TRANSACTION EXECUTION IN MULTIDATABASE SYSTEMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Timuçin Devirmiş
July, 1996

TRANSACTION EXECUTION IN MULTIDATABASE SYSTEMS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION SCIENCE AND THE INSTITUTE OF ENGINEERING AND SCIENCE OF BILKENT UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

By

Timuçin Devirmiş

July, 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Dr. Özgür Ulusoy (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Dr. Cevdet Aykanat

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Dr. İlyas Çiçekli

Approved for the Institute of Engineering and Science:

M. Baray

Prof. Dr. Mehmet Baray,
Director of Institute of Engineering and Science

ABSTRACT

TRANSACTION EXECUTION IN MULTIDATABASE SYSTEMS

Timuçin Devirmiş
M.S. in Computer Engineering and Information Science
Advisor: Asst. Prof. Dr. Özgür Ulusoy
July, 1996

Most work in the multidatabase systems (MDBSs) area has focused on the issues of transaction management and concurrency control. It is difficult to implement traditional transaction management techniques in a MDBS due to the heterogeneity and autonomy of the connected local sites. In this thesis, we present a new transaction execution model that captures the formalism and semantics of various extended transaction models and adopts them to a MDBS environment. The proposed model covers nested transactions, various dependency types among subtransactions, and commit-independent transactions. The execution model does not make any assumption regarding the concurrency control protocols executed at the local sites connected to the MDBS.

We also present a detailed simulation model of a MDBS to analyze the performance of the proposed model. The performances of both the traditional transaction model and the proposed transaction model are evaluated under a range of workloads and system configurations. The performance impact of global transactions' behavior on local transactions is also discussed.

Keywords: Multidatabases systems, distributed databases, transaction models, performance evaluation.

ÖZET

ÇOKLU VERİTABANI SİSTEMLERİNDE HAREKETLERİN İŞLETİLMESİ

Timuçin Devirmiş
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışman: Yrd. Doç. Dr. Özgür Ulusoy
Temmuz, 1996

Çoklu veritabanı alanında yapılan çalışmalar genellikle hareketlerin yönetimi ve kontrolü üzerine yoğunlaşmıştır. Klasik hareket yönetimi tekniklerini çoklu veritabanı sistemlerinde uygulamak, sisteme katılan yerel veritabanlarının dışarıdan kontrol edilemeyişi ve heterojenliğinden dolayı çok zordur. Bu tez çalışmasında, çoklu veritabanları ile ilgili, birçok genişletilmiş hareket modellerinin anlamsal özelliklerini ve formatını içeren yeni bir hareket modeli sunulmaktadır. Önerilen model iç içe geçmiş hareket modellerini, alt hareketlerin bağımlılıklarını ve bağımsız sona erebilen hareket biçimlerini kapsamaktadır. Sunulan işletim modeli yerel veritabanları hakkında herhangi bir öngörü gerektirmemektedir.

Önerilen hareket modelinin performans değerlendirilmesi için ayrıca detaylı bir simulasyon modeli de sunulmuştur. Simulasyon modeli kullanılarak, klasik hareket modeli ile beraber önerilen modelin performans değerlendirmeleri yapılmıştır. Yeterli sistem kaynakları olduğunda, önerilen hareket modelinin, klasik hareket modelinden çok daha iyi sonuçlar verdiği gözlemlenmiştir. Yapılan deneylerde, çoklu veritabanı hareketlerinin yerel hareketler üzerindeki etkisi de incelenmiştir.

Anahtar sözcükler: Çoklu veritabanı sistemleri, dağıtık veritabanları, hareket modelleri, performans değerlendirmeleri.

ACKNOWLEDGMENTS

I am very grateful to my supervisor, Assistant Professor Özgür Ulusoy for his invaluable guidance and motivating support during this study.

I would also like to thank to Assoc. Prof. Dr. Cevdet Aykanat and Asst. Prof. Dr. İlyas Çiçekli for reading and commenting about the thesis.

Finally, I want to express my deepest gratitude to my family, for everything they did to bring me to this position. I dedicate this thesis to them.

Contents

1 Introduction **1**

2 Transaction Model **4**

2.1 Related Work 4

2.2 Proposed Transaction Model 6

3 Execution Architecture **12**

3.1 Ensuring Global Atomicity 15

3.2 Ensuring Global Serializability 17

3.2.1 Employing The Optimistic Ticketing Method (OTM) . . 17

3.2.2 Employing The Conservative ticketing method (CTM) 19

3.3 Commit Independent Subtransactions 20

3.4 The Global Deadlock Problem 23

4 Simulation Model **25**

4.1 Introduction 25

4.2 MDBS Simulation Model 27

4.2.1 Transaction Model 28

4.2.2 Simulation Model Components 32

5 Simulation Experiments **36**

5.1 Introduction 36

5.1.1 Algorithm Settings 36

5.1.2 Performance Metrics 37

5.1.3 General Parameter Settings 38

5.2 Experiments on the Classical Transaction Model 40

5.2.1 The Impact of Data Contention 41

5.2.2 The Impact of Resource Contention 46

5.2.3 The Impact of Transaction Length 48

5.2.4 The Impact of Local Transaction Behavior on Global Transactions 50

5.2.5 Algorithms' Impact on Local Transactions 51

5.3 Experiments on the Extended Global Transaction Model 55

5.3.1 The Impact of Subtransaction Dependencies 55

5.3.2 The Impact of Commit Independent Subtransactions 61

5.3.3 The Impact of Data Contention 64

5.3.4 The Impact of Extended Transactions on Local Transactions 65

5.4 Experiments on Global Deadlock Detection Algorithms 69

5.5 Performance Comparison of the Classical Transaction Model and the Extended Transaction Model 71

6 Conclusions **77**

List of Figures

2.1 A transaction tree representation of Example 1. 10

2.2 A transaction tree representation of Example 2. 11

3.1 The architecture of the MDBS 14

4.1 MDBS closed network model 28

4.2 Global transaction model 31

4.3 Simulation Model Components 32

5.1 Global throughput vs. GWriteProb, LResourceUnit=20 41

5.2 Global throughput vs. GWriteProb, LResourceUnit=20. GSG-Time=0.05 42

5.3 Global blocking time vs. GWriteProb, LResourceUnit=20 42

5.4 Global conflict ratio vs. GWriteProb, LResourceUnit=20 43

5.5 Global abort ratio vs. GWriteProb, LResourceUnit=20 43

5.6 Global aborts of OTM, LResourceUnit=20 44

5.7 Global aborts of CTM, LResourceUnit=20 44

5.8 Global throughput vs. GWriteProb, LResourceUnit=20, LHotRegion=0.2 45

5.9 Global abort ratio vs. GWriteProb, LResourceUnit=20, LHotRegion=0.2 . . . 45

5.10 Global throughput vs. GWriteProb, NumChild=4, LResourceUnit=20 . . . 46

5.11 Global abort ratio vs. GWriteProb, NumChild=4, LResourceUnit=20 . . . 46

5.12 Global throughput vs. GWriteProb, LResourceUnit=1 47

5.13 Global abort ratio vs. GWriteProb, LResourceUnit=1 47

5.14 Global throughput vs. GWriteProb, NumChild=4, LResourceUnit=1 48

5.15 Global throughput vs. transaction length, LResourceUnit=20 48

5.16 Global throughput vs. transaction length, LResourceUnit=1 . . 49

5.17 Global abort ratio vs. transaction length, LResourceUnit=20 49

5.18 Global abort ratio vs. transaction length, LResourceUnit=1 50

5.19 Global throughput vs. LWriteProb, LResourceUnit=1 51

5.20 Global abort ratio vs. LWriteProb, LResourceUnit=1 51

5.21 Local throughput vs. GWriteProb, LResourceUnit=20 52

5.22 Local blocking time vs. GWriteProb. LResourceUnit=20 52

5.23 Local conflict ratio vs. GWriteProb, LResourceUnit=20 53

5.24 Local throughput vs. GWriteProb, LResourceUnit=1, LHotRegion=0.2 53

5.25 Local throughput vs. transaction length, LResourceUnit=20 . . 54

5.26 Local abort ratio vs. GWriteProb, LResourceUnit=20 54

5.27 Local abort ratio vs. transaction length, LResourceUnit=20 55

5.28 Global throughput vs. AlternativeProb, LResourceUnit=20 . . . 56

5.29 Global abort ratio vs. AlternativeProb, LResourceUnit=20 . . . 57

5.30 Global throughput vs. AlternativeProb, LResourceUnit=1 58

5.31 Global throughput vs. PreferenceProb, LResourceUnit=20 . . . 58

5.32 Global abort ratio vs. PreferenceProb, LResourceUnit=20 59

5.33 Global throughput vs. GWriteProb, precedence relation, LResourceUnit=20 59

5.34 Global abort ratio vs. GWriteProb, precedence relation, LResourceUnit=20 60

5.35 Global throughput vs. GWriteProb, precedence relation, LResourceUnit=1 60

5.36 Global throughput vs. RetriableProb, LResourceUnit=20 62

5.37 Global abort ratio vs. RetriableProb, LResourceUnit=20 62

5.38 Global throughput vs. CompensatableProb, LResourceUnit=20 63

5.39 Global abort ratio vs. CompensatableProb, LResourceUnit=20 63

5.40 Global throughput vs. GWriteProb, CTM, LResourceUnit=20 . 64

5.41 Global throughput vs. GWriteProb, OTM, LResourceUnit=20 . 65

5.42 Global abort ratio vs. GWriteProb, CTM, LResourceUnit=20 65

5.43 Global abort ratio vs. GWriteProb, OTM, LResourceUnit=20 66

5.44 Local throughput vs. AlternativeProb, LResourceUnit=20 66

5.45 Local throughput vs. PreferenceProb, LResourceUnit=20 67

5.46 Local throughput vs. GWriteProb, precedence relation, LResourceUnit=20 67

5.47 Local throughput vs. RetriableProb, LResourceUnit=20 68

5.48 Local throughput vs. CompensatableProb, LResourceUnit=20 . 68

5.49 Deadlock detection algorithms, global throughput vs. GWriteProb, LResourceUnit=20 69

5.50 Deadlock detection algorithms, global abort ratio vs. GWriteProb, LResourceUnit=20 69

5.51 Deadlock detection algorithms, global throughput vs. transaction length, LResourceUnit=20 70

5.52 Deadlock detection algorithms, local throughput vs. GWriteProb, LResourceUnit=20 71

5.53 Deadlock detection algorithms, local throughput vs. transaction length, LResourceUnit=20 71

5.54 Global throughput vs. GWriteProb, LResourceUnit=20 73

5.55 Global abort ratio vs. GWriteProb, LResourceUnit=20 73

5.56 Global throughput vs. GWriteProb, NumSites=16, LResource-Unit=20 74

5.57 Global abort ratio vs. GWriteProb, NumSites=16, LResource-Unit=20 74

5.58 Global throughput vs. GWriteProb, LResourceUnit=1 75

5.59 Global abort ratio vs. GWriteProb , LResourceUnit=1 75

5.60 Local throughput vs. GWriteProb, NumSites=16, LResourceUnit=20 76

5.61 Local throughput vs. GWriteProb, LResourceUnit=1 76

List of Tables

4.1 System model parameters 28

4.2 Local transaction model parameters 29

4.3 Global transaction model parameters 30

4.4 Parameter settings to obtain classical global transaction model . 31

4.5 Resource parameters 34

5.1 Performance metrics 38

5.2 System parameter settings 38

5.3 General Workload parameter settings 39

5.4 Variable workload parameter settings for classical transaction model experiments 40

5.5 Parameter settings for the analysis of the dependency relations 56

5.6 Parameter settings for the analysis of commit-independent transaction types 61

5.7 Global transaction parameter settings for the experiments that compare classical transaction model and extended transaction model 72

Chapter 1

Introduction

The recent progress in communication and database technologies has facilitated the sharing of information from different sources. The globalization of enterprises has also started to enforce existing information systems to cooperate with each other. As a result of these facts, a need has arisen for the integration of pre-existing database systems.

A multidatabase system (MDBS) is an integrated database system that provides a global view and uniform access to different local components without requiring the users to know the individual characteristics of the participant databases. Each local database system (LDBS) can have a different data model, and different transaction management and concurrency control mechanisms. Integration of those heterogeneous components should not violate the autonomy of LDBSs. This is the most important feature of MDBSs that distinguishes it from conventional distributed database systems.

Heterogeneity of the components in a MDBS leads to a requirement for flexible and powerful ways of accessing the data. The need for the coordination of the activities that belong to the independent data sources makes it difficult to adopt traditional transaction control methods in a MDBS environment. Since the control of a MDBS is totally dependent on LDBSs, designing a transaction model and control mechanism for MDBSs requires to consider LDBS transaction management functions.

Traditional transaction models developed for distributed database systems are quite restrictive for multidatabases. Traditional models generally assume a competition among transactions, but in a MDBS sometimes a cooperation besides the competition is also required for efficient processing of transactions. Defining and observing dependencies among the transactions executed over different sites can significantly affect the system performance. The variance among the execution time of transactions over different local DBMSs also forces the existing models to be reorganized accordingly. Also the properties like atomicity and isolation introduced by the traditional transaction model are sometimes inapplicable in a MDBS environment. Under all those considerations, we can safely argue that it is necessary to modify and extend existing distributed transaction models for MDBS environments.

In this thesis, we propose a new transaction execution model that captures the formalism and semantics of various extended transaction models and adopts them to a MDBS environment. The proposed model covers nested transactions [18], the flexible transaction model that provides various dependency types among subtransactions [24], and the model that involves a relaxed version of transaction atomicity, namely semantic atomicity, to increase the level of concurrency [9, 17]. While including the semantics of all those transaction models, the global serializability in our execution model was ensured through the use of the ticketing method [13].

We also describe a detailed simulation model of a MDBS to analyze the performance of the proposed transaction model. We discuss the performance implications of both the classical transaction execution model and the proposed execution model. We state the range of workloads and the system configurations under which our model can provide performance improvements over the traditional transaction model.

The remainder of this thesis is organized as follows. Chapter 2 discusses the previous work on MDBS transaction models and introduces our extended transaction model. Chapter 3 presents the execution architecture of the proposed transaction model and provides how global serializability and atomicity are achieved with this execution architecture. This chapter also includes detailed execution strategies for commit-independent transaction types. The global deadlock problem and its solutions are also discussed in this chapter.

Chapter 4 describes a simulation model of a MDBS and gives the implementation details of this model. Chapter 5 explains the simulation experiments carried out and presents the results obtained. In the experiments, first the performance implications of the ticketing method with the classical transaction model are studied; then the performance results with the extended transaction model are presented. Comparison of various global deadlock detection algorithms is also provided in this chapter. The experimental results that compare the classical transaction model with the extended transaction model are discussed at the end of the same chapter. Finally, Chapter 6 includes the concluding remarks.

Chapter 2

Transaction Model

2.1 Related Work

In the literature, a number of models that extend the traditional transaction model have been proposed. An extended transaction model called *Sagas* was introduced by Garcia-Molina and Salem [12] for the long-lived transactions to increase the level of concurrency. In this model, a global transaction consists of a number of subtransactions $S_1, S_2, ..., S_N$ and corresponding compensating transactions $C_1, C_2, ..., C_N$. In *Sagas*, either all of the subtransactions are committed, or partially committed ones are undone using their compensating transactions. Furthermore, the subtransactions do not need to see the same database state. If a subtransaction is ready-to-commit, it can be committed without waiting for the other subtransactions of the same transaction. Concurrency is increased in *Sagas* model in the expense of relaxing the isolation property. This model can be well suited for the environments where the subtransactions are relatively independent.

The *nested transaction* model introduced by Moss [18] is another alternative for the traditional transaction model. In the nested transaction model, flat transactions are enhanced by a hierarchical control structure. Each nested transaction consists of either primitive transactions or some nested transactions that are called subtransactions of the containing transaction. The whole transaction structure can be represented by a tree and the top-level transaction is called the root of the tree. The transaction that contains subtransactions is called a parent, and the subtransactions are called its children. In the nested transaction model, a child starts after its parent, and terminates before the parent terminates. The parent is not allowed to terminate before all of its child transactions are terminated. However, if a child is aborted, parent does not need to be aborted. A two level version of the nested transaction model has been adopted to a MDBS environment [7].

The transaction model developed for the Interbase MDBS [9] allows composition of flexible transactions. In this model, subtransaction failures can be discarded if a given function can be accomplished by more than one local database component. Furthermore, both compensatable and non-compensatable subtransactions can be defined within a multidatabase transaction. The execution dependencies among the subtransactions are specified as negative and positive dependencies. If a negative dependency is defined between subtransactions S_1 and S_2, it means that S_1 cannot start until S_2 fails. A positive dependency, on the other hand, means that S_1 cannot start until S_2 succeeds.

The DOM transaction model that has been developed at GTE laboratories is another possible model that can be applied to MDBSs [6]. DOM allows both nested and compensating transaction models to work together. This model also makes it possible to define contingency transactions which can be executed as an alternative when a subtransaction fails. The subtransactions can be classified as vital and non-vital in the DOM model. If a vital subtransaction aborts, its parent should also be aborted. However, if a non-vital transaction aborts, its parent may continue. Consequently, if a child transaction fails, the parent has the following alternatives: ignoring the condition, retrying the child transaction, executing a contingency subtransaction, or abort. The DOM model seems to be promising and should further be investigated in detail for MDBSs.

In a recent work by Rusinkiewicz, Kiychniak, and Cichocki [19], a multidatabase transaction, also called a global transaction, consists of subtransactions that can be in one of the following execution states: initial, executing, aborted, prepared to commit, or committed. Scheduling pre-conditions are associated with each subtransaction according to their execution states. In order to complete the execution of a global transaction, an acceptable termination state is defined using the execution states of its subtransactions. Although the model is powerful enough to express various types of transactions in a MDBS, an efficient scheduling of the subtransactions is a major problem.

In [17], the following types are defined for the subtransactions of a distributed transaction. A *compensatable* transaction can commit before its containing transaction commits, and if that transaction aborts, its effects on the local database can be undone by executing the associated compensating transaction. The *retriable* transactions are subtransactions that eventually succeed if they are retried a sufficient number of times. A retriable subtransaction can be allowed to commit later than its containing transaction. The compensatable and retriable transactions, which are also called *commit independent* transactions, reduce the blocking effects of the commitment protocols.

An integration of the *flexible* transaction model [10] and the transaction model of [17] has been proposed in [24] as an alternative model for MDBS transactions. The commit dependencies among the subtransactions have been investigated and a transaction execution model has been introduced to take the advantage of those dependencies in scheduling subtransactions. Two kinds of dependency relation, *precedence* and *preference*, have been examined in that work. The relaxed version of atomicity provided in [17] has also been extended to support flexible transactions.

2.2 Proposed Transaction Model

In this section, we introduce a new transaction model for MDBSs. In this model, we aim to capture the formalism and semantics of various extended transaction models and adopt it to a MDBS environment.

In a MDBS, two kinds of transactions can coexist: *local transactions* and *global transactions*. A transaction that is executed at a single site is called a local transaction. A global transaction on the other hand, can submit operations to multiple sites, and at each of those sites a subtransaction is executed on behalf of the global transaction. A subtransaction of a global transaction is not different from a local transaction from the LDBS point of view.

In a multidatabase environment, some subtransactions can be committed independent of its global transaction. If a subtransaction's effects on the database can be semantically undone by executing a compensating transaction, the subtransaction can be allowed to commit earlier. A subtransaction that reserves a seat in an airline reservation system is compensatable by a transaction that cancels the reservation. Another kind of commit independent transactions is the retriable transactions which eventually commit if they have been retried a sufficient number of times. A retriable transaction can be allowed to commit later. Crediting a bank account is an example of retriable transactions. Consequently, as stated in [17] and [24], the transaction type (TT) of a subtransaction can be either

- Compensatable (C),
- Retriable (R), or
- Ordinary (O) (neither compensatable nor retriable).

A formal definition for a subtransaction can now be provided:

Definition 1 *A subtransaction S is a 2-tuple S=(TT,CT) where*

- *TT is the transaction type of S;*
- *CT is the set of compensating transactions of S, if TT is compensatable (an empty set, otherwise).*

Since a global transaction is executed over multiple sites in the form of subtransactions, we cannot ignore the dependencies that can occur among the subtransactions. A possible dependency among subtransactions is the execution order dependency in which a subtransaction cannot be executed before some others complete their executions. That kind of dependency relation is often referred to as precedence relation between subtransactions. Another kind of dependency can be specified if some subtransactions are alternative of some others. In an alternative dependency, one of the functionally equivalent subtransactions needs to be executed. If the user assigns priority to alternative subtransactions, a preference relation exists among the subtransactions.

Definition 2 *Let S_i and S_j be two subtransactions. We define four types of dependency relation between S_i and S_j:*

- *Precedence relation (<), $S_i < S_j$ means that S_j cannot begin execution until S_i successfully finishes its execution.*

- *Alternative relation ($\diamond$), $S_i \diamond S_j$ means that S_j and S_i are alternative of each other and any of them can be executed. It is also possible to execute them together, but only one of them should be committed.*

- *Preference relation ($\triangleright$), $S_i \triangleright S_j$ means that among two alternative subtransactions S_i and S_j, S_i is preferred to S_j. If they are executed together, S_j can be committed only if S_i fails. If they are not allowed to execute together, S_i should execute first, and if it fails, S_j can be executed.*

- *No-dependency relation ($\square$), $S_i \square S_j$ means that S_i and S_j can execute independently.*

Now a formal definition of a global transaction can be provided as follows:

Definition 3 *A global transaction G is a 3-tuple G=(ST,DT,TO) where*

- *ST is the set of global transactions and/or subtransactions that are the children of G;*

- *DT is the dependency type among the transactions in ST;*

- *TO is the total order on ST according to the dependency specified in DT.*

A global transaction in our model is syntactically a nested transaction with extended semantics. A global transaction is a set of child transactions each of which is either a subtransaction or again a global transaction. This transaction model can be represented as a tree where the internal nodes are global transactions and the leaf nodes are subtransactions. The root of the tree is the overall global transaction. The level of a global transaction is not fixed at 2, but it can vary depending on the transaction complexity.

The transaction model introduced here is simply a mixture of three extended transaction models: the nested transaction model [18], the flexible transaction model that provides the dependency relations types [24], and the model described in [9, 17], that provides the commit independent transaction type. We can give some real-life examples to demonstrate the practicality of the proposed transaction model.

Example 1 *Consider a travel agent information system [9]. In this system, a transaction may consist of the agent's negotiation with airlines for the flight ticket, the negotiation with car rental companies for car reservation, and the negotiation with hotels to reserve rooms. Assume for a given trip that the only airlines available are Northwest and United, the only car rental company is Hertz, and the only hotels in the destination city are Hilton, Sheraton and Ramada. The agent can order a ticket from either Northwest or United, but Northwest is preferred, a car is mandatory for the trip and any of the three hotels is suitable for the customer needs. Further, only the reservation of the hotel room can be canceled. The following subtransactions can be defined for a global transaction that should be executed for this application:*

- S_1: *Order a ticket at Northwest airlines;*
- S_2: *Order a ticket at United airlines;*
- S_3: *Rent a car at Hertz;*
- S_4: *Reserve a room at Hilton;*
- S_5: *Reserve a room at Sheraton;*
- S_6: *Reserve a room at Ramada;*
- S_7: *Cancel a room reservation at Hilton;*
- S_8: *Cancel a room reservation at Sheraton;*
- S_9: *Cancel a room reservation at Ramada.*

The global transaction G_{trip} can be specified as follows:

$$\begin{aligned}
G_{trip}(ST = \{G_{airlines}(ST = \{\ & S_1(TT = O, CT = \{\}),\\
& S_2(TT = O, CT = \{\})\},\\
DT = {} & Preference,\\
TO = {} & S_1 \triangleright S_2),\\
S_3(TT = O, CT = \{\}),\\
G_{hotel}(\quad ST = \{\ & S_4(TT = C, CT = \{S_7\}),\\
& S_5(TT = C, CT = \{S_8\}),\\
& S_6(TT = C, CT = \{S_9\})\},
\end{aligned}$$

$$\begin{aligned} &DT = Alternative, \\ &TO = S_4 \diamond S_5 \diamond S_6)\}, \\ DT &= No - dependency, \\ TO &= G_{airlines} \Box S_3 \Box G_{hotel}) \end{aligned}$$

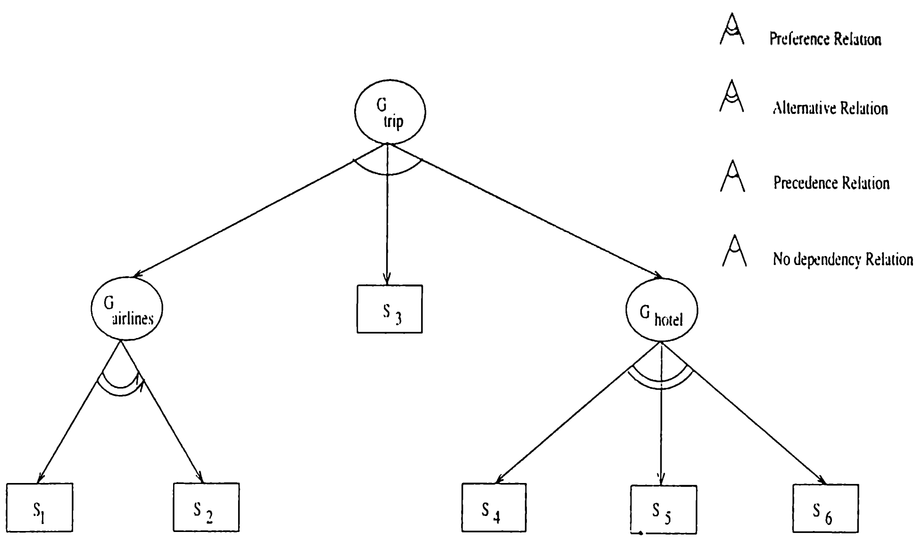

Figure 2.1: A transaction tree representation of Example 1.

Example 2 *Consider a banking system [24]. A client at bank b_1 wants to withdraw \$50 from his account a_1, and deposit it in his friend's account a_2 in bank b_2. If this is not possible he wants to withdraw \$50 from his account a_3 in bank b_3 and deposit it in a_2. We assume that depositing an account is always successful if it is retried sufficient number of times. The subtransactions can be described as follows:*

- S_1: *Withdraw \$50 from account a_1 in bank b_1;*
- S_2: *Deposit \$50 in account a_2 in bank b_2;*
- S_3: *Withdraw \$50 from account a_3 in bank b_3;*
- S_4: *Deposit \$50 in account a_1 in bank b_1;*
- S_5: *Deposit \$50 in account a_3 in bank b_3.*

A global transaction $G_{transfer}$ can be constructed using these subtransactions:

$$
\begin{aligned}
G_{transfer}(ST = \{G_{withdraw}(ST = \{\ & S_1(TT = C, CT = \{S_4\}),\\
& S_3(TT = C, CT = \{S_5\})\},\\
& DT = Preference,\\
& TO = S_1 \triangleright S_3),\\
& S_2(TT = R, CT = \{\}),\\
DT = Precedence,&\\
TO = G_{withdraw} < S_2)&
\end{aligned}
$$

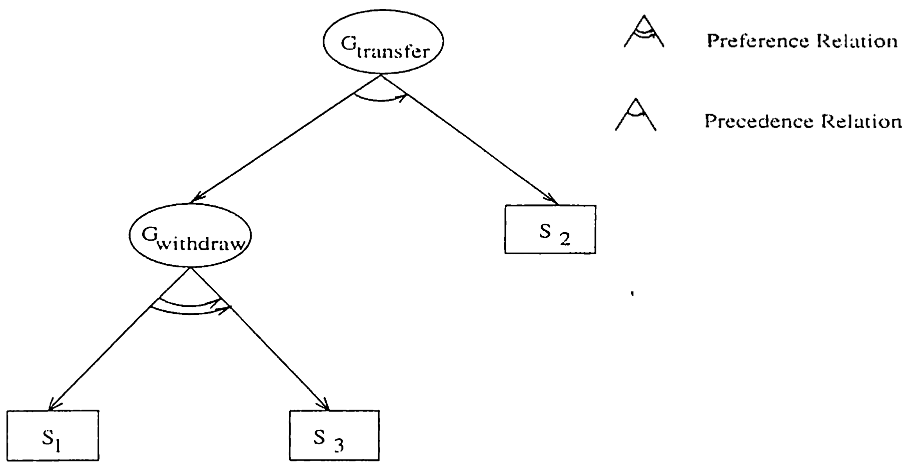

Figure 2.2: A transaction tree representation of Example 2.

Chapter 3

Execution Architecture

Due to the autonomy of local sites, the local transactions are directly submitted to the LDBSs, while global transactions use a common MDBS interface. The execution of local transactions is controlled by the local transaction manager (LTM) that exists at each site, and the execution of global transactions is controlled by the global transaction manager (GTM).

The objectives of GTM are to avoid inconsistent retrieval of data, and to preserve global consistency and atomicity [13]. These objectives are difficult to achieve, because:

- Local database systems are not aware of each other and the MDBS.
- Both local and global transactions can run concurrently at each site.
- LTMs do not export any concurrency control information to GTM.
- From the local database systems' viewpoint, a global subtransaction is not different from a local transaction.

LTM at each site ensures the local consistency and isolation properties by generating serializable schedules. GTM can achieve the global serializability by coordinating the participant LDBSs. Global serializability can be provided by obtaining the information of relative serialization order of subtransactions at each local site and guaranteeing the same relative order at all those sites [19]. Achievement of global serializability is difficult, because the execution order of global subtransactions may not be consistent with the serialization order due to the local transactions. Even though the global subtransactions do not conflict with each other at a particular site, local transactions can cause indirect conflicts among them.

In the literature, several approaches have been proposed to solve this problem. Some of those approaches suggest a relaxed version of the global serializability, like quasi serializability [8], and two level serializability [16], which can maintain global consistency in restricted applications. Some other approaches assume that the local serialization information can be available to some degree, and propose some techniques based on this assumption. Another solution to the global concurrency control problem is to assume conflicts among global subtransactions whenever they are executed at the same site, but this method has some drawbacks due to its low degree of concurrency.

The ticketing method proposed in [13] seems to be the first method to show successfully that the serialization order of global subtransactions at a local site can be determined at the global level without violating the autonomy of that site. The ticketing method uses a regular data object, called a ticket, to determine the serialization order of global subtransactions. A ticket in a database can be seen as a logical timestamp. One ticket value is maintained at each local site. Multidatabase concurrency controller forces each subtransaction read, increment and update the ticket value at the site it executes. The ticket value obtained by a subtransaction reflects the relative serialization order at that site. This approach eliminates the effects of indirect conflicts generated by local transactions even if the multidatabase system cannot detect their existence.

Accomplishing the atomicity of global transactions is another problem in MDBS transaction management. In traditional distributed database systems, atomicity can be achieved by using the two phase commit (2PC) protocol. In a MDBS, due to the heterogeneity of local components, we can not expect every participant site to support 2PC. One possible solution to this problem is using a simulated 2PC protocol. An additional set of application programs called agents can be built on top of each site to establish this necessary protocol. The agents in our model are responsible for controlling the execution of subtransactions that are sent to its site. The MDBS architecture assumed for this model is given in Figure 3.1.

Now, we can start discussing proposed execution model. We assume that

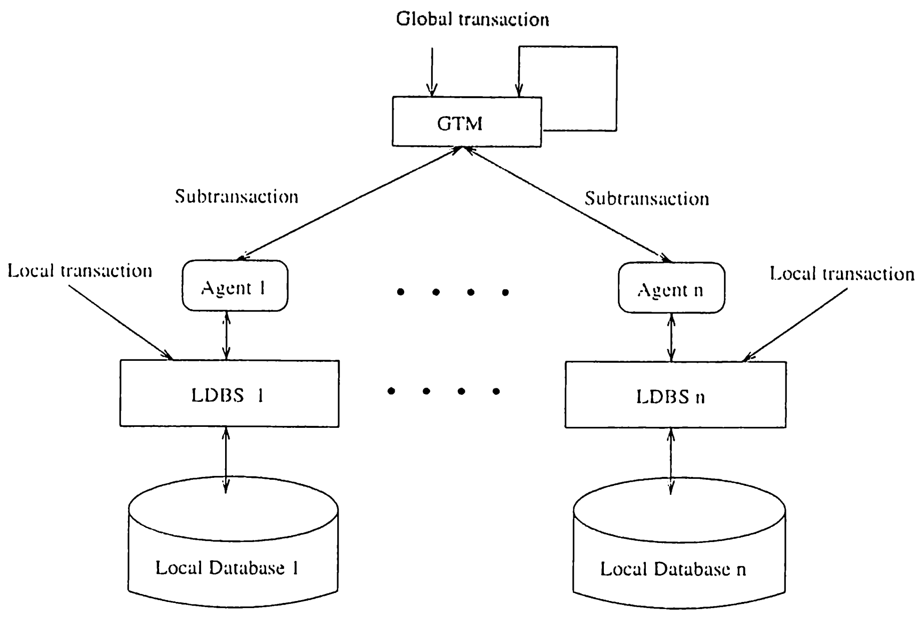

Figure 3.1: The architecture of the MDBS

each global transaction has at most one subtransaction at each local site. We also assume that a local transaction or a subtransaction consists of four basic operations: $r(x)$, $w(x)$, c, and a. $r(x)$ and $w(x)$ are read and write operations on data item x, and c and a are commit and abort operations. Similar to the execution model presented in [19] ,the execution state of a transaction can be one of the following types:

- Initial (I),
- Executing (E),
- Ready-to-commit (R),
- Committed (C),
- Aborted (A).

A transaction is assumed to be in the ready-to-commit state after it completes all of its read and write operations. It stays in this state until a commit or an abort operation is issued.

We have to reconsider the concepts of the global serializability and the atomicity of a global transaction to establish the correctness of our execution model.

3.1 Ensuring Global Atomicity

In a MDBS environment, a relaxed version of atomicity, namely *semantic atomicity*, has been discussed in [24, 17]. In traditional atomicity, a global transaction can be atomic if either all or none of its subtransactions complete their execution successfully. However, in semantic atomicity subtransactions can commit at different times. A global transaction can commit if all of its subtransactions commit; otherwise the effects of committed subtransactions are undone, and global transaction is aborted. We need to extend this definition to capture the semantics of dependency relations among subtransactions. The execution of a global transaction G preserves the semantic atomicity, if the following conditions are satisfied:

- When a precedence or a no-dependency relation exists among its children, G can commit if all of its child transactions commit. If one of its child transactions is aborted, G is aborted and the other child transactions are either aborted or the effects of committed ones are undone.
- If an alternative or preference relation exists, G can commit if one of its child transactions commits[1]. When a child transaction is committed, other child transactions that are executing are aborted.

The execution of a global transaction containing only ordinary children[2] proceeds as follows:

- First, the global transaction is constructed with the initial execution state.
- GTM spawns the children of the global transaction according to the specified dependency type:
  - If either a no-dependency, or an alternative dependency, or a preference dependency exists, all of the child transactions are created.
  - Otherwise (if a precedence relation was specified), the children are created on the basis of the given total order.

[1]Remember that, with the preference relation, if $S_i \triangleright S_j$, then S_j can be committed only if S_i fails.

[2]The execution of a global transaction that can have commit independent (compensatable/retriable) transactions is described in Section 3.3.

- If GTM reaches a leaf node in the nested transaction tree and creates a subtransaction, it submits the subtransaction to the corresponding site through the agents.
- When a subtransaction finishes its database operations, the agent of that site sends a ready-to-commit message to the GTM.
- After receiving a ready-to-commit message for a subtransaction, GTM checks the dependency type associated with the parent of the subtransaction to find out what to do next.
  - If a precedence relation exists among its children, the next child transaction in the given order is created by the GTM. If all of the child transactions enter the ready-to-commit state, the parent also enters the ready-to-commit state.
  - If an alternative relation exists, the parent enters the ready-to-commit state and GTM sends messages to the relevant agents to abort the other child transactions.
  - If a preference relation exists, the parent enters the ready-to-commit state if the completed subtransaction is the most preferred one. When the parent becomes ready-to-commit, GTM broadcasts the abort message for the other child transactions.
  - If a no-dependency relation exists, the execution state of the parent becomes ready-to-commit after all of its children enter the ready-to-commit state.
- If the root transaction reaches the ready-to-commit state, GTM decides to commit or abort the transaction according to the concurrency control algorithm executed.
- After a commit or abort is issued for the root transaction, GTM broadcasts a message to child transactions down to the leaves of the transaction tree to commit or abort the subtransactions at local sites.

As we can understand from this execution protocol, the ready-to-commit messages are sent in a bottom-up fashion, from leaf transactions to the root transaction, and the commit or abort messages are transmitted in a top-down fashion from the root transaction to leaf transactions. By that way, atomic commitment of a global transaction is ensured.

3.2 Ensuring Global Serializability

A global schedule S is serializable if each local restriction of S is serializable, and there is a total order O on the global transactions in S such that in each local schedule of S, the serialization order must be consistent with the order O [11]. We need to have additional requirements that the serialization order of child subtransactions in S must be consistent with the serialization order of their parent global transaction. Specifically, it is sufficient to guarantee that the relative order of subtransactions in local sites is the same as their parents' order in O.

To provide concurrency control, we can apply the ticketing method which ensures the serialization of global transactions. The ticket values obtained by subtransactions are transferred to their parents up to the root transaction. GTM ensures the same relative serialization order at all sites of the global root transaction using the ticket values obtained. Two possible methods that can be used for concurrency control are the optimistic ticketing method (OTM), and the conservative ticketing method (CTM) [13].

3.2.1 Employing The Optimistic Ticketing Method (OTM)

OTM allows subtransactions of global transactions to be executed as soon as they are submitted to the local sites. A global transaction is committed when all of the tickets obtained by its subtransactions have the same relative order in all participant LDBSs. If OTM is adopted, a global transaction G is processed as follows:

- First, a time-out period is set for G (for the detection of a potential deadlock).
- The GTM spawns the child transactions of G according to the rules given above up to the subtransactions executed at local sites.
- Subtransactions are allowed to execute under the control of agents until they become ready-to-commit.
- When G enters the ready-to-commit state, it is validated by GTM.

- If the validation of G is successful, it is committed; otherwise, it is aborted and then restarted.

GTM uses a global serialization graph (GSG) to validate the commitment of transaction G. GSG is a directed graph which contains nodes for recently committed global transactions. For any pair of recently committed global transactions G_i and G_j, there is a directed edge $G_i \rightarrow G_j$, if G_i obtained a smaller ticket value than G_j at a site they were executed together.

A global transaction G in the ready-to-commit state can be validated as follows:

- First, a node is created for G in GSG.
- Then GTM attempts to insert an edge between G and other nodes in GSG.
- If G has obtained a smaller (larger) ticket value than a recently committed global transaction G_c at a site, an edge $G \rightarrow G_c$ ($G_c \rightarrow G$) is inserted.
- If all such edges can be added to GSG without creating a cycle, G is validated.
- Otherwise, the node for G and all related edges are removed from the graph, and G is aborted.

A validation can be performed on GSG either,

- when a global child transaction becomes ready-to-commit (i.e., *early validation*), or
- when a global root transaction becomes ready-to-commit (i.e., *late validation*).

The aim of early validation is to detect the conflicts among global transactions as early as possible and to minimize the global transaction restarts. If a global child transaction fails in GSG test, GTM can abort that transaction. If a preference or an alternative relation exists among the transactions that belong to the same parent, GTM can execute an alternative transaction for the failed child transaction. If a no-dependency relation or a precedence relation exists, GTM restarts the aborted global child transaction.

If the validation test for a global root transaction is successful, a commit message is transmitted to its children. Otherwise, an abort message is sent to its children and the entire global transaction is restarted. To remove a node for a committed global transaction G from GSG, the following properties should be satisfied [13]:

- The node has no incoming edges.
- The transactions that were active when G was committed have all been terminated.

3.2.2 Employing The Conservative ticketing method (CTM)

CTM was introduced to eliminate the global restarts experienced by OTM due to the ticketing conflicts. CTM controls the order in which the subtransactions take their tickets. In order to apply CTM, we need an additional ready-to-take-a-ticket state for both global transactions and subtransactions. A subtransaction enters the ready-to-take-a-ticket state after it completes all of the database operations before obtaining its ticket value. The agents over the local sites are responsible to detect ready-to-take-a-ticket states of subtransactions and send appropriate messages to GTM. Similar to the ready-to-commit messages, the ready-to-take-a-ticket messages are also sent from leaves of a transaction tree up to the root to provide atomicity in obtaining a ticket value. CTM processes a set of global transactions as follows:

- Initially a time-out period is set for each global transaction.
- Subtransactions are allowed to execute under the control of LDBSs until they enter the ready-to-take-a-ticket state.
- A ready-to-take-a-ticket message is transmitted up to the global root transaction, according to the execution rules specified for the ready-to-commit message in Section 3.1.

- Global transactions in ready-to-take-a-ticket state are allowed to take their tickets according to the order in which they enter the ready-to-take-a-ticket state. If a global transaction G_1 becomes ready to take a ticket before transaction G_2, G_1 is assigned a smaller ticket value than that of G_2.

- A global transaction that enters the ready-to-commit state is committed by GTM. If the time-out of the transaction expires before it is committed, the transaction is aborted and restarted.

3.3 Commit Independent Subtransactions

Before the description of the execution model for commit independent subtransactions, let us specify the necessary assumptions and restrictions for the underlying MDBS environment:

- There should be no value dependencies among the commit independent subtransactions.

- If a compensating transaction is initiated, it completes successfully [15].

The commit independent transaction type was proposed to minimize the blocking effect of the 2PC global atomic commitment protocol. If a child subtransaction commits before its parent, it is called an *early committed subtransaction*. Similarly, if a child subtransaction commits after its parent it is a *late committed subtransaction*. Compensatable subtransactions can be early committed, and retriable subtransactions can be late committed. To achieve semantic atomicity with commit independent transactions, the following conditions should hold for a global transaction G [17]:

- If G is aborted, the effects of early committed subtransactions of G on the database are not seen by other transactions.

- if G is committed, the effects of its late committed subtransactions are seen by the transactions serialized after G.

Consequently, for a compensatable subtransaction S with its compensating transaction CS, if the parent of S is aborted, commitment of S is required to be undone by executing CS. The effects of committed subtransactions are not seen, if no other subtransaction is serialized between the S and CS [17]. Therefore, if GTM ensures that no other subtransaction takes its ticket before the commitment of CS, consistency of the MDBS is preserved.

The compensating transaction execution is handled by agents. If a global transaction G has a compensatable subtransaction S with its associated compensating transaction CS, the execution of S is provided as follows when OTM is being used:

- CS is sent to the relevant agent with the submission of S.
- When S enters the ready-to-commit state,
  - The agent sends a ready-to-commit message to GTM.
  - The ticket value obtained by S is recorded and S is early committed by the agent.
- The agent sends an abort message for the other subtransactions that has obtained greater ticket value than S before a commit message arrives for S.
- If the agent receives an abort message for S, it submits CS to LDBS.
- The agent sends an abort message for the other subtransactions that has obtained greater ticket value than S before CS is committed.

If CTM is being used for concurrency control:

- CS is sent to the relevant agent with the submission of S.
- When S enters a ready-to-take-a-ticket state, the agent sends a ready-to-take-a-ticket message to GTM.
- When a take-a-ticket message arrives for S, the agent does not permit other subtransactions to enter their ready-to-take-a-ticket states until S takes its ticket.

- If S successfully takes its ticket and completes all its operations, the agent early commits S and sends a ready-to-commit message for S to GTM.
- The agent does not allow other subtransactions to take their tickets until a commit or an abort is issued for S.
- If an abort is issued for S, the agent submits CS to the LDBSs and does not submit any other subtransaction operation until CS is committed.

In the case of retriable transactions, the global transactions do not see an inconsistent database, if GTM avoids serialization of any subtransaction between the commitment of a global transaction and the commitment of a retriable subtransaction that belongs to the committed global transaction [17]. A global transaction G that contains a retriable transaction RS can be committed without waiting RS to finish its execution. GTM can commit G, while RS is still being executed at a site, but it does not permit another subtransaction to take a ticket at that site until RS takes its ticket.

If OTM is being used, the protocol handling the execution of retriable subtransaction RS of G can be described as follows:

- The state of RS is made ready-to-commit before GTM submits it to the relevant agent. Therefore, the commitment of G does not require RS to be completed.
- If G enters the ready-to-commit state before a ready-to-commit message arrives for RS, a $+\infty$ ticket value is used for G in GSG test. Since the RS has not taken its ticket yet, the $+\infty$ ticket value in GSG test ensures that no other subtransaction is serialized between the commitment of G and the commitment of RS.
- When RS is committed, the agent sends a commit message to GTM in order to update the ticket value of G.

If CTM is being used, agents are responsible for the correct execution of retriable transactions. They simply do not allow other subtransactions to take ticket until RS successfully commits. The extension to the standard execution model, for a retriable transaction RS, can be described as follows:

- Once the agent receives a take-a-ticket message for RS, it does not send ready-to-take-a-ticket messages for other subtransactions executed at that site until RS takes its ticket successfully.
- GTM makes the state of RS ready-to-commit after it sends a take-a-ticket message for it.
- Once GTM issues a commit for RS, the agent does not submit ticketing operation of other subtransactions to the LDBS until RS is committed.

3.4 The Global Deadlock Problem

In a MDBS environment, the global deadlock problem can occur if the LDBSs employ a lock-based concurrency control. The local deadlock detection can be assumed to be handled by local schedulers. GTM may not be aware of global deadlock situations since the LDBSs may not export any information about local deadlocks. Using a time-out strategy for a global transaction is the easiest way to detect global deadlocks. However, the time-out value set for global transactions significantly effects the throughput of the system. Using a too small time-out value for a global transaction may result in unnecessary global transaction aborts, while a too large time-out value may result in blocking of deadlocked transactions for a long time.

Especially in our transaction model, it is difficult to estimate expected execution time of a global transaction due to the extended semantics. One possible time-out mechanism that can be adopted is to set a time-out for the entire global transaction. This method cannot be very effective, because the number of subtransactions executed in a global transaction can vary according to the dependencies specified among the subtransactions. Instead of using a global time-out value for the entire global transaction, we can estimate the subtransaction execution at each site and calculate the time-out period of a global transaction as follows:

- If child transactions are submitted concurrently, in other words, the dependency type among the child transactions is either no-dependency or alternative dependency or preference dependency, then the time-out value for the parent transaction can specified to be the maximum time-out value of its children.

- If the dependency type among its children is the precedence dependency, the time-out value for the parent transaction is the sum of the child transactions' time-out values since the children will be executed serially.

Another solution to the global deadlock problem is to apply a deadlock detection algorithm. One of the deadlock detection algorithms proposed in [5] is based on the Potential Conflict Graph (PCG). PCG is a directed graph where the nodes in the graph are the global transactions that have at most one child transaction that is waiting to obtain lock. The edge $G_i \rightarrow G_j$ exists in PCG, if G_i's subtransaction is in the waiting state and G_j's subtransaction is in the executing state at the same site. A cycle in PCG represents a potential global deadlock. A similar algorithm is employed to estimate the global deadlocks occurring due to ticket waiting. In our PCG algorithm, an edge is inserted between G_i and G_j, if G_i is in the executing state and G_j is in the ready-to-take-a-ticket state. The PCG algorithm executed for our transaction model is as follows:

- A timestamp and a time-out value is assigned to global transaction G when it is submitted to the system.
- If one of the children of G enters the ready-to-take-a-ticket state, a node for G is created and related edges are inserted into PCG.
- If the time-out of G expires, PCG is checked for cycles including G.
- If G appears at least in one cycle then,
  - If it has the smallest timestamp value among the transactions in the same cycle it continues to execute with a reinitiated time-out.
  - Otherwise, if the dependency type among its subtransactions is the alternative or preference dependency, only the child transaction of G that causes the cycle in PCG is aborted. In all other cases, G is aborted.
- If G enters the ready-to-commit state, its node and incident edges are removed from the graph.

Chapter 4

Simulation Model

4.1 Introduction

In this chapter, we study the global concurrency control problem of MDBSs from the performance point of view. In the literature, most of the researchers have concentrated on the serializability problem of global transactions and developed various concurrency algorithms for MDBSs. However, the performance implication of MDBS transaction management and the cost of transaction processing in a MDBS environment have not been investigated in detail.

In a work by Hung et al. [14], a performance analysis of various optimistic and pessimistic global concurrency control algorithms has been provided. They have made an analysis of throughput, response time and abort ratio of both LDBSs and MDBSs. In their model, they assumed that all the LDBSs apply strict two-phase locking (2PL) for local concurrency control. Their performance results heavily depend on the local concurrency control algorithm. A general multidatabase simulation model has also been proposed in [14].

Schaad, Schek and Weikum have compared the transaction processing that uses 2PC protocol with distributed multi-level transaction management [20]. They have developed a prototype implementation of a MDBS system. Strict 2PL is also the concurrency control algorithm of LDBSs in their work. They have performed an analysis of average transaction response time under the various workloads. The effect of global concurrency control mechanisms on the performance of LDBSs and the performance impact of local transactions' behavior on the global transaction response time have not been considered in their model.

If we assume each LDBS applies a rigorous concurrency control algorithm like strict 2PL in which all the data locks obtained by a transaction are released together when the transaction commits or aborts [3], the global serializability can be easily achieved by controlling the commitment order of subtransactions [1, 13]. Consequently, if all of the LDBSs employ strict 2PL, the global concurrency control problem is reduced to detect possible global deadlocks. As a result, the performance of a MDBS transaction management system also depends on how the global deadlock problem is handled. In the literature, Scheuermann, Tung and Teng have studied the performance of two global deadlock detection algorithms [21]. They have compared the potential conflict graph (PCG) [5] approach with transaction-blocked-at site graph (TBSG) algorithm [22]. In another work by Baldoni and Salzo, the performance of PCG algorithm and simple global time-out method has been studied in a MDBS environment where participant LDBSs employ only strict 2PL algorithm [2].

In our work, we have focused on the performance analysis of OTM and CTM algorithms based on the proposed extended transaction model. Performance of the ticketing methods has not been investigated by anyone even with the classical transaction model [21, 13, 14] yet. Therefore, first a performance implication of OTM and CTM algorithms on the classical transaction model has been investigated in detail, and then their performance with the extended transaction model has been studied. We have also analyzed and compared the performance results of various global deadlock detection algorithms suitable for the proposed execution model. Finally, experiments that compare the extended transaction model with the classical transaction model have been performed. In our performance study, we did not restrict participant LDBSs to employ only strict 2PC concurrency control algorithm as the others did. We made the followings assumptions and simplifications about the MDBS simulation model, as we focus only on the performance of the global transaction model and the concurrency control algorithms.

- No communication or site failures occur, consequently the recovery related issues are ignored in the simulation model.
- A centralized version of MDBS where GTM resides at one site is implemented since the proposed algorithms work on the centralized MDBSs.

- LDBSs can abort a transaction that executes at its site only due to local deadlock detection algorithm.
- LDBSs notify the transaction programs when unilaterally abort a transaction. This means that MDBS will be aware of subtransaction aborts at local sites.
- LDBSs permit serializable and recoverable schedules.
- Subtransactions have a visible ready-to-commit state.

In the next section, we present a detailed simulation model of an MDBS which is similar to the one provided in [14].

4.2 MDBS Simulation Model

A MDBS in our system is a closed network with one global site and a fixed number of local sites. GTM resides at the global site alone as a server to global clients. All of the global transaction requests are submitted to the GTM interface. We also assume that only one LDBS resides at each local site. A global transaction agent (GTA) is also built on top of each LDBS. GTAs are responsible for submitting global subtransactions to the corresponding LDBSs; as well as communicating with the GTM. Both local clients and GTAs submit their requests to LDBS interfaces. The architecture of our MDBS is illustrated in Figure 4.1.

The parameters describing the MDBS model are listed in Table 4.1. The multiprogramming level of the MDBS is the maximum number of global transactions that GTM can process at a time. To keep the *global multiprogramming level* (GMPL) constant throughout the simulation, global clients submit their requests one after another. The *local multiprogramming level* (LMPL) of each LDBS is the number of local transactions plus the number of global subtransactions submitted to that site. The local clients also submit their requests one after another to keep the local transaction load constant. At local sites, the minimum LMPL is LNumClient, and the maximum LMPL is (LNumClient + GNumClient). The size of the local database is assumed to be constant for each site and LDBsize represents LDBS size in pages. *Hot region* is the part of the database which is accessed most frequently. LHotRegion parameter can

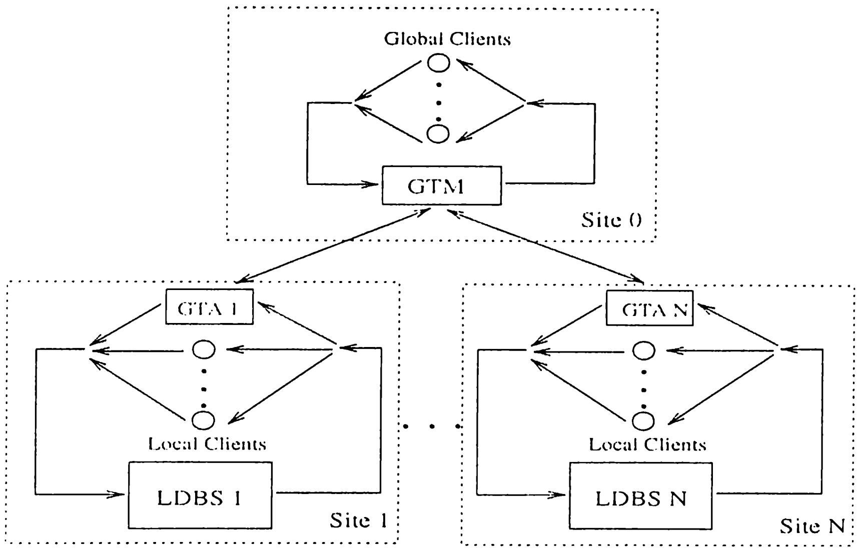

Figure 4.1: MDBS closed network model

be in between 0 and 1, and used to specify the ratio of the hot region in the local database. The ratio of the hot region is also assumed to be the same for every LDBS.

System Parameters	Meaning
NumSites	Number of local sites
GNumClient	Number of global clients
LNumClient	Number of local clients at each site
LDBsize	Size of each LDBS in pages
LHotRegion	Ratio of the hot region in LDBSs

Table 4.1: System model parameters

4.2.1 Transaction Model

In a MDBS environment, two kinds of transactions affect the performance of the system: local and global transactions. A local transaction contains read and write operations on data pages. It can be modeled in a simulation environment with few basic parameters. Local transaction model parameters are described in Table 4.2.

A Local transaction size can vary between LTranMinSize and LTranMaxSize

Local Transaction Parameters	Meaning
LTranMinLen	Minimum local transaction length in pages
LTranMaxLen	Maximum local transaction length in pages
LHotAccessProb	Local transaction hot region access probability
LWriteProb	Local transaction update probability
LThinkTime	Local clients' think time in seconds
LRestartTime	Restart time of an aborted local transaction

Table 4.2: Local transaction model parameters

with uniform distribution. LHotAccessProb is used determine local transactions' access probability to the hot region. LWriteProb represents the probability of update operation. LThinkTime parameter is used to model the client think time between consecutive local transactions. When a local transaction finishes its execution, a new one is submitted to the LDBS after LThinkTime seconds.

The global transaction model is more complex due to its semantics. As discussed in the previous sections a global transaction can be modeled as a tree where the internal nodes are global transactions and the leaf nodes are subtransactions. In the simulation model, a global transaction is characterized by the Height and NumChild parameters. These parameters represent maximum values for a global transaction and vary depending on the experiment types. In the experiment that compares the classical model with the extended model, a global transaction is modeled as a full tree. Each subtransaction, like a local transaction, contains several read and write operations on data pages. Ticketing operations of a subtransaction are assumed to be read and a write operations on specific pages. A representation of a global transaction is provided in Figure 4.2.

The global transaction parameters used in our simulation model are listed in Table 4.3. TreeHeight parameter represents the maximum height of the tree and NumChild parameter represents the maximum number of children at each internal node. The maximum number of subtransactions executed in a global transaction is then $(NumChild)^{TreeHeight}$. The number of subtransactions executed in a global transaction can vary according to the dependencies among the subtransactions. Since we assume that at most one subtransaction

Global Transaction Parameters	Meaning
TreeHeight	Maximum height of a global transaction tree
NumChild	Maximum number of children in each global transaction
NoDependencyProb	Probability of no-Dependency relation
PrecedenceProb	Probability of precedence relation
PreferenceProb	Probability of preference relation
AlternativeProb	Probability of alternative relation
OrdinaryProb	Probability of ordinary subtransactions
CompensatableProb	Probability of compensatable subtransactions
RetriableProb	Probability of retriable subtransactions
GTranMinLen	Minimum global subtransaction length in pages
GTranMaxLen	Maximum global subtransaction length in pages
GHotAccessProb	Global subtransaction hot region access probability
GWriteProb	Global subtransaction update probability
GValRestartTime	Global transaction restart time after GSG aborts
GTimeoutRestartTime	Global transaction restart time after timeout aborts
GRestartTime	Global transaction restart time after local site aborts
GThinkTime	Global clients' think time in seconds

Table 4.3: Global transaction model parameters

of a global transaction can be executed at each site, the number of subtransactions executed also determines the number of local database sites that a global transaction may access.

The dependencies among the children of a global transaction is determined by the probability of each dependency type. NoDependencyProb, PrecedenceProb, PreferenceProb and AlternativeProb represent the distribution of dependencies among the children of global transactions. To analyze the effects of compensating and retriable transactions, OrdinaryProb, CompensatableProb, and RetriableProb parameters are defined. Those probabilities represent on the average the ratio of subtransactions' types in the overall global transaction.

GTranMinLen and GTranMaxLen parameters are defined to determine the subtransaction length. Similar to the local transaction parameters, GHotAccessProb and GWriteProb represent the access probabilities of global subtransactions at local sites. GThinkTime parameter is also defined to model the think

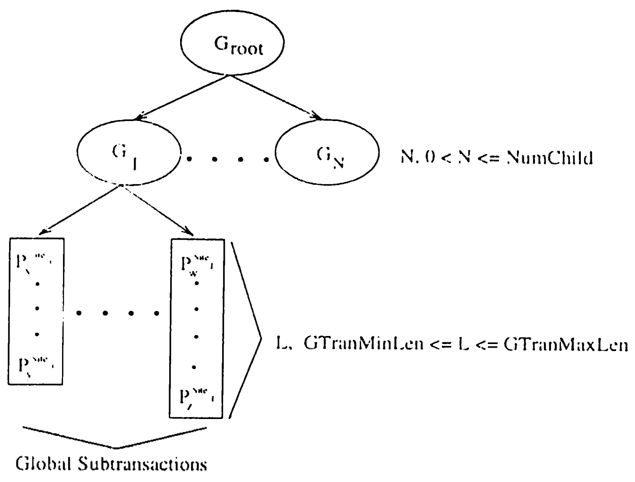

Figure 4.2: Global transaction model

time of a global client. Restart time of aborted global transactions is determined according to the abortion type. GValRestartTime, GTimeoutRestartTime, and GRestartTime specify the restart time after a validation, timeout, and local site aborts, respectively.

We can obtain the classical global transaction model of [14], [13] by setting the parameter TreeHeight to 1 and NumChild to the number of subtransactions in a global transaction. Table 4.4 provides parameter settings to simulate the classical global transaction model.

Transaction Parameters	Settings
TreeHeight	1
NumChild	Number of subtransactions
NoDependencyProb	1.0
PrecedenceProb	0.0
PreferenceProb	0.0
AlternativeProb	0.0
OrdinaryProb	1.0
CompensatableProb	0.0
RetriableProb	0.0

Table 4.4: Parameter settings to obtain classical global transaction model

4.2.2 Simulation Model Components

Figure 4.3 illustrates the simulation model components. The model is flexible to integrate different kinds of concurrency control algorithms and database components for investigating various aspects of the system. The details of each component in the system can be described as follows:

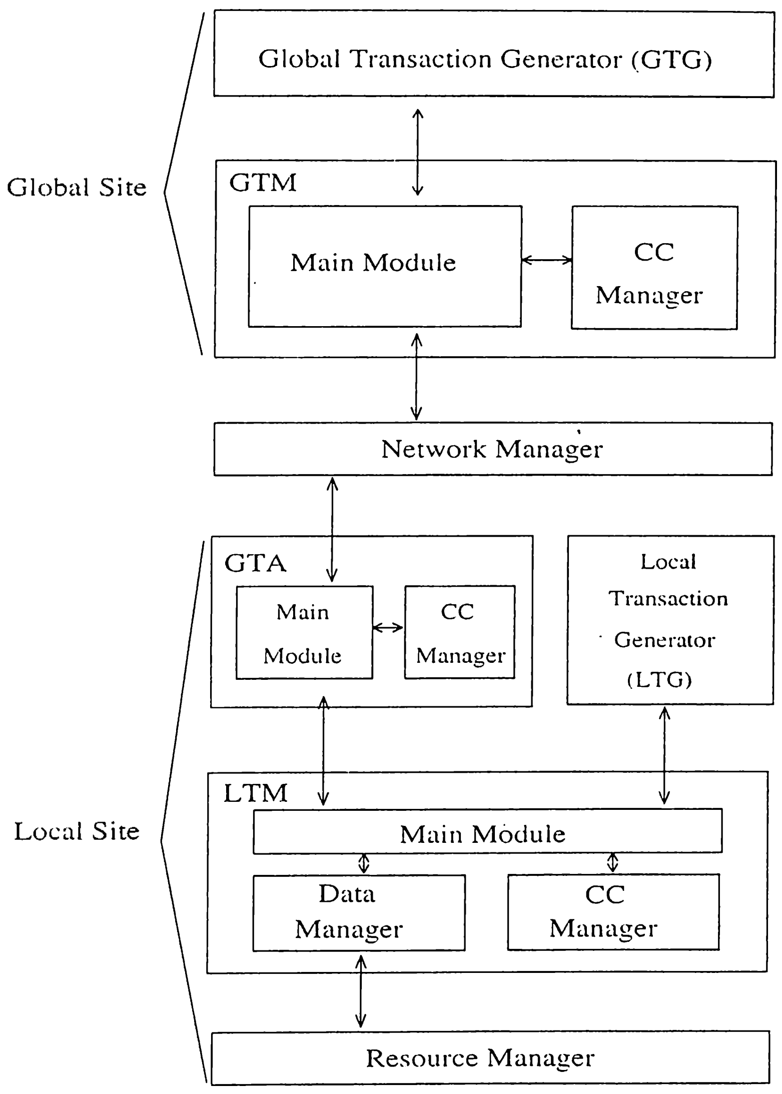

Figure 4.3: Simulation Model Components

- Global Transaction Generator (GTG) : GTG resides at the global site and simulates the global client behavior by generating global transactions using the parameters in Table 4.3. At the beginning of the simulation, GNumClient global transactions are created and submitted to the GTM. During the simulation, a new transaction can only be created after the termination of a global transaction. For the last set of experiments, this component also converts an extended global transaction to a set of classical global transactions.

- Global Transaction Manager (GTM) GTM accepts global transactions from GTG and models their execution. It consists of 2 modules:
  - Main Module: This module models the transaction execution with the help of Concurrency Control (CC) manager. There are two main jobs of this module. First, it accepts global transactions and decomposes them to their subtransactions executed at each local site according to the rules in Section 3. Second, it establishes 2PC protocol with agents and coordinates in-coming and out-going messages for subtransactions. When a global transaction enters the ready-to-commit state it decides commitment or abortion of that transaction by communicating with the CC manager.
  - Concurrency Control (CC) Manager : CC manager models the execution of a concurrency control algorithm for serialization of global transactions. It also performs the deadlock detection, if it is necessary. This module enables us to plug-in different global concurrency control and deadlock detection algorithms for performance studies.
- Local Transaction Generator (LTG) : This component simulates the local client behavior and generates local transactions using the parameters in Table 4.4. Like GTG, it submits a new transaction when one of the previously submitted local transactions completes its execution.
- Global Transaction Agent (GTA): GTA resides at each local site and models the execution of global subtransactions at that site. Similar to GTM, it consists of two modules.
  - Main Module : GTA Main Module is responsible for controlling submission of subtransactions at its site. It determines the submission time of a subtransaction's operations with the help of the CC manager and the messages coming from GTM. It behaves like a local client for LDBS. Submission time of the ticketing operation is also determined in GTA main module according to the local concurrency control algorithm. GTA submits the ticketing operations at the end, if LTM of its site applies the 2PL algorithm [13]. GTA main module also handles the submission of compensating subtransactions with the coordination of CC manager.

  - CC Manager : It is the local agent part of the concurrency control algorithm implemented in GTM. It carries out the global concurrency control for subtransactions at its site.

- Network Manager : It models the network resource between the GTM and GTA. We defined MessTransTime parameter to simulate the time duration to transmit a message between GTM and GTA. MessTransTime is assumed to be the same for each local site. We also assumed that a subtransaction is transmitted in one MessTransTime. CPUMessTime is defined to simulate CPU message coding-decoding time.

- Local Transaction Manager (LTM) : LTM accepts and models the execution of local transactions and subtransactions. It consists of three modules.

  - Main Module : Main module models the local transaction execution with the help of CC manager and data manager.

  - CC manager : CC manager models the local concurrency control algorithm as well as the local deadlock detection algorithm.

  - Data manager : Data manager models the data accessing and processing by interacting with the resource manager and the main module of LTM.

- Resource manager : This component models the CPU and disk accesses at its site.

Resource Parameters	Meaning
CPUMessTime	CPU message coding-decoding time in seconds
MessTransTime	Message transmission time in seconds
LResourceUnit	Number of resource units at each site
LCPUTime	CPU time for processing one page
LDiskTime	Disk time for read/write of one page
GSGTime	Execution time of GSG algorithm in OTM

Table 4.5: Resource parameters

Disk and CPU resource parameters are included in Table 4.5. A resource unit is modeled as one CPU and two disks as in [1]. Each site has equal number of resources which is determined with the LResourceUnit parameter. We also employ a parameter to model the execution time of a global serialization graph algorithm in OTM.

Chapter 5

Simulation Experiments

5.1 Introduction

Our performance model has been implemented on a simulation testbed using the CSIM simulation package from MCC [23]. We can categorize our experiments in four sections:

- Experiments on the classical transaction model.
- Experiments on the extended transaction model.
- Experiments to evaluate global deadlock detection algorithms based on the extended model.
- Performance comparison of classical transaction model with the extended transaction model.

Before providing the details of each experiment, we will first discuss the algorithm settings, performance metrics, and general parameter settings.

5.1.1 Algorithm Settings

In our performance model, we do not restrict local concurrency control algorithms to be either strict or cascadeless. Since we have aimed to cover a wide range of local concurrency control algorithms, we have employed the basic 2PL concurrency control algorithm in each LDBS, which is not restricted to strict or cascadeless schedules. Using a locking-based local concurrency control algorithm also gave us a chance to model and compare the global deadlock detection algorithms besides evaluating OTM and CTM. We have implemented the transaction wait-for graph algorithm [3] to handle local deadlock situations. In the performance comparison of OTM and CTM algorithms on the classical global transaction model, we have implemented a simple global time-out method. In this method, the time-out period was calculated using the formula below [14]:

$$TimeoutPeriod = GlobalResponseTime + 2 * std(GlobalResponseTime)$$

where *GlobalResponseTime* is the average response time of a global transaction, and std(*GlobalResponseTime*) is the standard deviation of the global response time. *GlobalResponseTime* changes dynamicly during the simulation and reflects the on-line estimation of global transaction execution time.

For the extended transaction model, we have also implemented the extended global time-out method mentioned in Section 3.4. In the extended time-out method, time-out period of each subtransaction was also calculated using the same formula given above by replacing *GlobalResponseTime* with subtransaction response time.

5.1.2 Performance Metrics

Primary and secondary performance metrics employed in the experiments are provided in Table 5.1. Our experimental results present the mean values of the performance measures. We looked only the statistically significant performance differences for the evaluation of performance results. In all experiments, throughputs are measured over a long simulation time periods. The response times are measured between the transaction submission time and the transaction commitment time. Local throughputs are computed as the averages of all local sites. Local throughputs and response times are measured to investigate the effects of global transactions on local transactions. Since the global aborts are composed of validation aborts, global time-out aborts, and local sites aborts, we also examined the each category of aborts separately. Compensated transaction ratios are analyzed to comment on the compensatable transaction

Performance Metrics	Meaning
Global/local throughput	Number of global/local transactions completed per second
Global/local response time	Average response time measured between the global/local transaction submission time and completion time
Global/local abort ratio	Total number of global/local transaction aborts over the total global/local transactions submitted to the system
Global/local conflict ratio	Total number of global/local access conflicts over the total number of global/local access requests
Global/local blocking time	Average global/local transaction waiting time per page request
Resource Utilization	Fraction of time that the disk resources are busy

Table 5.1: Performance metrics

execution in the extended transaction model. False global deadlock ratio was measured in the performance analysis of deadlock detection algorithms. We examined the disk I|O utilization to determine the rate of resource contention.

System Parameters	Settings
NumSites	8 sites
LDBSize	1000 page per site
CPUMessTime	0.02 seconds
MessTransTime	0.05 seconds
LCPUTime	0.10 seconds
LDiskTime	0.20 milliseconds

Table 5.2: System parameter settings

5.1.3 General Parameter Settings

System parameter settings can be seen in Table 5.2. The parameters are common in all performance experiments unless specified otherwise. The local database size was set to 1000 pages to create high levels of conflicts in the system. Number of sites was set to 8 to have a reasonable number of sites for the extended transaction model. All the other system parameter values chosen

Parameters	Settings
GNumClient	20 clients
LNumClient	30 clients
LHotRegion	0.5
LTranMinLen	8 pages
LTranMaxLen	8 pages
LHotAccesProb	0.5
LWriteProb	0.25
LThinkTime	0.0 second
LRestartTime	LResponseTime
TreeHeight	1, 3
NumChild	2
GTranMinLen	8 pages
GTranMaxLen	8 pages
GHotAccessProb	0.5
GWriteProb	0.25
GThinkTime	0.0 second
GValRestartTime	0.0 second
GTimeoutRestartTime	GResponseTime
GRestartTime	GResponseTime
GSGTime	0.0 second
LResourceUnit	1, 20

Table 5.3: General Workload parameter settings

are similar to those used in [1], [14] to be able to obtain competable results with the previous performance studies.

The general workload parameter settings are listed in Table 5.3. Those are the standard parameter settings of all experiments. Their variations will be given with the description of the relevant sections. The general experiments were performed on both low and high resource contentions by setting LResourceUnit to 20 and 1, respectively. It was observed from the simulation results that, 20 resource units were enough to avoid resource contention. The local, global transaction lengths and write probabilities were selected to create reasonable number of transaction conflicts. Those values are also similar to those used in the previous performance models. For the classical transaction model experiments we set ThreeHeight to 1. For the other experiments, we set it to 3.

Our preliminary experiment results showed that an adaptive restart delay depending on the observed average response time is the best for the aborted transactions. For an aborted global transaction, we set the validation abort delay to 0, since the global transaction already completed all its operations before being validated. Therefore there is no need to wait for resubmission. The preliminary experiment results also confirm that no delaying for validation aborts provides the best performance. However, for the timeout and local site aborts, an adaptive delay based on the committed global transaction response time performs slightly better than zero delay especially on high data conflict situations. An adaptive delay of one global response time period gave less global timeout and local abort ratios. Therefore, we chose to employ a dynamicly changing restart delay value for the timeout and local site abort situations of global transactions. In addition, these restart times do not affect the performance in low abort and data conflict ratios. We did not employ parameters related to the hot region in standard experiments, since we did not want to create high levels of data conflicts in those experiments.

5.2 Experiments on the Classical Transaction Model

In the following experiments we compare OTM and CTM algorithms using the classical transaction model. The simple global time-out method is employed for global deadlock detection. We vary one of the parameters given in Table 5.4 at each experiment, and examine the performance results.

Variable Parameters	Settings
LWriteProb	0.0, 0.25, 0.50, 0.75, 1.0
GWriteProb	0.0, 0.25, 0.50, 0.75, 1.0
GTranMinLen *GTranMaxLen*	2, 4, 8, 12, 16 pages 2, 4, 8, 12, 16 pages

Table 5.4: Variable workload parameter settings for classical transaction model experiments

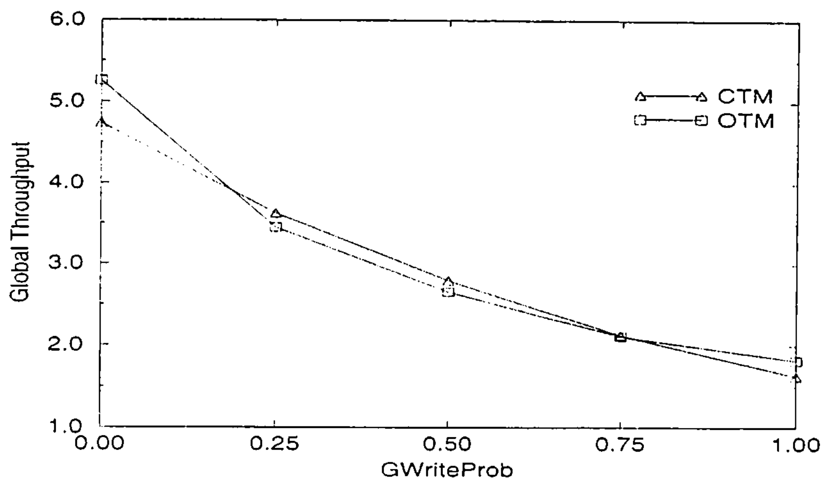

Figure 5.1: Global throughput vs. GWriteProb, LResourceUnit=20

5.2.1 The Impact of Data Contention

We have examined the effects of data contention in both high and low resource contention situations. The variation in the data contention was achieved by changing the value of GWriteProb. First, we set LResourceUnit to 20 to isolate the effects of resource contention. Figure 5.1 shows the global throughput of the two algorithms. When the data contention is very low, OTM performs better than CTM since the global blocking and conflict ratios are low. This situation minimizes the variation of subtransaction completion time which also decreases the validation aborts. Further, since we isolate the resource contention, under low conflict ratios the throughput loss from re-submission of aborted transactions is at the minimum values which is in favour of OTM. On the other hand, if we do not ignore the execution time of the GSG algorithm, OTM loses its performance advantage to CTM even for the low data conflict ratios. Figure 5.2 compares the global throughput of the two algorithms when GSGTime is set to 0.05 second.

As the data contention increases, OTM is no longer the winner, because of the high validation aborts. Finally, when the data contention is very high CTM suffers from global and local deadlocks. The reason for this result is that a subtransaction that completes its data operations has to wait its siblings to enter the ready-to-take-a-ticket state before taking its ticket and releasing its locks. Hence the blocking times for both subtransactions and local transactions

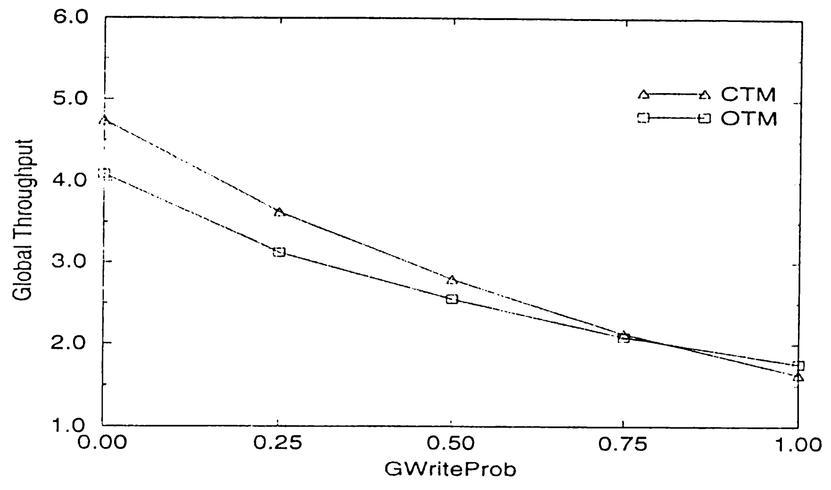

Figure 5.2: Global throughput vs. GWriteProb, LResourceUnit=20. GSG-Time=0.05

increase which reduces the CTM throughput. On the other hand, OTM submits the ticketing operation immediately after the subtransaction completes its data operations. Therefore, the global conflict ratio and the blocking times are smaller with OTM. Figures 5.3 and 5.4 confirm the OTM's lower global blocking times and conflict ratios under high data contention.

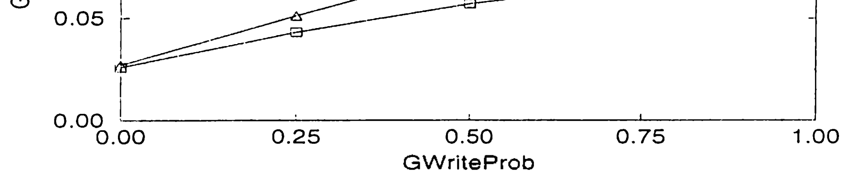

Figure 5.3: Global blocking time vs. GWriteProb, LResourceUnit=20

Figure 5.5 illustrates the total global abort ratios of OTM and CTM. OTM has a higher abort ratio due to validation aborts. Figures 5.6 and 5.7 give the individual abort ratios of two algorithms respectively. The high abort ratio of OTM algorithm is mainly due to local and validation aborts. At higher

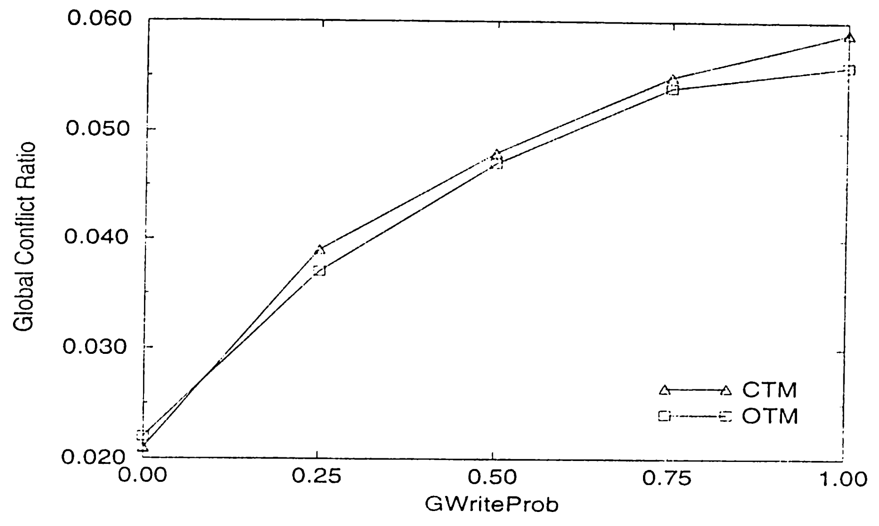

Figure 5.4: Global conflict ratio vs. GWriteProb, LResourceUnit=20

data conflicts, the local abort ratio dominates the other abort ratios for both algorithms. CTM has higher timeout and local abort ratios compared to OTM.

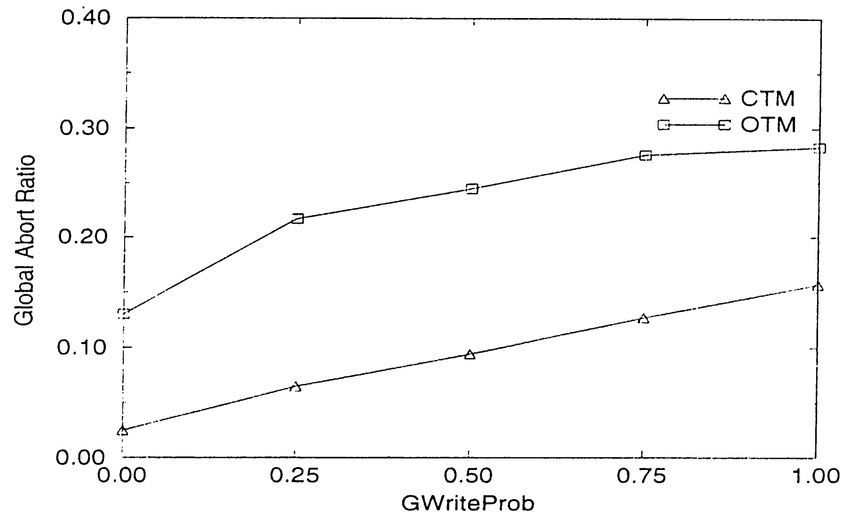

Figure 5.5: Global abort ratio vs. GWriteProb, LResourceUnit=20

CTM's poor performance under high data conflicts becomes more clear when we repeat the same experiment with the setting of LHotRegion to 0.2. Figure 5.8 shows the comparison of the global throughput ratios when the data contention among transactions is increased with the new setting of LHotRegion. For the GWriteProb values that are greater than 0.25, CTM faces a great performance loss due to very high rate of local and timeout aborts. We can conclude that CTM is much more sensitive to data contention than OTM. Figure 5.9 compares the global abort ratios of two algorithms.

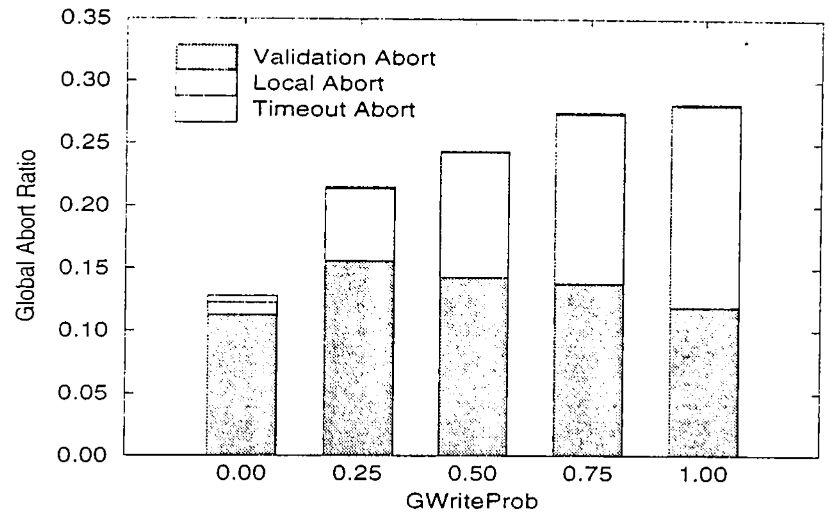

Figure 5.6: Global aborts of OTM, LResourceUnit=20

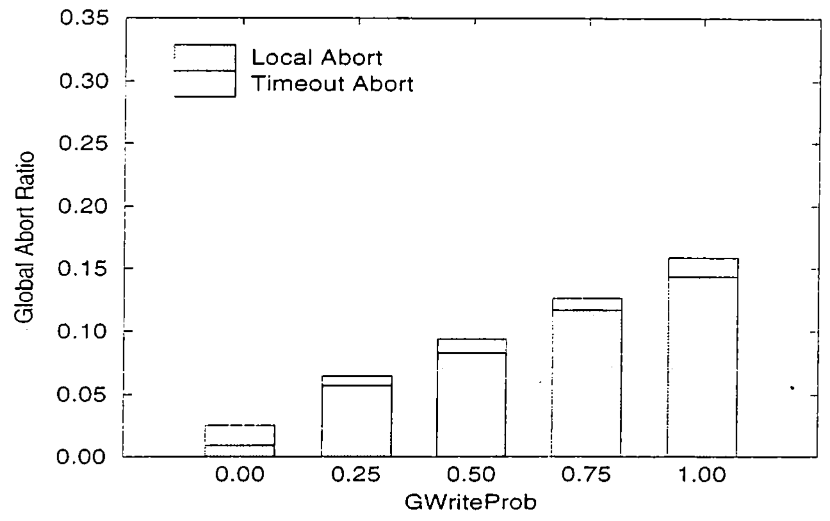

Figure 5.7: Global aborts of CTM, LResourceUnit=20

OTM's problem with large amount of validation aborts is more visible when we look at the situation where the number of children in a global transaction is 4. As we can see from Figure 5.10, OTM performs very bad since around half of the submitted transactions are aborted. Figure 5.11 shows the global abort ratio versus update probability with 4 children. It is very difficult to achieve the situation that every child subtransaction is serialized at the same order when the number of subtransactions in a global transaction increases. CTM is the algorithm of choice when the number of sites that a global transaction accesses is large. But again with high data conflict rates, CTM's performance drops rapidly as a result of the increase in the total global blocking time and conflict ratio. This situation also increases the local aborts of global transactions, because the local abort probability of a global transaction is higher with a greater number of subtransactions.

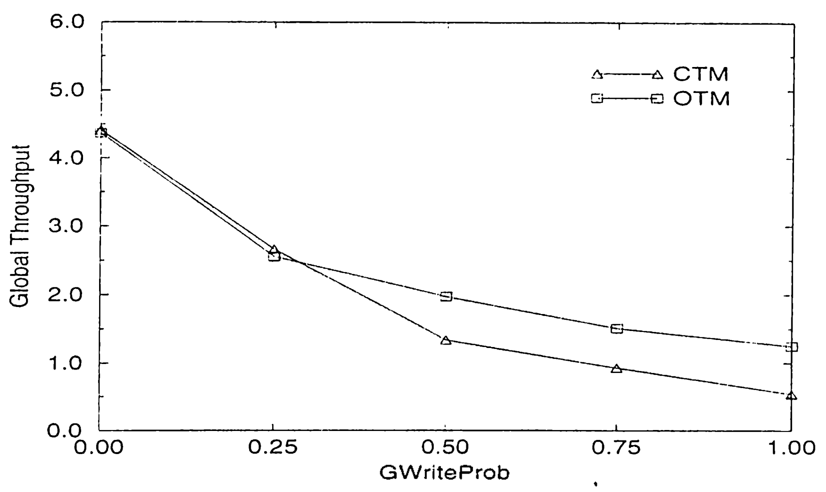

Figure 5.8: Global throughput vs. GWriteProb. LResourceUnit=20. LHotRegion=0.2

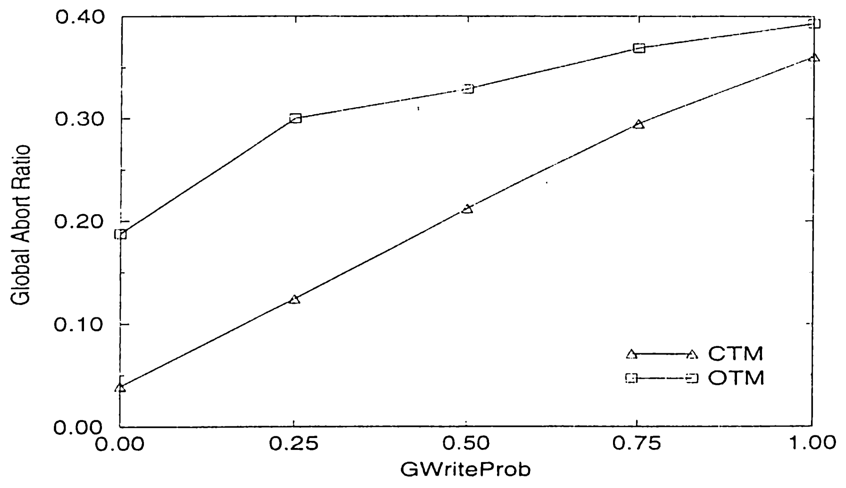

Figure 5.9: Global abort ratio vs. GWriteProb, LResourceUnit=20, LHotRegion=0.2

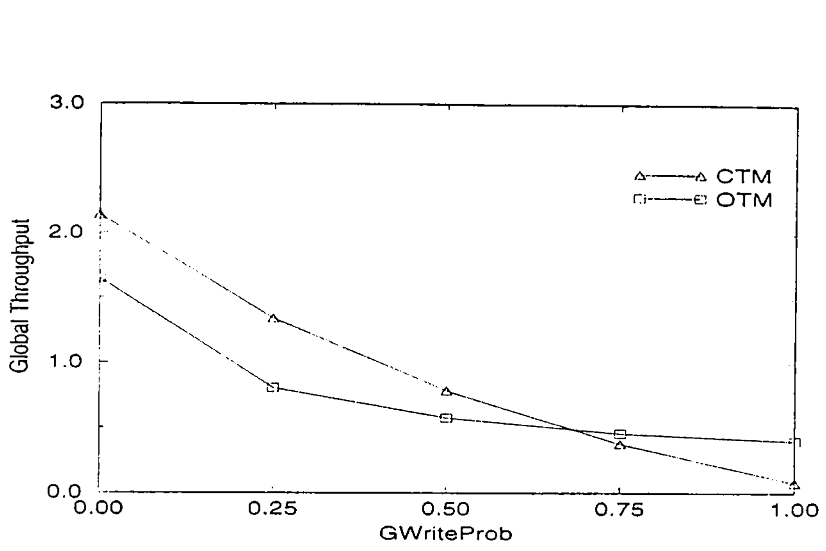

Figure 5.10: Global throughput vs. GWriteProb, NumChild=4, LResourceUnit=20

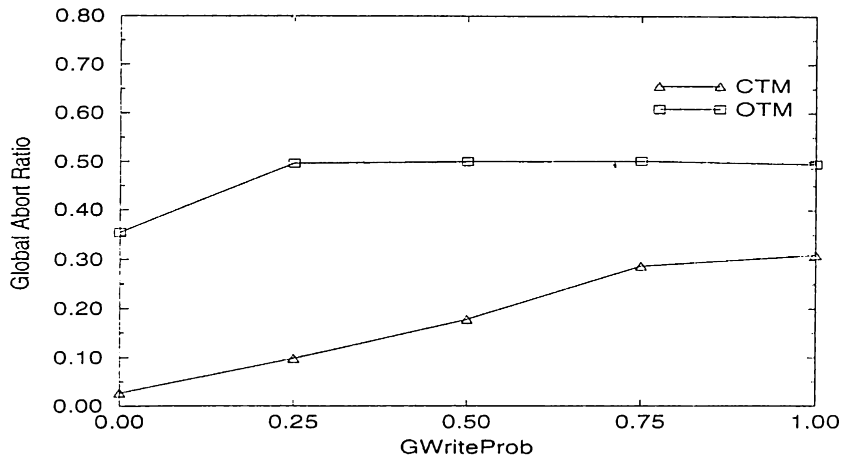

Figure 5.11: Global abort ratio vs. GWriteProb, NumChild=4, LResourceUnit=20

5.2.2 The Impact of Resource Contention

In this experiment, we set LResourceUnit to 1 and by changing the update probability, we examine the effects of data contention under the condition that a resource contention also exists. Figures 5.12 compares global throughput values of the two algorithms. Although CTM throughput decreases faster than OTM as the data contention increases, CTM performs well in the overall. Under high resource contentions, the cost of aborting a transaction is higher, consequently validation aborts significantly affect the global throughput of OTM.

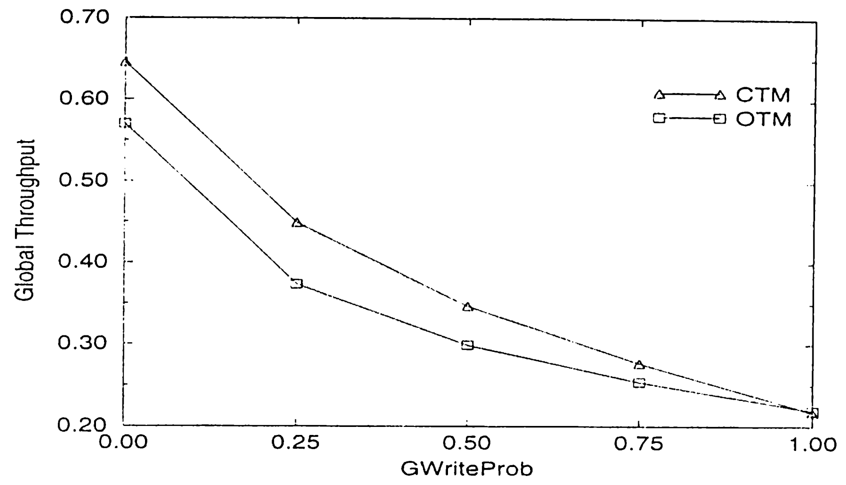

Figure 5.12: Global throughput vs. GWriteProb, LResourceUnit=1

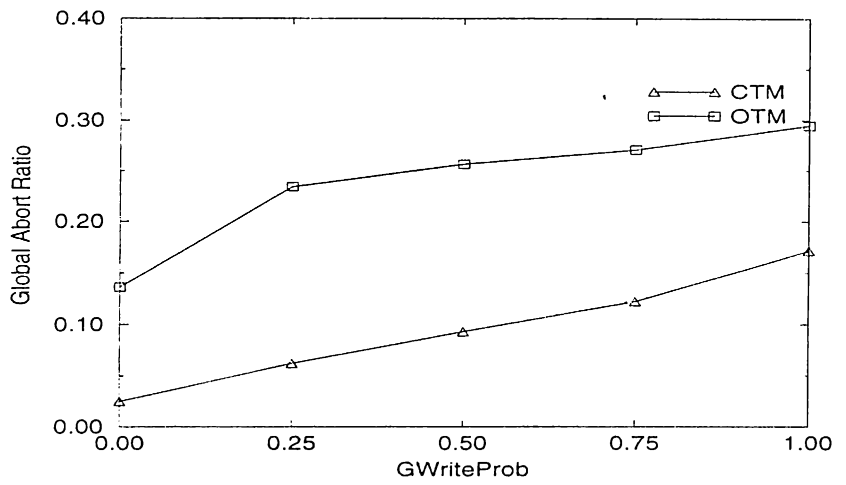

Figure 5.13: Global abort ratio vs. GWriteProb, LResourceUnit=1

The abort ratios of OTM and CTM are plotted in Figure 5.13. The trends of the abort ratios are nearly the same as the situation where there exists no resource contention (Figure 5.5). In addition to this, as the data contention increases, we observed slightly higher abort ratios under high resource contention.

When we compare OTM and CTM by setting NumChild to 4 in Figure 5.14, CTM's superior performance under low resource contention is noticed clearly. Waste of resources due to restarts negatively affects the OTM's performance. OTM has a chance to perform better only when the CTM's throughput sharply decreases with the high rate of data conflicts.

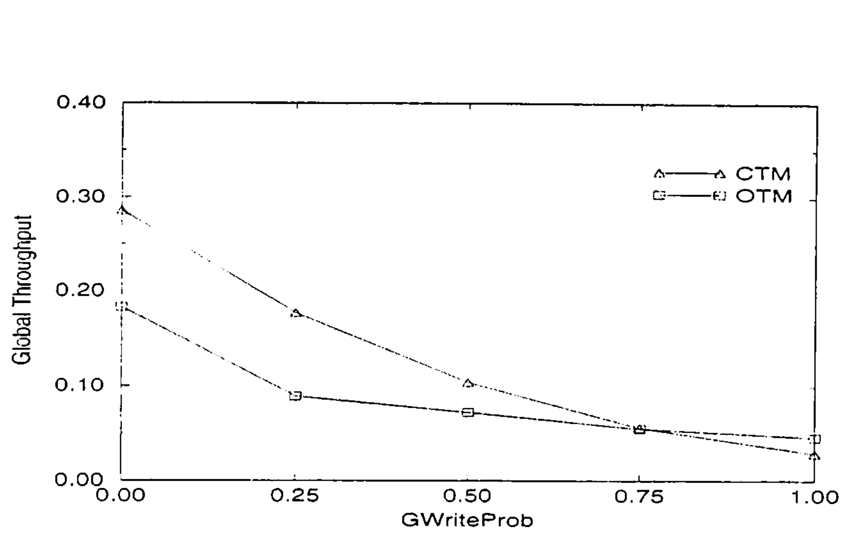

Figure 5.14: Global throughput vs. GWriteProb, NumChild=4, LResourceUnit=1

5.2.3 The Impact of Transaction Length

In this experiment, the performance of two algorithms was measured by varying the subtransaction length from 2 pages to 16 pages. Again the standard workload parameter values in Table 5.3 were used throughout this experiment.

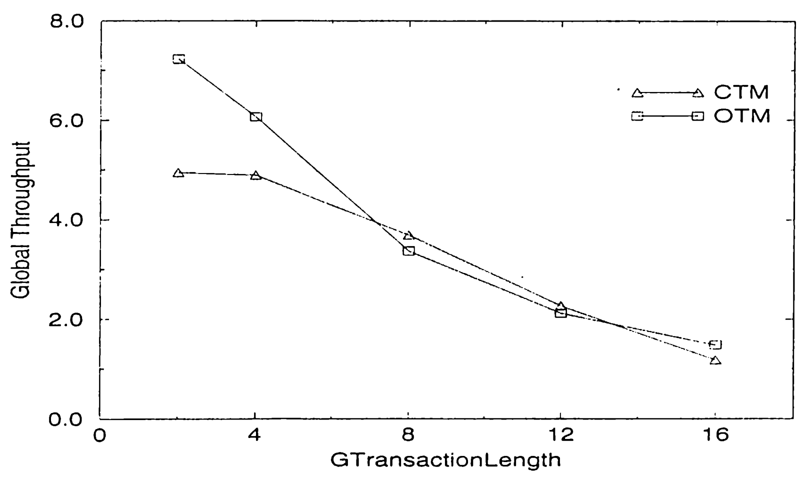

Figure 5.15: Global throughput vs. transaction length, LResourceUnit=20

Figure 5.15 shows the throughput of the algorithms under low resource contention. When the system has no resource problem and the subtransaction length is smaller than 8, OTM performs better. This result is not surprising since under low resource contention the response time of the global transactions is short; consequently, restarts with OTM have a little effect on the throughput.

On the other hand, since CTM is conservative about ticketing time, response time of a global transaction becomes larger due to the extra waiting time before ticketing. As the subtransaction length increases, OTM's throughput decreases sharply below CTM's throughput. Finally, with very large transaction lengths, in addition to the transaction execution time, the data access conflict ratio also increases which makes CTM behave worse. Explanation of this follows that of Figure 5.1.

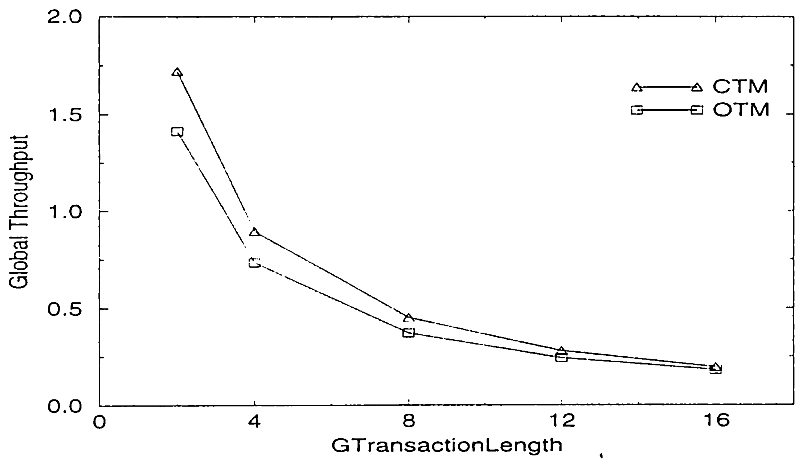

Figure 5.16: Global throughput vs. transaction length, LResourceUnit=1

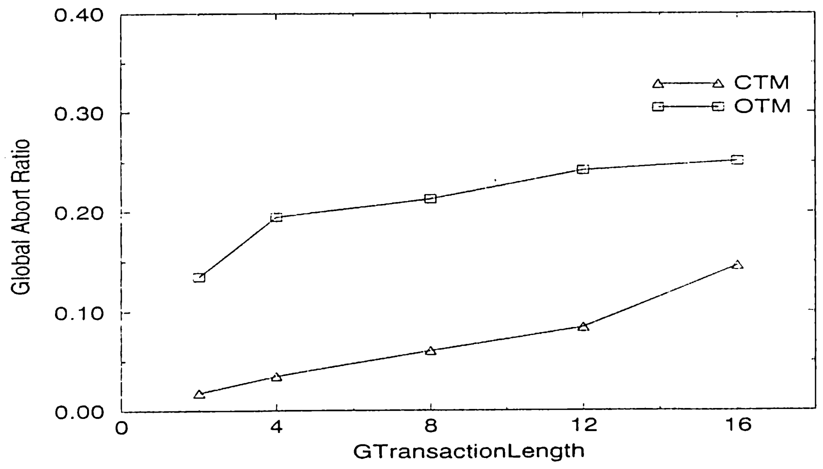

Figure 5.17: Global abort ratio vs. transaction length, LResourceUnit=20

When we look at the high resource contention situation in Figure 5.16, OTM losses its advantage with the increase in resource costs. CTM produces higher throughput rate than OTM in the overall, although it suffers from long transaction waiting as the subtransaction length increases.

Figures 5.17 and 5.18 illustrate the total global abort ratios on low and high resource situations. OTM has an higher abort rate in both figures as expected due to its GSG aborts. The abort rate of both algorithms slightly increases as a function of the subtransaction length.

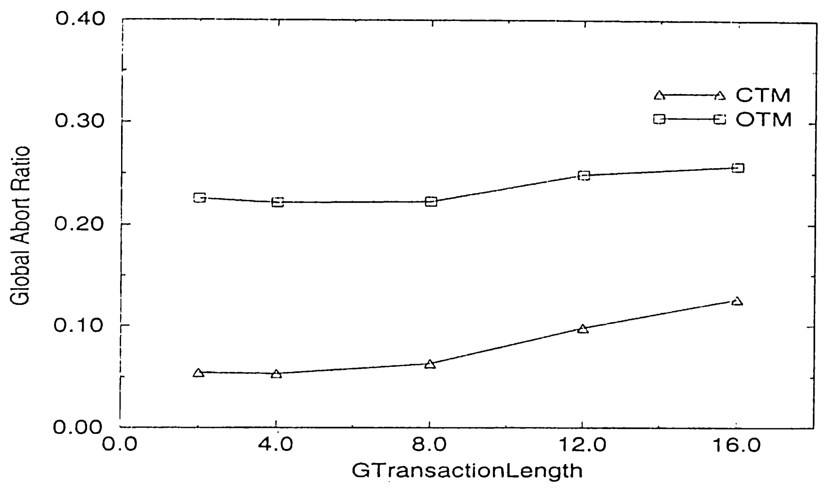

Figure 5.18: Global abort ratio vs. transaction length. LResourceUnit=1

5.2.4 The Impact of Local Transaction Behavior on Global Transactions

The effects of local transaction behavior on global transactions were tested by changing the update probability of local transactions. We have performed the experiments by varying LWriteProb from 0 to 1 using the standard workload parameter values and setting LResourceUnit to 1.

We can understand from Figure 5.19 that both algorithms' performance decreases when the local transactions create more conflicts and hold the resources longer time. CTM's performance is more sensitive to the local transaction behavior. As the update probability of local transactions increases, the CTM's performance sharply decreases. When we look at the global abort ratios in Figure 5.20, both OTM and CTM abort ratios steadily increase as the local data access conflicts increase. The increase in abort ratios is mainly due to the local aborts because of the deadlock situations between global transactions and local transactions.

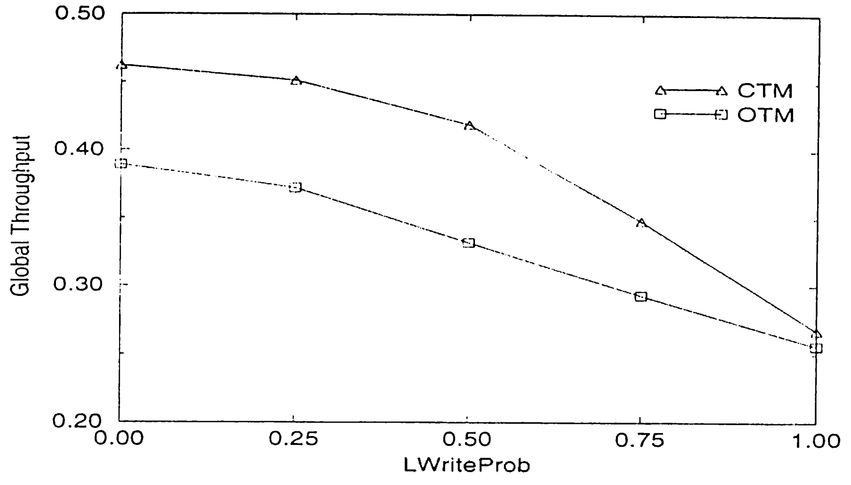

Figure 5.19: Global throughput vs. LWriteProb, LResourceUnit=1

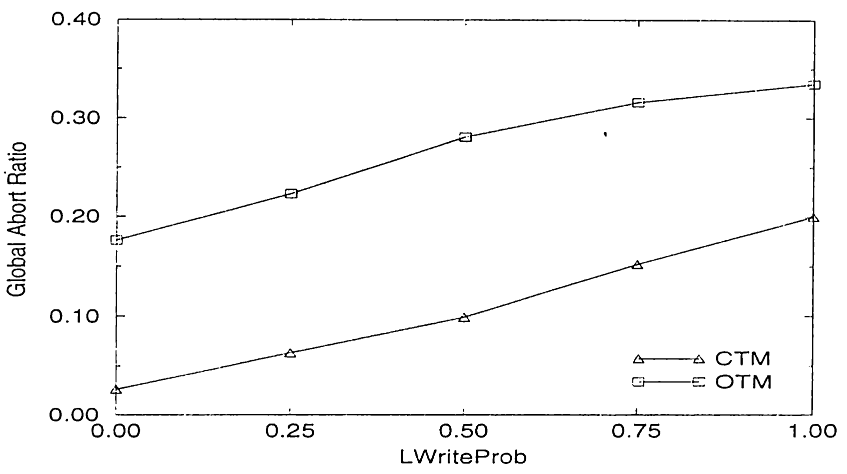

Figure 5.20: Global abort ratio vs. LWriteProb, LResourceUnit=1

5.2.5 Algorithms' Impact on Local Transactions

In the experiments discussed above, we also examined the impact of OTM and CTM algorithms on the local database performance. We can understand from Figure 5.21 that under no resource contention, as we increase GWriteProb of global transactions, local throughput decreases for both algorithms. However, OTM has better performance in terms of local throughput under all levels of data conflicts. The reason for this result is that, with OTM, global transactions do not hold page locks for very long periods. Global transactions take their

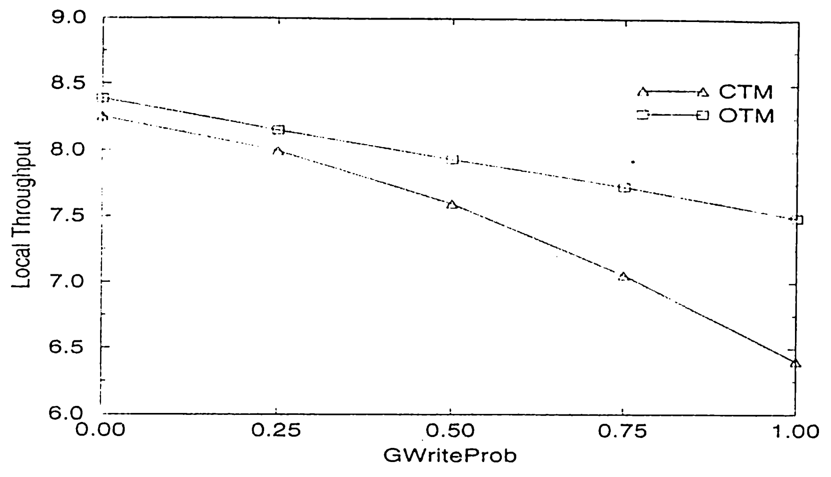

Figure 5.21: Local throughput vs. GWriteProb, LResourceUnit=20

tickets and release their locks as soon as they complete their data operations, thus the local blocking time and conflict ratio are minimized.

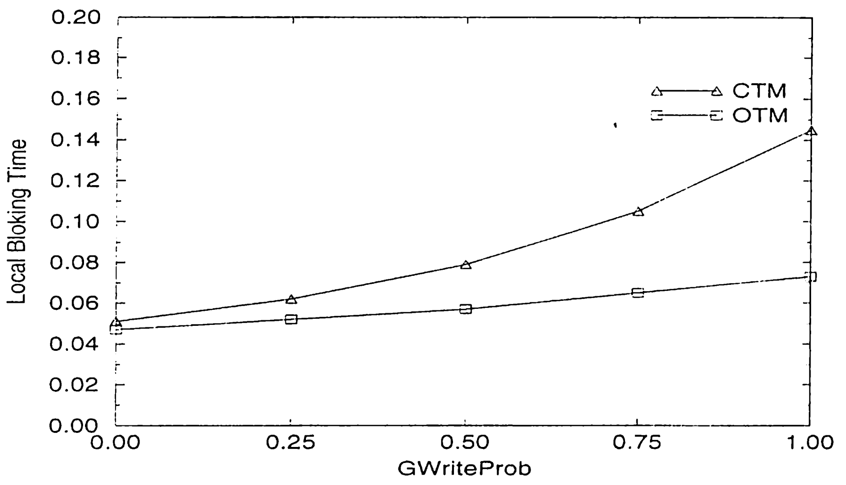

Figure 5.22: Local blocking time vs. GWriteProb, LResourceUnit=20

On the other hand, CTM follows just the opposite way and prefers to hold locks until it becomes sure about the serialization order of subtransactions using the take-a-ticket command. Figures 5.22 and 5.23 confirm our intuition about the worse performance of CTM for local transactions. Especially at high data contention, local transactions have large average blocking times and conflict ratios.

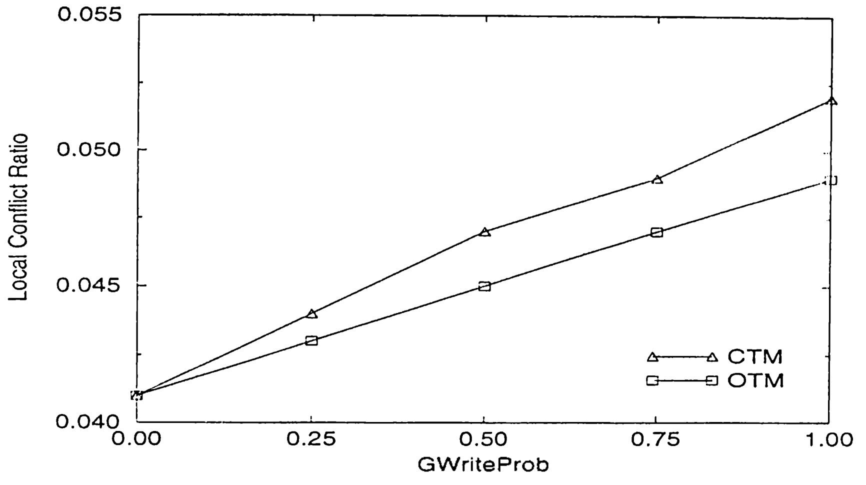

Figure 5.23: Local conflict ratio vs. GWriteProb, LResourceUnit=20

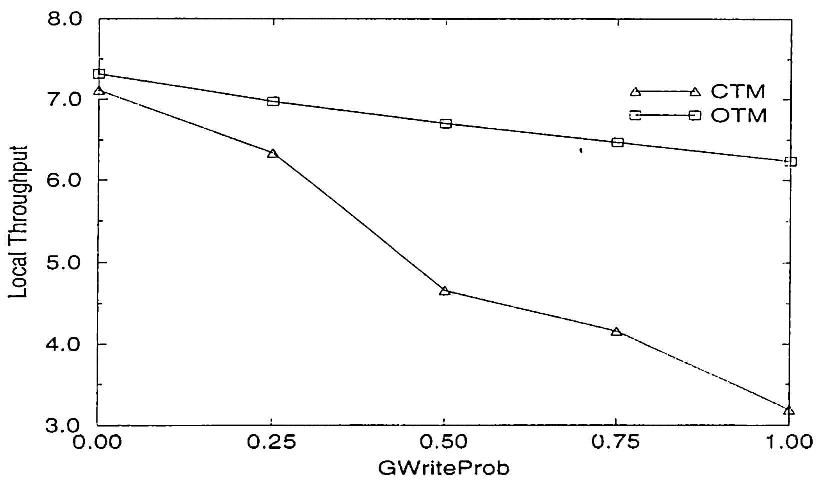

Figure 5.24: Local throughput vs. GWriteProb, LResourceUnit=1, LHotRegion=0.2

The impact of CTM algorithm on local transaction throughput is much worse when we create high levels of data conflict by setting LHotRegion parameter to 0.2. In Figure 5.24, the local throughput decreases very sharply with CTM algorithm, since as the number of global and local deadlocks increases, response time of a local transaction becomes longer. From the the subtransaction length experiments, we can observe similar results. Local transaction behavior in response to subtransaction length is plotted in Figure 5.25. As the global subtransaction length increases the local throughput decreases. However, the local transaction response time is not very sensitive to varying subtransaction length with OTM. On the other hand, CTM does not perform well when the subtransaction length is long.

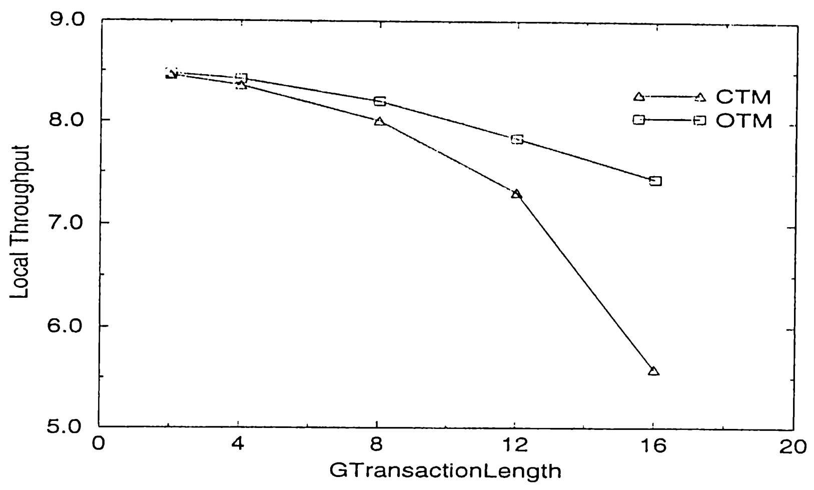

Figure 5.25: Local throughput vs. transaction length, LResourceUnit=20

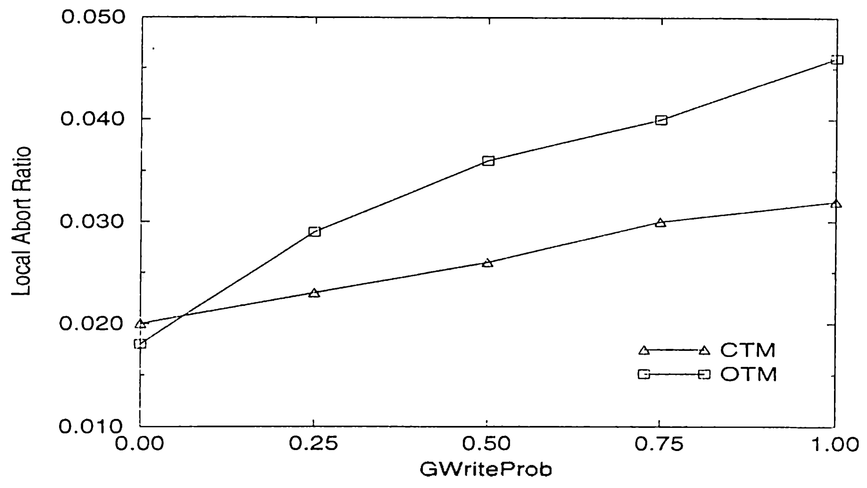

Figure 5.26: Local abort ratio vs. GWriteProb, LResourceUnit=20

When we look at the local abort ratios of two algorithms in Figure 5.26, at high data contention, OTM has higher local abort ratio due to cascade aborts caused by validation aborts. Figure 5.27 shows the local abort ratios as the subtransaction length increases. Although overall performance trends are similar, at very large subtransaction lengths, CTM causes more local deadlocks. The reasons for this result is that long transactions increase both the transaction blocking times and data access conflict rates.

Overall, our observations indicate that we cannot ignore the global concurrency control algorithms' impact on the local system performance. Algorithms like CTM that holds the data resources for longer periods of time, have a significant effect on the performance of the local transactions.

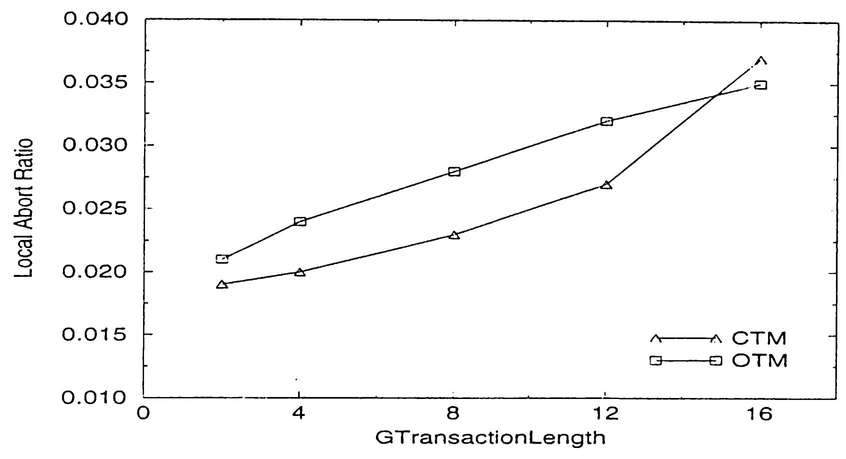

Figure 5.27: Local abort ratio vs. transaction length, LResourceUnit=20

5.3 Experiments on the Extended Global Transaction Model

In the experiments of this section, we have investigated the performance impact of the extended OTM and CTM algorithms and the extended transaction model characteristics. We set TreeHeight to 3 and NumChild to 2 where at most 8 global subtransactions can be executed in a global transaction. In all of these experiments, we limited the number of subtransactions required to be committed in a global transaction to achieve consistent global throughput and response times measurements. The other workload parameters of these experiments are the same as the settings given in Table 5.3. For global deadlock detection, simple global time-out mechanism was applied.

5.3.1 The Impact of Subtransaction Dependencies

In this set of experiments , we have investigated the effects of each dependency type individually. For the alternative and preference dependencies, one

Variable Parameters	Settings
NoDependencyProb	1.00, 0.75, 0.75, 0.50, 0.50, 0.25, 0.25, 0.00, 0.00
PreferenceProb	0.00, 0.25, 0.00, 0.50, 0.00, 0.75, 0.00, 1.00, 0.00
AlternativeProb	0.00, 0.00, 0.25, 0.00, 0.50, 0.00, 0.75, 0.00, 1.00

Table 5.5: Parameter settings for the analysis of the dependency relations

of the dependency probabilities was varied against the NoDependencyProb parameter and the throughput and average response time of global transactions were measured. Table 5.5 shows a sample of dependency parameter settings for this experiment. As the probability of each dependency type increases, the number of subtransactions executed in the system also increases. However, the number of subtransactions to be committed is limited to 2 in all settings. All of these experiments were conducted under both high and low resource contention.

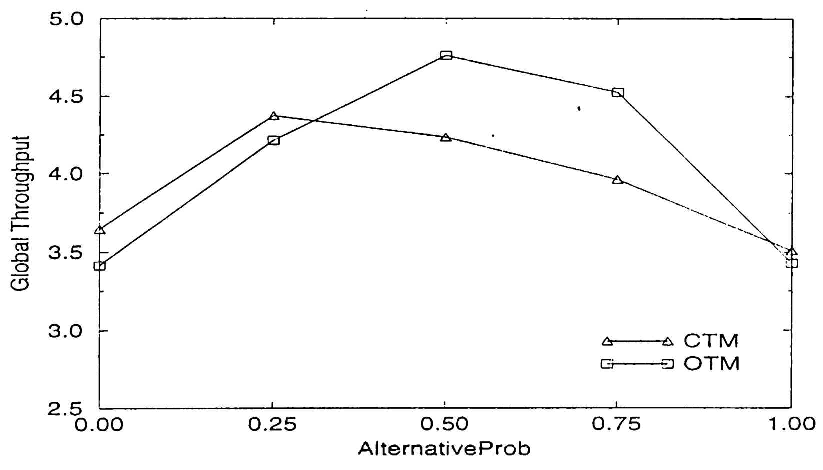

Figure 5.28: Global throughput vs. AlternativeProb, LResourceUnit=20

Figure 5.28 illustrates the effects of the alternative relation on the performance of OTM and CTM when there is no resource contention. The throughput of the system increases with both algorithms as the number of alternative transactions in a global transaction increases. However, OTM shows a sharp climbing and achieves better performance when the AlternativeProb is around 0.5. As we further increase AlternativeProb, both algorithms perform worse, because the additional alternative subtransactions introduce no advantage, and even unnecessarily increase the workload of the system. Also, alternative subtransactions execute more data operations which increases the global conflict probability. Consequently, throughput of both CTM and OTM are negatively affected when we increase the number of alternative subtransactions. Figure 5.29 illustrates the total global abort ratios of both algorithms. The rapid decrease in global abort ratio seems to be the main reason for both algorithms to perform better when AlternativeProb is between 0 and 0.5.

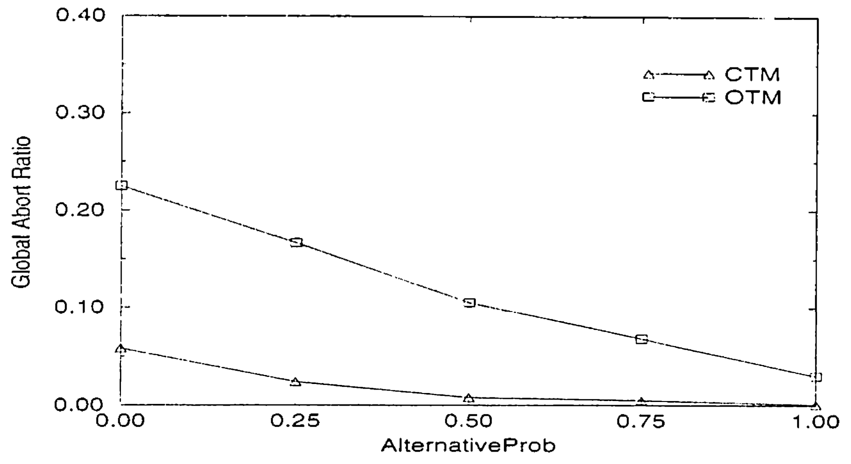

Figure 5.29: Global abort ratio vs. AlternativeProb, LResourceUnit=20

If we concentrate on a situation where there exists a high resource contention (Figure 5.30), as AlternativeProb increases, both algorithms' performance becomes better. If a global transaction has more alternative subtransactions, it has a chance to select the one which has a shorter response time. Besides, when an alternative relation is specified, the abortion of a single subtransaction does not lead to an immediate abortion of its parent transaction. As we increase AlternativeProb beyond 0.5, a global transaction accesses more resources and creates more conflicts which results in a performance loss for both algorithms. Nevertheless, the global throughput of the system is not below the situation where no alternative relation is specified.

When we have examined the preference dependency's effects on the algorithms' performance, the results obtained were different from that of the alternative dependency. Figure 5.31 illustrates the global throughput versus PreferenceProb under low resource contention. Initially, the throughput of both algorithms slightly increases but this does not last long. As the number of preferred subtransactions increases, OTM's and CTM's throughputs sharply decrease. The reason for this result is that, in preference relation GTM submits all of the alternative subtransactions, but waits for the preferred one. Executed alternative subtransactions are not committed unless the preferred one is aborted by the local site. Therefore, subtransactions which are not the

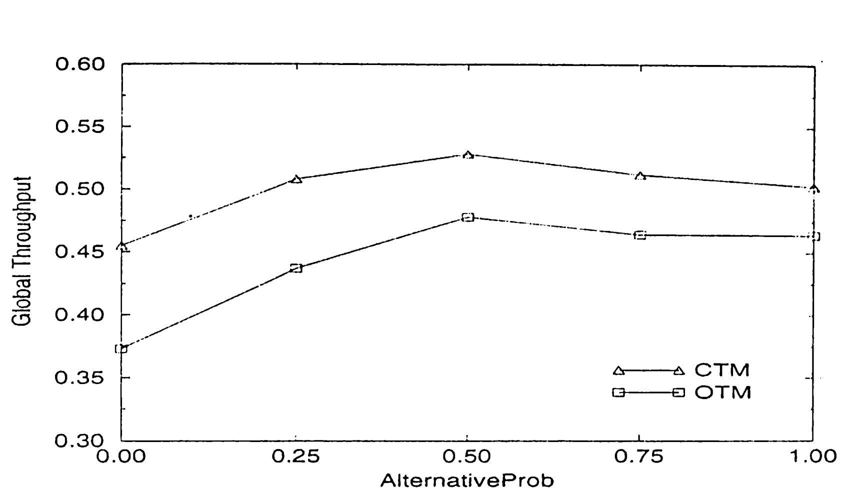

Figure 5.30: Global throughput vs. AlternativeProb, LResourceUnit=1

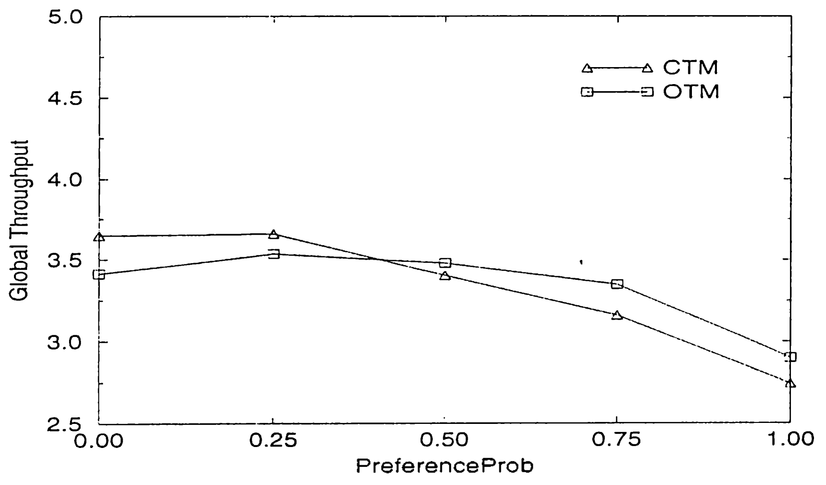

Figure 5.31: Global throughput vs. PreferenceProb, LResourceUnit=20

preferred ones cannot minimize their parents' response time, although they have completed their data operations. As we can understand from Figure 5.32, the global abort ratios slightly decrease as the preference relation probability increases. This result shows that the abort possibility of preferred transactions is low which makes most of the alternative subtransactions unnecessary. In addition to these, with CTM algorithm, alternative subtransactions do not release their locks before the preferred one enters to the ready-to-take-a-ticket state. Especially when PreferenceProb is high, holding the data locks a long period of time negatively affects CTM's performance. This is the main reason that throughout of CTM abruptly drops below the throughput of OTM as

PreferenceProb increases.

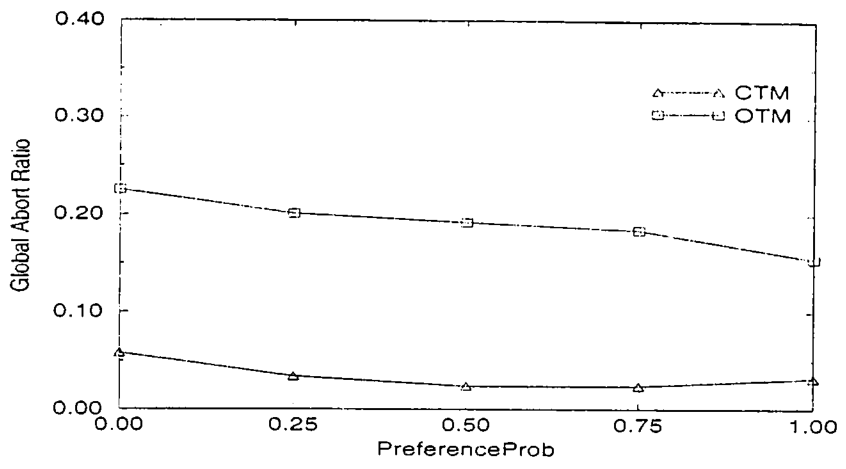

Figure 5.32: Global abort ratio vs. PreferenceProb, LResourceUnit=20

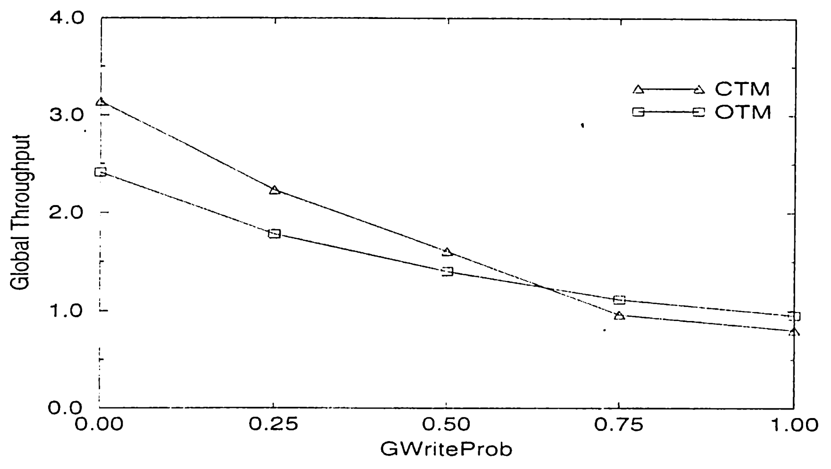

Figure 5.33: Global throughput vs. GWriteProb, precedence relation, LResourceUnit=20

To analyze the impact of precedence relation, we set both PrecedenceProb and TreeHeight to 1 and by changing GWriteProb, we examined the behavior of OTM and CTM algorithms. Figure 5.33 shows the global throughput of OTM and CTM under no resource contention. If the dependency relation is precedence, the response time of a global transaction is longer than the response time of transactions with no-dependency relation. Hence, the abortion cost of a global transaction is higher. When there is no resource contention, CTM performs better than OTM under low data contention. As the data contention increases, CTM's performance rapidly decreases due to the same reasons mentioned in the section of the classical transaction model experiments. Abort ratios of two algorithm are plotted in Figure 5.34. OTM suffers from validation aborts as expected. Under the high resource contention, OTM's poor performance is also verified in Figure 5.35. All of these results show that global transactions with precedence relation negatively affect the throughput of the systems in which restart rate is higher.

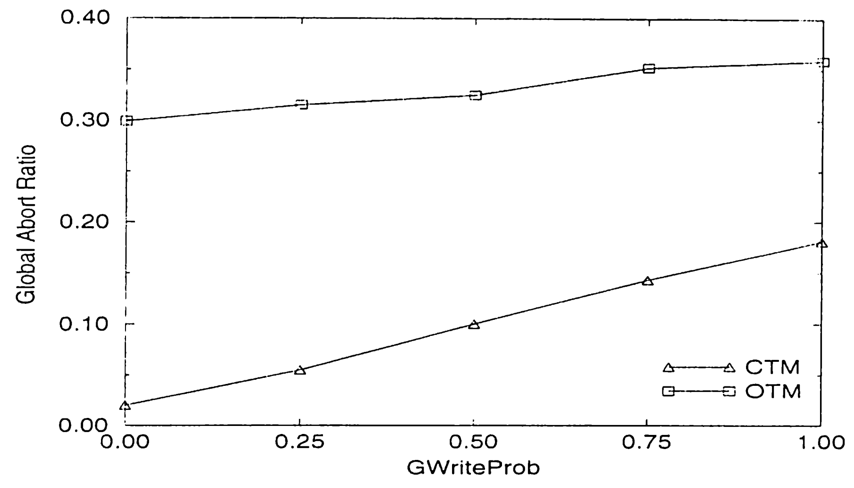

Figure 5.34: Global abort ratio vs. GWriteProb, precedence relation, LResourceUnit=20

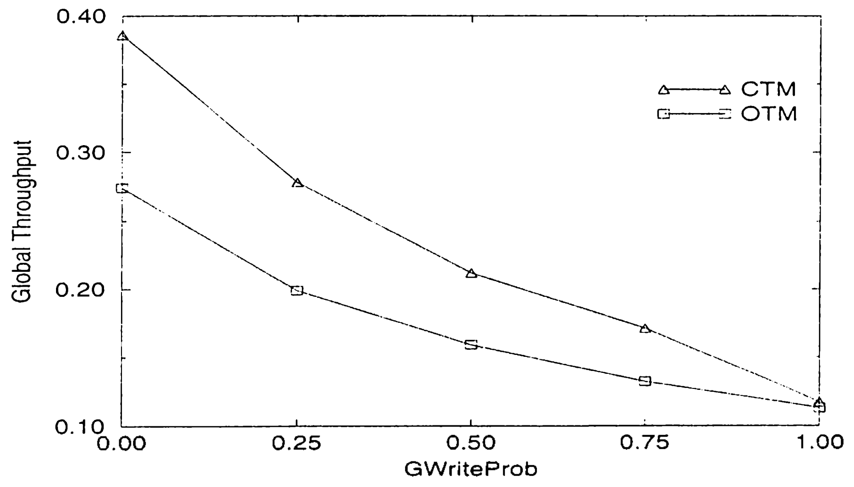

Figure 5.35: Global throughput vs. GWriteProb, precedence relation, LResourceUnit=1

5.3.2 The Impact of Commit Independent Subtransactions

In this set of experiments, we have investigated how compensatable and retriable subtransactions affect the overall performance. Varying CompensatableProb and RetriableProb parameters against OrdinaryProb parameter has provided us various situations to evaluate the effects of commit independent subtransaction types. The settings of the experiments on the commit independent types are listed in Table 5.6. To evaluate the performance impact of commit-independent subtransactions, we employ classical transaction model by setting TreeHeight and NoDependencyProb to 1. Thus, we isolate the effects of subtransaction dependencies. In these experiments, we also isolate the impact of resource contention by setting LResourceUnit to 20.

Parameters	Settings
OrdinaryProb	1.00, 0.75, 0.75, 0.50, 0.50, 0.25, 0.25, 0.00, 0.00
CompensatableProb	0.00, 0.25, 0.00, 0.50, 0.00, 0.75, 0.00, 1.00, 0.00
RetriableProb	0.00, 0.00, 0.25, 0.00, 0.50, 0.00, 0.75, 0.00, 1.00

Table 5.6: Parameter settings for the analysis of commit-independent transaction types

Figure 5.36 shows the effects of retriable transactions to the performance of OTM and CTM. In general both algorithms' performance slightly increases as the probability of retriable transactions increases. It can be said that, the global throughput is not very sensitive to the RetriableProb. Since the subtransactions executed at the same site have to access the same data item to take their tickets, the retriable subtransactions indirectly affect performance of the other subtransactions executed at the same site. Especially, with the CTM algorithm, if GTM decides to commit a retriable subtransaction, other subtransactions executing at the same site has to wait for that subtransaction to take its ticket. Figure 5.37 represents the global abort ratios of the two algorithms. OTM's abort ratio increases with increasing RetriableProb. The reason for this result is that OTM uses GSG validation mechanism to ensure that no other transaction is serialized between a global transaction and its retriable subtransaction.

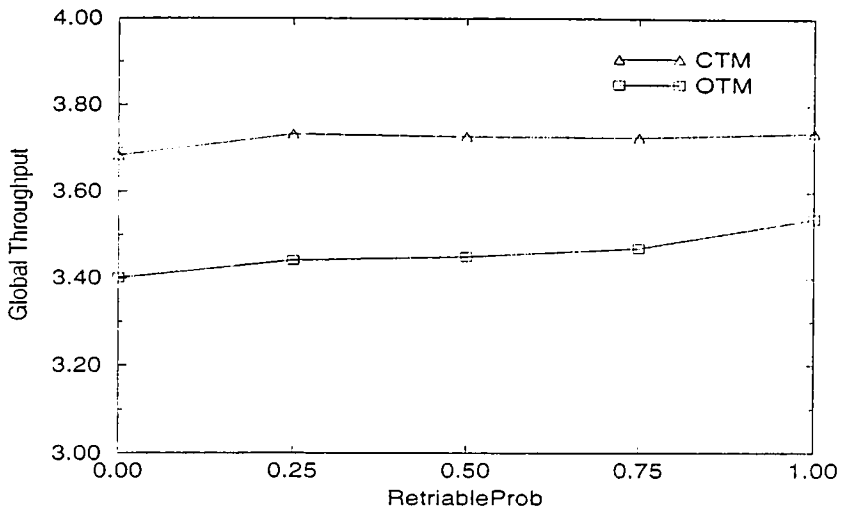

Figure 5.36: Global throughput vs. RetriableProb, LResourceUnit=20

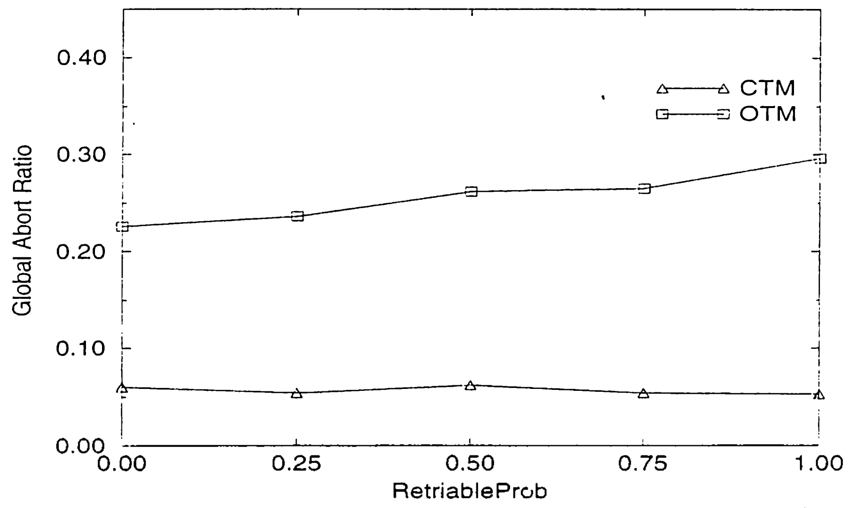

Figure 5.37: Global abort ratio vs. RetriableProb, LResourceUnit=20

The impact of compensatable transactions on the global throughput is illustrated in Figure 5.38. CTM's performance is not very sensitive to the compensatable transactions, while OTM behaves worse as we increase CompensatableProb. Compensating transactions have a negative effect on the overall performance of the OTM algorithm. Like retriable transactions, due to the ticketing approach, compensatable transactions affect the execution of other subtransactions at their sites. With the OTM algorithm, GTA aborts the subtransactions that have obtained higher ticket values than the compensatable transaction before its parents commits. Therefore, these additional aborts to ensure serializability of compensatable subtransactions decrease the throughput of OTM. Global abort ratios of the two algorithms plotted in Figure 5.39 confirm this observation.

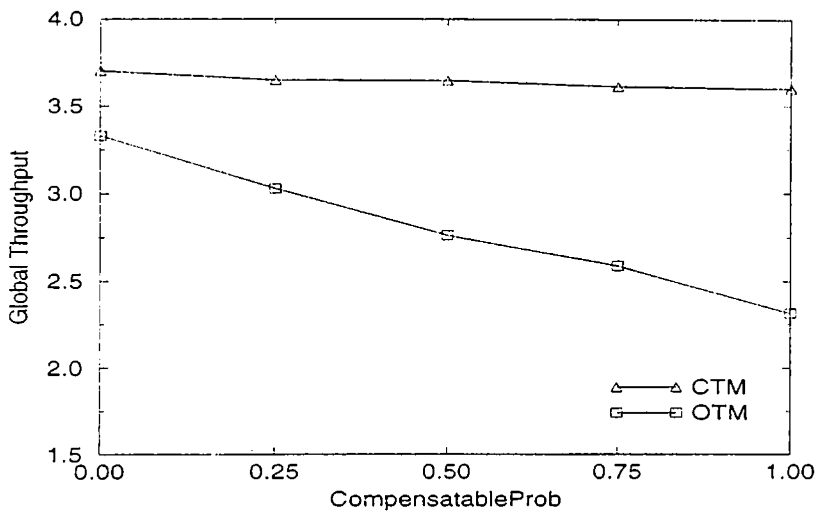

Figure 5.38: Global throughput vs. CompensatableProb, LResourceUnit=20

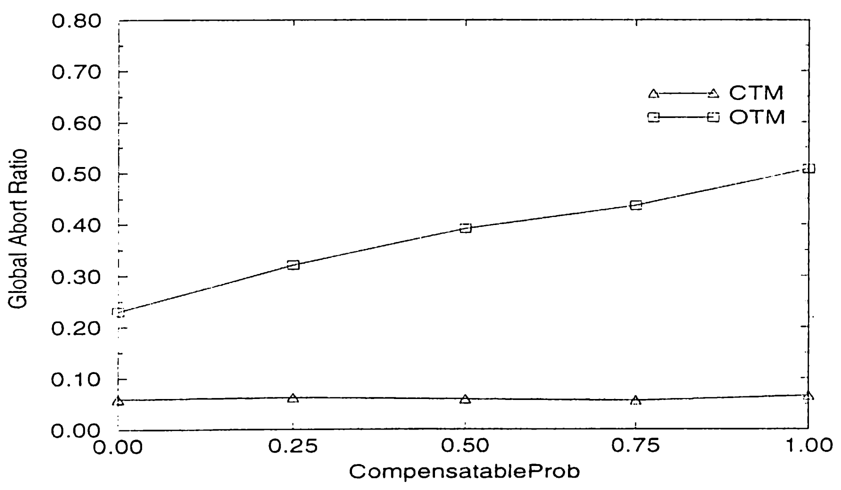

Figure 5.39: Global abort ratio vs. CompensatableProb, LResourceUnit=20

We should not expect overall performance gains from the compensatable transactions since the response time of a global transaction does not depend on the execution of its compensatable children. However, from the local database point of view, early committed transactions can improve the local transaction throughput as the locks are released earlier. This prediction is confirmed by the experiments of Section 5.3.4.

5.3.3 The Impact of Data Contention

In order to examine the impact of data contention on the performance of the extended transaction model, we have compared alternative, preference, and no-dependency relations by varying GWriteProb. In this experiment, we have examined the performance results with three different workloads: the one that only contains Nodependency relation, the one with 0.5 AlternativeProb and the one with 0.5 PreferenceProb.

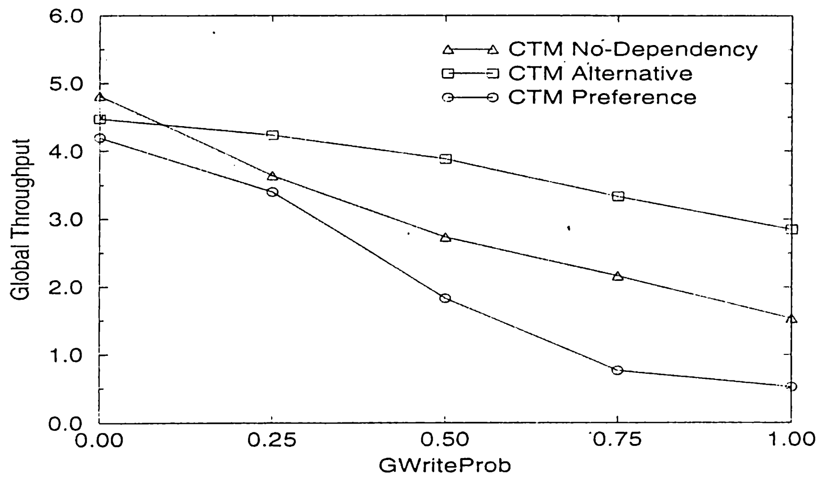

Figure 5.40: Global throughput vs. GWriteProb, CTM. LResourceUnit=20

Figures 5.40 and 5.41 illustrate throughput of each dependency type with CTM and OTM, respectively. From these figures, we can understand that global transactions that have alternative subtransactions provide better performance with both OTM and CTM. Transactions with preference relation perform better than the transactions that has no dependency when OTM is employed for global concurrency control. Contrary, throughput of the transactions with the preference relation face a sharp decrease when CTM is the algorithm of choice. This is due to the fact that all of the alternative subtransactions do not release their locks until the preferred one completes its execution. On the other hand, with OTM, alternative subtransactions do not have to wait for preferred ones to release their locks. Figures 5.42 and 5.43

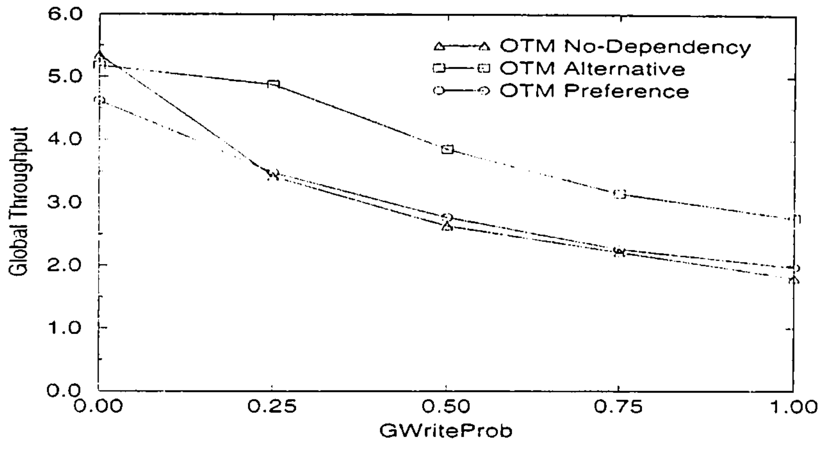

Figure 5.41: Global throughput vs. GWriteProb, OTM, LResourceUnit=20

show global abort ratios obtained with these experiments. The low abort ratios of the global transactions with the alternative relation explain their better performance.

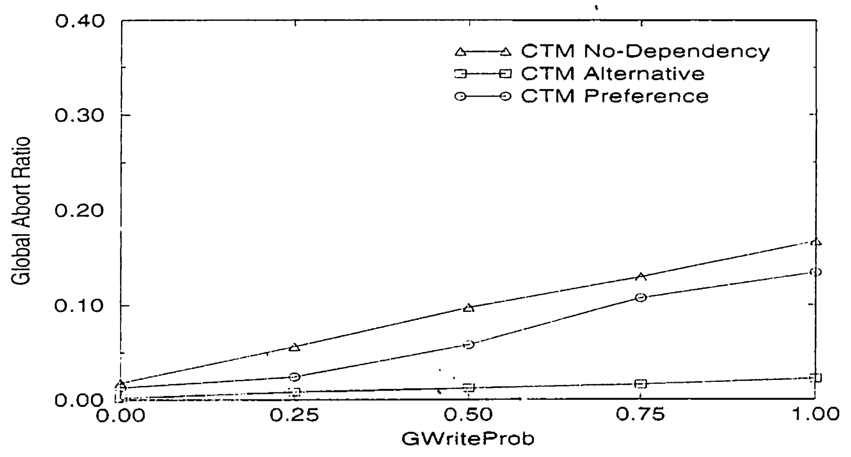

Figure 5.42: Global abort ratio vs. GWriteProb, CTM, LResourceUnit=20

5.3.4 The Impact of Extended Transactions on Local Transactions

If we look at the performance impact of the alternative and the preference dependencies from the local database point of view, the local throughput is negatively affected for both OTM and CTM as shown in Figures 5.44 and 5.45.

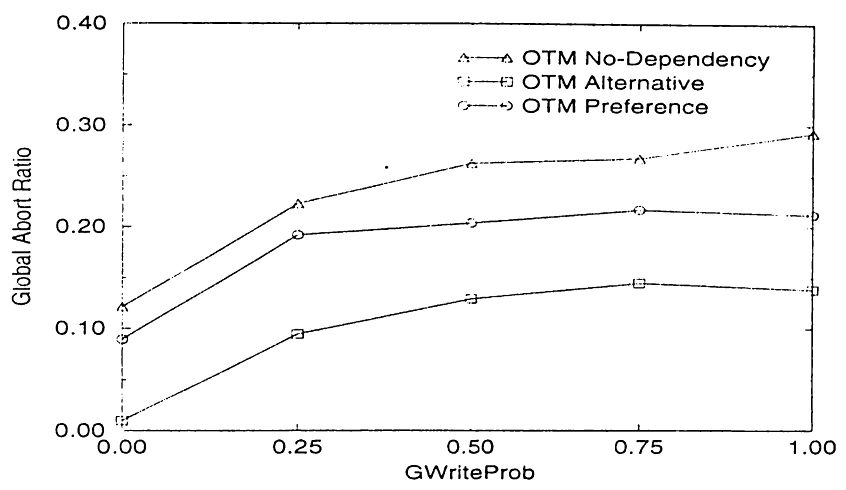

Figure 5.43: Global abort ratio vs. GWriteProb, OTM, LResourceUnit=20

The reason for this result is that global transactions with alternative subtransactions access more data resources and create more conflicts with the local transactions. In general, OTM performs better than CTM in terms of local transaction throughput. For the alternative relation, OTM's local throughput performance also drops due to the cascading aborts of local transactions as a results of the large amount of conflicts with alternative subtransactions.

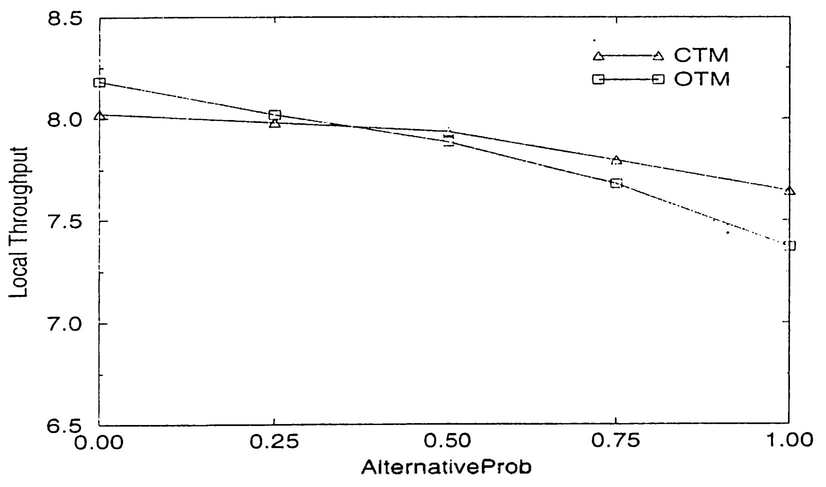

Figure 5.44: Local throughput vs. AlternativeProb, LResourceUnit=20

When we consider the impact of precedence relation on the local transaction throughput, as we increase GWriteProb (Figure 5.46), OTM performs better than CTM since it releases allocated resources of previously executed

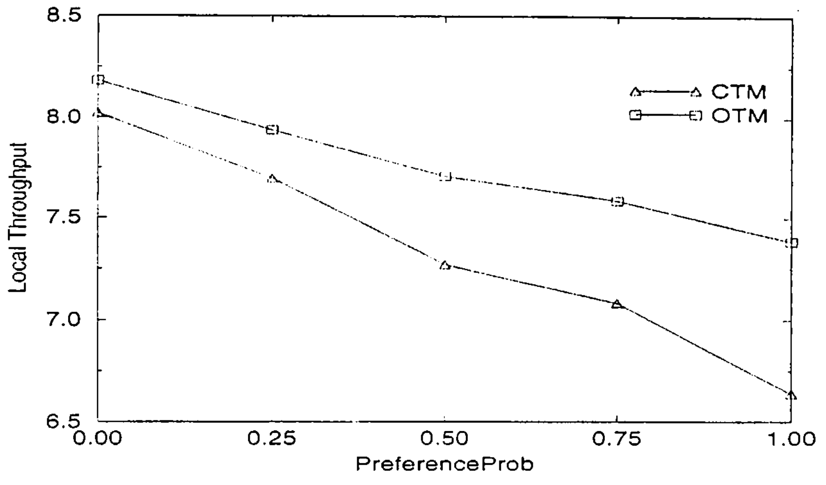

Figure 5.45: Local throughput vs. PreferenceProb, LResourceUnit=20

transactions before submitting the next one. In other words, OTM leaves more resources for local transactions compared to CTM.

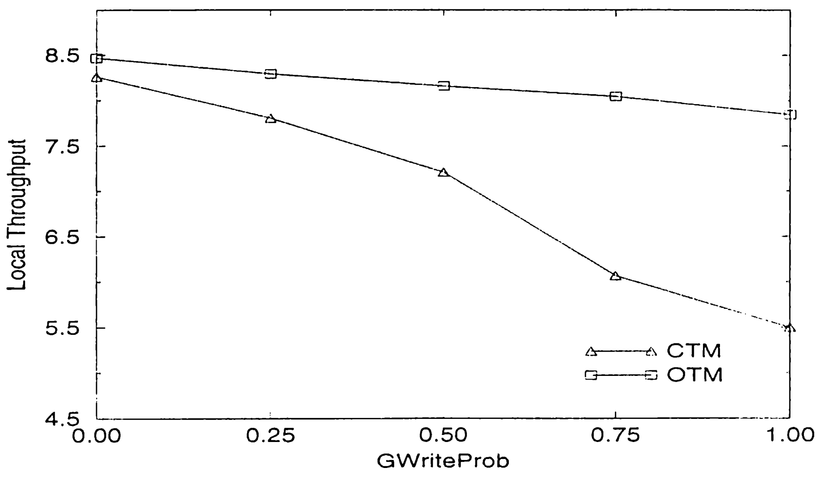

Figure 5.46: Local throughput vs. GWriteProb, precedence relation, LResourceUnit=20

When we look at retriable transactions' impact on the local throughput in Figure 5.47, the throughput with both algorithms slightly decreases as the ratio of retriable transaction increases. However, CTM's local throughput is not affected as much as that of OTM. The situation is not the same when we look at the compensatable transactions' impact on the local transactions. Figure 5.48 compares OTM's and CTM's local throughput as the ratio of compensatable transactions increases. Again, the performance with CTM is not much affected by the compensatable transactions. On the other hand, we can observe an improvement in local transaction performance with OTM, as we increase CompensatingProb from 0 to 0.5. As we further increase the CompensatingProb, GTA aborts of OTM algorithm reduces the local throughput.

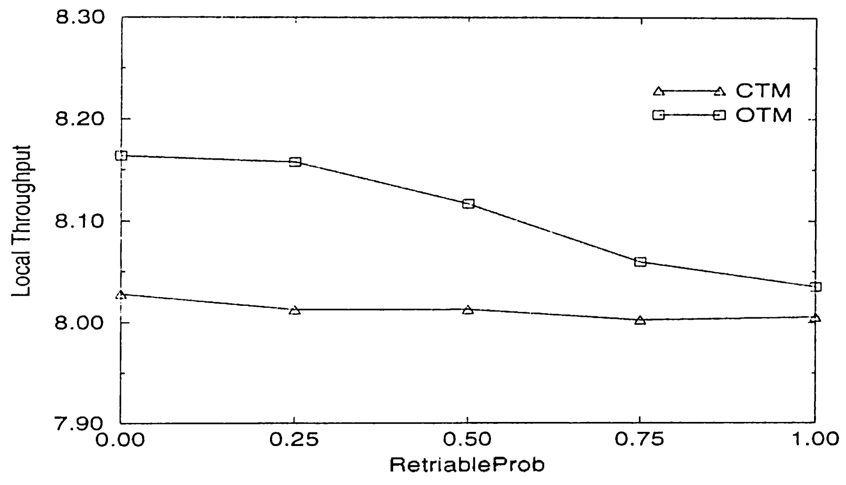

Figure 5.47: Local throughput vs. RetriableProb, LResourceUnit=20

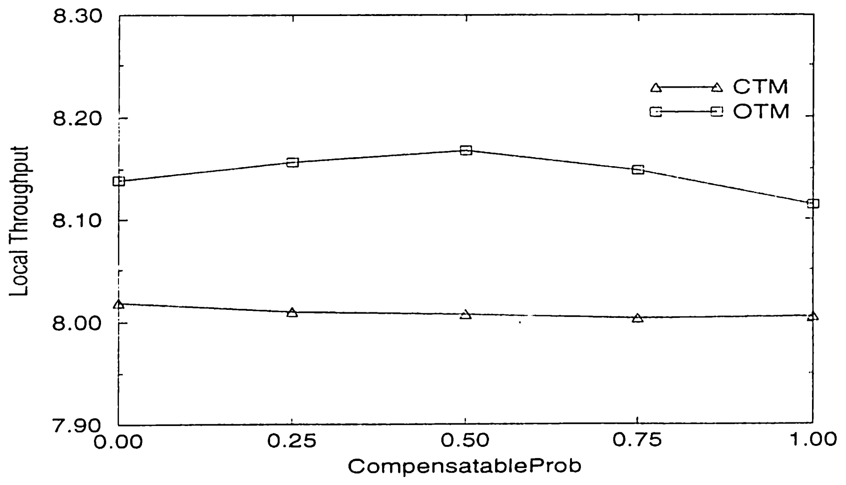

Figure 5.48: Local throughput vs. CompensatableProb, LResourceUnit=20

5.4 Experiments on Global Deadlock Detection Algorithms

In the experiments of this section, we have examined the effects of deadlock detection algorithms on the performance of the system. The basic timeout, the extended timeout, and the PCG algorithms have been compared. CTM has been employed in these experiments since it has a higher rate of global deadlock situations. In all of these experiments the global transaction parameter settings were the same as the ones in Table 5.3.

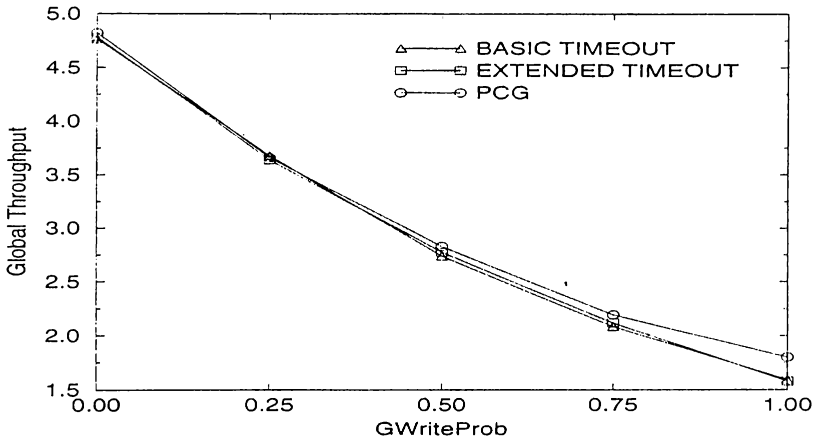

Figure 5.49: Deadlock detection algorithms, global throughput vs. GWriteProb, LResourceUnit=20

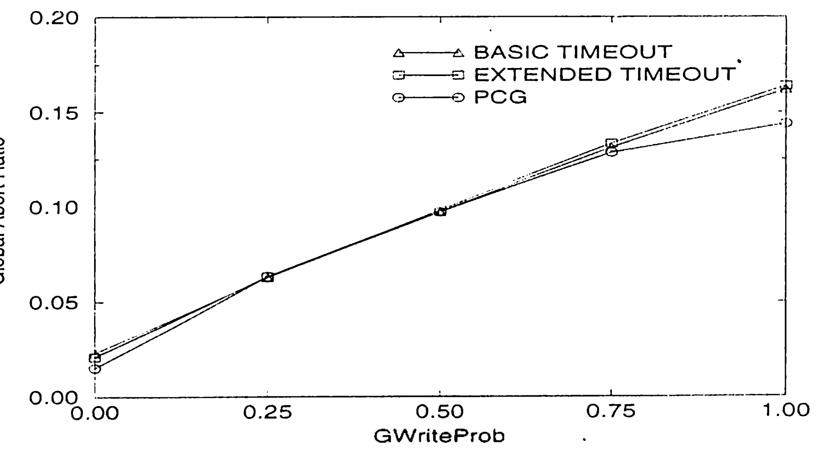

Figure 5.50: Deadlock detection algorithms, global abort ratio vs. GWriteProb, LResourceUnit=20

In data contention experiments, we varied GWriteProb parameter to estimate the performance impact of three algorithms under no resource contention. Figure 5.49 shows the performance results obtained with the three algorithms as a function of increasing data contention. Initially, all the algorithms' behaviors are similar to each other, since the timeout abort ratios are very small under low data contentions. PCG algorithm performs slightly better than the others when the data contention becomes higher. In spite of this, PCG algorithm cannot achieve huge performance improvements since its performance is also dependent on the timeout interval used to check deadlock situation. There is no considerable performance difference between the basic and extended timeout algorithms, since both of them solve the deadlock problem by employing a timeout period. PCG's better performance under very high data contention is also validated by the global abort ratios plotted in Figure 5.50.

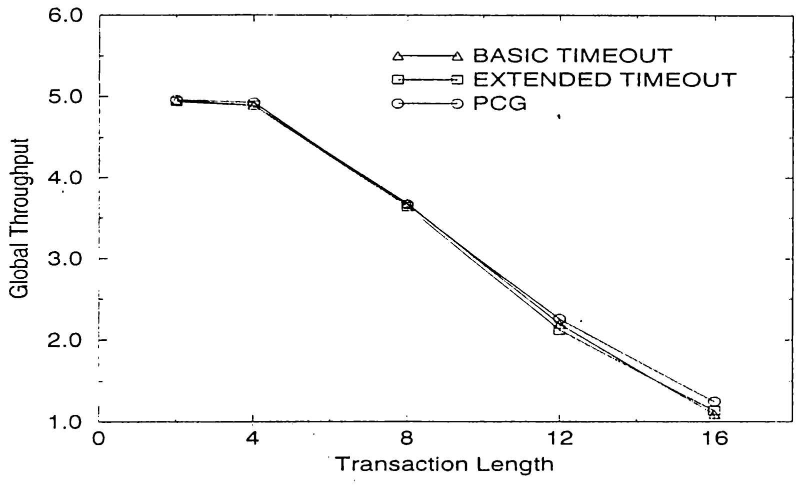

Figure 5.51: Deadlock detection algorithms, global throughput vs. transaction length. LResourceUnit=20

We have also investigated how the performance of each algorithm is sensitive to the global subtransaction length. Figure 5.51 shows the behavior of three algorithms as the transaction length is increased. The results are similar to those of the data contention experiments. The advantage of PCG algorithm is visible only when the subtransaction length is long. If we look at the performance impact of the deadlock detection algorithms on local transactions in Figures 5.52 and 5.53, the performance improvement of PCG over the others can be notified more clearly.

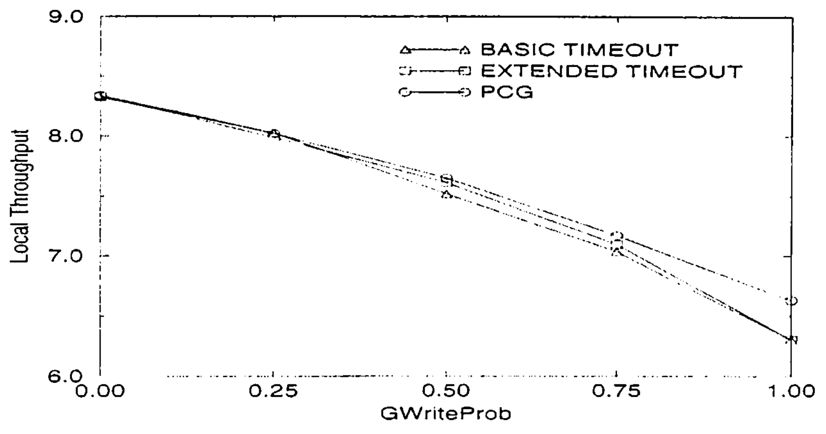

Figure 5.52: Deadlock detection algorithms, local throughput vs. GWriteProb, LResourceUnit=20

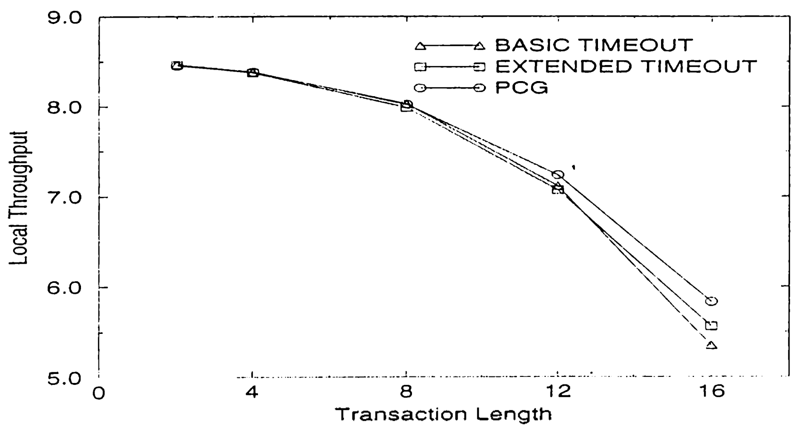

Figure 5.53: Deadlock detection algorithms, local throughput vs. transaction length, LResourceUnit=20

5.5 Performance Comparison of the Classical Transaction Model and the Extended Transaction Model

The final set of experiments have been performed for the comparison of classical transaction model and the extended transaction model. In these experiments, the performance of extended global transactions and the performance of their semantically equivalent classical transactions have been compared.

Parameters	Settings
TreeHeight	3
NumChild	2
NoDependencyProb	0.34
PrecedenceProb	0.0
PreferenceProb	0.33
AlternativeProb	0.33
OrdinaryProb	1.0
CompensatableProb	0.0
RetriableProb	0.0

Table 5.7: Global transaction parameter settings for the experiments that compare classical transaction model and extended transaction model

We implemented a semantic analyzer inside the Global Transaction Generator (GTG) to capture the semantics beyond the extended global transactions. An extended global transaction is created by using the parameter values listed in Table 5.7, and then the analyzer parses that extended transaction and creates a semantically equivalent set of classical global transactions for submission. GTG semantic analyzer also coordinates the submission of global transactions from the corresponding set of transactions according to the GTM's abort and commit response on the previously submitted transaction. We employed the simple global time-out mechanism for the execution of both the classical and the extended transactions in order to be consistent in performance evaluation.

In the first set of experiments. we compared the classical transaction model and the extended transaction model with both CTM and OTM algorithms under no resource contention. Figure 5.54 illustrates the variation of the global throughput as the data contention increases. Initially, both CTM and OTM perform better on the classical global transaction model, since the additional characteristics of the extended model do not introduce any advantages in a situation where global conflict ratios and abort possibilities are low. But as GWriteProb increases the situation changes. While the classical transaction model shows a heavy performance loss due to the high abort ratio. the extended transaction model minimize the global aborts by executing alternative subtransactions. The global abort ratio values are shown in Figure 5.55. Low global abort ratio of CTM and OTM on the extended transaction model verifies the appropriateness of the extended transaction model.

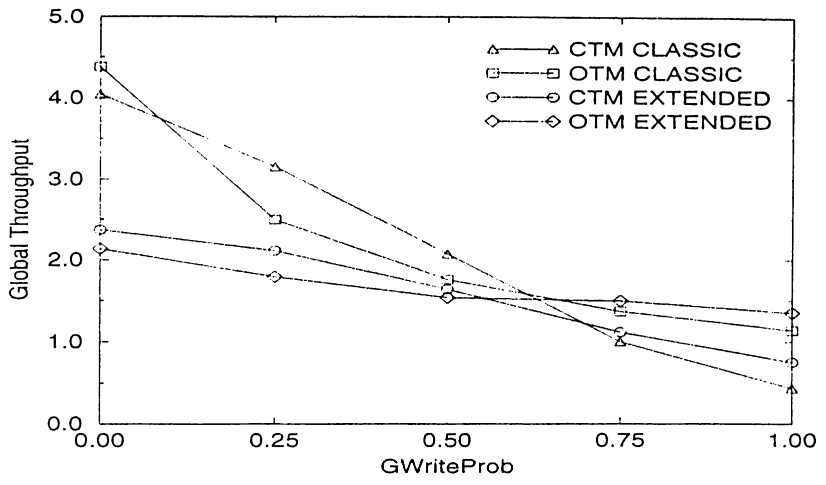

Figure 5.54: Global throughput vs. GWriteProb, LResourceUnit=20

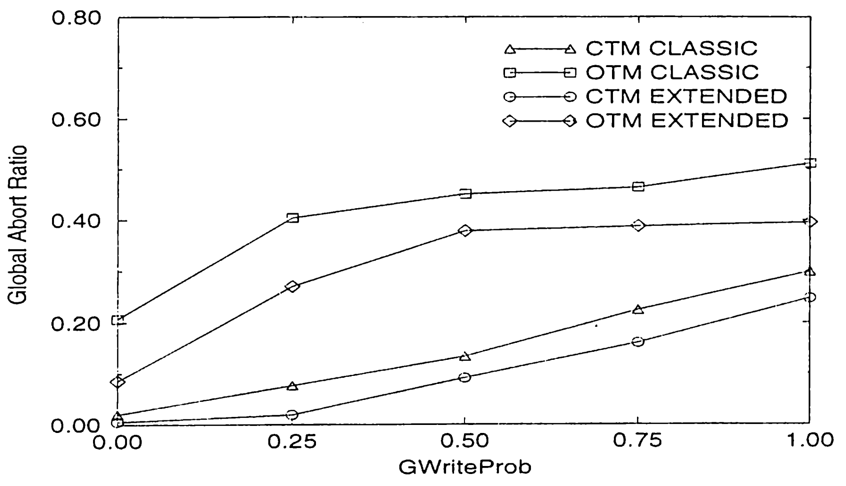

Figure 5.55: Global abort ratio vs. GWriteProb, LResourceUnit=20

When we repeat this experiment by setting the NumSites parameter to 16, thus reducing the conflicts among the subtransactions of global transactions, CTM and OTM algorithms perform better in the overall. Figures 5.56 and 5.57 illustrate the global throughput and the abort ratios when we set NumSites parameter to 16. In these figures we have also confirmed that CTM achieves better performance than OTM algorithm.

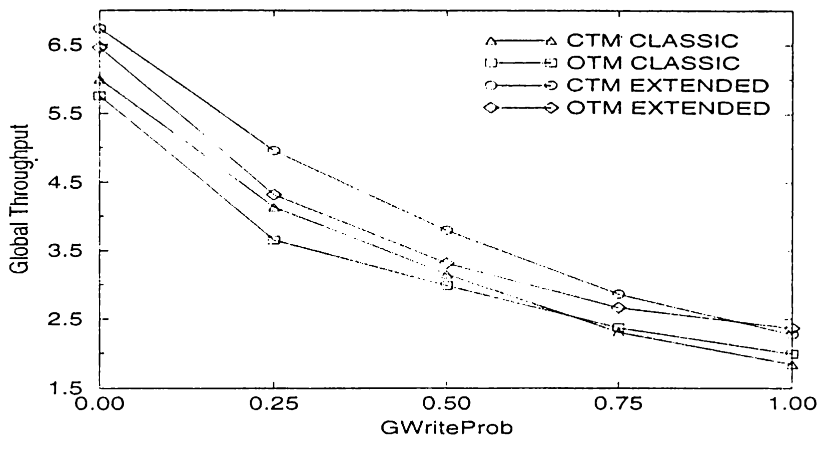

Figure 5.56: Global throughput vs. GWriteProb, NumSites=16, LResourceUnit=20

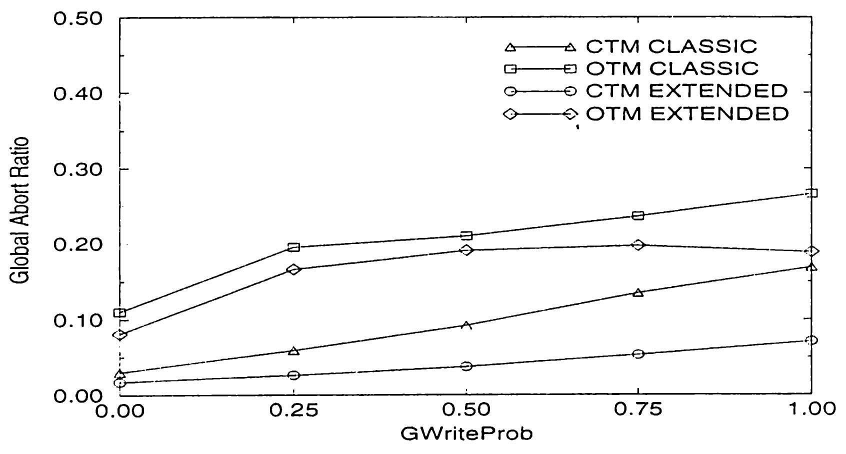

Figure 5.57: Global abort ratio vs. GWriteProb, NumSites=16, LResourceUnit=20

To examine the situations where the system has resource contention, we repeated the above experiments by setting the number of resource unit to 1. Figure 5.58 illustrates the performance of OTM and CTM on both transaction models. Under the high resource contention, OTM and CTM perform slightly better with the classical transaction model than with the extended model. Since the extended model is based on extra resource usage and the resources are now restricted, additional properties of the extended transaction model does not improve the performance. Global abort ratios of the algorithms are plotted in Figure 5.59. Again, OTM and CTM achieve low abort ratios for the extended transaction model.

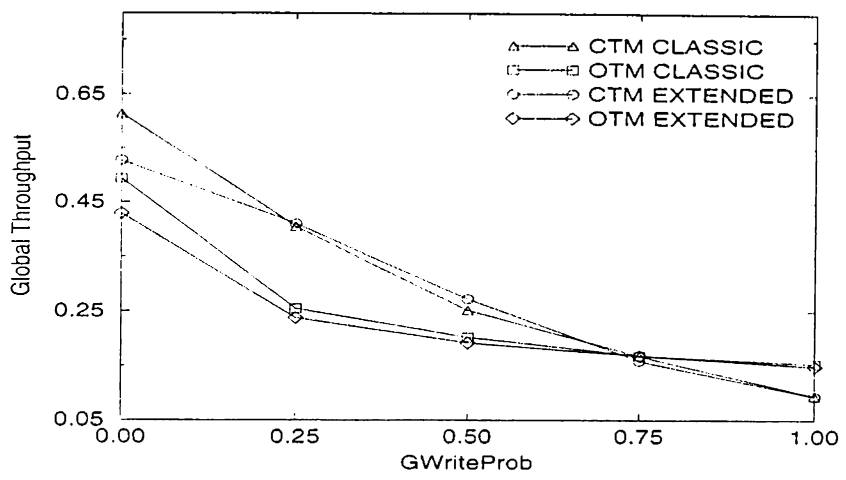

Figure 5.58: Global throughput vs. GWriteProb, LResourceUnit=1

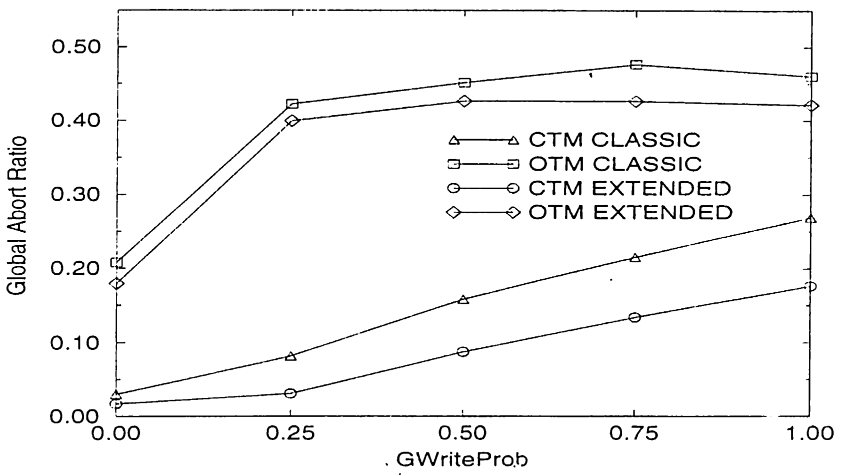

Figure 5.59: Global abort ratio vs. GWriteProb , LResourceUnit=1

If we compare the two global transaction models in terms of local transactions, by looking at Figure 5.60 we can say that the extended transactions have negative impact on the local throughput. The local throughput of both algorithms decreases faster with the extended transaction model than with the classical model. This is an expected result since the blocking time of the local transactions increases as the global transactions are allocated more resource from the local sites. When the system has resource contention, again the classical transaction model has higher local throughput than the extended transaction model. Local throughput values of the system are plotted in Figure 5.61. From the figure, we can also observe the worse performance of CTM for both transaction models while the data contention is increased.

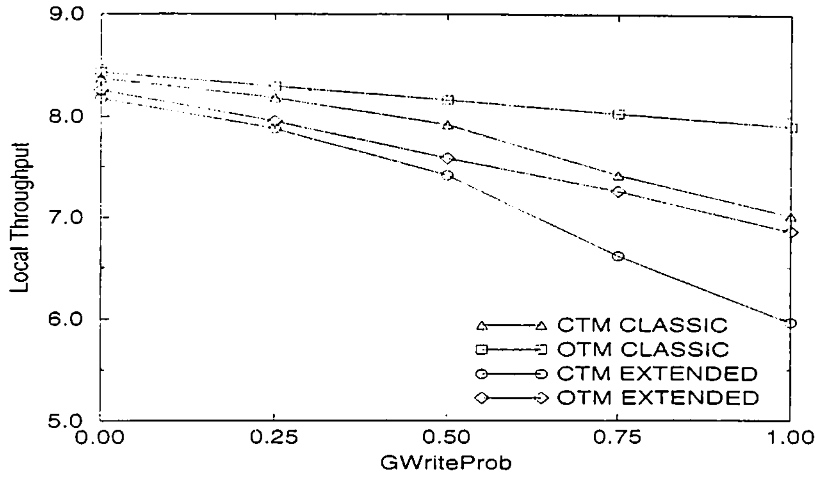

Figure 5.60: Local throughput vs. GWriteProb, NumSites=16, LResourceUnit=20

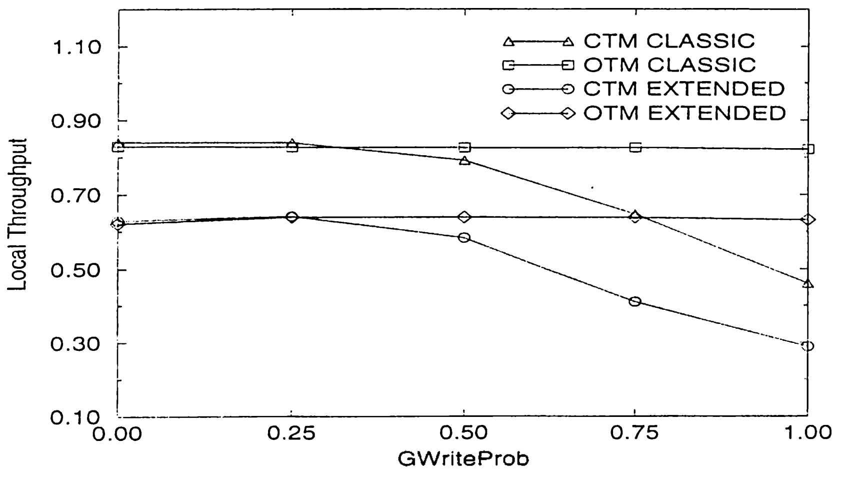

Figure 5.61: Local throughput vs. GWriteProb, LResourceUnit=1

Chapter 6

Conclusions

It is difficult to implement traditional transaction scheduling methods in a multidatabase system (MDBS) due to the heterogeneity and autonomy of the connected local sites. In this thesis, we introduced an extended transaction model for MDBSs. The proposed transaction model covers nested transactions, various dependency types among subtransactions, and commit-independent transactions that make the model much more flexible and powerful than the traditional transaction model. The formulation of complex MDBS transaction types can easily be accomplished with the extended semantics captured in the model. The execution model does not make any assumption regarding the concurrency control protocols executed at the local sites connected to the MDBS. The global serializability is ensured through the ticketing method proposed by Georgakopoulos et al. [13]. Atomic commitment of global transactions is provided through the use of two-phase commit (2PC) protocol. The blocking effect of 2PC is reduced by executing commit independent transactions.

We handled the global deadlock problem by employing a time-out mechanism for the execution of global transactions. A global deadlock detection algorithm based on potential conflict graph (PCG) has been adopted to our execution model to reduce unnecessary global aborts that can occur due to the estimation errors with the time-out mechanism.

We also proposed a detailed simulation model of a MDBS to analyze the performance of the proposed transaction model. Using this simulation model, first the performance implications of the classical transaction model with both the conservative ticket method (CTM) and the optimistic ticket method (CTM) have been investigated. Experimental results show that CTM seems to be the algorithm of choice when the hardware resources at local sites are limited. OTM does not yield better performance unless there exists high data contention. Our observations indicate that with both algorithms, global transactions negatively affect the performance of local transactions. This impact is more serious when CTM is employed.

With the second set of experiments, we have investigated the performance impact of the additional features of the extended transaction model with both OTM and CTM. We have observed that although the alternative subtransactions introduce additional workload to the system, the global transactions with alternative subtransactions perform better than the independent transactions with both OTM and CTM algorithms. Unlike the alternative relation, the preference relation does not yield significant throughput improvement since the additional subtransactions do not provide any advantage unless the preferred subtransactions are aborted.

When we have studied the impact of the precedence relation, we have seen that the response times of global transactions with this type of relation are larger than that of global transactions with no-dependency. Consequently, the cost of transaction abort is higher with the precedence relation which causes OTM to perform worse than CTM.

The performance results obtained for commit-independent subtransactions can be summarized as follows. While retriable transactions slightly improve both algorithms' global throughput, early committed compensable transactions do not provide any performance advantage for global transactions. Nevertheless, the overall performance impact of commit-independent subtransactions is not significant with either OTM or CTM.

When we have looked at the performance impact of extended transaction characteristics from the local sites' point of view, we have observed some performance trade-offs between the local and the global transaction throughput. As we introduce more additional features of the extended transaction model, the local transactions' performance becomes worse. This negative effect is more noticeable with CTM than that with OTM.

The performance implication of global deadlock detection algorithms has been analyzed with the third set of experiments. The considered algorithms have not shown significant performance differences. The performance of PCG has been observed to be a little bit better than the other algorithms under high levels of data contention.

With the final set of experiments, we have compared the performance of the extended transaction model and the classical transaction model. We have observed that under low resource contention, our extended transaction model outperforms the classical transaction model as the data conflict ratio among transactions becomes higher. The lower global abort ratio of the extended transaction model is the main reason for its better performance. On the other hand, the extended transaction model yields lower throughput for local transactions. Therefore, our execution model may not be suitable for the systems in which the fraction of local transactions executed is much more than that of global transactions.

Bibliography

[1] R. Agrawal, M.J. Carey, and M. Livny. Concurrency control modeling: Alternatives and implications. *ACM Transaction on Database Systems*, 12(4), 1987.

[2] R. Baldoni and S. Salza. Deadlock detection in multidatabase systems: A performance analysis. Technical Report 949, Istitut de Recherche En Informatique Systemes Aleatoires, 1995.

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

[4] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. Technical Report STAN-CS-92-1432, Department of Computer Science, Standford University, 1992.

[5] Y. Breitbart, W. Litwin, and A. Silberschatz. Deadlock problems in a multidatabase environment. In *COMPCON*, 1991.

[6] A.P. Buchmann, M.T. Ozsu, D. Georgakopoulos M. Hornick. and F.A. Manola. A transaction model for active distributed object system. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufman Publ., 1992.

[7] A. Deacon, H. Scheck, and G. Weikum. Semantic-based multilevel transaction management in federated systems. In *10th International Conference of Data Engineering*, pages 452–461, 1994.

[8] W. Du and A. Elmagarmid. Quasi serializability: A correctness criterion for global concurrency control in heterogeneous distributed database system. In *15th International Conference on Very Large Databases*, pages 347–355, 1989.

[9] A. Elmagarmid, Y.Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *16th International conference on Very Large Databases*, pages 507–518, 1990.

[10] A. Elmagarmid and A. Zhang. Modelling flexibility in distributed transactions. Technical Report CSD-TR-93-060, Department of Computer Science, Purdue University, 1993.

[11] A. Elmagarmid and A. Zhang. A theory of global concurrency control in multidatabase systems. *VLDB Journal 2*, 2(3):331–360, 1993.

[12] H. Garcia-Molina and K. Salem. Sagas. In *ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[13] D. Georgakopoulos, M. Rusinkiewicz, and A.P.Sheth. Using tickets to enforce the serializability of multidatabase transaction. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):166–180, 1994.

[14] J. Huang, S. Hwang, and J. Srivastava. Concurrency control in federated database systems: A performance study. Technical Report TR93-15, Department of Computer Science, University of Minnesota, 1993.

[15] E. Levy, H. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *ACM SIGMOD International Conference on Management of Data*, pages 88–97, 1991.

[16] S. Mehrotra, R. Rastogi, H.F. Korth, and A. Silberschatz. Non-serializable execution in heterogenous distributed database systems. In *1st International Conference on Parallel and Distrubuted Information Systems*, 1991.

[17] S. Mehrotra, R. Rastogi, H.F. Korth, and A. Silberschatz. A transaction model for multidatabase systems. Technical Report TR-92-14, Department of Computer Science, University of Texas at Austin, 1992.

[18] J.E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[19] M. Rusinkiewicz, P. Krychniak, and A. Cichocki. Towards a model for multidatabase transactions. Technical Report UH-CS-92-18, Department of Computer Science, University of Huston, 1992.

[20] W. Schaad, H.J. Schek, and G. Weikum. Implementation and performance of multi-level transaction management in a multidatabase environment.

In *5th International Workshop on Research Issues on Data Engineering*, pages 108–115, 1995.

[21] P. Scheuerman, H. Tung, and C.K Teng. Performance analysis of two global deadlock detection algorithms for multidatabase systems. Technical report, Department of Electrical Engineering and Computer Science, Nortwestern University, 1992.

[22] P. Scheuermann and H. Tung. A deadlock checkpointing scheme for multidatabase systems. In *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 184–191, 1992.

[23] H. Schwetman. *CSIM User's Guide.* MCC Technical Report Number ACT-126-90, 1990.

[24] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring relaxed atomicity for flexible transaction in multidatabase systems. In *ACM SIGMOD International Conference on Management of Data*, pages 67–78, 1994.