator described in Chapter 5 was modified to suit the requirements of a firm deadline distributed RTDB system. In this chapter, we present the results of these experiments and discuss the performance implications of supporting transaction atomicity in a real-time environment.

10.1 Performance Metric

The performance metric of our experiments is $MissPercent$, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. MissPercent values in the range of 0 to 20 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 20 percent to 100 percent represent system performance under "heavy" loads. A long-term operating region where the miss percentage is high is obviously unrealistic for a viable distributed RTDB system. Exercising the system to high miss levels (as in our experiments), however, provides valuable information on the response of the algorithms to brief periods of stress loading. All the MissPercent graphs in this chapter 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 20000 transactions were processed by the system. Only statistically significant differences are discussed here.

We also discuss *transaction throughput*, that is, the number of transactions per second completing successfully within their deadlines. Unlike OLTP systems, in RTDB systems all input transactions may not complete successfully, some of them being killed as their deadlines expire. Even though the transaction throughput in an RTDB system does not have much significance from a performance perspective, measuring transaction throughput gives an indication of the optimal arrival rate the system can sustain without severe degradation in the performance, that is, the arrival rate beyond which system starts thrashing resulting in a decrease in the throughput. Knowing the optimal value of the arrival rate, admission control mechanisms can be applied to control the arrival rate to this value.

The simulator was instrumented to generate a host of other statistical information, including resource utilizations, number of transaction restarts, number of force log writes per transaction, page conflict probability, borrow ratio (for RT-OPT), etc. These secondary measures help to explain the MissPercent behavior of the commit protocols under various workloads and system conditions.

Comparative Protocols

As was the case for performance experiments in OLTP systems, we simulate the performance behavior for two additional scenarios, that is, **CENT** and **DPCC**. While CENT scenario represents an equivalent *centralized system*, DPCC scenario represents an artificial system where the data processing is distributed but the commit processing is centralized. Modeling these scenarios help to isolate the effect of data distribution and commit processing on MissPercent performance. The detailed descriptions of these scenarios are given in Section 5.2.

Table 10.1: Baseline Parameter Values

NumSites	8	*NumCPUs*	2
DBSize	2400 pages	*NumDataDisks*	3
ArrivalRate	0.5 - 10	*NumLogDisks*	1
SlackFactor	4.0	*PageCPU*	5 ms
TransType	Sequential	*PageDisk*	20 ms
DistDegree	3	*MsgCPU*	5 ms
CohortSize	6 pages	*BufHit*	0.1
UpdateProb	1.0	–	–

10.2 Experiment 1: Baseline Experiment

We began the performance evaluation of the real-time commit protocols by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. The values of the various parameters for the baseline experiment are listed in Table 10.1. These values 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, without having to generate very high transaction arrival rates.

For the baseline experiment, Figures 10.1a and 10.1b show the MissPercent behavior under normal load and heavy load conditions, respectively. In these graphs, we first observe that there is considerable difference between centralized performance (CENT) and the performance of the standard commit protocols throughout the loading range. For example, at an arrival rate of 2 transactions per second at each site, the centralized system misses less than 5 percent deadlines whereas 2PC and 3PC miss in excess of 25 percent of the deadlines. This difference highlights the extent to which a conventional implementation of distributed processing can affect real-time performance.

We see from Figure 10.1c that the peak throughput is achieved at an arrival rate of approximately 2 transactions per second for 2PC and 3PC, while for CENT and DPCC this value is approximately 3 transactions per second. These values of arrival rates correspond to transaction MissPercent range of 20 percent to 30 percent, for all these protocols. Thus, we observe that the systems is being utilized to the maximum as long as the MissPercent value is in the range of 20 percent to 30 percent, beyond which it starts thrashing.

Moving on to the relative performance of 2PC and 3PC, we observe from Figures 10.1a and 10.1b that there is a noticeable but not large difference between their performance at normal loads. The difference arises from the additional message and logging overheads involved in 3PC. As shown in Figure 10.1d, the overheads of force-writes for 3PC are significantly higher than that of 2PC. Under heavy loads, however, the performance of 2PC and 3PC is virtually identical. This is explained as follows: Although their commit processing is different, the *abort* processing of 3PC is identical to that of 2PC. Therefore, under heavy loads, when a large fraction of the transactions wind up being killed (aborted) the performance of both protocols is essentially the same. Since their performance difference is not really large for normal loads also, it means that, in the real-time domain, the price paid during normal processing to purchase the nonblocking functionality is comparatively modest.

Shifting our focus to the PA and PC variants of the 2PC protocol, we find that their performance is only marginally different to that of 2PC. Thus, we see that as was the case in conventional OLTP environment, here too, these optimizations fail to provide tangible performance benefits. We would have, however, expected PC to perform at least slightly better than 2PC at normal loads, because the utilization of the resources is reasonably high at normal loads and most of the transactions are finally committed—the conditions favorable to PC. Similarly, we would have expected PA to perform better at high loads, because most of the transactions are aborted. The reason that PA and PC do not show much improvement in the performance, is that the performance in an RTDB system is measured in *boolean* terms of meeting or missing the deadline. So, although PA and PC do reduce overheads under abort and commit conditions, respectively, all that happens is that the resources released by this reduction only allow executing transactions to execute further before being restarted or killed but is not sufficient to result in many more *completions*. This was confirmed by measuring the number of force-writes and the number of acknowledgements, on a per transaction basis, shown in Figures 10.1d

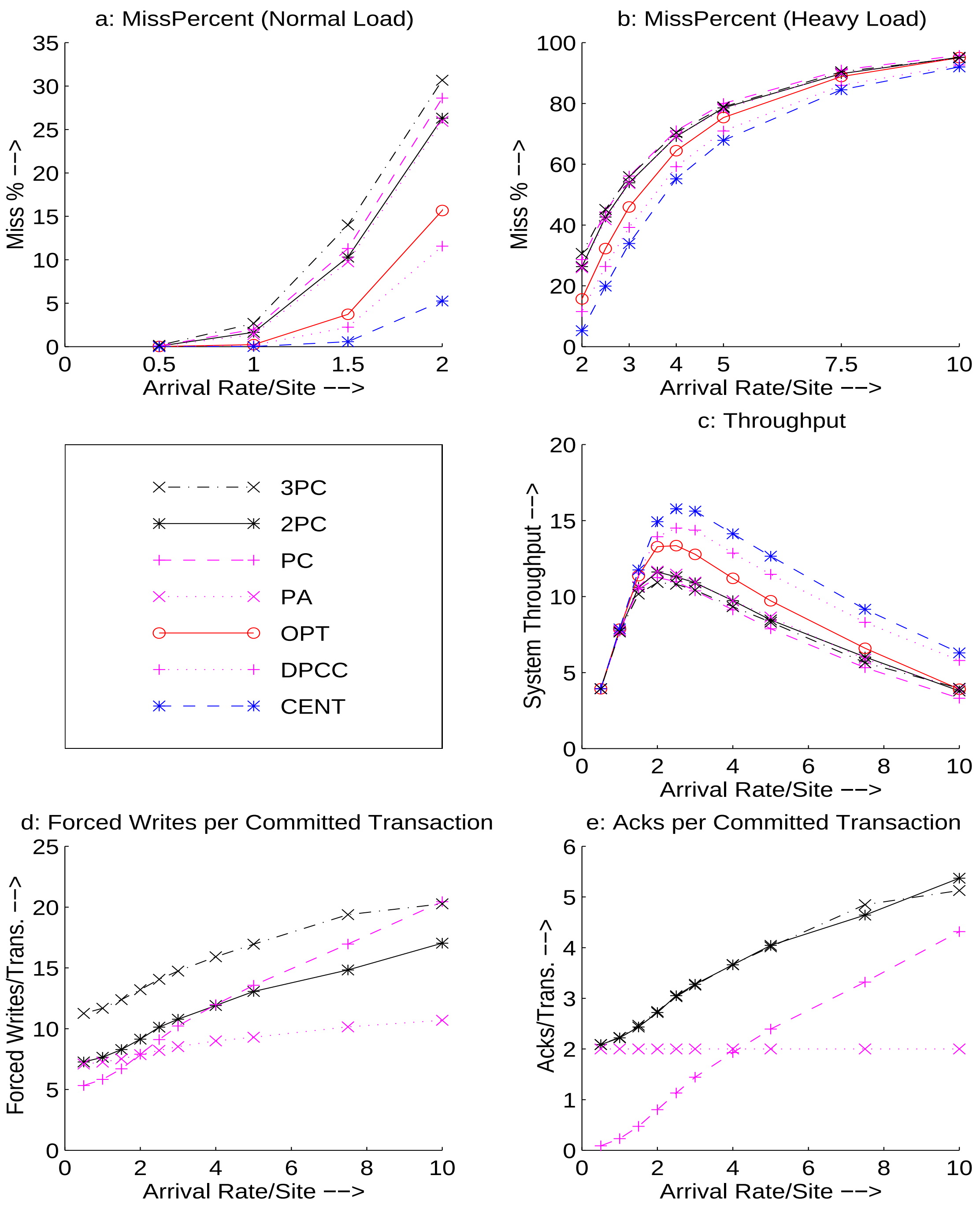


Figure 10.1: (a–e) Baseline Experiment (Sequential, RC+DC)

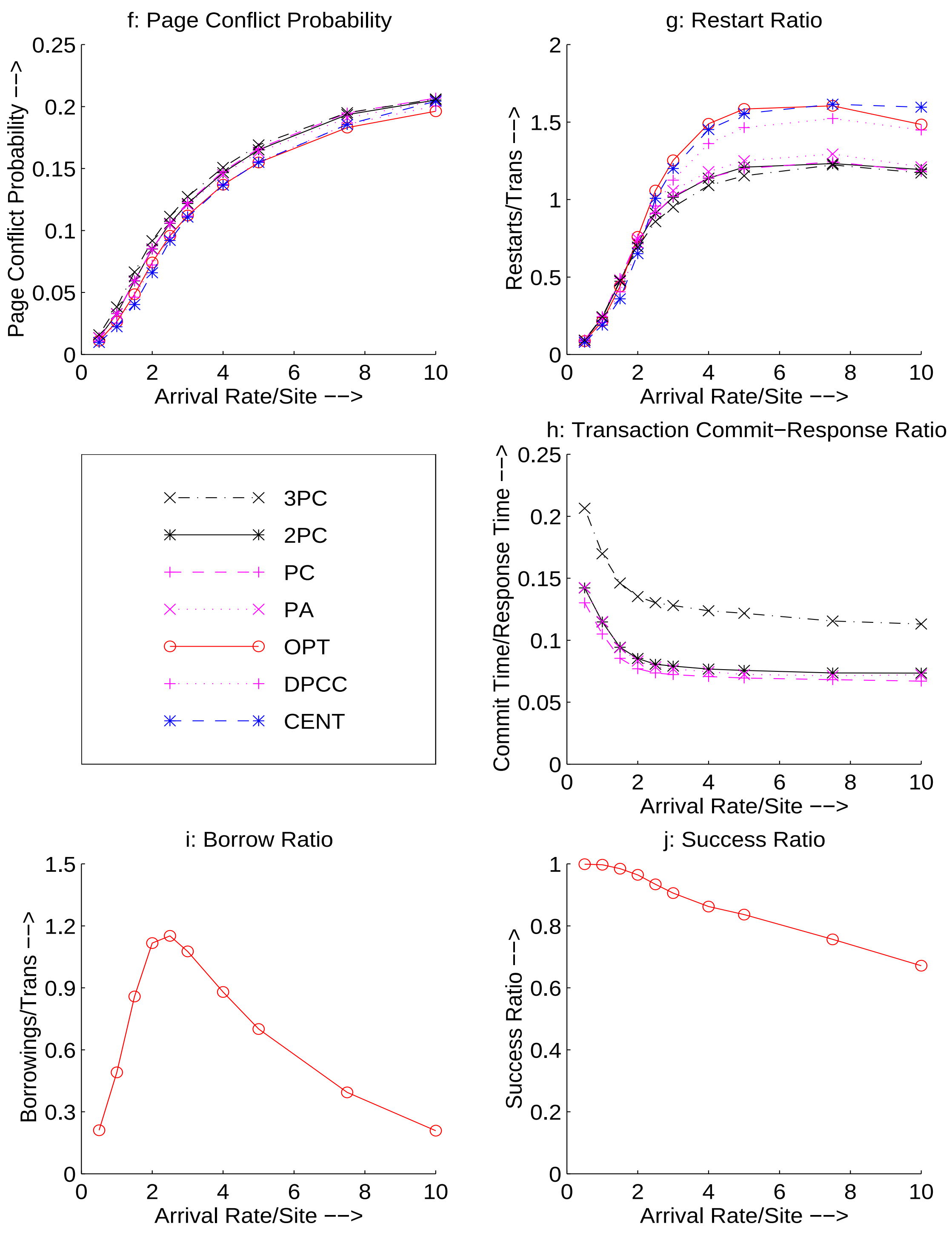


Figure 10.1: (f–j) Baseline Experiment (Sequential, RC+DC)

and 10.1e. In these figures we see that PA has significantly lower overheads at heavy loads (when aborts are more) and PC has significantly lower overheads at normal loads (when commits are more). Moreover, while PA always does slightly better than 2PC, PC actually does slightly worse than 2PC since PC has higher overheads than 2PC for aborts.

Finally, turning our attention to the new protocol, RT-OPT, we observe that its performance is considerably better than that of the standard algorithms over most of the loading range and especially so at normal loads. An analysis of its improvement showed that it arises primarily from the optimistic access of uncommitted data and from the active abort policy. The silent kill optimization (not sending abort messages for aborts arising out of deadline misses), however, gives only a very minor improvement in performance. At low loads, this is because the number of deadline misses are few and the optimization does not come into play; at high loads, the optimization's effect is like that of PA and PC for the standard 2PC protocol—although there is a significant reduction in the number of messages, the resources released by this reduction only allow transactions to proceed further before being restarted but does not result in many more *completions*. This was confirmed by measuring the number of pages that were processed at the CPU—it was significantly more when silent kill was included.

RT-OPT helps improve the performance of the system due to the following reason: By allowing access to the uncommitted data of the prepared cohorts, it reduces the data contention in the system. This is clear from Figure 10.1f where the page conflict probability (the probability that a page being requested by a cohort is already held in a conflicting mode by another cohort) for RT-OPT is less than that for all other commit protocols. In fact the data contention in the system when using RT-OPT is similar to that in DPCC.

As part of this experiment, we wanted to quantify the degree to which the RT-OPT protocol's optimism about accessing uncommitted data was well-founded – that is, is RT-OPT safe or foolhardy? To evaluate this, we measured the "success ratio", that is, the fraction of times that a borrowing was successful in that the lender committed after loaning the data. This statistic is shown in Figure 10.1j and clearly shows that under normal loads, optimism is the right choice since the success ratio is almost one. Under heavy loads, however, there is a decrease in the success ratio. The lower success ratio results in more transactions being aborted. This is evident from Figure 10.1g which shows higher restart ratio (the number of times a transaction restarts before it is either committed or killed) for RT-OPT at high arrival rates. The reason for this is that transactions reach their commit phase only close to their deadlines and in this situation, a lending transaction may often abort due to missing its deadline.

Figure 10.1h shows the transaction commit-response ratio (the ratio of the average transaction commit time to the average transaction response time measured for the committing transactions), which shows an initial decrease but reaches a plateau for high transactions. The initial decrease is due to the fact that initially the response time of a transaction increases as the load in the system increases. This is where the RT-OPT helps in reducing the response time and thus making more transactions meet their deadlines. At high loads, however, the response time has reached its maximum possible value—defined by transaction's deadline, beyond which it is killed—and hence the increase in load increases the number of transactions missing their deadlines rather than affecting the average transaction response time. Thus, the chances are that a transaction that somehow manages to reach the commit phase will be very close to its deadline.

These results indicate that under heavy loads, the optimistic policy should be modified such that transactions borrow only from "healthy" lenders, that is, lenders who still have considerable time to their deadline—this issue is further discussed in Chapter 11.

Figure 10.1i shows the "borrow ratio" for RT-OPT, that is, the average number of data items borrowed per transaction (measured as the ratio of total number of borrowed data items to the total input transactions in the system). We see from this figure that the borrow ratio increases initially as the arrival rate is increased, but then starts decreasing. The reason for this is that initially the number of transaction in the system increases, thus providing more opportunities for optimistic feature to come into play, but at higher arrival rates there are very few transactions that are able to make it to the commit processing phase, and thus there are very few lenders available. Therefore, at very high loads, there are very few opportunities for the optimistic feature to come into play. This also controls to some extent the impact of unsuccessful lending at high arrival rates.

Another interesting point to note is the following: In Figures 10.1a and 10.1b the difference between the CENT and DPCC curves shows the effect of distributed *data* processing whereas the difference between the commit protocol curves and the DPCC curve shows the effect of distributed *commit* processing. We see in these figures that the effect of distributed commit processing is considerably more than that of distributed data processing. These results clearly highlight the necessity for designing high-performance real-time commit protocols.

10.3 Experiment 2: Pure Data Contention

The goal of our next experiment was to isolate the influence of *data contention* (DC) on the real-time performance of the commit protocols. Therefore, for this experiment, the physical resources (CPUs and disks) were made "infinite", that is, there is no queueing for the these resources. The other parameter values are the same as those used in Experiment 1. The MissPercent performance results for this system configuration are presented in Figures 10.2a and 10.2b, and the supporting statistics are shown in Figures 10.2c through 10.2j.

We observe in these figures that the relative performance of the various protocols remains qualitatively similar to that seen under resource-cum-data contention (RC+DC) in the previous experiment. CENT still shows the best performance and the performance of DPCC is slightly worse than that of CENT. The performance of the standard protocols relative to the baselines is, however, markedly worse than before. This is because in the previous experiment under RC+DC, 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 the MissPercent. Under pure DC, however, the performance is limited only by data contention. Therefore, although the MissPercent under pure DC is lower than that under RC+DC, the effect of the commit phase on MissPercent performance is felt to a greater extent as the data held in the commit phase is not accessible to other transactions, thus increasing the data contention. We also

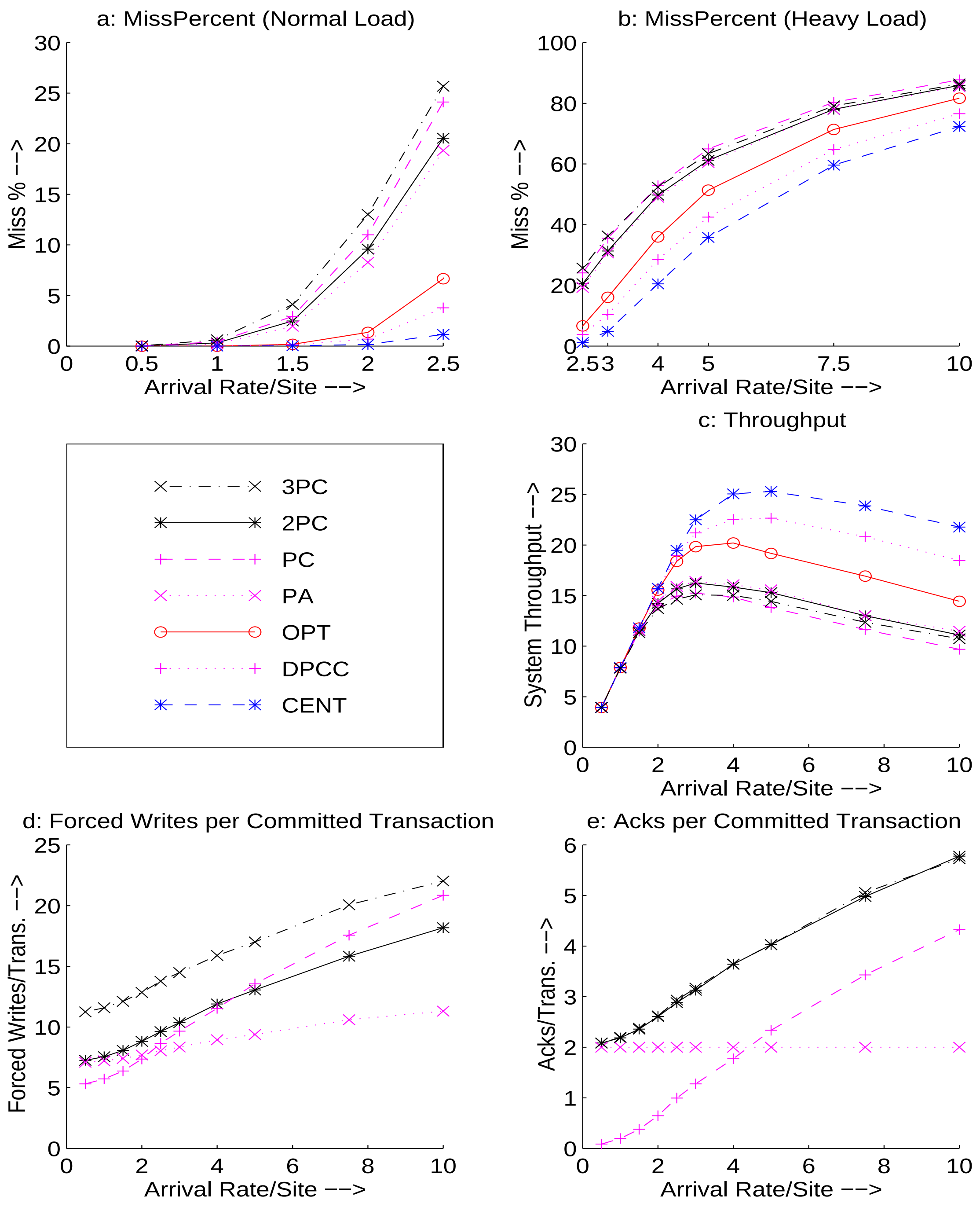


Figure 10.2: (a–e) Pure Data Contention (Sequential)

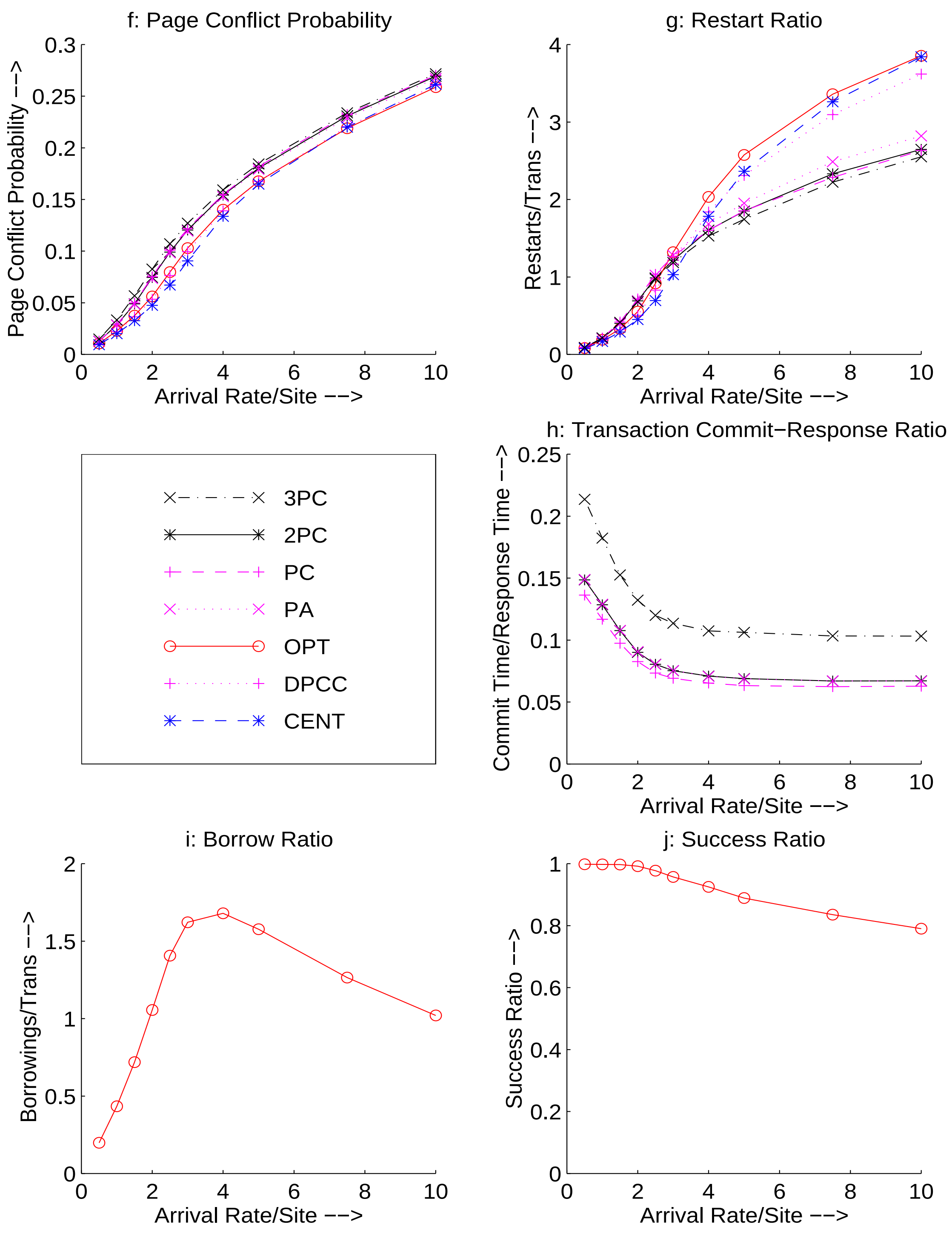


Figure 10.2: (f–j) Pure Data Contention (Sequential)

observe the performance of RT-OPT in this experiment is much superior to standard commit protocols over the entire loading range as compared to the previous experiment. This is for the same reason explained above. RT-OPT allows access to the data held by the transactions in the commit phase, thus reducing the data contention in the system. Moreover, its success ratio does not go below 75 percent even at the highest loading levels (Figure 10.2i).

An important observation here is that while resource contention can be reduced by purchasing more and/or faster resources, there exists no equally simple mechanism to reduce data contention. It should be noted here that while abundant resources may not be expected in conventional database systems, that may not be the case with RTDB systems since many real-time systems are designed to handle transient heavy loading.

10.4 Experiment 3: Parallel Transaction Execution

In the previous experiments, we considered *sequential* transaction execution. In this experiment, we consider *parallel* transaction execution in order to investigate the effect of parallel execution of cohorts on the relative real-time performance of the various commit protocols. We conducted this experiment for both RC+DC and pure DC scenarios. The parameter values under RC+DC and pure DC scenarios were the same as those used in Experiment 1 and Experiment 2, respectively—except $TransType$, which of course is "parallel" for this experiment. Figures 10.3a–10.3j present the results of this experiment for RC+DC scenario while Figures 10.4a–10.4j present the results for pure DC scenario.

From these figures we see that the observations made in the previous experiments for sequential transactions, to a large extent, hold true for the parallel transactions also. In particular, the effect of distributed commit processing on the MissPercent performance remains considerably more than that of distributed data processing. The performance of PA is similar to that of 2PC, while the performance of PC is slightly worse than that of 2PC. The performance of 3PC is worse than that of 2PC, and RT-OPT continues to provide better performance as compared to 2PC.

For the RC+DC case (Figures 10.3a–10.3j), we observe that the performance differ-

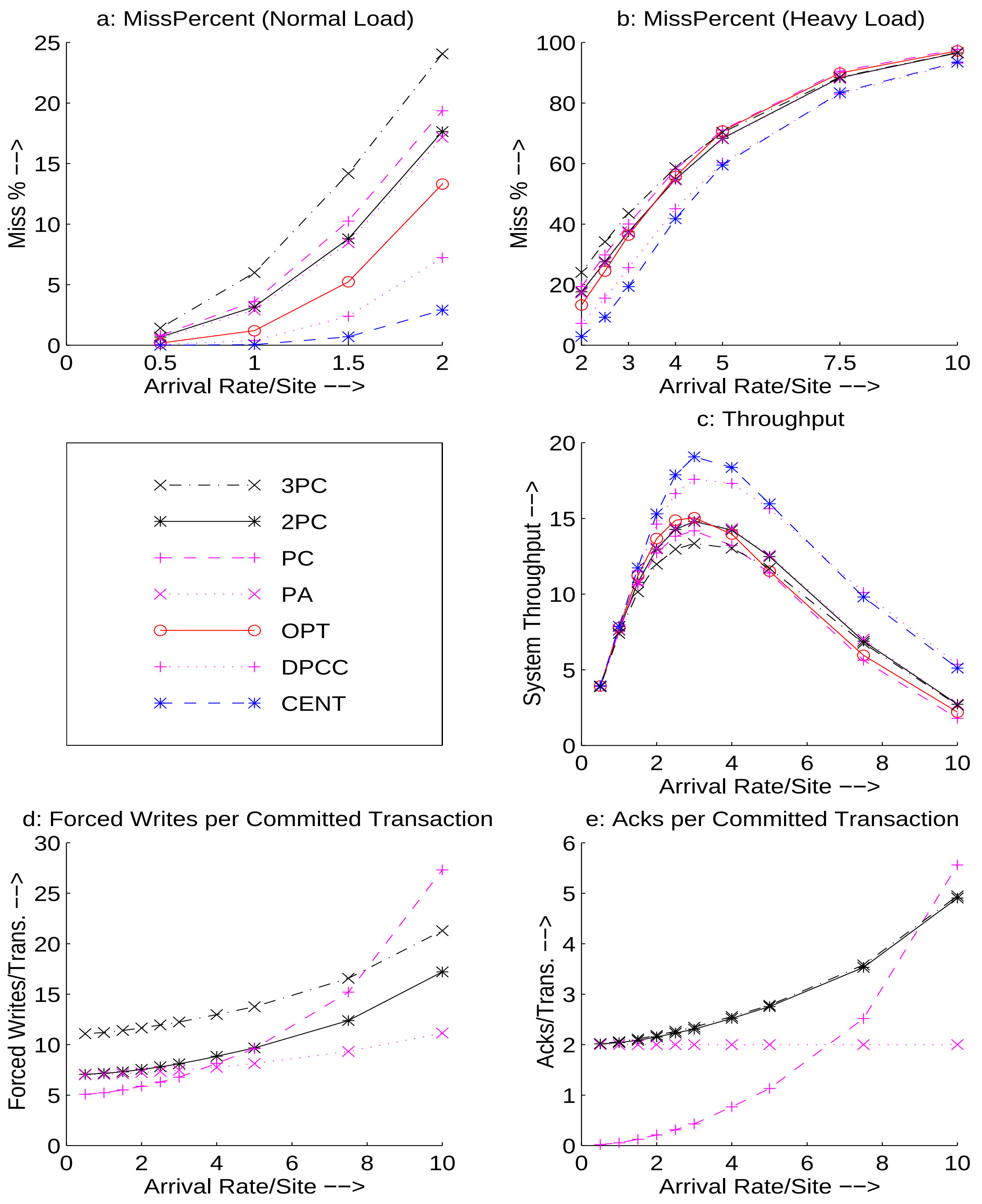


Figure 10.3: (a–e) Parallel Transactions (RC+DC)

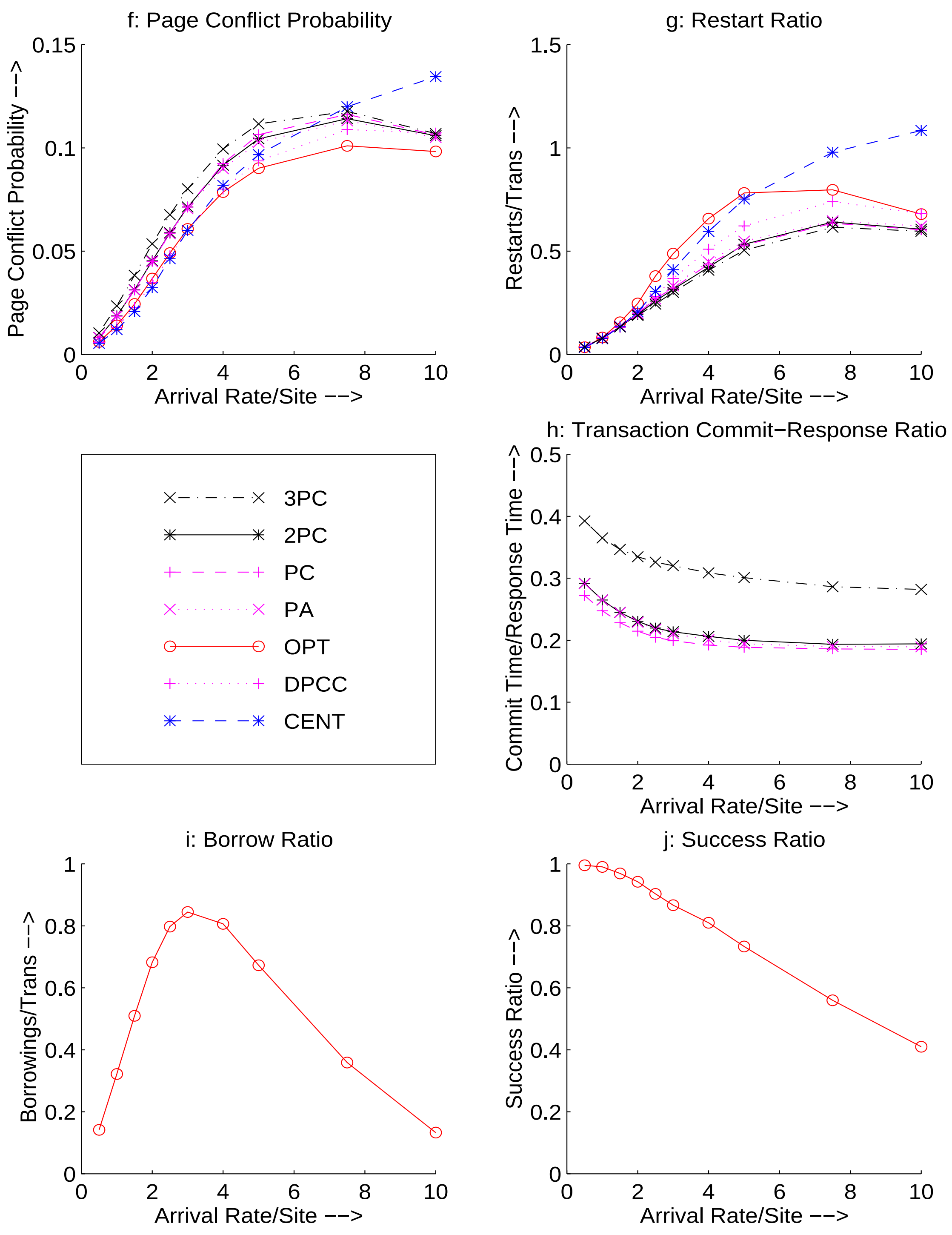


Figure 10.3: (f–j) Parallel Transactions (RC+DC)

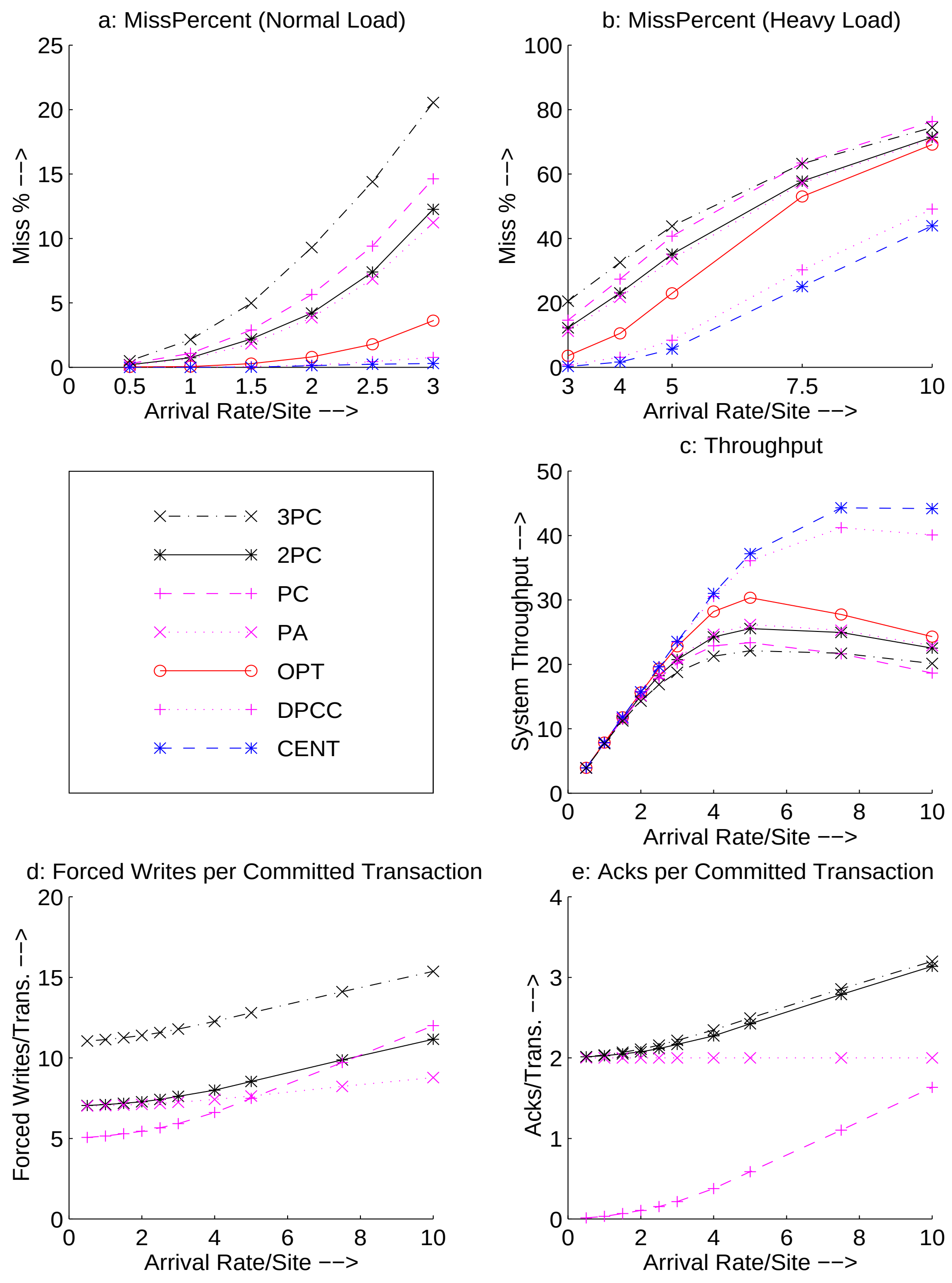


Figure 10.4: (a–e) Parallel Transactions (Pure DC)

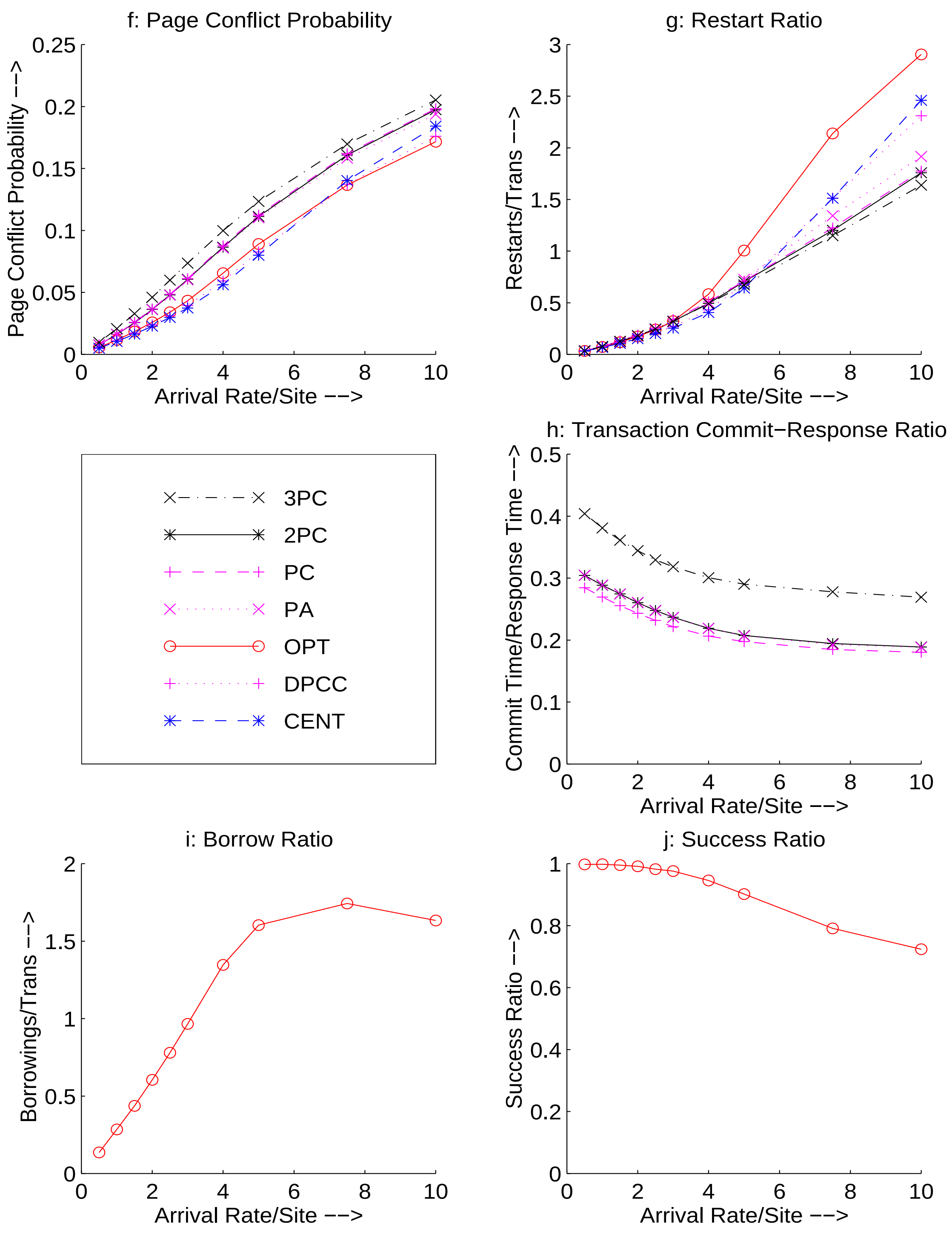


Figure 10.4: (f–j) Parallel Transactions (Pure DC)

ences between 2PC and 3PC have increased as compared to those seen for the sequential transaction execution experiments. The reason for this is that in parallel transaction execution, the cohorts execute in parallel, thus reducing the transaction response time, but the time required for the commit processing remains the same. This is also clear from Figure 10.3h where the commit-response ratio for 2PC and 3PC is significantly higher than for the corresponding sequential transaction execution experiment. Therefore, even though the overall MissPercent is less in parallel execution, the relative performance differences between CENT and 2PC, and between CENT and 3PC are more than those for the sequential transaction execution. As the overheads of 3PC are more than that of 2PC, the effects of 3PC are felt to a greater extent.

Another observation from these figures is that the effect of RT-OPT on the MissPercent performance is reduced to some extent as compared to the sequential transaction execution. This is explained as follows: In sequential transaction execution, the benefits of RT-OPT are due to the optimistic feature and due to the active abort policy. In parallel transaction execution, however, the active abort policy does not improve the performance. This is because even if a cohort notifies the master about the abort, the chances are that the master has already sent the PREPARE messages to all the cohorts before receiving such a message. Moreover, because the cohorts execute in parallel, there are much less chances that a cohort may be aborted after sending the WORKDONE message, but before receiving the PREPARE message—that is where the active abort policy helps in the sequential transaction execution. Therefore, in the parallel case, there are no benefits due to the active abort policy, and hence the performance of RT-OPT is less impressive as compared to the sequential case.

Under pure DC scenario, however, the performance of RT-OPT is again comparable to that seen for the corresponding sequential transaction execution experiment. The reason for this is that under pure DC, as there is no resource contention, RT-OPT reduces the data contention significantly as is clear from Figure 10.4f. In this case, the borrow ratio and the success ratio for RT-OPT are significantly high (Figures 10.4i and 10.4j), thus making RT-OPT perform considerably better than the standard commit protocols.

10.5 Experiment 4: Fast Network Interface

In all of the previous experiments, the cost for sending and receiving messages modeled a system with a slow network interface ($MsgCpu = 5\,ms$). We conducted another experiment wherein the network interface was faster by a factor of *five*, that is, where $MsgCpu = 1\,ms$. The results of this experiment are shown in Figures 10.5a–10.5d for sequential transactions under RC+DC and pure DC. Figures 10.5e–10.5h present these results for parallel transactions.[1]

In these figures, we see that all the curves become closer to CENT, and in fact, under pure DC scenario, CENT and DPCC are virtually indistinguishable for both sequential and parallel transactions. 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. The relative performance of the protocols remains similar to what was observed in the previous experiments. Note also that the RT-OPT protocol now provides a performance that is close to that of DPCC, that is, close to the best commit processing performance that could be obtained in a distributed RTDB system, especially in the pure DC scenario.

This experiment shows that adopting the RT-OPT principle can be expected to be of value even in the gigabit networks of the future. While the technological improvements may help to reduce the resource contention and provide faster processing capabilities at the network interface, there exists no equally simple mechanism to reduce *data contention*.

10.6 Experiment 5: 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 on *six* sites. The $CohortSize$ in this experiment was set to 3 pages as compared to the $CohortSize$ of 6 pages used in the previous experi-

[1]The default parameter values under 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. In this and the following experiments, both sequential and parallel transactions are considered.

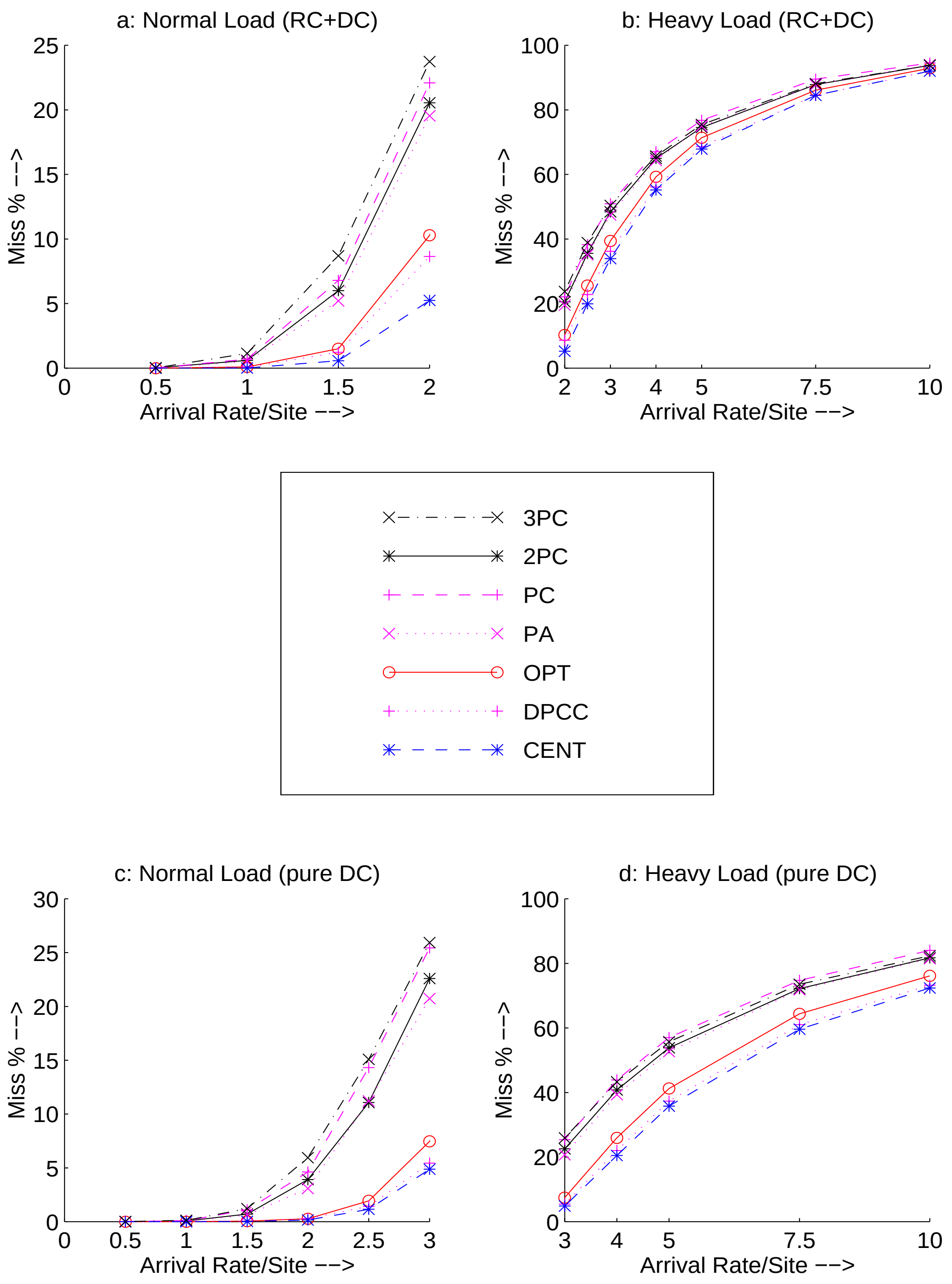


Figure 10.5: (a–d) Fast Network (MissPercent, Sequential Transactions)

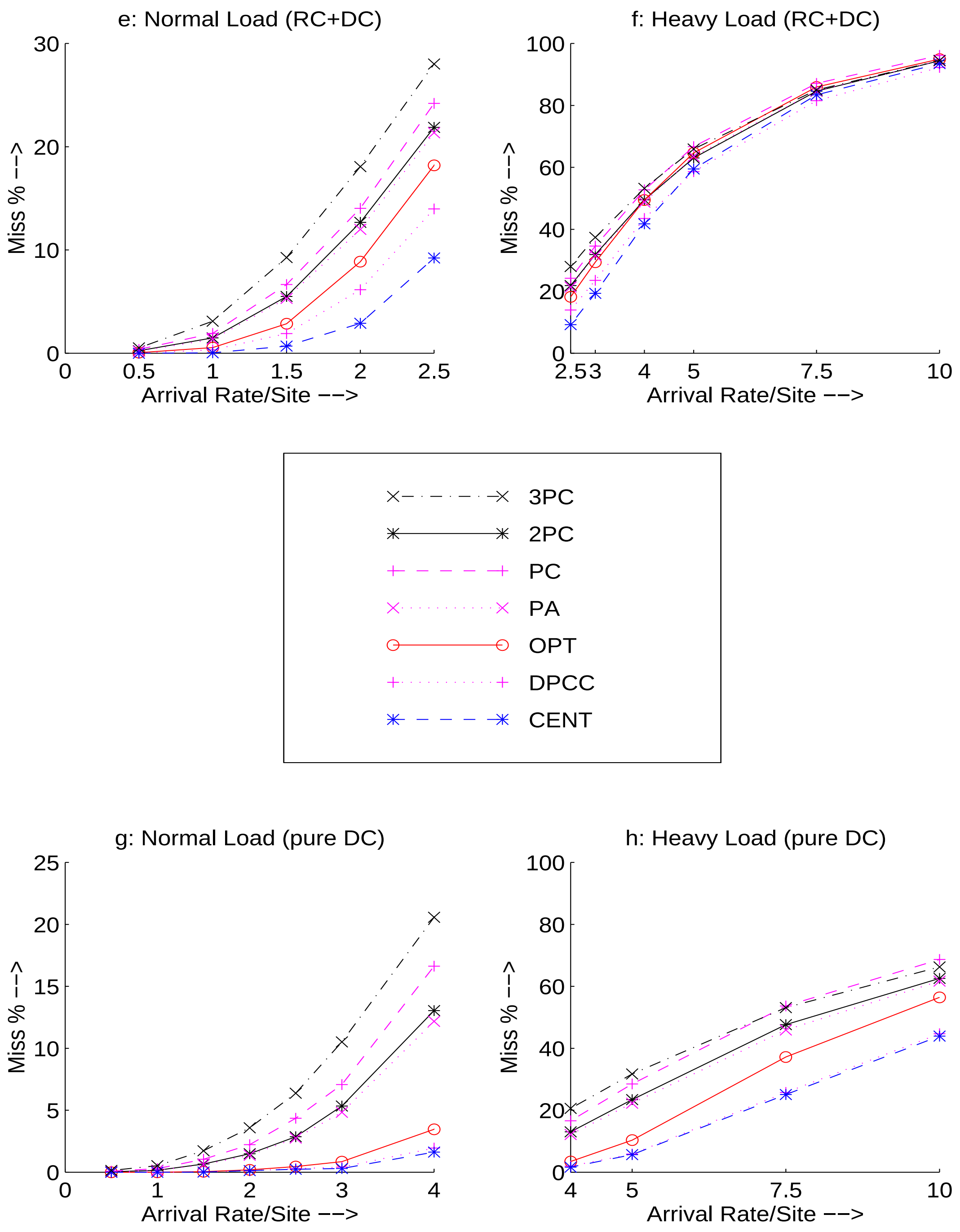


Figure 10.5: (e–h) Fast Network (MissPercent, Parallel Transactions)

ments. This was done in order to keep the average transaction length in this experiment equal to that of the previous experiments. In addition, a higher value of $SlackFactor = 6$ was used to cater to the increase in response times caused the increased distribution. The results of this experiment for both RC+DC and pure DC scenarios are shown in Figures 10.6a–10.6d for the sequential transactions, and in Figures 10.6e–10.6h for the parallel transactions.

In these figures, we observe that increased distribution results in an increase in the magnitudes of the performance differences among the commit protocols. This is to be expected since the commit processing overheads are larger in this experiment. Also, the performance of CENT and DPCC is improved significantly because the heightened degree of distribution results in more efficient utilization of the physical resources due to load balancing (as explained in Section 5.2 for OLTP systems). The increase in the message overheads of DPCC is not sufficient to cause a severe degradation in the performance of DPCC. The *relative* performance of the protocols, however, remains qualitatively the same with RT-OPT continuing to perform better than the standard protocols.

10.7 Experiment 6: Reduced Update Probability

In the previous experiments, each page accessed by the transaction was updated. We conducted another experiment where the value of *UpdateProb* was set to 0.5, that is, each page accessed by the transaction was updated with 0.5 probability. Reduced page update probability decreases the data contention in the system because the read locks are shared. In addition, the read locks are released when a cohort enters the prepared state.

The results of this experiment are shown in Figures 10.7a–10.7d for sequential and parallel transactions under RC+DC and pure DC conditions. These results show almost similar behavior as observed in corresponding experiments for fully update transactions discussed earlier. The performance of all protocols has increased, because now there is less data contention in the system. RT-OPT still performs better than 2PC, even though the gain in its performance is reduced as compared to previous experiments. The relative performance of the protocols remain similar to observed in previous experiments.

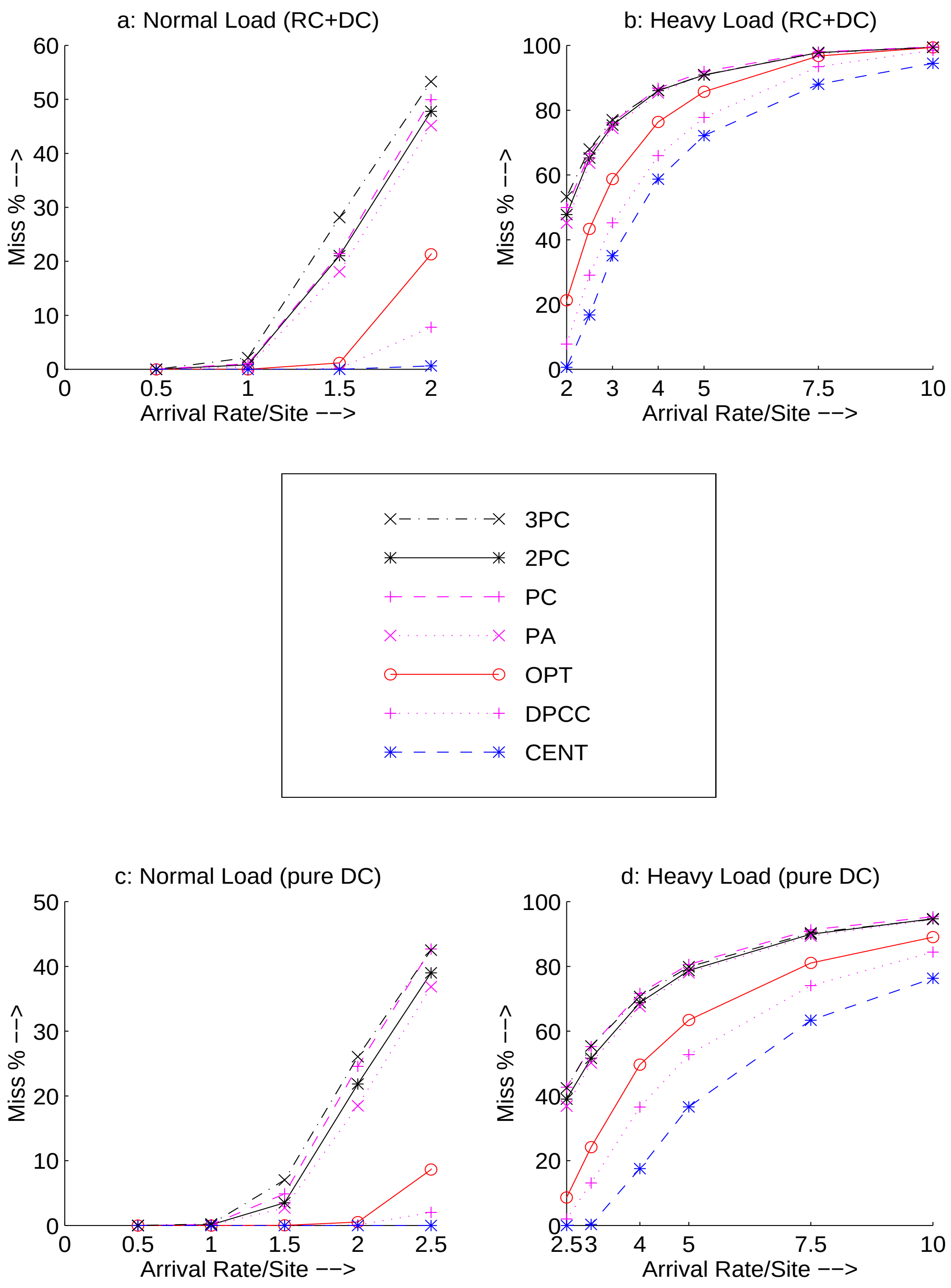


Figure 10.6: (a–d) Higher Distribution (MissPercent, Sequential Transactions)

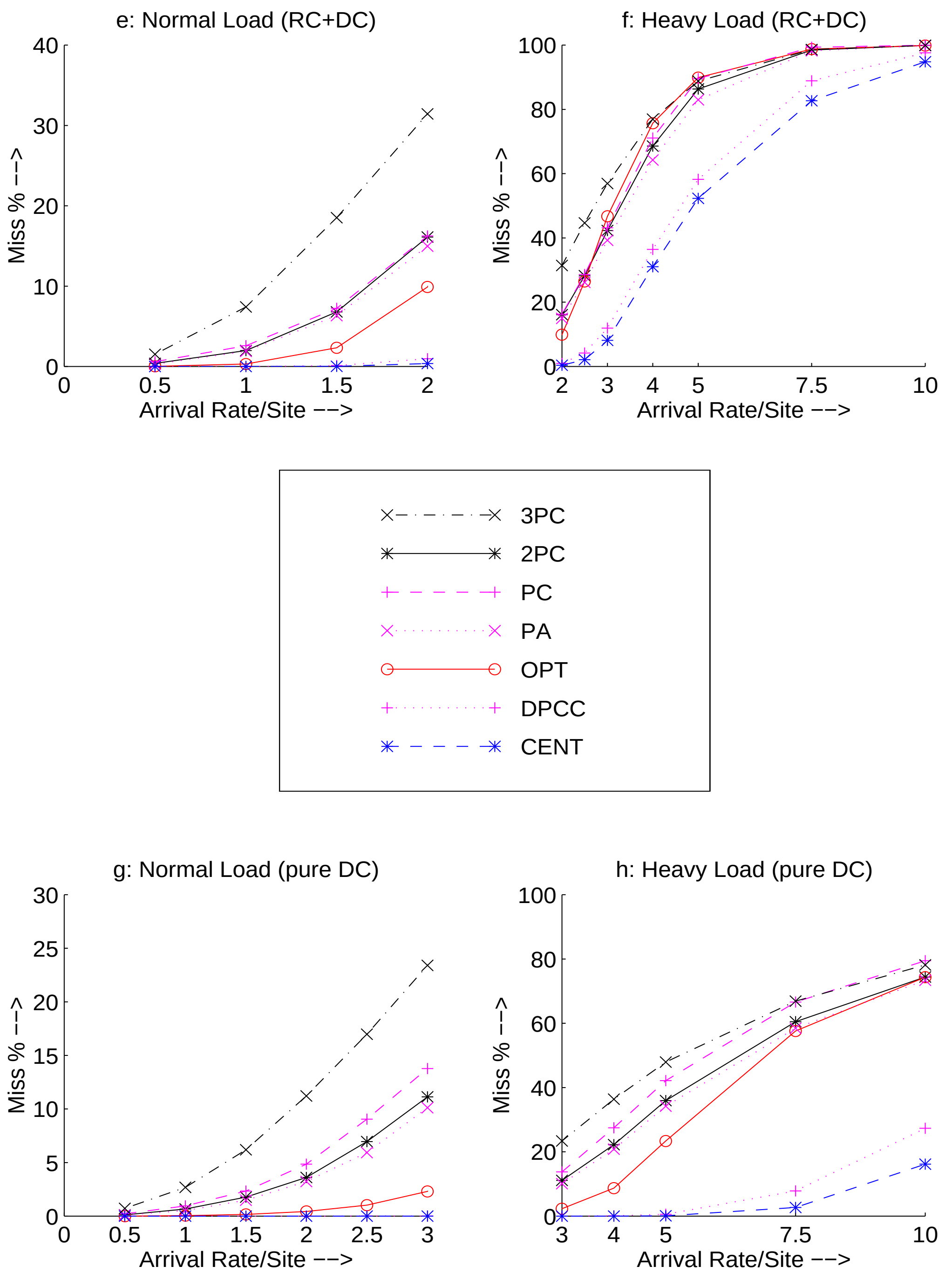


Figure 10.6: (e–h) Higher Distribution (MissPercent, Parallel Transactions)

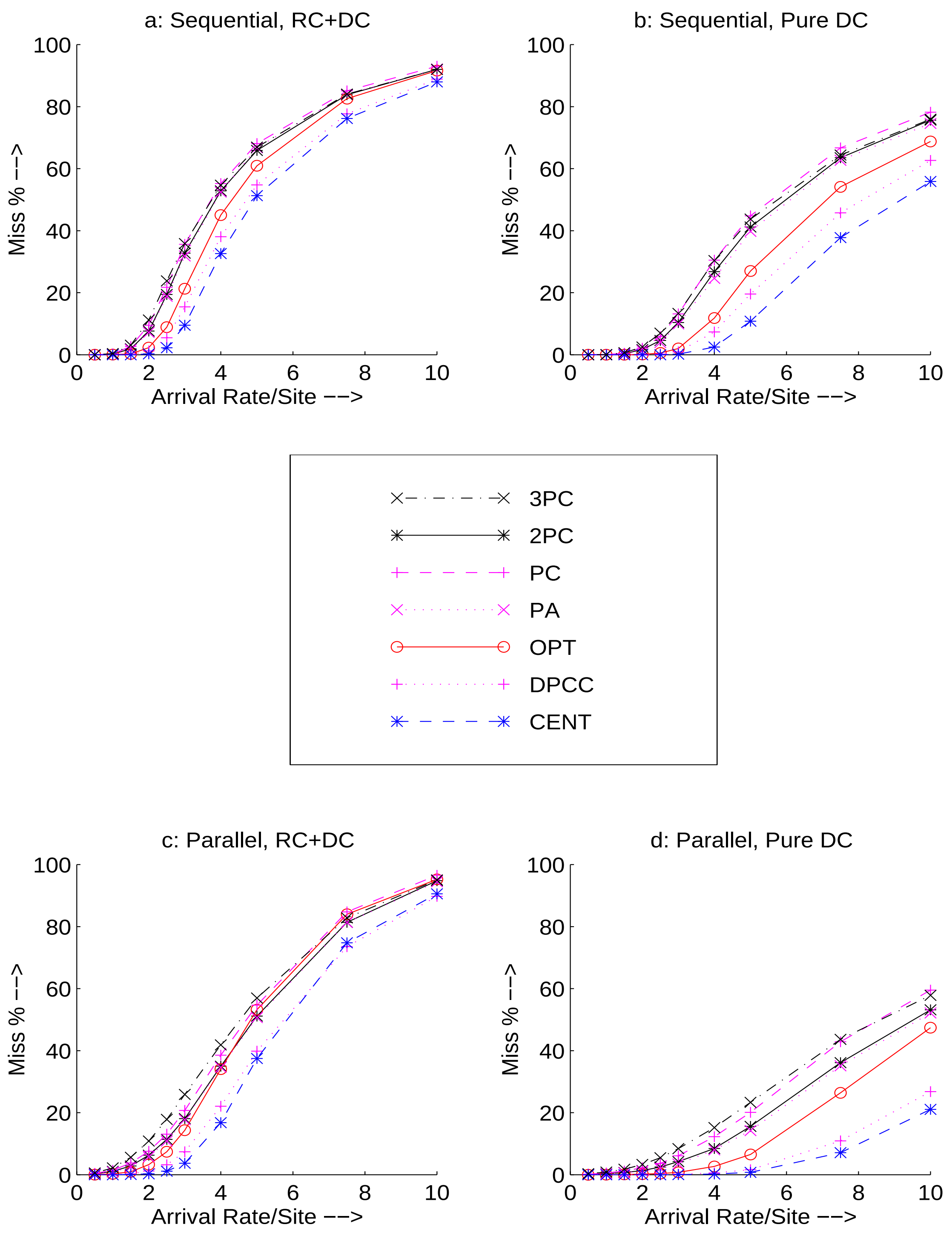


Figure 10.7: Update Probability = 0.5 (MissPercent)

10.8 Experiment 7: RT-OPT-PA and RT-OPT-PC

The implementation of RT-OPT used in the previous experiments incorporated only the optimistic access, active abort and silent kill optimizations. We also conducted an experiment to investigate the effect on performance of adding the PA or PC optimizations to RT-OPT (called RT-OPT-PA and RT-OPT-PC, respectively). The results of this experiment are shown in Figures 10.8a–10.8d for the sequential and parallel transactions under both RC+DC and pure DC scenarios, for the limited distribution ($DistDegree = 3$) case.

In these figures, we observe that just as PA and PC provided little improvement on the performance of standard 2PC, here also they provide no tangible benefits to the performance of the RT-OPT protocol and for the same reasons. While RT-OPT-PA is very slightly better than basic RT-OPT, RT-OPT-PC performs worse than basic RT-OPT under heavy loads, especially under pure DC scenario.

In the OLTP system experiments, we had observed that the performance of PC optimization (and PA optimization, when aborts were considered) was superior to that of 2PC for the higher degree of distribution case. For RTDB systems, however, we saw in Experiment 5 that PC and PA optimizations do not result in significant improvement in the real-time performance even when the degree of distribution is high. PC, in fact, performs slightly worse than 2PC, as in all previous real-time experiments. Thus, we expect that PA/PC optimizations will fail to provide tangible performance benefits when combined with RT-OPT even for increased distribution case. To verify this, we have plotted in Figure 10.8e–10.8h, the performance of RT-OPT-PA and RT-OPT-PC for the $DistDegree = 6$ case, for the sequential and parallel transaction executions under RC+DC and pure DC scenarios. The results are as expected—the performance of RT-OPT-PA is only marginally better than basic RT-OPT, while the performance of RT-OPT-PC is worse than that of basic RT-OPT, more so for the pure DC case.

In summary, PA and PC optimizations fail to provide tangible benefits even when combined with RT-OPT, for a variety of workloads and distribution-degrees considered in our experiments.

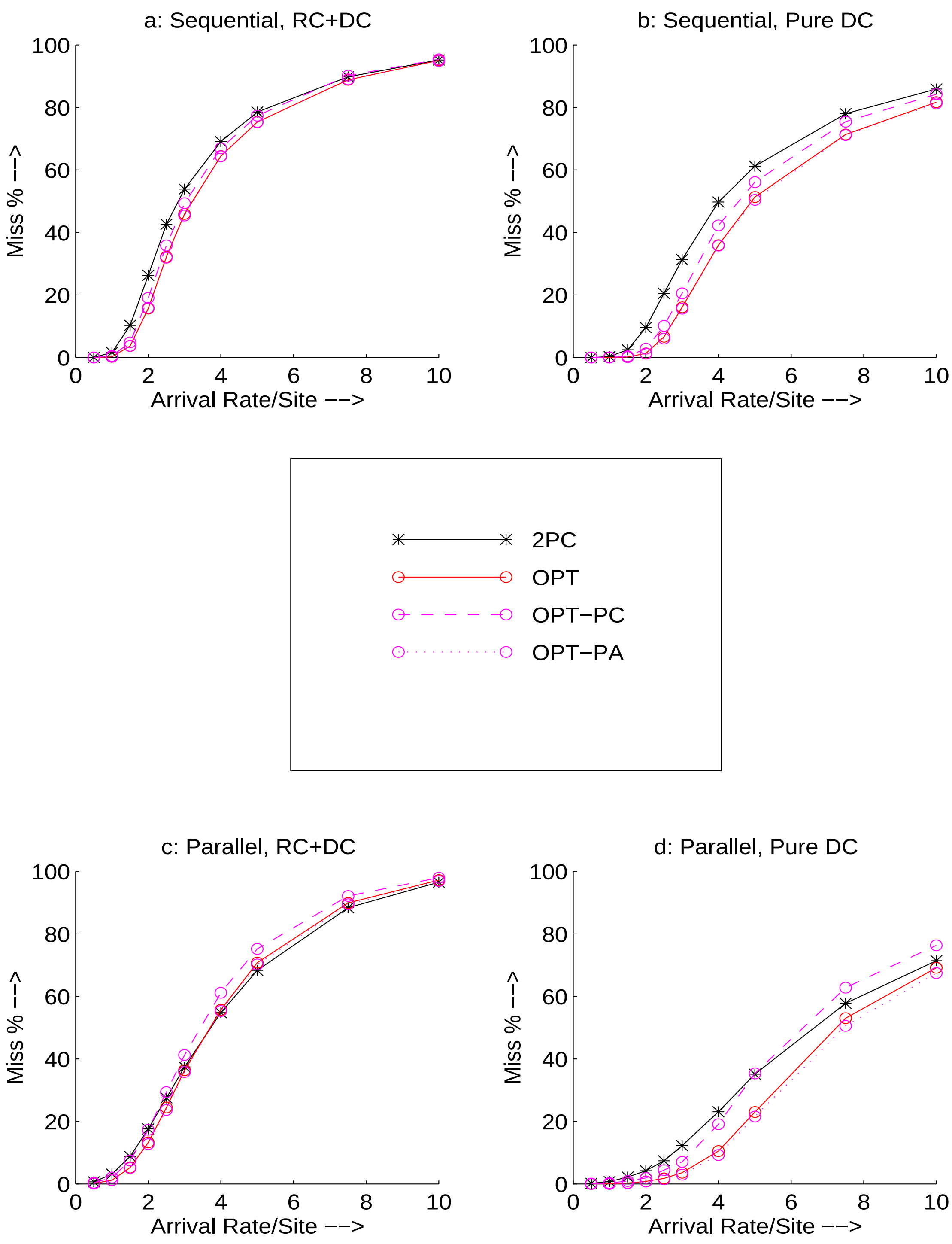


Figure 10.8: (a–d) PA / PC Optimizations (MissPercent, $DistDegree = 3$)

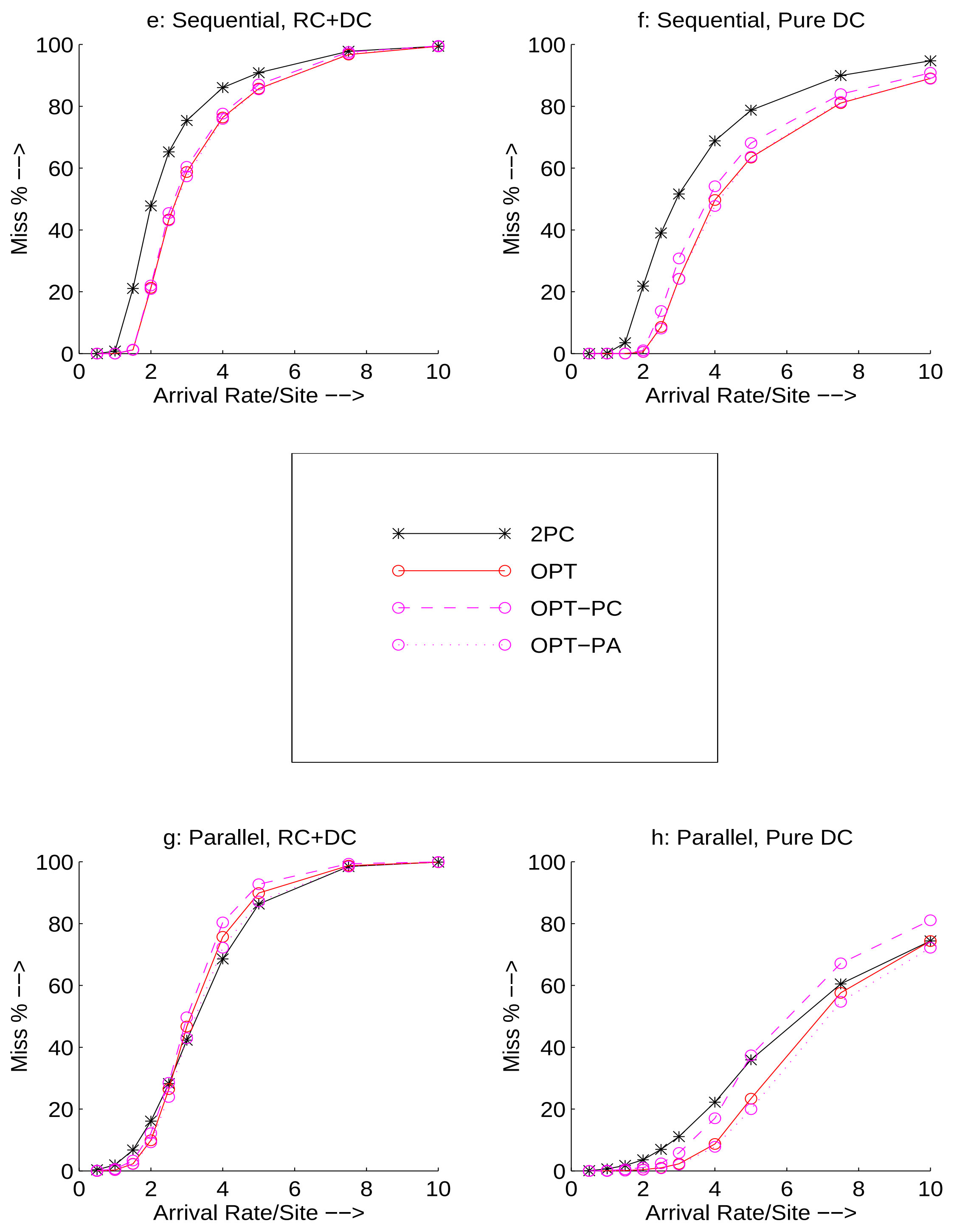


Figure 10.8: (e–h) PA / PC Optimizations (MissPercent, $DistDegree = 6$)

10.9 Experiment 8: Non-Blocking RT-OPT

In the previous experiments, we observed that RT-OPT, which is based on 2PC, performed significantly better than the standard protocols. In this experiment, we evaluate the effect of incorporating the same optimizations in **3PC**. We refer to this protocol as RT-OPT-3PC. The results of this experiment are shown in Figures 10.9a—10.9d for the sequential transactions, and in Figures 10.9e–10.9h for the parallel transactions, under both RC+DC and pure DC conditions.

We observe from these figures that the performance of RT-OPT-3PC is not only superior to that of 3PC, but also significantly better than that of 2PC. In fact, the performance of RT-OPT-3PC is close to that of RT-OPT itself. The reason for this superior performance of RT-OPT-3PC is that the optimistic feature has more effect on 3PC than on 2PC due to the larger commit processing phase of 3PC.

These results indicate that, in the real-time domain, nonblocking functionality which is extremely useful in case of failures can be purchased at a relatively modest increase in routine processing cost.

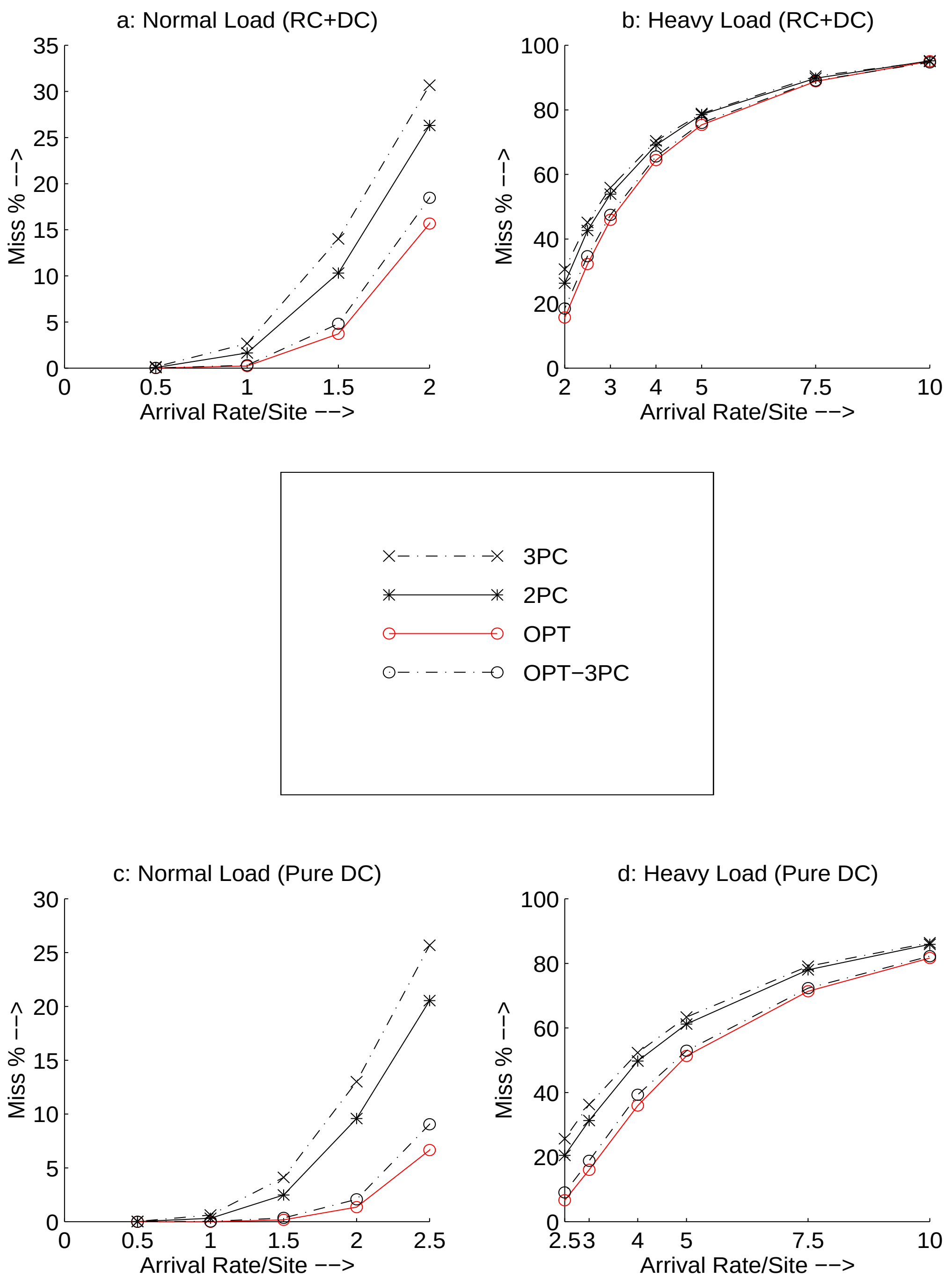


Figure 10.9: (a–d) Non-Blocking RT-OPT (MissPercent, Sequential Transactions)

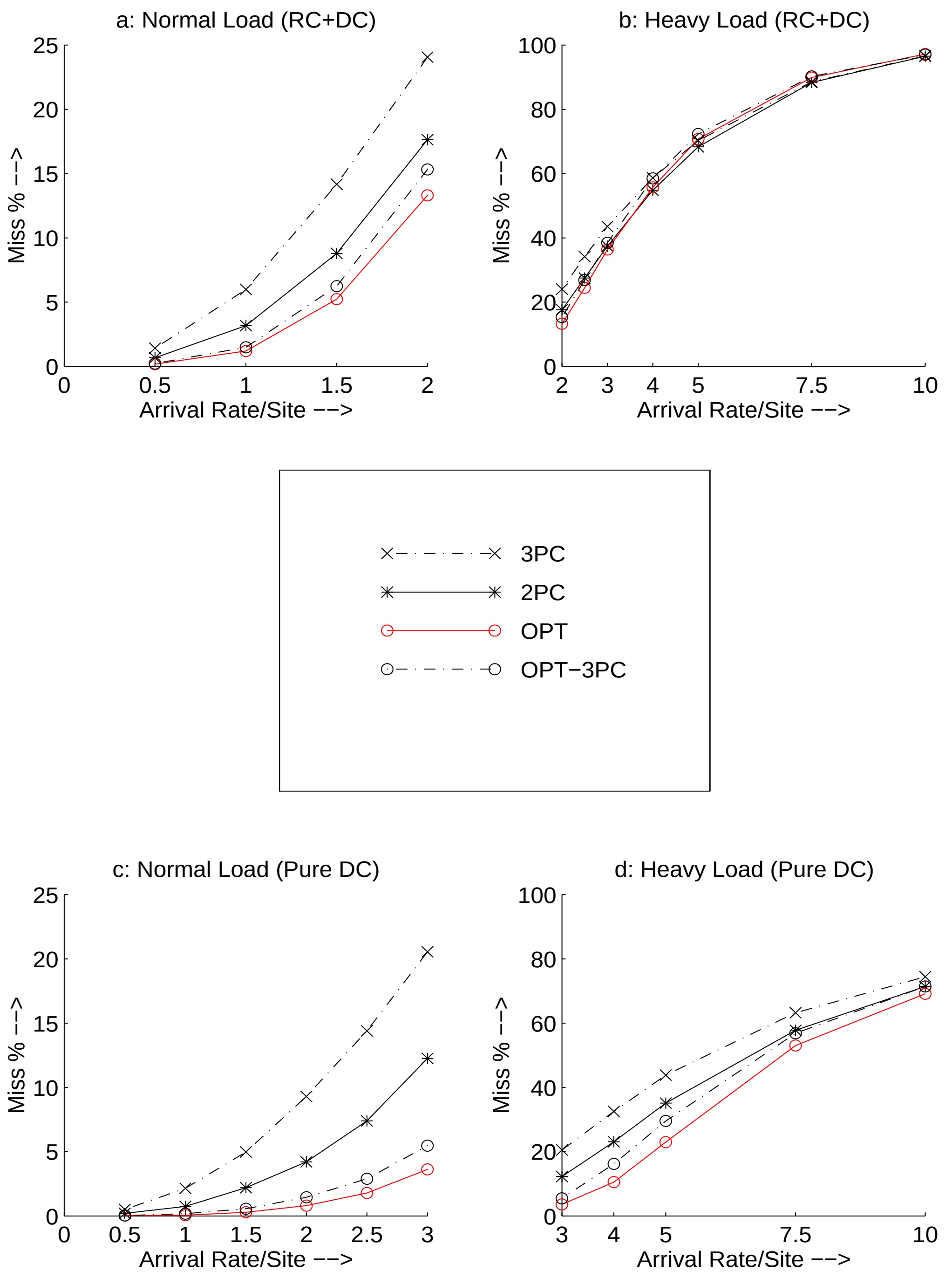


Figure 10.9: (e–h) Non-Blocking RT-OPT (MissPercent, Parallel Transactions)

CHAPTER

11 Optimizing RT-OPT

We have seen in the previous chapter that significant improvement in the real-time performance can be achieved by using RT-OPT. We also observed, however, that the success ratio of RT-OPT decreases at heavy loads because a significant number of cohorts are close to their deadlines when they become lenders. One way to improve the success ratio could have been to allow only those cohorts to become lenders that are *healthy*, that is, not close to their deadlines. In this chapter we discuss some such optimizations that can be applied to RT-OPT in order to further improve its performance. These optimizations include the **RT-OPT-B/BM** optimizations, the **Healthy Lenders** optimization, and the **Shadow Transaction** optimization, described below. Further, we evaluate the performance of these optimizations by conducting the simulation experiments using the model and the methodology described in previous chapters, and discuss various tradeoffs involved in using these optimizations.

11.1 RT-OPT-B / BM

In RT-OPT, a situation may arise where the borrower cohort finishes its execution before the lender cohort receives any decision from its master (Section 8.4). In such a case, the borrower cohort goes on the *shelf* waiting for the lender to commit, and is not allowed to immediately send the WORKDONE message to its master. Thus, it increases the response time of the transaction, more so for the sequential transaction execution. If, on the other hand, the borrower was allowed to send the WORKDONE message, it could have led to the problem of *cascading aborts*, where a borrower in turn becomes a lender. However, if we somehow ensure that a borrower is never allowed to become a lender despite the fact that it does not wait on the shelf for the lender to commit before sending the WORKDONE message, it can have a favorable impact on the performance of RT-OPT. The guarantee that a borrower is never allowed to become a lender is achieved in the **RT-OPT-B** variant of RT-OPT, in the following manner: A borrower sends the WORKDONE message without waiting for the lender to commit. However, it also sends one extra bit, called the "borrower bit" with the WORKDONE message. The presence of the borrower bit with the message indicates to the master that the cohort is still a borrower and therefore, the transaction should not be allowed to become a lender. The master, while sending the PREPARE messages to the cohorts, passes the borrower bit to each cohort. A cohort, on receiving a borrower bit, enters the prepared state in the normal fashion, but the lock manager is notified not to allow the optimistic access to the data items locked by this cohort. Thus, sending a borrower bit with the WORKDONE message turns off the optimistic feature for a particular transaction, and hence avoids the possibility of cascading aborts.

While RT-OPT-B may help reduce the response time of a transaction, it has the risk of preventing many potential lenders from lending the data. Another variant, **RT-OPT-BM**, attempts to overcome this problem in the following manner: A borrower cohort sends the WORKDONE message with the borrower bit (just like RT-OPT-B). However, the cohort later sends a BORROWOVER message to the master if the lender subsequently commits and the cohort has not yet received any PREPARE message from the master. This message at the master effectively invalidates the borrower bit sent earlier. Note that the master ignores this message if received after it has sent the PREPARE messages. Further, if more than one cohort had sent the borrower bit, each one needs to send the BORROWOVER message in order to invalidate all the borrower bits received by the master, otherwise the transaction will not be allowed to become a lender. While RT-OPT-BM attempts to gain from the advantages of both RT-OPT and RT-OPT-B, it can have adverse impact on the performance due to the CPU overheads of the extra message

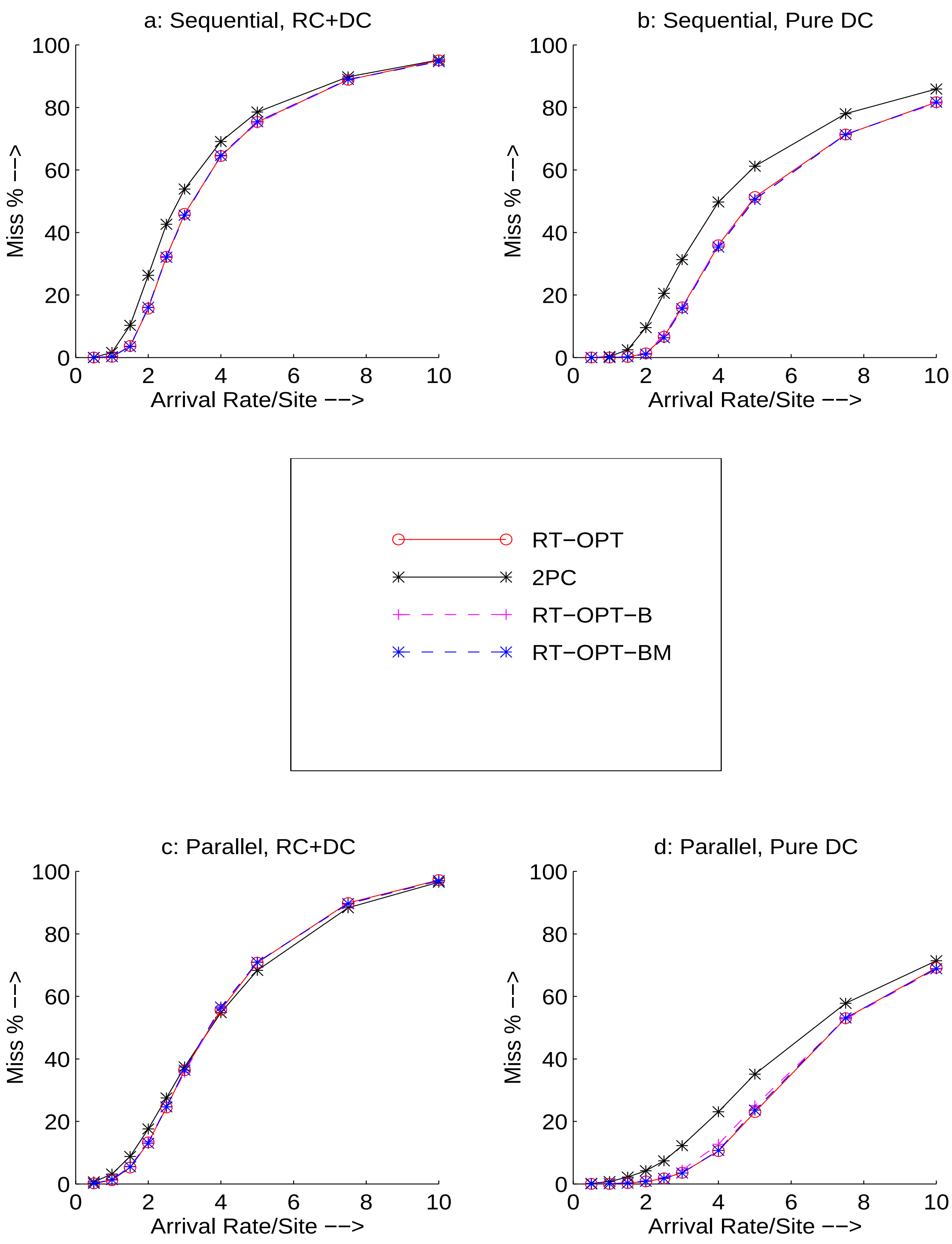


Figure 11.1: RT-OPT-B / BM variants of RT-OPT (MissPercent)

processing, if there is heavy resource contention in the system.

The deadline miss performance of these variants of RT-OPT is shown in Figures 11.1a–11.1d for sequential and parallel transactions under RC+DC and pure DC conditions. We observe from these figures that the performance difference in these variants and the RT-OPT are negligible for almost all the cases. A marginal difference at low loads is observed only under pure DC in the parallel case, where RT-OPT-B performs slightly worse than RT-OPT because the borrower bit does not really help in the parallel case and it permanently tags some potential lenders as non-lenders. These performance losses due to RT-OPT-B are regained by RT-OPT-BM, which performs almost identically to RT-OPT.

11.2 Healthy Lenders

RT-OPT does not take into account the fact that a transaction that is close to its deadline can be killed before the commit processing is over. In order to improve the success ratio of RT-OPT at heavy loads, a transaction should be allowed to become a lender only if it is not close to its deadline and is expected to receive its final decision before the deadline expires. A variant of RT-OPT attempts to address this issue by using the concept of **Healthy Lenders**. We refer this variant as **RT-OPT-Hn**, where n is a *permissible health factor*. The health factor of a transaction (at the point of time when the master is ready to send PREPARE messages) is defined as the ratio of the time left to its deadline to the minimum time required for the commit processing of the transaction (a minimum of two messages and one force-write are processed before the master can take a decision). If the health factor of a transaction is more than the permissible health factor, it is allowed to become a lender, otherwise it is not allowed to become the lender. It should be noted here that we do not make use of the health factor to decide the fate of the transaction—we merely use it to decide whether the transaction can lend the data to the borrowers. Thus, even erroneous estimates about the message processing times and log force-write times will only affect the extent to which the optimistic feature of RT-OPT can be used. For extremes, the value of n may not help RT-OPT at all if it is too low (as compared to the time required for commit processing), or it can turn off the optimistic feature completely if it is very high (for example, comparable to the ratio of average transaction response time to the minimum required commit processing time).

The MissPercent results for this experiment are shown in Figures 11.2a–11.2d for sequential and parallel transactions under both RC+DC and pure DC cases. These figures show the results for two different values (1.0 & 2.0) of permissible health factor. We see from these figures that the benefits from the healthy lenders are only marginal. In Figure 11.2c, where the performance of RT-OPT is slightly worse than that of 2PC at heavy loads, the performance of RT-OPT-H1 is not worse than that of 2PC. The performance of RT-OPT-H1 is visibly superior to that of RT-OPT at heavy loads for the parallel transactions under pure DC case (Figure 11.2d), as the transactions close to their deadlines are not allowed to become lenders. This is evident from Figures 11.2e–11.2h where the success ratio of RT-OPT-H1 is considerably higher than that of RT-OPT. Figures 11.2i–11.2l present the borrow ratio for RT-OPT, RT-OPT-H1, and RT-OPT-H2. It is clear from these figures that *not many* potential lenders are tagged as non-lenders by RT-OPT-H1 (the reason being that there was no error in the message processing delays and force-writes).

Finally, the performance of RT-OPT-H1 and that of RT-OPT-H2 are almost identical. This shows that health factor of unity is sufficient to filter out the transactions vulnerable to deadline kill in the commit phase. This is because the transactions close to deadlines have the highest priority at the physical resources. Thus, they do not see any resource contention in the system, and hence the minimum time required to do the commit processing is actually sufficient for them.

11.3 Shadow Transactions

The abort of a transaction at any point usually results in the complete restart of the transaction. Thus all the work done by the borrowing transaction is wasted. A scheme for concurrency control mechanism has been suggested in [Bes94, BB94] wherein a transaction has *shadows*—replicas of the transaction that are created in case of a data access conflict.

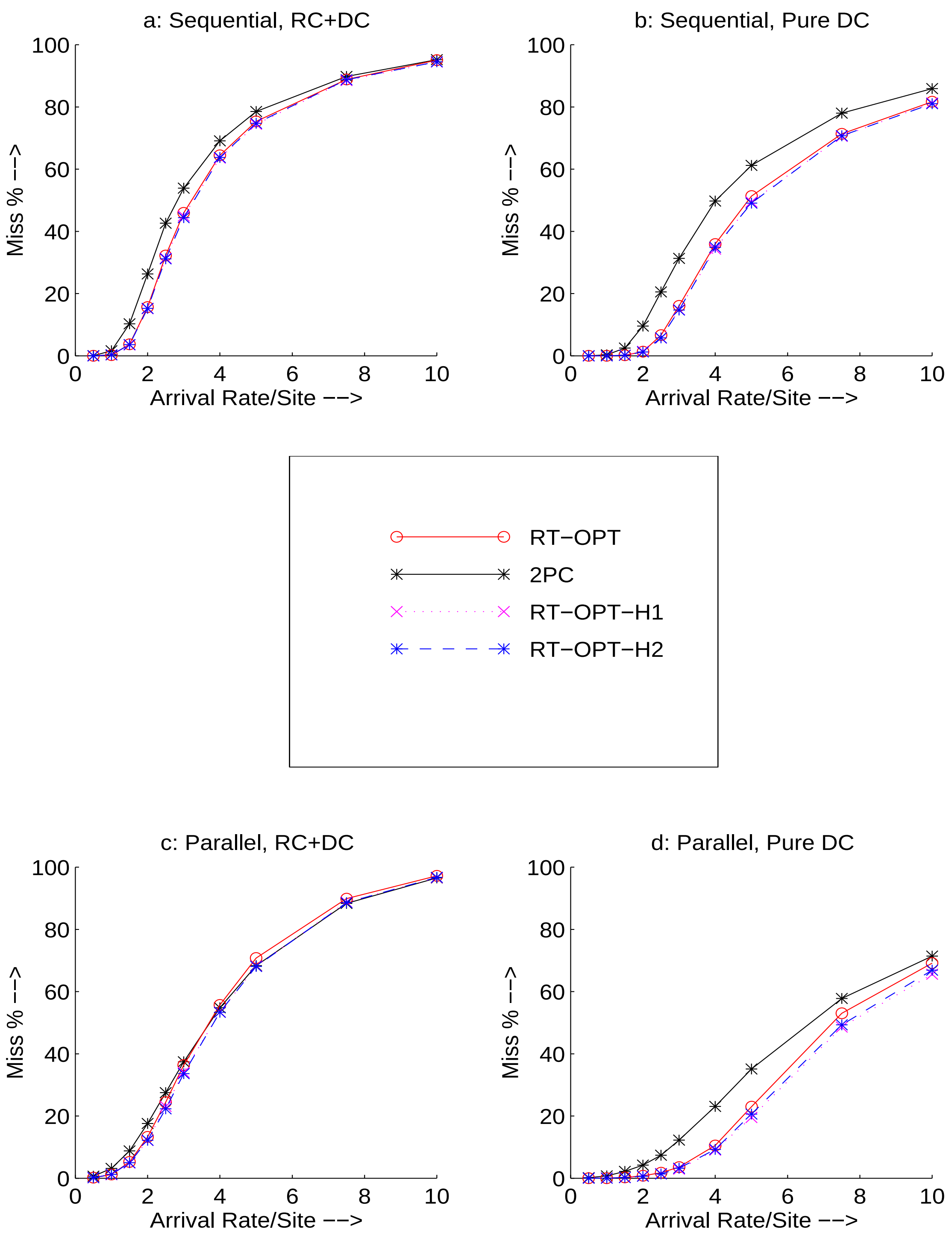


Figure 11.2: (a–d) Healthy Lenders (MissPercent)

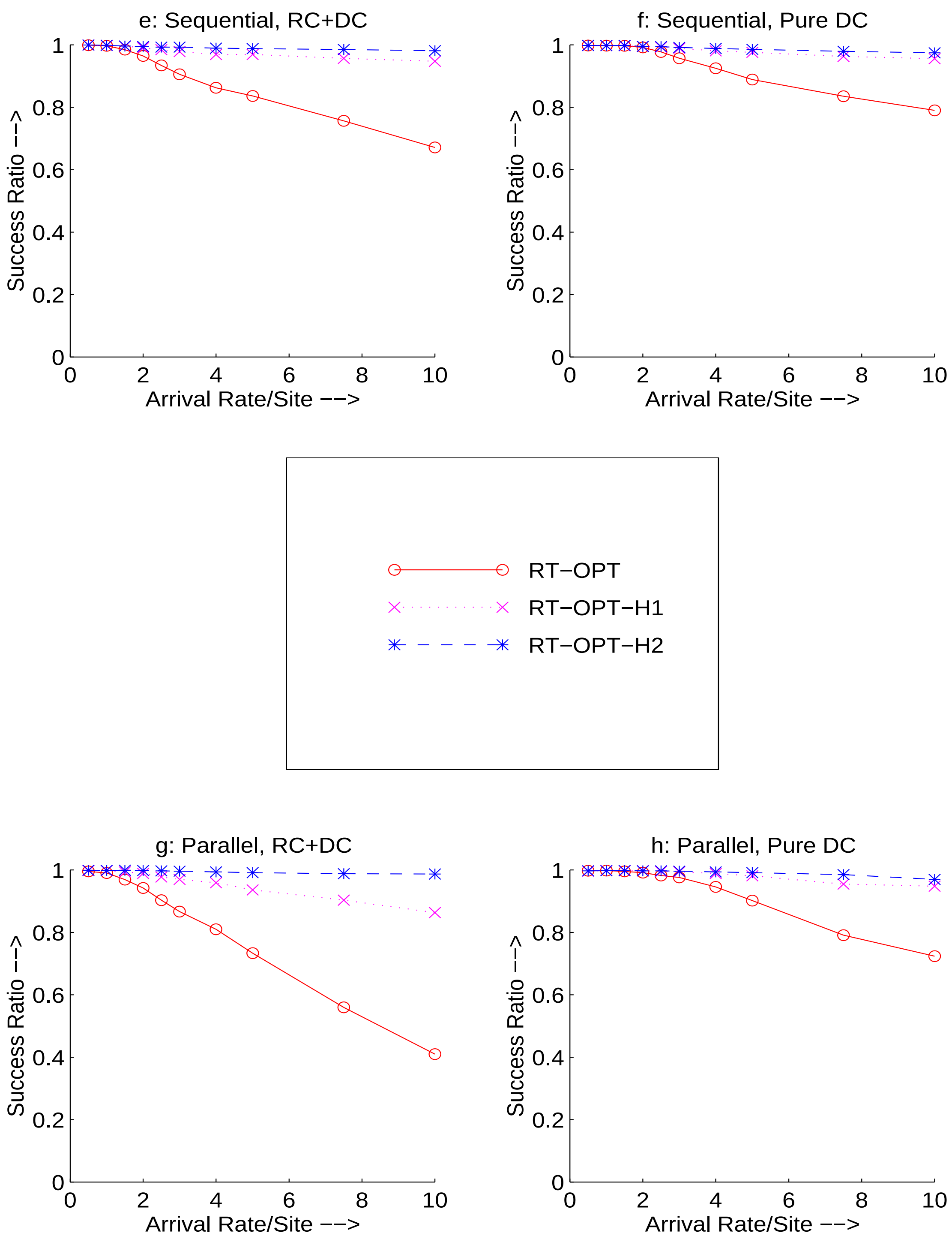


Figure 11.2: (e–h) Healthy Lenders (Success Ratio)

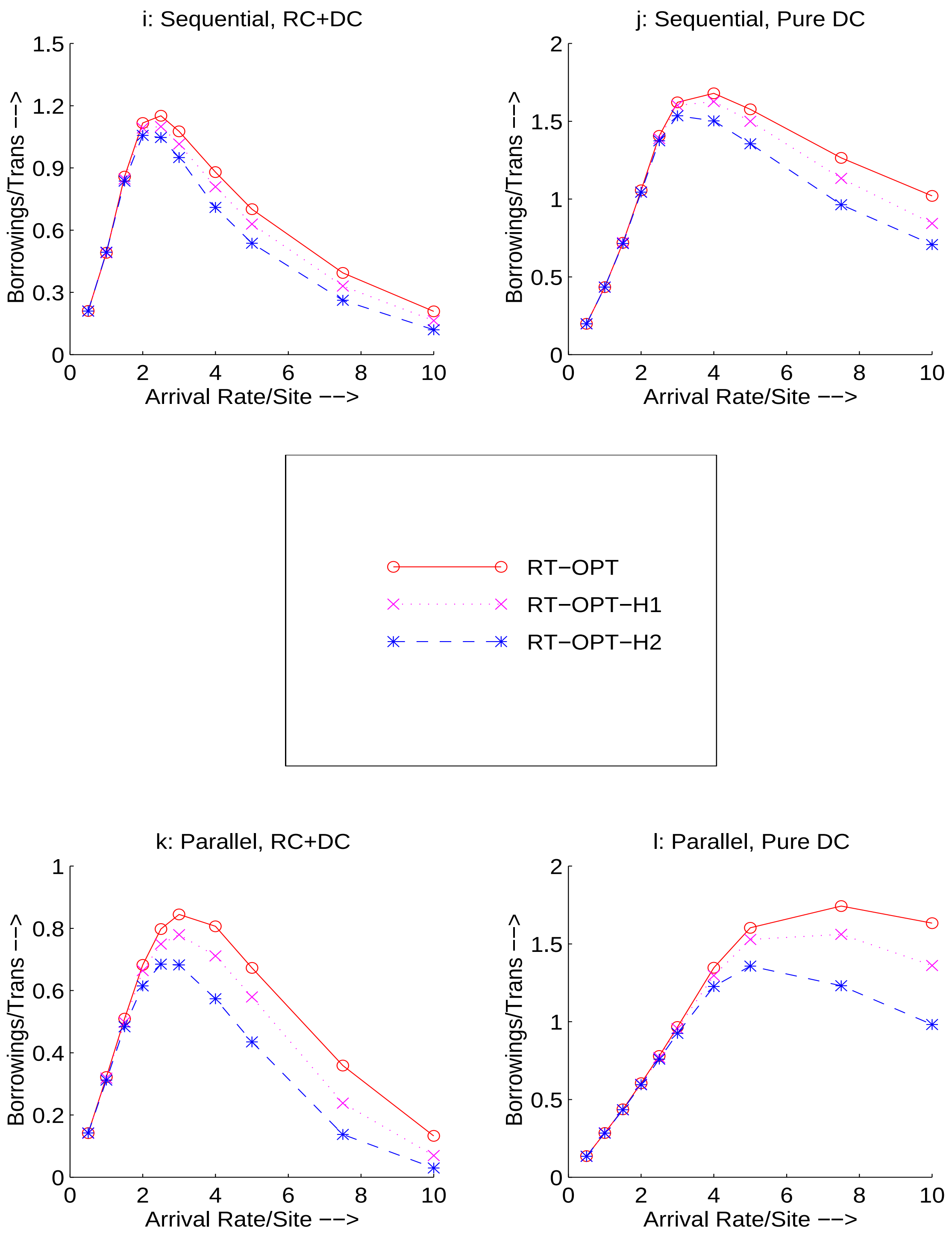


Figure 11.2: (i–l) Healthy Lenders (Borrow Ratio)

The original incarnation of the transaction continues the execution by accessing the data held by some other yet uncommitted transaction, while the shadow transaction is blocked at the point of conflict. If the original transaction finally aborts, the shadow resumes the execution, otherwise the shadow is discarded. We use a similar scheme to improve the performance of the RT-OPT protocol wherein a cohort forks off a shadow when it borrows a data page. The original cohort continues if the lending transaction is finally committed, otherwise the shadow resumes the execution and the original cohort is aborted. Thus the work done by the cohort before borrowing is not wasted.

For correctness, a shadow cohort can resume execution only if the original cohort had not exchanged any message with the master after the creation of the shadow. Otherwise, there can be dependencies among the original cohort and the master of which the shadow cohort is unaware of, and these dependencies need to be handled before the shadow cohort can resume the execution. In our experiments, we do not assume such dependencies and we discard the shadow cohort if the original cohort exchanges any message (e.g., WORKDONE message) with the master. Also, we do not implement the shadow mechanism for all data access as we are interested in finding its effect on the performance of RT-OPT. Thus, the shadow cohorts are created only when a cohort borrows some data from a lender. Also, there may be a possibility of multiple shadows if a borrower cohort borrows multiple times during its execution. For sake of simplicity, we allow only one shadow to exist at a time which is created at the first borrowing—creation of another shadow is allowed only if the cohort aborts and the shadow resumes the execution replacing the original cohort.

We refer to the above shadow optimization of RT-OPT as **RT-OPT-S**. The results for this experiment are shown in Figures 11.3a–11.3d for the limited distribution case, for sequential and parallel transactions under RC+DC and pure DC conditions. We see from these figures that the performance of RT-OPT-S is better than that of RT-OPT only by a very marginal amount, under almost all situations. Only in case of parallel transaction execution for the pure DC case (Figure 11.3d), RT-OPT-S shows a visible improvement (not very significant, and that too only at heavy loads) over RT-OPT, which may not be sufficient to justify the efforts involved in the implementation of shadow transaction concept. Similar results are observed for various other workloads and systems, for example, fast network (Figures 11.3e–11.3h), higher degree of distribution (Figures 11.3i–11.3l), and reduced update probability (Figures 11.3m–11.3p) experiments. In fact, similar gains can be obtained by using *healthy lenders* as observed in the previous section.

In summary, the performance gains achieved by using RT-OPT-S are marginal for most workloads, and similar gains can be achieved by using RT-OPT-H1.

11.4 Concluding Remarks

In this chapter, we presented a set of optimizations for the RT-OPT protocol in order to improve its performance (that is, the *RT-OPT-B/BM* optimization, the *healthy lenders* optimization, and the *shadow transactions* optimization). We also performed simulation experiments to study the impact of using these optimizations on the deadline miss percent in an RTDB system.

Generally, we expect the *shadow transactions* optimization to provide the *optimal* performance, at least under pure DC scenario, because the aborting of a lender cohort does not result in the borrower transaction being restarted. Instead, the shadow cohort replaces the original incarnation of the borrower cohort and resumes execution at the point where it was originated. The amount of work wasted due to aborting the cohort does not have much impact when the resources are not the bottleneck.

The experimental results showed that the above optimizations provide only marginal performance improvements over the basic RT-OPT protocol for most of the workloads. The performance of the RT-OPT-B/BM optimizations is virtually similar to that of the RT-OPT protocol. Also, the performance of the *healthy lenders* optimization and that of the *shadow transactions* optimization is only *marginally* better than that of the RT-OPT protocol for most of the workloads. A slight benefit of using the *healthy lenders* optimization or the *shadow transactions* optimization is visible only for a limited set of workloads—in particular, for parallel transaction execution under the pure DC scenario in heavily loaded situations. Note that even for these workloads the performance gains

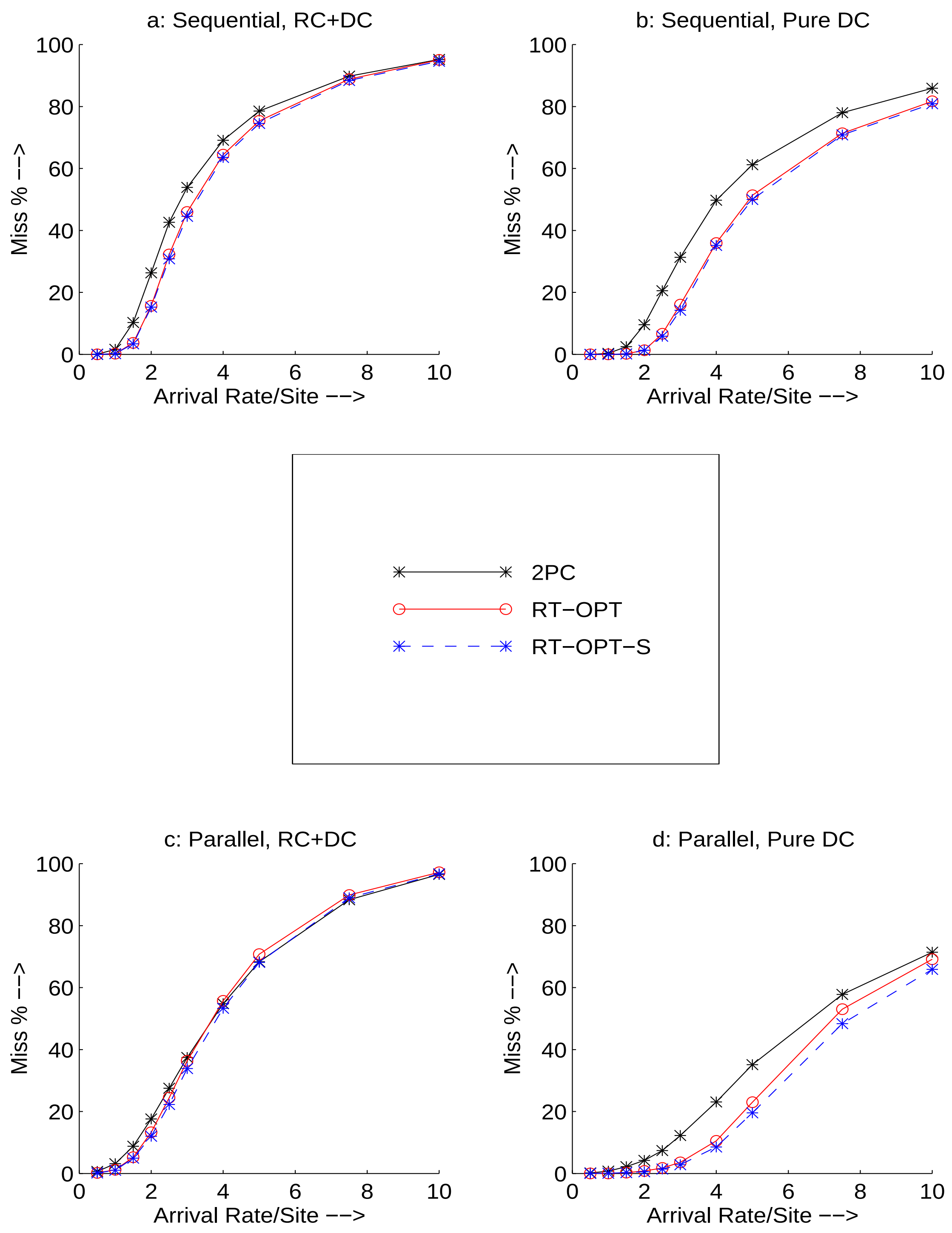


Figure 11.3: (a–d) Shadow Transactions (MissPercent, $DistDegree = 3$)

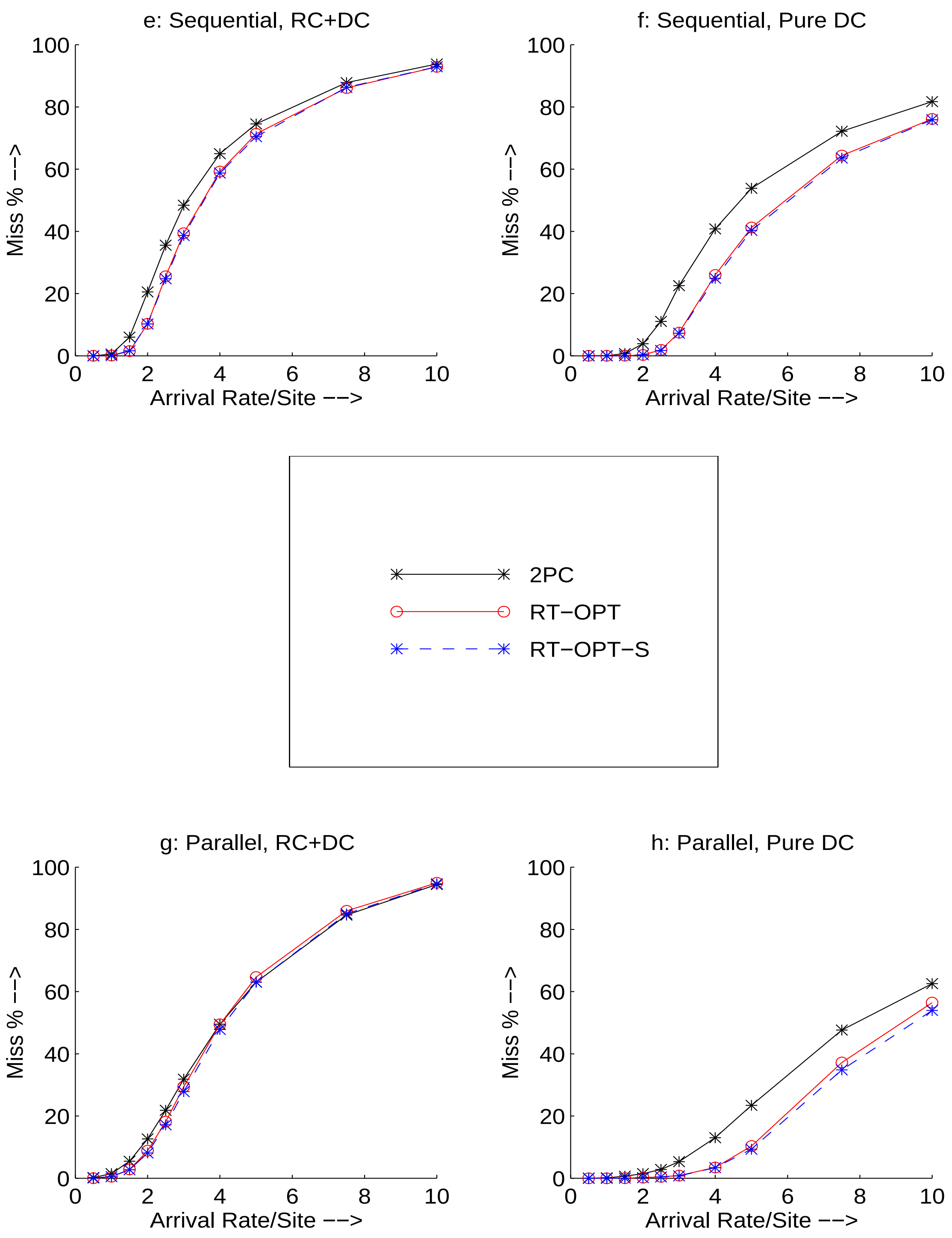


Figure 11.3: (e–h) Shadow Transactions (MissPercent, $MsgCpu = 1\,ms$)

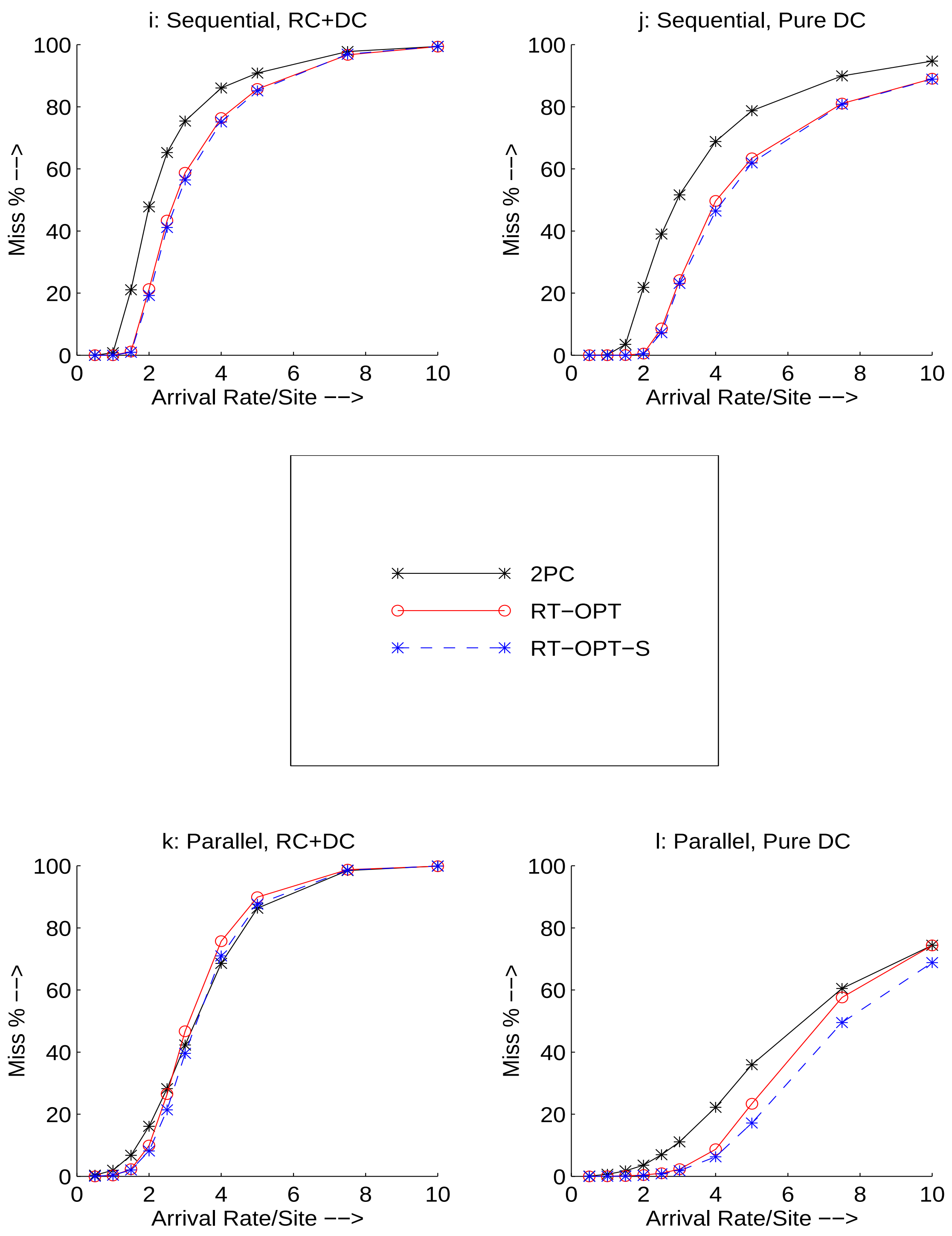


Figure 11.3: (i–l) Shadow Transactions (MissPercent, $DistDegree = 6$)

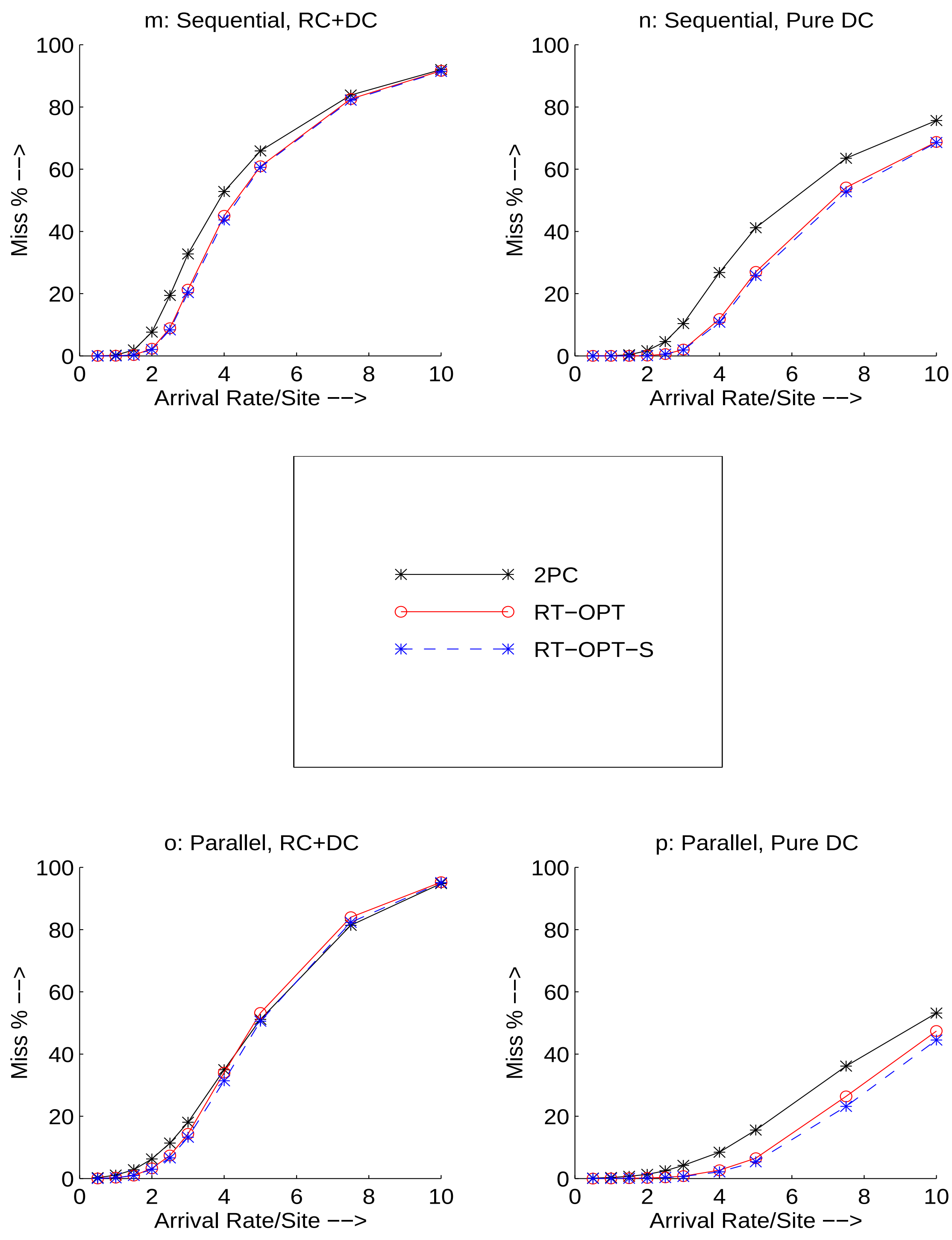


Figure 11.3: (m–p) Shadow Transactions (MissPercent, $UpdateProb = 0.5$)

achieved by the above optimizations are not very significant—the performance of these optimizations is *slightly* better than that of the RT-OPT protocol.

We may conclude from the above discussion that the performance achieved by the RT-OPT protocol is, in some sense, close to the *optimal* performance. If, even then some systems are willing to include the above discussed optimizations in order to improve (slightly) the performance, the *healthy lenders* optimization appears to be the appropriate choice as the performance improvement achieved by this optimization is almost *same* as that achieved by the *shadow transactions* optimization. In addition *healthy lenders* optimization does not incur extra overheads required in the *shadow transactions* optimization.

CHAPTER

12 Conclusions

Although a significant body of research literature exists for centralized real-time database systems, comparatively little work has been done on distributed RTDB systems. In particular, the problem of commit processing in a distributed environment has not yet been addressed in detail. The few papers on this topic assume that it is necessary to relax the traditional notion of atomicity when commit protocols are designed for distributed real-time databases. In this part of the thesis, we have proposed and evaluated new mechanisms for designing high performance real-time commit protocols that do not require transaction atomicity requirements to be weakened.

We first precisely defined the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a firm deadline distributed real-time environment. Subsequently, we made a detailed study of the relative performance of different commit protocols in a firm deadline distributed real-time database system. Using a detailed simulation model of a firm deadline distributed RTDB system, we evaluated the deadline miss performance of the real-time variants of a variety of standard commit protocols including 2PC, Presumed Abort, Presumed Commit, and 3PC. We also developed and evaluated a new commit protocol, RT-OPT, that was designed specifically for the real-time environment and included features such as controlled optimistic access to uncommitted data, active abort and silent kill. To the best of our knowledge, these are the first quantitative results in this area.

Our experiments, which covered a variety of transaction workloads and system configurations, demonstrated the following:

1. Distributed commit processing can have considerably more effect than distributed data processing on the real-time performance. This highlights the need for developing commit protocols tuned to the real-time domain.

2. The standard 2PC and 3PC algorithms perform poorly in the real-time environment due to preventing access to data held by cohorts in the prepared state, and due to their passive nature.

3. The PA and PC variants of 2PC, although reducing protocol overheads, fail to provide tangible benefits in the real-time environment. These protocols provide improved performance in conventional database systems in certain situations, for example, higher degree of distribution. In real-time domain, however, not even a single experiment in our study exhibited more than marginal performance improvement of PA, while PC actually performs slightly worse than 2PC under many situations. However, this 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] due to which the performance impact of these protocols in a system having large number of these kinds of transactions could be different.

4. The new protocol, RT-OPT, provides significantly improved performance over the standard algorithms for almost all of the workloads and system configurations considered in this study. Its good performance is attained primarily due to its optimistic borrowing of uncommitted data and active abort policy. The optimistic access of the prepared data significantly reduces the effect of priority inversion which is inevitable in the prepared state. Supporting statistics showed that RT-OPT's optimism about uncommitted data is justified, especially under normal loads. The other optimizations of silent kill and presumed commit/abort, however, had comparatively little beneficial effect.

5. Experiments combining the optimizations of RT-OPT with 3PC indicate that the non-blocking functionality can be obtained in the real-time environment at a relatively modest cost in normal processing performance. This is especially encouraging given the high desirability of the non-blocking feature in the real-time environment.

6. The performance of the RT-OPT protocol is, in some sense, close to the *optimal* performance. The other optimizations of RT-OPT (e.g., *RT-OPT-B/BM*, *healthy lenders*, *shadow transactions*, etc.) provide only marginal performance improvements over RT-OPT for most workloads. If, even then some systems are willing to include the above discussed optimizations in order to slightly improve the performance, the *healthy lenders* optimization appears to be the appropriate choice.

In summary, our results have shown that the conventional commit protocols perform poorly in a real-time environment, the PA/PC optimizations fail to provide tangible benefits even though these optimizations are used in many commercial conventional database products, and the performance of RT-OPT in firm deadline RTDB systems is considerably superior to that of the standard commit protocols.

CHAPTER

13 Summary and Future Research

In this thesis we have addressed the issue of profiling the *quantitative performance* of distributed transaction commit protocols in the context of two important categories of distributed database systems: (1) Distributed On-Line Transaction Processing Systems (OLTP), and (2) Distributed Real-Time Database Systems (RTDB). Using detailed simulation models of distributed OLTP systems and distributed RTDB systems, we evaluated the performance of a representative suite of commit protocols in the context of these systems for a variety of distributed transaction workloads and system configurations. This work addresses the lack of adequate studies of the relative merits of various commit protocols with respect to their quantitative impact on the transaction processing performance.

We also proposed and evaluated two new commit protocols—OPT for the OLTP systems, and RT-OPT for the RTDB systems. These new protocols are specifically designed to reduce the *blocking* arising out of locks held on prepared data by allowing transactions to "optimistically" borrow this *uncommitted* data. OPT and RT-OPT are easy to implement and incorporate in current systems. Further, these protocols can coexist and be integrated, often synergistically, with most of the other commit protocol optimizations proposed in the literature.

Salient observations from our experiments are the following:

- Distributed *commit* processing for most workloads (in both OLTP and RTDB systems) can have considerably more effect than distributed *data* processing on the system performance, thus, making the choice of commit protocol an important decision from a performance perspective.

- The PA and PC variants of 2PC which reduce protocol overheads and have been incorporated in a number of database products and transaction processing standards, provide tangible benefits over 2PC only in a few restricted situations in the context of OLTP systems, while for RTDB systems no tangible performance gains are achieved by using these variants.

- For OLTP systems the new protocol, OPT, either by itself or in conjunction with other standard optimizations, provides significantly improved performance over the standard commit protocols for all the workloads and system configurations considered in the study. In fact, the performance of OPT was often close to that obtained with the DPCC baseline, which represents an upper bound on the performance of commit protocols. Its good performance is attained primarily due to its reduction of the blocking arising out of locks held on prepared data.

- For the RTDB systems too, the performance of RT-OPT is significantly superior to that of the standard commit protocols for almost all of the workloads and system configurations considered in the study. Its good performance is attained primarily due to its *optimistic borrowing* of uncommitted data and *active abort policy.* The optimistic access of the prepared data significantly reduces effect of priority inversion which is inevitable in the prepared state. Supporting statistics showed that RT-OPT's optimism about uncommitted data is justified, especially under normal loads.

  A feature of the OPT (resp. RT-OPT) protocol design is that it limits the abort chain to one, thereby preventing cascading aborts. This is done by allowing *only* the prepared transactions to lend their data and ensuring that borrowers cannot enter the prepared state until their borrowing is terminated.

- A combination of OPT (resp. RT-OPT) and 3PC provides the performance that is not only superior to that of 3PC, but also significantly better than that of classical

2PC in OLTP (resp. RTDB) systems. Thus, by using 3PC with OPT (resp. RT-OPT), the non-blocking functionality can be obtained at a relatively modest increase in routine processing cost

- OPT's design is based on the assumption that the transactions that lend their uncommitted data will almost always commit (in the context of OLTP systems). However, OPT is fairly *robust* in that it maintains its superior performance unless the probability of *surprise transaction aborts* in the commit phase is *unrealistically* high. In our experiments, OPT exhibited superior performance even when the transaction abort probability was as high as 10 percent. A *performance crossover* was observed at the transaction abort probability of 15 percent—a level much higher than what might be expected in practice. Beyond this level, OPT's performance becomes progressively worse than that of the classical protocols.

- In the real-time domain, the performance of RT-OPT is close to the *optimal* performance in the sense that further improvements in its performance by using a variety of other optimizations were of only marginal value for most of the workloads.

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 (resp. RT-OPT) algorithm, that is, OPT, OPT-PA or OPT-PC (resp. RT-OPT, RT-OPT-PA or RT-OPT-PC). If having non-blocking functionality is important but 3PC has not been used due to its excessive overheads resulting in poor performance, then OPT-3PC (resp. RT-OPT-3PC) appears to be an attractive choice since it provides a performance that is better than that of the standard 2PC protocols.

Future Research Directions

It will be interesting to actually implement OPT (resp. RT-OPT) in a commercial system and verify the performance benefits indicated by the simulation experiments. The experience gained from the implementation of OPT (resp. RT-OPT) could provide many insights and could help in designing even more efficient commit protocols.

We have conducted this study for update-oriented transaction workloads. It will be interesting to consider different mixes of read-only and update-oriented transaction workloads and investigate the impact of PA/PC optimizations in such situations. Further, 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) would be another interesting problem to investigate.

Finally, we believe that in the real-time domain, the understanding gained from this work (that is, in the context of firm deadline RTDB systems) will provide necessary insight for addressing the more complex framework of soft deadline applications. It will be interesting to profile the performance implications of the commit protocols in soft deadline RTDB systems and compare the resulting behavior of the commit protocols in the two RTDB frameworks.

References

[AA90] Divyakant Agrawal and Amr El Abbadi. Locks with Constrained Sharing. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, April 1990.

[AA91] Divyakant Agrawal and Amr El Abbadi. Ordered Sharing: A New Lock Primitive for Database Systems. Technical Report TRCS 91-18, Department of Computer Science, University of California at Santa Barbara, September 1991.

[AAL91] Divyakant Agrawal, Amr El Abbadi, and A. E. Lang. Performance Characteristics of Protocols With Ordered Shared Locks. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, April 1991.

[AAL94] Divyakant Agrawal, Amr El Abbadi, and A. E. Lang. The Performance of Protocols Based on Locks with Ordered Sharing. *IEEE Transactions on Knowledge and Data Engineering*, 6(5), 1994.

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987.

[AD85] Rakesh Agrawal and David J. Dewitt. Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation. *ACM Transactions on Database Systems*, 10(4), 1985.

[AGM88] Robert K. Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, August 1988.

[AGM92] Robert K. Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation. *ACM Transactions on Database Systems*, 17(3), 1992.

[AHC95] Yousef J. Al-Houmaily and Panos K. Chrysanthis. Two-Phase Commit in Gigabit-Networked Distributed Databases. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, September 1995.

[AHC96] Yousef J. Al-Houmaily and Panos K. Chrysanthis. The Implicit-Yes Vote Commit Protocol with Delegation of Commitment. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, September 1996.

[AHCL97] Yousef J. Al-Houmaily, Panos K. Chrysanthis, and Steven P. Levitan. An Argument in Favor of the Presumed Commit Protocol. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, April 1997.

[BB94] Azer Bestavros and Spyridon Braoudakis. Timeliness via Speculation for Real-time Databases. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994.

[BC92] Paul M. Bober and Michael J. Carey. Multiversion Query Locking. In *Proceedings of the 18th International Conference on Very Large Data Bases*, August 1992.

[BC95] Sujata Banerjee and Panos K. Chrysanthis. Data Sharing and Recovery in Gigabit-Networked Databases. In *Proceedings of the 4th International Conference on Computer Communications and Networks*, September 1995.

[BC96] Sujata Banerjee and Panos K. Chrysanthis. A Fast and Robust Failure Recovery Scheme for Shared-Nothing Gigabit-Networked Databases. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, September 1996.

[Bes94] Azer Bestavros. Multi-version Speculative Concurrency Control with Delayed Commit. In *Proceedings of the 1994 International Conference on Computers and their Applications*, March 1994.

[Bes96] Azer Bestavros. Advances in Real-Time Database Systems Research. *Special Section on Advances in Real-Time Database Systems, ACM SIGMOD RECORD*, 25(1), 1996.

[BH95] Mikael Berndtsson and Jörgen Hansson, editors. *Active and Real-Time Database Systems (ARTDB-95), Proceedings of the First International Workshop on Active Real-Time Database Systems, Skövde, Sweden.* Workshops in Computing. Springer-Verlag, London, June 1995.

[BH96] Mikael Berndtsson and Jörgen Hansson. Workshop Report: The First International Workshop on Active and Real-Time Database Systems (ARTDB-95). *ACM SIGMOD RECORD*, 25(1), 1996.

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

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[BRGP78] Philip A. Bernstein, J. B. Rothnie, Nathan Goodman, and C. H. Papadimitriou. The Concurrency Control Mechanism of SDD-1: A System for Disrtributed Databases (the Fully Redundant Case). *IEEE Transactions on Software Engineering*, SE-4(3), 1978.

[CKL90] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. Load Control for Locking: The 'Half-and-Half' Approach. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, April 1990.

[CL88] Michael J. Carey and Miron Livny. Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication. In *Proceedings of the 14th International Conference on Very Large Data Bases*, August 1988.

[CL89] Michael J. Carey and Miron Livny. Parallelism and Concurrency Control Performance in Distributed Database Machines. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, June 1989.

[CL91] Michael J. Carey and Miron Livny. Conflict Detection Tradeoffs for Replicated Data. *ACM Transactions on Database Systems*, 16(4), 1991.

[Cod70] E. F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6), 1970.

[CP84] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases: Principles & Systems.* McGraw-Hill, New York, 1984.

[DGS$^+$90] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[DLW89] Susan Davidson, Insup Lee, and Victor Wolfe. A Protocol for Timed Atomic Commitment. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), 1976.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[FHRT93] Peter A. Franaszek, Jayant R. Haritsa, John T. Robinson, and Alexander Thomasian. Distributed Concurrency Control Based on Limited Wait-Depth. *IEEE Transactions on Parallel and Distributed Systems*, 4(11), 1993.

[FR85] Peter A. Franaszek and John T. Robinson. Limitations of Concurrency Control in Transaction Processing. *ACM Transactions on Database Systems*, 10(1), 1985.

[FRT92] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency Control for High Contention Environments. *ACM Transactions on Database Systems*, 17(2), 1992.

[FZT$^+$92] Michael J. Franklin, Michael J. Zwilling, C. K. Tan, Michael J. Carey, and David J. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, June 1992.

[GG94] S. Guella and Le Gruenwald. Recovery for Real-Time Main Memory Database Systems. In *Proceedings of the* 22^{nd} *ACM Computer Science Conference*, 1994.

[GHOK81] Jim Gray, P. Homan, Ron Obermarck, and H. Korth. A Strawman Analysis of the Probability of Waiting and Deadlock in a Database System. IBM Research Report RJ3066, IBM San Jose Research Laboratory, February 1981.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[Gra78] Jim Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. 60, Springer-Verlag, Berlin, 1978.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, September 1981.

[Gra96a] Jim Gray. Personal communication, August 1996.

[Gra96b] Jim Gray. Personal communication, November 1996.

[Hae84] Theo Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2), 1984.

[HCL90] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, December 1990.

[HCL92] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Data Access Scheduling in Firm Real-Time Database Systems. *The Journal of Real-Time Systems*, 4(3), 1992.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4), 1983.

[IBM93] *Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2.* Document No. GC30-3084-5, IBM, July 1993.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990.

[Kna87] Edgar Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4), 1987.

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

[KR81] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), 1981.

[KS91] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, New York, second edition, 1991.

[LAA94a] Mei-Ling Liu, Divyakant Agrawal, and Amr El Abbadi. The Performance of Two-Phase Commit Protocols in the Presence of Site Failures. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, June 1994.

[LAA94b] Mei-Ling Liu, Divyakant Agrawal, and Amr El Abbadi. The Performance of Two-Phase Commit Protocols in the Presence of Site Failures. Technical Report TRCS94-09, Department of Computer Science, University of California at Santa Barbara, April 1994.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 1978.

[LKS91a] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. A Theory of Relaxed Atomicity. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1991.

[LKS91b] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, May 1991.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1), 1973.

[LL93] Butler Lampson and David Lomet. A New Presumed Commit Optimization for Two Phase Commit. In *Proceedings of the 19th International Conference on Very Large Data Bases*, August 1993.

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

[LSG$^+$79] Bruce G. Lindsay, Patricia G. Selinger, Cesare A. Galtieri, Jim Gray, Raymond A. Lorie, Thomas G. Price, Franco Putzolu, Irving L. Traiger, and Bradford W. Wade. Notes on Distributed Databases. IBM Research Report RJ2571, IBM San Jose Research Laboratory, July 1979.

[MBCS92] C. Mohan, Kathryn Briton, Andrew Citron, and George Samaras. Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols. IBM Research Report RJ8684, IBM Almaden Research Center, March 1992.

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

[MHL$^+$92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992.

[ML83] C. Mohan and Bruce Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. IBM Research Report RJ3881, IBM Almaden Research Center, June 1983.

[MLO86] C. Mohan, B. Lindsay, and Ron Obermarck. Transaction Management in the $R^\star$ Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986.

[MN82] Daniel A. Menascé and Tatuo Nakanishi. Performance Evaluation of a Two-Phase Commit Based Protocol for DDBs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[MN94] C. Mohan and Inderpal Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, May 1994.

[Moh92] C. Mohan. Less Optimism About Optimistic Concurrency Control. In *Proceedings of the* 2^{nd} *International Workshop on RIDE: Transaction and Query Processing*, February 1992.

[Moh93] C. Mohan. IBM's Relational DBMS Products: Features and Technologies. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, May 1993.

[MSF83] C. Mohan, H. R. Strong, and S. Finkelstein. Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors. IBM Research Report RJ3882, IBM San Jose Research Laboratory, June 1983.

[ÖV91] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Pap79] C. H. Papadimitriou. Serializability of Concurrent Updates. *Journal of the ACM*, 26(4), 1979.

[SBCM93] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, April 1993.

[SBCM95] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *International Journal on Distributed and Parallel Databases*, 3(4), 1995.

[SC90] James W. Stamos and Flaviu Cristian. A Low-Cost Atomic Commit Protocol. In *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, October 1990.

[SC93] James W. Stamos and Flaviu Cristian. Coordinator Log Transaction Execution Protocol. *International Journal on Distributed and Parallel Databases*, 1(4), 1993.

[She93] Mark Sherman. Architecture of the Encina Distributed Transaction Processing Family. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, May 1993.

[SIM94] *C++SIM User's Guide, Public Release 1.5.* Department of Computing Science, Computing Laboratory, University of Newcastle upon Tyne, 1994.

[SJR91] Peter M. Spiro, Ashok M. Joshi, and T. K. Rengarajan. Designing an Optimized Transaction Commit Protocol. *Digital Technical Journal*, 3(1), 1991.

[SK92] Sang H. Son and Spiros Kouloumbis. Replication Control for Distributed Real-Time Database Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[Ske81] Dale Skeen. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, May 1981.

[SKPO88] Michael Stonebraker, Randy Katz, David Patterson, and John Ousterhout. The Design of XPRS. In *Proceedings of the 14th International Conference on Very Large Data Bases*, August 1988.

[SL76] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems*, 1(3), 1976.

[SLKS92] Nandit Soparkar, Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. Adaptive Commitment for Real-Time Distributed Transactions. Technical Report TR-92-15, Department of Computer Science, University of Texas-Austin, 1992.

[Son88] Sang H. Son, editor. *ACM SIGMOD RECORD, Special Issue on Real-Time Database Systems*, 17(1), 1988.

[Son90] Sang H. Son. Real-Time Database Systems: A New Challenge. *Data Engineering*, 13(4), 1990.

[SRL87] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, 1987.

[SRL88] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Concurrency Control for Distributed Real-Time Databases. *ACM SIGMOD RECORD*, 17(1), 1988.

[Sto79] Michael Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5(3), 1979.

[TGS85] Y. C. Tay, Nathan Goodman, and Rajan Suri. Locking Performance in Centralized Databases. *ACM Transactions on Database Systems*, 10(4), 1985.

[Tho93] Alexander Thomasian. Two-Phase Locking Performance and Its Thrashing Behavior. *ACM Transactions on Database Systems*, 18(4), 1993.

[Tri82] Kishor S. Trivedi. *Probability & Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[UB92] Özgür Ulusoy and Geneva G. Belford. Real-Time Lock-Based Concurrency Control in Distributed Database Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[Ulu92] Özgür Ulusoy. Current Research on Real-Time Databases. *ACM SIGMOD RECORD*, 21(4), 1992.

[Ulu94a] Özgür Ulusoy. A Study of Two Transaction Processing Architectures for Distributed Real-Time Database Systems. Technical Report BU-CEIS-94-22, Department of Computer Engineering and Information Science, Bilkent University, 1994.

[Ulu94b] Özgür Ulusoy. Research Issues in Real-Time Database Systems. Technical Report BU-CEIS-94-32, Department of Computer Engineering and Information Science, Bilkent University, 1994.

[WD95] Seth J. White and David J. DeWitt. Implementing Crash Recovery in QuickStore: A Performance Study. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, May 1995.

[WDH$^+$82] R. Williams, Dean Daniels, Laura M. Haas, George Lapis, Bruce G. Lindsay, Pui Ng, Ron Obermarck, Patricia G. Selinger, A. Walker, Paul F. Wilms, and R. A. Yost. $R^\star$: An Overview of the Architecture. In *Proceedings of the 2nd International Conference on Databases: Improving Usability and Responsiveness*, June 1982.

[Yoo94] Yong Ik Yoon. *Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems.* PhD thesis, Department of Information and Communication Engineering, Korea Advanced Institute of Science and Technology, 1994.