Branching Transactions: A Transaction Model for Parallel Database Systems

Albert G. Burger

PhD

The University of Edinburgh
1996

To my parents: *Josef and Rosa Burger*

and my wife: *Jessica*

Declaration

I hereby declare that the work described in this thesis is my own and that I have composed this thesis. This work was carried out under the supervision of Dr. Peter Thanisch.

Material included in the thesis has been published in a paper at the 12th British National Conference on Databases (BNCOD12) [16].

Acknowledgements

I would like to thank my supervisor, Dr. Peter Thanisch, for his excellent guidance and advise throughout this thesis. His expertise in parallel computing and database systems has been most beneficial to my work. I am particularly thankful for his strong interest in my work on branching transactions, the time he was prepared to spend with me in the many discussions we had, and his invaluable feedback on the various drafts of this document.

I also would like to thank Dr. Stephen Gilmore, my second supervisor, for his help. In particular, his comments on the correctness proofs included in this thesis were most appreciated. Furthermore, thanks go to Dr. Rob Pooley for his help and advise on the use of Simula and Demos.

Without the strong support from my parents throughout my entire PhD studies, it would have simply been impossible for me to obtain this degree. To them I can only say *Ein herzliches Vergeltsgott!*.

Finally, I would like to thank my wife, Jessica, who in spite of having had to sacrifice various of her own interests in favour of my studies, has always been wonderfully supportive throughout these years.

Abstract

In order to exploit parallel computers, database management systems must achieve a high level of concurrency when executing transactions. In a high contention environment, however, concurrency is severely limited due to transaction blocking, and the utilisation of parallel hardware resources, e.g. multiple CPUs, can be low. In this dissertation, a new transaction model, *Branching Transactions*, is proposed. Under branching transactions, more than one possible path of execution of a transaction is followed up in parallel, which allows us to avoid unnecessary transaction blockings and restarts. This approach uses additional hardware resources, mainly CPU — which would otherwise sit idle due to data contention — to improve transaction response time and throughput.

A new transaction model has implications for many transaction processing algorithms, in particular concurrency control. A family of locking algorithms, based on multi-version two-phase locking, has been developed for branching transactions, including an algorithm which can dynamically switch between branching and non-branching modes. The issues of deadlock handling and recovery are also considered. The correctness of all new concurrency control algorithms is proved by extending traditional serializability theory so that it is able to cope with the notion of a branching transaction.

Architectural descriptions of branching transaction systems for shared-memory parallel databases and hybrid shared-disk/shared-memory systems are discussed. In particular, the problem of cache coherence is addressed. The performance of branching transactions in a shared-memory parallel database system has been investigated using discrete-event simulation.

One field which may potentially benefit greatly from branching transactions is that of so-called "real-time" database systems, in which transactions have execution deadlines. A new real-time concurrency control algorithm based on branching transactions is introduced.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1983, Boral and DeWitt [15] predicted that there would be no future for parallel database machines. Contrary to that statement, however, in the ten years that followed, Teradata and Tandem — two companies marketing and selling parallel database systems — were very successful. Partially, Boral and DeWitt's prediction was wrong, since they could not foresee at that time that the newly introduced relational data model, which is ideally suited for parallel database systems, was about to become the dominating database technology in the market.

In 1992, another paper [98] critical of the need of parallel database technology suggested that most commercial applications do not need highly parallel database systems since modern processors are fast enough to cope with them. We believe that this assessment is similarly flawed as the one by Boral and DeWitt in 1983: it does not consider the emergence of new application types which will want to take advantage of the services, e.g. query processing and transaction management, of a database management system; and it does not consider growing requirements for transaction throughput due to an enterprise's business growth.

Looking at the current market situation, it is obvious that industry considers parallel databases as a lucrative area for profits; all major commercial database vendors have or are working on ports of their DBMS for various parallel computer platforms. They are also identifying possible new application areas for database systems. One such area is "media server"; a media server application typically involves audio and video data and must satisfy near real-time performance constraints.

Hence, in spite of the occasional doubtful paper, most indicators today suggest that parallel databases will occupy a considerable share of the database market in the future.

The primary motivation to use parallel computers for database applications is performance, which typically is either expressed in the average number of transactions a DBMS can process per second (*throughput*) or the average time required to execute a transaction (*response time*). Both are important parameters. For example, a bank customer using an ATM to retrieve money from his/her account expects to receive a response from the ATM within a few seconds after having typed in the request. Long delays at the ATM clearly would not be tolerated by the customer. Fast response time alone, however, is not sufficient. It is equally important to have a large enough number of ATMs available in a given area. Large numbers of ATMs may lead to many requests being made simultaneously. The throughput of a system describes how many such transactions the system is able to cope with within a certain period of time.

Many of the algorithms used in parallel DBMS today were originally designed for single-processor or distributed computer systems and then adapted to a parallel environment. Although performance improvements have been achieved this way, we believe it is important to take a fresh look at today's DBMS and decide whether or not a redesign — tailored to the particular aspects of modern parallel computer architectures — of some of their components might be beneficial. One such area to be looked at is transaction management and concurrency control.

A key component of most major DBMS products is their transaction management subsystem, which is responsible for the correct scheduling of transactions and the consistency of the database in spite of possible failure situations, e.g. system or media failure. A transaction, when executed in isolation, is assumed to transform the database from one consistent state to another, but serial execution of transactions is highly inefficient and most database systems, therefore, execute multiple transactions simultaneously. Not all concurrent schedules of transactions can be allowed, however, or else the database may become corrupt. It is the responsibility of the *Concurrency Control Manager* (CCM) to prevent inconsistent schedules.

Although many concurrency control algorithms have been proposed in the past (see [10] for an overview of basic techniques), there is still a strong interest in this area. In particular, in the context of parallel database systems, concurrency control can be a key factor in preventing a DBMS from exploiting parallel hardware resources to the fullest. *Data contention* leads to transactions being blocked, i.e. waiting for access permission to data which is currently accessed by other transactions. A number of studies have shown that under two-phase locking — the most commonly used concurrency control algorithm — data contention can be the cause of system performance degradation [5,6,21,37,48,86].

The central objective of this dissertation is to address this issue of data contention in parallel database systems. In particular, we propose a new transaction model, *branching transactions*, which has been designed to reduce data contention by taking advantage of the higher availability of CPU resources in a parallel computer system.

The key issue of almost all previous concurrency control algorithms is what we refer to as the "wrong decisions" problem. These algorithms take a particular action at the time when two transactions have a database access conflict. Pessimistic algorithms either *block* or *restart* the transactions involved, whereas optimistic algorithms allow them all to proceed, but may have to *abort* and *restart* transactions later if their verification fails. Since these decisions are based on incomplete knowledge — not until some time after the decision has been made does it become clear whether or not it is correct — some eventually are found to be "wrong".

A "wrong decision" does not compromise the consistency of the database, but imposes a performance penalty on the system. Blocking causes response time problems since the execution of a transaction is halted for some period. Aborting and restarting also prolongs response time since a transaction must start execution all over again from the beginning after is has been aborted. Furthermore, restarts lead to extra overhead for resources, e.g. CPU and I/O, since part or all of the execution of a transaction must be repeated.

Our new transaction model is designed to avoid making the "wrong decision" to block a transaction by simultaneously following up alternative execution paths for a transaction. Once it becomes clear which alternative is correct, all others can be discarded. The key goals of our model are: that the successful branch of a transaction can be executed with 1) minimal delay (due to waiting for locks), and 2) a minimal number of restarts.

Since none of the existing transaction models are able to support the idea of branching, the traditional flat model has been extended to incorporate the notion of alternative paths of execution. To illustrate the uniqueness of our new model, we compare it with a variety of previously suggested models, e.g. nested transactions, distributed transactions, multi-level transactions, split transactions and flex transactions.

Existing transaction processing algorithms cannot simply be transferred to a new transaction model such as branching transactions. Hence, new algorithms for concurrency control, transaction commit and logging and recovery had to be developed.

A new multi-version two-phase locking algorithm is proposed which supports the notion of alternative transaction branches. As in traditional multi-version algorithms, a Write operation to the database yields a new version of the corresponding data item. A Read request may result in branching of the reader transaction if currently more than one version of the requested item exists.

We recognise that unrestricted branching may lead to an exponential growth of the number of transaction branches simultaneously executed. Even a very powerful parallel computer may not be able to cope with too much branching. Hence, we propose several strategies to limit the maximum level of branching allowed. We describe a *static* method, in which case each transaction is limited to a pre-determined number of branches, and a more *dynamic* view, in which branching is made dependent on the availability of system resources (primarily CPUs). A hybrid approach is also feasible.

To allow dynamic switching between the branching and non-branching modes of transaction execution, we extend our original concurrency control algorithm into a hybrid branching algorithm which merges single-version two-phase locking, multi-version two-phase locking and branching transaction multi-version two-phase locking into one single concurrency control policy (we refer to this new algorithm as HBT-MV2PL).

A common problem with proposals for new concurrency control mechanisms is that they require a substantial rewrite for many of the components of the concurrency control system. To reduce this problem for branching transactions, we suggest a two-layer approach for the development of an HBT-MV2PL concurrency control manager, where the lower layer captures general (non-branching) 2-phase locking, and the higher layer provides full HBT-MV2PL functionality. Separating the scheduler in these two components supports the possibility of reusing existing 2PL scheduler code.

As with all locking algorithms, HBT-MV2PL may lead to deadlocks. The notion of a deadlock is redefined in the context of branching transactions; although there be may a deadlock between a set of branches, the transactions to which these branches belong are not necessarily deadlocked as well. We take this into consideration when defining wait-for graphs in this new context. New deadlock detection and resolution algorithms are proposed for branching transactions.

I/O overhead for logging should be kept to a minimum. We investigated alternative solutions and decided to apply an incremental log with deferred updates strategy [10]. Using this technique, we are able to avoid any extra overhead due to branching of transactions. Since log records are written to disk at commit time only, and the correct branch of execution of a transaction is known at that time, no logging or undo I/O overhead is caused by transaction branches which are aborted.

To illustrate how branching transactions can be used in a parallel computer environment, an overall system architecture for our new model is proposed. We focus on two parallel hardware platforms: shared memory parallel computers and shared something systems (multiple shared memory nodes — connected via some interconnect — which are sharing disks) [84,93]. System architectures are described in terms of their component resource managers and the interaction between them.

In the case of shared something systems, we identify how load balancing can be achieved; load balancing in a shared memory environment is comparatively easy and assumed to be handled by the underlying operating system. A further complication in the case of shared something is cache coherence. To speed-up access to data, cache memory is used to buffer database pages. Copies of the same database item in more than one (shared memory) node can lead to coherence problems. Since in a branching transaction system committed as well as uncommitted data may be copied at more than one node, cache coherence is an even more complicated issue.

Following the terminology used by Rahm [75], our cache coherence scheme for committed data could be classified as using on-request invalidation, selective notification, some sort of page sequence numbers, horizontal propagation for cache page updates, and force or no-force disk update. We apply a similar scheme to maintain cache coherence of uncommitted data, although no disk update strategy is required since uncommitted data is never written to disk. The two schemes interact with each other to form a new, combined cache management mechanism for a branching transaction system.

As a concrete example we use the Convex Exemplar — a parallel computer which contains a number of connected shared-memory nodes — and map our proposed system architecture onto its hardware and system software environment.

With the introduction of any new transaction model and concurrency control algorithm, the issue of correctness arises. Unless we are certain that our new mechanisms do not permit interleavings of transactions which can corrupt the database, they are of little use. Hence, a formal proof of correctness is required. Traditional serializability theory [10] is not powerful enough to cope with the concept of branching transactions. After considering a number of alternative proof systems, e.g. I/O Automata [59] and ACTA [26], we decided to develop an extension of traditional serializability theory.

We introduce the notions of basic agents, agents and agent histories which allow us to incorporate internal structure (branching) of transactions in the formal description of concurrent transaction execution. Based on these, we define the concept of a branching transaction history (BTH). By formalising the properties of our concurrency control algorithms we can prove that the committed projection of a BTH is always equivalent to some (non-branching) multi-version history (as described in Bernstein et al. [10]). With the help of existing work on such multi-version histories we are then able to prove the correctness of our new transaction model and concurrency control algorithms.

To gain some insights into the performance behaviour of branching transactions, a simulation study has been carried out for the shared-memory architecture case. After demonstrating the problem of data contention in a parallel database system, the performance of a branching transaction system is compared with that of flat transactions using "normal" (non-branching) two-phase locking. The study shows that significant improvements can be achieved using branching transactions, even if CPU resources are limited.

One particular area in which we believe branching transactions to be potentially of great benefit is real-time scheduling[1]. We propose a new real-time concurrency control algorithm which is designed to minimise tardiness of transactions (due to data conflicts) through the use of branching transaction's capability of parallel execution of alternative paths. This new algorithm also incorporates the idea of delayed commits, which has been shown to be beneficial in the context of real-time concurrency control [3]. We do not attempt to commit a transaction once it has reached the end of execution, but delay its commitment until shortly before its deadline. Under the branching transactions model, most locks on items held by a

[1]A comprehensive, annotated bibliography on real-time database systems, including concurrency control, can be found in [92].

transaction prior to its commit phase do not conflict with other transactions, and hence, we minimise the impact of transactions with later deadlines on transactions with earlier deadlines.

The remainder of this dissertation is organised as follows. Chapter 2 provides a general introduction to the area of transaction processing and parallel database systems. In particular, it discusses the issues of DBMS system architectures, concurrency control and recovery, as well as aspects of performance evaluation of database systems. The key concepts of the branching transaction model and our motivation is described in Chapter 3. Also, part of this chapter is dedicated to a comparison of our new transaction model with other, related work. Similarities and differences with existing methods are discussed and the uniqueness and novelty of our approach established. Chapter 4 mostly describes the development of new concurrency control algorithms for branching transactions. In addition, it includes a discussion of deadlock handling and logging and recovery. Possible system architectures for branching transactions for parallel computer hardware platforms are presented in Chapter 5. In particular, we discuss shared memory and shared something architectures. Load balancing and load control issues are considered, and a cache coherence protocol introduced. This chapter uses the Convex Exemplar as a concrete hardware platform example in its descriptions. Chapter 6 contains an introduction to traditional serializability, extensions of it for branching transactions, the formalisation of branching transactions and a correctness proof of our new algorithms. Our performance study of branching transactions is presented in Chapter 7. Branching transactions for real-time scheduling are discussed in Chapter 8. The dissertation is concluded in Chapter 9; the primary achievements of this dissertation are summarised and future work discussed.

Chapter 2

Transaction Processing and Parallel Database Systems

In this chapter, we give an introduction to the field of transaction processing in centralised, distributed and parallel databases. We also discuss various possibilities for performance evaluation of such systems. The information given in this chapter should provide sufficient background information for the reader to understand the context in which the work described in this dissertation has been carried out. For a comprehensive treatment of the field of transaction processing the reader is referred to Gray and Reuter [40].

2.1 Transactions

A fundamental requirement for database management systems is their support of multi-user access to shared information. Several users must be allowed to simultaneously query and modify the content of a database. To avoid interference between these users, synchronisation techniques must be applied, or else the database may be left in an inconsistent state. Furthermore, the database management system should be able to maintain the consistency of a database in spite of various failure situations (which will be described in more detail below). To satisfy these requirements the notion of *transactions* was introduced [40,41].

Definition 1 *A transaction is the execution of a program that includes database access operations and has the following properties:*

Atomicity: *a transaction is either executed entirely or not at all.*

Consistency: *a correct execution of a transaction transforms the database from one consistent state to another.*

Isolation: *even though transactions execute concurrently with transaction T, it appears to T that each other transaction either executed before or after T.*

Durability: *changes made to the database by a committed transaction will not be lost through any subsequent failures in the system.*

It should be noted that the given definition does not exclude the possibility of temporary inconsistent states during the execution of a transaction. Through isolation, however, such inconsistencies are not visible to other transactions. To give a better understanding of the atomicity property it is helpful to discuss the transaction state transition diagram presented in Figure 2–1.

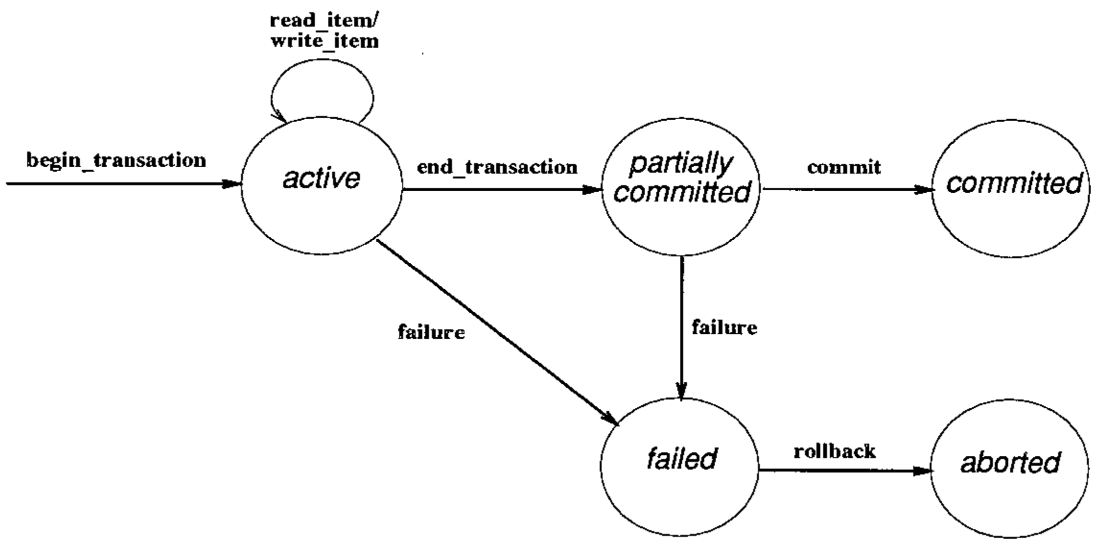


Figure 2–1: Transaction State Transition Diagram

In our description, we assume the following simple structure for transactions. After a transaction has been initialised through the execution of a *begin_transaction* statement it accesses the database through *read_item* and *write_item* operations. During this phase the transaction is *active.* An *end_transaction* statement indicates to the database management system that the transaction has finished its execution — it has *partially committed* — and all modifications to the database should now be made permanent. If these changes can be applied to the database, the transaction *commits.*

Sometimes, however, transactions fail due to various reasons, for example because their execution would violate serializability. Since a transaction failure may occur while the database is in an inconsistent state, any changes made to the database must be undone, i.e. the state of the database prior to the execution of the transaction must be restored. The process of undoing a transaction is called *rollback.*

A transaction terminates in either the *committed* state (the transaction has been executed entirely) or the *aborted* state (the effects are as if the transaction had never executed). In either case, the atomicity property is satisfied.

In addition to other database management system components, e.g. the query optimiser and the data manager, there exist two modules, the *concurrency control manager (CCM)* and the *recovery manager (RM)*, which are responsible for guaranteeing the ACID properties of transactions. While isolation and consistency (and serializability) are enforced by a CCM, the RM ensures atomicity and durability. The details of these modules are somewhat dependent on the underlying system architecture on top of which the DBMS is operating. Hence, before discussing more about CCMs and RMs, we briefly sketch the differences between centralised, distributed and parallel database systems.

2.2 Database System Architectures

2.2.1 Centralised Database Systems

The simplest environment to deal with is a *centralised database system*, where one DBMS is running on a single computer, all data are held on local disks and all users use this computer to access the database. Centralised systems range from simple, one-user, PC-based databases to large mainframe databases[1] which are shared by many users. All DBMS modules, such as the concurrency control manager and the recovery manager, use "centralised" algorithms, i.e. these modules do not need to communicate with any other computers.

2.2.2 Distributed Database Systems

Although mainly centralised database systems have been used in the past (and still are today for small database applications), for some time now there has been a trend away from centralised, mainframe systems to more distributed solutions for medium to large-sized organisations. The de-centralised structure of modern companies is better served by a database system which operates across multiple computers which are connected through some communication network[2]. Such a database system is usually referred to as a *distributed database system* [7,24,70].

---

[1]Mainframes are likely to be Symmetric Multi-Processor (SMP) machines — more details on SMPs later in Chapter 5 — but this does not significantly impact on the DBMS architecture.

[2]With the proliferation of powerful parallel database systems which can be configured into several, fairly independent, database subsystems, a new trend has emerged: distributed databases are being consolidated within one parallel database server. Consolidation and distribution are somewhat contradictory developments and at this stage it is not clear how these trends will evolve in the future database markets.

Distributed database systems are difficult to describe within a single definition. Instead, it may be better to give a brief description of the three most common forms of distributed database technology.

- Homogeneous distributed database systems
- Client/Server architectures
- Multi-database systems

Homogeneous Distributed Database Systems In a homogeneous distributed database system each node runs the same DBMS software. Nodes are not autonomous and follow the same protocols throughout the system. Note that this does not exclude the possibility that the underlying hardware and operating systems are heterogeneous. To the user the system appears as one logical database system — the actual distribution of the DBMS and the database is transparent to him/her. Data are located at various nodes of the system and may for performance reasons even be replicated at one or more nodes.

Although this kind of distributed technology has been given enormous attention by the database research community, it has not had a very significant success in industry. A homogeneous distributed database system does not grow out of a number of already existing database systems, but needs to be developed and planned as one entity. In most companies, however, a number of initially independent, frequently heterogeneous, database solutions exist prior to the wish of integrating all of them into one coherent database system.

Concurrency control and recovery are more complicated than in centralised systems. We will discuss them in more detail later in this chapter.

Client/Server Architectures The *client/server architecture* is currently the most common form of distributed database technology. As the name indicates, the database system is divided into *clients* and *servers*. There must be at least one server and one client, which typically reside on different computers (connected by some form of network, e.g. LAN). The server provides standard services such as storage and access to the database, transaction management and communication interfaces. The client computer runs applications which request services, e.g. access to the database server, as needed. The reader can find several popular accounts of client/server computing, and its impact on database systems technology, in the bibliography; see [28] [60] [68].

We categorise client/server architectures into three groups, depending on whether the client performs query processing, and whether data from the database is buffered by the client. In the first group, the client merely requests data from the server. No data is buffered locally and all query processing is performed by the server. A client in the second group receives data from the server and performs some query processing locally. However, no data is buffered by the client and each request to the database requires access to the database server. In the third group, data from the server is actually buffered and a client only needs to access the server if the data needed is not in its local buffer.

An important characteristic of these client/server systems is their capability to deal with heterogeneous environments. It is relatively easy to integrate clients and servers running on different platforms.

If there exists only one server in the system, then concurrency control and recovery mechanisms developed for centralised systems can be applied. Distributed techniques are required if more than one server is active and a single transaction can access data from more than one server.

Multidatabase Systems The development from centralised to distributed systems often happens gradually. In most cases initially a number of different, independent databases exist and a need to integrate them arises with the introduction of a new application. Unfortunately, not only are these systems independent, but they also frequently use different software (including different DBMS) and hardware. The system that results from the integration of such heterogeneous, independent database systems is known as a *multi-database system.*

To provide the user with one logical interface and to make the heterogeneous and distributed nature of the system transparent, an additional software layer is built on top of the various existing DBMS. A significant difference between multi-databases and homogeneous distributed databases is the level of autonomy at each site. In a multi-database environment it is possible to simply run an application on one node without making use of services provided by the multidatabase software layer. At any node a mix of local transactions and global transactions (issued through the multi-database software) may run concurrently. In a distributed homogeneous system all transactions are always managed by the distributed DBMS.

The most difficult problems in multi-database systems arise from the autonomy of nodes. It leads to a mixture of different transaction management systems, concurrency control mechanisms, recovery schemes, and database schemes.

This thesis does not deal with problems of multi-database systems, and therefore, for further details on this topic, the reader is referred to [58], [79] and [87].

2.2.3 Parallel Database Systems

Unlike distributed database systems (DDBS), which typically run on a number of workstations which communicate through a LAN or WAN and are possibly hundreds or thousands of miles away from each other (e.g. airline reservation systems), in *parallel database systems* all processing elements and disks are located closely together.

Parallel database systems can be categorised according to their underlying hardware characteristics. In [30] parallel architectures are divided into 3 groups:

shared-memory: all processors share direct access to a common global memory and to all disks.

shared-disks: each processor has a private memory but has access to all disks.

shared-nothing: each processor has a private memory and one or more disks; processors act as servers for data on disks owned by them.

It should be noted that shared-memory architectures as described above are different from so-called *distributed shared memory* (DSM) systems [73]. The above definition of shared-memory implies *physically* shared memory by processors, whereas a DSM system only gives the illusion that there exists a global shared memory by providing a global address space (in spite of having physically distributed memory).

Some influential researchers in this field stated that a shared-nothing architecture is the preferred option for parallel databases [30,84], and as a result a significant amount of research work has focussed on shared-nothing systems. Although a shared-nothing architecture seems very suitable for large relational databases, other types of workloads may benefit from different architectures. Valduriez [93] makes the case for a shared-disk system where each node itself is a shared-memory multi-processor. He refers to such an architecture as *shared-something.* Norman et al. [66] point out that in spite of marketing statements in support of shared-nothing architectures, quite often the underlying hardware of such "shared-nothing" systems displays some level of sharing, e.g. disks can be accessed by more than one node.

Of the major parallel DBMS products, examples of shared-nothing DBMS architectures are Sybase, Tandem and Teradata. Examples of shared-disk architectures are OpenIngres, Oracle7 and Red Brick. Other leading products, such as Informix and DB2, cannot be simply put in either classification.

We will discuss various architectures in more detail in Chapter 5 in the context of DBMS software architectures for branching transactions.

2.3 Concurrency Control

2.3.1 Concurrency Control in Centralised Database Systems

Although the average transaction workload is known for most database applications, in general individual arrivals of transactions follow some random distribution and cannot be predicted by the database management system. The *concurrency control manager* dynamically has to decide whether a requested access to the database is going to violate serializability and what actions have to be taken. A great number of algorithms have been proposed (see [10] for a survey) to solve this problem. In general they fall into two categories.

- locking algorithms
- timestamp algorithms

Each of these groups can further be subdivided into:

- optimistic algorithms
- pessimistic algorithms

All techniques described in this section are used to achieve serializable schedules.

Locking Techniques for Concurrency Control Locking mechanisms achieve transaction synchronisation through mutual exclusion of data accesses. Before a transaction can access a data item, it has to acquire an appropriate lock for that data item. If a *read_item(X)* operation is requested, a read-lock on X must be obtained first. Several transactions can hold a read-lock on X simultaneously, and hence, read-locks are usually referred to as *shared locks.* A transaction wanting to modify (write) a data item X must first acquire a write-lock on X. Since only one transaction can have a write-lock on X at any time, and no shared-locks are allowed simultaneously, write-locks are called *exclusive locks.* If a transaction cannot be granted the lock it requested, it must wait until the transaction currently holding a lock on the item releases (unlocks) that lock.

If each transaction acquires and releases locks in two distinct phases, i.e. all locking operations precede the first unlock operation of a transaction, the schedules which result from such a policy will always be serializable [10]. Concurrency control algorithms of this kind are called *two-phase locking.* To guarantee transaction isolation, all locks obtained by a transaction during execution are usually collectively released at commit time of that transaction.

Two-phase locking algorithms, however, may lead to deadlocks, which we illustrate with the example given in Table 2–1. First, transaction T_1 requests and acquires a shared lock on item X (step 1). Then T_2 requests and acquires a shared lock on item Y (step 2). When T_1 tries to get an exclusive lock on Y it gets blocked (step 3), since T_2 has a shared lock on Y already; analogous step 4. In this situation, both transactions are waiting for each other to release a lock and the system has entered a deadlock.

Step	*Transaction T_1*	*Transaction T_2*	*Concurrency Control Manager*
1	read_lock(X)		grant shared lock on X for T_1
2		read_lock(Y)	grant shared lock on Y for T_2
3	write_lock(Y)		block transaction T_1
4		write_lock(X)	block transaction T_2

Table 2–1: Transaction deadlock in locking protocol

Algorithms which apply deadlock detection and resolution techniques are called *general two-phase locking.* There exist other algorithms which use deadlock prevention techniques, e.g. *wound-wait* and *wait-die* [77].

Timestamp Techniques for Concurrency Control Timestamp algorithms [8] are based on the notion of a unique timestamp which is created by the DBMS to identify a transaction. Transactions usually use their startup time as their timestamp. We refer to the timestamp of a transaction as $TS(T)$. Each data item in the database has two timestamps associated with it, following the rules given below (taken from [33]).

1. The *read timestamp* of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X.

2. The *write timestamp* of item X is the largest of all the timestamps of transactions that have successfully written item X.

Timestamp ordering is a concurrency control technique that enforces a schedule equivalent to a serial schedule which has all transactions in the order of their timestamp values. The algorithm applies the following rules. ($read\_TS(X)$ is the read timestamp for item X, $write\_TS(X)$ is the write timestamp for item X). Again these descriptions are given in [33].

1. Transaction T issues a $write\_item(X)$ operation:
   - if $read\_TS(X) > TS(T)$, then abort and roll-back T and reject the operation.
   - if $write\_TS(X) > TS(T)$, then do not execute the write operation, but continue processing.
   - if neither of the two conditions above are true, then execute the write_item(X) operation of T and set $write\_TS(X)$ to $TS(T)$.
2. Transaction T issues a read_item(X) operation:
   - if $write\_TS(X) > TS(T)$, then abort and roll-back T and reject the operation.
   - if $write\_TS(X) \leq TS(T)$, then execute the read_item operation of T and set $read\_TS(X)$ to the larger of $TS(T)$ and the current $read\_TS(X)$.

It is possible to increase the level of concurrency by keeping more than one version of the same data item in the database. A timestamp ordering algorithm based on this principle is called *multi-version concurrency control* [77]. Under this scheme, write operations create new versions of data items, rather than overwriting the old values. A write operation is rejected if it arrives too late. To decide whether a write_item(X) operation by transaction T is too late, the version of X with the highest timestamp smaller than $TS(T)$ is determined. If for this version there exists a read timestamp $> TS(T)$, then the operation write_item(X) arrived too late. Read operations are never rejected. If a transaction T wants to read a version which was created by another uncommitted transaction T', T has to wait until T' either commits or aborts. In the latter case the read operation is scheduled for another version. Although additional overhead is caused through the maintenance of multiple versions, various studies [17,23,57,82] have shown that a significant performance advantage can be achieved through multi-version techniques. Multi-version concurrency control algorithms also exist for locking protocols. We will discuss multi-version two-phase locking in more detail in Chapter 4.

Pessimistic vs. Optimistic Algorithms Both locking and timestamp ordering algorithms can be *pessimistic* or *optimistic*. The algorithms described above are pessimistic, i.e. when a request is made, the concurrency control mechanism immediately decides whether or not it can be granted. Optimistic algorithms [55] do not interfere with the execution of a transaction until it tries to commit, at which point the concurrency control manager will try to verify that no violation of serializability has happened, and if not, the transaction can commit. Otherwise the transaction is rolled-back. Optimistic concurrency control algorithms perform well under low data contention, i.e. for workloads where there is little interference between transactions [22].

2.3.2 Concurrency Control in Distributed Database Systems

Distributed systems are designed in such a way that no single node needs to know the current status of the global system. Nodes have to communicate with each other to find out about the activities at other nodes. This *partial information* problem makes concurrency control in distributed database systems much harder. The following description of *distributed serializability* will show why.

Distributed Serializability As in the centralised case, for a transaction schedule in a distributed system to be correct it must be serializable. The problem, however, is that it is not sufficient to enforce a serializable schedule at all nodes. Assume, for example, that transactions T_1 and T_2 are both running on nodes N_1 and N_2. At node N_1 a schedule may be produced which is equivalent to a serial execution where T_1 is executed before T_2. At the same time, at node N_2 a schedule may be produced which is equivalent to a serial schedule in which T_2 is executed before T_1. While both local schedules are serializable, there exists no single global serial order which can satisfy both local schedules, and hence, the execution is not serializable. In a distributed system any ordering of transactions must be the same at *all* nodes.

Data Replication An additional difficulty is the possibility of data replication. In a distributed system it may be beneficial for availability and reliability purposes to keep more than one copy of a data item at different nodes. The CCM has to ensure that no inconsistencies between copies emerge.

Classification of Distributed Concurrency Control Algorithms To deal with the problems of global (distributed) serializability and data replication, three possible solutions were proposed:

- *Centralised Approach:* One site in the system is responsible for all concurrency control decisions. All other nodes have to communicate with the designated node if they want to access the database. The problem is basically reduced to that of a centralised database system, and all techniques discussed previously in Section 2.3.1 can be applied.

- *Primary Copy:* For each data item one copy — recall that more than one may exist due to data replication — is designated as the primary copy. All concurrency control for that item is handled by the primary copy. Again, the problem is very similar to a centralised case, and the appropriate mechanisms can be used.

- *Distributed Approach* Under this approach all nodes share equal responsibility for concurrency control. If data items are not replicated, this method is identical to the primary copy approach. In case of data replication, the following two protocols can be used. Both mechanisms ensure that a conflict between two transactions will always be detected.

- *Read-One-Write-All:* If a transaction wants to read an item it will read only one of the copies. If a transaction wants to update a data item, it has to update all existing copies of the item.
- *Majority Consensus:* Whether a transaction wants to read or update an item, it has to obtain the approval to do so by a majority of the nodes responsible for copies of this item.

Distributed Locking Algorithms Locking algorithms can be used with any of the aforementioned distributed techniques. An additional problem of distributed locking mechanisms is the possibility of distributed deadlocks. We will not discuss any details of distributed deadlocks in this introduction; for more detail the reader is referred to [51].

Distributed Timestamp Algorithms Centralised timestamp algorithms can easily be adapted to a distributed environment, by generating timestamps which are system-wide unique. This is usually done by concatenating a locally unique timestamp with the node number [56].

A more detailed discussion of distributed concurrency control algorithms can be found in [7,9,24,70].

2.3.3 Concurrency Control in Parallel Database Systems

There is no distinct set of concurrency control algorithms for parallel databases. Depending on the architecture of the system, either centralised or distributed concurrency control algorithms can be applied, perhaps with some modification. There exists, however, the added complexity of cache coherency for shared-disk architectures. We will discuss this in more detail in Chapter 5.

Although it is possible to provide a parallel implementation of a concurrency control algorithm — for example, one may wish to implement a parallel algorithm for centralised deadlock detection in a shared-memory system — we do not address implementation details of any algorithm presented in this dissertation, but focus on conceptual problems instead.

2.4 Recovery

2.4.1 Recovery in Centralised Database Systems

In an ideal environment transactions never conflict, the system never crashes, and no hardware failures occur. This, of course, is unrealistic and a DBMS has to provide special mechanisms to guarantee the transaction properties of atomicity and durability in spite of such failures. The part of a DBMS dealing with failures is called the *Recovery Manager (RM)*. We categorise failures into the following two groups.

- *Transaction failure:* a single transaction fails while the rest of the system remains operational. This may be caused by a concurrency control decision, the user aborting the transaction, or an error in the application program (e.g. division by zero).

- *System failure:* the entire database system fails. This may be caused by a power failure, operating system crash, or a serious hardware failure (e.g. CPU).

For either failure situation it is necessary to guarantee the atomicity of a transaction. Different techniques can be applied based on whether a transaction updates the database immediately or if updates are applied to the database at commit time. A more detailed discussion of the following techniques can be found in [52].

Incremental Log with Deferred Updates Under this scheme, no updates to the database are performed immediately. Instead, a log file is maintained which keeps a record for each write operation issued by a transaction. In case a transaction fails, all records (relating to this transaction) kept in the log are simply discarded. Since the database was not modified, no further actions are required to roll-back the transaction. In case the transaction partially-commits, the log records are used to apply all updates to the database. Log records are kept on disk, rather than in main memory, to ensure atomicity in spite of system failures.

Incremental Log with Immediate Updates All write operations by transactions are immediately performed on the database. For each item updated, however, a record is kept in a disk-based log file containing the value of the items prior to their modification. In case a transaction fails these log records are used to restore the original state of those items which were modified by the failed transaction.

Shadow Paging An alternative solution to log files is the use of shadow pages. The basic idea behind this approach is to keep two versions of a database page while a transaction is actively updating it. The *current page* is accessed and updated by the transaction while the *shadow page* is kept in its original state. If a transaction successfully commits, the current page becomes the new page in the database, otherwise the shadow page stays a part of the database.

In case of a system failure, any transaction that has not committed yet will be treated as a failed transaction when the system is restarted, and hence, rolled-back.

The recovery manager of a DBMS is also responsible for dealing with media failures, i.e. a disk crash. For a discussion of this area we refer the reader to the literature [7,33,52].

2.4.2 Recovery in Distributed Database Systems

In a distributed system, a transaction may modify data on more than one node, depending on where the data which the transaction wants to access are located.

A transaction that executes on more than one node during its lifetime is called a *distributed transaction.* Since atomicity is also an essential property for distributed transactions, the recovery manager has to ensure that a transaction either successfully commits at all nodes or not at all. Additional failure sources, such as site failure and communication failure, make this a complicated problem.

In addition to those techniques described in Section 2.4.1, a new family of protocols, named *atomic commit protocols*, have been developed to guarantee atomicity in a distributed system. The most common algorithm is called *two-phase commit protocol* (which is not the same as two-phase locking).

Two-Phase Commit Protocol The node where a transaction originates is called the *coordinator*, all other nodes visited by the transaction are *participants.* Once a transaction has partially committed, the coordinator sends a *prepare* message to all participants to ask whether they are ready to commit the transaction. If for any reason (e.g. concurrency control) a participant cannot commit the transaction it replies with an *abort* message, otherwise it sends a *ready* message. The coordinator collects all the answers from the participants, and if one (or more) participants cannot commit, the entire transaction is aborted, i.e. the coordinator sends *abort* messages to all participants who replied with a *ready* message. If all participants reply with a *ready* message, then the first phase of the protocol has successfully been completed. At this stage all participating nodes guarantee (using local recovery techniques) that they will be able to commit the transaction in spite of any failures that may occur. In the second phase the coordinator sends *commit* messages to all participants, which then commit the transaction at their node and acknowledge this to the coordinator with a *committed* message. Once the coordinator has received *committed* messages from all participants, the transaction is committed, and the protocol ends.

In case of a site failure during the two-phase commit protocol it may happen that the protocol is blocked until the corresponding site has recovered. An extension of two-phase commit was developed, called the *three-phase commit* protocol, which can avoid the blocking problem in case of a site failure. There are still problems in case of communication failures where all sites are still operational but the network has been divided into several partitions. We will not discuss these issues in detail; the reader is referred to [7,9,24,70].

2.4.3 Recovery in Parallel Database Systems

Recovery mechanisms developed for centralised and distributed systems can also be applied to parallel databases. A failure in a shared-memory or shared-disk system is usually treated as a total failure and handled by using centralised recovery mechanisms. In a shared-nothing system, using distributed recovery algorithms would allow the system to continue to operate in spite of individual node failures. Since such node failures, however, occur relatively rarely, it may be tolerable to interrupt regular database processing for recovery purposes, i.e. to treat a single node failure as a total system failure. The advantage of this would be a lower system overhead during normal, failure free, processing.

Although existing recovery techniques can be applied in parallel database systems, some work has been carried out which deals with the particulars of recovery in parallel database systems: for example, Keen and Dally [49] describe a protocol which reduces the problem of I/O costs for logging purposes in a parallel database system. Kumar et al. [54] show that multiple processors can be used to reduce logging overhead on transaction response time and speed up recovery after system failure.

2.5 Performance Evaluation of Database Systems

One of the issues in this thesis is the performance of concurrency control algorithms. These mechanisms significantly contribute to the overall efficiency of a database, and the performance of a particular algorithm can be understood best when studied in the context of an entire system. In this section we will, therefore, briefly introduce several aspects of database systems performance. We begin this discussion with a description of systems metrics used to express the performance of a database system.

2.5.1 Database Performance Metrics

A useful distinction between *global* and *internal* database performance analysis is made in [25]. In global analysis the performance metrics refer to the global system behaviour, i.e. the system is seen as a black box which produces certain outputs when given certain inputs. Examples of such metrics are *response time* and *throughput.* While end users are primarily interested in performance indexes produced through global analysis, researchers and developers try to understand the reasons behind a certain system behaviour. They are, therefore, looking inside the system (internal analysis). Internal analysis is used to determine the effect of particular internal hardware and software resources on global indexes (throughput and response time). Internal metrics such as *utilisation* and *average queue length* of system components are used to determine bottlenecks and poor utilisation of resources within the DBMS.

Although transaction response time, transaction throughput and utilisation are standard performance metrics in the database literature, we discuss them briefly for completeness.

Transaction Response Time is the time between the arrival of a transaction in the database system and the exit of the transaction from the system. The time elapsed between a request for withdrawal of money made by a bank customer using an ATM (automatic teller machine), and the dispensation of the money to the customer is a typical example for transaction response time.

Transaction Throughput is a measure of how many transactions are completed per given time unit, e.g. how many requests by customers can be processed by an ATM in a given unit of time. It indicates the quantity of information that can be processed by a given system in a given period.

In general, if longer transaction response times are acceptable, higher throughput can be achieved. There are, however, many applications where response time is expected to stay within certain limits, e.g. it is not very desirable to let a customer wait for 30 minutes before he receives his money from an ATM.

Utilisation indicates how intensively a particular resource, e.g. CPU, memory, IO or network, is used within the system. It is defined as the ratio between the length of time a resource was used during a given time interval and the length of that interval. Utilisation statistics help to identify bottlenecks in the system and/or inefficient use of resources (low utilisation).

2.5.2 Measuring Database Performance

If an actual implementation of a DBMS exists one can evaluate its efficiency by running a specific *target application* or a *benchmark* test on it and measure its performance. Researchers and developers, however, frequently have to make design decisions without implementing and analysing the alternatives first; it would be too expensive to do that. They, therefore, use models of database systems to study their performance. *Analytical models* and *simulation models* are developed to predict the efficiency of a particular algorithm or system design. We will examine each of these performance evaluation techniques in more detail next.

Test-runs of target applications The most accurate information about the performance of a database system can be obtained if the target application is actually implemented on the DBMS of interest. This, however, is usually rather expensive and can only be done in very few cases. An alternative to this approach is the use of benchmarks.

Benchmarks A benchmark is a program that creates an artificial workload for a DBMS. Measurements taken during the execution of a benchmark test give a fairly accurate picture of a DBMS' performance with regard to the workload modelled by that benchmark. There exist a number of database benchmarks, e.g. TPC-A, TPC-B, TPC-C, the Wisconsin Benchmark, the Set Query Benchmark, etc. (for a description of these benchmarks and further references see [39]), each trying to capture a different kind of workload. In spite of the variety of benchmarks that exist today, they are sometimes criticised for not accurately representing the workload of real-world applications [19]. Furthermore, benchmarks developed by DBMS vendors are often designed around that vendors' systems. If none of the existing benchmarks seems fit to represent a certain workload, implementing a new one is often a viable alternative. Although costs of developing a comprehensive benchmark can be around one million US dollars and higher, it is still cheaper to develop a new benchmark than a complete application.

Hitherto, we assumed the existence of the DBMS under consideration. As pointed out earlier, this frequently is not the case, and system models must be used instead.

Analytical and Simulation Models In stochastic models (analytical and simulation models), systems are represented as queueing networks where a separate queue is provided for every resource, and transactions arriving in the system and requesting services are modelled as a number of customers circulating in the network.

Analytical models are models for which exact or approximate solution algorithms exist. Their advantage is a relatively low computational overhead. They do, however, require a fairly sophisticated theoretical background (mathematics, queueing theory), and sometimes simplifying assumptions, needed to keep the system solvable, lead to unrealistic models.

Most work on analytical modelling in the area of concurrency control was done for centralised database systems (e.g. [86,85,90]), though some researchers have also suggested analytical models for distributed concurrency control algorithms ([99]). It would have been very difficult to extend the latter work to include branching transactions.

On the other side, simulation models are more intuitive than analytical models and allow the modeller to represent a system in as much detail as desired. This advantage is, however, balanced by the greater computational cost of running simulation programs.

A discussion of these performance measurement techniques can be found in [25].

Complexity of Database Algorithms

Another approach to evaluate performance characteristics of database algorithms is to determine their computational complexity. For example, in [71] it is shown that implementing a conflict graph scheduler for a given set of transactions on two sites in order to minimise communication is PSPACE-complete. [71] also shows that there is a tradeoff between the computational and communication cost for a distributed scheduler. A proof is given which shows that unless NP=PSPACE, there is no distributed scheduler which realizes serializability, operates in polynomial time, and uses the minimum possible number of messages. Complexity results of this kind are important, because they can identify performance problems of a particular design or architecture which are not merely the consequence of a poor implementation.

Chapter 3

Branching Transaction Model

We begin this chapter with a discussion of the data contention problem and the "wrong decisions" issue of traditional concurrency control mechanisms. This will lead us to the basic idea of branching transactions. The principle concepts of this new transaction model are presented and compared with existing similar approaches.

3.1 Data Contention

In an ideal environment, transaction throughput increases linearly with the number of transactions submitted to the database. A high transaction workload, however, can lead to the problem of thrashing. Thrashing is a well known issue and has been reported, in the case of operating systems, as early as 1968 [29]. Thrashing in the context of a database system can either manifest itself as *resource contention* or *data contention* [10]. Under resource contention most transactions are waiting for resources, such as processors, memory and I/O; under data contention most transactions are waiting for locks on data items.

As discussed in Chapter 2, the most commonly used concurrency control mechanism in today's database management systems is two-phase locking (2PL). Various studies have shown that 2PL can be the cause of system performance degradation due to data contention [5,6,21,37,48,86]. Since a parallel database system is running on a parallel hardware platform, resources such as processors, memory and I/O are more readily available than in uni-processor systems, and hence, a parallel database system is more likely to be bound by data contention than by resource contention. There are certainly transaction workloads where this argument does not apply: for example, decision support type of queries which perform mostly read operations across large databases do not cause many data conflicts between transactions, since multiple read operations on the same data item are allowed concurrently. We are, however, more interested in *O*n-*L*ine *T*ransaction *P*rocessing (OLTP) workloads, where a mix of read and write operations is leading to data conflicts.

3.2 The "Wrong Decision" Problem

In Chapter 2 we explained that almost all existing concurrency control algorithms make particular decisions at the time of data conflicts, i.e. the time when two transactions want to access the same data item. Pessimistic algorithms either *block* or *restart* one of the transactions involved, optimistic algorithms allow them all to proceed, but may have to *abort* and *restart* transactions later if their *verification* fails.

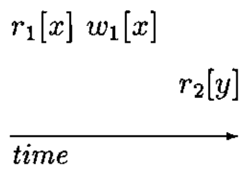
All these algorithms share a particular problem: at the time of conflict it is not known which decision is the best one to make; the best decision in this context is the one which guarantees serializability with the least negative impact on performance. Blocking and aborting transactions cause performance degradation. Pessimistic concurrency algorithms block and abort transactions, because otherwise serializability *might* be compromised or a deadlock *might* occur. There is no guarantee that this eventually really happens, and in case it doesn't, blocking or aborting is unnecessary. In an optimistic algorithm, a transaction, which is allowed to proceed after a data conflict, may later fail its verification. In this case, it would have been better to abort the transaction immediately at the time of conflict.

This "wrong decision" problem exists, since the scheduler, which carries out a particular concurrency control policy, is forced to make a decision at a time when it is not yet known which decision is best. In general, this can only be determined some time in the future. Since the scheduler cannot predict the future, it works with certain default assumptions. In the case of pessimistic algorithms, the assumption is that unless some action (blocking or abort) is taken, serializability will become compromised. In case of optimistic algorithms, the assumption is that most likely no serializability problem will occur in spite of a data conflict. In either case, if it later turns out that this assumption was incorrect, a "wrong decision" has been made. We illustrate the "wrong decision" problem with the following example.

Example 1 *Let us assume we have two transactions, T_1 and T_2 ($r_i[x]$ denotes a read operation by transaction T_i on item x; $w_i[x]$ denotes a write operation by transaction T_i on item x):*

$$T_1\text{: } r_1[x]\ w_1[x]\ w_1[z]$$
$$T_2\text{: } r_2[y]\ r_2[x]\ w_2[x]\ w_2[z]$$

<u>Basic Scenario</u> *Initially T_1 has read and written data item x and T_2 has read item y:*



The next operation to arrive at the scheduler is transaction T_2's read operation on x. At this point we have a data conflict, since T_1 has just written item x. Without any interference, i.e. no blocking and aborting of transactions, the following two scenarios might emerge from this point.

Scenario 1:

$$\begin{array}{lllllll} r_1[x] & w_1[x] & & & & w_1[z] & \\ & & r_2[y] & r_2[x] & w_2[x] & & w_2[z] \end{array}$$

$$\xrightarrow[\textit{time}]{\hspace{6cm}}$$

T_2 reads and writes item x, then T_1 writes z and finally T_2 writes z.

Scenario 2:

$$\begin{array}{lllllll} r_1[x] & w_1[x] & & & & & w_1[z] \\ & & r_2[y] & r_2[x] & w_2[x] & w_2[z] & \end{array}$$

$$\xrightarrow[\textit{time}]{\hspace{6cm}}$$

T_2 continues to execute all of its operations before T_1 writes item z.

The first scenario describes a serializable schedule, whereas the second one is an incorrect schedule which should not be allowed[1].

In case of scenario 1, if strict 2-phase locking was applied, T_2's read operation on x would have been delayed until after T_1 has finished. The blocking of T_2 would be an example of a "wrong decision". In case of scenario 2, under an optimistic concurrency control manager, the read operation of T_2 on x would not have been delayed. Since the schedule that follows is incorrect, transaction verification would have failed and at least one of the two transactions would be aborted and restarted. In this case, it would have been better immediately to abort T_2 at the time of conflict. Letting T_2 proceed here, is another example of a "wrong decision".

"Wrong decisions" are not wrong in the sense that they lead to incorrect schedules, which can leave the database inconsistent, but they impose a performance penalty on the system. In *branching transactions* we try to avoid this problem by delaying a decision until it is clear which is the best action to take. We discuss the principles of branching transactions next.

[1]Readers not familiar with the concept of serializability are referred to Chapter 6 for an introduction to this topic.

3.3 Principles of Branching Transactions

When a transaction wants to read a data item, it may be the case that there currently exist temporary (uncommitted) versions of that item in the system. Blocking the reader until these versions have committed may not be the best option, as we have seen in the above example. Simply to read an uncommitted version, however, may also be a problem: an abort of the transaction which wrote the version forces an abort of the reader transaction (cascading aborts). Whatever decision the scheduler takes in this situation, it may be a "wrong decision".

Since the correct decision becomes clear once the "writer" transaction terminates (either aborts or commits), we propose to delay any particular action until that point. To do so, we must concurrently follow up alternative paths of execution (each based on a different assumption). Once the correct path of execution is known, all others can be aborted.

Executing alternative paths of a transaction concurrently increases demand on hardware resources, in particular, CPUs. However, as we pointed out in our discussion of data contention, in a parallel database system data contention can lead to low CPU utilisation, and it seems appropriate to use this idle CPU time to reduce the problem caused by sharing data. The idea of "sacrificing" hardware resources to improve concurrency in a database system is not entirely new: multiversion concurrency control algorithms [76] use additional memory and disk space — to store multiple versions of the same data item — to improve the level of concurrency.

We will use the following three transactions, T_1, T_2 and T_3, to illustrate the basic idea of branching transactions ($r[x]$ denotes a read operation on data item x; $w[x]$ denotes a write operation on data item x.).

T_1: $r[z]$, $r[x]$, $r[y]$, $r[t]$, $w[t]$, $r[m]$, $r[n]$, $w[n]$
T_2: $w[x]$, $r[z]$, $r[u]$, $w[u]$
T_3: $w[y]$, $r[l]$, $r[k]$, $w[k]$, $r[u]$, $w[u]$, $r[p]$

If these transactions were executed under a two-phase locking algorithm, the schedule in Table 3–1 would be a possible interleaving of their execution [2] ($r_i[x_j]$ denotes T_i reading the value of data item x written by T_j; $w_i[x_i]$ denotes T_i updating x; c_i denotes the Commit operation of T_i. The values of data items prior to the execution of this schedule are indicated by subscript 0.). At step (2), when T_1 tries to read data item x, it is blocked by T_2's lock on x; T_2 has written to x at step (1), and must therefore hold an exclusive lock on it. T_1 remains blocked until T_2 releases its lock on x. Similarly, T_1 gets blocked again at step (7), because of T_3's lock on y.

The scheduler blocks T_1 at step (2), since it cannot decide whether T_1 should read the value written to x by T_2, or the value x had prior to step (1). In case T_2 aborts, or commits after T_1, T_1 should read x_0, otherwise it should read x_2. Since T_2's fate is not known at the time of conflict, the scheduler delays its decision — blocks T_1 — until it has sufficient information to decide. In case T_2 commits before T_1, blocking of T_1, and the delay of its response time that follows from it, is unnecessary.

Table 3–2 shows a schedule in which T_1 is executed as a branching transaction. This time when T_1 tries to read data item x, it branches into two components: $T_{1,2}$ and $T_{1,3}$, the first proceeds using the original value of x, the second reads x_2. At step (3), further branching is necessary since y has been updated by T_3 at step (2). At step (6), it has become clear — since T_2 just committed — that the correct decision at step (2) was to read x_2. Therefore, it is not necessary to pursue further those components that were started under the assumption that x_0 should be read, and $T_{1,4}$ and $T_{1,5}$ abort.

When a particular path of a branching transaction has executed all operations, it is not allowed to commit until it is known whether all assumptions made by it

[2]The notion of a *step* in this table does not mean that actions at the same step have to be executed exactly at the same time, but an action at step n is executed before an action at step $n+1$.

Step	T_1	T_2	T_3
1	$r_1[z_0]$	$w_2[x_2]$	
2	blocked	$r_2[z_0]$	$w_3[y_3]$
3	blocked	$r_2[u_0]$	$r_3[l_0]$
4	blocked	$w_2[u_2]$	$r_3[k_0]$
5	blocked	c_2	$w_3[k_3]$
6	$r_1[x_2]$		$r_3[u_2]$
7	blocked		$w_3[u_3]$
8	blocked		$r_3[p_0]$
9	blocked		c_3
10	$r_1[y_3]$		
11	$r_1[t_0]$		
12	$w_1[t_1]$		
13	$r_1[m_0]$		
14	$r_1[n_0]$		
15	$w_1[n_1]$		
16	c_1		

Table 3–1: Schedule under Two-phase Locking (No Branching)

are fulfilled: $T_{1,6}$ and $T_{1,7}$ are blocked at step (9) since they cannot commit until after T_3 committed (or aborted). Since T_3 indeed commits, $T_{1,6}$ aborts and $T_{1,7}$ commits at step (10).

3.3.1 Transaction Graphs and Components:

To be able to refer to the various parts of a branching transaction we use the term *(branching) transaction component* (BTC). A branching transaction originally starts as a single transaction component. In case the transaction needs to branch, the original component creates two (or more) new components. If necessary, these new components can then themselves branch again, and so on.

We represent the branching hierarchy of a branching transaction T_i by a *trans-*

Step	**Branching Transaction T_1**				T_2	T_3
-	$T_{1,1}$:					
1	$r_{1,1}[z_0]$				$w_2[x_2]$	
-	$T_{1,2}$:		$T_{1,3}$:			
2	$r_{1,2}[x_0]$		$r_{1,3}[x_2]$		$r_2[z_0]$	$w_3[y_3]$
-	$T_{1,4}$:	$T_{1,5}$:	$T_{1,6}$:	$T_{1,7}$:		
3	$r_{1,4}[y_0]$	$r_{1,5}[y_3]$	$r_{1,6}[y_0]$	$r_{1,7}[y_3]$	$r_2[u_0]$	$r_3[l_0]$
4	$r_{1,4}[t_0]$	$r_{1,5}[t_0]$	$r_{1,6}[t_0]$	$r_{1,7}[t_0]$	$w_2[u_2]$	$r_3[k_0]$
5	$w_{1,4}[t_{1,4}]$	$w_{1,5}[t_{1,5}]$	$w_{1,6}[t_{1,6}]$	$w_{1,7}[t_{1,7}]$	c_2	$w_3[k_3]$
6	$a_{1,4}$	$a_{1,5}$	$r_{1,6}[m_0]$	$r_{1,7}[m_0]$		$r_3[u_2]$
7			$r_{1,6}[n_0]$	$r_{1,7}[n_0]$		$w_3[u_3]$
8			$w_{1,6}[n_{1,6}]$	$w_{1,7}[n_{1,7}]$		$r_3[p_0]$
9			blocked	blocked		c_3
10			$a_{1,6}$	$c_{1,7}$		

Table 3–2: Schedule for Branching Transaction

action graph (a rooted, directed tree), with T_i's BTCs as nodes ($T_{i,1}$ as root), and an edge $T_{i,j} \rightarrow T_{i,k}$, if $T_{i,j}$ created $T_{i,k}$. The transaction graph for T_1 in our example is shown in Figure 3-1. A BTC $T_{i,j}$ is a *descendant* of BTC $T_{i,k}$, if there exists a path from $T_{i,j}$ to $T_{i,k}$ in T_i's transaction graph.

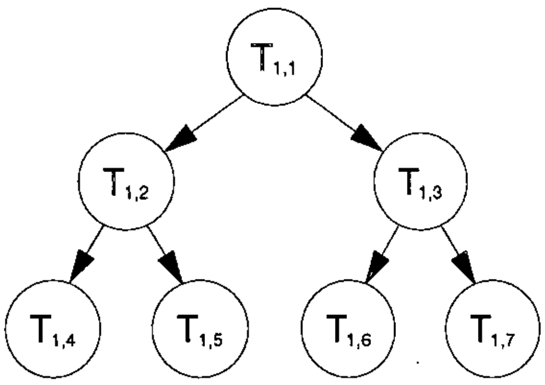


Figure 3–1: Transaction Graph

Figure 3–2 shows the state transition diagram of BTCs. After a component has been created, it is *ACTIVE*, i.e. it is executing the operations of the corresponding transaction. (We include the temporary blocking of a component in this state.) If the component is involved in a conflict and branching is necessary, it creates two or more descendant components and enters the *BRANCHED* state. In this state no more operations are executed by the component. If all descendants of a component eventually abort, then it must be aborted as well. An active transaction component can be aborted either by the system or by an explicit abort command issued by the component itself. If a component is active and has executed the last operation of its transaction, it needs to be verified: the assumptions under which it was running must be confirmed to be true. Once all components along one path in the transaction graph have been verified, that path (all of its components) can be committed. A branching transaction is only committed if all components along (exactly) one path are committed, and if a component is committed, it must be part of a committed path. The problem of ensuring this "all-or-nothing" commit condition is similar to the 2-phase commit protocol for distributed transactions. We will discuss this in more detail in Chapter 4.

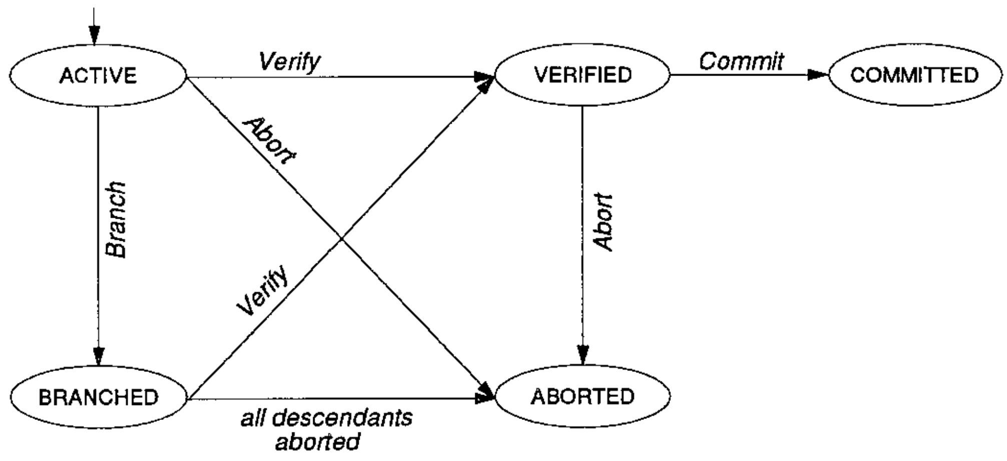


Figure 3–2: Transaction Component State Transition Diagram

In Figure 3–3 we show the dynamic creation and abortion of transaction components for our example branching transaction of Table 3–2. Originally, a single component ($T_{1,1}$) is created. Due to a conflict on item x, two new components are created by $T_{1,1}$ and the original component enters the branched state. The new components ($T_{1,2}$ and $T_{1,3}$) execute in parallel. Both have to branch as well because of a conflict on item y. We now have 4 new active components and three branched ones (after step 3). Once T_2 committed, it was clear that $T_{1,1}$ and all its descendants were executed under a wrong assumption — that T_2 will abort — and hence they are all aborted. After T_3 commits, it is known that the path containing $T_{1,7}$ has been executed under the right assumptions — that T_2 and T_3 commit — and all of its components can be verified, and $T_{1,6}$ must be aborted. Subsequently, the verified path is committed.

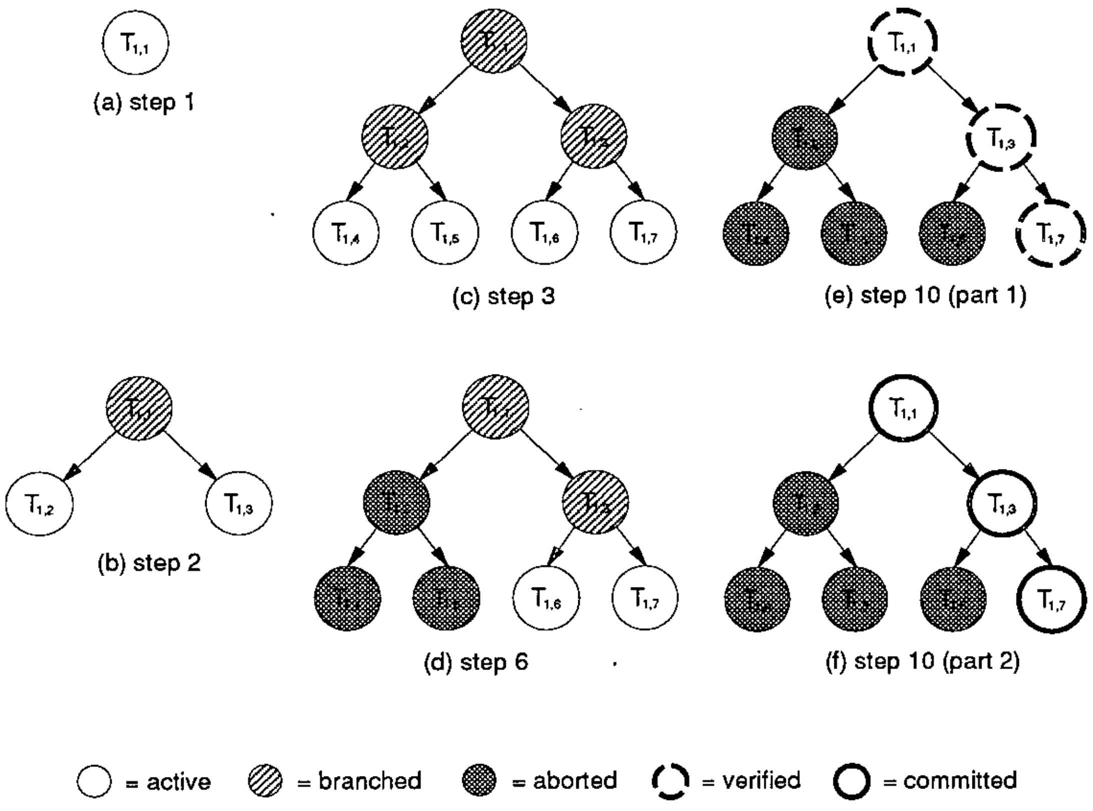


Figure 3–3: Transaction Component State Changes within a Branching Transaction

It is a key property of branching transactions that only one path in every transaction graph can commit, and all components not part of this path are aborted. Any updates on the database performed by aborted transaction components are rolled-back.

Any two transaction components of the same transaction tree can be executed in parallel, unless there exists a path in the tree between them. Components from different paths execute in isolation: they do not read any updates made by the other and are allowed to update the same data items independently.

Although we have not shown this in our example, it is possible that although a component has been verified, it must later be aborted. For example, it may be the case that, even though a component can be verified, one of its ancestor components cannot.

Since we need to support concurrent access to more than just the most recently committed version of a data item, multi-version concurrency control algorithms are required for branching transactions. We will discuss these algorithms in detail in Chapter 4.

3.3.2 Performance Considerations

As pointed out before, "wrong decisions" cause performance problems through blocking and rollbacks (with subsequent restarts) of transactions. In this section, we present an intuitive argument why branching transactions help to reduce the data contention problem in parallel database systems. A simulation based performance study is presented in Chapter 7.

Blocking: Under two-phase locking, a read operation is blocked if a transaction cannot obtain the appropriate read lock. This is only the case if there exists a write lock on the same item. Such a write lock indicates that there is currently another active transaction which created a new, so far uncommitted, version of the same item. As we explained earlier, under BT the reader transaction would branch and continue to execute alternative paths without any delay due to blocking. If a write operation is blocked under two-phase locking, some other active transaction has either read or written the same data item. The BT multi-version concurrency control algorithm described in Chapter 4 never delays a write operation, but simply creates a new version of the item. Hence, write operations don't have to be blocked either.

Restarts: Earlier we saw an example (of an optimistic scheduler) where it would have been better to abort a transaction at the time of conflict, rather than letting it continue and fail during validation. Clearly, running the entire transaction, aborting it, and then running it again from the beginning has a negative impact on transaction response time. Using branching transactions, the goal is to have at least one path of execution to complete without ever having been aborted and restarted, even though other paths need to be aborted. If we label the two[3] runs — the original, aborted one and the restarted one — of a restarted transaction under an optimistic scheduler as "Run1" and "Run2", then we can think of a branching transaction as executing these two runs in parallel, while the optimistic scheduler executes them in sequence. This should lead to shortening of transaction response times when using branching transactions.

Undoubtedly, branching transactions require more CPU resources than non-branching systems, since alternative paths must be computed in parallel; though this extra overhead may be limited when one considers the additional computational resources required for rolling back and restarting of (non-branching) transactions. As we pointed out before, in a multi-processor environment suffering from data contention, CPUs are largely underutilised, and hence, increased CPU needs will in general not pose a problem. Depending on the level of data contention, the transaction workload and the actual number of CPUs, it is, however, possible that unlimited branching might lead to an overload of the system. A mechanism to prevent such "BT thrashing" is discussed next.

3.3.3 Branching Control

To prevent "BT thrashing" we need a policy to regulate the branching of transactions, a policy which will only allow transactions to branch if sufficient system resources are available. A function *branch_control* must be defined which determines whether branching should be permitted. Since BT requires primarily additional

[3]For the sake of argument we assume here that the transaction successfully completed after one restart. The same comparison, however, would also apply to more than one restart. In fact, the case for branching transactions would then be even more compelling.

CPU time, *branch_control* may allow branching only if the average CPU utilisation is below a certain threshold. To prevent a single branching transaction from monopolising too many resources we can also apply a static control policy: each branching transaction is only allowed a maximum number of components. Once this limit has been reached, no further branching for this transaction is allowed. Combining the dynamic and static control we can define *branch_control* as follows ($NumComp(bt\_id)$ is the total of number of components created for branching transaction T_{bt_id} ; $AvgCPUUtil$ describes the current average CPU utilisation in the system):

$$branch\_control(bt\_id) = \begin{cases} true & \text{if } NumComp(bt\_id) \leq \text{MAX\_COMPS and} \\ & \quad AvgCPUUtil \leq \text{MAX\_CPU\_UTL} \\ false & \text{otherwise} \end{cases}$$

In case branching of a transaction component is denied by *branch_control*, the corresponding read operation follows a non-branching concurrency control policy. A concurrency control algorithm which can deal with the integration of branching and non-branching policies is discussed in Chapter 4.

Using a branching control policy, branching of transactions can be controlled dynamically. It is adapted to the current system workload in the sense that branching is only allowed as long as enough resources are available.

3.4 Related Work

Transaction models and concurrency control algorithms are very active research topics. A number of extensions to the flat transaction model have been proposed in the literature [32]. Furthermore, work has been done in exploiting the semantics of database operations beyond the scope of read and write operations. To underline the differences between branching transactions and other work in the same area, we discuss various transaction models and approaches and compare them with our own transaction model.

3.4.1 Transaction Models

All advanced transaction models described below have extended the traditional flat model with some form of subtransaction[4] structure. As in the case of branching transactions, most use a hierarchy of subtransactions, some only allow a set of subtransactions without further subdivision. In spite of this similarity, they differ in their objectives and approaches. We only discuss those models which appear similar in their approach or their goals to branching transactions. When we refer to ACID properties, we mean the usual characteristics: atomicity, consistency, isolation and duration of transactions.

Nested Transactions

Introduction: A *nested transaction* [63,64] is a tree of subtransactions, where each subtransaction may contain other subtransactions. Nested transactions are ACID at the top level, i.e. they are isolated from each other and in case of failure they must be rolled-back entirely. If a subtransaction fails, all of its child subtransactions must be aborted as well. The parent of a failed subtransaction, however, does not have to be aborted just because one of its child subtransactions fails. In fact, it may react to such a failure by initiating a *contingency subtransaction* which implements an alternative solution to the failed one, or simply retry the failed subtransaction. Even if a subtransaction commits, it may have to be aborted later, if one of its parents has to be aborted[5].

There are two main advantages of nested transactions: 1) finer control of transaction failure, and 2) intra-transaction parallelism. Better transaction failure

[4]A subtransaction in the context of branching transaction, is what we call a "branching transaction component".

[5]This is the reason why a subtransaction alone only has ACI properties, i.e. *D*urability of the effects of a subtransaction is not guaranteed once the subtransaction commits, but only after the entire nested transaction commits.

handling is due to the ability of a subtransaction to take "evasive action" and continue operating in case a child subtransaction aborts, unlike in the traditional flat model, where the abort of any operation always leads to an abort of the entire transaction. Intra-transaction parallelism is achieved by executing subtransactions concurrently[6].

Comparison with Branching Transactions Although both models are based on a hierarchical structure of substransactions, there are a number of important differences between nested and branching transactions: (1) nested transactions support a higher level of concurrency within a transaction, while branching transactions are designed to improve concurrency between transactions; thus, nested transactions support higher intra-transaction parallelism, while branching transactions support a higher level of inter-transaction parallelism; (2) siblings within nested transactions must be synchronised, "sibling transaction components" within a branching transaction execute independently from each other; or in other words: all subtransactions under nested transactions execute within the same context, whereas different branches of a branching transaction are executed in different contexts, i.e. under different assumptions; (3) only one descendant of a branching transaction component is allowed to commit, but all subtransactions in a nested transaction can commit (and in fact should, unless the transaction explicitly deals with the failure of subtransactions); and (4) in nested transactions the nesting structure is designed by the user, whereas branching of transactions is transparent to the user. Although both models can be applied to relatively simple as well as more complex transactions, there is probably little use for nesting transactions which are comparatively simple already.

It appears that nested transactions and branching transactions are two fairly orthogonal concepts, and it may be possible to combine these models to gain

[6]It should be pointed out that not every commercial DBMS which claims to support nested transactions, necessarily implements the full functionality of nested transactions as described above.

both benefits: higher inter-transaction parallelism and higher intra-transaction parallelism. A detailed study of such a hybrid model is, however, beyond the scope of this thesis and not discussed any further here.

Distributed Transactions

Introduction: The term *distributed transactions* is usually used to refer to flat transactions executing in a distributed environment. Since data is located at different nodes within a network, different parts of a transaction may have to be executed on different hosts. The part of a transaction executing at one node is in general called a subtransaction. Unlike nested transactions, if any of its subtransactions fails, the entire distributed transaction must be aborted. The division of a distributed transaction into subtransactions depends on the data distribution in the network, and not on a functional decomposition of the transaction (as in nested transactions) or the existence of data conflicts during runtime (as in branching transactions).

Comparison with Branching Transactions: Distributed transactions were not designed to address the issue of data contention, but to deal with transaction management in a distributed database environment. Unlike all the other models discussed here, the motivation for distributed transactions was not to improve concurrency control related problems. Hence, branching transactions and distributed transactions are two fairly independent concepts, and again it appears as if a hybrid model would be feasible.

Multi-Level Transactions

Introduction: Multi-level transactions [96,97] are a variant of nested transactions where the tree of subtransactions is balanced. Different levels in the subtransaction tree correspond to different levels of abstraction of operations. All transaction trees have the same height, which reflects the number of layers in the underlying system architecture. The child nodes in the tree correspond to a sequence of lower level operations which are executed to implement the operation of the (parent) node at the level above. The key idea of multi-level transactions is to exploit *level specific semantics* to achieve a higher degree of concurrency than in traditional transaction systems. A special case of this model are *sagas* [38], where we only have a two-level system and there are no conflicts at the higher level.

Comparison with Branching Transactions: Multi-level transactions and branching transactions are designed for the same problem: performance loss due to data contention. While the basic idea in branching transactions is to follow up alternative paths of solutions in parallel, multi-level transactions try to solve the problem by exploiting the semantics of operations. Neither approach employs the tactics of the other. Due to the complexity of analysing and resolving conflicts at higher abstract levels — not merely at the read/write level — it is currently difficult to envision a combination of branching and multi-level transactions.

Split Transactions

Introduction: The basic idea of *split transactions* is to split an ongoing transaction into two serializable transactions [74]. Resources held by the original transaction are shared between the two new ones. While one of the new transactions commits, the other continues its execution. This allows the early release of results from the original transaction without compromising the ACID properties of it.

Comparison with Branching Transactions Split transactions were designed to address the issue of open-ended applications such as CAD/CAM projects, VLSI design and software development. Transactions in such environments typically run from hours to months. Although splitting such long transaction is a feasible task, it does not seem an appropriate approach to deal with data contention in OLTP type transaction workloads — which, of course, was never the target of split transactions in the first place. Hence, except that both models, split transactions and branching transactions, were designed to address concurrency control problems, they do so for rather different workloads.

Flex Transactions

Introduction: Under the *flex transaction* model [31], a transaction consists of a set of tasks. For each of these tasks the user can specify a set of functionally equivalent subtransactions. As long as one of the corresponding subtransactions successfully completes, a task is accomplished. A flex transaction succeeds if all of its tasks are successfully completed. *Failure* and *success dependencies* between subtransactions of a flex transaction can be specified by the user. These dependencies are used to define the execution order, such as parallel or sequential, of subtransactions. The flex transaction model has been implemented in VPL [53], a superset of Prolog.

Comparison with Branching Transactions Flex transactions were designed to address the issue of transaction management in multidatabase systems. Hence, this model is looking at problems of how to integrate various independent transaction management systems, rather than data contention. There is, however, an interesting similarity to branching transactions. Given the possibility of parallelism and dependencies between flex subtransactions, various possible solutions for a task can be followed up in parallel and once a subtransaction accomplishes a task, all other alternative solutions, i.e. subtransactions for the same task, can be aborted. The difference is that alternative subtransactions are designed by the user before execution time and not driven by data conflicts (as is the case in branching transactions) during runtime. Also, alternative paths of execution in a branching transaction are transparent to the user, but alternative solutions to a task of a flex transaction must be specified by the user. Finally, different subtransactions for the same task are only *semantically equivalent*, but may run completely different transaction code. Sibling transaction components in a branching transaction run the exact same code, but with a different data set.

3.4.2 Semantics Based Concurrency Control

In traditional concurrency control, data objects are passive entities which are either read from or written to. Consequently, synchronisation of transactions was performed with respect to read and write operations. In his Ph.D. thesis [78], Schwarz redefined transactions as a sequence of typed operations on instances of shared abstract types. Object-oriented databases follow this idea: a data object is an instance of an abstract data type (referred to as a "class" in object-oriented terminology) — OODBs are more than just abstract data type systems; they (in addition to persistence) incorporate such concepts as inheritance and polymorphism. With this new view of transactions it was possible to define synchronisation protocols which could take into account the operational semantics of data objects. Such protocols allow a higher level of concurrency than traditional protocols, which only consider read and write operations.

Skarra et al. [80] divide semantics based synchronisation protocols into *intertype synchronisation* and *local concurrency control.* Intertype synchronisation protocols consist of local concurrency control components, i.e. protocols that implement type-specific synchronisation policies within data objects, and a global protocol through which different types cooperate to ensure global consistency. The global protocol may guarantee serializability, or it may use a weaker correctness criterion. Local Concurrency Control was discussed in [36,45,94,95]. What is different about local concurrency control, is its ability to achieve global atomicity by enforcing properties that are local to individual data objects. As long as each object guarantees a local atomicity property, it follows that every execution of transactions in the system is atomic. Since no single global algorithm is required, each object can use a different concurrency control algorithm. An example of such a local atomicity property is *dynamic atomicity.* A brief introduction to dynamic atomicity and other concurrency control concepts based on data object semantics can be found in [80].

Considering the semantics of operations to reduce conflicts between transactions is a useful approach to limit performance problems caused by concurrency control. The disadvantage of it, however, is the added complexity of determining conflict relationships between operations. Not only is it necessary to define these relationships for any two existing operations on the same data item (object), but for any new operation added to an object the relationships with existing operations must be determined. Branching transactions do not consider extra semantic information; rather, conflicts are dealt with at the read/write level (as in traditional concurrency control protocols).

3.4.3 Ordered Shared Locks

Introduction: Ordered Shared Locks [2,4] are the basis for a family of new locking protocols. While traditional locking algorithms only distinguish between two kinds of relationships between locks: *shared* and *non-shared*, a new mode has been introduced here: *ordered shared.* Agrawal et al. [2,4] argue that traditional locking algorithms only take into consideration whether operations conflict with each other or not, but they do not take into account the order in which two conflicting operations are executed. For serializability theory, however, this order is essential. Using ordered shared locks, their concurrency control algorithms allow (serializable) schedules which would not be allowed using traditional locking algorithms. In fact, they state that their locking protocol is the first one that can recognise the entire class of conflict-preserving serializable schedules.

As with traditional locking, a *shared* relationship between lock types implies that multiple locks of these types can exist for the same data item simultaneously, and a *non-shared* relationship between lock types implies that no such multiple locks are allowed. Read locks can usually be shared with other read locks, but write locks are exclusive. This reflects the fact that read operations do not conflict with each other, but write locks conflict with read locks and other write locks. *Ordered shared* locks allow conflicting operations to proceed without blocking, but only with certain constraints. These constraints are expressed in the *Ordered Sharing Acquisition Rule* and the *Lock Relinquishing Rule.* The first rule says that for any two ordered shared locks, the corresponding operations must be executed in the same relative order in which the locks were acquired. For example, if one transaction acquires a read lock before a second transaction acquires a write lock for the same item (and read and write locks have an ordered-shared relationship), then the read operation of the first transaction must be executed before the write operation of the second. The Lock Relinquishing Rule says that a transaction must not release any locks as long as it is waiting for some other transactions. A transaction T_j is waiting for another transaction, say T_i, if T_j acquired a lock with an ordered shared relationship with respect to a lock held by T_i, and T_i has not released *any* of its locks.

Comparison with Branching Transactions: Ordered shared locks were developed to address the same issue as branching transactions: data contention through locking protocols. By considering the order of operations in their protocol, Agrawal et al. have successfully addressed some of the issues of what we call the "wrong decision" problem, i.e. their protocols recognise more correct schedules than normal two-phase locking. They are, however, left with the problem that some of the decisions taken by their scheduler may later turn out to be incorrect. This problem manifests itself in the form of deadlocks when transactions get blocked due to the Lock Relinquishing Rule. Some transactions need to be aborted and restarted. The restarts have the usual negative performance impact; the initial run and subsequent re-runs of a transaction are executed in sequence. As explained earlier, branching transactions aim to reduce this problem by effectively running the initial run and re-runs in parallel.

Both mechanisms, ordered shared locks and branching transactions, are based on the same observation: due to data contention, concurrency control algorithms using traditional locking protocols are not well suited for modern database systems with their ever increasing hardware resources. In both cases, performance improvements are achieved by making better use of hardware resources. Performance studies by Agrawal et al. [4] have shown that their algorithms perform better if the availability of hardware resources is high. The same is true for branching transactions (see Chapter 4).

3.4.4 Speculative Computing

Introduction: *Speculative computing* is an approach in parallel computing where results are computed before it is certain that they will be required [18, 69]. It is based on the idea that one trades additional, possibly unnecessary, computation for potentially faster execution. For example, in a functional language if it takes some time to compute the truth value of a condition of an if-statement, one could in parallel compute the values of the "then-expression" as well as the "else-expression". Once the condition has been evaluated, the value of the required "then" (or "else") part will already have been computed (at least partially). Since computation resources are generally limited, two policies are applied: 1) once it is known that a particular computation is unnecessary, it is stopped as soon as possible and its resources reclaimed, and 2) the allocation of resources favours computations which are more promising than others.

Comparison with Branching Transactions: There is clearly a similarity between branching transactions and speculative computing. In fact, one could refer to branching transactions as *speculative transactions* instead. Branching transactions apply the concept of speculation in the context of transactions. The speculation here is whether some other transactions, which created new versions of data, will eventually commit or abort. Although the approaches are similar, the goals are very different: speculative computation tries to improve parallelism within a program, whereas branching transactions try to improve parallelism between transactions.

In spite of the particular meaning of *speculative computing* in the context of parallelising functional programs and the idea of speculation in branching transactions, *speculative computing* can be considered as a general approach to computing. Other examples of the general concept of speculative computing can be found in pipeline execution of instructions within processors and speculative concurrency control, the latter of which we describe next.

3.4.5 Speculative Concurrency Control

Introduction: *Speculative Concurrency Control (SCC)* is the approach most similar to branching transactions. It was developed at the Computer Science Department at Boston University and first published [11,12,13] at about the same time as branching transactions [16]. SCC is based on the idea of running so-called *shadow transactions* for a transaction; each shadow speculates on a different possible outcome of conflicts between transactions. The original transaction is executed on the assumption that it will commit before the commit of any of the transactions with which it has a conflict. If conflicts materialise which do not allow the original transaction to commit, it is aborted and one of the shadow transactions takes over its role. Multiple shadow transactions can execute in parallel. Only one version (original or shadow) will eventually commit, all others abort.

Comparison with Branching Transactions: Speculative concurrency control and branching transactions are rooted in the same basic idea: the use of redundant computation of transaction versions[7] in order to anticipate possible outcomes of conflicts between transactions. Both mechanisms assume that sufficient resources, primarily CPUs, are available to achieve performance improvements in spite of higher resource requirements. The developers of SCC see this to be the case in real-time database systems and have concentrated most of their work in that area. The development of branching transactions is primarily concerned with the exploitation of parallel computers for database systems, although we also recognise the potential of this approach in the context of real-time transaction scheduling; a real-time concurrency control algorithm based on branching transactions is proposed in Chapter 8.

In spite of their obvious similarities, there are a number of significant differences between SCC and BT. SCC is described in the context of the traditional flat

[7]In SCC these versions are called shadow transactions, in BT they are called transaction branches.

transaction model; forking of a transaction leads to a new flat (shadow) transaction. In BT, we have conceptualised this forking into a new transaction model, which incorporates the notion of alternative executions and explicitly supports a branch (or fork) operation for transactions. This is very useful when one has to reason about such transactions.

A key difference between these two approaches is the way that transaction access to the database is synchronised. SCC adopts an approach where for each transaction shadow a so-called "speculated order of serialization" (SOS) is maintained. An SOS specifies the orderings of transactions that must be observed by a shadow. SOSs are similar to serialization graphs. In the past, however, serialization graph based approaches for concurrency control have not been adopted in commercial DBMSs due to their high overheads. Whether or not this might also prove problematic for SCC is an open question. Most commercial DBMS today use a locking approach to concurrency control. Synchronisation of branching transactions is also based on locking (see Chapter 4). A key advantage of our approach is its ability to dynamically switch — at run-time — between branching and non-branching modes, i.e. as long as there are enough resources available we employ our branching strategy, but when resources are scarce the system changes to "normal" two-phase locking, which is known to perform well under resource contention.

For branching transactions we have proposed a possible two-layer approach for implementation, where the lower layer represents a basic two-phase locking policy and the upper layer provides the full branching transaction locking algorithm. This layered approach supports an easier migration path for existing DBMS that wish to provide branching transaction facilities. Since SCC is not based on locking, moving an existing DBMS to SCC would be more complicated.

Since branching transactions are considered in the context of parallel databases, this dissertation addresses a number of issues, e.g. cache coherence and parallel DBMS architectures, which are not looked at in the context of SCC (neither was there any need to do so, since the developers of SCC were focussing their work on real-time DBMS and not parallel DBMS).

Chapter 4

Concurrency Control and Recovery

Existing concurrency control algorithms cannot directly be applied to our new transaction model, and hence, a new concurrency control mechanism was developed for branching transactions. In this chapter, we explain this new mechanism in detail. We also describe the problems of deadlock detection and resolution, and logging and recovery in the context of branching transactions. Since the new concurrency control algorithm is based on multi-version two-phase locking, we begin our discussion with a brief review of single-version[1] and multi-version two phase locking.

4.1 Two-Phase Locking (Single-Version)

Under 2PL, a transaction has to acquire a Read lock before it can read a data item in the database, and it must acquire a Write lock before it can update an item. A transaction can acquire a Read lock if no other transaction is holding a Write lock on the same item; multiple Read locks are allowed. A transaction can acquire a

[1]Although an introduction to two-phase locking (2PL) was given in Section 2.3.1, we briefly repeat it here to keep this chapter as self contained as possible.

Write lock on an item if no other locks (Read or Write) exist at the time for that item. This kind of lock compatibility information is usually given in so-called *lock compatibility matrices.* The lock compatibility matrix for 2PL is shown in Table 4–1.

2PL	*Lock Held*	
Lock Requested	Read	Write
Read	yes	no
Write	no	no

Table 4–1: Lock Compatibility Matrix for 2PL

The two-phase property of 2PL says that the life time of a transaction can be divided into two distinct phases. During the first phase a transaction is allowed to acquire locks; during the second phase a transaction can release locks, but it cannot acquire new ones. In other words, as soon as a transaction has released one of its locks, it is not allowed to acquire any new ones.

A particular variant of 2PL is called *strict 2PL*: all locks of a transaction are released *together* after the transaction commits or aborts. Strict 2PL prevents transactions from reading "uncommitted data", i.e. data that was written by a transaction which has not yet committed.

So far, we have assumed that at any point of time there is only one version of every data item available in the database and that a transaction can only read the most recently updated version of an item. To allow a higher level of concurrency, some mechanisms keep older versions of data items in addition to the most recent one. They are referred to as *multi-version concurrency control mechanisms.* A multi-version two-phase locking algorithm is described in the following section.

4.2 Multi-Version Two-Phase Locking

Under *multi-version two-phase locking* (MV-2PL), when a transaction wants to update a data item, it does not overwrite the previous value of it, but creates a new version of the same item. There are two key advantages of MV-2PL: 1) two transactions can write the same data item concurrently (although some synchronisation must take place at commit time; more detail below), and 2) a transaction can read one version of an item, while another transaction creates another version at the same time.

Under MV-2PL, for any data item in the database there exists exactly one committed version, and zero or more uncommitted versions[2]. Once a transaction which created a new version of an item commits, the previously committed version of the item is discarded. It follows that a transaction can in principle read either the most recently committed version or one of the currently existing, uncommitted versions of an item, though some versions of MV-2PL restrict transactions to reading committed versions of data only.

In general MV-2PL allows more concurrency in schedules than would be possible under single-version 2PL, but it must impose some ordering on operations to prevent non-serializable concurrent executions of transactions. In addition to Read locks and Write locks, MV-2PL makes use of a third kind of lock: the *Certify Lock*. Certify locks are used at commit time to synchronise transactions' Read and Write operations. We discuss a locking protocol based on these lock types next. Although it is not a necessary condition for MV-2PL, for now we assume that transactions only read committed versions to avoid cascading rollbacks.

If a transaction wants to read an item, it first has to acquire a Read lock on the last certified version. A Read lock is granted if there are currently no Certify

[2]We consider a version to be *committed*, if the transaction which created it has committed, otherwise the version is said to be *uncommitted*.

locks or pending Certify lock requests on that version. Read locks are compatible with Write locks on different versions of the same item. If another transaction holds a Certify lock or has a pending lock request on the last committed version, the reader must wait until that lock has been released.

If a transaction wants to write an item, it simply creates a new version. The transaction is automatically granted a Write lock for that version. Hence, Write operations are not delayed. In case it is necessary to limit the total number of uncommitted versions in the system, a more restrictive protocol can be used. We will return to this issue later in this chapter when we discuss a hybrid branching transaction MV-2PL algorithm.

At commit time, a transaction has to "certify" all its updates, i.e. for every data item it updated it must acquire a Certify lock on (the committed version of) that item; it must upgrade its Write lock to a Certify lock on the corresponding committed version of a data item.

A Write lock can be upgraded if there are currently no Read locks on the committed version of that item. If a Read lock exists, the transaction requesting the Certify lock is blocked until all Read locks on that item are released. A transaction can be committed, once all of its Write locks have been certified. The compatibility of locks under MV-2PL is summarised in Table 4–2.

MV-2PL	*Lock Held*		
Lock Requested	Read	Write	Certify
Read	yes	yes	no
Write	yes	yes	yes
Certify	no	yes	no

Table 4–2: Lock Compatibility Matrix for MV-2PL

Since Write locks never conflict with any other locks, one might question their purpose. In fact, for the particular variant of MV-2PL discussed here, they are only used to determine for which items a transaction has to acquire Certify locks at commit time. Write locks, however, can be used to implement a more restrictive MV-2PL protocol which allows at most two versions of an item at any time [10].

So far we have assumed that a transaction will always read the committed version of a data item. We will now see what happens if we relax this condition, i.e. if we allow transactions to read uncommitted data.

If a transaction which created a number of new data item versions during its execution has to be aborted, then all other transactions which read these versions must also be aborted. In turn, this may cause yet more transactions to be aborted for the same reason (cascading aborts). Since a committed transaction cannot subsequently be aborted, the commit of a transaction must be delayed until all data versions it read have become committed. Hence, if we allow transactions to read uncommitted data, the transaction must perform a two-step certification process: 1) wait until all read data is committed, and 2) certify all write locks, before it is allowed to commit.

4.3 Branching Transaction Multi-Version Two-Phase Locking

The concurrency control algorithm used for branching transactions is an extension of the MV-2PL algorithm described above. Before we discuss *branching transaction multi-version two-phase locking* (BT-MV-2PL) in more detail, it is necessary to take a brief look at some of the aspects of a branching transaction system architecture. Figure 4–1 shows part of such an architecture[3].

A transaction is submitted to the *transaction manager* (TM) for execution. The TM creates the initial branching transaction component (BTC) for this transaction. The BTC executes the transaction code, and hence, it is the BTC which makes lock requests to the concurrency control manager (CCM). The CCM replies to a BTC directly if a lock request is granted, but in case a roll-back decision is made, the CCM informs the TM instead. The TM in turn communicates the

[3]Architectural issues are discussed in great detail in Chapter 5.

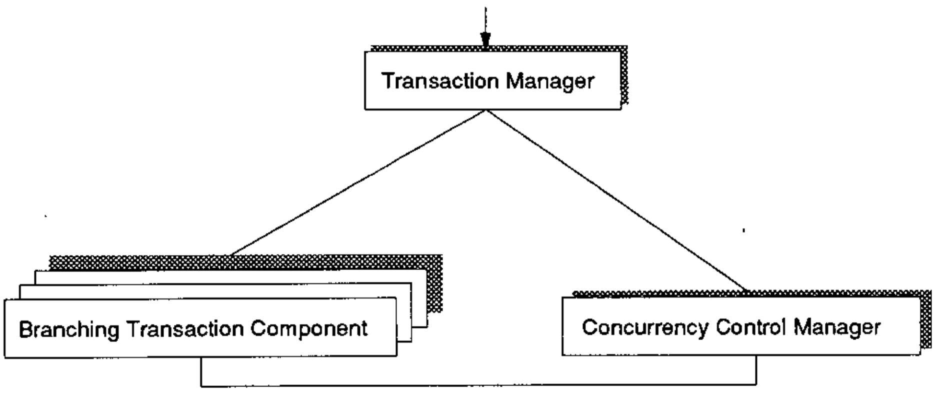


Figure 4–1: Simple Module Architecture

roll-back decision to the BTC which requested the lock, and all other BTCs which need to be aborted as a consequence.

If a BTC wants to update a data item, it follows the same protocol as under MV-2PL: it simply creates a new version, for which it is automatically granted a Write lock. As before, these Write locks must be certified at commit time.

If a BTC wants to read a data item, it requests a Read lock for it. Depending on whether there exist any uncommitted versions for that item, two possible scenarios follow:

No uncommitted versions exist: Since there are no uncommitted versions, there exists no uncommitted transaction which has updated this data item. It follows that there are no Certify locks or lock requests on that item. The Read lock of the requesting BTC is, therefore, granted and a Read lock is set on the committed version of the requested data item.

Uncommitted versions exist: At least two versions of the data item exist (one committed and at least one uncommitted one), and the requesting transaction needs to branch. The CCM contacts the TM to get the ID's for the new BTC's and to inform the TM about the branching decision. The CCM assigns one data item version to each new BTC and records Read locks accordingly. The TM creates the new BTCs which continue by reading the data version they were assigned; they are automatically granted Read locks for the respective data item versions as part of the branching process.

The second case is somewhat problematic. Read locks on the committed version of a data item are not compatible with Certify locks. Since, at the time of branching, a BTC may be granted such a Read lock irrespective of any existing Certify lock, the locking rules of MV-2PL may become violated[4]. To prevent possible database inconsistencies (violation of serializability), if a BTC holds a Certify lock on (the committed version of) an item and it commits, then all BTCs which hold Read locks on that committed version must be aborted. We illustrate this point in Example 2.

Example 2 *Let us assume the case of two very simple transactions:*

$$T_1\colon w_1[x]$$
$$T_2\colon r_2[x]$$

The following sequence of events illustrates the case of a Read/Certify lock conflict on item x:

1. *Both transactions are submitted to the transaction manager for execution. $BTC_{1,1}$ and $BTC_{2,1}$ are created and start execution.*

2. *$BTC_{1,1}$ wants to write x and thus requests a Write lock on x. The request is granted and $BTC_{1,1}$ creates a new version of x: $x_{1,1}$. The lock table for x after this step is shown in Figure 4–2, part a).*

3. *$BTC_{1,1}$ requests to be certified; it requests a Certify lock on the committed version of x. Since there are no other locks, the Certify lock is granted. The lock table for x after this step is shown in Figure 4–2, part b).*

4. *Now $BTC_{2,1}$ requests a Read lock on x. Since there exist two versions, i.e. the original one and the one newly created by $BTC_{1,1}$, branching takes place.*

[4]The lock compatibility rules do not change from MV-2PL to BT-MV-2PL; compare Tables 4–2 and 4–3.

$BTC_{2,1}$ branches into $BTC_{2,2}$ and $BTC_{2,3}$. $BTC_{2,2}$ is assigned the original version of x ($x_{0,0}$) and a Read lock is set accordingly. Similarly, for $BTC_{2,3}$. The lock table for x after this step is shown in Figure 4–2, part c).

At this point we have violated the MV-2PL rule that Read locks and Certify locks are not compatible, and only one, $BTC_{1,1}$ or $BTC_{2,2}$ must be allowed to commit.

5. *If $BTC_{1,1}$ can be certified — in our simple example there is no reason why it shouldn't, since the Write on x is its only operation and it has been certified already — it eventually commits. In this case $BTC_{2,2}$ aborts. Since $BTC_{2,3}$ had a Read lock on $x_{1,1}$, it has now a lock on the committed version of x. The lock table for x after this step is shown in Figure 4–2, part d).*

6. *(alternative to the previous step) If for some reason $BTC_{1,1}$ was aborted, $BTC_{2,3}$ needs to be aborted as well, but $BTC_{2,2}$ keeps its lock on the committed version of x. The lock table for x in this case is shown in Figure 4–2, part e).*

Once a BTC has reached the end of its transaction code, it informs the TM that certification should be performed. Before a transaction can commit, all BTCs along one root-to-leaf path of the corresponding branching transaction tree must be certified first. The certification of a BTC follows the same two steps as for transactions under MV-2PL:

Read Certification: the BTC must wait until all versions it read have become committed.

Write Certification: all Write locks held by the BTC must be certified, i.e. corresponding Certify locks must be obtained.

In Example 2, in Figure 4–2, Part c), $BTC_{2,3}$ has a Read lock on an uncommitted version of x, hence, the corresponding Read operation is not certified. In Part d), however, this Read lock has become a lock on the now committed version of

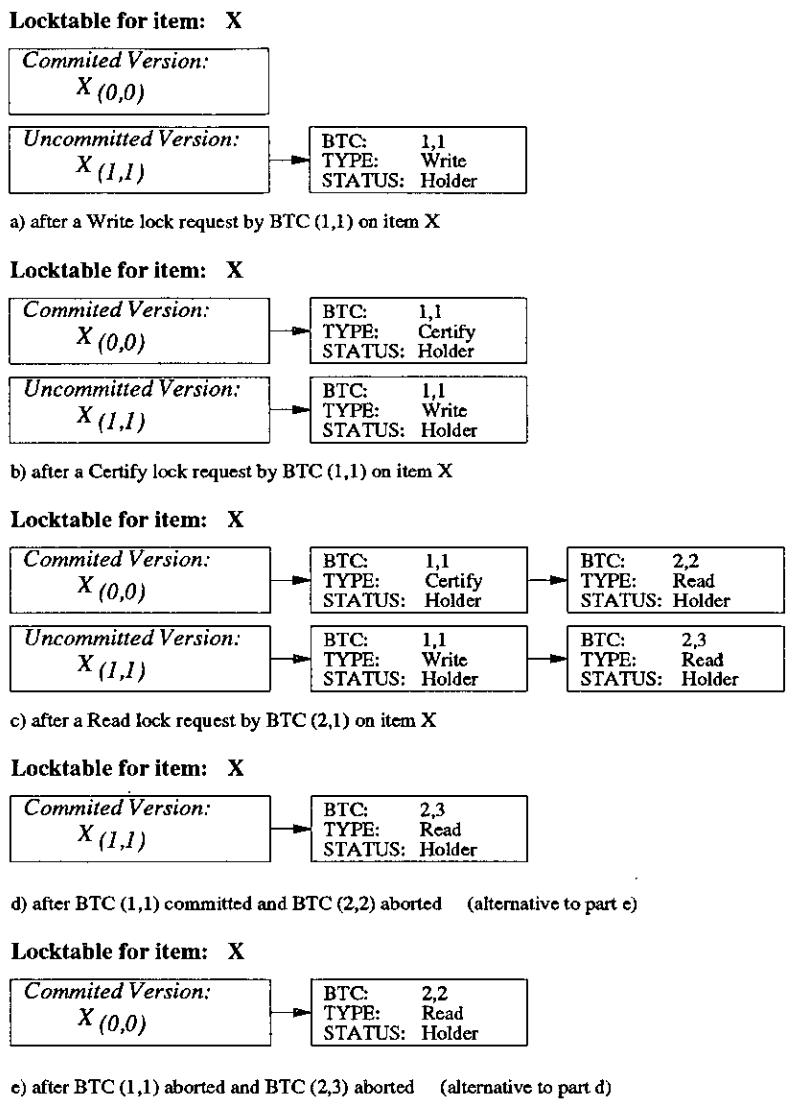

Locktable for item: X
Commited Version: X (0,0)
Uncommitted Version: X (1,1)
BTC: 1,1 TYPE: Write STATUS: Holder
a) after a Write lock request by BTC (1,1) on item X
Locktable for item: X
Commited Version: X (0,0)
BTC: 1,1 TYPE: Certify STATUS: Holder
Uncommitted Version: X (1,1)
BTC: 1,1 TYPE: Write STATUS: Holder
b) after a Certify lock request by BTC (1,1) on item X
Locktable for item: X
Commited Version: X (0,0)
BTC: 1,1 TYPE: Certify STATUS: Holder
BTC: 2,2 TYPE: Read STATUS: Holder
Uncommitted Version: X (1,1)
BTC: 1,1 TYPE: Write STATUS: Holder
BTC: 2,3 TYPE: Read STATUS: Holder
c) after a Read lock request by BTC (2,1) on item X
Locktable for item: X
Commited Version: X (1,1)
BTC: 2,3 TYPE: Read STATUS: Holder
d) after BTC (1,1) committed and BTC (2,2) aborted (alternative to part e)
Locktable for item: X
Commited Version: X (0,0)
BTC: 2,2 TYPE: Read STATUS: Holder
e) after BTC (1,1) aborted and BTC (2,3) aborted (alternative to part d)

Figure 4–2: BT-MV-2PL Example

x, and hence, the Read is now certified. There is an important difference between Read and Write certification. Read operations may be certified before a BTC has entered its actual certification phase. The certification of Read operations is entirely outside the control of a BTC. It merely depends on whether or not the transaction which has written the corresponding version commits. Write certification, however, takes place during the certification phase and the BTC (which is trying to be certified) has to actually request Certify locks for its updates. The lock compatibility matrix for BT-MV-2PL is shown in Table 4–3.

BT-MV-2PL	*Lock Held*		
Lock Requested	Read	Write	Certify
Read	yes	yes	no
Write	yes	yes	yes
Certify	no	yes	no

Table 4–3: Lock Compatibility Matrix for BT-MV-2PL

Commitment of Branching Transactions: In Chapter 3, we pointed out that when a branching transaction commits, only the BTCs along one root-to-leaf path of the branching transaction tree can commit; all others are aborted. Once all BTCs of such a path are certified, the TM initiates a *commit protocol.* The protocol is similar to the *two-phase commit* protocol in distributed systems (a description of which was given in Chapter 2). During the first phase, all BTCs are requested to prepare for commit[5]. They reply to the TM once they have carried out all logging activities needed to guarantee their ability to commit even in the event of various system failures. After the TM has been informed by all committing BTCs that they are prepared to commit, it sends a Commit message to them. At that time, BTCs can commit and release their locks on data items. This commit protocol is required to ensure the atomicity of branching transactions; it guarantees that either all BTCs along the selected path of a transaction tree are committed or none are.

BTC Non-Interference Rule: Once a transaction has branched, it is quite likely that BTCs belonging to alternative paths of execution access the same data items, and hence, in principle, conflict with each other. Since eventually only one path commits and all others abort, their interference can be ignored: Read, Write and Certify locks of "sibling" BTCs never conflict with each other. It is, for

[5]Although we explain this protocol in terms of participation of BTCs, in reality certain resource managers are involved and operate on behalf of BTCs. The following chapter discusses such architectural issues in more detail.

example possible, that two BTCs of the same branching transaction simultaneously hold Certify locks on the same data item. Furthermore, BTCs belonging to the same transaction, but executing in different paths, will never read data from each other. For example, let us assume that $BTC_{i,1}$ branched into $BTC_{i,2}$ and $BTC_{i,3}$, and $BTC_{i,2}$ created a new version of x. In case $BTC_{i,3}$ issues a Read operation on x, the version created by its sibling ($BTC_{i,2}$) is ignored.

The logging and recovery techniques required for this algorithm are the same as for the hybrid algorithm described below; they will be discussed in detail later in this chapter.

The above BT-MV-2PL algorithm and the hybrid locking algorithm that follows are "strict", i.e. a BTC does not release any of its locks until it is aborted or committed. In case of a commit, all locks of BTCs along the committing branch are released together after the commit of that transaction.

4.4 Hybrid Branching Transaction Multi-Version Two-Phase Locking

In Chapter 3, we have considered the problem of unlimited branching and introduced a control function to allow branching only if enough system resources are available. If we dynamically turn branching on and off, we must adapt our concurrency control algorithm accordingly; we need a mechanism which can switch between different concurrency control algorithms without having to shutdown the system in between. Such a hybrid algorithm is the subject of this section.

4.4.1 Hybrid Locking Algorithm

The first step towards integrating 2PL, MV-2PL and BT-MV-2PL is to establish the relationships between the various lock types in these algorithms. Read locks in

2PL and MV-2PL are always locks on committed versions of data items[6]. A Read lock in BT-MV-2PL may exist for an uncommitted or a committed data item version. Write locks in 2PL are always on committed versions, whereas Write locks in MV-2PL and BT-MV-2PL exist only for uncommitted versions. Certify locks are only used in MV-2PL and BT-MV-2PL, where they are only applied on committed versions. Comparing the various lock compatibility matrices, it becomes clear that Certify locks in MV-2PL and BT-MV-2PL play the same role as Write locks in 2PL, i.e. they ensure exclusive access to the committed version of a data item. Read locks in 2PL and MV-2PL make sure that no other transaction can overwrite the committed version of a data item while the reader transaction has not committed (or aborted) yet. As discussed above, under BT-MV-2PL a Read lock on the committed version of an item may be pre-empted by a Certify lock.

To be able to capture the semantics of all three locking algorithms, we use the following five new lock types instead of Read, Write and Certify locks:

VRL (Version Read Lock): equivalent to Read locks in BT-MV-2PL;

VWL (Version Write Lock): equivalent to Write locks in MV-2PL and BT-MV-2PL;

CRL (Certified Read Lock): equivalent to Read locks in 2PL and MV-2PL;

CWL (Certified Write Lock): equivalent to Write locks in 2PL and equivalent to Certify locks in MV-2PL and BT-MV-2PL;

TCRL (Tentative Certified Read Lock): similar to CRL, but it can be preempted;

These relationships between our new lock types and the traditional Read, Write and Certify locks are summarized in Table 4–4 (= indicates equivalence, p indicates equivalence with the possibility of a lock being preempted).

[6]We are assuming a version of MV-2PL which does not read uncommitted data.

	2PL		MV-2PL			BT-MV-2PL		
	R	W	R	W	C	R	W	C
VRL						=		
VWL				=			=	
CRL	=		=					
CWL		=			=			=
TCRL	P		P					

Table 4–4: Lock Type Relationships

During the execution of a transaction, the mode of concurrency control may dynamically change between 2PL, MV-2PL and BT-MV-2PL. Depending on which mode is active at the time of request, Read and Write requests must be handled in different ways. The six possible locking request scenarios are as follows (NOTE: CWL and CRL locks are always on the committed version of data items; VRL and VWL are always on uncommitted versions of data items):

- **Read Request** under **2PL:** the BTC must first obtain a CRL on the requested item. It then reads the committed version of the item.

- **Write Request** under **2PL:** the BTC must first obtain a CWL on the requested item. It then overwrites the existing committed version of the item[7].

- **Read Request** under **MV-2PL:** same as Read request under 2PL.

- **Write Request** under **MV-2PL:** the BTC simply creates a new version of the corresponding data item and is automatically granted a VWL for it.

[7]It does not actually overwrite the committed version, since we apply a deferred-update recovery policy, but the old committed version is not accessible to any other BTC any longer, unless it is recovered due to a failure of the BTC which has overwritten it.

- **Read Request** under **BT-MV-2PL:** if no uncommitted versions exist, the BTC simply reads the committed version and a TCRL is set. If uncommitted versions exist, branching takes place; one new BTC for each existing version that has not been created by another BTC of the same transaction (which is branching). The new BTCs read the corresponding versions; accordingly VRLs are set on the relevant uncommitted data item versions and a TCRL is set on the committed version.

- **Write Request** under **BT-MV-2PL**: same as Write request under MV-2PL.

As with MV-2PL and BT-MV-2PL, before a BTC can be committed, it must be certified. Certification follows the same rules as before: 1) a BTC must wait until all versions it has read have become committed, and 2) all Write locks (VWL) must be certified, i.e. a CWL must be obtained. VRLs are automatically converted to CRLs once the corresponding writer BTCs have committed. A TCRL must explicitly be upgraded to CRL before the corresponding BTC can be certified. The circumstances under which a TCRL can be upgraded, as well as all other lock compatibility modes (as described in Table 4–5) are discussed next.

VRLs and VWLs are never rejected: they do not conflict with any other locks. There exists, however, by definition, always only one VWL per uncommitted version. Since version locks (VRL and VWL) only exist on uncommitted data and certified locks (CRL and CWL) are only requested for committed data, there can be no conflict between these two groups. CRLs are compatible with TCRLs and other CRLs. TCRLs are compatible with all other locks, except CWLs. CWLs are incompatible with TCRLs, CRLs and other CWLs.

If a lock request is made for which there exists an incompatible lock already, the requesting BTC is blocked. As with all blocking concurrency control algorithms, deadlocks must be dealt with. A later section in this chapter will deal with deadlock detection and resolution in more detail.

The relationship between CWLs and TCRLs is somewhat different from all other lock conflicts; depending on the circumstances, a CWL may pre-empt a

TCRL and abort the corresponding BTC. We allow this if the BTC which is holding that TCRL has a "sibling" BTC that read the version of the item which was created by the BTC which is requesting the CWL; this sibling allows the possibility that the corresponding transaction can eventually commit (without a restart) in spite of the abort of the branches which contain the BTC with the pre-empted TCRL. We illustrate this case later in Example 3. Table 4–5 shows the lock compatibility matrix for HBT-MV2PL.

HBT-MV2PL	*Lock Held*				
Lock Requested	VRL	VWL	TCRL	CRL	CWL
VRL	yes	yes	yes	yes	yes
VWL	yes	yes	yes	yes	yes
TCRL	yes	yes	yes	yes	no
CRL	yes	yes	yes	yes	no
CWL	yes	yes	no	no	no

Table 4–5: Lock Compatibility Matrix for HBT-MV2PL

The certification and commitment of transactions under HBT-MV2PL follows the same principles as for BT-MV-2PL. Before a transaction can commit, all branching transaction components (BTCs) along one branch of the transaction must be certified. For a BTC to become certified, all its version locks must be upgraded to the appropriate certified locks (VRL→CRL, VWL→CWL) and all TCRLs must be upgraded to CRLs (TCRL→CRL).

Once all BTCs along one path are certified, the transaction manager follows the same protocol described for BT-MV-2PL's commit policy in order to ensure that either all or none of the BTCs along the certified path commit. As in the case of BT-MV-2PL, if a transaction did not branch, it is simply enough to certify the transaction's only BTC and commit its changes to the database.

If a transaction was entirely or partially executed in 2PL mode or MV-2PL mode, some or all of the locks it holds are certified locks. The certification process of such transactions is only concerned with their VRLs, VWLs and TCRLs. In the special case of a transaction which was executed entirely in 2PL mode, there is no need for certification since all locks are either CRLs or CWLs, and since only one BTC has been created — branching only happens in BT-MV-2PL mode — there is also no need for a 2 phase commit protocol (as described above).

4.4.2 Properties of HBT-MV2PL

We will not formally establish the correctness of HBT-MV2PL in this chapter, but instead describe several properties of it. We formulate these properties in terms of 5 rules which are always observed by HBT-MV2PL. These rules will be formalised in Chapter 6, and used within a formal proof of correctness.

Overwrite Rule : If a transaction commits, it must have acquired a CRL on every item it read. It follows that it must have read the most recently committed versions of these data items. Another transaction cannot overwrite the (committed) versions read by the committing transaction, since it would have to acquire corresponding CWLs, which are incompatible with the existing CRLs of the committing transaction, and the CRLs are not released until the transaction has committed.

Write Lock Rule : When a transaction writes a data item, it can either overwrite the existing committed version, in which case it needs to acquire a CWL first, or it can create a new version of the item first and then overwrite the existing committed version at commit time. In the latter case it is sufficient to acquire a VWL first, which is then later (during certification) upgraded to a certified write lock. In both cases it holds that a CWL must have been obtained for an item that is updated, before the updating transaction can commit.

Read Lock Rule : If a transaction reads the committed version of a data item, it must acquire a CRL or TCRL for it first. A TCRL must be upgraded to a CRL before a transaction can commit. If the transaction reads an uncommitted version it acquires a VRL on that version, but the transaction which created the version must commit before the reading transaction can, which automatically upgrades the corresponding VRLs to CRLs. Hence, in all cases it holds that a CRL is obtained for every read item before a transaction can commit.

Write/Write Lock Conflict Rule : A transaction can only acquire a CWL on a data item version if no other transaction holds a CWL on the same item. In other words, no two CWLs can exist for the same data item at any point in time.

Read/Write Lock Conflict Rule : A transaction can only obtain a CWL on a data item if no other transaction holds a CRL on that data item at the same time, and vice versa, i.e. CRL and CWL are not compatible.

4.4.3 HBT-MV2PL Example

To illustrate various aspects of HBT-MV2PL, in Example 3 we now describe a possible execution schedule for a set of 5 transactions and show the corresponding lock tables at various stages.

To simplify our example, we only use two possible modes: 2PL and BT-MV-2PL. This is sufficient to illustrate all points of interest. Adding MV-2PL would not add any more useful scenarios. The switch between 2PL and BT-MV-2PL is assumed to be caused by some kind of resource contention, which we leave unspecified here. It should be noted though, since branching mostly requires additional CPU overhead and multi-version algorithms have increased memory requirements, a sensible policy for switching between these three modes is to change from BT-MV-2PL to MV-2PL if CPU utilisation is high and from MV-2PL to 2PL if memory becomes a bottleneck.

Example 3 T_1*:* $r[x]\ w[x]\ r[z]$

T_2*:* $w[y]\ r[x]$

T_3*:* $w[z]\ r[u]$

T_4*:* $w[x]$

T_5*:* $r[x]$

We assume that initially the system operates in BT-MV-2PL mode and there exist no uncommitted versions for items u, x, y and z. At first $BTC_{1,1}$ starts execution in behalf of T_1. Since it wants to read item x, it sends a Read request to the scheduler. A TCRL lock is granted on the committed version of x and $BTC_{1,1}$ reads x. The next request by $BTC_{1,1}$ is a Write operation on x, which leads to a new version of it: ($x_{1,1}$); a new VWL is set accordingly. Now $BTC_{2,1}$ starts execution and writes a new version of y ($y_{2,1}$); a corresponding VWL is set for it. $BTC_{2,1}$'s next operation is a Read on x. Since we currently operate under BT-MV-2PL and more than one version of x exists, branching takes place. Two new components are created: $BTC_{2,2}$ and $BTC_{2,3}$. $BTC_{2,2}$ reads the original version of x; we denote it by $x_{0,0}$. $BTC_{2,3}$ reads $x_{1,1}$. TCRL and VRL locks are set accordingly. At this stage, the lock tables look as shown in Figure 4–3.

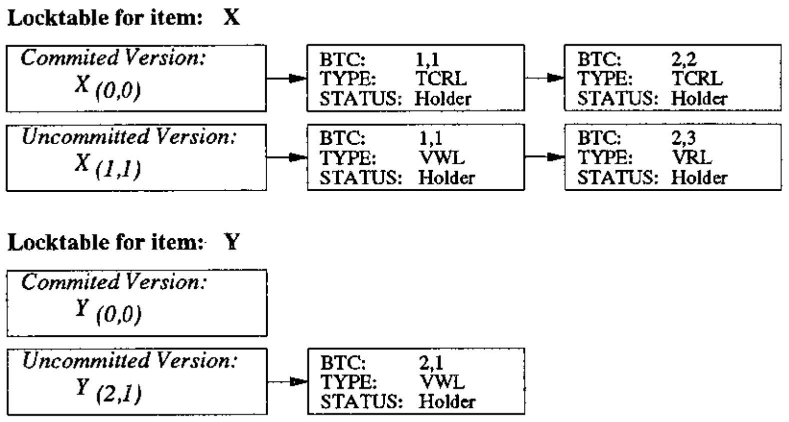

Locktable for item: X
Commited Version: X (0,0)
BTC: 1,1 TYPE: TCRL STATUS: Holder
BTC: 2,2 TYPE: TCRL STATUS: Holder
Uncommitted Version: X (1,1)
BTC: 1,1 TYPE: VWL STATUS: Holder
BTC: 2,3 TYPE: VRL STATUS: Holder
Locktable for item: Y
Commited Version: Y (0,0)
Uncommitted Version: Y (2,1)
BTC: 2,1 TYPE: VWL STATUS: Holder

Figure 4–3: HBT-MV2PL Locktable

We now assume that due to resource shortage the concurrency control mode is dynamically switched to 2PL. At this point T_3 is submitted for execution and $BTC_{3,1}$ requests write access to item z. Since the system operates in 2PL mode, a CWL on z is requested and granted; we assume that no other transaction has accessed z at that moment. Next, $BTC_{1,1}$ wants to read item z. The corresponding CRL request is blocked because of $BTC_{3,1}$'s CWL on z.

Due to a drop in resource utilisation, the concurrency control mode is now switched back to BT-MV-2PL. $BTC_{3,1}$ requests a Read on item u, is granted a TCRL on it and reads the committed version of the item. The lock tables at this stage are shown in Figure 4–4.

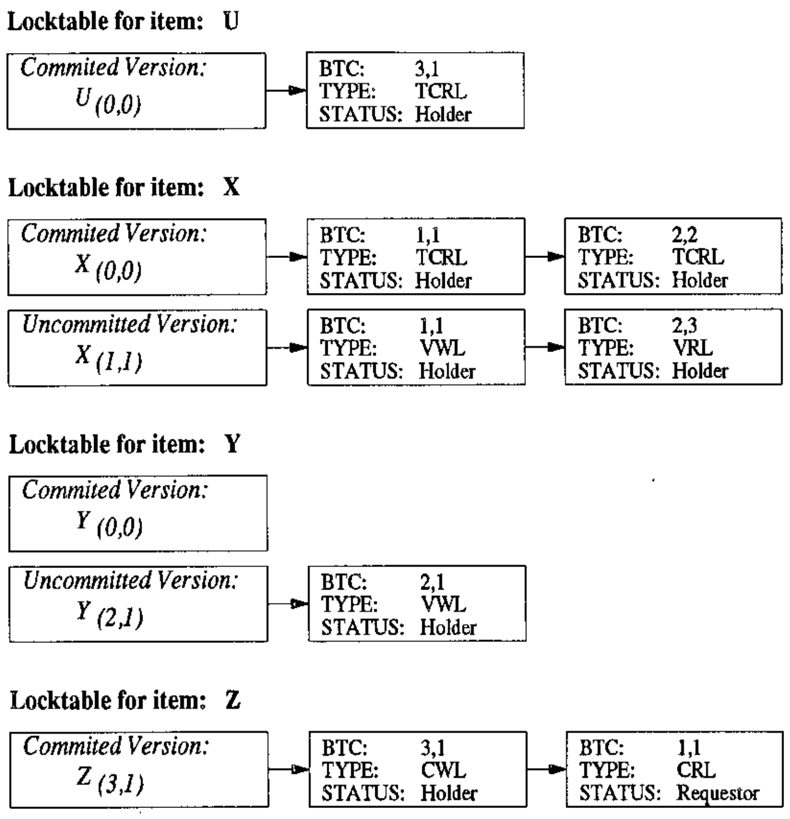

Locktable for item: U
Commited Version: U (0,0)
BTC: 3,1 TYPE: TCRL STATUS: Holder
Locktable for item: X
Commited Version: X (0,0)
BTC: 1,1 TYPE: TCRL STATUS: Holder
BTC: 2,2 TYPE: TCRL STATUS: Holder
Uncommitted Version: X (1,1)
BTC: 1,1 TYPE: VWL STATUS: Holder
BTC: 2,3 TYPE: VRL STATUS: Holder
Locktable for item: Y
Commited Version: Y (0,0)
Uncommitted Version: Y (2,1)
BTC: 2,1 TYPE: VWL STATUS: Holder
Locktable for item: Z
Commited Version: Z (3,1)
BTC: 3,1 TYPE: CWL STATUS: Holder
BTC: 1,1 TYPE: CRL STATUS: Requestor

Figure 4–4: HBT-MV2PL Locktable

$BTC_{3,1}$ has completed its execution and wants to commit. It must, therefore, be certified first. Its TCRL (on u) can be upgraded to a CRL since no other locks exist on that item, and its write lock on z is already certified since it was obtained under 2PL mode. $BTC_{3,1}$ never branched and, therefore, all BTCs for transaction T_3 are certified and the transaction can commit. As all its locks are released, $BTC_{1,1}$'s CRL on z can be granted now.

After another switch to 2PL mode, T_4 and T_5 arrived and requested Write and Read access to x, respectively. The corresponding CWL request from $BTC_{4,1}$ was blocked, because of the existing TCRLs on x. $BTC_{5,1}$'s CRL request on x was blocked, because of the blocked CWL by $BTC_{4,1}$. Figure 4–5 shows the lock tables at this stage.

After the switch back to BT-MV-2PL mode, $BTC_{1,1}$ has reached the end of its execution and needs to be certified. Its TCRL on x can be upgraded to a CRL, since there are currently no CWL holders for x. The Read on z is already

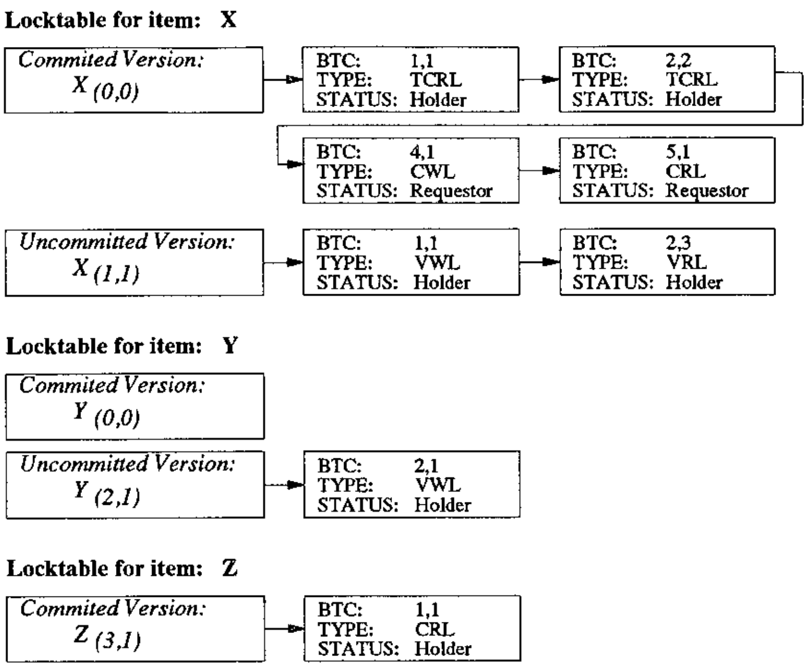

Locktable for item: X
Commited Version: X (0,0)
BTC: 1,1 TYPE: TCRL STATUS: Holder
BTC: 2,2 TYPE: TCRL STATUS: Holder
BTC: 4,1 TYPE: CWL STATUS: Requestor
BTC: 5,1 TYPE: CRL STATUS: Requestor
Uncommitted Version: X (1,1)
BTC: 1,1 TYPE: VWL STATUS: Holder
BTC: 2,3 TYPE: VRL STATUS: Holder
Locktable for item: Y
Commited Version: Y (0,0)
Uncommitted Version: Y (2,1)
BTC: 2,1 TYPE: VWL STATUS: Holder
Locktable for item: Z
Commited Version: Z (3,1)
BTC: 1,1 TYPE: CRL STATUS: Holder

Figure 4–5: HBT-MV2PL Locktable

certified, since it was requested under 2PL mode. The upgrade from a VWL to a CWL on x can also be granted to $BTC_{1,1}$, in spite of the existence of $BTC_{2,2}$'s TCRL; that TCRL can be pre-empted since there exists an alternative branch, $BTC_{2,3}$, which has read the new version of x written by $BTC_{1,1}$. Due to the pre-emption of its TCRL, $BTC_{2,2}$ is aborted. Once T_1 committed, $BTC_{2,3}$'s VRL on x is automatically promoted to a CRL. Since it is now a lock on the committed version of x, it has been moved from the queue for an uncommitted version to the queue for the committed version of x[8]. Since this CRL is actually held by $BTC_{2,3}$ rather than requested, it is placed in front of all currently blocked requestors. Hence, the CWL requested by $BTC_{4,1}$ and the CRL requested by $BTC_{5,1}$ are now queued after the CRL of $BTC_{2,3}$. The locktable for item x after $BTC_{1,1}$'s certification and then after T_1's commit is shown in Figure 4–6.

At this point $BTC_{2,3}$ reaches end of execution. Certification of $BTC_{2,1}$ and

[8]Note that there may be zero, one or more queues for uncommitted versions of data items, but there is always only one "committed version" queue, even though the committed version changes, in this case from $x_{0,0}$ to $x_{1,1}$.

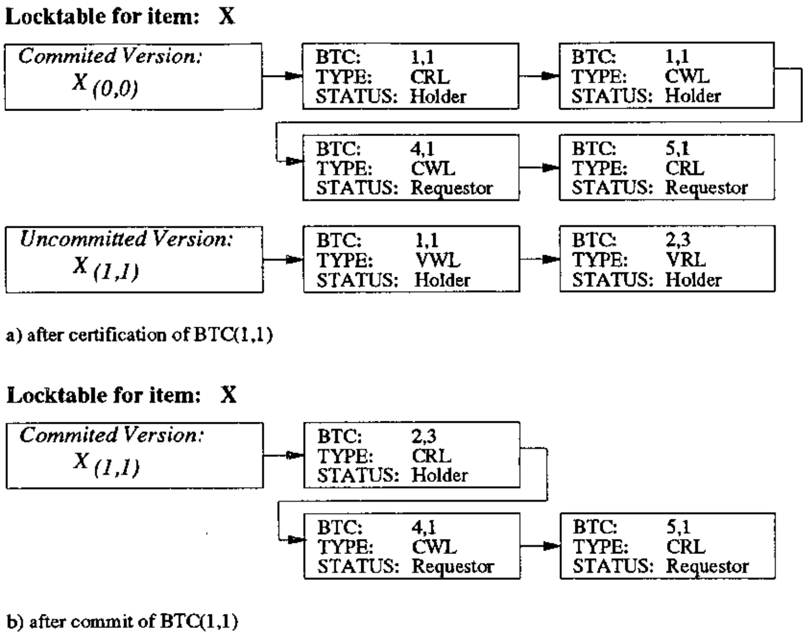

Locktable for item: X
Commited Version: X(0,0)
BTC: 1,1 TYPE: CRL STATUS: Holder
BTC: 1,1 TYPE: CWL STATUS: Holder
BTC: 4,1 TYPE: CWL STATUS: Requestor
BTC: 5,1 TYPE: CRL STATUS: Requestor
Uncommitted Version: X(1,1)
BTC: 1,1 TYPE: VWL STATUS: Holder
BTC: 2,3 TYPE: VRL STATUS: Holder
a) after certification of BTC(1,1)
Locktable for item: X
Commited Version: X(1,1)
BTC: 2,3 TYPE: CRL STATUS: Holder
BTC: 4,1 TYPE: CWL STATUS: Requestor
BTC: 5,1 TYPE: CRL STATUS: Requestor
b) after commit of BTC(1,1)

Figure 4–6: HBT-MV2PL Locktable

$BTC_{2,3}$ takes place and T_2 commits. The remainder of the execution of $BTC_{4,1}$ and $BTC_{5,1}$ is omitted here, since it does not illustrate any particular aspect of HBT-MV2PL.

4.4.4 HBT-MV2PL Layered Approach

Implementing a branching transaction scheduler from scratch is not an easy task. To be able to make use of existing DBMS components, we suggest a two-layer approach for the development of a HBT-MV2PL concurrency control manager, where the lower layer captures general (non-branching) 2-phase locking, and the higher layer provides full HBT-MV2PL functionality. Separating the scheduler in these two components supports the possibility of reusing existing 2PL scheduler code.

In addition to the usual Read and Write lock, and Commit and Abort requests, a 2PL module used in our two layer model would also need to support the following features:

Read/upgrade: a Read lock request which is granted if no Write locks exist for that item at that time (blocked Write lock requests may exist). If the lock cannot be granted, the requesting BTC must be aborted. This request is needed to upgrade TCRLs to CRLs.

Read/set: Sets a Read lock for a given BTC. This is needed to upgrade a VRL to a CRL.

Write/upgrade: A Write lock request which, if it cannot be granted, is queued in front of other blocked requests; except for other blocked upgrade requests. Upgrade requests from VWL to CWL should be handled with higher priority than regular lock requests.

We will now describe how requests to the HBT-MV2PL layer can be implemented on top of a 2PL layer which includes the aforementioned features. Figure 4–7 shows the overall two-layer structure.

A Write request to the HBT layer can directly be passed as a Write request to the 2PL layer, if the current CCM mode is 2PL. Otherwise, in MV-2PL and BT-MV-2PL mode, a VWL entry is made in the HBT lock table. Under 2PL and MV-2PL mode, Read requests are passed directly to the 2PL layer. A Read request under BT-MV-2PL mode is entirely handled by the HBT-MV2PL layer: it decides whether branching needs to be performed and sets TCRLs and VRLs accordingly.

When certification of a BTC is requested, the HBT-MV2PL layer first waits until all VRLs for that BTC have been upgraded to CRL. It then tries to upgrade all of this BTC's TCRLs by making Read/upgrade requests to the 2PL layer. Finally, it tries to upgrade the BTC's VWLs. For each VWL it must first determine if there are any conflicting TCRLs and whether they can be preempted. If no conflicting TCRLs remain, a Write/upgrade request is made to the 2PL layer.

To commit a BTC, the HBT-MV2PL layer must upgrade all VRLs which have read data which was written by the committing BTC. It does so by sending Read/set requests to the 2PL layer for the appropriate items. All locks held by

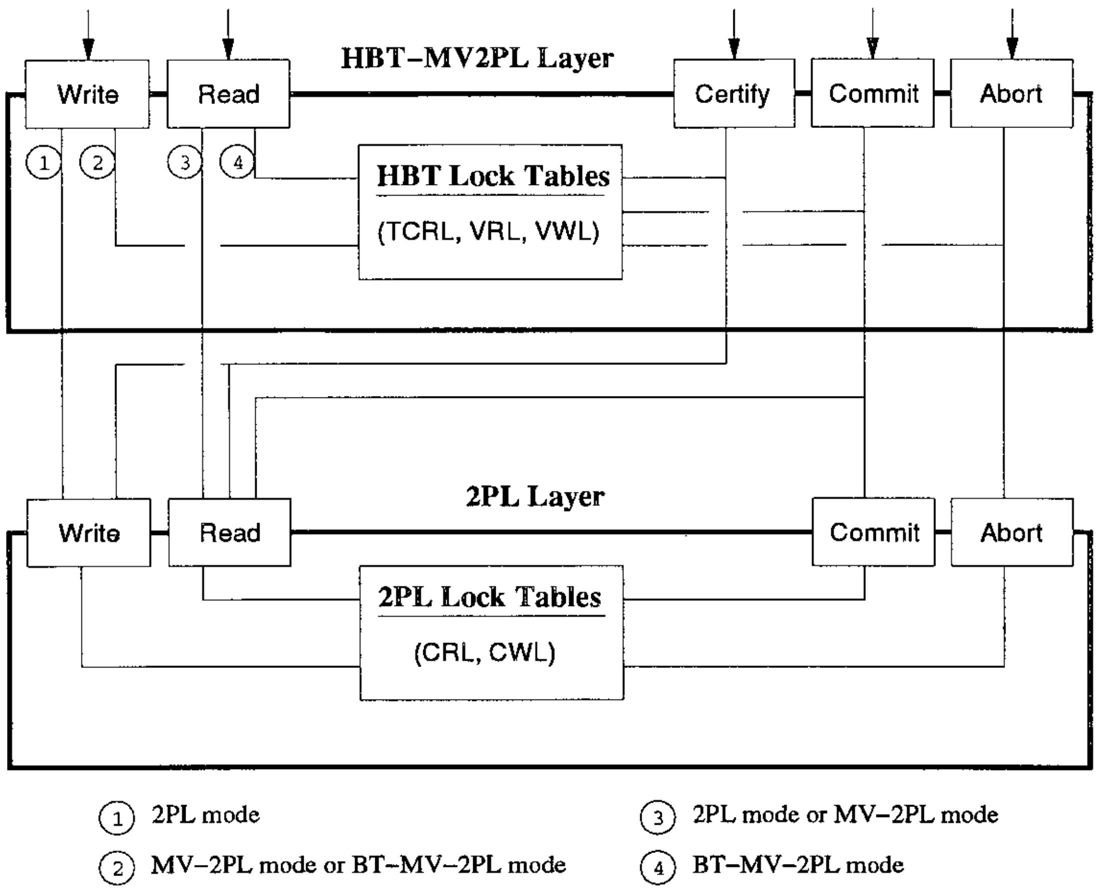


Figure 4–7: HBT-MV2PL Two-Layer Structure

the committing BTC are then released in the HBT lock table. A commit message is sent to the 2PL layer to release the BTC's locks there as well.

Abort processing also affects both layers. Hence, in addition to aborting the BTC within the HBT-MV2PL layer, an Abort request is also sent to the 2PL layer.

Any DBMS vendor considering the implementation of a branching transaction system as part of their DBMS will clearly consider the costs involved in developing the corresponding software and compare these with the potentially higher sales of their product. Only if the expected costs do not outweigh the perceived benefits, will it be of interest to pursue the idea of branching transactions in a commercial system. The proposed two-layer structure described above is one step towards keeping such software development costs sufficiently low.

4.5 Deadlock Detection and Resolution

Due to the blocking nature of HBT-MV2PL, deadlocks between transactions are possible. These deadlocks must be detected and resolved. Traditional deadlock detection and resolution policies cannot be applied, since they are unable to deal with the particular aspects of branching transactions. Before discussing solutions to this problem, we first describe what makes deadlock handling in branching transaction systems different.

In a non-branching environment, transactions are blocked by other transactions, whereas in a BT system, branching transaction components are blocked by other branching transaction components. As a result, it is quite possible that there exists a deadlock between BTCs of two (or more) transactions although there is no deadlock between these two (or more) transactions. The following example illustrates this point.

Example 4 *Let us assume the following two transactions:*

$$T_1\textit{:}\ w[x]\ w[y]$$
$$T_2\textit{:}\ r[x]\ r[y]$$

Initially the system is in BT-MV-2PL mode. $BTC_{1,1}$ writes a new version of x. Next, $BTC_{2,1}$ wants to read x and branching takes place: $BTC_{2,2}$ reads the committed version of x, $BTC_{2,3}$ reads $x_{1,1}$. At this time, the concurrency control mode switches to 2PL. The new components of T_2 obtain CRLs on y. When $BTC_{1,1}$ requests Write access to y, it is blocked since the corresponding CWL cannot be granted. If $BTC_{2,3}$ was to request certification at this stage, a deadlock between $BTC_{1,1}$ and $BTC_{2,3}$ occurs. $BTC_{1,1}$ is blocked by $BTC_{2,3}$ because of the CRL/CWL conflict on y. $BTC_{2,3}$ is blocked by $BTC_{1,1}$ since its VRL on x cannot be certified until $BTC_{1,1}$ commits. There is, however, still possibility for both transactions to commit without any special handling of this deadlock situation: if $BTC_{2,2}$ reaches its end of execution, it can be certified immediately — its TCRL on

x can be upgraded to a CRL and it already holds a CRL on y — and since $BTC_{2,1}$ hasn't actually performed any operation, this branch of T_2 can be committed. This leads to an Abort of $BTC_{2,3}$ and a resolution of the deadlock situation.

From the above example it follows that we need to distinguish between two kinds of deadlocks in a branching transaction system: *branching transaction component deadlocks* and *branching transaction deadlocks.*

Definition 2 *A set of branching transaction components S_{BTC} is deadlocked, i.e. a **branching transaction component deadlock** exists, if each component (BTC) in S_{BTC} is blocked by one or more other components (BTCs) in S_{BTC}.*

Definition 3 *A set of branching transactions S_{BT} is deadlocked, i.e. a **branching transaction deadlock** exists, if* all *transaction components of transactions in S_{BT} are deadlocked.*

The above example also shows that a BTC may be blocked not only by another BTC because of a lock conflict, but also because of a read dependency. This read dependency wait-for relationship is not unique to branching transactions, but is a consequence of applying multi-version concurrency control and allowing BTC's to read uncommitted data. There is, however, a further wait-for relationship, which is due to parent-child relationships between BTCs. Again, we use examples to illustrate the issue.

Example 5 *Let us assume the following two transactions:*

$$T_1\!: r[x]\ r[y]\ w[z]$$
$$T_2\!: r[z]\ w[x]$$

At the beginning, 2PL mode is active and $BTC_{1,1}$ and $BTC_{2,1}$ acquire CRLs on x and z, respectively. Let us suppose that, during a phase of BT-MV-2PL mode, $BTC_{1,1}$ has to branch because of the existence of an uncommitted version of y. Subsequently, once again in 2PL mode, $BTC_{2,1}$ attempts to update x, but is blocked due to a CWL/CRL conflict with $BTC_{1,1}$; and $BTC_{1,2}$ is trying to acquire *a CWL for z, but is blocked due to a CWL/CRL conflict with $BTC_{2,1}$. If we further assume that $BTC_{1,3}$ has been aborted due to reading a version of y which has meanwhile been aborted, Figure 4–8 describes the wait-for situation between the remaining BTCs.*

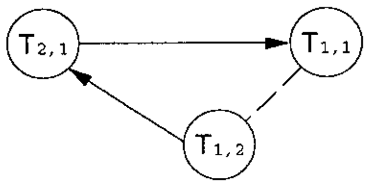


Figure 4–8: BTC-based Wait-For Graph

Although there exists no branching transaction component deadlock (as defined above) in Example 5, $BTC_{1,1}$, $BTC_{1,2}$ and $BTC_{2,1}$ are effectively deadlocked, since $BTC_{1,1}$ is not able to commit before $BTC_{1,2}$ is certified. Hence, we have actually another wait-for dependency between $BTC_{1,1}$ and $BTC_{1,2}$.

One could consider the possibility of adding edges from parent to child BTCs in a wait-for-graph to capture this sort of parent/child dependency. Unfortunately, this is not a solution to the problem as can be demonstrated with Example 6.

Example 6 *Let us assume the situation of Figure 4–9. $BTC_{i,2}$ may have read a data item version created by $BTC_{j,1}$ and it, therefore, needs to wait for $BTC_{j,1}$'s commitment before it can be certified. $BTC_{j,1}$ is blocked by $BTC_{i,4}$; there might be some CRL/CWL conflict similar to the ones described in the previous example. Although $BTC_{i,4}$ can be certified, it will not commit until $BTC_{i,2}$ has also been certified, and hence, we arrive at a deadlock situation involving $BTC_{i,2}$, $BTC_{i,4}$ and $BTC_{j,1}$. Adding a $BTC_{i,2} \rightarrow BTC_{i,4}$ edge to the wait-for graph would not solve the problem, since the wait-for dependency in this case goes from child to parent BTC.*

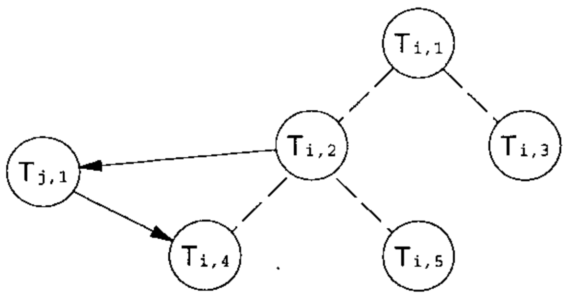


Figure 4–9: BTC-based Wait-For Graph

4.5.1 Branch-Based Wait-For-Graphs for Branching Transactions

The key to the above problem is the fact that no BTC can commit on its own, but only as part of an entire path of a branching transaction. Hence, if any BTC along a branch is in a wait-for relationship with some other BTC, then effectively all other BTCs of that path are also included in this wait-for dependency. This leads to yet another definition of deadlocks in a BT system:

Definition 4 *A set of branching transaction branches S_{BTB} is deadlocked, i.e. a* ***branching transaction branch deadlock*** *exists, if each branch in S_{BTB} is blocked by one or more other branches in S_{BTB}.*

To detect such branch-based deadlocks, we use a special variation of wait-for-graphs (WFG), where a node represents an entire branch of a transaction — each branch is identified by its leaf-node BTC — and there exists an arc from $BTC_{i,j}$ (a leaf node BTC of T_i) to $BTC_{k,l}$ (a leaf node BTC of T_k), if $BTC_{i,j}$ or any of its ancestors is blocked by $BTC_{k,l}$ or any of its ancestors. As we pointed out earlier, a BTC may be blocked by another BTC for two reasons: 1) a BTC has read data which was created by another BTC which has not committed yet, and 2) a lock conflict between two BTCs.

Proposition 1 *Let S_{BT} be a set of branching transactions, S_{BTB} the set of all branches of transactions in S_{BT}, and S_{BTC} the set of all branching transaction components of transactions in S_{BT}. If there exists a branching transaction com-*

ponent deadlock between components in S_{BTC}, then there also exists a branching transaction branch deadlock between branches in S_{BTB}.

Proof: A component deadlock exists if there is a cyclic wait-for relationship between components in S_{BTC}. For every wait-for arc between two components, there is a wait-for arc between the corresponding transaction branches in S_{BTB}. Hence, a cycle in a component-based WFG has a corresponding cycle in the branch-based WFG. To prove that not every branch-based deadlock is associated with a component-based deadlock, we refer to the counter example in Example 5. □

Proposition 1 has important implications for the definition of a branching transaction deadlock: it is possible for a set of branching transactions to be deadlocked, even so not all of its BTCs are involved in a component-based deadlock. A branch-based deadlock which involves all branches of all branching transactions in S_{BT} is sufficient for a system wide BT deadlock. We modify the definition of a branching transaction deadlock accordingly:

Definition 5 *A set of branching transactions S_{BT} is deadlocked, i.e. a* ***branching transaction deadlock*** *exists, if* all *branches of transactions in S_{BT} are deadlocked.*

4.5.2 Deadlock Detection and Resolution

When implementing a deadlock handling mechanism for branching transactions, one of the decisions to be made is when deadlock detection should take place. The two basic options are: 1) when a transaction gets blocked, or 2) periodically. By choosing the first option, we can eliminate deadlocks as soon as they occur, but the overhead for deadlock detection may be considerable if blocking takes place often. Using periodic checks limits the overhead caused by deadlock detection, but if time intervals between checks are too long, deadlocks may remain unresolved for relatively long periods. The correct choice depends on the frequency with which deadlocks occur: the more often deadlocks happen, the more often the detection mechanism should be called. Clearly, checking for deadlocks every time a transaction gets blocked is only useful if deadlocks develop relatively often.

Once a deadlock has been detected, we must distinguish whether a branch-based deadlock occurred which does not involve all active branches of transactions — in this case the deadlock may still be resolved without any need for intervention (as seen in Example 5) — or if indeed a *branching transaction deadlock* exists and some deadlock resolution policy must be applied.

In case of a branch-based deadlock, which may or may not resolve itself later, the deadlock manager can either intervene immediately and resolve this deadlock, or it can delay any action until an actual branching transaction deadlock occurs. Not resolving it immediately allows for the possibility that the deadlock gets resolved "naturally", i.e. the branches which are aborted are those which need to be aborted anyway due to the commit of alternative branches or due to cascading rollbacks.

Once the decision has been made that a deadlock needs to be resolved, one or more branching transaction components must be selected for abort. Various *victim selection policies* can be applied. A simple solution is to abort the most recently blocked BTC. Alternatively, one might take into consideration the progress which has been made by all victim candidates so far, and then abort those which have made the least progress so far. Yet another possibility is to consider possible alternative branches of victim candidates; the prefered victim BTCs are those for which there exists an alternative branch which may still succeed in committing a branching transaction in spite of the forced abort of one of its branches. Again, there is a tradeoff: to apply a good victim selection policy may cause considerable overhead in deciding on the victim.

4.5.3 Deadlock Prevention using Cautious Waiting

Instead of allowing deadlocks to occur, detecting and then resolving them, several *deadlock prevention* algorithms have been proposed in the past, e.g. Wait-Die and Wound-Wait [77], Cautious Waiting [47] and Running Priority [37]. These algorithms are designed to prevent any deadlocks from occurring. In this section we describe a variation of Cautious Waiting (CW) for branching transactions which has been used as the deadlock handling mechanism in our simulation study in Chapter 7.

Under Cautious Waiting (for flat transactions), a transaction is aborted if it gets blocked by another transaction which itself is already blocked. This prevents any cyclic wait-for dependencies; deadlocks cannot, therefore, occur.

In the case of branching transactions, a BTC is aborted if it gets blocked by another BTC which belongs to a transaction which has all of its own BTCs either blocked, aborted or branched. This will prevent branching transaction deadlocks, as defined in Definition 5, since it prevents cycles of blocked transactions.

To deal with issues of transaction aborts due to deadlock handling as well as other reasons, e.g. system failure, some recovery mechanism must be in place in a branching transaction processing system. Logging and recovery is, therefore, discussed next.

4.6 Logging and Recovery

In Chapter 2 we discussed the need for a recovery manager and the kind of logging and recovery techniques which have been proposed in the past. To maintain database consistency in spite of various failure situations, a branching transaction processing system must also apply some form of recovery management.

Since a branching transaction system follows several alternative paths of execution of transactions in parallel, and only one path per transaction can eventually be committed, aborting of BTCs is a frequent operation and must, therefore, be performed as efficiently as possible.

To avoid excessive I/O overhead for BTC roll-backs we apply a modified *incremental log with deferred updates* strategy [52]. During execution of a BTC, all updates to the database are kept as new versions of an item in main memory only; the database on disk is not updated and no log records are written. If a BTC aborts, all data versions created by it are simply discarded; no I/O is necessary for aborting BTCs. To commit a BTC, for each item to be written, first a log record is written and then the database is updated on disk. Assuming that log records and the database are stored on different disks, I/O to log disks and database disks can proceed in parallel (as long as for each item the log record is written before updating the database). Recovery from a system crash works as with traditional recovery techniques (see [10]). Since no log records are written until commit time — the time at which it is known which path of execution of a branching transaction is correct — logging is only required for one path of a branching transaction, and hence, branching does not impose any extra I/O overhead for logging on the system; in both cases, branching and non-branching, two I/O accesses (one to write a log record and one to update the database) are required per database update.

Chapter 5

System Architecture

The previous two chapters introduced the basic concept of branching transactions and described concurrency control and recovery mechanisms for this new transaction model. So far, the discussion has been on an abstract level and did not consider aspects of the underlying computer architecture on which we expect a branching transaction processing system to run. Clearly though, the actual system architecture is to some extent influenced by what kind of parallel computer we intend to use. In this chapter, therefore, we describe branching transaction system architectures for various parallel computer configurations. In addition to the overall architecture, issues of cache coherence, centralised vs. distributed lock management and load balancing and load control are discussed in more detail.

We begin this chapter with a review of current parallel computer architectures and a classification model of parallel DBMS. This creates a context which relates hardware and software aspects of a parallel database system within which we then discuss various issues of branching transaction system architectures.

5.1 Parallel Computer Architecture

The most commonly used categorisation of parallel computer architectures for database management systems is the one proposed by Stonebraker [84]. He distinguishes the following architectures:

shared-memory (SM): [1] all processors access one global, shared memory and the same set of disks;

shared-disk (SD): each processor has its own private memory, but can access all of the disks;

shared-nothing (SN): each processor has its own private memory and has exclusive access to its own set of disks;

A 4th category has evolved in recent years and is described by Valduriez [93]:

shared-something (SS): the multi-computer comprises a set of nodes, where each node is a shared-memory multi-processor, possibly with its own disks. The nodes communicate by message passing through an interconnect.

Although Valduriez suggests that the shared-memory nodes access a set of shared disks, it is also possible that each node has its own set of disks instead. An example of such an architecture is the Convex Exemplar [91], which we will discuss in more detail below. In the following sections we will take a closer look at each of these architectures. Figure 5–1 shows the four architectures discussed in this chapter. (One should note that although commercial systems are classified according to these architectures, they do not always strictly follow their definitions. For example, commercial shared-nothing systems quite often have some level of disk sharing for fault tolerance purposes.)

---

[1]As mentioned in Section 2.2.3, this refers to physically shared memory and not distributed shared memory.

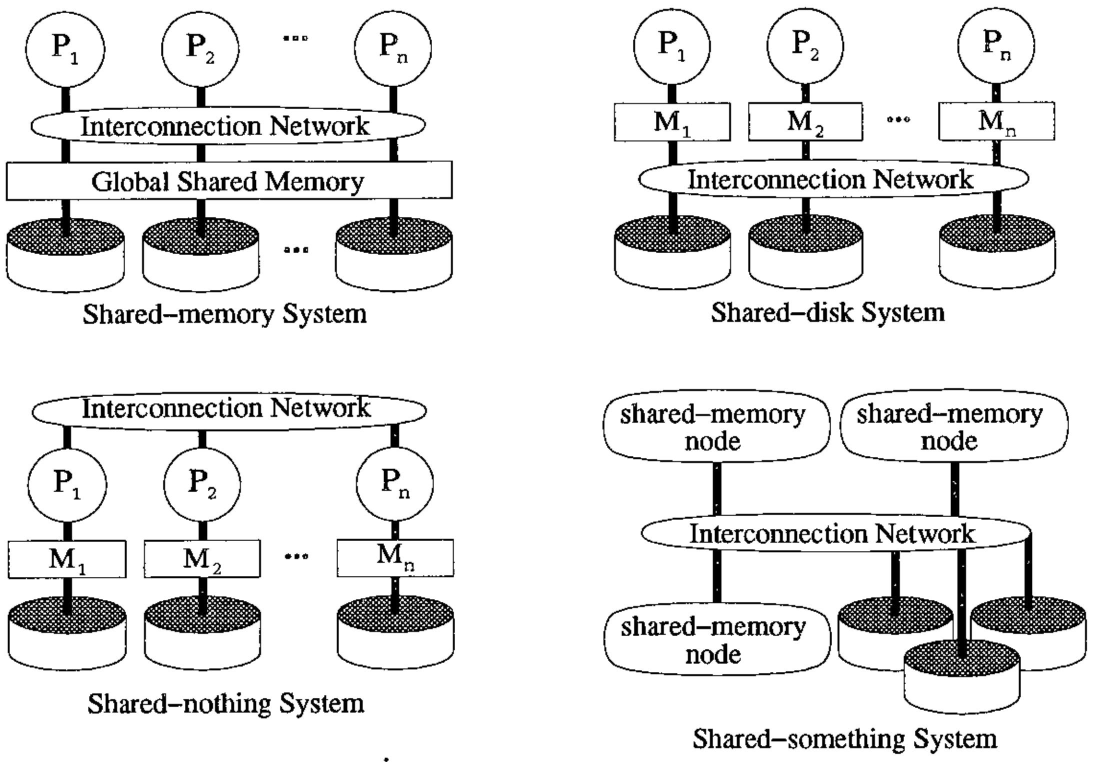


Figure 5–1: Parallel Hardware Architectures

5.1.1 Shared Memory

In a *shared memory* architecture[2] each processor has access to a global, shared memory and to each disk in the system. Each processor typically uses its own cache memory (processor cache) to improve access times to the memory. Coherence between these caches is often ensured via some hardware mechanism, for example a shared bus which is continuously "snooped" on behalf of each processor; if a processor does not have the latest version of data in its cache, it ignores the cache content and retrieves it from memory.

Shared memory systems allow good load balancing, since tasks can be assigned to any processor; all processors have fast access to all data, since memory as well as disks are shared. There is no significant communication overhead between

[2]Shared memory architectures are also known as *Symmetric Multi-Processor* (SMP) or *shared everything* architectures.

processors, as they communicate via shared memory, instead of sending messages across an interconnect.

The primary disadvantage of shared memory systems is their limited scalability. If the number of processors becomes too large, the bus becomes a bottleneck. Currently, manufacturers claim to be able to build machines with up to 64 processors (Cray CS6400 [91]), although most systems used today use less than that. A second problem of these systems is low availability. Since memory is shared by all processors, a memory fault can effect the entire system. Furthermore, the bus is a single point of failure.

The structure of a shared memory architecture is shown in Figure 5–2.

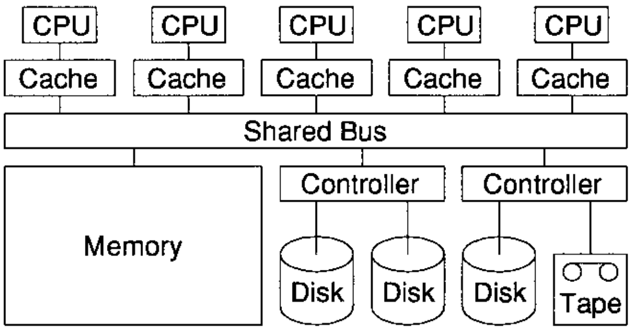


Figure 5–2: Shared Memory Architecture

5.1.2 Shared Disk

In a *shared disk* system, each processor has access to all disks through an interconnect, but exclusive access to its own memory module. As before, processor caches are used to improve access times to memory, but cache coherence is less of a problem as there is no shared memory.

Shared disk systems have better scalability, since memory access is now local to a processor's own memory module; to access data in memory the processor does not have to use a shared bus anymore, but its own local bus. The bottleneck of the shared bus is thereby eliminated. Load balancing is still relatively easy, since all processors can access all data in the database without the need to communicate with other processors; the disks on which the database resides are shared.

Availability is better than in a shared memory system since a memory fault only affects one processor rather than all.

Copying data from the database (shared disks) to processors' own memories, however, creates a new form of coherence problem. Conflicting access to the same page by different processors must be controlled using certain protocols. We will discuss so-called "cache coherence" protocols in more detail below. "Cache" in this context does not refer to the processor cache discussed earlier, but copies of database pages in a processor's local memory. Shared disks may become a bottleneck if too many processors try to access them frequently. To address this issue, one could use larger cache memories for each processor, or replicate data on disk.

The structure of a shared disk architecture is shown in Figure 5–3.

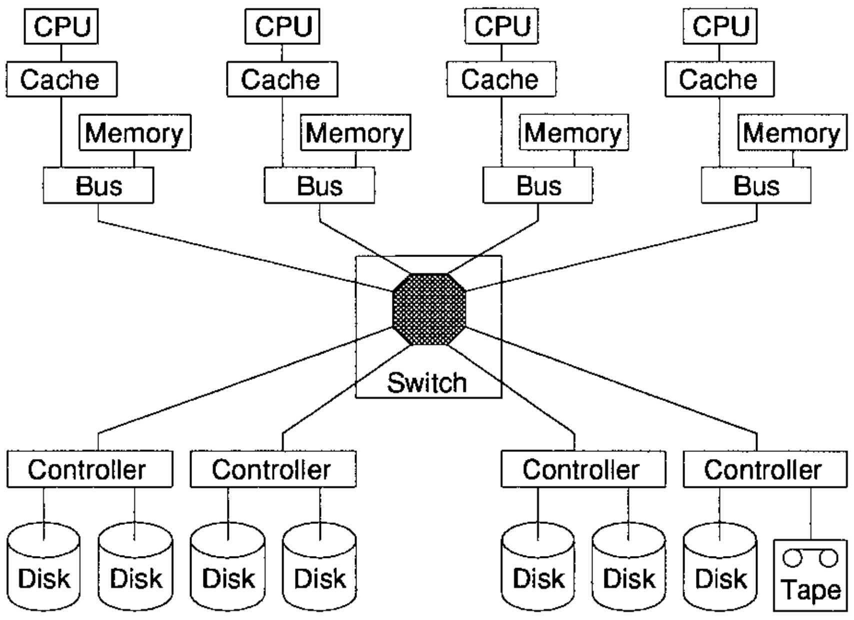


Figure 5–3: Shared Disk Architecture

5.1.3 Shared Nothing

The architecture promoted by several researchers, in particular by Stonebraker [84] and DeWitt and Gray [30], as the most appropriate for parallel database systems is *shared nothing.* In a shared nothing system, each processor has its own private memory — again, processor caches are used — and its own set of disks. Any access to data on disk can only be performed by the processor which "owns" the disk, or by communicating with that processor; the processor acts as a server to that disk. To access their local memory and disks, processors use a local bus; to communicate with other processors, however, some interconnect network is used.

The perceived advantages of shared nothing systems are very good scalability and good availability. Load balancing is known to be more difficult, since communication between processors is expensive and data cannot be shared easily between processors.

The structure of a pure shared nothing architecture is shown in Figure 5–4.

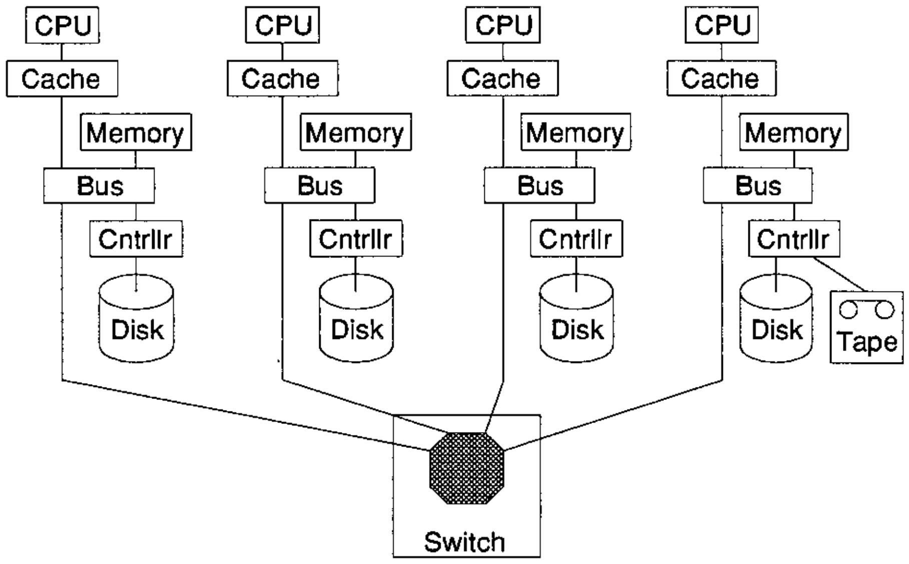


Figure 5–4: Shared Nothing Architecture

5.1.4 Shared Something

Contrary to what supporters of shared nothing are saying, several people [65,66,93] have suggested that pure shared nothing systems are not the best way forward, but that hybrid architectures with some aspects of sharing are to be favoured. Their claim is supported by current trends in the parallel database systems market: currently very few manufacturers actually produce pure shared-nothing systems.

The motivation behind hybrid architectures is to combine the advantages of the architectures described above, i.e. to achieve good scalability and availability without losing the advantages of sharing. Valduriez [93] coined the term *shared something*. In his definition, a shared something system consists of a number of shared-memory nodes (with more than one processor each) connected with each other through some interconnection network. Disks are attached to the interconnect and are shared by all nodes. Alternatively, instead of connecting the disks to the interconnect, each shared memory node may have its own set of disks attached to it. The Convex Exemplar is an example of such an architecture (more detail below).

Figure 5–5 shows the structure of a shared something architecture. We have drawn the disks attached to shared memory nodes in dashed lines to indicate that it is not likely that there are disks attached to each node in addition to the shared disks connected to the interconnect. As explained above, we consider these to be alternative versions of a shared something system.

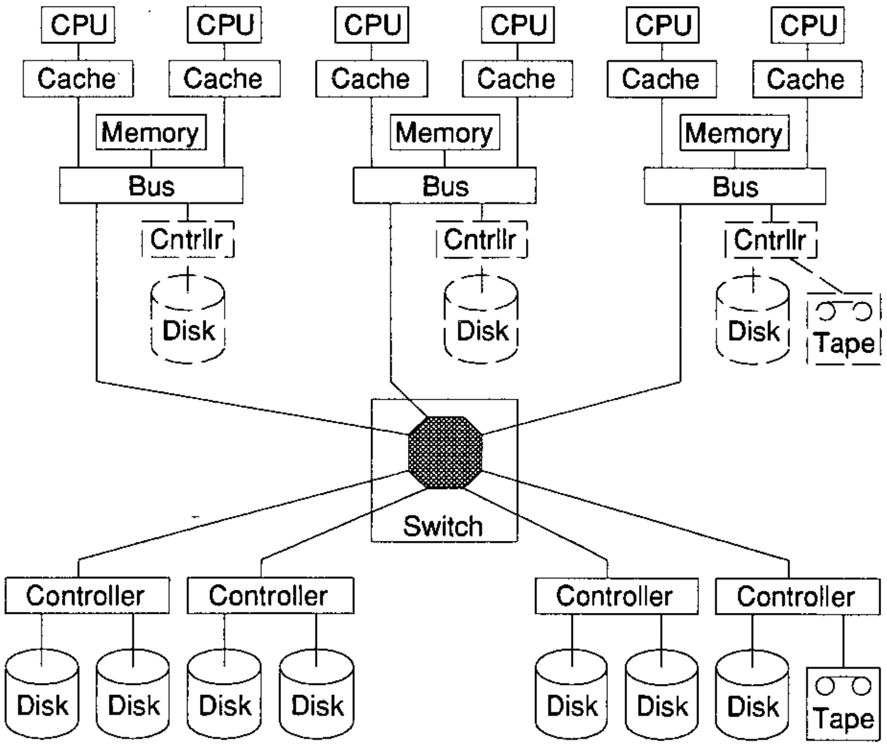


Figure 5–5: Shared Something Architecture

5.1.5 Example: Convex Exemplar

Convex Computer Corporation (which has been owned by Hewlett Packard since 1995) produces a computer, the Exemplar, which we will look at in more detail. It will serve as an example for a modern architecture for parallel database systems and we will map the software architecture of a branching transaction system to this machine architecture.

Following the above categorisation, the Exemplar is a shared something architecture where disks are attached to shared memory nodes, rather than the interconnect. Each node consists of up to eight PA 7100 or PA 7200 processors from Hewlett Packard. Each processor has a local cache (processor cache). Processors in one node share one cache coherent, global memory. Instead of the traditional approach of using a bus, processors and memory are connected via a 5 by 5 crossbar switch for better performance. Up to 16 shared memory nodes may be connected by the Convex Toroidal Interconnect (CTI). CTI has a ring topology. Each node may have up to 105 disks (4GB each) connected — via some I/O adapter — to it. Figure 5–6 shows the architecture of the Convex Exemplar.

A key feature of the Exemplar is its support of one single global address space across all nodes. The system maintains memory coherence between the nodes. Hence, logically it can be treated as one large shared memory system, although currently its total address space is limited to 4GBytes. Memory access across the CTI is slower than within a single node. To reduce this added memory access latency, a part of the local node memory is used as "node cache" for memory which resides on a different node.

5.2 Parallel DBMS Architecture Classification

Although the architecture of a parallel DBMS is to some extent dependent on the architecture of the parallel computer it is running on, different versions of the same DBMS may run on different hardware platforms. Hence, instead of

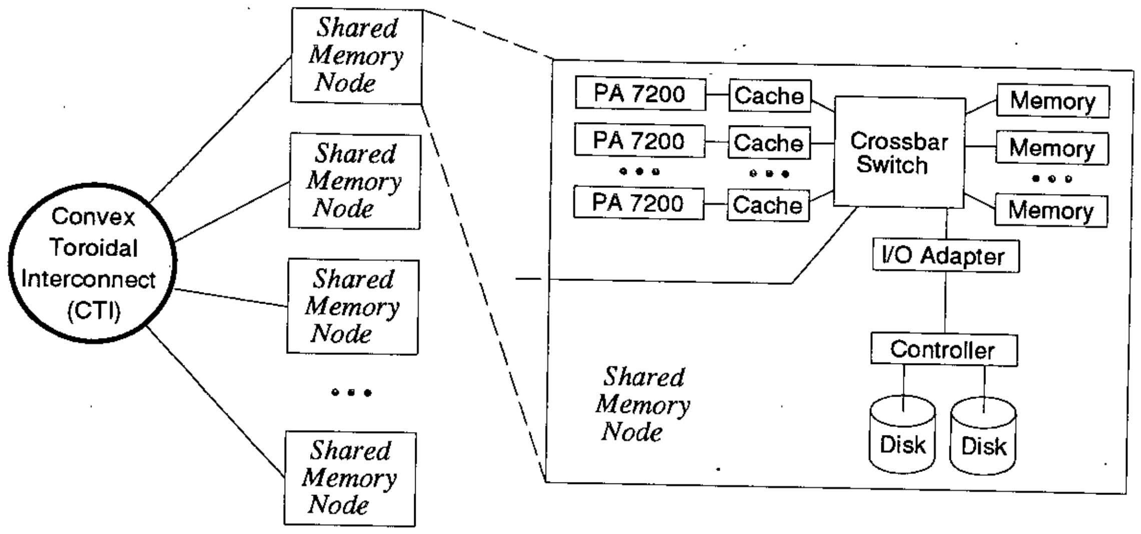


Figure 5–6: Convex Exemplar Architecture

classifying parallel DBMS based on the hardware platform they are running on, Norman and Thanisch [65] have suggested a comparison based on a 5 layer model of the DBMS software architecture.

We discuss this 5 layer model here, since later in the chapter we will map the architecture of a parallel DBMS using branching transactions onto this model. This mapping will help us to discuss aspects of cache coherence and load balancing. In this section, we introduce their 5 layer model, first in general terms and then by using the example of the DBMS Oracle7 [67]. We use Oracle7, since it is very flexible and illustrates many aspects of a modern parallel DBMS, such as for example load balancing and query parallelism.

The 5 layers of this model are: *dispatchers*, *servers*, *slaves*, *disk accessors* and *disks*; each representing DBMS components at a different level of abstraction. The dispatchers communicate with the users. They accept query requests and pass them on to servers. A server either executes a query itself or breaks it down into sub-queries which are dispatched to slaves. The slaves execute sub-queries and use disk accessors for access to the database. At the bottom of these layers are the actual disks which contain the database (Figure 5–7).

Figure 5–8 shows a possible configuration of the DBMS Oracle 7, running on a machine with two SMP nodes, where each node contains a couple of processors

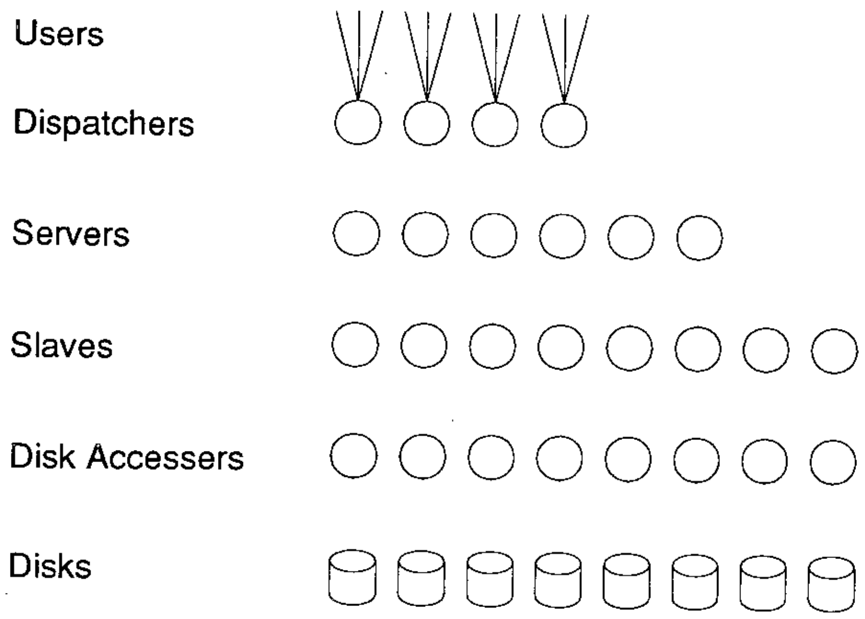


Figure 5–7: General 5 Layer Model

and some shared memory. No memory is shared between nodes and each disk can be accessed by each node. In the above hardware classification, this corresponds to a shared-something architecture (with shared disks).

We assume that Oracle's multi-threaded server is running, so that users can connect to dispatchers, which distribute queries from the users to server processes[3]. A query may be processed sequentially by one server process (running on one node) or, using Oracle Parallel Query, the query may be broken down into several sub-queries which are then executed by query slaves. A sub-query may be executed on a node different to the one from which it originated. Disk access is performed by disk accessors — database writers, in Oracle's terminology — which can access all disks in the system (shared disks).

Having discussed a hardware and a software oriented model for parallel database systems in general, the next section looks at possible system architectures for branching transactions in particular. The discussion focusses on what has previously been described as shared-memory and shared-something environments and

[3]In Oracle 7, it is also possible for a connection to be established which bypasses the dispatchers layer.

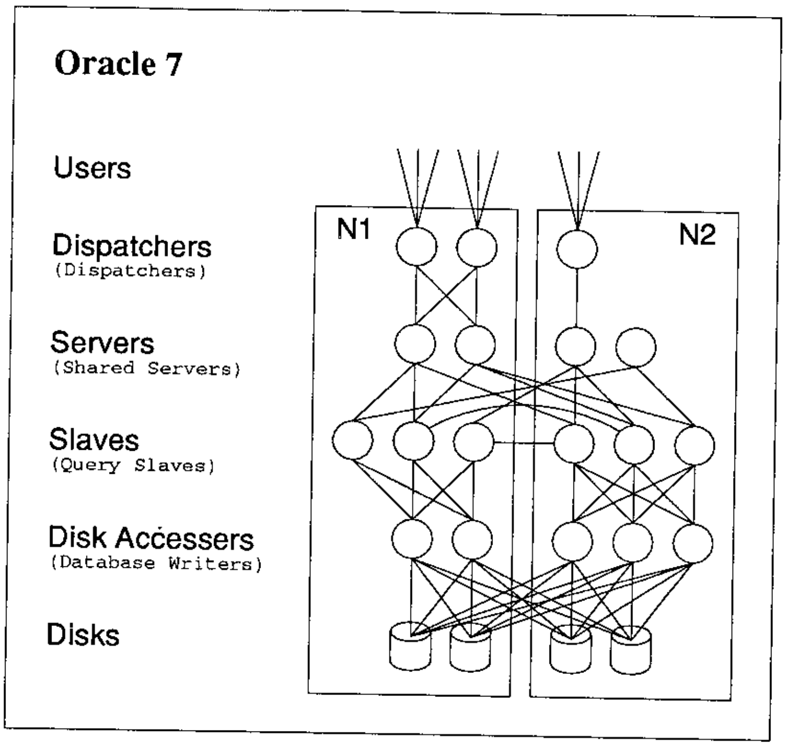


Figure 5–8: 5 Layer Model for Oracle 7

it uses the above five layer DBMS architecture model to explain such issues as load balancing and cache coherency.

5.3 Branching Transaction System Architecture

The architecture of a system describes the overall decomposition of the system into its primary functional components (subsystems). In the context of transaction processing (TP) systems, these components are often referred to as resource managers; a resource manager is a subsystem of the TP system, offering particular services to applications or other resource managers. For example, Gray and Reuter [40] describe a *transaction processing monitor* as the combined functionality of a *transaction processing operating system (TPOS)*, e.g. transactional remote procedure calls, transaction identifiers, lock management, log management, transaction commit processing, and authentication; and the functionality of *transaction processing services (TRAPS)*, e.g. operations interfaces, monitoring, load balancing, security mechanisms and configuration management.

In this section, we give an overview of resource managers which are required to run a branching transaction processing system. This division into a number of resource managers is primarily functional and does not directly correspond to the implementation of such a system, i.e. not every resource manager needs to be implemented as a separate process[4]. We will first describe the situation for a shared-memory hardware platform and then for shared-something systems.

5.3.1 BTs in a Shared Memory Environment

We identify below resource managers for a branching transaction processing system on a shared memory hardware platform. The same set of resource managers is actually used for other architectures as well, but at this time we only focus on their functionality within a shared memory environment. The overall architecture of the system is shown in Figure 5–9.

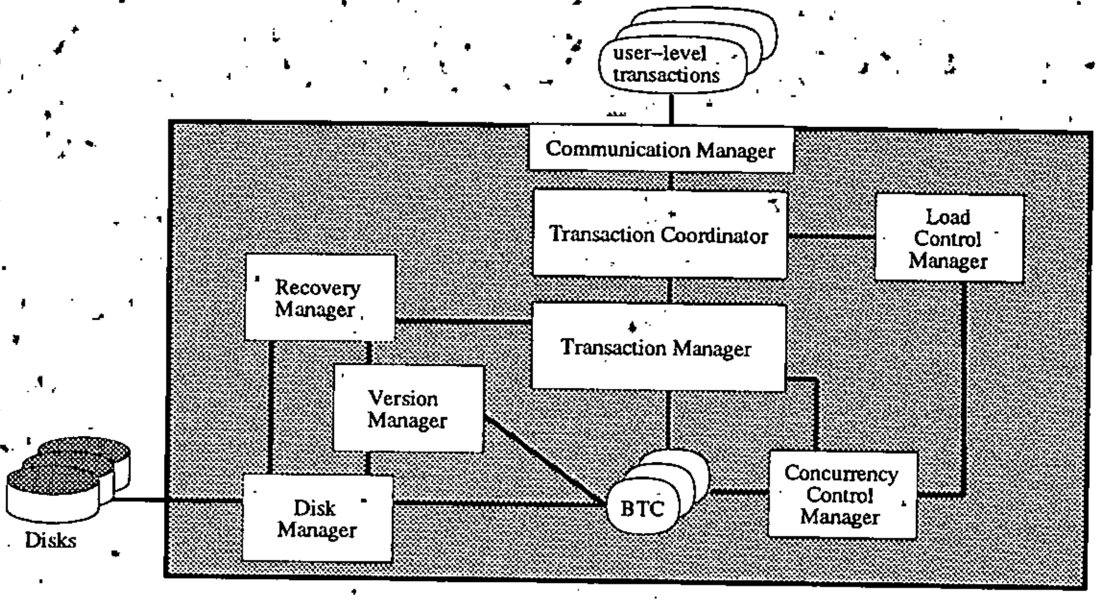


Figure 5–9: BT Architecture for Shared-Memory System

[4]In the Camelot transaction system [34], for example, the communication manager is bundled with the transaction manager for efficiency reasons.

Communication Manager (CM): The CM provides an interface between user-level applications and the transaction coordinator.

Transaction Coordinator (TC): A user submits (via the communication manager) a transaction for execution to the transaction coordinator. The TC queues the transaction if the current system workload is too high or submits it to the transaction manager for execution[5].

Transaction Manager (TM): The TM is responsible for the execution of a transaction. It creates, aborts, and coordinates the commit of the branching transaction components (BTCs) for a transaction.

Load Control Manager (LCM): The LCM maintains resource utilisation data and system performance statistics, which are used to control submission of new transactions to the system and branching of currently running transactions. The LCM's responsibility is to prevent thrashing behaviour.

Concurrency Control Manager (CCM): The CCM implements the hybrid branching transaction multi-version two-phase locking algorithm described previously. It consists of three subsystems: 1) the *lock manager*: responsible for locking and unlocking of data items, 2) the *deadlock manager*: responsible for deadlock detection and resolution, and 3) the *branch control manager*: responsible for when and how transaction components should branch; it consults the LCM for necessary system statistics.

Version Manager (VM): The VM provides access to uncommitted versions of data objects.

[5]In a shared memory system, the functionality of the transaction coordinator could easily be included within the transaction manager. The TC, however, assumes more functionality in a shared something environment.

Disk Manager (DM): The DM is responsible for disk space management. It stores and retrieves data objects and log records on hard disks. It also manages the database cache in main memory.

Recovery Manager (RM): The RM facilitates transaction abort and recovery, as well as recovery from system and media failure.

To get a better understanding of the role of these resource managers and their interaction, we now discuss the execution of a transaction in such an environment in more detail.

When a user-level transaction is submitted to the *transaction coordinator* for execution, the TC consults the *load control manager* whether enough resources are available to execute the new transaction. If not, the new transaction is queued within the TC, waiting to be scheduled at some later time. If yes, the TC passes the transaction to the *transaction manager* for execution. Upon receipt of a new transaction, the TM creates a new branching transaction component (BTC) and schedules it for execution.

BTCs request Read/Write locks from the *concurrency control manager*. If a lock request is granted for a Read operation, the BTC either requests the item from the VM (if it wants to read uncommitted data) or the DM (if it wants to read committed data). Whether the item to be read is committed or not, is indicated in the Grant message from the CCM. A Write request is always directed at the VM, which creates a new version of the corresponding item. If a Read request is made to the DM, and the requested item is not in main memory yet, it is fetched from disk by the DM. If the CCM decides — after consultation with the LCM — to branch an existing BTC, it informs the TM, which creates the required, new BTCs.

The commit of a branching transaction's components is coordinated by the TM. Once an entire path has been certified, all its BTCs must be committed. The TM sends appropriate commit instructions to the *recovery manager*. The RM instructs the VM to copy the newly committed versions to the DM and removes them from

VM's memory. The DM then writes log records of updated data to the log disk and flushes updated items to the database disks. The TM also instructs the RM to abort all BTCs (of that transaction) which are not part of the committing path. Since all updates to the database are deferred, i.e. no data is written to disk until commit time, to abort a BTC the RM simply informs the VM to discard any data versions which were created by the aborting BTC. The RM informs the TM once the abort of a transaction is complete, at which time the TM instructs the *concurrency control manager* to release all locks which were held by that BTC. Similarly, the RM informs the TM about the completion of a BTC commit. After *all* — we apply strict two-phase locking — BTCs of a path have been committed by the RM, the TM informs the CCM to release their locks. The successful completion of a transaction is communicated back to the user via the TC and CM.

5.3.2 BTs in a Shared Something Environment

As described earlier, under *shared something* we assume that the system consists of a number of shared memory nodes (where each node contains more than one processor), that nodes are connected with each other through some interconnect hardware, and that transactions on all nodes can access data on all disks (whether these disks are shared, or some inter-node communication needs to take place).

The architecture of a branching transaction system in such an environment is very similar to the one described above. In fact, there is no need to introduce any additional resource managers, though the functionality of existing ones must be extended. Furthermore, we designate one node to be the *central system node* (CSN), and all others as *general transaction processing nodes.* The CSN consists of a communication manager, the transaction coordinator, the concurrency control manager and the load control manager. All other (general processing) nodes consist of a communication manager, a transaction manager, a version manager, a recovery manager and a disk manager.

A transaction is submitted to the CSN, where the TC either queues it for later execution (to avoid system thrashing) or sends it to the TM of one of the general processing nodes. A TM is responsible for the execution of transactions on its node. Its basic functionality is the same as it was for the shared memory case. Newly created BTCs remain on the same node as their parent node. Sibling BTCs, however, are likely to be executed by different processors within that node. Lock requests by BTCs are sent across the interconnect to the CCM on the CSN. Again BTCs request access to data items through either the VM or the DM; the fact that items may be located at other nodes or disks on other nodes is hidden from BTCs. If an uncommitted version of an item is located on a different node, it must be fetched from there. Similarly, a disk manager may need to access data on any of the disks within the system. This data transfer between VMs and DMs across the system raises issues of data buffering (caching) and, therefore, the problem of cache coherence must be addressed. We will discuss it in more detail in Section 5.5 below.

The overall architecture of a branching transaction system is largely independent of how disks are shared, i.e. whether disks are attached to nodes, as in Figure 5–10, or whether they are directly connected to the interconnect (shared disks), as in Figure 5–11. The specifics of these aspects are hidden within the disk managers.

There are two important aspects which need further discussion: 1) cache coherence, and 2) submission of transactions from the CSN to general processing nodes (load balancing). The following two sections deal with these aspects in more detail.

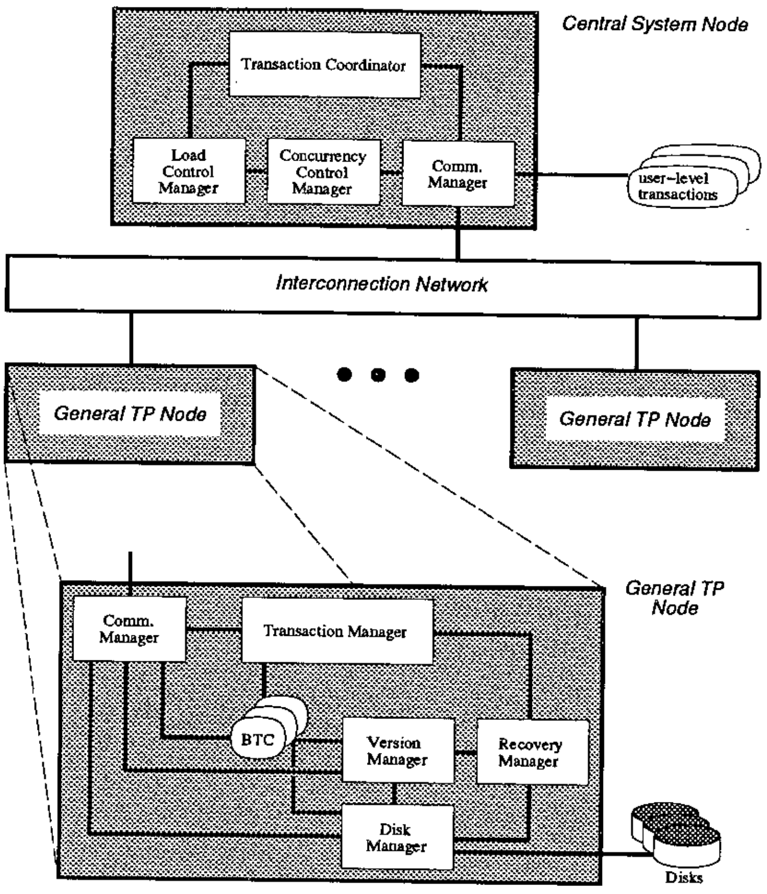


Figure 5–10: BT Architecture for Shared-Something System (Version 1)

5.4 Load Balancing and Load Control

An uneven distribution of workload between nodes and within nodes leads to poor resource utilisation and, therefore, to low performance. We apply a load balancing strategy to address this issue. To have a better basis for this discussion, we map the architecture of a branching transaction system in a shared something environment onto the 5 layer model of parallel DBMS introduced earlier.

As shown in Figure 5–12, the transaction coordinator on the CSN assumes the role of the *dispatcher*. It sends a transaction to one of the transaction managers (*servers*, in the general model terminology). BTCs correspond to *slaves*. The recovery manager, version manager and disk manager fall roughly into the level of *disk accessors*. Although our BT architecture does not match the 5 layer model

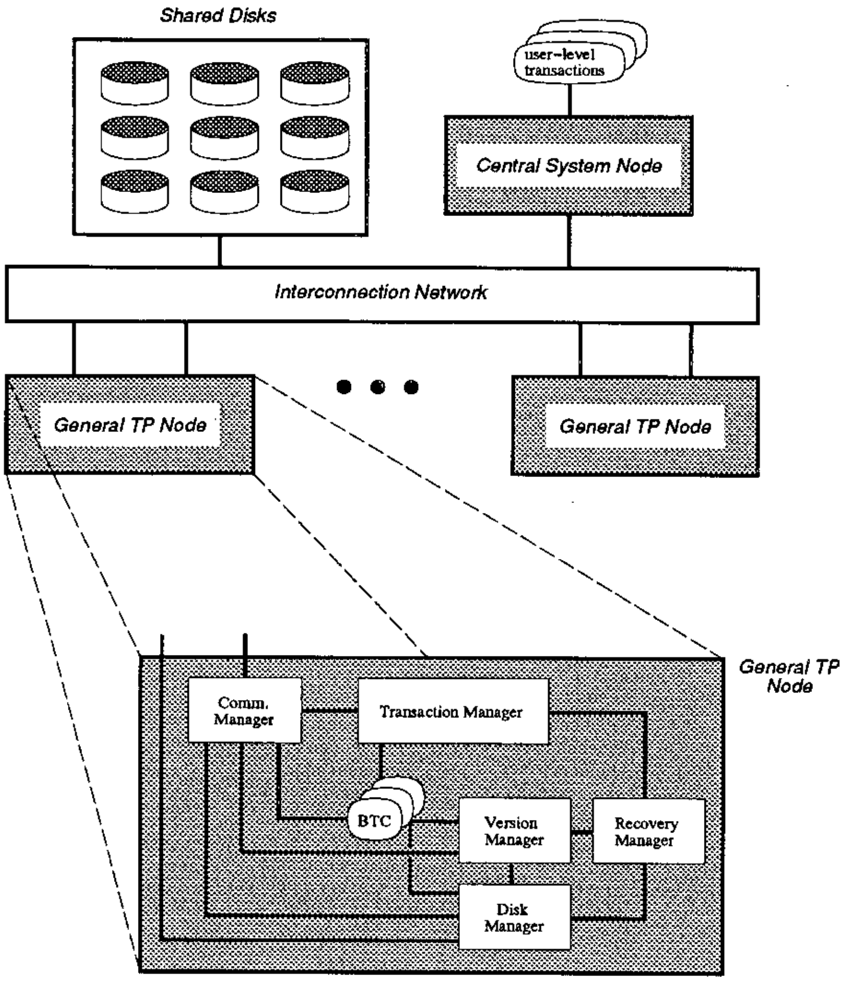


Figure 5–11: BT Architecture for Shared-Something System (Version 2)

perfectly, it clearly illustrates where load balancing can be applied: 1) by TC when allocating transactions to TMs, and 2) by TM when scheduling BTCs on local processors.

Scheduling BTCs on a shared memory node is relatively easy, since all processors share the same memory. In fact, load balancing within a node needs little interference from the DBMS and is left to the underlying operating system.

A more difficult task is to determine a good scheduling strategy for the TC. The basic rule is that the TC allocates a transaction to the TM on the node with the lowest average CPU utilisation at the time. This may, however, not always be the best solution. Another factor that needs to be considered is *data locality.* If it is known in advance which data items a transaction wants to access, it may

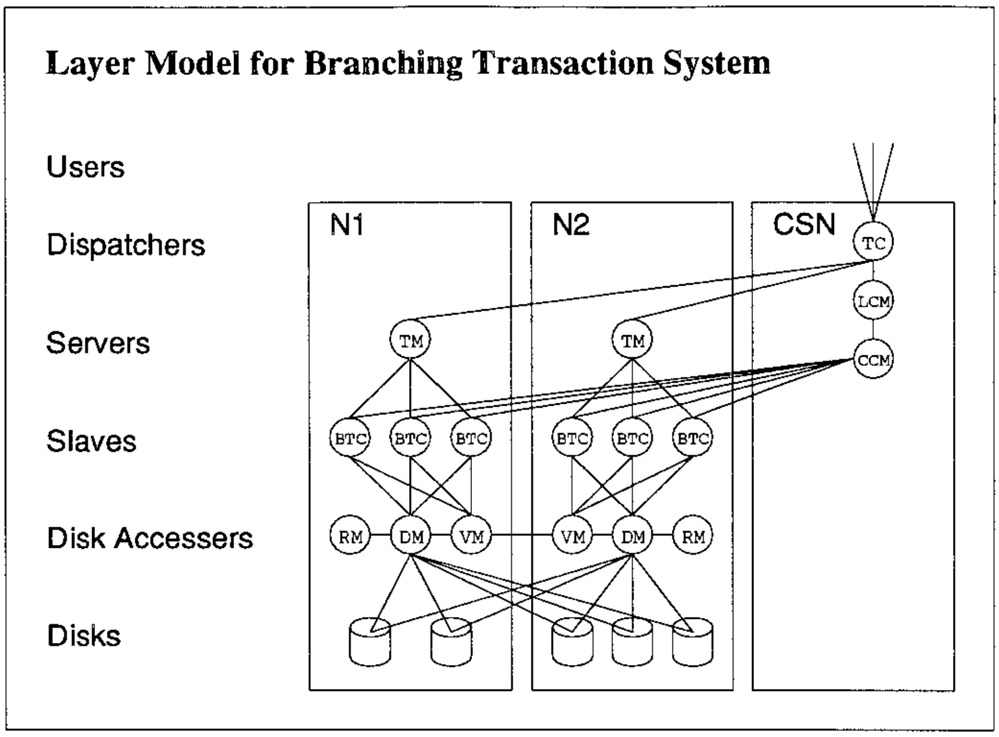


Figure 5–12: Layer Model for Branching Transaction System

be beneficial to execute that transaction on a node which will require the least amount of inter-node data transfer. Reducing communication costs and balancing CPU utilisation may become conflicting goals. There is no general answer to this problem. The optimal load balancing strategy depends on the actual system particulars, i.e whether or not its inter-connect is a bottleneck, whether CPU utilisation is relatively low or rather high, what kind of access patterns transactions display, etc. The actual architecture of our system, however, does not depend on which load balancing policy is applied. The details of the policy are encapsulated within the load control manager.

In addition to load balancing, the LCM also determines the load control policy of the system; it is responsible for the prevention of thrashing. Thrashing may happen for two reasons: 1) too many transaction are allowed into the system, and 2) too much branching of transactions leads to too many BTCs.

The most common approach to restrict the number of transactions in a system is to define a *multi-programming level* (MPL). The MPL is the maximum number of active transactions allowed in the system. Once that level has been reached, any transaction sent to TC is held in a waiting queue until one of the active transaction completes its execution. Although this is a very effective way of preventing thrashing, it has been criticised for its static decision process; the MPL approach does not take into consideration whether or not thrashing actually occurs. Several researchers [20,61,62] have proposed more dynamic methods which use current system statistics to decide whether a transaction's execution should be delayed. The LCM may apply any appropriate load control policy[6].

We have already described the need of a branch-control function in Section 3.3.3. Such a control function should be implemented by the LCM. The CCM consults the LCM before making a branching decision. Branching may be rejected by the LCM if thrashing becomes a problem.

5.5 Cache Coherence

To speed-up access to data, cache memory is used in several places of the memory hierarchy within a computer system (see Figure 5–13). Main memory data is buffered within the processor cache for faster CPU access. Database data is usually buffered within the main memory to reduce disk I/O, and the disk device itself may make use of a disk cache (on-board cache) to reduce delays due to disk rotation and read/write head movements.

A problem of cache coherence exists if more than one cache buffers data of the same memory entity. For example, if two processor caches contain a copy of the same main memory page and one is changing its value, then the copy in the other processor cache is out-of-date, and we have an inconsistency. Similarly, if the same database item was read from disk into two different database caches, as

[6]Since the work in [20,61,62] is based on flat transactions, it is not clear how these techniques perform for branching transactions. More performance studies would be required to evaluate their suitability.

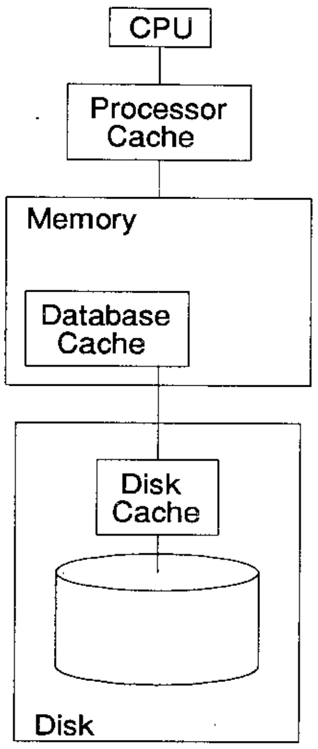


Figure 5–13: Memory Hierarchy

soon as an update takes place at one of the two database caches, the other one is out-of-date.

Each processor in a shared memory node accesses the same "node memory", and since each processor makes use of a processor cache, some form of cache coherence control must be applied. As we explained earlier (Section 5.1.1), this control is usually carried out by some hardware mechanism, and there is no need for the DBMS to interfere with it. Hence, in this section we are only concerned about the problem of inconsistencies between database caches on different shared-memory nodes. (From now on, when we discuss cache coherence problems, we only refer to the database caches, unless stated otherwise.)

5.5.1 Cache Coherence in Disk Manager

The database cache in our BT architecture is part of the DM. Hence, each shared memory node maintains its copy of cache memory. Clearly, if two DMs contain a copy of the same item, and one updates it, the other copy becomes invalid. To deal with this problem, we must apply some cache coherence protocol. For any such protocol, there are two fundamental problems: 1) invalidation of out-of-date data in the cache, and 2) propagation of newly updated items.

Following the terminology used by Rahm [75], our cache coherence scheme could be classified as using *on-request invalidation*, *selective notification*, some sort of *page sequence numbers*, *horizontal propagation* for cache page updates, and *force* or *no-force* disk update. Each of these is discussed below.

On-request invalidation, which is also referred to as "check-on-access", can be used together with locking protocols. The basic idea is to keep database cache information together with the lock tables, so that the validity of a cache page can be checked during the processing of a lock request, thus eliminating extra communication overhead for separate cache invalidation messages. Hence, it is the task of the concurrency control manager to determine whether the cache at the node from where the lock request was sent has an up-to-date copy of the requested item. For this, we use a technique similar to what Rahm refers to as "page sequence numbers".

"Page sequence numbers" are attached to pages. The CCM keeps track of which is the most up-to-date sequence number for any given page. Since the CCM also knows the sequence numbers of the cached pages at each node, it can easily decide whether or not a cache is up-to-date. In our case, the page sequence numbers are the indices of the BTC which created the page, i.e. a page created by $BTC_{2,3}$ has the "sequence number" $(2,3)$. If a cache does not hold the requested data, it is updated, using a *horizontal propagation* policy.

Horizontal Propagation describes a cache update scheme where the most up-to-date copy of a page is forwarded to a cache which needs access but does not have a copy of it. In the case of a BT system, the CCM finds out that a request from one node, say N_1, needs to read x, but that the latest version of x is only available from the cache at another node, say N_2. In this case, the CCM sends a "forward request" to the DM of N_2, which in turn sends its (up-to-date) copy of x to N_1's DM; x has been (horizontally) propagated from one node's cache to another.

Force and NoForce Disk Update: Earlier in this chapter we said that database updates made by a transaction are flushed to disk before that transaction is allowed to commit. This technique is referred to as the *force* approach. Rahm [75] points out that this may not be suitable for high performance transaction processing since its high I/O overhead causes significant response time delays for update transactions. To avoid this problem, the *noforce* approach only ensures that all relevant redo log records are written to disk before commit time, and that updates to the database can take place after the commit. To prevent transactions reading out-of-date data from disk, the horizontal propagation scheme described above must be used to allow each DM to have access to the latest version of an item. Since we use a horizontal propagation scheme already, it would only require minor changes to our DM to switch from a force to a noforce disk update strategy.

5.5.2 Cache Coherence in Version Manager

The version managers at each node are faced with a problem similar to the one faced by the disk managers: they may need to access a version of an item which only exists at a remote node's VM. The solution is the same as for DMs: the CCM maintains information about the existence of versions within its locktables, and the same kind of horizontal propagation scheme is used to update a VM memory. It is important to remember that VMs are only concerned with uncommitted data, and hence, there is no issue of disk updates. What happens at commit time, i.e. when uncommitted data become committed data, and how the version managers interact with disk managers is discussed below.

5.5.3 Combined Disk Manager and Version Manager Cache Coherence

A transaction requests read access to a data item, but not a particular version of an item. It is the CCM which decides which version of the requested item should be read by the requesting BTC. This version information is tagged to the Grant reply from the CCM to the BTC. The Grant message includes two other pieces of information:

Commit Flag: indicates whether the specified version of the item should be requested from the local VM (if flag is not set) or the local DM (if flag is set);

Forward Flag: indicates whether the specified version will be forwarded from a remote node to the local VM/DM;

When a BTC receives a Grant message after a Read lock request, it first checks the Commit flag. Depending on the value of the flag, it then makes a read request to either VM or DM. In either case, the Forward flag is attached to the read request. Four possible scenarios exist:

Read request to VM, Forward flag not set: the requested data item version should be in the local VM, and a copy is passed to the requesting BTC;

Read request to VM, Forward flag set: the requested data item version has to be forwarded from a remote VM. The local VM has to wait for the arrival of the (horizontally) propagating version, before it can satisfy the read request;

Read request to DM, Forward flag not set: the requested data item version should be in the local DM, and a copy is passed to the requesting BTC;

Read request to DM, Forward flag set: the requested data item version has to be forwarded from a remote DM. The local DM has to wait for the arrival of the (horizontally) propagating version before it can satisfy the read request;

It may happen that none of the DMs has a copy of the most recently committed version of the requested item; it only exists on disk. In this case, the DM needs to fetch it from disk.

When a transaction commits, all its updates change from non-committed to committed status, and hence, they need to be moved from VM to DM. This is not only true for the node where the transaction was executed, but also for those nodes to where a copy of the item version has propagated. Hence, the CCM has to send Commit messages to all nodes where copies of items exist which were created by the committing transaction.

To illustrate some aspects of the overall cache management mechanism in a branching transaction DBMS running on a shared something (or shared disk) platform we use the following example.

Example 7 *Assume that we have a system with 3 nodes, where each node is a shared memory system. One node is the designated CSN, the other two are general processing nodes. A copy of data item x (x_{00}) is on disk; no copy exists in any of the caches. We are assuming a Force disk update approach. Figure 5–14, part a) shows the initial cache situation.*

Now BTC_{11}, which executes at node N_2, wants to read x, and a copy of x_{00} is read into N_2's DM (Figure 5–14, part b). Next BTC_{21}, which executes at node N_2, also wants to read x, and a copy of x_{00} is propagated from N_2's DM to N_1's DM. A Write request of BTC_{11} at N_2 leads to a new version of x (x_{11}), which is held in N_2's VM (Figure 5–14, part c). After the Commit of BTC_{11}, x_{11} becomes a committed data item, i.e. it is copied to N_2's DM and removed from the VM there.

Copies of x_{00} are eventually purged from all DMs, since it is no longer the last committed version of x, so no transaction will ever be allowed access to it again (Remember, we only keep the most recent of all committed versions of an item.)

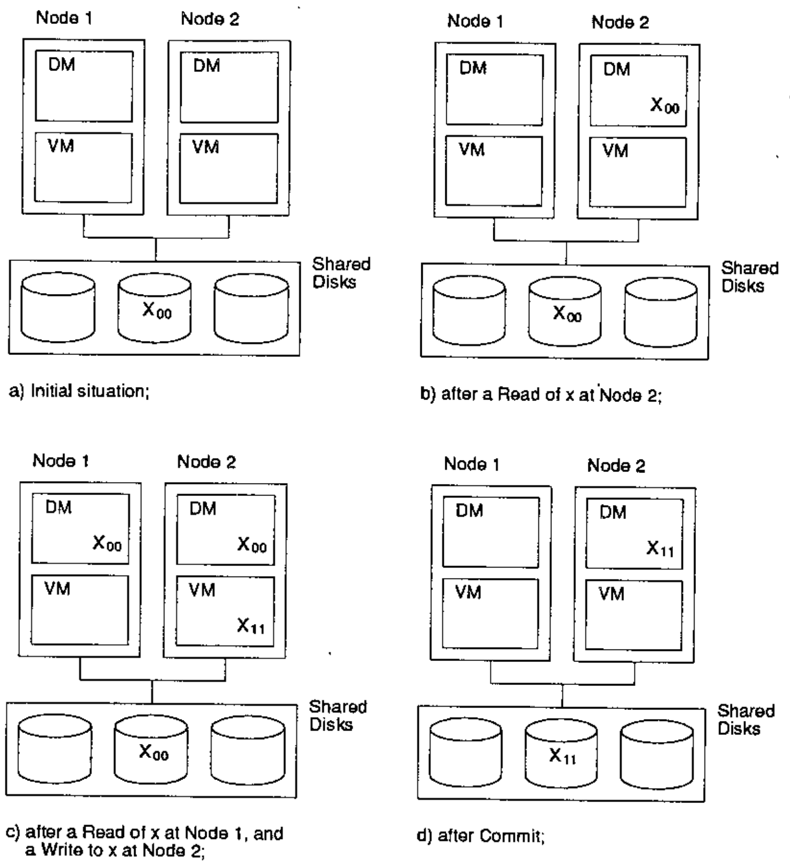

Node 1
Node 2
DM
VM
Shared Disks
X00
X11
a) Initial situation;
b) after a Read of x at Node 2;
c) after a Read of x at Node 1, and a Write to x at Node 2;
d) after Commit;

Figure 5–14: Cache Management Example

To avoid additional communication overhead, the instruction from CCM to DMs to discard x_{00} can be "piggy-backed" whenever the CCM has to send a message to the corresponding nodes next time. There is no need to immediately remove these copies from the DMs, since the CCM will not allow access to it in any case. Finally, we will be left with the updated version of x on disk and in N_2's DM (Figure 5–14, part d).

5.6 Centralised vs. Distributed Lock Management

It is easier to implement a centralised concurrency control/lock manager rather than a distributed one, but centralisation creates several potential problems:

communication overhead: each lock request involves the sending of two messages across the node interconnect: one to the CSN (Central System Node) for the lock request and one for the reply;

system bottleneck: the progress of all transactions depends on how fast the CSN can reply to their lock requests and, hence, an overloaded CSN can become a system performance bottleneck;

single point of failure: a failure of the CSN would lead to a total system failure, since none of the other nodes is designed to take over the functionality of the CSN;

To avoid these problems, the lock manager can be distributed over several nodes. Distributed concurrency control has been studied extensively ([10,24,70]). Most of the algorithms proposed in this field, however, are targeted at distributed databases, where the database is partitioned over several nodes and in general a transaction at one node does not have access to data at a remote node (unless it spawns a sub-transaction at that node). As a consequence, the performance and behaviour of distributed lock management is closely linked to how the database has been partitioned.

Since we are assuming that all transactions, no matter on which node they are executed, have access to all database disks, the distribution of lock management is less restricted. We can either use dedicated lock management nodes or assign lock management responsibilities to general processing nodes. In either case, the lock space is partitioned using some hashing function. This hashing function must be known to all nodes, so they can send their lock requests to the right lock manager. Although this scheme still suffers from a high communication overhead, it reduces the problem of a system bottleneck and achieves a higher level of fault tolerance. It is, however, important to find a good hashing function to distribute lock processing overhead evenly; it may be necessary to recalibrate this hashing function occasionally to adapt to changing system characteristics (e.g. transaction workload changes). A more flexible approach is to assign lock management responsibilities dynamically and allow them to migrate from one node to another. In this case, though, extra communication overhead is caused by the need to constantly update all nodes about the current lock management distribution.

These issues of distributed lock management are largely the same whether or not branching transactions are involved. We will, therefore, not discuss the various solutions to this problem in more detail here. For a good overview in this area, the reader is referred to Rahm [75].

5.7 Branching Transactions on a Convex Exemplar

We introduced the hardware architecture of the Convex Exemplar earlier in this chapter. In this section, we will briefly outline its system software architecture and how a BT DBMS can be mapped to it.

5.7.1 System Software Architecture of a Convex Exemplar

As described earlier, the Convex Exemplar consists of a number of shared memory nodes, called *hypernodes*, which are connected through a Coherent Toroidal Interconnect (CTI). Each hypernode runs a copy of a *microkernel* which provides basic kernel functionality such as virtual memory and scheduling of processors. All other system functions are provided through servers running in user space. The entire operating system, called SPP-UX, is based on the Open Software Foundation OSF/1 AD distributed microkernel, and is binary compatible with HP-UX, Hewlett-Packard's Unix operating system.

To support more efficient and easier management of a potentially large number of resources, e.g. processors and memory, the system can be divided into a number of *sub-complexes.* A sub-complex is allocated processors and memory, possibly from different hypernodes. Each resource can only be allocated to at most one sub-complex, and every user process is executed within one sub-complex, i.e. it can only use the resources which were allocated to that sub-complex.

A *process* consists of one or more *tasks*, which may be executing multiple *threads.* A task is executed on one hypernode; all threads belonging to this task are run on the same hypernode.

5.7.2 Branching Transaction System Architecture on a Convex Exemplar

It is not the purpose of this section to describe an optimal configuration of a Convex Exemplar for branching transactions, but to demonstrate how the elements of a BT architecture can be related to the concepts of the system software architecture of the Exemplar.

For the purpose of running a BT DBMS, the resources of a Convex Exemplar could be divided into the following three sub-complexes (Figure 5–15):

CSN Sub-Complex: runs a transaction coordinator process and a load control manager process and handles all communication with the user;

Distributed Lock Management Sub-Complex: runs the concurrency control manager process, which consists of four tasks, one responsible for each hypernode;

General TP Sub-Complex: contains processes for transaction management, recovery management, version management and disk management;

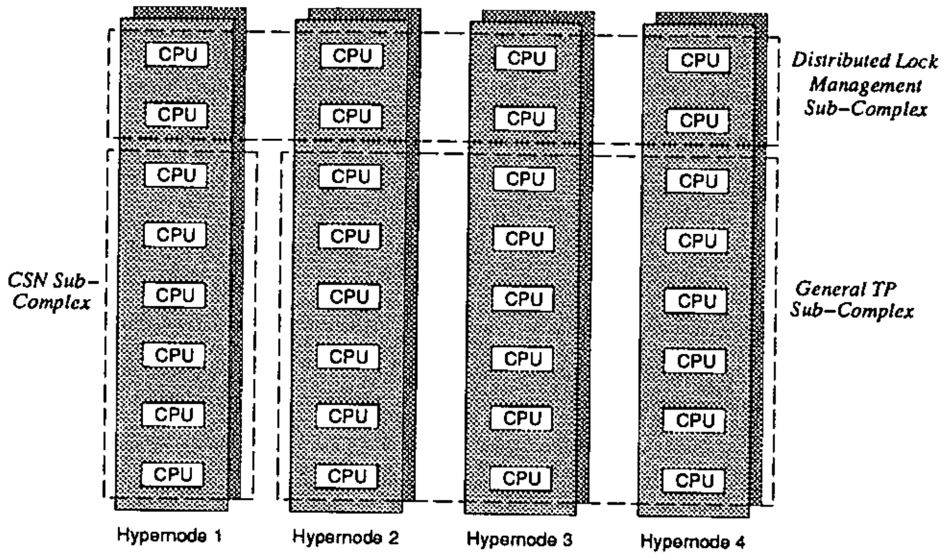


Figure 5–15: BT Configuration of Convex Exemplar

The separation of the resources for lock management and general transaction processing should always allow us to maintain a quick response to lock requests, whether or not the system is heavily loaded at the time of the request. Since the lock management sub-task is spread over more than one hypernode, we are obviously assuming a distributed approach in this example; another measure to achieve quick lock request responses.

The system may run one global transaction manager process which is divided into three tasks (one for each general TP node), or one transaction manager process at each general TP node. In either case, BTCs are executed as threads within the TM. The transaction manager may maintain a pool of threads to reduce the overhead of creating and destroying BTCs, and only create new ones if necessary. BTCs are automatically scheduled by the operating system on any of the local CPUs which are allocated to the general TP sub-complex, and no separate scheduling mechanism needs to be implemented by the BT DBMS. If, however, it is necessary to take some control of scheduling of BTCs — for example, to implement a particular real-time scheduling policy — one can do so by setting priorities of threads as needed.

The BT transaction coordinator runs as a user process in the CSN sub-complex.

The load control manager could be part of this process or a separate one. Also, the recovery managers, version managers and disk managers execute as user processes.

Although SPP-UX provides support for a distributed file system and maintains cache coherence across its file servers, a disk manager of a BT system will need to implement its own disk access, since the cache coherence protocol for branching transactions (as described above) is different from the one supported by SPP-UX.

The implementation of communication services is largely dependent on which programming model one would adapt; the Exemplar provides support for several levels of memory sharing and/or message passing communication.

Figure 5–16 describes the layer model for a branching transaction system on a Convex Exemplar configuration as shown in Figure 5–15. The model is in principle — the example here uses 4 instead of 3 nodes — identical to the one shown in Figure 5–12, except for concurrency control management.

In the previous case (Figure 5–12) a centralised concurrency control manager was used, which was located at the CSN. To avoid some of the problems associated with centralised concurrency control, the CCM has been distributed across all nodes in this example. Whenever a BTC needs to make a lock request, it submits it to its local CCM. If the local CCM is not responsible for that particular data item, it forwards the request to one of the other CCMs in the system. Although the requesting BTC is located on the same hypernode as its local CCM, they do not compete for resources, since they are part of different sub-complexes (each with its own pool of resources).

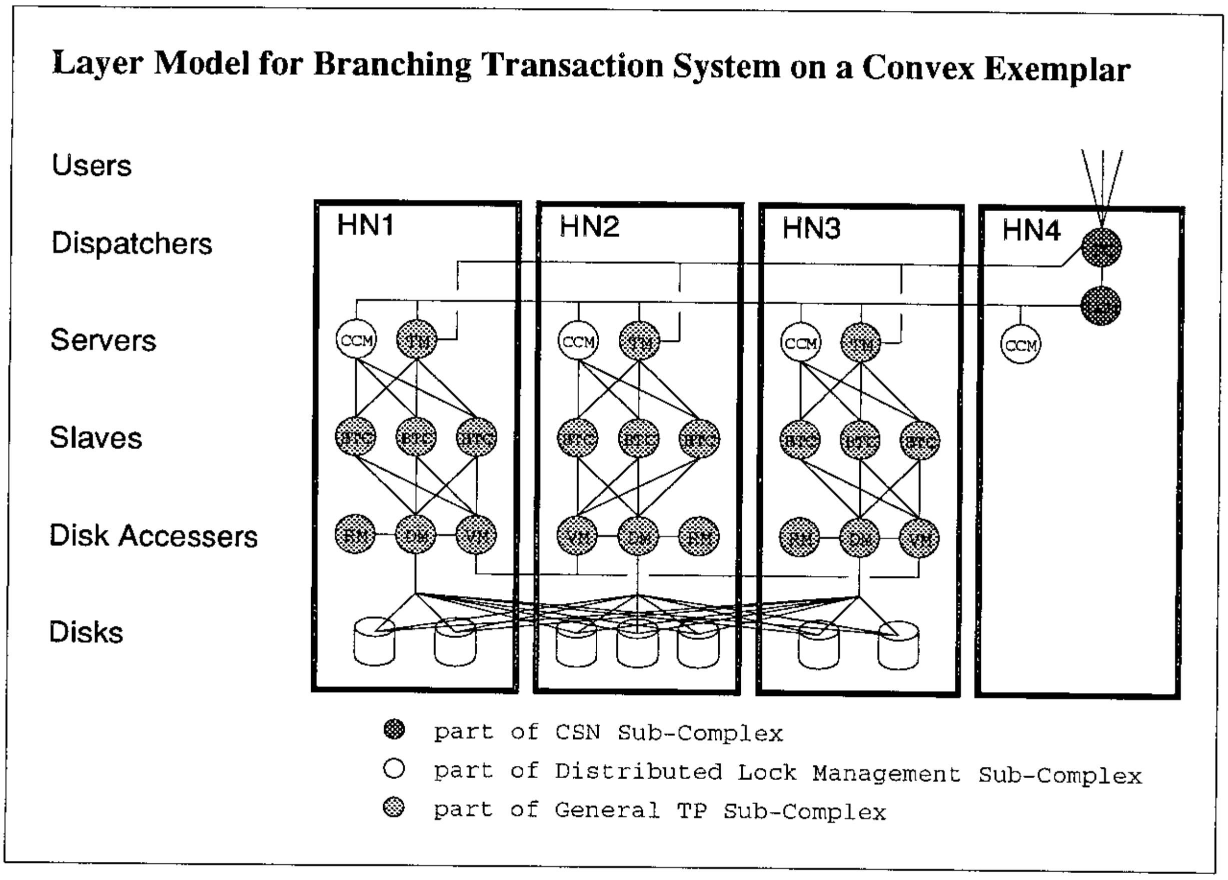

Layer Model for Branching Transaction System on a Convex Exemplar
Users
Dispatchers
Servers
Slaves
Disk Accessers
Disks
HN1
HN2
HN3
HN4
CCM
TM
BTC
RM
DM
VM
part of CSN Sub-Complex
part of Distributed Lock Management Sub-Complex
part of General TP Sub-Complex

Figure 5–16: Layer Model for Branching Transaction System on a Convex Exemplar

Chapter 6

Correctness of BT Schedulers

6.1 Introduction

6.1.1 Transactions and Serializability

While performance is an important issue for any database system, a certain delay in transaction response time is acceptable. Except for real-time applications, a temporary performance drop of the database can often be tolerated. Not acceptable, however, would be the loss of consistency in the database and wrong transaction results due to an incorrect scheduler mechanism. For any new transaction model and concurrency control algorithm it is, therefore, necessary to show that no concurrent executions of multiple transactions are allowed which would either leave the database inconsistent, or result in a transaction seeing the database in a, possibly temporary, inconsistent state.

To prove the correctness of branching transactions we considered various existing methods for reasoning about transactions. Although all of them are based on the idea of serializability (a detailed discussion of serializability will follow below), they differ in notation, complexity, power, and the proof methods they apply. The most widely used method is traditional serializability theory, as described in Bernstein et al. [10]. Many algorithms have been proved using this approach, including multi-version concurrency control. Traditional serializability, however, does not allow one to reason about transactions with internal structure, such as nested transactions and branching transactions. An alternative method is described by Lynch et al. [59]. Based on their notion of I/O automata, they have proved the correctness of a wide variety of algorithms. Their model is very powerful and allows reasoning about multi-version algorithms as well as transactions with internal structure. The disadvantage of this method is its relative complexity compared to traditional serializability. At this time, it also is not as widely used as the traditional approach. A second alternative is the ACTA framework [26]. ACTA allows one to reason about transactions with internal structure and is less complex to work with than the I/O automata model. It has become increasingly popular in recent years. Although the authors of ACTA have shown how one can model multiple versions of data, at the time of this writing no explicit work has been done on proving correctness of multi-version concurrency control algorithms[1].

We decided to adopt traditional serializability theory and make the necessary extensions to deal with the internal structure of branching transactions. There were two reasons for this: 1) this would allow us to draw on the existing work of correctness of multi-version concurrency control, and 2) the proofs presented in this dissertation are accessible to a wider audience. It should be noted, however, that both the I/O automata model and ACTA, could have been used to reason about branching transactions.

6.1.2 Partially Ordered Sets

Throughout this chapter we will use the mathematical notation of a *partially ordered set* (*poset*). We found that the definitions of posets in database texts [10,59,71] differ somewhat from those in mathematics texts [46,83]. Since proofs

[1]This was confirmed by direct communication with one of the ACTA developers, Dr. K. Ramamritham.

presented in this dissertation are based on the work by Bernstein et al. [10], we will follow their definition of a *poset*:

Definition 6 *A relation R is said to be a* partial ordering *on a set X, if and only if R is an antisymmetric, irreflexive*[2] *and transitive binary relation on X. We call the ordered pair (X, R) a* ***partially ordered set*** *or* ***poset****, for short.*

We sometimes use Hasse diagrams to describe posets. For example, let $X = \{a, b, c\}$ and $R = \{(a,b), (a,c), (b,c)\}$; we can then describe the poset (X, R) with the Hasse diagram shown in Figure 6–1. Also, we usually write aRb instead of $(a, b) \in R$.

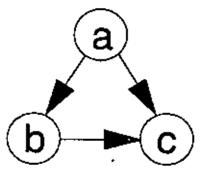


Figure 6–1: Hasse Diagram of Poset

Given a partially ordered set, say $L = \{X, <\}$, we say that $L' = \{X', <'\}$ is a **restriction** of L on domain X' if $X' \subseteq X$ and for all $a, b \in X'$, $a <' b$ if and only if $a < b$. A restriction L' of L is called a **prefix** of L, if for each $a \in X'$, all predecessors of a in L (i.e. all elements $b \in X$ such that $b < a$) are also in X'.

[2]In mathematics texts, the binary relation of a poset is often defined to be reflexive rather than irreflexive.

6.2 Classical Serializability Theory

The correctness proofs presented in this chapter are based on "classical serializability theory", a theory which has its origins in work by Gray et al. [42] and Eswaran et al. [35]. A comprehensive treatment of it is given by Bernstein et al. [10] in their book *Concurrency Control and Recovery in Database Systems.* To keep this dissertation as self-contained as possible, a summary of classical serializability is given next. Definitions, propositions and theorems included in this summary are taken from Bernstein et al. For the convenience of the reader who wishes to study their material in more depth, we give the details of where these definitions and theorems appear in the book. With a few exceptions, the proofs of their theorems are not repeated here. The interested reader is referred to the book.

6.2.1 Transactions and Histories

The scheduler of a database management system tries to make efficient use of hardware resources, such as CPUs and disks, by interleaving the execution of multiple transactions. This concurrent execution of transactions may, however, lead to inconsistencies in the database, due to interferences between transactions. It is the responsibility of the scheduler to prevent such incorrectly interleaved executions.

It is assumed that a transaction, if executed in isolation, does not corrupt the database and produces correct results. It follows that executing multiple transactions in serial order, i.e. each transaction is executed entirely before the next one is started, also produces correct results and the database remains consistent. To prove the correctness of an interleaved execution of a set of transactions, it is sufficient to show that the interleaved execution is equivalent to some serial execution of the same transactions. This is the basic idea of serializability theory.

The classical serializability theory model of a transaction represents a particular execution of a program by describing the Read and Write operations which were executed by the transaction as well as some ordering of these operations. It is assumed that no transaction reads or writes an item more than once. Serializability theory does not depend on this assumption, but it keeps the notation simpler. In addition to Reads and Writes, Commit and Abort operations are used to specify whether a transaction committed or aborted. For each Read and Write operation, the name, but not the value, of the data item involved is given. In addition, various other aspects of the execution, such as the initial state of the database and assignments and conditional statements, are not specified. The analysis of a scheduler must be independent of these so-called *uninterpreted* features, i.e. it must hold for all possible initial states of the database and for all possible executions of a program. Formally, a transaction is defined as follows:

Definition 7 *A* ***transaction*** T_i *is a partially ordered set* $(OP_i, <_i)$, *where*

1. $OP_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$;

2. $a_i \in OP_i$ *iff* $c_i \notin OP_i$;

3. *if* t *is* c_i *or* a_i *(which ever is in* OP_i*), for any other operation* $p \in OP_i$, $p <_i t$;

4. *if* $r_i[x], w_i[x] \in OP_i$, *then either* $r_i[x] <_i w_i[x]$ *or* $w_i[x] <_i r_i[x]$.

(Bernstein et al. [10], page 27.)

Condition (1) describes the type of operations which can be executed by a transaction. Condition (2) says that a transaction either commits or aborts, but not both. Condition (3) requires that the Abort or Commit operation (whichever is in OP_i) is the last operation of a transaction. Condition (4) says that if a transaction reads as well as writes a data item, then these two operations have to be ordered in some way.

Note that operations c_i and a_i model the actual event of transaction commits and aborts, respectively. This is not to be confused with commit and abort statements in the program of which the transaction is a particular execution. Although an abort statement will always lead to the actual abort of a transaction, the execution of a commit statement may trigger a transaction abort instead of a commit; e.g. the certification of a transaction in an optimistic concurrency control environment may fail or a system failure may prevent a server from being able to commit a transaction. Similarly, $r_i[x]$ and $w_i[x]$ describe the events of reading from and writing to the database. Again, the statements in a program which cause a read or write attempt may actually lead to transaction aborts; the corresponding lock request of a transaction may precipitate a deadlock which needs to be resolved via a transaction abort.

Two transactions potentially interfere with each other through Read and Write operations on shared data items; Read operations do not interfere with each other. Hence, if two transactions access the same data item, and at least one writes the item, then an order on these two operations must be specified in the corresponding history; two such operations are said to *conflict.* Furthermore, any ordering on operations specified by a transaction must also hold in a history which involves that transaction.

Definition 8 *Let* $T = \{T_1, T_2, \ldots, T_n\}$ *be a set of transactions. A* ***complete history*** H *over* T *is a partially ordered set* $(OP_H, <_H)$*, where*

1. $H = \bigcup_{i=1}^{n} OP_i$*,*

2. $<_H \subseteq \bigcup_{i=1}^{n} <_i$*,*

3. *for any two conflicting operations* $p, q \in OP_H$*, either* $p <_H q$ *or* $q <_H p$*.*

(Bernstein et al. [10], page 29.)

Condition (1) requires that all operations executed by transactions $T_1, \ldots, T_n$ are included in the history, and that no other operations are involved. Condition (2) says that any ordering stipulated by a transaction T_i is also honoured by the history. Condition (3) requires that some ordering is imposed on any pair of conflicting operations in the history.

A complete history models the execution of a set of transactions which all either committed or aborted. A transaction which has done neither yet is said to be *active.* A history which involves active transactions is simply referred to as a *history,* instead of a *complete history,* and is simply a prefix (as defined in Section 6.1.2) of a complete history. Histories which are not complete can be used to model failure situations. Given a history H, $C(H)$ refers to the *committed projection* of H. The committed projection of a history is obtained by deleting all operations from the history which do not belong to transactions which have committed in H. Clearly, for any H, $C(H)$ is a complete history over the committed transactions in H.

We illustrate the definitions of transactions and histories with the following example. The three transactions used are the same as in Section 3.3. Note that although these transactions are actually described as total orders, this is not a requirement for the definition of a transaction.

Example 8 *Given the following three transactions, Figure 6–2 shows a possible history of an interleaved execution of them. Note that the diagram for the history does not show orderings (arrows in the diagram) which follow by transitivity. For example, although the definition of a history requires an ordering on all conflicting operations, their is no arrow from* $r_2[u]$ *to* $w_3[u]$. *It has not been drawn since it follows by transitivity from* $r_2[u] \rightarrow w_2[u] \rightarrow r_3[u] \rightarrow w_3[u]$.

T_1: $r_1[z] \rightarrow r_1[x] \rightarrow r_1[y] \rightarrow r_1[t] \rightarrow w_1[t] \rightarrow r_1[m] \rightarrow r_1[n] \rightarrow w_1[n] \rightarrow c_1$
T_2: $w_2[x] \rightarrow r_2[z] \rightarrow r_2[u] \rightarrow w_2[u] \rightarrow c_2$
T_3: $w_3[y] \rightarrow r_3[l] \rightarrow r_3[k] \rightarrow w_3[k] \rightarrow r_3[u] \rightarrow w_3[u] \rightarrow r_3[p] \rightarrow c_3$

6.2.2 Serializable Histories

As mentioned earlier, an interleaved execution of transactions is correct if its results and effects on the database are the same as some serial execution of the same transactions. The equivalence of two histories is defined as follows.

$$\begin{tikzcd}
r_1[z] \arrow[r] & r_1[x] \arrow[r] & r_1[y] \arrow[r] & r_1[t] \arrow[r] & w_1[t] \arrow[r] & r_1[m] \arrow[r] & r_1[n] \arrow[r] & w_1[n] \arrow[r] & c_1 \\
 & w_2[x] \arrow[u] \arrow[r] & r_2[z] \arrow[r] & r_2[u] \arrow[r] & w_2[u] \arrow[r] \arrow[dr] & c_2 & & & \\
 & w_3[y] \arrow[uur] \arrow[r] & r_3[l] \arrow[r] & r_3[k] \arrow[r] & w_3[k] \arrow[r] & r_3[u] \arrow[r] & w_3[u] \arrow[r] & r_3[p] \arrow[r] & c_3
\end{tikzcd}$$

Figure 6–2: A History

Definition 9 *Two histories H and H' are* ***equivalent*** *($\equiv$), if*

1. *they are defined over the same set of transactions and have the same operations; and*

2. *they order conflicting operations of non-aborted transactions in the same way; that is for any conflicting operations p_i and q_j belonging to transactions T_i and T_j (respectively) where $a_i, a_j \notin H$, if $p_i <_H q_j$, then $p_i <_{H'} q_j$.*

(Bernstein et al. [10], page 30.)

A complete history H is *serial*, if the operations of transactions in that history are not interleaved, i.e. if any operation of transaction T_i precedes any operation of transaction T_j, then all operations of T_i precede all operations of T_j. In such a serial history each transaction is executed entirely before the next transaction is started.

A partially executed transaction (an active transaction) may leave the database temporarily in an inconsistent state. For this reason, it would not be acceptable to compare interleaved executions with serial, but incomplete, histories. Therefore, an interleaved history H is correct — it is said to be *serializable (SR)* — if its committed projection, $C(H)$, is equivalent to some serial history H_s of the set of transactions that committed in $C(H)$.

6.2.3 Serializability Theorem

To test whether a history H is serializable, a *serialization graph*, denoted by $SG(H)$, is used. Given history H, the nodes of $SG(H)$ are all transactions which committed in H. There is an edge between two nodes T_i and T_j, if $\exists\ p_i \in OP_i$, $q_j \in OP_j$ and p_i and q_j are conflicting operations. If $p_i <_H q_j$, then there is an edge $T_i \rightarrow T_j$, otherwise $T_j \rightarrow T_i$. The serialization graph for the history shown in Figure 6–2 is given in Figure 6–3.

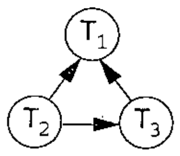


Figure 6–3: Serialization Graph

Theorem 1 *(The Serializability Theorem) A history H is serializable iff $SG(H)$ is acyclic. (Bernstein et al. [10], page 33)*

Proof Outline: (if) Let $\{T_1, \ldots, T_m\}$ be the set of committed transactions in H. Since $SG(H)$ is acyclic it may be topologically sorted into $T_{i_1}, \ldots, T_{i_m}$ (a permutation of $T_1, \ldots, T_m$). One can show that the serial history $H_S = T_{i_1} \rightarrow \ldots \rightarrow T_{i_m}$ is equivalent to $C(H)$. In short, the order of any pair of conflicting operations in $C(H)$ is maintained in H_S, because of the corresponding edges in $SG(H)$ and the fact that the topological sort maintains the order on transactions imposed by these conflicts. Since $C(H) \equiv H_S$, H is serializable.

(only if) Assuming H is serializable, if there was a cycle in SG(H) it would also imply a cycle in any equivalent serial history H', which of course is not possible by the definition of a serial history, and hence, there can be no cycle in SG(H), if H is serializable.

This is only the outline of the proof given in Bernstein et al. [10] (page 33). For more details, the reader is referred there. □

Because they are based on the concept of conflicting operations, the definitions of equivalence and serializability described above are usually referred to as *conflict* *equivalence* and *conflict serializability*. There exist alternative definitions, *view equivalence* and *view serializability*, but we will not discuss them in this dissertation. Although view serializability is less restrictive than conflict serializability, i.e. the set of histories allowed under conflict serializability is a proper subset of the set of histories allowed under view serializability, algorithms based on it are of little practical value since testing whether a history is view serializable is an NP-complete problem [71].

A similar argument applies to multi-version histories which are described in the following sections. Papadimitriou [71] differentiates between multi-version serializability based on view serializability and conflict multi-version serializability. As before, the set of schedules accepted by the latter is a proper subset of the former, but there exist efficient algorithms to test for it; testing whether a history is multi-version serializable (based on view serializability) is an NP-complete problem.

Bernstein et al. [10] only describe a conflict based version of multi-version concurrency control. Our work on branching transactions in based on their notion of multi-version histories. The following sections, therefore, are again based on Bernstein et al. [10].

6.2.4 Multi-version Histories

As was discussed in previous chapters, in a multi-version concurrency control algorithm, every Write operation of a transaction creates a new version of a data item. When a transaction wants to read a data item, the scheduler must decide which of the currently existing versions of the requested item should be read. To reason about the interleaved execution of transactions in a multi-version environment, a *multi-version (MV) history* is used. From now on we refer to histories described in previous sections as *single version histories* to distinguish them from MV histories. The mapping of single-version data item operations, which are sent by transactions to the scheduler (transactions are not aware of the existence of multiple versions), to the appropriate multi-version operations, which are executed by the scheduler, is captured in a mapping function h. h maps each $w_i[x]$ into $w_i[x_i]$, each $r_i[x]$ into $r_i[x_j]$ for some j, each c_i into c_i, and each a_i into a_i.

Definition 10 *A* ***complete multi-version (MV) history*** *H over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ and a given translation function h is a partially ordered set $(OP_H, <_H)$, where*

1. $OP_H = h(\bigcup_{i=1}^{n} OP_i)$;
2. *for each T_i and all operations $p_i, q_i \in OP_i$, if $p_i <_i q_i$ then $h(p_i) <_H h(q_i)$;*
3. *if $h(r_j[x]) = r_j[x_i]$, then $w_i[x_i] <_H r_j[x_i]$;*
4. *if $w_i[x] <_i r_i[x]$, then $h(r_i[x]) = r_i[x_i]$; and*
5. *if $h(r_j[x]) = r_j[x_i]$, $i \neq j$ and $c_j \in OP_H$, then $c_i <_H c_j$.*

Condition (1) says that every single-version operation submitted by a transaction is translated into the appropriate multi-version operation. Condition (2) states that any ordering described by a transaction is also observed in the MV history. Condition (3) requires that a version of an item has to be produced before it can be read. Condition (4) says that if a transaction has written a particular data item before it reads it, it must read the version of the item it has written. The last condition specifies that a transaction can only commit if all those transactions which have written versions of data items read by this transaction have committed first. A history which satisfies condition (4) is said to *preserve reflexive reads-from relationships.* A history is *recoverable*, if it satisfies condition (5).

A *MV history H* is a prefix of a complete MV history. As for one-version (1V) histories, the *committed projection C(H)* of a MV history can be obtained by removing all operations from the history which do not belong to committed transactions in H. Two MV operations *conflict* if they access the same version of a data item, and one is a Write operation. The only possible kind of conflict is $w_i[x_i] <_H r_j[x_i]$. There are no conflicts between Write operations (since they always operate on different versions), and $r_j[x_i] <_H w_i[x_i]$ is not possible, because of condition (3) of Definition 10.

Two MV histories over a set of transactions are *equivalent*, iff the histories have the same MV operations . If there is a bijective function — such as the mapping function h described earlier — from the operations of a 1V history (H_{1V}) to the operations of a MV history (H_{MV}), and if both histories have the same reads-from relationships, then the two histories are equivalent: $H_{1V} \equiv H_{MV}$ (MV history equivalence in Bernstein et al. [10], pp. 148-149).

Serialization graphs of MV histories are similar to those of 1V histories. The nodes consist of all transactions in the committed projection of the history ($C(H_{MV})$). There is an edge $T_i \rightarrow T_j$ ($i \neq j$), iff for some x, T_j reads from T_i, i.e. $r_j[x_i]$ is an operation of $C(H_{MV})$.

Proposition 2 *Let H and H' be MV histories. If $H \equiv H'$, then $SG(H) = SG(H')$. (Bernstein et al. [10], page 149)*

6.2.5 One Copy Serializability

A complete MV history is *serial* if the operations of transactions in that history are not interleaved. That is, for any two transactions, T_i and T_j, if any operation of T_i precedes any operation of T_j, then all operations of T_i precede all operations of T_j. A serial MV history may not be equivalent to any serial 1V history. For example, assume that T_1, T_2 and T_3 are transactions in a serial MV history (H_{MV}), that $T_1 \rightarrow T_2 \rightarrow T_3$, and that $w_1[x_1]$, $w_1[y_1]$, $r_2[y_1]$, $w_2[x_2]$, $w_2[z_2]$, $r_3[x_1]$ and $r_3[z_2]$ are operations in H_{MV}. There can be no equivalent serial 1V history to H_{MV}, since T_2 would have to be ordered after T_1 (due to the reads-from relationship on item y), and T_3 would have to be ordered after T_2 (due to the reads-from relationship on item z). In this case, however, T_3 could not have read the version of x written by T_1, since T_2 has already overwritten it.

The subset of serial MV histories for which there exists an equivalent serial 1V history are said to be *one-copy serial* or *1-serial* (Definition 11). An MV history is *one-copy serializable* (1SR), if its committed projection is equivalent to a 1-serial MV history.

Definition 11 *A serial MV history H_{MV} is* ***one-copy serial*** *if for all i, j and x, if T_i reads x from T_j, then $i = j$, or T_j is the last transaction preceding T_i that writes into any version of x. (Bernstein et al. [10], page 150)*

The relationship between MV histories and 1V histories described in Theorem 2 is the basis for the *1-Serializability Theorem* described next.

Theorem 2 *Let H_{MV} be an MV history over a set of transactions T. $C(H_{MV})$ is equivalent to a serial, 1V history over T, iff H_{MV} is 1SR (Bernstein et al. [10], page 150).*

6.2.6 1-Serializability Theorem

Given Theorem 2, we know that a scheduler that applies a multi-version concurrency control algorithm is correct if it only allows MV histories which are 1SR. To test whether a MV history H_{MV} is 1SR, a *multi-version serialization graph* is used.

Given a multi-version history H_{MV} and a data item x, the *version order* $\ll_x$ describes a total ordering of versions of x in H_{MV}. A *version order* $\ll$ for H_{MV} is the union of the version orders of all data items.

Definition 12 *Given an MV history H_{MV} and a version order $\ll$, the* ***multi-version serialization graph*** *for H_{MV}, $MVSG(H_{MV}, \ll)$ is $SG(H_{MV})$ with the following* version order edges *added: for each $r_k[x_j]$ and $w_i[x_i]$ in $C(H_{MV})$, where i, j and k are distinct, if $x_i \ll x_j$ then include $T_i \rightarrow T_j$, otherwise include $T_k \rightarrow T_i$. Note that there is no version order edge if $j = k$. (Bernstein et al. [10], page 152)*

Figure 6–4 shows a possible multi-version history for the three transactions of Example 8. The serialization graph for this history is $T_3 \rightarrow T_2 \rightarrow T_1$. Since there are no version order edges to be added, its multi-version serialization graph is identical to the serialization graph.

To demonstrate the use of version order edges, let us assume that T_1 performs one more operation, $w_1[u_1]$, just before it commits; the last three operations of T_1 are now $w_1[n_1] \rightarrow w_1[u_1] \rightarrow c_1$. For a version order that contains $u_1 \ll u_3$ the edge $T_1 \rightarrow T_3$ would have to be added to the corresponding multi-version serialization graph, otherwise $T_2 \rightarrow T_1$ should be added.

$$\begin{tikzcd}
r_1[z_0] \arrow[r] & r_1[x_2] \arrow[r] & r_1[y_0] \arrow[r] & r_1[t_0] \arrow[r] & w_1[t_1] \arrow[r] & r_1[m_0] \arrow[r] & r_1[n_0] \arrow[r] & w_1[n_1] \arrow[r] & c_1 \\
 & w_2[x_2] \arrow[u] \arrow[r] & r_2[z_0] \arrow[r] & r_2[u_3] \arrow[r] & w_2[u_2] \arrow[r] & c_2 & & & \\
w_3[y_3] \arrow[r] & r_3[l_0] \arrow[r] & r_3[k_0] \arrow[r] & w_3[k_3] \arrow[r] & r_3[u_0] \arrow[r] & w_3[u_3] \arrow[r] \arrow[ull] & r_3[p_0] \arrow[r] & c_3 &
\end{tikzcd}$$

Figure 6–4: Multi-Version History

A multi-version serialization graph describes dependencies between transactions due to conflicting operations as well as dependencies due to orderings of versions in a multi-version system. It can be shown (Theorem 3) that there exists a version order for a particular multi-version history, such that the multi-version serialization graph for that history and order is acyclic, if and only if the history is 1SR.

Theorem 3 *An MV history H_{MV} is 1SR iff there exists a version order $\ll$ such that $MVSG(H_{MV}, \ll)$ is acyclic. (Bernstein et al. [10], page 152)*

Theorem 3 is central to the proofs of correctness for branching transaction schedulers. Using this theorem, we will show that the committed projections of histories produced by branching transaction schedulers are always 1SR, and hence correct.

Classical serializability, as described in this section, is extended for branching transactions next.

6.3 Serializability of Branching Transactions

As explained above, transactions in classical serializability theory are modelled as flat structures, i.e. they are described as sets of operations with some ordering defined on them. To be able to reason about the correctness of branching transactions, however, we need to add some notion of internal structure to a transaction; we must be able to model the relationships between a branching transaction and its components, and the interaction between different branching transactions and their components.

In this section, therefore, we extend the concepts of *transactions* and *histories*, as defined in classical serializability theory, to incorporate the additional internal structure of branching transactions. First, a general framework of *agents* and *basic agents* is presented. This framework allows us to work out some general concepts without the need to discuss details of branching transactions. Although the intended association between agents/basic agents and branching transactions/branching transaction components should be obvious (it will be discussed in detail later in this section), the framework is meant to be more general. In particular, it may also prove useful for an extension of classical serializability theory which can deal with *nested transactions* [27,63,64,76][3]. After the introduction of the agent framework, we formally define branching transactions and their histories.

6.3.1 Basic Agents, Agents and Agent Histories

The concept of agents has primarily been introduced to provide a framework within which we can extend the work of Bernstein et al. [10] for branching transactions. Introducing some general definitions now keeps subsequent definitions of branching transactions simpler.

[3]This extension is, however, beyond the scope of this dissertation, and is not discussed any further here.

As was the case with "classical" transactions, the model of agents and basic agents is merely used to facilitate a retrospective analysis of the correctness of a scheduler. It is not used in an operational sense to describe the execution of agents/basic agents. Furthermore, it is a theoretic framework and should not be confused with issues in implementing a system such as branching transactions.

An *agent* A_i is an entity which performs operations. For a given system, the kind of operations which can be executed by an agent must be defined. An agent A_i is associated with one or more *basic agents* A_{ij}, which execute operations on behalf of A_i. *An operation* is a triple (p, i, j), where p denotes the operation type, such as Read or Write, and i and j identify the basic agent (A_{ij}) which executed it.

A basic agent must be created before it is allowed to execute any operation. One basic agent creates another by executing a *branch*-operation: $b(k, l)$, where k and l refer to the newly created basic agent A_{kl}. Hence, the triple $(b(k, l), i, j)$ — from now on written as $b_{ij}(k, l)$ — indicates that A_{ij} created A_{kl}. We use the notation $creator(k, l)$ to denote the second subscript of the basic agent which created A_{kl}, i.e. if $b_{ij}(k, l)$ was executed to create A_{kl}, then $creator(k, l) = j$.

Of course, unless we have at least one basic component to start with, no operations will ever be executed. We use A_{00} to denote this special basic agent. Its only task is to create new agents A_i by creating their first basic agent A_{i1}; we allow no other basic agent to do that. It follows that for every A_{i1}, $creator(i, 1) = 0$. A_{00} exists by definition and does not have to be created. Basic agents, other than A_{00}, are only allowed to create new basic agents which are associated with the same agent, i.e. have the same first index.

Depending on the system we would like to model, some operations may have to precede others. For example, as mentioned above, a basic agent must be created before it can execute any operation (with the exception of A_{00}). Such precedence relationships are described in the form of partial orderings on the operations of an agent. Formally, we define basic agents and agents as follows:

Definition 13 *Let P denote the type of operations which are allowed in our system. A* ***basic agent*** *A_{ij} is a partially ordered set $(OP_{ij}, <_{ij})$, where*

- *$OP_{ij} \subseteq \{(p,i,j) \mid p \in P\}$ (These are the operations the basic agent may consist of.)*
- *$<_{ij}$ is a partial ordering on OP_{ij}*

Definition 14 *Let $BA_i = \{A_{ij_1}, A_{ij_2}, A_{ij_3}, \ldots, A_{ij_m}\}$ be a set of basic agents with first subscript i. We define an* ***agent*** *A_i over the set of basic agents BA_i to be the partially ordered set $(OP_i, <_i)$, where*

- *$OP_i = \bigcup_{k=1}^{m} OP_{ij_k}$:*
  the set of operations of A_i is the union of the sets of operations of its basic agents, and
- $$<_i = \bigcup_{k=1}^{m} \Big(<_{ij_k} \cup \{(b_{ij_x}(i,j_k),(p,i,j_k)) \mid (p,i,j_k) \in OP_{ij_k} \wedge j_k \neq 1 \wedge x \neq k \wedge b_{ij_x}(i,j_k) \in OP_{ij_x}\} \Big):$$
  any ordering on the execution of operations observed by its basic agents is also observed by the agent, and unless basic agent A_{ij} has been created by A_{00} — for any agent A_i this initial basic agent is A_{i1} — any of its operations must be preceded by its creation,

and the following conditions must hold:

1. *every basic agent is uniquely identified by its indices:*
   let A_{ij} and A_{ik} be two different basic agents of A_i, then $j \neq k$.
2. *every basic agent is only created once:*
   let $p_1 = b_{i_1j_1}(i_1,k_1)$ and $p_2 = b_{i_2j_2}(i_2,k_2)$. If $i_1 = i_2$ and $k_1 = k_2$, then $p_1 = p_2$.

3. *all basic agents, other than A_{00}, can only create new basic agents which are associated with the same agent:*
   if $b_{ij}(k,l) \in OP_i$ and $i \neq 0$, then $i = k$.

To describe the hierarchy of basic agents within an agent, we use the notion of an *agent tree.* Such a tree reflects the substructure of an agent, i.e. which basic agents were created by which other basic agents. Formally we define an agent tree as follows:

Definition 15 *An* ***agent tree*** *AG_i for agent A_i is a rooted tree $[V, E, r]$, where*

- *V is the set of basic agents A_{ij} which are associated with agent A_i:*
  $V = \{A_{i1}\} \cup \{A_{ik} \mid b_{ij}(i,k) \in OP_i,$ *for some* $j\}$,
- *E is the set of edges between basic agents. There is an edge from A_{ij} to A_{ik}, if and only if A_{ij} created A_{ik}:*
  $E = \{(A_{ij}, A_{ik}) \mid b_{ij}(i,k) \in OP_i\}$, *and*
- *the root r is basic agent A_{i1}*

Example 9 *Let A_i be an agent, such that*

- $OP_i = \{b_{i1}(i,2), b_{i1}(i,3), b_{i3}(i,4)\}$, *and*
- $<_i = \{(b_{i1}(i,3), b_{i3}(i,4))\}$.

The agent graph AG_i is shown in Figure 6–5.

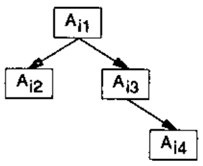


Figure 6–5: An Agent Graph

We draw the nodes of an agent graph as rectangles instead of circles to emphasise the difference between Hasse diagrams describing the order of operations in an agent and an agent graph, which describes the basic agent hierarchy within an agent.

Definition 16 *Basic agent A_{ij} is a* ***proper ancestor*** *of basic agent A_{ik} if and only if there exists a path of length one or more from A_{ij} to A_{ik} in agent graph AG_i. We use $anc_p(i,k)$ to denote the set of proper ancestors of basic agent A_{ik}. We use $anc(i,k) = anc_p(i,k) \cup \{A_{ik}\}$ to denote the set of* ***ancestors*** *of basic agent A_{ik}.*

Definition 17 *Basic agent A_{ik} is a* ***proper descendant*** *of basic agent A_{ij}, if and only if there exists a path of length one or more from A_{ij} to A_{ik} in agent graph AG_i. We use $desc_p(i,j)$ to denote the set of proper descendants of basic agent A_{ij}. We use $desc(i,j) = desc_p(i,j) \cup \{A_{ij}\}$ to denote the set of* ***descendants*** *of basic agent A_{ik}.*

We now extend our model of agents to deal with the interleaved execution of multiple agents by a scheduler. As we have seen in the case of multi-version histories (Definition 10), an operation submitted to a scheduler may have to be mapped to some other appropriate operation by the scheduler. In this general framework we are not interested in specifying a particular mapping, as the actual mapping will be dependent on the details of the system that needs to be modelled. We are establishing the correctness of a family of schedulers for which some such mapping exists.

Also system-dependent is the notion of *conflicting operations.* We say that two operations *conflict*, if the order in which they are executed has an effect on the results of the schedule. If two operations don't conflict, then the final state of the system and the results obtained by agents are independent of the order in which these two operations are executed.

The *agent history* of a set of agents describes the interleaved execution of the associated basic agents in terms of operations involved and some partial ordering of these operations. Formally, we define an agent history as follows:

Definition 18 *Let h be a mapping function for a scheduler and Let $A = \{A_0\} \cup \{A_{i_1}, A_{i_2}, \ldots, A_{i_n}\}$ be a set of agents. Given a mapping function h, the* ***agent history*** *H is a partially ordered set $(OP_H, <_H)$, where*

- $OP_H = h(\bigcup_{j=1}^{n} OP_{i_j})$*:*
  The operations in history H involve exactly the operations of agents in A.

- $<_H \supseteq \bigcup_{j=1}^{n} <_{i_j}$ *:*
  All operation orderings stipulated by agents in A are observed by the history.

- *for any two conflicting operations $p, q \in OP_H$, either $p <_H q$ or $q <_H p$:*
  Any pair of conflicting operations must be ordered in H.

Based on this framework of agents, a formal definition of branching transactions is given next.

6.3.2 Formal Model of Branching Transactions

As was the case for the definition of "classical" transactions (Definition 7), the formal definition of a branching transaction does not have to model every observable aspect of the execution of a branching transaction. We are only interested in information which is used by a BT scheduler to guarantee serializable schedules. Classical serializability describes a particular execution of a transaction in terms of the Read and Write operations it performed on the database, and whether the transaction successfully committed or aborted. Similarly, we use the definition of a branching transaction to describe the Read and Write operations which were executed by its components and whether these components aborted or committed. In addition, we use Branch operations to capture the dynamic creation and termination of branching transaction components. Table 6–1 summarises the types of operations which can be executed by a branching transaction component, or which can be associated with a branching transaction component by the system.

Read, Write and Branch operations have a data item as a parameter. We model a database as a finite set of data items; we let DB denote this set. In

Operation Type		*Explanation*
Read:	$r[x]$	read data item x
Write:	$w[x]$	write data item x
Commit:	c	commit branching transaction component
Abort:	a	abort branching transaction component
Branch:	$b(k, l, x)$	create new branching transaction component BTC_{kl}, because of data conflict on data item x

Table 6–1: Branching Transactions Operation Types

general, we us the letter x when referring to a data item. If we have to distinguish between multiple data items, we use subscripts: x_1, x_2, etc.[4]

A BT operation describes the execution of some type of BT operation by a branching transaction component, and is denoted by a triple (p, i, j), where p is one of the BT operation types listed in Table 6–1, and i and j identify the BT component which executed it. For better readability we use the notation summarised in Table 6–2 instead of triples.

Operation	*Notation*
$(r[x], i, j)$	$r_{ij}[x]$
$(w[x], i, j)$	$w_{ij}[x]$
(c, i, j)	c_{ij}
(a, i, j)	a_{ij}
$(b(k, l, x), i, j)$	$b_{ij}(k, l, x)$

Table 6–2: BT Operations Notation

[4]This is different from the notation we used earlier, where x_1, x_2, etc. was used to describe different versions of item x. In our new notation, different versions of item x_d will be denoted as x_{d,i_1j_1}, x_{d,i_2j_2}, etc. This will be described in more detail later in this chapter.

Given the kind of operations which can be executed in a branching transaction system, we can now define branching transactions and their components in terms of basic agents and agents. A branching transaction component is a particular kind of basic agent, and a branching transaction is a particular kind of agent. Each branching transaction consists of as many *branches* as there are leaf nodes in its agent tree. Each path from the root of the tree to a leaf node describes a branch, i.e. a branch consists of the set of basic transaction components along one such path. Formally, we define branching transactions and their components as follows:

Definition 19 *A* ***branching transaction component*** BTC_{ij} *is a basic agent* $(OP_{ij}, <_{ij})$*, where*

1. $OP_{ij} \subset \{r_{ij}[x], w_{ij}[x] \mid x \in DB\} \cup$
   $\{b_{ij}(i,k,x) \mid k > 1 \wedge x \in DB\} \cup$
   $\{a_{ij}, c_{ij}\}$

2. $<_{ij}$ *is a partial ordering on* OP_{ij}

Definition 20 *Let* $BC = \{BTC_{ij_1}, BTC_{ij_2}, BTC_{ij_3}, \ldots, BTC_{ij_m}\}$ *be a set of branching transaction components. We define a* ***branching transaction*** BT_i *to be the agent* $(OP_i, <_i)$ *over* BC*, with the following additional conditions:*

- *At most one component of a branching transaction executes a Commit:*
  if $c_{ij} \in OP_i$ *and* $c_{ik} \in OP_i$*, then* $j = k$*.*

- *Every component of a branching transaction executes a Branch, a Commit or an Abort operation (the execution of one type excludes the execution of the other two types):*
  $c_{ij} \in OP_i$*, iff for all* k*,* l *and* x*:* $a_{ij}, b_{ij}(k,l,x) \notin OP_i$*;*
  $a_{ij} \in OP_i$*, iff for all* k*,* l *and* x*:* $c_{ij}, b_{ij}(k,l,x) \notin OP_i$*;*
  for some k*,* l *and* x*:* $b_{ij}(k,l,x) \in OP_i$ *iff* $c_{ij}, a_{ij} \notin OP_i$*.*
  Note that more than one Branch operation may be executed by one branching transaction component.

- *The Commit, Abort or Branch operations are the last operations in every branching transaction component:*
  if t is a_{ij} or c_{ij}, then for any other operation $p \in OP_i$, $p <_i t$;
  if t is $b_{ij}(i,k,x)$, then for any other operation $p \in OP_i$, such that $t <_i p$, $p \in \{b_{ij}(i,l,y) \mid l \neq k \text{ and } x = y\}$.
  The second condition implies that although there may be more than one Branch operation at the end of a branching transaction component, they must all be related to the same data item x.

- *If, along a single branch of a branching transaction, the same data item is read as well as written, then these two operations (Read and Write) must be ordered in some way:*
  Let BTC_{ij} and BTC_{ik} be two components in BC, such that $BTC_{ij} \in anc(i,k)$ or $BTC_{ik} \in anc(i,j)$. If $r_{ij}[x] \in OP_i$ and $w_{ik}[x] \in OP_i$, then either $r_{ij}[x] <_i w_{ik}[x]$ or $w_{ik}[x] <_i r_{ij}[x]$.

In Chapter 3, we have seen that a BT scheduler maps the Read and Write operations submitted to it by branching transaction components to the appropriate version operations. We, therefore, need to specify a mapping function h for BT schedulers before we can give the definition of a BT history. We denote the version of data item x which was written by BTC_{ij} by x_{ij}, versions of x_d by $x_{d,ij}$.

$$h(p) = \begin{cases} r_{ij}[x_{kl}], \text{for some } k \text{ and } l & \text{if } p = r_{ij}[x] \\ w_{ij}[x_{i,j}] & \text{if } p = w_{ij}[x] \\ b_{ij}(k,l,x) & \text{if } p = b_{ij}(k,l,x) \\ c_{ij} & \text{if } p = c_{ij} \\ a_{ij} & \text{if } p = a_{ij} \end{cases}$$

Our mapping function h does not exactly specify which version of an item is to be read by a Read operation; the k and l values in $r_{ij}[x_{kl}]$ are undefined, even so some constraints (for example $w_{ij}[x_{ij}] <_i r_{ik}[x_{lm}]$, see Definition 22) do apply. Hence, h is strictly speaking not a function, but a family of functions. The details of the mapping of a Read differ from one BT scheduler to another. Since we want our proofs to hold for all BT schedulers we are not making any assumptions about the details of the mapping of a Read, i.e. the proofs that follow do not depend on these details.

The interleaved execution of a set of branching transactions is modelled as a *BT history.* A *complete BT history* is an agent history for the above mapping function h and the following definition of *conflicting operations.*

Definition 21 *Two branching transaction operations, p and q,* ***conflict*** *with each other if $h(p)$ is a Read operation, say $(r_{ij}[x_{kl}])$, reading the version of a data item (x_{kl}) which was written by $h(q)$ $(w_{kl}[x_{kl}])$, and p and q are operations of different transactions $(i \neq k)$.*

Definition 22 *A* ***complete BT history*** *H over a set of branching transactions $BT = \{BT_0\} \cup \{BT_{i_1}, BT_{i_2}, \ldots, BT_{i_n}\}$ is an agent history $(OP_H, <_H)$, with the following additional conditions:*

1. *A branching transaction component can only read a data item version after it has been created:*
   if $r_{ij}[x_{kl}] \in OP_H$, then $w_{kl}[x_{kl}] \in OP_H$ and $w_{kl}[x_{kl}] <_H r_{ij}[x_{kl}]$.

2. *If a branching transaction component reads a data item which was updated by one of its ancestors (including itself), then the version of the item that it reads must be the one which the ancestor created:*
   if $w_{ij}[x_{ij}] <_i r_{ik}[x_{lm}]$, and $BTC_{ij} \in anc(i,k)$, then $i = l$ and $j = m$.

3. *If a branching transaction commits, then all the versions of data items which were read by components of the committing branch must be committed versions, i.e. the transaction components which created them must have committed*[5]*:*

[5]We say that a branching transaction component has committed if it is part of a branch that has committed; a branch has committed, if its leaf node component has committed.

if $r_{ij}[x_{kl}] \in OP_H$ and $i \neq k$ and $BTC_{id} \in desc(i,j)$ and $c_{id} \in OP_H$, then there must be a $BTC_{km} \in desc(k,l)$ such that $c_{km} \in OP_H$ and $c_{km} <_H c_{id}$.

A BT history of the execution of our three example transactions is shown in Figure 6–6. This history corresponds to the execution of these transactions as shown in Table 3–2 of Chapter 3.

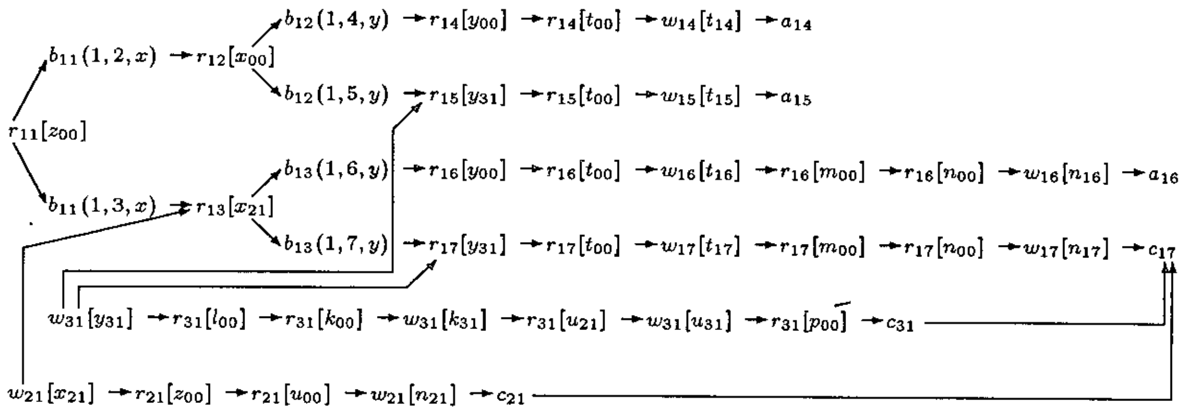

$r_{11}[z_{00}]$
$b_{11}(1,2,x) \to r_{12}[x_{00}]$
$b_{12}(1,4,y) \to r_{14}[y_{00}] \to r_{14}[t_{00}] \to w_{14}[t_{14}] \to a_{14}$
$b_{12}(1,5,y) \to r_{15}[y_{31}] \to r_{15}[t_{00}] \to w_{15}[t_{15}] \to a_{15}$
$b_{11}(1,3,x) \to r_{13}[x_{21}]$
$b_{13}(1,6,y) \to r_{16}[y_{00}] \to r_{16}[t_{00}] \to w_{16}[t_{16}] \to r_{16}[m_{00}] \to r_{16}[n_{00}] \to w_{16}[n_{16}] \to a_{16}$
$b_{13}(1,7,y) \to r_{17}[y_{31}] \to r_{17}[t_{00}] \to w_{17}[t_{17}] \to r_{17}[m_{00}] \to r_{17}[n_{00}] \to w_{17}[n_{17}] \to c_{17}$
$w_{31}[y_{31}] \to r_{31}[l_{00}] \to r_{31}[k_{00}] \to w_{31}[k_{31}] \to r_{31}[u_{21}] \to w_{31}[u_{31}] \to r_{31}[p_{00}] \to c_{31}$
$w_{21}[x_{21}] \to r_{21}[z_{00}] \to r_{21}[u_{00}] \to w_{21}[n_{21}] \to c_{21}$

Figure 6–6: Branching Transaction History

A BT history may be incomplete due to some failure situation, i.e. a system crash may leave some branching transaction components incomplete; they neither aborted nor branched nor committed. Condition (3) of Definition 22 ensures that no committed transaction can see the effects of such failed transactions. Since all updates of failed transactions are removed from the database by the recovery manager, the final state of the database is also independent of these failed transactions. Hence, as was the case previously, in order to prove that a BT history is serializable, it is sufficient to show that its committed projection is serializable.

Definition 23 *The **committed projection** of BT history $H = (OP_H, <_H)$ is obtained through the following three steps, and is denoted by $C(H)$.*

1. *Delete all operations $p_{i,j} \in BTC_{i,j}$ from H for which for all $BTC_{i,d} \in desc(i,j)$, $c_{i,d} \notin OP_H$ and*

2. *Delete all $b_{i,j}(i,k)$ operations from H.*

3. Map each $r_{i,j}[x_{k,l}]$ into $r_i[x_k]$, each $w_{i,j}[x_{i,j}]$ into $w_i[x_i]$, and each $c_{i,j}$ into c_i.

Step (1) eliminates all operations which were not executed by those components that are part of the committing branch of committed BT transactions. To determine whether a history is serializable, we do not need the information about which operation was executed by what transaction component, but only by which transaction. Hence, we can delete all BT Branch operations and map all remaining operations $o_{i,j}$ to o_i (steps (2) and (3)). The latter implicitly leads to a mapping of data versions $x_{i,j}$ to x_i. Figure 6–7 shows the committed projection of the BT history given in Figure 6–6.

$$\begin{tikzcd}
r_1[z_0] \arrow[r] & r_1[x_2] \arrow[r] & r_1[y_3] \arrow[r] & r_1[t_0] \arrow[r] & w_1[t_1] \arrow[r] & r_1[m_0] \arrow[r] & r_1[n_0] \arrow[r] & w_1[n_1] \arrow[r] & c_1 \\
 & w_2[x_2] \arrow[u] \arrow[r] & r_2[z_0] \arrow[r] & r_2[u_0] \arrow[r] & w_2[u_2] \arrow[r] & c_2 \arrow[urrr] & & & \\
w_3[y_3] \arrow[uurr] \arrow[r] & r_3[l_0] \arrow[r] & r_3[k_0] \arrow[r] & w_3[k_3] \arrow[r] & r_3[u_0] \arrow[r] & w_3[u_3] \arrow[r] & r_3[p_0] \arrow[r] & c_3 \arrow[uur] &
\end{tikzcd}$$

Figure 6–7: Committed Projection of Branching Transaction History

To be able to reason about the correctness of a BT history we now define the equivalence between the committed projection of a BT history and a multi-version history. We say that the committed projection of a BT history, $C(H_{BT})$, is equivalent to a multi-version history, H_{MV}, if both contain the same set of transactions, all reads-from relationships are the same, and all final writes to the database are the same. Since both create multiple versions of data items, the latter is trivially true if they both contain the same Write operations. To maintain the same reads-from relationships, it must further hold that if $r_i[x_j] \in OP_{C(H_{BT})}$ then $r_i[x_j] \in OP_{H_{MV}}$, and vice versa. That is, if in $C(H_{BT})$ transaction T_i reads a data item version created by T_j, then the same is true in H_{MV}; and vice versa. Hence, we can use the following definition.

Definition 24 *Let H_{BT} be a BT multi-version history. The committed projection of H_{BT}, denoted $C(H_{BT})$, is **equivalent** to a multi-version history, say H_{MV}, iff* *they have the same operations. We write $C(H_{BT}) \equiv H_{MV}$ to denote this equivalence.*

6.3.3 Serializability Theorem for Branching Transactions

Theorem 2 tells us that a multi-version history is correct if it is 1SR (one-copy serializable). This is the case if there exists a version order for it such that the corresponding multi-version serialization graph is acyclic (Theorem 3). Hence, as long as a branching transaction scheduler only allows histories whose committed projections are equivalent to such correct (non-branching) MV histories, serializability is guaranteed. This gives rise to the following theorem.

Theorem 4 *(The Branching Transaction Serializability Theorem) A branching transaction history, say H_{BT} is serializable if its committed projection is equivalent to a multi-version history, say H_{MV}, for which there exists a version order, $\ll$, such that $MVSG(H_{MV}, \ll)$ is acyclic.*

Proof: Follows directly from Theorems 2 and 3. □

6.4 Correctness of Two-Phase Locking for Branching Transactions

The previous chapter introduced various forms of locking algorithms for branching transactions. The most general of these was HBT-MV2PL, a hybrid 2-phase locking algorithm which dynamically switches between strict 2PL (single version, non-branching), multi-version 2PL (non-branching) and BT-MV-2PL. As was pointed out before, we can consider these algorithms as special cases of HBT-MV2PL. Hence, it is sufficient to show that all schedules allowed under HBT-MV2PL are serializable.

To capture the semantics of locking, we introduce 4 more BT operations[6]:

- $CRL_i[x_i]$: transaction BT_i acquires a *certified read lock* on x_i.
- $CWL_i[x_i]$: transaction BT_i acquires a *certified write lock* on x_i.
- $VRL_i[x_i]$: transaction BT_i acquires a *version read lock* on x_i.
- $VWL_i[x_i]$: transaction BT_i acquires a *version write lock* on x_i.

The syntax of these operations does not follow the previously introduced "triple"-notation of BT operations, i.e. we do not distinguish which operation was executed by which branching transaction component, but only by which branching transaction. There is no need to make a more detailed distinction at this stage; we are only trying to say something about the committed projection of a BT history (no aborted components are involved). Furthermore, to increase readability, we write H_c instead of $C(H)$, and $(OP_c, <_c)$ instead of $(OP_{C(H)}, <_{C(H)})$.

We are not using explicit "unlock"-operations since all algorithms apply strict locking, i.e. all locks are released at commit time, but not before. A Commit (c_i) in a history, therefore, implies the release of all locks held by that transaction.

2-Phase Locking Rules, Lock Compatibility Rules and the Overwrite Rule In Section 4.4.2 we introduced HBT-MV2PL properties in the form of 5 rules. These rules are now formally described.

Overwrite Rule :

- if $w_j[x_j], r_k[x_j] \in OP_c$ $(j \neq k)$ then for any i $(i, j, k$ distinct) such that $w_i[x_i] \in OP_c$ it follows that $c_i <_c c_j$ or $c_k <_c c_i$.

[6] x_i here refers to the version of data item x created by BT_i.

If a transaction commits, it must have acquired a CRL on every item it read. It follows that it must have read the most recently committed versions of these data items. Another transaction cannot overwrite the (committed) versions read by the committing transaction, since it would have to acquire corresponding CWL, which are incompatible with the existing CRLs of the committing transaction, and the CRLs are not released until the transaction has committed.

Write Lock Rule : If $w_i[x_i] \in OP_c$, then either

- $CWL_i[x_i] <_c w_i[x_i] <_c c_i$ (overwriting of the committed version), or
- $VWL_i[x_i] <_c w_i[x_i] <_c CWL_i[x_i] <_c c_i$ (creating a new version first).

When a transaction writes a data item, it can either overwrite the existing committed version, in which case it needs to acquire a certified write lock first, or it can create a new version of the item first and then overwrite the existing committed version at commit time. In the latter case it is sufficient to acquire a version write lock first, which is then later (during certification) upgraded to a certified write lock. In both cases it holds that $CWL_i[x_i] <_c c_i$.

Read Lock Rule : If $r_i[x_j] \in OP_c$, then either

- $CRL_i[x_j] <_c r_i[x_j] <_c c_i$ (reading of a committed version, i.e. $c_j <_c CRL_i[x_j]$), or
- $VRL_i[x_j] <_c r_i[x_j]$ and $c_j <_c CRL_i[x_j] <_c c_i$ (reading of an uncommitted version).

If a transaction reads the committed version of a data item, it must acquire a certified read lock for it first. If it reads an uncommitted version it acquires a version read lock on that version, but the transaction which created the version must commit before the reading transaction can. In both cases it holds that $CRL_i[x_j] <_c c_i$.

Write/Write Lock Conflict Rule : If $CWL_i[x_i], CWL_j[x_j] \in OP_c$ $(i \neq j)$, then either $c_i <_c CWL_j[x_j]$ or $c_j <_c CWL_i[x_i]$.

A transaction can only acquire a certified write lock on a data item version if no other transaction holds a certified write lock on any version of the same item. In other words; no two certified write locks can exist for the same data item at any time.

Read/Write Lock Conflict Rule : If $CRL_k[x_j], CWL_i[x_i] \in OP_c$ $(k \neq i$ and $j \neq i)$, then either $c_k <_c CWL_i[x_i]$ or $c_i <_c CRL_k[x_j]$.

A transaction can only obtain a certified write lock, if no other transaction holds a certified read lock on the same time, and vice versa, i.e. certified read locks and certified write locks are not compatible.

The following proposition (Proposition 3) states that if a transaction reads a data item version written by another transaction, then the "writer"-transaction must commit before the "reader"-transaction.

Proposition 3 *If $r_i[x_j] \in OP_c$, then $c_j <_c c_i$.*

Proof: By the Read Lock Rule it holds (for both cases: committed read and uncommitted read) that $c_j <_c CRL_i[x_j]$ and that $CRL_i[x_j] <_c c_i$ By transitivity, $c_j <_c c_i$. □

Proposition 4 requires that the commit of any two transactions writing the same data item must be ordered in some way.

Proposition 4 *If $w_i[x_i], w_j[x_j] \in OP_c$, then either $c_i <_c c_j$ or $c_j <_c c_i$.*

Proof: By the Write Lock Rule we know that $CWL_i[x_i] <_c c_i$ and $CWL_j[x_j] <_c c_j$. By the Write/Write Lock Conflict Rule, we know $c_i <_c CWL_j[x_j]$ or $c_j <_c CWL_i[x_i]$ By transitivity, therefore, $c_i <_c c_j$ or $c_j <_c c_i$. □

Theorem 5 *Every committed projection of a BT history, H_c, produced by a HBT_MV2PL scheduler is equivalent to a (non-branching) MV history H_{MV} for which there exists a version order $\ll$, such that $MVSG(H_{MV}, \ll)$ is acyclic.*

Proof: Let $\ll$ be a version order such that for every $w_i[x_i]$, $w_j[x_j]$ $(i \neq j)$ $x_i \ll x_j$ if $c_i <_c c_j$ (because of Proposition 4, $\ll$ is indeed a version order). We will show that all edges in $MVSG(H_c, \ll)$ are in commit order, i.e. if $BT_i \rightarrow BT_j$ in $MVSG(H_c, \ll)$, then $c_i <_c c_j$.

Let $BT_i \rightarrow BT_j$ be in $SG(H_c)$, then BT_j must read a data item version created by BT_i: $r_j[x_i] \in OP_c$. By Proposition 3, $c_i <_c c_j$ follows directly.

Let $w_i[x_i]$, $w_j[x_j]$, $r_k[x_j] \in OP_c$, where i, j, k are distinct. Then either $x_i \ll x_j$ or $x_j \ll x_i$. If $x_i \ll x_j$, then the version order edge is $BT_i \rightarrow BT_j$. By the choice of our version order $\ll$, $c_i <_c c_j$ follows immediately. If $x_j \ll x_i$, then the added version order edge is $BT_k \rightarrow BT_i$. In this case we must show that $c_k <_c c_i$. Let us assume that this is not the case, i.e. let $c_i <_c c_k$. By the Overwrite Rule, $c_i <_c c_j$ follows. This, however, contradicts $c_j <_c c_i$ as required by $x_j \ll x_i$. Hence, $c_i <_c c_k$ cannot be true. By the Read/Write Lock Conflict Rule we know that either $c_i <_c c_k$ or $c_k <_c c_i$ must hold. Since we have just eliminated the first case, $c_k <_c c_i$ follows, as desired.

We have shown that all edges in $MVSG(H_c, \ll)$ are in certification order; and since the certification is embedded in a history which is acyclic by definition, $MVSG(H_c), \ll)$ is acyclic, too. $\square$

setcounterchapter6

Chapter 7

Performance Evaluation of Branching Transactions

The concept of branching transactions has been introduced to address the issue of data contention in a parallel database system. A simulation study which illustrates this problem for the conventional, flat transaction model (using two-phase locking concurrency control) and which investigates various performance aspects of branching transactions under a number of system and workload assumptions is described in this chapter.

If we were to introduce the concept of branching transactions into transaction management, it would influence system design in many ways, including concurrency control, recovery, deadlock handling, dynamic workload control and cache management. Indeed there are far too many aspects to include them all in a single performance study. The simulation study presented as part of this dissertation, therefore, focusses on branching transactions in a shared-memory parallel computer environment. Extending this work towards shared-disk and shared-something parallel computers — this would include an analysis of branching transactions using different cache coherence protocols and distributed lock management — is left as future work.

We begin our discussion with a description of the simulation model and some comments on the implementation of the simulation program, followed by an analysis of the data contention problem for traditional, flat transactions. Thereafter, the performance of branching transactions is discussed.

7.1 Simulation Model

A good simulation model should be general enough to be able to capture a variety of system scenarios, but must also be detailed enough to be a meaningful representation of a real system; we will say more about verification and validation of the model and its implementation later. The simulation model used for this study is similar to the one described by Agrawal et al. [5]. Unlike them, however, we draw the model in one single diagram, rather than describing it with the help of two separate diagrams for the logical and physical aspects (of the same queueing model), respectively. Not separating the two makes it easier to understand the life-cycle of a transaction. Figure 7–1 shows our simulation model. The flow of a transaction through this model is discussed next and the parameters which determine the transaction workload, the database system (including its hardware resources) and the workload control in operation are discussed.

The system is modelled as a closed queueing network where transactions are submitted by *NumTerm* terminals. After the completion of a transaction is reported to a terminal, there is a random delay — drawn from an exponential distribution with mean *ThinkTime* — before the next transaction is started. Each transaction accesses between $0.5 *$ *TransSize* and $1.5 *$ *TransSize* (uniform distribution) pages[1]. The probability that an access to the database is a Write operation is *WriteProb*. It takes *PageCPU* CPU time to process one page. The transaction workload parameters are summarised in Table 7–1.

[1]We assume that a page contains exactly one data item, and each item requires one page.

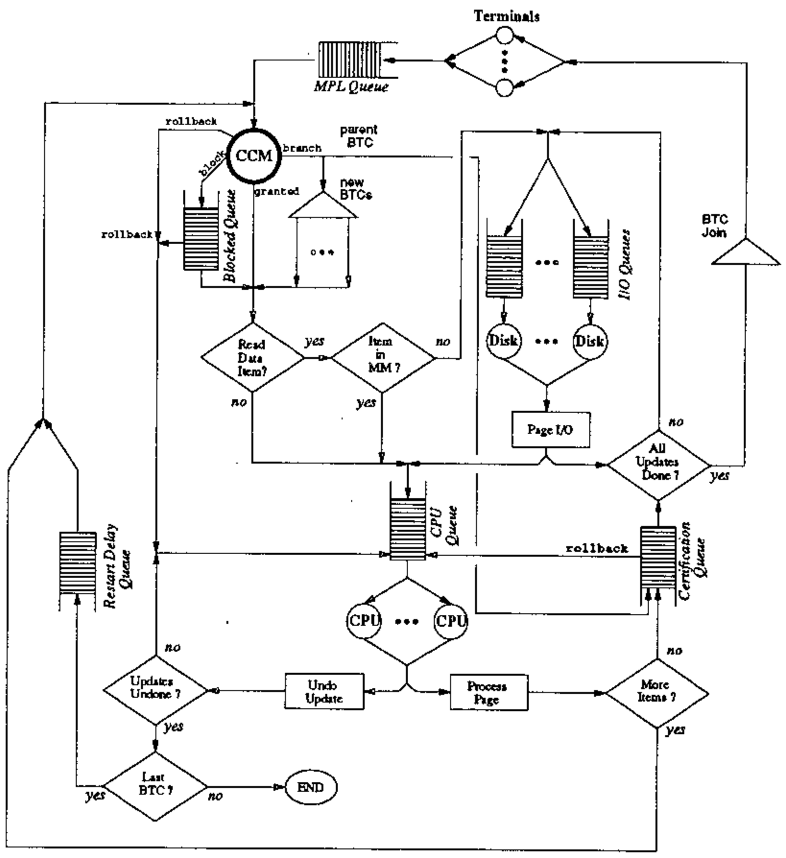


Figure 7–1: Basic Simulation Model

The workload in the system can be controlled via the multi-programming level, i.e. only *MPL* transactions are allowed to be active at any time[2]. If too many transactions are submitted by terminals, some are queued in the *MPL Queue*. The queue is serviced in first-come-first-served order. In case a branching mechanism is used, static and dynamic branching control is applied. Each transaction is limited to a maximum of *MaxBTC* BTCs (static control). Branching is dynamically turned off if the CPU utilisation exceeds *MaxCPU* and is only turned on again once CPU utilisation drops below *MinCPU*. Load control parameters are summarised in Table 7–2.

[2]To reduce the number of variable parameters in our simulation experiments, we keep the number of terminals small enough to avoid an overload of the system. Further load control via the MPL is, therefore, not applied; the MPL is set equal to the number of terminals in the system

Parameter	Meaning
NumTerm	number of terminals
ThinkTime	avg. think time between transactions
TransSize	avg. transaction size
WriteProb	write operation probability
PageCPU	CPU time for processing one page (item)

Table 7–1: Transaction Workload Parameters

Parameter	Meaning
MPL	multi-programming level
MaxBTC	maximum number of BTC per transaction
CPUHigh	upper CPU utilisation limit for dynamic branching control
CPULow	lower CPU utilisation limit for dynamic branching control

Table 7–2: Load Control Parameters

The size of the database is *DBSize* pages. Before every Read and Write access to the database, a BTC sends a lock request to the concurrency control manager. If the request is blocked, the BTC joins the *Blocked Queue*. It remains there until the lock can eventually be granted, or must be aborted. A BTC may also be directly aborted by the CCM (roll-back path in the diagram). If the system is in branching mode, a read lock request may lead to branching and new BTCs are created. These new BTCs have implicit read lock permission for the versions of items for which they were created (as described in Section 4.3). The original BTC, which has now been branched into several new ones, joins the *Certification Queue*.

After a BTC has acquired access permission to a data item and it wants to read a particular page, it is determined whether the required page is already in main memory. There is a *CacheHit* probability that this is the case. If the page must be fetched from disk, one of *NumDisks* disks is randomly selected; each disk is equally likely to be chosen (uniform access distribution). The BTC joins the *I/O Queue* for that disk and, once the disk is available, loads the page into main memory. The time it takes to transfer one page between main memory and disk is specified by parameter *PageIO.*

Before a BTC can process a page, it must be dispatched to a CPU. There is one queue for all CPUs and as soon as one of *NumCPU* CPUs is available, the first BTC in *CPU Queue* is removed from the queue and executed by this CPU. It takes *PageCPU* time units to process a page. Once there are no more items to be processed, a BTC joins the *Certification Queue*[3].

If a BTC joins the *Certification Queue* because it has indeed completed all operations, certification is initiated immediately. A BTC which has joined the *Certification Queue* because it has branched does not start certification until one of its children has been certified (for more detail on the certification process see Chapter 4). Before it can commit (after it has been certified), a BTC must flush all of its updates to disk (deferred update policy). For each page that was updated the BTC follows the same steps as for fetching a page from disk, except that *PageIO* now represents a main memory to disk transfer, rather than the other way round.

Once all BTCs of the committing branch of a transaction have been certified and their updates flushed to disk, the transaction commits and the end of this transaction is reported to the corresponding terminal. In other words, the completion of a transaction is only reported after all BTCs of the committing branch have "joined" again. In case of a non-branching transaction, there is, of course, only one BTC and once it has been certified and its updates have been flushed to disk, the transaction commits immediately (without having to wait for any other BTCs).

[3]This "Certification Queue", as well as the "Blocking Queue", is not strictly speaking a single queue. The time a BTC has to wait in this "queue" depends on when other BTCs complete their execution and when locks become available. The queue symbol is used, since some form of queueing for locks is involved.

When a BTC has to be rolled-back, either because the concurrency control manager has to resolve a deadlock or some branch of a transaction becomes obsolete or invalid, all updates performed in main memory by this BTC need to be undone. Rollback activities take priority over normal transaction processing in order to release locks as early as possible. Hence, an aborting BTC enters the *CPU Queue* in front of all BTCs scheduled for regular processing, but is queued after other aborting BTCs which are already in the queue. Once the BTC has acquired a CPU, it performs an undo operation. It takes *UndoCPU* CPU time per one undo operation. If the aborted BTC was the last of a transaction — the entire transaction has now effectively been aborted — the BTC joins the *RestartDelay Queue*, where it is delayed for a random period (based on a uniform distribution between 0 and *RestartDelay* $*$ *Current Average Response Time* time units). If there are still other active BTCs for the same transaction, the aborted BTC is simply discarded. Once a delayed BTC is restarted, it does not actually restart as the same BTC, but starts as a new initial BTC for this transaction; we do not really restart a BTC, but only a completely failed transaction. System parameters are shown in Table 7–3.

Parameter	**Meaning**
DBSize	size of database in pages
NumCPU	number of CPUs
NumDisks	number of disks
PageIO	I/O time for one page
UndoCPU	CPU time per undo operation
CacheHit	probability of cache hit
RestartDelay	multiplier for delay period before transaction restart

Table 7–3: System Parameters

As in Agrawal et al. [5], we do not explicitly account for CPU overhead caused by concurrency control operations. In fact, we do not explicitly simulate overhead caused by any DBMS background processes/threads. These costs are approximated by associating part of the CPU cost for processing a page with these overheads. Using this approach, we do take into consideration the fact that branching transactions may cause more system overhead than flat transactions; since the same page may be processed by more than one branch of a transaction, more system overhead is simulated for a branching transaction. For more detailed discussions of database operating system issues, such as system performance degradation due to increased multiprogramming levels (which lead to problems such as page thrashing due to memory shortage, and increased context switching) the interested readers are referred to [43].

The deadlock handling algorithm used in this study is Cautious Waiting [47]. As described in Section 4.5.3, under this policy, a BTC is aborted if it gets blocked by another BTC which belongs to a transaction which has all of its own BTCs either blocked, aborted or branched.

The actual input parameter values used are summarised in Table 7–4. In the descriptions of experiments that follow, it is assumed that the default values of these parameters are used, unless explicitly stated otherwise. For (non-branching) two-phase locking, *MaxBTC*, *CPUHigh* and *CPULow* do not apply.

For the model described above, a simulation program has been developed in Simula [72] using the simulation package DEMOS [14]. Implementation, validation and verification aspects are discussed in the following section.

Parameter	Value	
NumTerm	50	[terminals]
ThinkTime	variable (default: 2.5)	[sec]
TransSize	variable (default: 10)	[pages]
WriteProb	variable (default: 0.5)	
PageCPU	10	[ms]
MPL	50	[transactions]
MaxBTC	variable (default: 5)	[BTCs]
CPUHigh	variable (default: 100)	[%]
CPULow	variable (default: 0)	[%]
DBSize	1000	[pages]
NumCPU	variable (default: 5)	[CPUs]
NumDisks	4	[disks]
PageIO	35	[ms]
UndoCPU	1	[ms]
CacheHit	0.7	
RestartDelay	1	

Table 7–4: Simulation Input Parameter Values

7.2 Simulator Implementation

7.2.1 Extensions to DEMOS

During the implementation of a prototype simulator, various shortcomings of the simulation package DEMOS were identified. For the purpose of the full simulator these problems were resolved by adding extensions to DEMOS[4]. Each of these extensions and the reason why they were necessary are discussed next.

Hashtables

The programming language Simula provides explicit support for the implementation of linked lists. Linked lists are, however, rather inefficient when access to particular elements of a list — based on some key value of these elements — is needed; the list is searched sequentially. To avoid this problem in the simulator, two hash table classes were added to DEMOS: `class HashEntry` and `class HashTable`. A `HashTable` object can store any objects of classes which inherit from `HashEntry`; it supports the operations: `Store`, `Retrieve` and `Delete` for a given element key. Furthermore, it uses procedure `Cardinality` to return the number of elements in the table, and procedure `Print` to print all elements in the table. For the latter, each class stored in the table must contain its own print procedure, or else only the key value of each element is printed. Hashtables were used extensively in the implementation of the simulator, e.g. for lock tables.

7.2.2 Asynchronous Message Passing

Simula, as an object-oriented programming language, supports message passing between objects. On arrival of a message, an object invokes the appropriate method, and if there is a return value, it is passed back to the message sending object immediately. In this simulator, an immediate reply is not always possible. For example, an *access request* message from a transaction object to the concurrency control manager object may not have an immediate reply; an *access granted* message may be returned at some later time, e.g. when another transaction releases

[4]These DEMOS extensions were incorporated into the existing version of DEMOS by creating a new class, `class ExtDemos`, which inherits from `class Demos` and adds all new functionality locally. To use `ExtDemos`, simulation programs are now pre-fixed with `ExtDemos` instead of `Demos`.

certain locks. This kind of asynchronous message passing has no direct support in DEMOS. To alleviate this problem, a new class, `class ExtEntity`, has been implemented. `ExtEntity` inherits all features of `Entity` and adds asynchronous message passing functionality to it.

A message sent to an extended entity (an object of `class ExtEntity`) must be an object of a sub-class of `class Message`. To send a message to an extended entity the `PutMsg` method of this entity must be invoked; the message to be sent is an argument of `PutMsg`[5]. If this extended entity is currently waiting for a message, it is activated after the message has been delivered. If the receiver is not waiting for a message, the message is queued for later retrieval. To receive a message, an extended entity invokes the `WaitForMsg` procedure. If no message is currently in the queue the entity suspends execution until a message arrives. If a message is in the queue, it is immediately retrieved and the entity continues execution without delay.

A number of database management modules, e.g. the transaction manager and the concurrency control manager, are implemented as extended entities in the simulator and share the same basic structure, as shown in Figure 7–2.

Access to Individual Statistics

In DEMOS, classes which produce statistics, e.g. `Res`, `Accumulate`, etc., output these statistics in the form of a report. These reports contain various important values, such as mean values, maximum and minimum values and average length of a queue. DEMOS does not support individual access to these parameters, and as a consequence, it is extremely difficult to alter the form of output generated by simulation experiments. This in turn makes automation of further processing of simulation results virtually impossible. To overcome this problem, sub-classes of these DEMOS classes were implemented which support individual access to

[5]This implies that the sender of a message must know the reference to the receiving extended entity object.

```
while true do
begin
        WaitForMsg(Msg);
        inspect Msg
                when <message type 1>    do    ProcessType1(Msg)
                when <message type 2>    do    ProcessType2(Msg)
                ...
                when <message type n>    do    ProcessTypen(Msg)
        otherwise                              ErrorHandler;
end;
```

Figure 7–2: Basic `ExtEntity` Simulation Loop

statistics data. This is particularly helpful when simulation results need to be prepared for storage and further analysis in a database system (more on this later).

7.2.3 Validation and Verification

Validation of a simulation model and verification of its implementation are essential to ensure the usefulness of any simulation study. Ideally, one would like to validate a model against the real system of which it is an abstract representation. Unfortunately, there are no branching transaction systems available yet.

Although not as effective as the comparison with a real system, other methods of validation and verification can be applied. The following were used in this study:

Comparison with related research results: Data obtained for experiments with traditional transactions and strict two-phase locking were compared with extisting results in this area, and found to be consistent with findings by other researchers.

Input-Output Analysis: Varying the values of input parameters, relevant output parameters were checked for plausibility. For example, reducing the number of CPUs was expected to yield a higher CPU utilisation. Such plausibility checks were carried out for all input/output parameters.

Trace Analysis: Traces of message communication between major system components were analysed for correctness. For example, by looking at the communication between the concurrency control manager and the transaction manager one can determine if locking protocols and commit procedures were implemented as intended.

7.2.4 Database Support for Simulation Study

As mentioned earlier, it is desirable to store simulation results in a way which supports further analysis through other tools. All results of this study are stored in the object-oriented database system ObjectStore (from Object Design, Inc). Instead of writing simulation results directly into the database, however, the simulator stores all data in normal Unix text files first. A separate tool is then used to move these data into the database. The motivation behind this two-step process is to keep the simulator independent of the availability of the database, i.e. even if the database system is unavailable, the simulator can still be executed. A separate small filter program is used to export data from the database into a format suitable for direct use with some graphics package.

7.3 Simulation Results

In the remainder of this chapter various simulation experiments and their results are discussed. The first part deals with the traditional, flat transaction model only. Data from this part are used for validation and verification of the model and the simulator and for comparison with branching transaction experiments. The second part discusses experiments and results for branching transactions.

Each experiment was replicated (using different random number streams) until a confidence level of 90% for a confidence interval of ±10% of the output variable was reached. Each run was executed until 2000 transactions committed, where statistics were reset after the first 200 transactions to reduce the initialisation (warmup) problem.

7.3.1 The Data Contention Problem

The first set of experiments was carried out to illustrate the data contention problem using the "normal", flat transaction model and strict two-phase locking concurrency control. Data contention can be increased in several ways. In this experiment contention was varied by 1) changing the percentage of Write operations and 2) changing the transaction workload. The latter is increased by reducing the time between the end of a transaction and the submission of a new transaction from the same terminal.

As long as all transactions only read the database and no updates take place, no lock conflicts exist. In the absence of such conflicts one would expect the throughput of the system to increase with higher workloads, unless one of the hardware resources becomes a bottleneck. For this experiment the number of CPUs and disks, however, is large enough not to create this problem. As one would expect, in Figure 7–3 the highest throughput is, therefore, achieved for 0% Writes and the highest transaction workload.

One can also see how the increase of Write operations negatively impacts on transaction throughput. Clearly, more exclusive locks lead to more lock conflicts. The drop of throughput is more significant for higher workloads (shorter think times). One would expect such behaviour since the probability of lock conflicts is lower if fewer transactions are in the system.

Figure 7–4 shows average transaction response times for the same set of experiments. The most obvious observation to be made here is a significant increase in response time with a growing number of Write operations. Although not easily visible in this diagram, there is also a slight increase in response time for higher

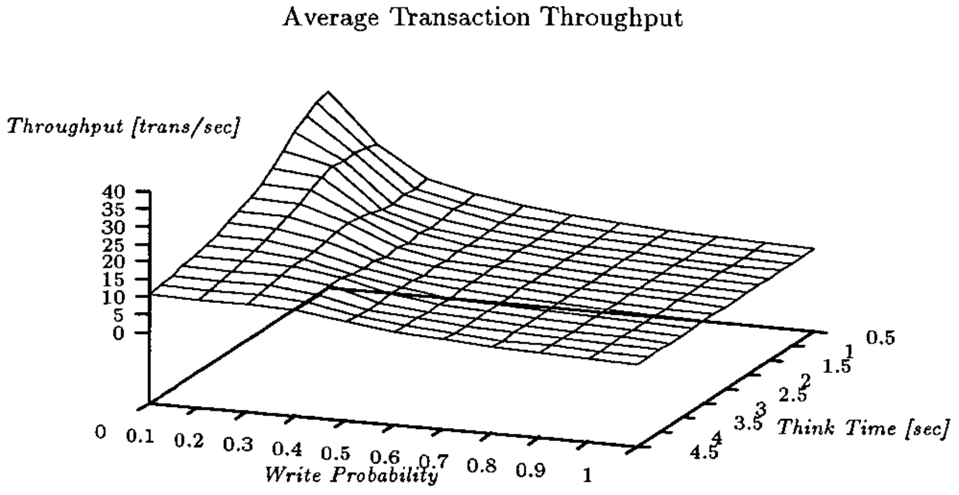


Figure 7–3: Transaction Throughput: 2PL

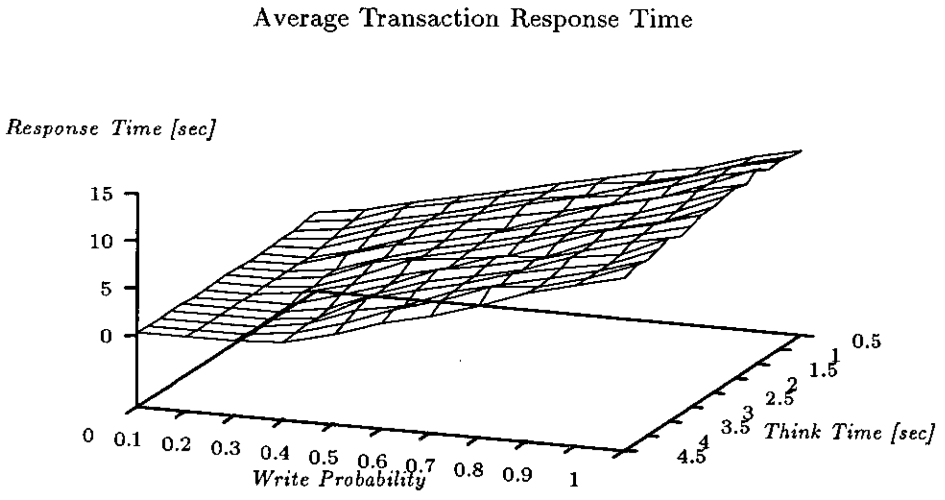


Figure 7–4: Transaction Response Time: 2PL

workloads. The dominating factor in this experiment, however, is the effect of lock conflicts due to the increasing number of exclusive locks needed for Write operations.

The effect of such data contention on the utilisation of CPUs — 5 CPUs were used for these experiments — is shown in Figure 7–5. Only for high transaction workloads with few Write operations was the CPU utilisation high. Since disk I/O is also assumed not to be a bottleneck, system performance must be largely determined by data rather than resource contention. If this is the case, one must assume that adding extra CPU resources to a system which is bound by data contention does not significantly improve performance.

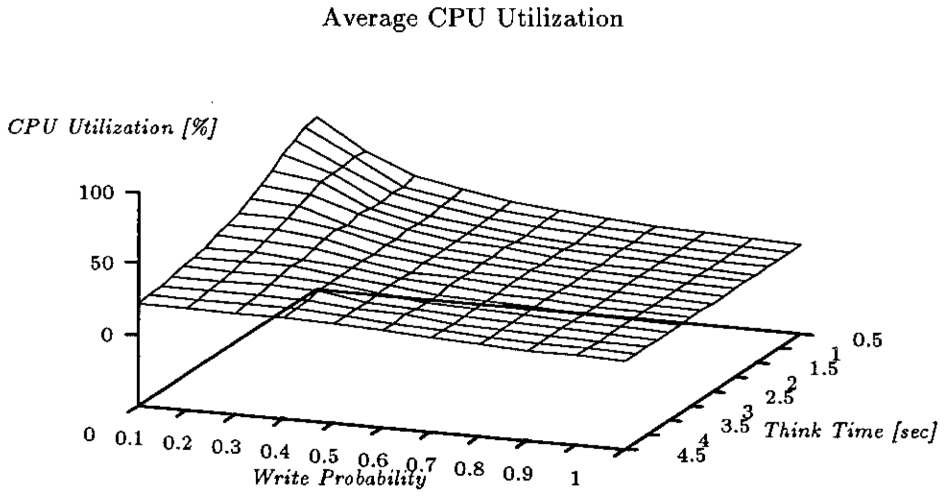


Figure 7–5: CPU Utilisation: 2PL

To confirm this hypothesis, a second set of experiments was carried out. This time the transaction workload was kept constant (*ThinkTime* = 2.5 sec) and performance measurements were taken for 1 to 10 CPUs in the system. Figure 7–6 shows the average transaction throughput for these experiments. For low Write probabilities, a considerable throughput improvement was achieved by adding one or two CPUs to a system with a single CPU. In all other cases, adding more CPU power did not significantly improve the performance.

A similar observation can be made for transaction response time (Figure 7–7). Adding one or two CPUs in a single CPU system shortens response time in case

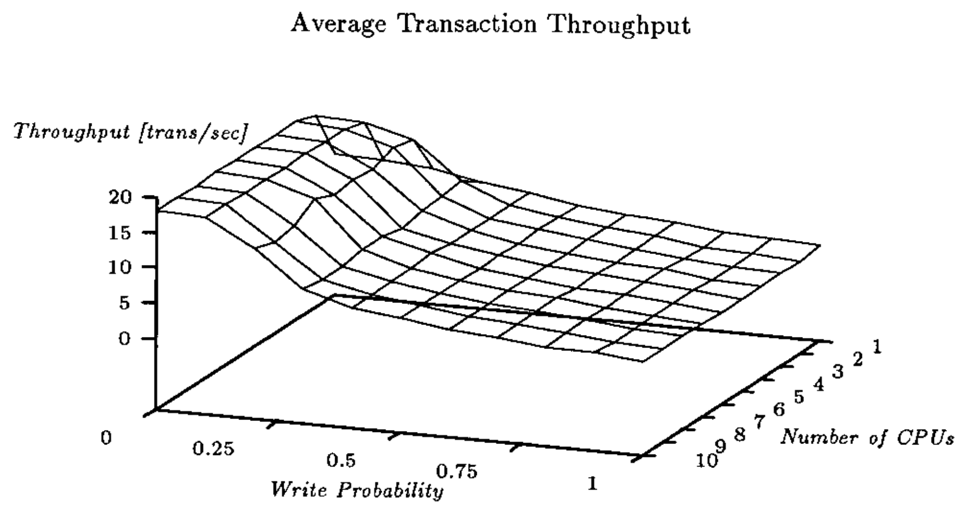


Figure 7–6: Transaction Throughput: 2PL (ThinkTime=2.5sec)

of low Write probabilities. In all other cases, adding more CPUs does not prove useful.

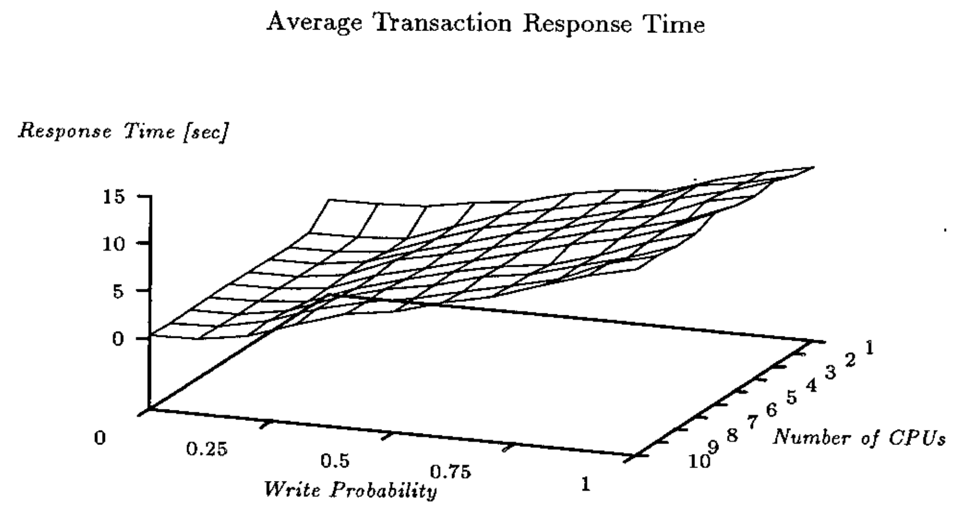


Figure 7–7: Transaction Response Time: 2PL (ThinkTime=2.5sec)

The average CPU utilisation for these experiments is shown in Figure 7–8. Changing from a single CPU to three CPUs leads to a significant drop in CPU utilisation for all Write probabilities. CPU utilisation drops below 50% when more than 5 CPUs are used, and below 25% for 10 CPUs. Considering these numbers and the fact that no significant performance improvement was achieved by increasing the number of CPUs beyond 5, we conclude that there is no significant resource contention for CPUs in systems with 5 or more CPUs.

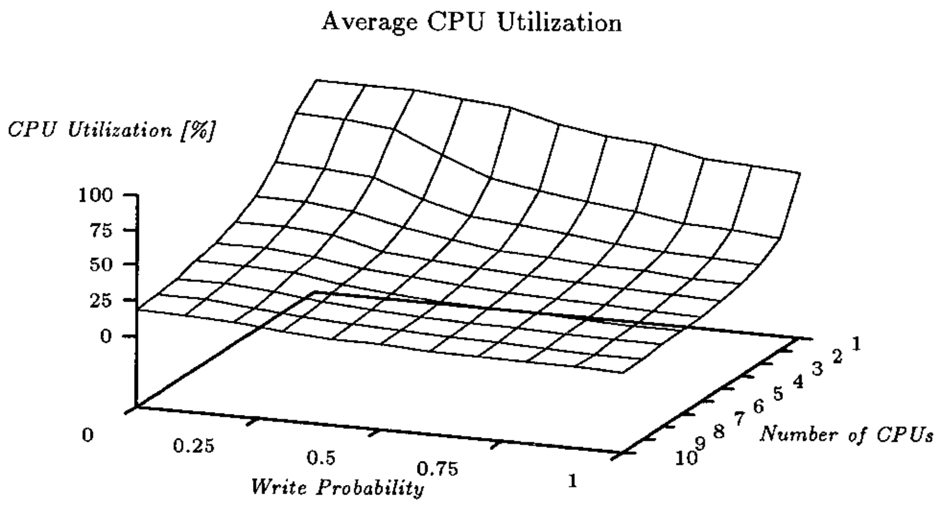


Figure 7–8: CPU Utilisation: 2PL (ThinkTime=2.5sec)

Comparison of Results with Related Work The basic conclusions drawn from these experiments are:

- if data contention is high, it may become the system bottleneck;
- in systems with high data contention, hardware resource utilisation is comparatively low;
- in systems where data contention has become the system bottleneck, adding more hardware does not improve performance;

These results are not new and similar conclusions have been reported by other researchers [5,6,21,37,48,86,90,88,89].

7.3.2 Experiments and Results for Branching Transactions

In the remainder of this chapter, we discuss the performance of branching transactions. The flat transaction model using strict two-phase locking is compared with branching transactions using HBT-MV2PL, the concurrency control algorithm introduced in Chapter 4. In particular, experiments were carried out to investigate the effects of increasing workloads, various Read/Write ratios of transactions, different numbers of CPUs and transaction size. For the branching transactions case, different branching control options are discussed as well.

Transaction Workload This first experiment measured throughput and response times for an increasing workload, where the workload was increased by reducing the *Thinktime* from 4.5 seconds to 0.5 seconds. Results are shown for 10% and 50% Write operations. The rest of the input parameters are set to their default values.

Figure 7–9 shows a clear performance advantage for HBT-MV2PL over 2PL in case of 50% Write operations. When only 10% of a transaction's operations were updates, no performance difference could be observed, except for high workloads, where HBT-MV2PL was again better than 2PL. The corresponding transaction response times are shown in Figure 7–10.

It is not surprising that virtually no difference in performance between 2PL and HBT-MV2PL was found in lower workloads with 10% Write operations. Since 90% of operations are Reads, which do not conflict with each other, there is only little data contention, and performance is mostly determined by the availability of hardware resources. The Read/Write ratio of transaction operations is obviously an important factor for the relative performance of HBT-MV2PL. The next experiment looks at this aspect in more detail.

Read/Write Ratio In this experiment the workload was fixed (*ThinkTime* = 2.5sec) and the percentage of Write operations varied from 0% to 100%, i.e. the

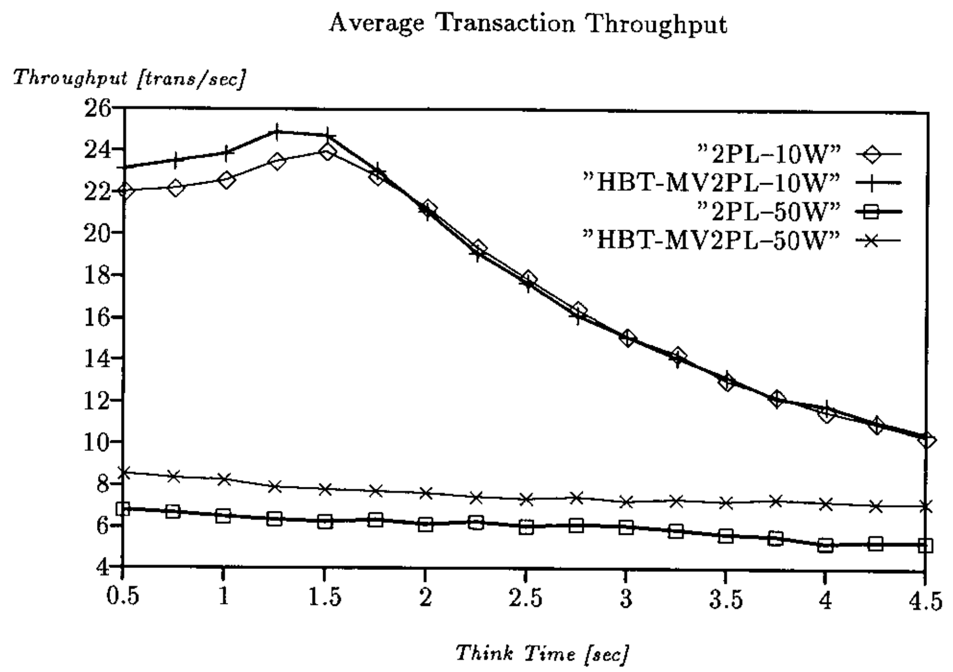


Figure 7–9: Transaction Throughput: 2PL vs HBT-MV2PL

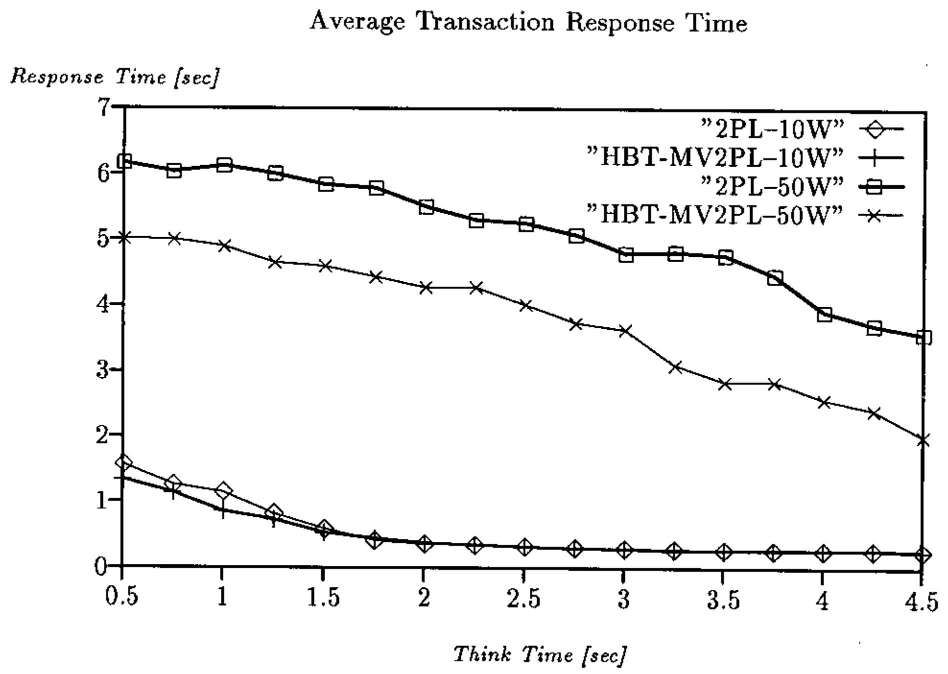


Figure 7–10: Transaction Response Time: 2PL vs HBT-MV2PL

Write probability was varied from 0.0 to 1.0. Results are shown in Figures 7–11 and 7–12.

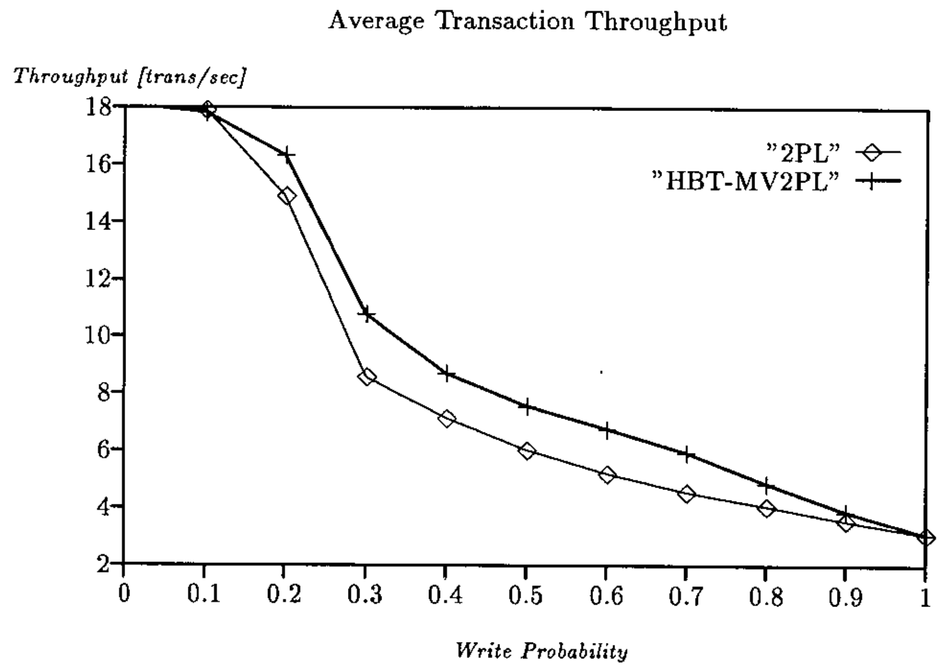


Figure 7–11: Transaction Throughput: 2PL vs HBT-MV2PL

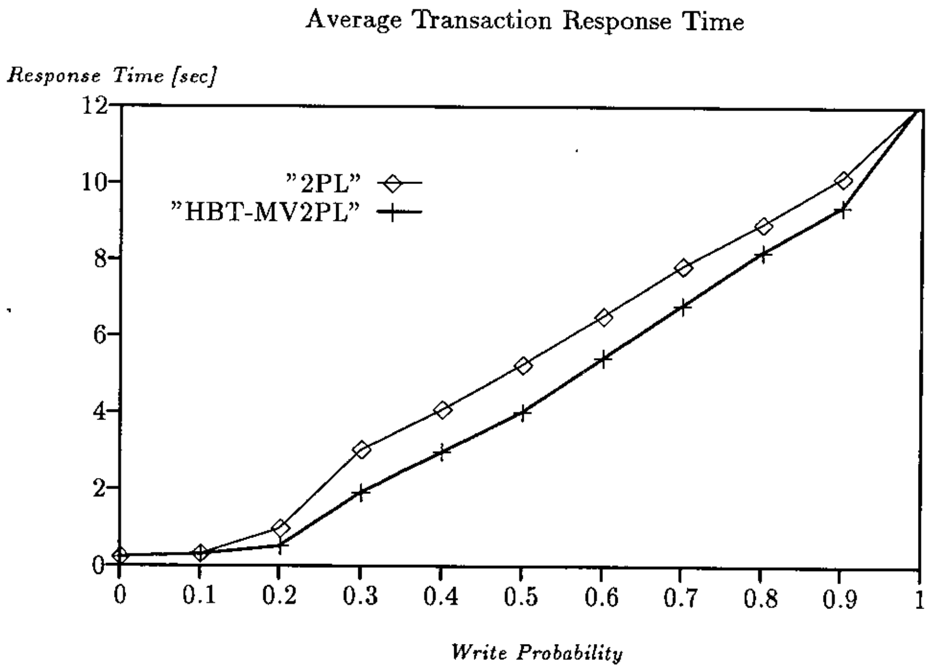


Figure 7–12: Transaction Response Time: 2PL vs HBT-MV2PL

As expected, there is no difference in performance between 2PL and HBT-MV2PL for a Read-only environment since no data access conflicts occur. Under both algorithms each Read-lock request is granted immediately; since no Write operations are executed, no exclusive Write-locks ever exist on any data item. Furthermore, no branching takes place, since no temporary, uncommitted versions of data items are ever created.

As the Write probability increases, HBT-MV2PL gains a performance advantage over 2PL. The improvement, however, is getting less once the probability of a Write operation exceeds 0.5, and is entirely lost in a Write-only environment.

To understand this behaviour it is important to point out that Write operations are modelled as so-called "blind writes", i.e. the item which is updated is not read before-hand. This means that in the Write-only case, no Reads are executed at all. Since branching transactions only branch on Read operations, no branching takes place at all for 100% Write, and hence, no improvement of performance can be achieved.

Since the biggest performance improvements are achieved for the 50/50 Read/Write ratio, one would also expect to find most branching activity in this area. This is indeed the case as can be seen from Figure 7–13. Furthermore, as one would expect, more branching takes place for higher workloads, since Read/Write conflicts are more likely with more transactions in the system.

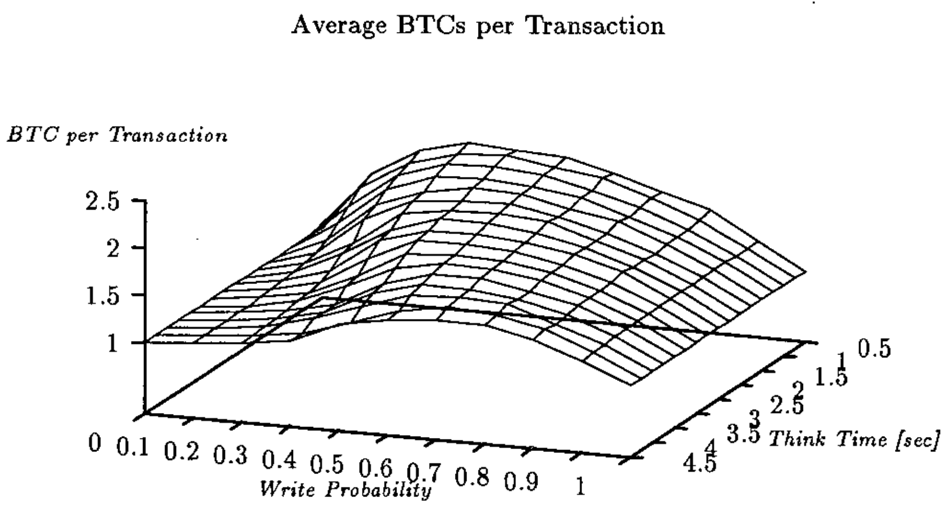


Figure 7–13: Average Number of BTCs per Transaction

Number of CPUs Since branching transactions compute several alternatives of transactions in parallel and all but one are eventually discarded, we expected a significantly higher utilisation of CPUs when branching transactions are used. As a consequence, it was also expected that non-branching two-phase locking would outperform HBT-MV2PL in experiments with only one or two CPUs available in the system.

To test this hypothesis, an experiment was carried out under which the number of available CPUs was varied from 1 to 10. All other parameters were fixed (*ThinkTime* is 2.5 seconds, *WriteProb* is 0.5). The results for response time and transaction throughput are shown in Figures 7–14 and 7–15.

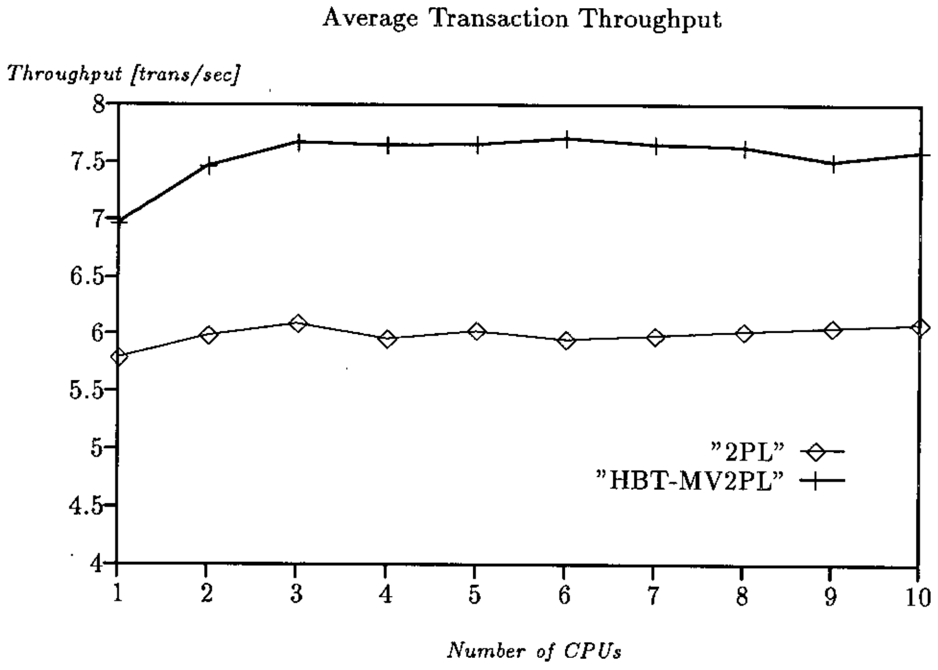


Figure 7–14: Transaction Throughput: 2PL vs HBT-MV2PL (ThinkTime=2.5sec, WriteProb=0.5)

Contrary to our expectations, even when CPU resources were sparse[6], branching transactions achieved a performance advantage over the non-branching ap-

[6]During this set of experiments, the only branching control mechanism applied was a maximum of 5 BTCs per transaction, i.e. we allowed transactions to branch in spite of high CPU utilisation. The effect of branching control is discussed later in this chapter.

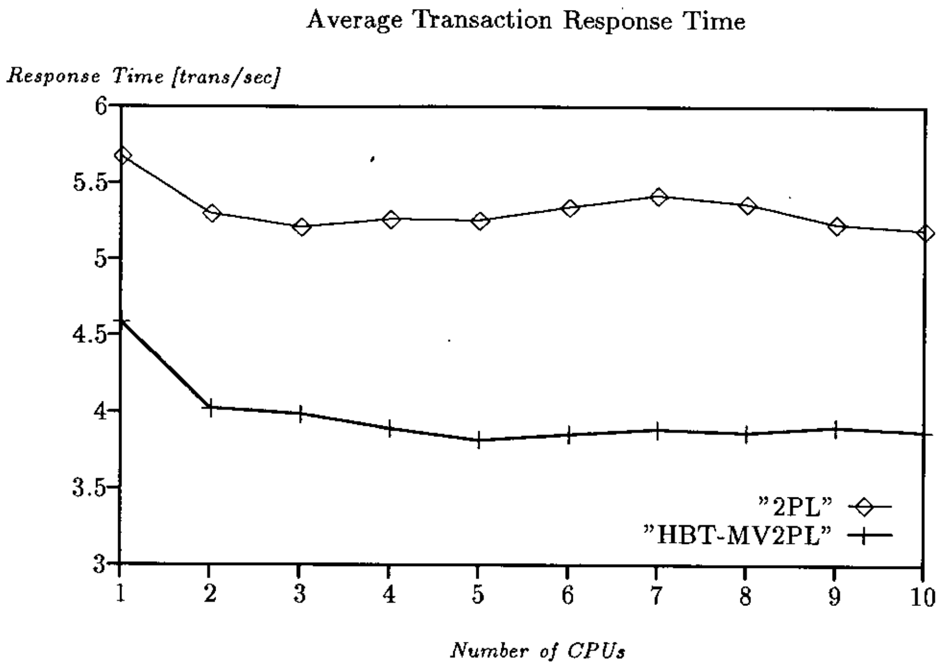


Figure 7–15: Transaction Response Time: 2PL vs HBT-MV2PL (ThinkTime=2.5sec, WriteProb=0.5)

proach. The explanation for this result lies in a wrong assumption that was made for the original hypothesis: under the same level of data contention, the branching transaction model leads to a significantly higher CPU utilisation due to its branching activities, and hence, CPU shortage is a much more severe problem for branching transactions. As shown in Figure 7–16, this turns out not to be the case, i.e. even though the CPU utilisation is somewhat lower for the non-branching case, the difference is not significant enough to offset the performance advantage of branching transactions.

The unexpected, relatively high CPU utilisation of (non-branching) two-phase locking under high data contention can be explained by the choice of deadlock handling strategy (Cautious Waiting) used in all experiments, which is known to produce more transaction restarts than a wait-for-graph based approach would do.

Transaction Size In this experiment, the effect of transaction size was considered. While keeping all other parameters fixed (*ThinkTime* is 2.5 seconds, *WriteProb* is 0.5) and *NumCPU* is 10), the average transaction size was varied from 6 to 16 average page accesses per transaction. As shown in Figures 7–17 and

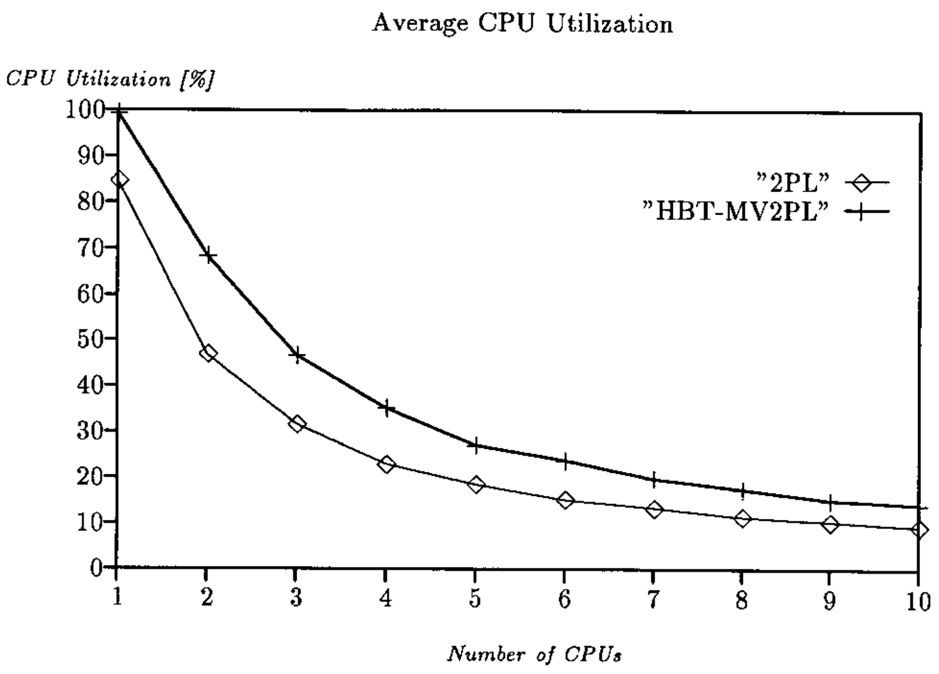


Figure 7–16: Average CPU Utilisation: 2PL vs HBT-MV2PL (Think-Time=2.5sec, WriteProb=0.5)

7–18, branching transactions performed better than its non-branching alternative for all transaction sizes tested.

It was originally expected that the performance advantage of branching transactions becomes more significant when the average transaction size is increased. To abort and restart a short transaction is less wasteful than to abort and restart a larger transaction. Hence, to be able to avoid unnecessary restarts through branching transactions should be more beneficial for larger transactions. This could not be confirmed by our simulation results. Yet again it appears to be the case that an increase of transaction restarts limits the improvements that can be achieved through branching transactions.

To see how Cautious Waiting deadlock prevention might be a problem for branching transactions, consider the following simple example. After transaction T_1 has created a new version of item x, transaction T_2 wants to read x, and therefore, $BTC_{2,1}$ branches into $BTC_{2,2}$ and $BTC_{2,3}$ reading the original value of x and the newly created one, respectively. Now assume that $BTC_{2,3}$ has finished execution and tries to become certified. If $BTC_{1,1}$ is blocked and has not been

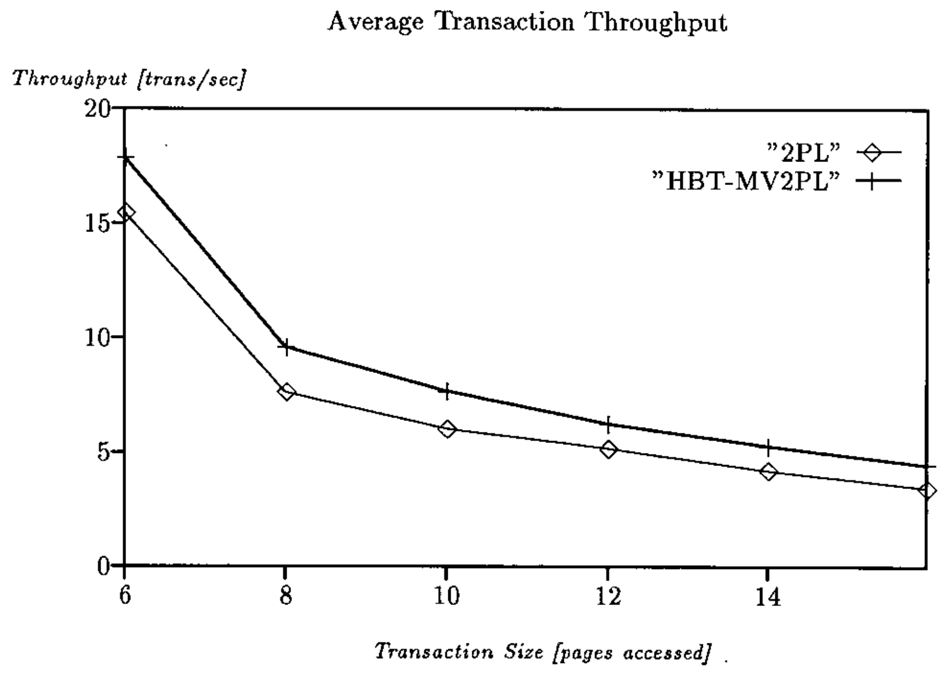


Figure 7–17: Transaction Throughput: 2PL vs HBT-MV2PL (Think-Time=2.5sec, WriteProb=0.5, NumCPU=10)

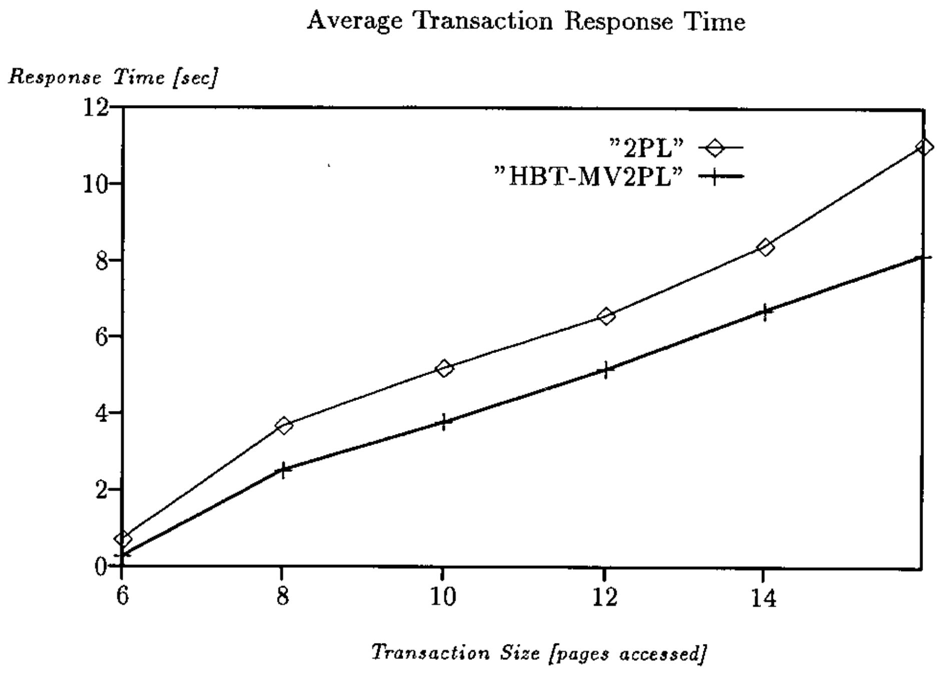


Figure 7–18: Transaction Response Time: 2PL vs HBT-MV2PL (Think-Time=2.5sec, WriteProb=0.5, NumCPU=10)

branched, then $BTC_{2,3}$ is aborted under Cautious Waiting. If, however, $BTC_{1,1}$ eventually commits, $BTC_{2,2}$ has to be aborted as well. In this case, all branches of T_2 are aborted and it needs to be restarted. The key problem here lies in the fact that what would have been the correct path of execution ($BTC_{2,3}$) has been aborted by Cautious Waiting deadlock prevention.

Branching Control Policies For all experiments carried out the maximum number of BTCs allowed per transaction was 5. Increasing this number did not lead to any performance advantages. Lifting *all* branching restrictions did not lead to an explosion of the number of average BTCs, but stayed about the same as reported earlier. This may not be that surprising, considering the fact that a (non-branching) system may suffer from data contention, even though the average queue length for a page lock is no more than 1. A further consideration is the effect of an increasing number of transaction aborts (see below) which is likely to limit the amount of branching that takes place. Increasing data contention between transactions, e.g. through a decrease in database size or an increase of transaction size, did not change this observation, i.e. a relatively low number of BTCs is not due to low data contention, but suspected to be related to issues of deadlock handling, as discussed next.

Deadlock Handling From the discussion so far it seems that the level of transaction aborts in the system is an important factor in the relative performance of branching and non-branching transactions. Furthermore, we believe that transaction aborts have a limiting effect on how much branching takes place in the system. Since we are not simulating user induced transaction aborts or system crashes, all transaction abort decisions are due to the deadlock handling policy applied.

As described earlier, the deadlock handling mechanism used in this study is Cautious Waiting, a mechanism which is more likely to cause transaction aborts than a wait-for-graph based approach. It would be of interest to carry out a comprehensive study of the effects of various deadlock handling algorithms on the performance of branching transactions. Such a detailed study of deadlock handling algorithms is, however, beyond the scope of this dissertation, and left for future work.

Note on Performance of Multi-Version Two-Phase Locking In our study, we concentrated on a comparison between (single-version) two-phase locking and HBT-MV2PL; non-branching multi-version two-phase locking (MV-2PL) was not included. Although performance improvements have been shown for MV-2PL over 2PL, most of these studies were particularly interested in a workload environment where short update transactions are mixed with longer read-only transactions, and it is there where multi-version algorithms (MVTO and MV-2PL) achieved significant performance improvements [82,23]. It is, therefore, no surprise that current DBMS systems which provide some form of multi-version concurrency control do so only for read-only transactions, i.e. only read-only transactions are allowed to see older versions of data items.

Carey and Muhanna [23] observed that MV-2PL did not lead to significant performance improvements over 2PL when the workload only consisted of transactions which read as well as updated the database; this is the type of workload which was of particular interest in our own study. Our own simulation data of MV-2PL vs 2PL confirmed Carey and Muhanna's result. In our simulation study, we found that although a transaction reaches the end of execution faster under MV-2PL, the delay imposed on transactions during certification compensates this advantage, so that in the end the response time (and throughput) of transactions under MV-2PL was approximately the same as under 2PL.

In summary, while MV-2PL is of interest for certain types of workloads — where large read-only transactions are mixed with short update transactions — branching transactions are aimed at OLTP applications where such read-only transactions do not play a significant role. It is for this reason that we do not include MV-2PL in above discussions of the performance of HBT-MV2PL.

7.4 Performance Study Conclusion

A number of important conclusions can be drawn from the experiments carried out:

- branching transaction can achieve performance advantages over non-branching systems over a variety of parameters, although there is no significant difference in performance between a branching and non-branching system if data access conflicts are rare;
- even relatively small levels of branching can achieve performance improvements;
- best performance improvements are achieved for a 50/50 mix of Read and Write operations;
- contrary to our expectations, the problem of exponential growth of BTCs was not observed in any of the experiments;
- future work should include a comprehensive study of the effects of deadlock handling strategies on the behaviour of branching transactions;

Chapter 8

Real-time Scheduling with Branching Transactions

In real-time database systems, transactions must be executed within certain timing constraints. It is more important that most, or preferably all, transactions are executed before their given deadlines than to achieve high transaction throughput and fast average response times. Traditional scheduling strategies are not well suited to cope with these timing constraints, and hence, new algorithms for real-time scheduling have been developed; two good overview papers in this area are Abbott and Garcia-Molina [1] and Yu et al. [100]. A comprehensive annotated bibliography on real-time databases can be found in [92].

In this chapter, we will show how branching transactions can be used in the context of real-time scheduling; we propose a new real-time concurrency control algorithm based on branching transactions, and illustrate it with the analysis of an extensive example. In order to provide the reader with some background in this field, we begin this chapter with a summary of various aspects of real-time scheduling.

8.1 Introduction to Real-time Transaction Scheduling

Real-time transactions have deadlines by which their execution must be complete, or else they are of no or little help to the user who initiated them. Abbott and Garcia-Molina [1] use a stock market information system as an example for real-time transactions: at any point of time the databases used for such a system must contain a sufficiently accurate representation of the current stock market situation. Hence, any updates to the database must be performed within rigid time constraints. Furthermore, a user query asking for the prices of a particular stock needs to be answered within a few seconds to be of any use. Another example is an arbitrage trading program which tries to find price discrepancies for objects, often on different markets. Since such price discrepancies are usually short-lived, the detection and exploitation of arbitrage opportunities must be done within very strict timing constraints. Other application areas include computer aided manufacturing, process control and radar tracking systems.

Depending on the type of application, late completion of a transaction may result in a number of problems: from loss of money (stock market) to possibly life threatening situations (late radar detection of an incoming enemy missile). In general, we distinguish between *soft* and *hard deadlines.* If a transaction has a soft deadline, then there is still some benefit, though a diminished one, in completing this transaction after its deadline has passed. A hard deadline means that there is no benefit in completing a transaction which is too late; once a transaction has passed a hard deadline, it is aborted.

The primary performance metric applied to real-time scheduling with hard deadlines is the percentage of transactions missing their deadlines; the smaller this percentage, the better. In systems where transactions have soft deadlines, the mean tardy time — the amount of time a transaction is late, i.e. completed after its deadline — of transactions is also an important performance indicator; again, the smaller this metric, the better the performance.

The problem of scheduling real-time transactions can be subdivided into four sub-problems:

- managing overloads
- assigning priorities to tasks
- I/O scheduling
- concurrency control

The first three of these problems are discussed in the following paragraphs. A separate section is dedicated to the issue of concurrency control; we discuss concurrency control in more detail, since it is there where we propose a new algorithm (based on branching transactions).

Managing Overloads: A system where transactions are missing their deadlines is said to be *overloaded.* Such an overload may cause many transactions to be tardy and it is better to abort some of the transactions in the system for the benefit of others being able to finish on time. Hence, a real-time scheduler must apply some policies for detecting and managing such overloads.

An overload can be detected by checking — either periodically or whenever the scheduler is called — whether all active transactions are still within their deadlines. Transactions which are too late are aborted. This method is called *observant*, since it checks (observes) if any real deadline misses occurred, unlike in so-called *predictive* methods, where the scheduler estimates for each transaction how much time it still needs to complete, and if that estimate indicates that a transaction will miss its deadline, it is aborted.

Assigning Priorities to Transactions: Unless transactions are given priorities, they are assigned CPUs on a *first come first serve* basis. This is not very suitable in a real-time scheduling environment, since it does not take into consideration the deadlines of transactions.

An alternative is to give the highest priority to the transaction with the *earliest deadline*. If no preventive steps are taken, however, this approach may give resources to transactions which have already passed their deadlines or which have no chance of meeting their deadlines. Some form of overload management should be applied to avoid this problem.

Even though a transaction may have an earlier deadline than another, there may be only little work left for the transaction with the earlier deadline, and it may be better to give priority to a transaction which has a later deadline, but needs a lot more processing before it can complete. The *least slack* approach addresses this issue. The slack time of a transaction is the estimated acceptable delay time in spite of which the transaction can still meet its deadline. The transaction with the least slack time is given highest priority, unless the slack time is negative, in which case the transaction has either already missed its deadline or is unable to meet its deadline, and should be aborted.

I/O scheduling: Traditionally, the I/O system uses a disk scheduling algorithm which minimises the disk seek time by ordering I/O requests according to the position of the requested data items on disk. While this helps to improve the average throughput of the I/O device, it may cause requests from high priority transactions to be ordered after those from lower priority transactions. The disk scheduling algorithm in a real-time environment may have to order I/O requests according to their priorities, even so this may not lead to the best possible overall throughput.

8.2 Concurrency Control for Real-time Scheduling

Database consistency is also an issue for real-time database systems, and hence, if transactions are to be executed concurrently, some form of concurrency control is required. Although the usual techniques (e.g. two-phase locking) can be used, they do not consider the particular real-time constraints of transactions and the problem of priority inversion may occur. (For the remainder of this chapter we will use HPT instead of high priority transaction and LPT instead of low priority transaction.)

Priority inversion exists if an LPT delays the execution of an HPT . For example, under two-phase locking, if an LPT holds a Write lock on item x and an HPT requests a Read lock on x, the HPT is blocked by the LPT. The following paragraphs describe various algorithms which were proposed to address the issue of priority inversion.

Wait Promote: Under this policy, whenever an HPT is blocked by an LPT, the LPT's priority is promoted to the priority of the HPT. Since most systems apply strict 2PL (locks are not released until a transaction terminates), the LPT keeps the higher priority until either it or the HPT terminates. In the latter case, the priority of the LPT is set back to its own value.

Since the LPT is given a higher priority it is able to complete faster, and hence, release its locks earlier. HPT's waiting time is thereby reduced; only transactions with a higher priority than HPT can preempt the CPU from LPT, or have their I/O scheduled before that of the LPT.

The problem with Wait Promote is that the HPT still needs to wait for the LPT. Furthermore, if there are many data conflicts, too many transactions' priorities are promoted and most of the transactions are executed under high priority; the meaning of priorities becomes highly distorted.

High Priority: The basic idea of this scheme is always to grant a lock to the transaction with the higher priority. If the lock holder has a lower priority than the requester, the holder transaction is rolled-back. If there are multiple holders and the requestor's priority is higher than the priority of each holder, the requestor will be granted the lock and all holders are aborted, otherwise the requestor transaction is blocked.

Abbott and Garcia-Molina [1] describe an interesting problem if High Priority is used together with the Least Slack priority scheme: since the aborted transaction will need to start execution from the beginning, its slack time will be smaller, and consequently, its priority higher. This may lead to an abort of the transaction with which it had a conflict when it was aborted. In turn, if that transaction is now aborted, it too will be restarted with a higher priority. To avoid this problem, a slightly modified version of the algorithm requires the requestor transaction to have a higher priority than the holder transaction would have if it were to be aborted, or else the holder is not aborted.

High Priority can be too strict in preventing priority inversion: it may abort an LPT — thereby losing all the work the LPT has done so far — in favour of an HPT, even though both transactions would have been able to complete within their given deadlines in spite of the HTP being blocked for some time. The following policy is designed to take account of such situations.

Conditional Restart: Under this scheme, the scheduler tries to prevent the abortion of an LPT if there is a chance for both the LPT and the conflicting HPT to complete their execution in time. Whenever an HPT request conflicts with an LPT holder, an estimate is made as to how much longer the LPT needs to complete its execution. If it is shorter than the slack time of the HPT, the HPT is blocked. The reasoning here is that there might still be enough time left for the HPT to meet its deadline in spite of being blocked by the LPT. While "holding up" an HPT, the LPT is running under the priority of HPT (as was the case in Wait Promote).

Multi-version Data Algorithms: Kim and Srivastava [50] have suggested real-time concurrency control algorithms based on two-version and multi-version two-phase locking. Since our own algorithm is based on multiple rather than only two versions, we will only discuss their multi-version algorithms. (For a reminder of multi-version two-phase locking (MV-2PL) see Section 4.2.)

In Kim and Srivastava's paper, a distinction is made between direct and indirect priority inversion. Direct priority inversion occurs if an HPT requests a lock on an item which is already locked (in an incompatible mode) by an LPT. For example, an LPT with a Certify lock blocks a Read lock from an HPT. Indirect priority inversion takes place if an LPT holds a Read lock and an HPT requests a Write lock (on the same item), or an HPT holds a Write lock and an LPT requests a Read lock (on the same item). In both cases, although the Read and Write locks are not conflicting, the HPT won't be able to upgrade its Write lock to a Certify lock during certification, unless the LPT has terminated before. Two algorithms were suggested to deal with this problem: 1) Unconditional multi-version 2PL, and 2) Conditional multi-version 2PL.

Unconditional Multi-Version 2PL (UMV2PL): In case of direct priority inversion, an LPT is aborted in favour of an HPT, unless the conflict can be resolved by demoting an LPT's Certify lock to the Write lock it initially was. The indirect priority inversion problem is handled by aborting an LPT with a Read lock on a data item if an HPT requests a Write lock on the same item. Also, if an HPT holds a Write lock on an item, any Read lock requests for this item from LPTs must wait. Since no HPT will ever wait for an LPT, deadlocks cannot occur[1].

Conditional Multi-Version 2PL (CMV2PL): For similar reasons that lead to the introduction of Conditional Restart — aborting a nearly finished LPT is wasteful and should be prevented, if possible — UMV2PL has been modified to include a conditional restart policy. The basic mechanism of CMV2PL is the same as for UMV2PL, except that if an LPT has already entered its Certify phase, it is in general not aborted when in conflict with an HPT. Since the LPT is expected to complete execution in the near future, a short delay of an HPT is considered acceptable, unless the HPT itself has already entered the Certify phase or there

[1]A deadlock can only happen if there exists a cyclic wait-for dependency, in which case at least one HPT must be waiting for an LPT.

exists a deadlock involving both, the LPT and the HPT. In these cases, it is still the LPT which is aborted; an HPT is never aborted[2].

This list of real-time concurrency control algorithms is by no means complete. For an overview of such algorithms and references to other relevant publications, we refer the interested reader to Yu et al. [100] and O. Ulusoy [92].

8.3 Real-time Concurrency Control with Branching Transactions

Having discussed some of the issues and algorithms for real-time scheduling of non-branching transactions, in this section we will describe the situation of branching transactions in a real-time environment. We begin by showing under what circumstances priority inversion occurs and how our existing algorithms must be modified to avoid priority inversion when using branching transactions.

8.3.1 Priority Inversion under HBT-MV2PL

Under HBT-MV2PL, the concurrency control algorithm presented in Section 4.4, a transaction, say T_r, is blocked by another transaction, say T_h, if

1. T_r has read an item which was written by T_h and T_h has not yet committed, or

2. T_r requests a lock on an item which is not compatible with an already existing lock on this item held by T_h. This is true, if

   (a) T_h holds a CRL and T_r requests a CWL (unless we have the case where an alternative sibling BTC for T_h exists; for details see Section 4.4), or

[2]Of course, if an HPT is involved as an LPT in another conflict, it may be aborted.

(b) T_h holds a CWL and T_r requests a CRL or a CWL;

If T_h is an LPT and T_r is an HPT, we have a priority inversion situation.

It should be pointed out that we do not distinguish between direct and indirect priority inversion. We disregard indirect priority inversion until an upgrade attempt for a VWL (to CWL) is made, in which case it becomes direct priority inversion. Kim and Srivastava [50] take a more conservative view of the problem: they take action based on the assumption that an indirect priority inversion situation will eventually lead to the blocking of an HPT by an LPT. This, however, is only true if the HPT enters certification before the LPT terminates.

8.3.2 Real-Time Concurrency Control Algorithms for Branching Transactions

Any of the previously discussed solutions for priority inversion can also be used for branching transactions:

Wait Promote: a low priority BTC blocking a high priority BTC has its priority raised;

High Priority: a low priority BTC blocking a high priority BTC is aborted;

Conditional Restart: the restart of a low priority BTC is avoided if the branch of the transaction to which it belongs is estimated to finish execution within the slack time of the high priority transaction whose BTC is blocked;

Multi-version Algorithms: the rules layed out for transactions can be applied to BTCs instead;

None of above techniques, however, take advantage of the particular characteristics of branching transactions.

Unless a system operates mostly in 2PL mode, the majority of lock conflicts occur when a transaction is trying to become certified; before certification mostly version locks (VRLs, TCRLs and VWLs) are acquired, and version locks do not conflict with each other. To keep transactions with later deadlines from blocking transactions with earlier deadlines we delay certification until shortly before a transaction's deadline. Since transactions with earlier deadlines have already committed or aborted, their locks are already released[3], and hence, they should not conflict with the certification of transactions with later deadlines.

Although many conflicts between transactions should be avoidable this way, some situations cannot be resolved that easily. Assume that two transactions, T_{early} and T_{late}, are accessing the same data item x. T_{early}'s deadline is before T_{late}'s, and T_{early} wants to update x, while T_{late} wants to read it. During the actual execution of these transactions the Read operation precedes the Write. Since, however, we are trying to produce a schedule which is equivalent to a serial execution in which T_{early} precedes T_{late}, T_{late} should read the version of x created by T_{early}. To achieve this we spawn a new branch of T_{late} which reads the copy of x produced by T_{early}. To improve the chances that this new branch of T_{late} completes within the given timing constraints, it should be created at the time T_{early} created the new version of x, rather than at T_{early}'s certification time. The example that follows later in this chapter will illustrate such a scenario.

Description of BT Real-time Concurrency Control Algorithm: When a BTC wants to read a data item, it branches if there exist versions of the requested item which were created by BTCs which have an earlier deadline than the requesting BTC. There is no need to branch for versions with a later deadline, since we only allow schedules which are equivalent to a serial execution based on transaction deadlines. TCRLs and VRLs are set as in the original BT-MV-2PL algorithm. If no branching is required, the requesting BTC simply reads the last

[3]This is only true, if there is enough time between the deadlines of two conflicting transactions to release locks of the earlier and certify the later transaction. If this is not the case, the second transaction will be slightly tardy.

committed version of the item, and an appropriate TCRL is set. This handling of Read requests in summarised in Figure 8–1.

```
Read Request(x):

if uncommitted versions of x exist:
    • branch requesting BTC (one new branch
      for the committed version of x and one
      for each uncommitted version which was
      created by a transaction with an earlier
      deadline than the requestor)
    • set TCRL and VRLs
else
    • read committed version of x
    • set TCRL
```

Figure 8–1: Read Request Algorithm

Every Write request creates a new version of the corresponding item, and a VWL is set accordingly. At this point we may face the situation described above: a BTC with a later deadline has already read the same item, and for both transactions — the current writer and the previous reader — to commit, the reader must have read the version just created by the writer BTC. To achieve this, we create a new branch for the reader BTC, just as if the new version had already existed at the time of the original Read request. We. do not abort any of the previously created branches for that reader, since we may need them in case of an abort of the BTC which just created a new version of the item. (Figure 8–2 shows this processing of a Write request.

```
Write Request (x):

• create new version of x
• set VWL on new version
for all transactions which hold TCRLs and
    have later deadline than requestor:
    • create new BTC
    • set VRL on new version (for new BTC)
```

Figure 8–2: Write Request Algorithm

When a transaction nears its deadline, it tries to certify those branches which have completed execution. To certify a branch, all its BTCs must be certified. Since a transaction only reads data written by transactions with earlier deadlines, and certification is delayed until shortly before a transaction's deadline, any data read by the transaction to be certified should by now be committed or aborted (except for when two deadlines are too close, as noted earlier). In the latter case, an abort of the BTC which we are trying to certify has already been initiated.

Since only one transaction at a time can be in its certification phase, there can be no conflicting CWLs or CRLs on any data item at any time, and hence, certification of a BTC merely consists of upgrading all its TCRLs to CRLs and all its VWLs to CWLs (as shown if Figure 8–3).

```
Certify Request:
  • upgrade all TCRLs to CRLs (no blocking)
  • upgrade all VWLs to CWLs (no blocking)
```

Figure 8–3: Certify Request Algorithm

Although there are no conflicting CRLs and CWLs at certification, TCRLs from other transactions may exist. Following the rules of BT-MV-2PL, any BTC holding such a TCRL must be aborted once the holder of a conflicting CWL commits. Hence, at commit time, our algorithm aborts all BTCs with such conflicting TCRLs before it releases the locks for the committing transaction (Figure 8–4).

```
Commit Request:
for all data items where committing BTC holds a CWL
    • abort BTCs which hold TCRLs
• release locks of committing BTC
```

Figure 8–4: Commit Request Algorithm

Except for a small delay of the certification of a transaction because a previous transaction with an earlier deadline has not yet released its locks, no transaction is ever blocked due to lock conflicts. Hence, there is no issue of priority inversion and there can be no deadlocks.

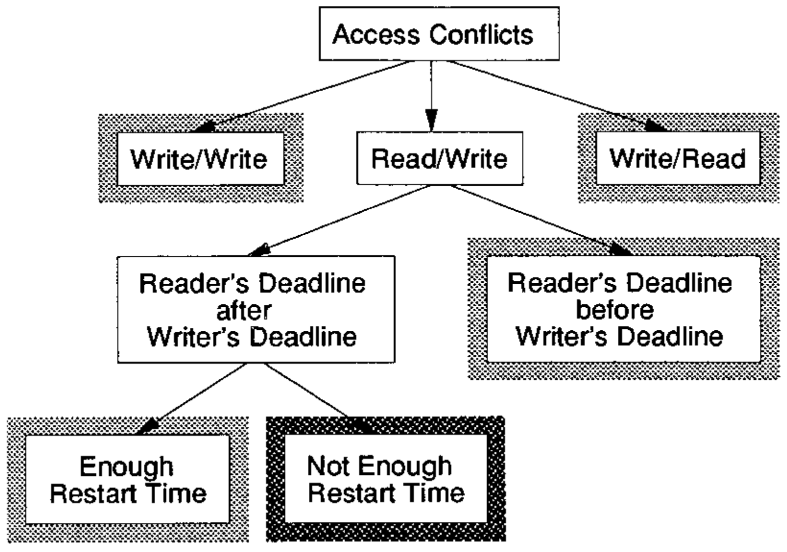
CRLs and CWLs are held for very short periods only, i.e. during certification. Although we neither block nor abort a BTC when a read/write conflict occurs, our approach is not optimistic since at the time of conflict we take some action:

branching of the reading BTC. This differs from optimistic algorithms in the sense that we assume that there will be a problem with certification and we choose to explore multiple possible outcomes in parallel.

To avoid running BTCs which have no chance of completing in time, one can estimate the minimum time needed for a new branch to finish execution, and only if there is enough time left before its deadline is a new BTC created.

Considering all possible types of access conflicts between transactions: 1) Write/Write, 2) Write/Read and 3) Read/Write, most of them can be resolved by our algorithm such that no transaction misses its deadline (due to concurrency control reasons). Write operations do not interfere with other Write operations in any way: before certification a Write simply creates a new version of an item, and since only one transaction can be in its certification phase at a time, no two conflicting CWL can ever exist simultaneously. An access conflict between a Write operation and a subsequent Read operation is resolved through branching. A Read operation followed by a Write operation from a transaction with an earlier deadline requires subsequent branching of the reader transaction. If there is enough time to complete the new branch before its deadline, this too does not cause any problems. The only case where a transaction will miss its deadline due to concurrency control is if there is not enough time for the restart of a new branch. A Read operation which is followed by a conflicting Write operation by a transaction with a later deadline than the writer does not cause a problem due to the commit ordering of transactions based on their deadlines. An overview of which conflicts can and which cannot be resolved by our algorithm is shown in Figure 8–5.

Usually algorithms try to complete the execution of a transaction as early as possible in order to hasten the release of locks held by that transaction and to achieve a better response time for it. It may, therefore, appear counter-productive to delay certification of a transaction until close to its deadline. This, however, is not the case. A delayed transaction neither consumes CPU or I/O resources nor holds any locks which would prevent the progress of other transactions. Furthermore, in real-time scheduling the primary object is not to minimise transaction



▩ *conflicts cannot be resolved*
▨ *conflicts can be resolved*

Figure 8–5: Conflict Resolution Overview

response times, but to maximise the proportion of transactions that successfully complete execution before their deadlines.

The algorithms introduced earlier — Wait Promote, Highest Priority and Conditional Restart — are all aiming at preventing higher priority transactions from being tardy in favour of lower priority transactions, where the priority is either based on which transaction has an earlier deadline or which has least slack time before its deadline. These basic ideas are still maintained in our new algorithm. When a transaction causes the abort of another one's BTC, it must have an earlier deadline — otherwise it wouldn't be its turn for certification — and, since certification is delayed until just before a transaction's deadline, it has less slack, i.e. none, than the other transaction (unless the later one is already doomed to be late). Our new algorithm also follows the idea that a transaction with an earlier deadline should not be aborted in favour of a transaction with a later deadline if there is a chance for both to commit. In fact, our algorithm never aborts a transaction with an earlier deadline in favour of a transaction with a later deadline.

The advantage of this algorithm is based on the capability of branching transactions to compute multiple possible outcomes: if one branch of a transaction needs to be aborted because some of the dirty data it read has been aborted, then there exists an alternative branch which was run with the assumption of that abort, and hence, the chance of that transaction completing on time is higher than if it had to restart from the beginning.

There is an important difference between the branching in BT-MV-2PL and this new algorithm: we now allow "subsequent branching". By this, we mean that a new branch can be created some time after the actual read request by the corresponding (now branched) reader BTC. Immediate branching, as opposed to subsequent branching, is easier, since the state of a BTC is known at the time of the branching decision. In order to facilitate subsequent branching, we must save the context of a BTC at the time of its Read operation, i.e. the starting point for any subsequently-created BTC. Saving the context for every Read for every BTC may cause an unacceptable overhead, and one may have to be more selective: only some of these possible contexts are saved. Effectively, these saved contexts provide some form of checkpoint from where a transaction can restart, without losing all previous work. In the most extreme case, a transaction restarts from the very beginning.

As in BT-MV-2PL, too much branching, immediate or subsequent, can cause the system to thrash, and some branch-control function may have to be applied. If branching is not allowed, a BTC reads the uncommitted version of an item with the most recent deadline[4] prior to its own or, if none exists, the committed version. Clearly though, only if sufficient resources are available to allow a reasonable amount of branching to take place can we expect performance advantages from our new algorithm.

Example In this example we assume five transactions: $T_1 \dots T_5$. Their deadlines are in the relative order of $T_2 \rightarrow T_4 \rightarrow T_3 \rightarrow T_1 \rightarrow T_5$. Database access by these transactions is as follows:

[4]With the deadline of a version of an item we mean the deadline of the transaction which created that version.

T_1: $r[u]\ w[y]$

T_2: $r[x]\ w[y]$

T_3: $w[u]\ r[z]$

T_4: $r[u]\ w[v]$

T_5: $r[v]\ r[z]$

Figure 8–6 shows a possible schedule of execution of these transactions under our new algorithm. We discuss the details of this schedule next and show the corresponding lock tables at various stages.

	T1		T2	T3	T4	T5	
Step 1	$r_{1,2}[u_{0,0}]$						
Step 2				$w_{3,1}[u_{3,1}]$			
Step 3		$r_{1,3}[u_{3,1}]$	$r_{2,2}[x_{0,0}]$				
Step 4	$w_{1,2}[y_{12}]$				$r_{4,2}[u_{0,0}]$		
Step 5	*Delay*	$w_{1,3}[y_{1,3}]$			$w_{4,2}[v_{4,2}]$		
Step 6		*Delay*	$w_{2,2}[y_{2,2}]$	$r_{3,2}[z_{0,0}]$	*Delay*	$r_{5,2}[v_{0,0}]$	$r_{5,3}[v_{4,2}]$
Step 7			*Ce+Co*	*Delay*		$r_{5,4}[z_{0,0}]$	$r_{5,5}[z_{0,0}]$
Step 8			*Deadline*		*Ce+Co*	*Delay*	*Delay*
Step 9				*Ce+Co*	*Deadline*	*Abort*	
Step 10	*Abort*	*Ce+Co*		*Deadline*			
Step 11	*Deadline*	*Deadline*					*Ce+Co*
Step 12						*Deadline*	*Deadline*

Figure 8–6: Real-time Schedule

Transaction T_1 begins execution and requests Read access to item u. T_1 starts execution as $BTC_{1,1}$, but since a Read operation is a potential source of subsequent branching it continues its operation as $BTC_{1,2}$. We effectively create a parent transaction $BTC_{1,1}$ which can be the starting point for any subsequent branch on this read, even though in this case no actual work has been done by $BTC_{1,1}$.

At step 2, $BTC_{3,1}$ performs a Write on item u. Since T_3's deadline is before T_1's, we must create a new branch for T_1 which reads the newly written version of u; we assume it is estimated that there is enough time for this new branch to complete before T_1's deadline. As a result we find that $BTC_{1,3}$ has been created and read $u_{3,1}$ at step 3. At the same time $BTC_{2,1}$ wanted to read x and did so after being continued as $BTC_{2,2}$.

At step 4, $BTC_{1,2}$ writes a new version of y ($y_{1,2}$) and $BTC_{4,1}$ becomes $BTC_{4,2}$ to read $u_{0,0}$. This latter read operation did not result in new branches for T_4, in spite of the currently existing uncommitted version of u ($u_{3,1}$), since T_4's deadline is before T_3's.

Although $BTC_{1,2}$ has completed, no certify request for this branch of T_1 is issued, since the deadline of T_1 is not imminent; the branch enters a "delay period" at step 5. At the same time, $BTC_{1,3}$ and $BTC_{4,2}$ create new versions of y and v, respectively.

At step 6, the branches of $BTC_{1,3}$ and $BTC_{4,2}$, too, enter a delay period. $BTC_{2,2}$ creates a new version of item y and $BTC_{3,1}$ wants to read z, and therefore becomes $BTC_{3,2}$ before reading the committed version: $z_{0,0}$. Since T_5's deadline is after T_4's, $BTC_{5,1}$ branches into $BTC_{5,2}$ and $BTC_{5,3}$, reading $v_{0,0}$ and $v_{4,2}$, respectively.

The lock table at this stage is shown in Figure 8–7. Since no transaction has entered certification yet, there are no CRLs and no CWLs on any data item.

At step 7, T_2 must begin its certification since its deadline is close. $BTC_{2,2}$'s TCRL on x and VWL on y can immediately be upgraded to the corresponding CRL and CWL, since there exist no conflicting other locks. Hence, a complete branch of T_2 is certified and it can commit. At the same time, $BTC_{3,2}$ has completed and entered a delay period, while both, $BTC_{5,2}$ and $BTC_{5,3}$, read the only existing version of z; they have become $BTC_{5,4}$ and $BTC_{5,5}$, respectively.

With T_4's deadline imminent, $BTC_{4,2}$ is certified — $BTC_{4,1}$ is automatically certified since it didn't perform any operations — and T_4 commits; Figure 8–8 shows the lock table for u and v after $BTC_{4,2}$'s certification, but before all of T_4's locks were released. Also at step 8, T_5's branches completed execution and are delayed.

As can been seen in Figure 8–8, the CWL of $BTC_{4,2}$ on v conflicts with a TCRL by $BTC_{5,2}$, and hence, once T_4 committed, an Abort was issued to the corresponding branch of T_5 (at step 9). Similarly to T_4, T_3 is certified and committed at step 9, and as a consequence the branch of $BTC_{1,2}$ is aborted at step 10. On

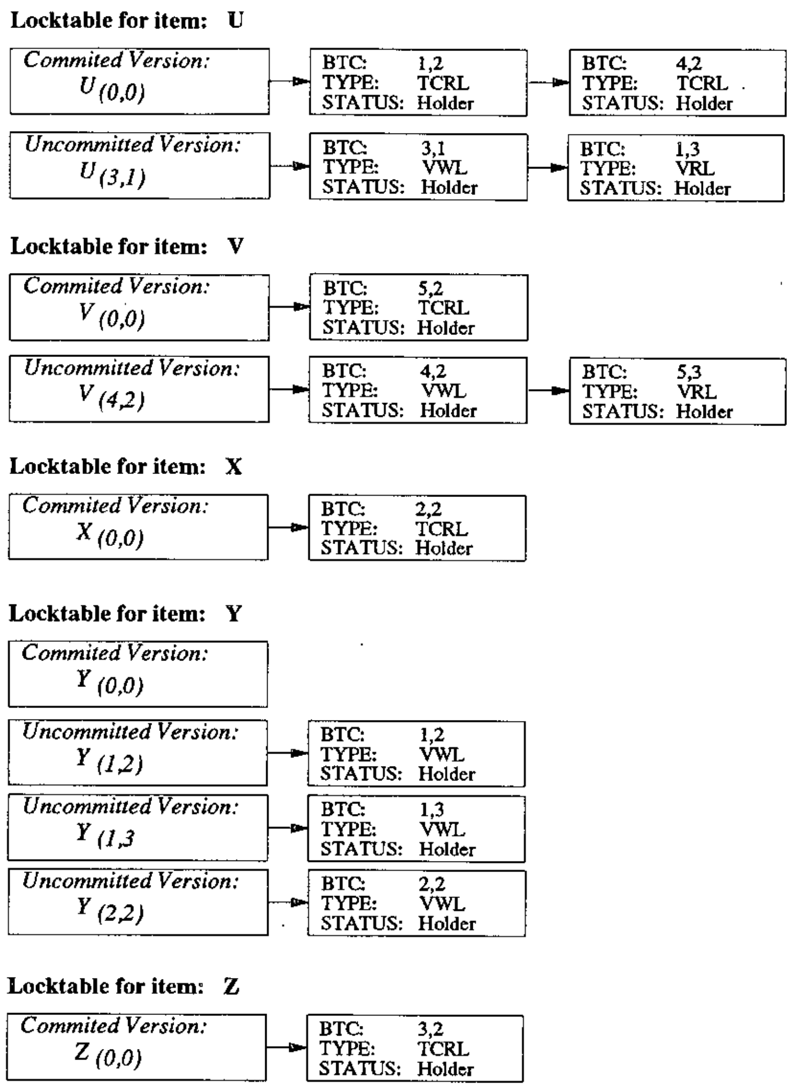

Locktable for item: U
Commited Version: U(0,0)
BTC: 1,2 TYPE: TCRL STATUS: Holder
BTC: 4,2 TYPE: TCRL STATUS: Holder
Uncommitted Version: U(3,1)
BTC: 3,1 TYPE: VWL STATUS: Holder
BTC: 1,3 TYPE: VRL STATUS: Holder
Locktable for item: V
Commited Version: V(0,0)
BTC: 5,2 TYPE: TCRL STATUS: Holder
Uncommitted Version: V(4,2)
BTC: 4,2 TYPE: VWL STATUS: Holder
BTC: 5,3 TYPE: VRL STATUS: Holder
Locktable for item: X
Commited Version: X(0,0)
BTC: 2,2 TYPE: TCRL STATUS: Holder
Locktable for item: Y
Commited Version: Y(0,0)
Uncommitted Version: Y(1,2)
BTC: 1,2 TYPE: VWL STATUS: Holder
Uncommitted Version: Y(1,3
BTC: 1,3 TYPE: VWL STATUS: Holder
Uncommitted Version: Y(2,2)
BTC: 2,2 TYPE: VWL STATUS: Holder
Locktable for item: Z
Commited Version: Z(0,0)
BTC: 3,2 TYPE: TCRL STATUS: Holder

Figure 8–7: Lock Table for Real-time Schedule after Step 6

the other side, the alternative branch of T_1 can be certified and committed at step 10. T_5's remaining branch is successfully certified and committed at step 11.

Note on Correctness: Although this algorithm is different in many ways from HBT-MV2PL, it applies the same basic locking rules as HBT-MV2PL (for details of these rules see Section 4.4.2):

- Overwrite Rule
- Write Lock Rule
- Read Lock Rule

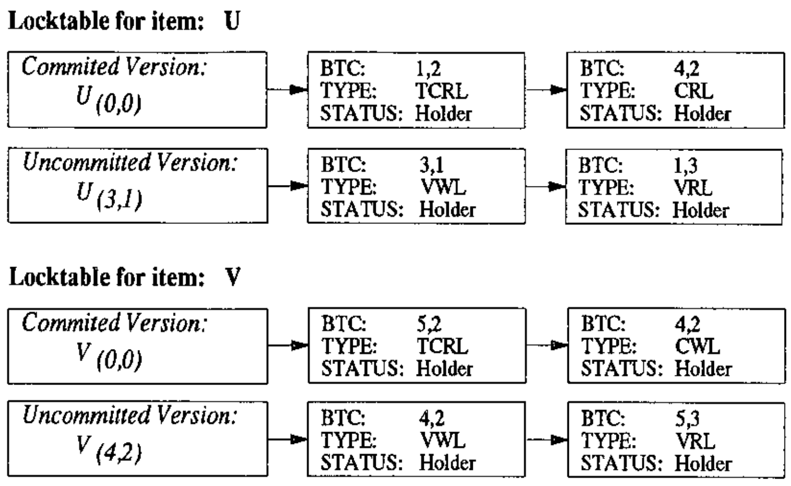

Locktable for item: U
Commited Version: U(0,0)
BTC: 1,2 TYPE: TCRL STATUS: Holder
BTC: 4,2 TYPE: CRL STATUS: Holder
Uncommitted Version: U(3,1)
BTC: 3,1 TYPE: VWL STATUS: Holder
BTC: 1,3 TYPE: VRL STATUS: Holder
Locktable for item: V
Commited Version: V(0,0)
BTC: 5,2 TYPE: TCRL STATUS: Holder
BTC: 4,2 TYPE: CWL STATUS: Holder
Uncommitted Version: V(4,2)
BTC: 4,2 TYPE: VWL STATUS: Holder
BTC: 5,3 TYPE: VRL STATUS: Holder

Figure 8–8: Lock Table for Real-time Schedule after $BTC_{4,2}$'s Certification

- Write/Write Lock Conflict Rule

- Read/Write Lock Conflict Rule

Since we have shown (in Chapter 4) that an algorithm applying these rules guarantees to only allow serializable schedules of transactions, it is also guaranteed that our new real-time concurrency control algorithm only allows serializable schedules.

Other Deferred-Commit Protocols The algorithm described above incorporates the idea of *deferred-commit*: a transaction delays its commit in order to allow other transactions to complete as well. This is a useful technique for real-time scheduling since the primary goal here is to meet transaction deadlines rather than optimising average response times. While there is no harm in delaying a transaction's commit until shortly before its deadline, it may allow other transactions — which would have been aborted in case the now delayed transaction had committed immediately — to complete as well.

Haritsa [44] reports that performance gains can be achieved, in the context of optimistic concurrency control for real-time scheduling, by making lower priority transactions wait after their validation and delay their commit. Son et al. [81] and

Agrawal et al. [3] describe real-time locking algorithms which incorporate the idea of deferred-commit.

Deferred-commit has also been considered by Bestavros and Broudakis [12,13] in their work on speculative concurrency control (SCC)[5] for real-time database systems. Since SCC is the approach most similar to branching transactions, a more detailed description of their (deferred-commit based) real-time concurrency control algorithms is given next, followed by a comparison with real-time scheduling with branching transactions.

Rather than simply considering the deadline of a transaction, SCC algorithms also take into account the value of a transaction; a transaction with an earlier deadline may have a lower value than a transaction with a later deadline[6] The relative "worth" (based on the transaction value and its deadline) of a transaction can be used to determine how much speculation it should be allowed, i.e. how many shadows are allowed to execute on its behalf. SCC with deferred commit (SCC-DC) does not immediately commit a transaction shadow once it has finished its execution, but evaluates whether it might be better for the overall performance of the system to delay the commit. Since it is computationally expensive to determine the optimal time of commit of a transaction — all possible serialization orders of active transactions in the system must be evaluated — SCC-DC uses an approximation by considering commit at certain discrete points in time only. SCC with Voted Waiting (SCC-VW) is an approximation heuristic, proposed to reduce the computing overhead caused by SCC-DC. Under this policy, all uncommitted transactions are allowed to vote for and against the commitment of a finished transaction shadow; we omit the details of the voting protocol, the interested reader is referred to Bestavros and Braoudakis [13].

[5]A brief introduction to SCC has been given in Section 3.4.5.

[6]Most other real-time concurrency control algorithms, including the one based on branching transactions, assume that all transactions in the system are assigned the same value.

In addition to the similarities and differences between SCC and branching transactions described in Section 3.4.5, SCC-DC and SCC-VW differ from our real-time BT algorithms and in a number of other ways. The BT algorithm is designed for transactions with hard deadlines and, as mentioned earlier, each transaction is assumed to be of equal value. The delay of a transaction commit under SCC is based on the value of a transaction — a transaction may still be of value after its deadline has passed — and the order in which transactions commit may be different from their respective deadlines. The delay of commit under BT is solely based on a transaction's deadline and transactions commit (or abort) in the order of their deadlines.

Chapter 9

Concluding Remarks

This chapter gives an overview of the results of this thesis and discusses possible future work based on branching transactions.

9.1 Summary of Results

9.1.1 Branching Transaction Model

The key contribution of this dissertation is a new transaction model: *branching transactions* which has been designed to address the issue of data contention in parallel database systems. Although data contention has been looked at in the past, no work has yet been reported (as far as is known to us) which addresses this issue in the particular context of parallel database systems. The key property of the branching transaction model is its ability to avoid unnecessary transaction blockings and restarts by following up more than one possible path of execution of a transaction in parallel in case of data conflicts.

The principles of this new transaction model were introduced in Chapter 3. Through a comparison with related work in this area, it was shown that our approach is indeed a unique and novel concept.

9.1.2 Concurrency Control and Recovery Algorithms

A substantial amount of work had been dedicated to the development of new concurrency control algorithms for branching transactions. Section 4.3 presented a new multi-version two-phase locking algorithm. Since it was recognised that unlimited branching may lead to system overloads in even the most powerful parallel computers, branching control policies were proposed in Section 3.3.3. This led to the development of a hybrid concurrency control algorithm, which is able to dynamically switch at run-time between (non-branching) two-phase locking, (non-branching) multi-version two-phase locking and our new branching transaction multi-version two-phase locking algorithm; this hybrid algorithm was discussed in Section 4.4. The ability to switch dynamically at run-time is an important result, since it allows a system to adapt from a more CPU expensive approach (branching transactions) to a resource conservative approach ("normal" two-phase locking). Due to the blocking nature of locking protocols, deadlocks can occur under these new concurrency control algorithms. The notion of a deadlock in the context of branching transactions has been defined and deadlock handling strategies were discussed in Sections 4.5.

A possible difficulty with branching was the potentially prohibitively high overhead due to additional logging activities. This has been resolved by a simple modification of the incremental log with deferred update recovery strategy (Section 4.6).

A further result in the context of concurrency control, is the development of a two-layer approach for the development of a concurrency control manager which applies to our new hybrid locking algorithm. Section 4.4.4 described how the full functionality of our new concurrency control algorithm could be implemented by taking advantage of an existing "normal" two-phase locking layer. This should prove useful for anyone wanting to migrate from an existing system towards branching transactions.

9.1.3 Architecture of BT Systems

In Chapter 5, it was shown how a branching transaction system may map onto various parallel hardware platforms. Possible system architectures for shared-memory and shared-something environments were introduced and the issue of load balancing and load control discussed. Section 5.7 discussed the concrete example of branching transactions on a Convex Exemplar computer.

Working in the context of a shared-something DBMS raised the problem of cache coherence. Since uncommitted as well committed data may be cached at different nodes, a new cache coherence protocol had to be developed. This new protocol was discussed in Section 5.5.

9.1.4 Correctness Proofs

To prove the correctness of our newly developed concurrency control algorithms under branching transactions, traditional serializability theory had to be extended to be able to cope with the concept of branching. The notion of basic agents, agents and agent histories (Definitions 13, 14 and 18) were introduced. Based on these, formal definitions of branching transactions (Definition 20) and branching transaction histories (Definition 22) were developed.

In the Branching Transaction Serializability Theorem (Theorem 4) we showed that as long as a branching transaction scheduler only allows histories whose committed projections are equivalent to a correct (non-branching) multi-version history, serializability is guaranteed. Having formalised the locking rules of the new hybrid BT concurrency control algorithm, we were then able to prove that these new algorithms indeed only allow correct schedules (Theorem 5).

9.1.5 Performance Study

The simulation study presented in Chapter 7 showed that branching transactions can achieve performance advantages over non-branching systems over a variety of parameters, assuming that a sufficient level of data contention exists. These performance improvements were achieved with even relatively small levels of branching. Best results were obtained for a 50/50 mix of Read and Write operations. The problem of exponential growth of the number of branches could not be observed in the experiments carried out.

The results also indicated that the choice of deadlock handling may have a relatively significant influence on the branching behaviour of the system, and that a high number of transaction aborts may have a detrimental effect on the branching that is carried out. Further studies are required to obtain a better understanding of these issues.

9.1.6 Real-time Scheduling

One area in which we felt branching transactions might be particularly suitable is real-time concurrency control. Chapter 8 describes the result of our work in this area, i.e. a new real-time concurrency control algorithm based on branching transactions has been developed. The new algorithm incorporates the idea of "deferred commit" which prevents, as far as possible, the abort of transactions with earlier deadlines by transactions with later deadlines.

9.2 Future Work

The basic research contribution of our new transaction model has led to a number of additional results in various areas, e.g. concurrency control, logging, deadlock handling, cache coherence, etc. Nevertheless, there still is more work to be done with branching transactions. The following three topics are of particular interest to the author and are intended to be the subject of future investigations.

9.2.1 Unifying Nested and Branching Transactions

In Section 3.4.1 we discussed the similarities and differences between nested transactions and branching transactions. Although both models are based on a hierarchical structure of subtransactions, nested transactions are aiming at a higher level of parallelism within a transaction, while branching transactions are designed to achieve better inter-transaction parallelism. Combining nesting and branching is expected to result in a new model which takes advantage of both approaches.

If the underlying sub-transaction management can be unified, for any DBMS with support for nested transactions it may be relatively easy to add support for branching transactions as well. This would provide a promising approach for introducing branching transaction systems into commercial DBMS.

9.2.2 Extension of Serializability Theory

The work on basic agents, agents and agent histories in the context of serializability theory provides a formal basis for dealing with sub-transaction structures. This work should be extended to be able to capture the semantics of nested transactions. A single theoretical framework for reasoning about branching and nested transactions would be ideally suited to carry out correctness proofs for the above-mentioned unified transaction model.

9.2.3 Performance Studies

The performance study of Chapter 7 was necessarily restricted in scope to keep it within what was feasible within this dissertation. Clearly, there a large number of additional experiments which one might want to carry out. We have already mentioned that a comprehensive study of the effects of deadlock handling strategies on a branching transaction system is desirable. Other studies of interest are branching transactions in the context of shared-something systems — this would introduce issues of cache coherence — and the performance of real-time scheduling with branching transactions, as discussed in Chapter 8. Furthermore, additional research is required to get a better understanding of the types of workloads and DBMS configurations for which *hybrid* branching/non-branching systems are most beneficial. To be able to compare branching transactions not only with the flat transaction model and "normal" two-phase locking, the current simulation software should be developed further to include modules for ordered shared locks and speculative concurrency control.

Appendix A

Paper Publication

This appendix contains a copy of the paper by Burger and Thanisch [16].

Branching Transactions: A Transaction Model for Parallel Database Systems

Albert Burger, Peter Thanisch

Department of Computer Science,
University of Edinburgh, Scotland

Abstract

In order to exploit massively parallel computers, database management systems must achieve a high level of concurrency when executing transactions. In a high contention environment, however, parallelism is severely limited due to transaction blocking, and the utilisation of hardware resources, e.g. CPUs, can be low.

We propose a transaction model, *Branching Transactions*, together with an appropriate concurrency control algorithm, which, in case of data conflicts, avoids unnecessary transaction blockings and restarts by executing alternative paths of transactions in parallel. Our approach uses additional hardware resources, mainly CPU — which would otherwise sit idle due to data contention — to improve transaction response time and throughput.

1 Introduction

In recent years, multi-processor systems based on fast and inexpensive micro-processors have become widely available. The total performance/price ratio of such systems is usually higher than that of traditional mainframe computers. We, therefore, see a trend towards the replacement of mainframes by parallel systems in high performance transaction processing environments.

In general, high transaction throughput and short transaction response time are the primary performance design goals for a database management system. In parallel transaction processing systems, *interference* [2] — the slowdown each new process imposes on all others when accessing shared resources — limits speedup and scaleup. In fact, data contention can be the limiting factor for performance in a shared-nothing parallel database machine [3] [5]; under such conditions the utilisation of CPUs and disks is relatively low.

The component of a database management system dealing with synchronisation of access to shared data, and therefore responsible for issues of data contention, is the *concurrency control manager*. All existing algorithms resolve conflicts either by *blocking* or *restarting* transactions at the time of conflict (pessimistic algorithms) or when a transaction tries to commit (optimistic algorithms). Common to both groups, the decision made at the time of conflict may not be the right one. In a pessimistic algorithm, for example, the roll-back of a transaction is frequently caused by a situation that *might* have led to a deadlock; or a transaction is blocked because it *might* have violated serializability. In an optimistic algorithm, a conflict that was ignored during the execution may require the restart of a transaction. The key problem is that at the time of conflict we usually don't know which is the right decision to make. Contrary to all CC algorithms known to us, we propose an approach where a transaction, instead of making a particular decision, follows up alternative paths of execution concurrently. Once it is known which was the right path to pursue, all others can be aborted. This approach allows us to avoid many unnecessary blockings and restarts of transactions which lead to performance problems in all existing CC algorithms.

Executing alternative paths of a transaction concurrently increases demand on hardware resources, in particular, CPUs. However, as we pointed out earlier, in a parallel database system data contention can lead to low CPU utilisation, and it seems appropriate to use this idle CPU time to reduce the problem caused by sharing data. The idea of "sacrificing" hardware resources to improve concurrency in a database system is not entirely new: multi-version algorithms [6] use additional memory and disk space — to store multiple versions of the same data item — to improve the level of concurrency.

The remainder of the paper is organised as follows: in Section 2 we present the branching transaction model; Section 3 contains a two-phase locking algorithm for branching transactions; Section 4 presents a formal proof of correctness for the two-phase locking algorithm presented before; Section 5 discusses aspects of logging and recovery; Section 6 briefly introduces the idea of branching restrictions; and Section 7 gives a conclusion and summary of future work.

2 The Branching Transaction Model

Existing concurrency control algorithms resolve a conflict by either blocking or restarting one of the transactions involved. We will use the following three transactions, T_1, T_2 and T_3, to illustrate this point, and to describe the principles of branching transactions. ($r[x]$ denotes a read operation on data item x; $w[x]$ denotes a write operation on data item x.)

T_1: $r[z]$, $r[x]$, $r[y]$, $r[t]$, $w[t]$, $r[m]$, $r[n]$, $w[n]$
T_2: $w[x]$, $r[z]$, $r[u]$, $w[u]$
T_3: $w[y]$, $r[l]$, $r[k]$, $w[k]$, $r[u]$, $w[u]$, $r[p]$

If these transactions were executed under a two-phase locking algorithm, the schedule in Table 1 would be a possible interleaving of their execution. ($r_i[x_j]$ denotes T_i reading the value of data item x written by T_j; $w_i[x_i]$ denotes T_i updating x; c_i denotes the Commit operation of T_i. The values of data items prior to the execution of this schedule are indicated by subscript 0.) At step (2), when T_1 tries to read data item x, it is blocked by T_2's lock on x; T_2 has written to x at step (1), and must therefore hold an exclusive lock on it. T_1 remains blocked until T_2 releases its lock on x. Similarly, T_1 gets blocked again at step (7), because of T_3's lock on y.

The scheduler blocks T_1 at step (2), since it cannot decide whether T_1 should read the value written to x by T_2, or the value x had prior to step (1). In case T_2 aborts, or commits after T_1, T_1 should read x_0, otherwise it should read x_2. Since T_2's fate is not know at the time of conflict, the scheduler delays its decision — blocks T_1 — until it has sufficient information to decide.

In case T_2 commits before T_1, blocking of T_1, and the delay of its response time that follows from it, is unnecessary. Other concurrency control algorithms have similar problems: optimistic algorithms, for example, need to abort a transaction and restart it, if the certification of it fails; timestamp ordering algorithms maintain serializability by enforcing the timestamp order on conflicting operations, even though not all aborted transactions would have violated serializability.

To overcome the problems of these "wrong decisions" by the scheduler, we propose to pursue alternative paths of execution of a transaction until it is known which was the correct path to follow. In other words, at the time of conflict a transaction branches into two or more alternative copies of itself which then continue to execute concurrently.

Table 2 shows a schedule in which T_1 is executed as a branching transaction. This time when T_1 tries to read data item x, it branches into two components: $T_{1,1}$ and $T_{1,2}$, the first proceeds using the original value of x, the second reads x_2. At step (3), further branching is necessary since y has been updated by T_3 at step (2). At step (6), it has become clear — since T_2 just committed — that the correct decision at step (2) was to read x_2. Therefore, it is not necessary to pursue further those components that were started under the assumption that x_0 should be read, and $T_{1,3}$ and $T_{1,4}$ abort.

Step	T_1	T_2	T_3
1	$r_1[z_0]$	$w_2[x_2]$	
2	blocked	$r_2[z_0]$	$w_3[y_3]$
3	blocked	$r_2[u_0]$	$r_3[l_0]$
4	blocked	$w_2[u_2]$	$r_3[k_0]$
5	blocked	c_2	$w_3[k_3]$
6	$r_1[x_2]$		$r_3[u_2]$
7	blocked		$w_3[u_3]$
8	blocked		$r_3[p_0]$
9	blocked		c_3
10	$r_1[y_3]$		
11	$r_1[t_0]$		
12	$w_1[t_1]$		
13	$r_1[m_0]$		
14	$r_1[n_0]$		
15	$w_1[n_1]$		
16	c_1		

Table 1: Schedule under Two-phase Locking (No Branching)

Step	**Branching Transaction T_1**				T_2	T_3
-	$\underline{T_{1,0}}$:					
1	$r_{1,0}[z_0]$				$w_2[x_2]$	
-	$\underline{T_{1,1}}$:		$\underline{T_{1,2}}$:			
2	$r_{1,1}[x_0]$		$r_{1,2}[x_2]$		$r_2[z_0]$	$w_3[y_3]$
-	$\underline{T_{1,3}}$:	$\underline{T_{1,4}}$:	$\underline{T_{1,5}}$:	$\underline{T_{1,6}}$:		
3	$r_{1,3}[y_0]$	$r_{1,4}[y_3]$	$r_{1,5}[y_0]$	$r_{1,6}[y_3]$	$r_2[u_0]$	$r_3[l_0]$
4	$r_{1,3}[t_0]$	$r_{1,4}[t_0]$	$r_{1,5}[t_0]$	$r_{1,6}[t_0]$	$w_2[u_2]$	$r_3[k_0]$
5	$w_{1,3}[t_{1,3}]$	$w_{1,4}[t_{1,4}]$	$w_{1,5}[t_{1,5}]$	$w_{1,6}[t_{1,6}]$	c_2	$w_3[k_3]$
6	$a_{1,3}$	$a_{1,4}$	$r_{1,5}[m_0]$	$r_{1,6}[m_0]$		$r_3[u_2]$
7			$r_{1,5}[n_0]$	$r_{1,6}[n_0]$		$w_3[u_3]$
8			$w_{1,5}[n_{1,5}]$	$w_{1,6}[n_{1,6}]$		$r_3[p_0]$
9			blocked	blocked		c_3
10			$a_{1,5}$	$c_{1,6}$		

Table 2: Schedule for Branching Transaction

When a particular path of a branching transaction has executed all operations, it is not allowed to commit until it is known whether all assumptions made by it are fulfilled: $T_{1,5}$ and $T_{1,6}$ are blocked at step (9) since they cannot commit until after T_3 committed (or aborted). Since T_3 indeed commits, $T_{1,5}$ aborts and $T_{1,6}$ commits at step (10).

Running T_1 as a branching transaction allowed the scheduler to commit the transaction action within 10 steps, as opposed to 16 in the previous case.

Transaction Graphs and Components: We represent the branching hierarchy of a branching transaction T_i by a *transaction graph* (a rooted, directed tree), with T_i's transaction components as nodes ($T_{i,0}$ as root), and an edge $T_{i,j} \rightarrow T_{i,k}$, if $T_{i,j}$ created $T_{i,k}$. The transaction graph for T_1 in our example is shown in Figure 1. A transaction component $T_{i,j}$ is a *descendant* of transaction component $T_{i,k}$, if there exists a path from $T_{i,j}$ to $T_{i,k}$ in T_i's transaction graph.

Figure 2 shows the state transition diagram of transaction components. After a component has been created it is *ACTIVE*, i.e. it is executing the operations of the corresponding transaction. (We include the temporary blocking of a component in this state.) If the component is involved in a conflict, and branching is necessary, it creates two or more descendant components

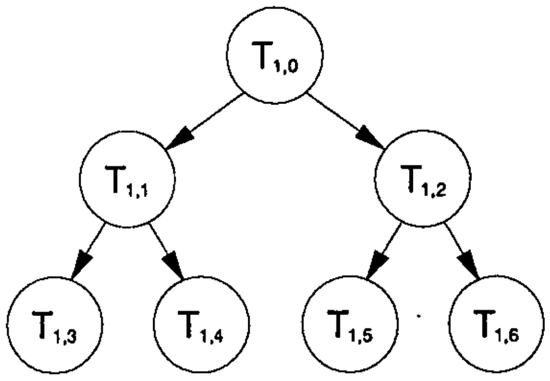


Figure 1: Transaction Graph

and enters the *BRANCHED* state. In this state no more operations are executed by the component. If all descendants of a component eventually abort, then it must be aborted as well. An active transaction component can be aborted either by the system (such as $T_{1,3}$ and $T_{1,4}$ in our example), or by an explicit abort command issued by the component itself. If a component is active and has executed the last operation of its transaction, and if no unresolved dependencies exist, it can commit. If a component commits, so do all its ancestors. We show the state changes for transaction components in our example in Figure 3.

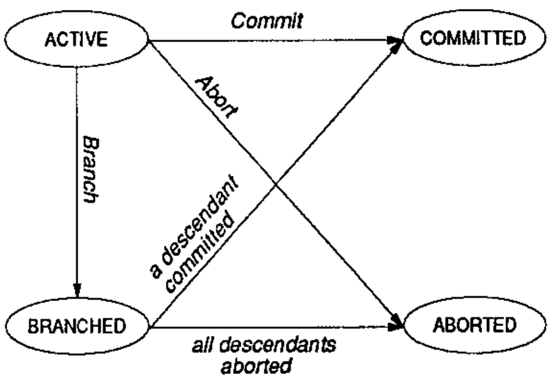


Figure 2: Transaction Component State Transition Diagram

It is a key property of branching transactions that only one path in every transaction graph can commit, and all components not part of this path are aborted. Any updates on the database performed by aborted transaction components are rolled-back.

Any two transaction components of the same transaction tree can be executed in parallel, unless there exists a path in the tree between them. Components from different paths execute in isolation: they do not read any updates made by the other and are allowed to update the same data items independently. To be able to maintain the ACID properties [1] of transactions, multiple versions of data items must be maintained.

Implementation of Branching Transactions: We use the abstract model of a database system depicted in Figure 4 to describe some aspects of implementing branching transactions [2]. A user submits a transaction for execution to the transaction manager (TM), which creates a

[1] A=Atomicity, C=Consistency, I=Isolation, and D=Durability [4]

[2] The model ignores the distributed nature of a parallel database system. A detailed discussion of this, however, would be beyond the scope of this paper.

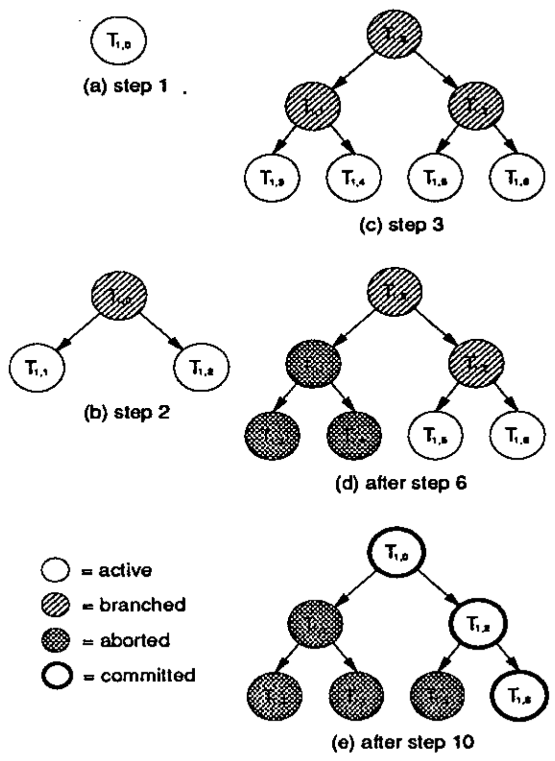


Figure 3: Transaction Component State Changes within a Branching Transaction

new transaction component for it. The TM maintains the transaction graph for each transaction and submits database operations (Read and Write) to the scheduler. The scheduler implements a concurrency control policy, and therefore decides when branching should occur. Although we only describe a locking policy in this paper, other mechanisms are possible. The scheduler is also responsible for preventing a transaction component from committing, if there exist some unresolved dependencies for it. Once a branching transaction has committed (or aborted), the TM reports back to the user. Branching of transactions is transparent to the user.

3 Two-Phase Locking for Branching Transactions

In this section we describe a multi-version two-phase locking algorithm for branching transactions (hereafter referred to as *BT_MV2PL*). Our algorithm is based on Bernstein, etal.'s [1] description of multi-version two-phase locking.

As mentioned earlier, branching transactions require a multi-version (MV) environment to enable the concurrent execution of alternative paths. Under *BT_MV2PL* the data manager has to store one or more versions of a data item, of which only one was created by a committed transaction. When a transaction component that created a new version of an item commits, its version overrides the previously committed one.

When the scheduler receives a Write from transaction component $T_{i,j}$ ($w_{i,j}[x]$), it translates it into $w_{i,j}[x_{i,j}]$, immediately schedules it for execution — the data manager creates a new version: $x_{i,j}$ — and sets a write lock ($wl_{i,j}[x_{i,j}]$) on this new version.

A transaction component $T_{i,j}$ cannot access versions of x created by another component, $T_{i,k}$, of the same transaction, unless $T_{i,j}$ is a descendant of $T_{i,k}$. In this case, or when $T_{i,j}$ itself has previously written to x, the scheduler translates a Read operation from $T_{i,j}$ ($r_{i,j}[x]$) into a version Read on $x_{i,k}$ or $x_{i,j}$, respectively (irrespective of any other existing versions of x).

If only one version (the most recently committed one) of x exists, the scheduler translates a

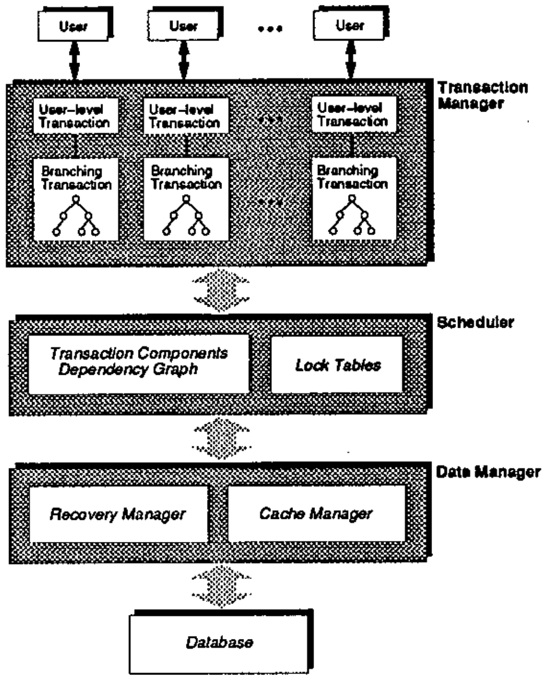


Figure 4: DBMS Model for Branching Transactions

Read operation from transaction component $T_{i,j}$, ($r_{i,j}[x]$), into a version Read on the committed version of x, i.e. $r_{i,j}[x_{k,l}]$, where $x_{k,l}$ is the committed version of x. As with Write operations, the operation (Read) is scheduled immediately, and an appropriate lock is set ($rl_{i,j}[x_{k,l}]$).

When a Read operation from transaction component $T_{i,j}$ ($r_{i,j}[x]$) arrives at the scheduler, and multiple versions of x exist (none of them created by $T_{i,j}$ or one of its ancestors), the scheduler informs the transaction manager that branching is required. The transaction manager then creates new transaction components — one for each version of x — and the scheduler immediately executes a version Read for each new component: $r_{i,k_l}[x_{m_l,n_l}]$, for $l = 1$..number of versions (T_{i,k_l} are the new transaction components, [x_{m_l,n_l} the versions of x). Each new component sets a read lock on the version it read.

If at the time of branching one of the versions of x had a certify lock on it, and the transaction that set this lock commits, then the transaction component that read the previously committed version of x must abort.

Before a transaction component can commit, it must be certified. *Certification* involves two phases: (1) a transaction component must wait until all data it or one of its ancestors read are committed, and (2) it must upgrade all its and its ancestors' write locks to certify locks. A transaction component trying to acquire a certify lock on x is blocked, if there exists a certify lock on any version of x already, or if there exists a read lock on the committed version of x.

As with other locking protocols, deadlocks may occur. Traditional deadlock detection and resolution techniques can be applied. Some (or possibly all — this is subject of further investigation) deadlocks can be resolved by aborting only some transaction components without having to roll-back any transaction completely.

Note that there are no conflicts between read locks and write locks, and between write locks and write locks. Transactions only conflict through read locks and certify locks, as described above. Write locks are simply markers to remember on which items a transaction needs to obtain a certify lock.

4 Correctness Proof

To prove the correctness of our new concurrency control algorithm, we will now formalise the concept of branching transactions, and show that a *BT_MV2PL* scheduler only allows serializable schedules of concurrent transactions. Our proof is based on the serializability theorem of Bernstein, etal. ([1]). We will present some of their theorems here to keep our paper as self contained as possible; proofs of their theorems are, however, omitted. The interested reader is referred to their book.

4.1 Branching Transaction Multi-Version Histories

As described earlier, the branching of transactions is solely the responsibility of the scheduler and is transparent to the user. Hence, at the user level, we consider a transaction as a sequence of operations on the database.

Definition 1 (from [1], page 27) *A **user-level transaction** UT_i is a partial order with ordering relation $<_i^U$, where*

1. $UT_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$.
2. $a_i \in UT_i$ *iff* $c_i \notin UT_i$.
3. *if* t *is* c_i *or* a_i *(whichever is in* UT_i*), for any other operation* $p \in UT_i$, $p <_i^U t$.
4. *if* $r_i[x]$, $w_i[x] \in UT_i$, *then* $r_i[x] <_i^U w_i[x]$ *or* $w_i[x] <_i^U r_i[x]$.

Condition (1) states that the only operations on the database are either Reads ($r_i(x)$) or Writes ($w_i(x)$) and that a transaction either Commits [3] (c_i) or Aborts (a_i). Condition (2) says that a transaction can either Commit or Abort, but not both. Condition (3) states that a Commit or Abort is the last operation of a transaction. Condition (4) requires the partial order $<_i^U$ to specify an order of execution on a Read and Write if they access the same data.

Since user-level transactions are described in terms of these 4 operations: $r_i[x]$, $w_i[x]$, c_i and a_i, we need some mapping function h that captures the fact that one user-level operation may be translated into a set of BT operations. The following BT operations are defined:

$r_{i,j}[x_{k,l}]$: transaction component $T_{i,j}$ reads the version of data item x written by transaction component $T_{k,l}$.

$w_{i,j}[x_{i,j}]$: transaction component $T_{i,j}$ creates a new version of data item x.

$b_{i,j}(i,k)$: transaction component $T_{i,j}$ spawns a new transaction component $T_{i,k}$.

$a_{i,j}$: transaction component $T_{i,j}$ aborts.

$c_{i,j}$: transaction component $T_{i,j}$ commits.

The mapping from user-level to BT operations depends on the concurrency control policy used by the scheduler. The following mapping rules, however, hold independently of what policy is used.

1. Mapping of Read operations:
   $h(r_i[x]) = \{r_{i,j}[x_{l,m}] \mid T_{i,j}$ is a transaction component executing in behalf of T_i and reading some version of $x\} \cup$
   $\{b_{i,j}(i,k) \mid T_{i,j}$ created transaction component $T_{i,k}$ $\}$

   A user-level Read operation may require multiple BT Read operations and the creation of new transaction components.

[3] The Commit here means that a transaction actually commits, not merely that it requests to commit, since in the latter case a transaction may still be aborted.

2. Mapping of Write operations:
   $h(w_i[x]) = \{w_{i,j}[x_{i,j}] \mid T_{i,j}$ is a transaction component updating $x\}$

   A user-level Write operation on x may be executed by more than one transaction component, although only one of these versions will ultimately be committed.

3. Mapping of Abort operations:
   $h(a_i) = \{a_{i,j} \mid T_{i,j}$ is a leaf node in T_i's transaction graph $\}$

   A user-level transaction aborts, if all leaf nodes in the corresponding BT's transaction graph abort.

4. Mapping of Commit operations:
   $h(c_i) = \{c_{i,j}\} \cup$
   $\{a_{i,k} \mid T_{i,k}$ is an active transaction component of T_i, and $j \neq k\}$

   If a user-level transaction T_i commits, then one of the active components commits, but all others abort.

A *branching transaction* T_i is the result of mapping the operations of a user-level transaction UT_i onto BT operations. Before we give a formal definition of branching transactions, we must present formal definitions for some of the terminology introduced earlier.

Definition 2 *Two BT operations, p and q,* ***conflict*** *with each other if p is a Read operation ($r_{i,j}[x_{m,n}]$), reading the version of a data item ($x_{m,n}$) which was written by q ($w_{k,l}[x_{k,l}]$, where $m = k$, $n = l$), and p and q are operations of different transactions ($i \neq k$).*

Definition 3 *A* ***transaction component*** *$T_{i,j}$ of branching transaction T_i is obtained by deleting all operations $p_{i,k}$ of T_i for which $k \neq j$.*

Definition 4 *A transaction component $T_{i,d}$ is a* ***descendant*** *of transaction component $T_{k,a}$ in history H, if $k = i$ and either (1) operation $b_{k,a}(k,d) \in H$, or (2) $\exists\ b_{k,\alpha_1}(k,\alpha_2), b_{k,\alpha_2}(k,\alpha_3), \ldots, b_{k,\alpha_{n-1}}(k,\alpha_n) \in H$, such that $\alpha_1 = a$, $\alpha_n = d$ and $n \geq 2$ The set of all descendants of transaction component $T_{k,a}$ is denoted by $desc(k,a)$.*

Definition 5 *A transaction component $T_{i,a}$ is an* ***ancestor*** *of transaction component $T_{k,d}$ in history H, if $k = i$ and either (1) operation $b_{k,a}(k,d) \in H$, or (2) $\exists\ b_{k,\alpha_1}(k,\alpha_2), b_{k,\alpha_2}(k,\alpha_3), \ldots, b_{k,\alpha_{n-1}}(k,\alpha_n) \in H$, such that $\alpha_1 = a$, $\alpha_n = d$ and $n \geq 2$ The set of all ancestors of transaction component $T_{k,d}$ is denoted by $anc(k,a)$.*

Definition 6 *A* ***transaction graph*** *(TG) for branching transaction T_i in history H, denoted $TG(H,i)$, is a rooted directed tree whose nodes are the transaction components of T_i in H, and whose edges are all $T_{i,j} \rightarrow T_{i,k}$, such that $b_{i,j}(i,k) \in H$. The root node is labelled $T_{i,0}$.*

Definition 7 *A* ***transaction path*** *in branching transaction T_i is a path in $TG(H,i)$ between the root node ($T_{i,0}$) and one of its leaf nodes.*

Definition 8 *A* ***branching transaction*** *T_i is obtained by translating the operations of a user-level transaction UT_i into BT operations according to some mapping function h. T_i is a partial order with ordering relation $<_i$, where*

1. let $q \in UT_i$; if $c_{i,j} \in H$, then there exists some $p_{i,k} \in H$, such that $p_{i,k} \in h(q)$, and $j = k$ or $T_{i,k} \in anc(i,j)$.

2. let $p, q \in h(o)$, where o is a user-level operation of UT_i. If $p \in T_{i,j}$ and $q \in T_{i,k}$, and there exists a transaction path which contains both, $T_{i,j}$ and $T_{i,k}$, then $j = k$ and $p = q$, unless $p = b_{i,j}(i,k)$ and $q = r_{i,k}[x_{m,n}]$ (for some $x_{m,n}$).

3. *if p_i and q_i are operations in UT_i and $p_i <_i^U q_i$, then for any $s_{i,j}, t_{i,k} \in T_i$, where $s_{i,j} \in h(p_i)$, $t_{i,k} \in h(q_i)$: if there exists a transaction path P in $TG(H, i)$ that contains transaction components $T_{i,j}$ and $T_{i,k}$, then $s_{i,j} <_i t_{i,k}$.*

4. *if $c_{i,j} \in T_i$, then for any transaction component $T_{i,k}$ where $k \neq j$, $a_{i,k} \in T_{i,k}$ or $b_{i,k}(i,l) \in T_{i,k}$ (for some l).*

5. *$c_{i,j} \in T_{ij}$ iff $a_{i,j}, b_{i,j}(i,k) \notin T_{i,j}$, $a_{i,j} \in T_{ij}$ iff $c_{i,j}, b_{i,j}(i,k) \notin T_{i,j}$, $b_{i,j}(i,k) \in T_{ij}$ iff $c_{i,j}, a_{i,j} \notin T_{i,j}$ (for some k).*

6. *let $p, q \in H$ be two Branch operations: $p = b_{i,j}(i,k)$ and $q = b_{l,m}(l,n)$. If $i = l$ and $k = n$, then $p = q$.*

7. *if t is $c_{i,j}$ or $a_{i,j}$ (whichever is in $T_{i,j}$), then for any other operation $p_{i,j} \in T_{i,j}$, $p_{i,j} <_i t$. If t is $b_{i,j}(i,k)$ (for some k), then all operations $p_{i,j}$, such that $t <_i p_{i,j}$, are Branch operations $b_{i,j}(i,l)$, where $l \neq k$.*

Condition (1) states that all operations requested by the user-level transaction are translated into the appropriate BT operations, i.e. all user-level operations are executed at least once along the successful transaction path of the corresponding BT transaction. Condition (2) states that each user-level operation is executed at most once along each transaction path. Condition (3) says that all orderings given in user-level transactions are preserved within branching transactions. Condition (4) states that only one branch (path) of a BT transaction is allowed to commit. Each transaction component either branches, commits or aborts (condition (5)). Condition (6) guarantees that all newly created transaction components are uniquely identified by their two indices. Condition (7) says that a transaction component either terminates with a Commit, an Abort, or a number of Branch operations.

Similar to histories and multi-version histories described in [1], we use the notion of a BT history to describe the interleaved execution of branching transactions.

Definition 9 *A complete BT history H over a set of top-level transactions $UT = \{UT_1, \ldots, UT_n\}$ is a partial order on the corresponding set of branching transactions $T = \{T_0, \ldots, T_n\}$ with ordering relation $<_H$, where*

1. *$H = h(\bigcup_{i=1}^n UT_i)$, for some translation function h.*

2. *$<_H \supseteq \bigcup_{i=1}^n <_i$.*

3. *for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.*

4. *if $r_{i,j}[x_{k,l}] \in H$, then $w_{k,l}[x_{k,l}] <_H r_{i,j}[x_{k,l}]$.*

5. *if $w_{i,j}[x_{i,j}] <_i r_{i,k}[x_{l,m}]$, and $k = j$ or $T_{i,k} \in desc(i,j)$, then $i = l$ and $j = m$.*

6. *if $r_{i,j}[x_{k,l}] \in H$ and $i \neq k$, then if $c_{i,j} \in H$ or $c_{i,d} \in H$ and $T_{i,d} \in desc(T_{i,j})$, then it also holds that either $c_{k,l} \in H$ or $c_{k,m} \in H$ and $T_{k,m} \in desc(T_{k,l})$, and that $c_{k,x} <_H c_{i,y}$, for some x, y.*

Condition (1) states that all operations submitted by user-level transactions are translated into appropriate BT operations. Condition (2) states that the execution of branching transactions maintains all orderings defined on BT operations. Condition (3) states that the scheduler must determine an order on all pairs of conflicting BT operations. Condition (4) states that a BT transaction cannot read a version before it was created. Condition (5) states that if a transaction component wants to read a data item which it or any of its ancestors has previously created a version of that data item, then it must read that version. Condition (6) states that if a transaction component wants to commit, then all versions read by it or one of its ancestors must be committed, i.e. the transaction component that created the version, or one of its descendants, must have committed.

To prove serializability for $BTMV$ histories we need only be concerned with those operations that were executed by transaction components which committed, or which have a descendant that committed. We use the notion of a *committed projection* to represent the committed part of a BT history.

Definition 10 *The committed projection of a BT history is obtained through the following three steps, and is denoted by $C(H)$.*

1. *Delete all operations $p_{i,j} \in T_{i,j}$ from H for which it is not the case that $c_{i,j} \in H$ or $c_{i,d} \in H$ and $T_{i,d} \in desc(T_{i,j})$.*
2. *Delete all $b_{i,j}(i,k)$ operations from H.*
3. *Map each $r_{i,j}[x_{k,l}]$ into $r_i[x_k]$, each $w_{i,j}[x_{i,j}]$ into $w_i[x_i]$, and each $c_{i,j}$ into c_i.*

Step (1) eliminates all operations which were not executed by those components that are part of the committed BT transactions. We have now effectively pruned all transaction graphs to one particular transaction path. To determine whether our history is serializable, we don't need the information about which operation was executed by what transaction component, but only by which transaction. Hence, we can delete all BT Branch operations and map all remaining operations $o_{i,j}$ to o_i (steps (2) and (3)). The latter implicitly leads to a mapping of data versions $x_{i,j}$ to x_i.

Proposition 1 *The committed projection of a BT history is a multi-version history (as defined in [1]).*

Proof: omitted for space reasons; □

4.2 Correctness of BT_MV2PL

To prove that all committed projections of histories produced by a BT_MV2PL scheduler are serializable, we first need to describe certain properties of these projections, i.e. an algorithm implementing BT_MV2PL policy guarantees the following properties of the committed projections of histories allowed by this algorithm. The certification of a transaction T_i is denoted by f_i.

BT_MV2PL_1: For every T_i, f_i follows all of T_i's Reads and Writes and precedes T_i's commitment. ($r_i[x_j], w_i[x_i] <_{C(H)} f_i <_{C(H)} c_i$, for all j)

The certification of a transaction, i.e. a transaction component and all of its ancestors in H, takes place after all of its Reads and Writes are executed; and only after successful certification can a transaction commit.

BT_MV2PL_2: For every $r_j[x_i]$ in $C(H)$, if $j \neq i$, then $c_i <_{C(H)} f_j$.

A Read of a transaction cannot be certified until the transaction that created the read version has committed.

BT_MV2PL_3: If $r_k[x_j]$ and $w_i[x_i]$ are in $C(H)$, then either $c_i <_{C(H)} r_k[x_j]$ or $r_k[x_j] <_{C(H)} f_i$.

A transaction's Read on x is either ordered after the Commit or before the certification of another transaction that creates a new version of x. This is because of the conflicts between Read Locks and Certify Locks.

BT_MV2PL_4: For every $r_k[x_j]$ and $w_i[x_i]$ (i, j, k distinct), if $f_i <_{C(H)} r_k[x_j]$, then $f_i <_{C(H)} f_j$.

A Read operation is only certified if it read the most recently certified version of x. This is because a Read cannot be certified until the version it read has been committed; and since the transaction does not release its Read Lock until commit time, it prevents any other transaction from overwriting x — to overwrite the last certified version of x, a transaction needs to obtain a Certify Lock, which is not possible before the Read Lock on the last certified version has been released.

BT_MV2PL_5: For every $r_k[x_j]$ and $w_i[x_i]$, where $i \neq j$ and $i \neq k$, if $r_k[x_j] <_{C(H)} f_i$ and $f_j <_{C(H)} f_i$, then $f_k <_{C(H)} f_i$

A transaction can only certify an update of x after all transactions that have read a previously certified version have certified their Read. This is because of the same reason as in BT_MV2PL_4: a transaction does not release its Read Locks until after it was certified.

BT_MV2PL_6: For every $w_i[x_i]$, $w_j[x_j]$ $(i \neq j)$, either $f_i <_{C(H)} f_j$ or $f_j <_{C(H)} f_i$.

The certification of every two transactions that write the same data item are ordered with respect to each other. This is because Certify Locks conflict with each other.

The proof of serializability of BT_MV2PL histories is based on two theorems by Bernstein, etal. [1]; we will present these theorems here, but omit their proofs. Furthermore, we shall use their notion of *serialization graphs*, denoted $SG(H)$ and *multi-version serialization graphs*, denoted $MVSG(H, \ll)$. In brief, a serialization graph describes transaction dependencies due to conflicting operations. In addition, a multi-version serialization graph, also describes transaction dependencies due to the ordering of versions in a multi-version system. For more details on these graphs and the omitted proofs, we refer the reader to [1].

Theorem 1 (Theorem 5.3 in [1]) *Let H be a MV history over a set of transactions T. $C(H)$ is equivalent to a serial, 1V history over T iff H is 1SR.*

Theorem 2 (Theorem 5.4 in [1]) *An MV history H is 1SR iff there exists a version order $\ll$ such that $MVSG(H, \ll)$ is acyclic.*

Theorem 3 *Every committed projection $C(H)$ of a BT history produced by a BT_MV2PL scheduler is serializable, i.e. is equivalent to a serial, 1V history.*

Proof: [4] To prove above theorem, we will show that all edges in $MVSG(C(H), \ll)$ are in certification order, i.e. if $T_i \rightarrow T_j$ in $MVSG(C(H), \ll)$, then $f_i <_{C(H)} f_j$. Because of BT_MV2PL_6, $\ll$ is indeed a version order.

Let $T_i \rightarrow T_j$ be in $SG(C(H))$, then T_j must read a version created by T_i: $r_j[x_i] \in C(H)$. By BT_MV2PL_2, we know that $c_i <_{C(H)} f_i$, and since $f_i <_{C(H)} c_i$ (by BT_MV2PL_1), it follows that $f_i <_{C(H)} f_j$.

Let $w_i[x_i]$, $w_j[x_j]$, $r_k[x_j] \in C(H)$, where i, j, k are distinct. Then either $x_i \ll x_j$ or $x_j \ll x_i$. If $x_i \ll x_j$, then the version order edge is $T_i \rightarrow T_j$. $f_i <_{C(H)} f_j$ follows directly from the definition of $\ll$. If $x_j \ll x_i$, then the version order edge is $T_k \rightarrow T_i$. By BT_MV2PL_3, either $c_i <_{C(H)} r_k[x_j]$ or $r_k[x_j] <_{C(H)} f_i$. In the first case, by BT_MV2PL_1, if follows that $f_i <_{C(H)} r_k[x_j]$, and then by BT_MV2PL_4 $f_i <_{C(H)} f_j$. But this contradicts $f_j <_{C(H)} r_k[x_i]$ (given by the definition of the version order), and therefore it follows that $r_k[x_j] <_{C(H)} f_i$; and since $f_j <_{C(H)} f_i$, it follows by BT_MV2PL_5 that $f_k <_{C(H)} f_i$, as desired.

We have shown that all edges in $MVSG(C(H), \ll)$ are in certification order; and since the certification is embedded in a history which is acyclic by definition, $MVSG(C(H), \ll)$ is acyclic, too. By theorems 1 and 2 it follows that $C(H)$ is equivalent to a serial 1V history. □

[4]This proof is based on Bernstein, Hadzilacos and Goodman's proof of correctness for 2-Version 2-Phase-Locking; Theorem 5.6 in their book.

5 Logging and Recovery

The component of a DBMS responsible for maintaining transaction atomicity is called the *recovery manager (RM)* [5]. A RM has to write log records to disk to allow transaction roll-backs in case of transaction or system failures. To avoid excessive I/O overhead under BT, we suggest the following logging and recovery strategy. During execution of a transaction component, all updates to the database are kept as new versions of an item in main memory only; the database on disk is not updated and no log records are written. If a transaction component aborts, all data versions created by it are simply discarded; no I/O is necessary for aborting transaction components. To commit a component, for each item to be written, first a log record is written and then the database is updated on disk. Assuming that log records and the database are stored on different disks, I/O to log disks and database disks can proceed in parallel (as long as for each item the log record is written before updating the database). Recovery from a system crash works as with traditional recovery techniques (see [1]). Since no log records are written until commit time — the time at which it is known which path of execution of a branching transaction is correct — logging is only required for one path of a branching transaction, and hence, the BT model does not impose any extra I/O overhead for logging on the system; in both cases, branching and non-branching, two I/O accesses (one for logging and one to update the database) are required per database update.

6 Branching Restriction Policies

If unlimited branching of transactions is allowed, the number of transaction components may grow exponentially and the system suffer from thrashing. We, therefore, need a policy to regulate the branching of transactions, a policy which will only allow transactions to branch if sufficient system resources are available. A function *branch_control* must be defined which determines whether branching should be permitted. Since BT requires primarily additional CPU time, *branch_control* may allow branching only, if the average CPU utilization is below a certain threshold x. The value to which x should be set and what other *branch_control* functions may be applied, are open questions. Simulation studies will help to clarify this issue.

In case branching of a transaction component is denied by *branch_control*, the corresponding read operation follows the (non-branching) $MV2PL$ policy as described in [1], i.e. the component tries to set a read-lock on the last committed version of the item to be read. Serializability is still maintained the same way as before (read-locks conflict with certify-locks).

Using such a policy, branching of transactions is controlled dynamically. It is adapted to the current system workload in the sense that branching is only allowed as long as enough resources are available.

7 Conclusion and Future Work

In this paper we have presented a transaction model which aims to reduce the problem of data contention in a parallel database system. Our approach takes advantage of low resource utilisation, which is frequently the result of data contention. Using a simple example we have shown the potential performance advantage of branching transactions. We also presented a two-phase locking algorithm for our new transaction model, and proved that it guarantees serializability. Although we used 2PL as a concrete example, branching transactions is a more general concept, and other algorithms, e.g. timestamp ordering, can be adapted to work with it.

More information is needed about the performance of branching transactions. In particular, we must investigate how various transaction workloads and hardware environments influence the performance of BTs. We also need a performance comparison with already existing transaction

[5] An overview of existing recovery techniques can be found in [1].

models/concurrency control algorithms. For this work, we are currently preparing a comprehensive simulation study. This study will also be used to investigate various branching restriction policies.

References

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

[2] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35,6:85–98, 1992.

[3] P. Franaszek and J.T. Robinson. Limitations of Concurrency in Transaction Processing. *ACM Transactions on Database Systems*, 10(1):1–28, March 1985.

[4] J.N. Gray. A transaction model. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science 85*, pages 282–298. Springer Verlag, Berlin, 1980.

[5] B-C Jenq, B.C. Twitchell, and T.W. Keller. Locking Performance in a Shared Nothing Parallel Database Machine. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):530–543, December 1989.

[6] R. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1978.

Bibliography

[1] R.K. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] D. Agrawal and A. El Abbadi. Locks with Constraint Sharing. In *Proc. of 9th ACM Symp. on Princinples of Database Systems*, pages 85–93, 1990.

[3] D. Agrawal, A. El Abbadi, and R. Jeffers. Using Delayed Commitment in Locking Protocols for Real-time Databases. In *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 104–113, 1992.

[4] D. Agrawal, A El Abbadi, and A.E. Lang. The Performance of Protocols Based on Locks with Ordered Sharing. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):805–818, October 1994.

[5] R. Agrawal, M.J. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654, Dec. 1987.

[6] R. Balter, P. Berard, and P. Decitre. Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management. In *Proceedings of 1st ACM Symp. on Principles of Dist. Computing*, August 1982.

[7] D. Bell and Grimson. J. *Distributed Database Systems.* Addison-Wesley, 1992.

[8] P. Bernstein and N. Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, October 1980.

[9] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

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

[11] A. Bestavros and S. Broudakis. Timeliness via speculation for real-time databases. In *Proceedings of the 14th IEEE Realtime System Symposium*, Dec. 1994.

[12] A. Bestavros and S. Broudakis. Value-cognizant Speculative Concurrency Control. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 122–133, 1995.

[13] A. Bestavros and S. Broudakis. Value-cognizant Speculative Concurrency Control for Real-Time Databases. *Information Systems*, 21(1):75–101, 1996.

[14] G. M. Birtwistle. *Discrete Event Modelling on Simula*. Macmillan, 1979.

[15] H. Boral and D. DeWitt. Database Machines: An idea whose time has passed? In *Proceedings of the 1983 Workshop on Database Machines*. Springer Verlag, 1983.

[16] A. Burger and P. Thanisch. Branching Transactions: A Transaction Model for Parallel Databases. In *Proceedings of the 12th British National Conference on Databases*, pages 122–136, July 1994.

[17] A.G. Burger. A Performance Study of Multi-Version Concurrency Control in a Distributed Database System. Master's thesis, University of Missouri - Kansas City, Kansas City, MO, July 1991.

[18] F.W. Burton. Speculative Computation, Parallelism and Functional Programming. *IEEE Transactions on Computers*, 16:1190–1193, December 1985.

[19] M. Butler, R. Bloor, and P. Beach. *DATABASE An Evaluation and Comparison.* Butler Bloor Computer Research, 1990.

[20] M. Carey, S. Krishnamurthi, and M Livny. Load control for locking: The 'half-and-half' approach. Technical Report 880, Computer Science Department, University of Wisconsin-Madison, October 1989.

[21] M. Carey and M. Stonebraker. The Performance of Concurrency Control Algorithms for Database Management Systems. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, August 1984.

[22] M.J. Carey. Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions. *ACM Transactions on Software Engineering*, SE-13(6):746–751, June 1987.

[23] M.J. Carey and W.A. Muhanna. The Performance of Multiversion Concurrency Control Algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, November 1986.

[24] S. Ceri and G. Pelagatti. *Distributed Databases, Principles and Systems.* McGraw Hill, 1984.

[25] F. Cesarini and S. Salza, editors. *Database Machine Performance: Modelling Methodologies and Evaluation Strategies.* Springer-Verlag, 1987.

[26] P.K. Chrysanthis and K. Ramamritham. Acta, a framework for specifying and reasoning about the transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

[27] D.T. Davies. Recovery Semantics for a DB/DC System. In *Proceedings of the 28th ACM National Conference*, pages 136–141. ACM, ACM Press, August 1973.

[28] D. Davis. Where Client/Server Fits. *Datamation*, pages 36–38, July 1991.

[29] P. Denning. Thrashing: Its Causes and Prevention. In *Proceedings of AFIPS (Fall Joint Computer Conf.)*, volume 33, 1968.

[30] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35,6:85–98, 1992.

[31] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 1990.

[32] A.K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[33] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, 1989.

[34] J.L. Eppinger, L.B. Mummert, and A.Z. Spector. *Camelot and Avalon, A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[35] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[36] A. Fekete, N.A. Lynch, M. Merrit, and W. Weihl. Nested Transactions, Conflict-Based Locking and Dynamic Atomicity. Technical Report MIT/LCS/TM-340, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.

[37] P. A. Franaszek and J. T. Robinson. Limitations of Concurrency in Transaction Processing. *ACM Transactions on Database Systems*, 10(1):1–28, March 1985.

[38] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Conference*, 1987.

[39] J. Gray, editor. *The Benchmark Handbook.* Morgan Kaufmann Publishers, Inc., 1991.

[40] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[41] J.N. Gray. Notes on Database Operating Systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course.* Springer Verlag, New York, 1978.

[42] J.N. Gray, R.A. Lorie, and G.R. Putzolu. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Proceed. International Conf. on Very Large Data Bases*, pages 428–451, 1975.

[43] T. Haerder and P. Peinl. Evaluating Multiple Server DBMS in General Purpose Operating System Environments. In *Proceed. of the 10th International Conference on Very Large Databases*, 1984.

[44] J.R. Haritsa, M.J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 1990 ACM PODS Symposium*, 1990.

[45] M.P. Herlihy and W. Weihl. Hybrid Concurrency Control for Abstract Data Types. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1988.

[46] A.P. Hillman, G.L. Alexanderson, and R.M. Grassl. *Discrete and Combinatorial Mathematics.* Collier MacMillan Publishers, 1987.

[47] M. Hsu and B. Zhang. Performance Evaluation of Cautious Waiting. *ACM Transactions on Database Systems*, 17(3):477–512, September 1992.

[48] B-C Jenq, B.C. Twitchell, and T.W. Keller. Locking Performance in a Shared Nothing Parallel Database Machine. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):530–543, December 1989.

[49] J.S. Keen and W.J. Dally. Management of Precommitted Transactions in a Concurrent DBMS. In *Proc. of Int. Conf. on Information and Knowledge Management*, pages 387–394, Baltimore, USA, November 1992.

[50] W. Kim and J. Srivastava. Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling. In *Proc. Real-Time Systems Symp.*, pages 222–231, 1991.

[51] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[52] H.F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1986.

[53] E. Kuehn and A Puntiam, F. Elmagarmid. Transaction Specification in Multidatabase Systems based on Parallel Logic Programming. In *Proc. of First International Workshop on Interoperability in Multidatabase Systems, IMS91*, April 1991.

[54] V. Kumar and A. Burger. Performance Measurement of Some Main Memory Database Recovery Algorithms. In *7th IEEE International Conference on Data Engineering*, Kobe, Japan, 1991.

[55] H.T. Kung and J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[56] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[57] W.-T. K. Lin and J. Nolte. Basic Timestamping, Multiple Version Timestamp, and Two-Phase Locking. In *Ninth Conference on Very Large Databases*, pages 109–119, Florence, Italy, 1983. IEEE.

[58] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22,3:267–293, 1990.

[59] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[60] J. Moad. Client/Server OLTP Arrives! *Datamation*, pages 26–30, March 1992.

[61] A. Moenkeberg and G. Weikum. Conflict-driven load control for the avoidance of data-contention thrashing. In *Proc. of the 7th Int'l. Conf. on Data Engineering*, April 1991.

[62] A. Moenkeberg and G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, August 1992.

[63] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1981.

[64] J.E.B. Moss. Nested transactions and reliable, distributed computing. In *Proceedings of the 2nd Symp. Reliability of Distributed Software and Database Systems*, pages 33–39, 1982.

[65] M.G. Norman and P. Thanisch. *Parallel Database Technology: An Evaluation and Comparison of Scalable Systems*. The Bloor Research Group, UK, 1995. ISBN 1-874160-17-1.

[66] M.G. Norman, T. Zurek, and P. Thanisch. Much Ado About Shared Nothing. *To appear in ACM SIGMOD Record.*

[67] ORACLE. Oracle7, Parallel Server Administrator's Guide. Technical Report MIT/LCS/TM-340, Oracle Corporation, December 1992.

[68] B. Orfali, D. Harkey, and J. Edwards. *Essential Client/Server Survival Guide*. John Wiley and Sons, 1994.

[69] R.B. Osborne. Speculative Computation in Multilisp. In *Proceedings of U.S./Japan Workshop on Parallel Lisp*, June 1989.

[70] M.T. Oszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[71] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[72] R. J. Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, 1987.

[73] J. Protić, M. Tomašević, and Milutinović. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, Summer 1996.

[74] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open ended activities. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 1988.

[75] E. Rahm. Concurrency and Coherency Control in Database Sharing Systems. Technical Report ZRI 3/91,Dec. 1991, Revised 1992, University of Kaiserslautern, Dept. of Computer Science, August 1992.

[76] R. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1978.

[77] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[78] P.M. Schwarz. *Transactions on Typed Objects.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1984.

[79] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22,3:183–236, 1990.

[80] A.H. Skarra and S.B. Zdonik. Concurrency control and object-oriented databases. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 395–421. ACM Press, New York, 1989.

[81] S. Son, S Park, and Y. Lin. An Integrated Real-time Locking Protocol. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 527–534, February 1992.

[82] Sang H. Son and Navid Haghighi. Performance Evaluation of Multiversion Database Systems. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 129–136, February 1990.

[83] Robert R. Stoll. *Set Theory and Logic.* Dover Publications Inc., 1979.

[84] M. Stonebraker. The Case for Shared-Nothing. *Database Eng.*, 9(1), 1986.

[85] Y. Tay. *Locking Performance in Centralized Databases.* Academic Press, 1987.

[86] Y. Tay, N. Goodman, and R. Suri. Locking Performance in Centralized Databases. *ACM Trans. on Database Sys.*, 10(4), December 1985.

[87] G. Thomas, G.R. Thompson, C.-W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman. Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys*, 22,3:237–266, 1990.

[88] A. Thomasian. Thrashing in Two-Phase Locking Revisisted. In *Eighth International Conference on Data Engineering*, pages 518–519, February 1992.

[89] A. Thomasian. Two-Phase Locking Performance and its Thrashing Behavior. *ACM Transactions on Database Systems*, 18(4):197–227, December 1993.

[90] A. Thomasian and I. K. Ryu. Performance Analysis of Two-Phase Locking. *IEEE Transactions on Software Engineering*, 17(5), May 1991.

[91] T. Thompson. The World's Fastest Computers. *Byte*, pages 44–64, January 1996.

[92] O Ulusoy. An Annotated Bibliography on Real-Time Database Systems. *ACM SIGMOD Record*, 24(4), Dec. 1995.

[93] P. Valduriez. Parallel Database Systems: The Case for Shared-Something. In *Proc. of 9th Int. Conf. on Data Engineering*, pages 460–465, Vienna, Austria, April 1993.

[94] W. Weihl. Data-dependent concurrency control and recovery. In *Proceedings of the Second Annual ACM Symposium on Principles of Distrib uted Computing*, 1983.

[95] W. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. In *IEEE Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, 1988.

[96] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[97] G. Weikum and H. Schek. Multi-level Transactions and Open Nested Transactions. *IEEE Data Engineering Bulletin*, March 1991.

[98] H.C. Young and W.-H. Wang. Computer Architecture for Commercial Workloads. In *Conference Record of The Twenty-Sixth Asilomar Conference on Signals, Systems and Computers*, pages 56–60, October 1992.

[99] P. S. Yu, D. M. Dias, and S.S. Lavenberg. On the Analytical Modeling of Database Concurrency Control. *Journal of the ACM*, 40(4), September 1993.

[100] P.S. Yu, K-L Wu, K-J Lin, and S.H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.