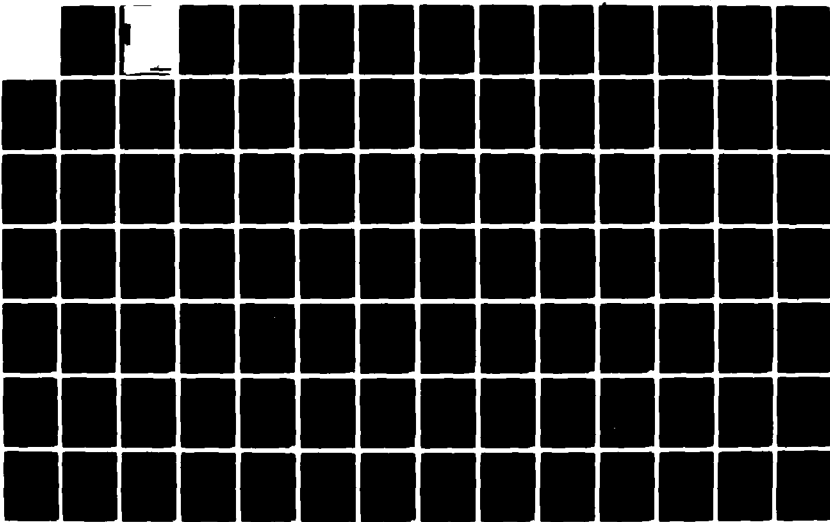
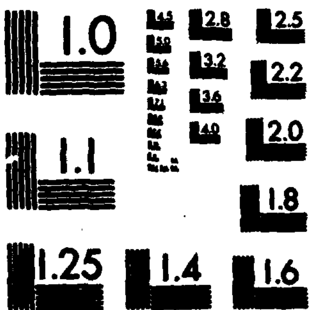
AD-A121 515 DESIGN OF CONCURRENCY CONTROLS FOR TRANSACTION PROCESSING SYSTEMS(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE J T ROBINSON 02 APR 82 1/2

UNCLASSIFIED N00014-76-C-0370 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A121515

82 11 17 065

CMU-CS-82-114

Design of Concurrency Controls for Transaction Processing Systems

John T. Robinson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

April 2, 1982

DTIC
ELECTE
NOV 17 1982
S
A

Copyright © 1982 by John T. Robinson

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and by the Office of Naval Research under Contract N00014-76-C-0370.

This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

The designer of a concurrency control is currently faced with a confusing situation: there has been a proliferation of proposed methods, but the performance of various methods under differing systems and applications is unknown. Furthermore, the optimal method for a given system might vary with system usage; in fact, some methods could later prove incompatible with desired system changes unforeseen at design time.

One way to approach these problems is to avoid commitment to any one method. This can be done by separating policy from correctness in the design of the concurrency control. This approach allows the concurrency control method to easily be modified to optimize desired performance characteristics, or to satisfy unforeseen performance criteria. Here, this approach has been successfully used for the case in which concurrency is hidden from record management and application programs. This case is defined more precisely by presenting a four-level framework for transaction processing system design.

This framework is suitable for uniprocessor, multiprocessor, or distributed systems. In this framework there are two subsystems, a concurrency control and a global memory manager, that are responsible for controlling access to all shared data objects. The concurrency control and global memory manager are application-independent, and are essentially autonomous. They may be functionally distributed between two processors, or distributed among many processors by partitioning the set of shared data objects. Since access to all shared data objects is controlled by these two subsystems, all other subsystems may be distributed in any fashion desired. In particular, in computer networks all other subsystems may be replicated.

A general paradigm for concurrency control is developed in which transactions are not required to follow any given protocol. Instead, possible conflicts are detected, and a policy determines how the possible conflict is handled. Thus, the two-phase locking protocol is just one of many possible policies; the optimistic method for concurrency control can be implemented by another policy. The paradigm is designed so that regardless of the choices made by the policy the database remains consistent. Policies may be defined at design-time, or they may be defined by modules that are executed at run-time. In the latter case, it is possible to dynamically change policies, even while the system is in use. This is shown to be highly convenient for policy development and experimentation.

The global memory manager designs support multi-version objects, which are used to solve the granularity problem for queries (read-only transactions) by making it unnecessary for queries to interact with the concurrency control. Some properties of the global memory manager designs are that new versions of objects may be written asynchronously by concurrent transactions, and that old versions of objects may be garbage-collected at the

earliest point at which it can be guaranteed that no future transaction or query will access that version.

Using these designs, a complete transaction processing system was developed for Cm*, a distributed multi-microprocessor. This system used a concurrency control in which all functions required by the paradigm were available, and the concurrency control used a policy module implementing a number of policies, any of which could be chosen at run-time. The record manager of this system supported a simple relational view of the database, with one or more B-tree indexes for each relation. The record manager was replicated, and copies of versions of shared data objects were cached. The usefulness of the design framework was illustrated by the fact that the record manager, the most complex subsystem, was earlier developed on a completely different centralized system, and required only minor modifications to be used in this system.

Experiments were performed with varying numbers of processors, and with various policies. For this system, the effect of waiting due to locking proved to be negligible, and so locking policies generally gave the best performance. This result may not apply to other systems, though: as an example of one of many possible differences, in this system individual processors were not multiprogrammed. However, using a concurrency control policy module, it is easy to investigate many different concurrency controls on any system, as demonstrated by these experiments.

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, H. T. Kung, for his continued encouragement and guidance throughout my stay at CMU. Thanks are also due the other members of my committee, Nico Habermann, Charles Eastman, and Christos Papadimitriou, for their suggestions and support.

The implementation of a transaction processing system on Cm*/Medusa might never have come to pass without the aid of Pradeep Sindhu. Discussions with George Robertson proved valuable in discovering the alternatives for query support described in Chapter 6. Other parts of this work have also benefited greatly from conversations with various members of the CMU CSD community. The graphs were generated by Ivor Durham's Plot program -- Ivor also helped me (as he does everyone) with various font, Bravo, Markup, etc. problems.

Last, I would like to thank Brian Bennett, Ray Bryant, Willy Chiu, and other members of the systems lab at T. J. Watson Research Center for their comments and encouragement.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
Distribution/
Availability Codes
Dist | Avail and/or Special
A
DTIC COPY INSPECTED 2

CONTENTS

1. Introduction 1
- 1.1. What is Transaction Processing? 1
- 1.2. Problems of Transaction Processing 4
- 1.3. Problems Considered Here 7
- 1.4. Summary of this Work 8
- 1.5. Relationship to Previous Work 10

2. Design Principles 12
- 2.1. On the Criteria to be Used in Decomposing (Transaction Processing) Systems into Modules 12
- 2.2. Problems of the Three-Level Architecture 13
- 2.3. A Four-Level Architecture 14
- 2.4. Example Systems 15
  - 2.4.1. A Centralized System 15
  - 2.4.2. A Personal Computer Network System 15
  - 2.4.3. A Multi-Microprocessor System 18
- 2.5. Serializability and Conflicts 18

3. System Overview 22
- 3.1. Communication between Subsystems 22
- 3.2. Data Objects 22
- 3.3. Read and Write Phases 23
- 3.4. Transaction Numbering and Versions 23
- 3.5. Local Memory Managers 24
  - 3.5.1. Cache Management 24
  - 3.5.2. Write-Phase Support 24
  - 3.5.3. Hiding Versions and Concurrency from the RcdM 25
  - 3.5.4. GMM/CC Interface 25
- 3.6. Summary of Subsystems and Interfaces 25
  - 3.6.1. Record Manager 25
  - 3.6.2. Local Memory Manager 26
  - 3.6.3. Concurrency Control 26
  - 3.6.4. Global Memory Manager 27
  - 3.6.5. Recovery Manager 27

4. A General Paradigm for Concurrency Controls 28
- 4.1. Controlling Transactions 28
- 4.2. Correctness of the Concurrency Control 30
- 4.3. The Paradigm 31
- 4.4. Policies 35
- 4.5. Generality of the Paradigm 37
- 4.6. Partitioning the Concurrency Control 38

5. Basic Policies 41
5.1. Definition of the Basic Policies 41
5.2. Deadlock Detection 42
5.3. Some "Interesting" Policies 43
6. Global Memory Managers 44
6.1. Memory Management 44
6.2. Transaction Support 44
6.3. Query Support 45
6.4. Garbage Collection 46
6.5. Partitioning the Global Memory Manager 47
7. Transaction Processing on Cm* 48
7.1. Overview of Cm*/Medusa 48
7.2. The Transaction Processing System 50
7.3. Maximum Throughput Experiments 53
7.4. Policy Experiments 58
8. Conclusion 62
8.1. On the Use of Physical Pointers 62
8.2. On Multi-Version Objects 62
8.3. On General Concurrency Controls 63
8.4. Implementation Experience 63
8.5. Implications of the Cm* Experiments 64
8.6. Further Research 66
Appendix I. Design of Record Managers 67
Appendix II. Concurrency Control Algorithms 77
References 100

FIGURES

Figure 1.1.	Transaction Processing	3
Figure 1.2.	Access Path Determined at Execution-Time	6
Figure 1.3.	Personal Computer Network with Shared Expensive Devices	7
Figure 2.1.	Three-Level Architecture	13
Figure 2.2.	Four-Level Architecture	14
Figure 2.3.	Centralized System	15
Figure 2.4.	Personal Computer Network System	16
Figure 2.5.	Multi-Microprocessor System	17
Figure 4.1.	Transaction State Transitions	30
Figure 4.2.	Only Violation of Correctness Criterion	32
Figure 7.1.	Cm* Architecture	48
Figure 7.2.	Medusa Task Force Structure	49
Figure 7.3.	Transaction Processing System Structure	51
Figure 7.4.	File Structure Used by Record Manager	52
Figure 7.5.	Part of an Experiment's Trace	54
Figure 7.6.	Throughput using Medusa File System	55
Figure 7.7.	Throughput using Kmap Block Moves	55
Figure 7.8.	Number of Restarts using Medusa File System	57
Figure 7.9.	Number of Restarts using Kmap Block Moves	57
Figure 7.10	Execution Time using Medusa File System	60
Figure 7.11	Execution Time using Kmap Block Moves	60
Figure 7.12	Number of Shared Page Accesses using Medusa File System	61
Figure 7.13	Number of Shared Page Accesses using Kmap Block Moves	61
Figure A1.	Example 2-D-B-Tree	71
Figure A2.	Example File Structure	75

TABLES

Table A.	Policy Experiments	59
Table B.	Growing K-D-B-Trees	74

1. Introduction

In this introduction the nature of transaction processing will be examined, followed by an identification of some of the problems unique to transaction processing. Then, after listing the problems of concern here, this work and its relationship to previous work will be summarized.

1.1. What is Transaction Processing?

The nature of transaction processing systems can best be illustrated by considering a number of examples. Some examples of commercial transaction processing systems include banking, airline reservation, and inventory control systems; some office automation examples include memo and appointment scheduling systems; an example of software engineering support is a documentation system where the documentation of modules is added or edited as the modules are developed; finally, some general information system examples include bulletin boards, shared bibliographies, and personnel directories.

In all of these examples there is an underlying database that is shared by a number of users. Thus, the problem of transaction processing system design is an extension of the more general problem of database design, and all problems of database design are also problems in transaction processing system design. The distinguishing property of transaction processing systems is that any of the users sharing the database can in principle modify the database.

Since the database is shared, users must be restricted in the manner in which they are allowed to modify the database -- otherwise the database could easily become internally inconsistent (e.g., damaged access structures) and unusable, or externally inconsistent (e.g., containing information not in agreement with reality) and unreliable.

A formal definition of internal consistency is implementation dependent. For example, if the database is structured as a tree, then the property that the graph formed by the nodes and pointers in this structure actually be a tree is an internal consistency property. In general, internal consistency properties are properties that can be determined to hold or not by examination only of the information in the database. The problem of preserving internal consistency leads naturally to the notion of a *transaction*: given a formal definition of the shared database and its internal consistency, a transaction can be formally defined as a procedure that modifies the database in such a fashion that if the database is internally consistent before the transaction is executed, then the database is internally consistent after the transaction has completed. By restricting users to modifying the database using only procedures that are strongly believed to be transactions, either through testing or correctness proofs, the problem of preserving internal consistency is partly solved -- the additional problems of protection, recovery, and concurrency are discussed below. Henceforth calling a procedure that is strongly believed to be a transaction simply a transaction, some examples of transactions are:

banking -- deposit or withdraw funds to or from an account, transfer funds from one account to another;

airlines -- make or cancel a reservation;

inventory control -- add or subtract from the quantity on hand of a certain item;

memo system -- send a memo to a list of recipients, remove a memo from a collection of memos;

appointment scheduling -- try to schedule a meeting for a set of people in a given time period, cancel a previously scheduled meeting;

documentation -- add, edit, or delete documentation for a new, modified, or obsolete module;

bulletin boards -- post, correct, or remove a new, incorrect, or old bulletin;

shared bibliographies -- add or edit a new or incorrect description of a bibliographic entry;

personnel directories -- add, edit, or delete information for a given person.

A procedure that accesses the database but does not modify the database is called a *query*. Some examples of queries are:

banking -- retrieve the current balance in an account or set of accounts, generate an account statement (history);

airlines -- retrieve the number of available seats on a given flight, find all flights or sequences of flights with given departure and destination points;

inventory control -- find the quantity on hand of a certain item, find the total value of all stock;

memo system -- read a memo, find all memos on a given subject;

appointment scheduling -- generate an appointment calendar;

documentation -- read the documentation for a given module, find all modules with a given keyword, check for consistency in the specifications of a set of modules;

bulletin boards -- read a bulletin, find all bulletins from a given date or with a given keyword;

shared bibliographies -- find all entries for a given author or for a given subject;

personnel directories -- find someone's office or phone number, find everyone with a given job classification in a given location.

Some common properties of all these transaction processing systems can now be identified. First, in every case, some real-world system (or potentially real-world) is being modelled. In

each case, a transaction corresponds to an event that has taken place, will take place, or could take place in the "real world," and a query corresponds to outside observation of the "real world," as illustrated in Figure 1.1. Note that in all of the above informal descriptions of transactions, transactions seem small in terms of the size of the part of the database that is accessed by the transaction. This is probably because the transactions model events in the "real world," and in the real world, there is a locality principle: objects cannot in general affect each other unless they are "close", and few objects can in general be simultaneously "close". In any case, in most transaction processing systems all transactions seem to be small in this sense. However, it is easy to imagine useful queries of all sizes, as should be clear from the above examples. These general properties of queries and transactions have implications for transaction processing system design, and will be discussed later.

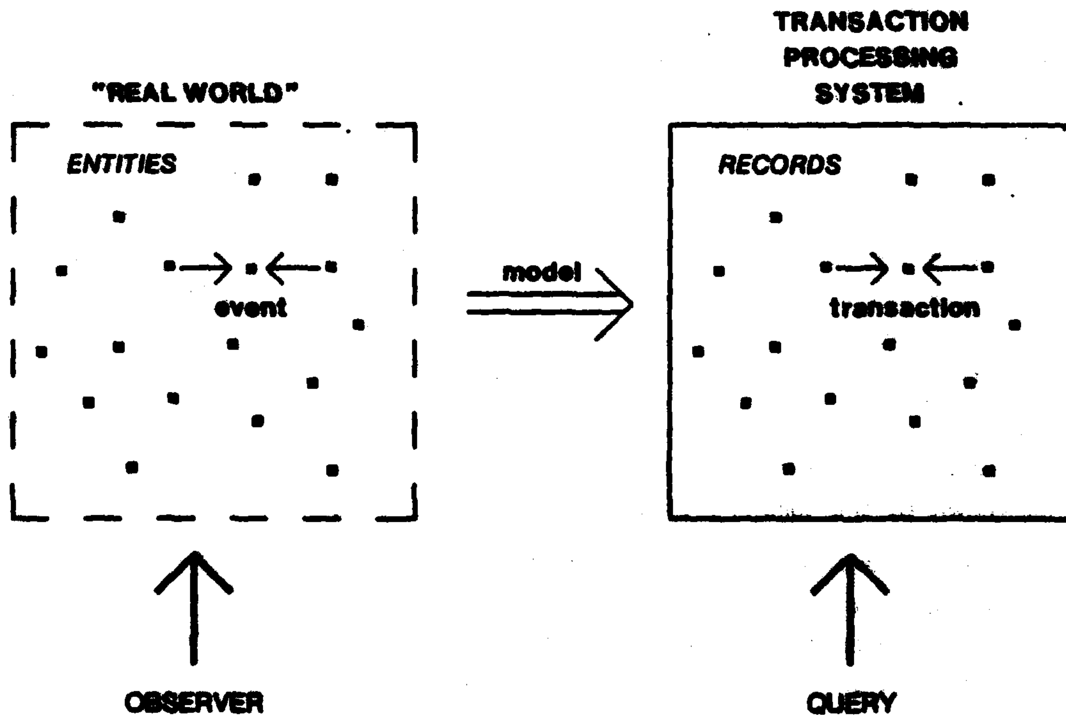


Figure 1.1. Transaction Processing

The problem of guaranteeing external consistency is usually approached by protection mechanisms -- the goal is to allow each user to modify only those parts of the database for which that user can be trusted or assumed, first, to know the true state of the corresponding part of the "real world", and second, to enter this information correctly. This is the transaction side of protection; there is also a query side of protection, where the goal is to allow each user to access in a query only those parts of the database for which that user has a right to examine.

1.2. Problems of Transaction Processing

Ignoring for the moment problems of database design, those problems introduced by transaction processing will now be considered. Two problems of transaction processing -- increasing confidence that a procedure is in fact a transaction, and protection -- have already been mentioned above. However, these problems are not unique to transaction processing: the first problem can be seen as an instance of the more general problem of program verification and testing, and the problem of designing protection mechanisms -- a problem for any system with shared resources -- does not seem significantly changed by the nature of transaction processing (however, protection policies may be more complex, as in statistical databases).

Another problem is that of *recovery*. The database can become internally inconsistent due to hardware failures or software errors; it can also become externally inconsistent due to human errors. In such cases it is desirable to restore the database to some earlier state that is believed to be consistent, relying on the hopefully increasing reliability of the lower levels of memory hierarchies for recovery of the previous state. However, it is also desirable to undo as little as possible, that is to restore the database to the most recent such state. This latter goal can be very important in some transaction processing applications (e.g., for economic reasons), thus in a sense making the problem unique to transaction processing, even though it is a desirable goal for any system.

The final problem is that of concurrency: even if transactions individually preserve internal consistency, concurrent execution of transactions may cause internal consistency to be lost, as shown by the following simple example.

> The database consists of four integer variables X, Y, Z, and W. Furthermore, internal consistency requires that $X+Y+Z+W = 4$. If currently $X = 1$, $Y = 3$, $Z = W = 0$, consider the following interleaved execution of two transactions A and B (Z and W are unused here, but will be referred to below). Note that *temp* is a local variable for each transaction.

	Transaction A	Transaction B	$temp_A$, $temp_B$, X, Y, Z, W
	move 1 from X to Y	*move 1 from Y to X*	--, --, 1, 3, 0, 0
(1)	$temp := X$		1, --, 1, 3, 0, 0
(2)		$temp := Y$	1, 3, 1, 3, 0, 0
(3)		$Y := temp\text{-}1$	1, 3, 1, 2, 0, 0
(4)		$temp := X$	1, 1, 1, 2, 0, 0
(5)	$X := temp\text{-}1$		1, 1, 0, 2, 0, 0
(6)	$temp := Y$		2, 1, 0, 2, 0, 0
(7)	$Y := temp+1$		2, 1, 0, 3, 0, 0
(8)		$X := temp+1$	2, 1, 2, 3, 0, 0

Clearly, each transaction individually preserves $X+Y+Z+W = 4$, but the result is that $X+Y+Z+W = 5$. A transaction processing subsystem that prevents or resolves interactions such as this will be called a *concurrency control.*

At first this problem may not seem significantly different from the general *synchronization* problem (see [Andler 79] for a survey). However, there is a fundamental difference: in general, *the objects that will be accessed by a transaction cannot be determined in advance* of the actual accesses. This is in contrast to an operating system, say, where the shared data structures accessed by a particular module are usually determined at design time. The problem is more closely related to that of allocation of shared resources, with no prior claiming of resouces. This similarity has led historically to locking-style concurrency controls, that is, concurrency controls in which access to an object is in some cases restricted to at most one transaction. However, there are many more concurrency controls than locking concurrency controls (for example, see Chapter 5). The difference is that concurrent access to an object need not be disastrous, as it usually would be if it were to a tape drive, for example.

The reason that accesses cannot be predicted in advance is that the actions taken by a transaction are in general dependent on the data read. This can be true at all levels of the system. For example, at the conceptual level (see Chapter 2), it is easy to imagine transactions of the form: "for all X, Y satisfying certain constraints, if $X < Y$ then update X, otherwise update Y".

An example at the physical access level is the use of dynamic index structures (see Appendix I). In such cases, the set of objects that will be accessed by a transaction, other than the root of the index, is completely unknown prior to execution (see Figure 1.2).

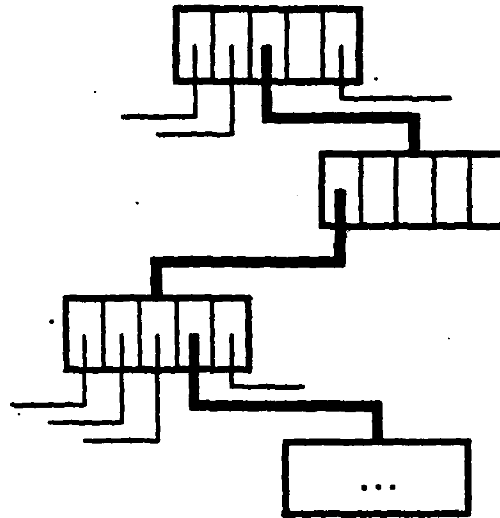


Figure 1.2. Access Path Determined at Execution-Time

Since accesses cannot in general be predicted in advance, it will be necessary to abort transactions (i.e., undo all modifications to the shared database). Consider the example of transactions *A* and *B* above. Until step (4) is reached, it just as well could have been the case that *A* would access *X* and *Z* only and that *B* would access *Y* and *W* only. However, once step (4) is reached, one of the two transactions will eventually have to be aborted. Also, it is desirable to allow users or application programs to abort transactions.

These, then, are some of the problems introduced by transaction processing. However, solutions to these problems must take into account the underlying database system. Some of the inadequacies in existing database systems result from needs to process (1) *more data* for (2) *more users*, (3) *more quickly*, (4) *more reliably*, and (5) *less expensively*. One current approach to these inadequacies relies on the rapidly decreasing cost of processing power. Apparently, though, the cost of large, reliable memory and various special devices (such as high-quality printers) is not decreasing nearly so rapidly. These economic considerations have led to architectures such as that of Figure 1.3, in which (cheap) processing power is distributed, but (expensive) large, reliable memory and special devices are shared.

The architecture of Figure 1.3 seems ideally suited for essentially non-shared, personal applications; the problem introduced by transaction processing in this case is to use this architecture effectively in a highly-shared application. For example, solving the concurrency control problem by executing transactions sequentially (batch processing) would in general be unacceptable since only the central computer would be utilized to any degree at all (this

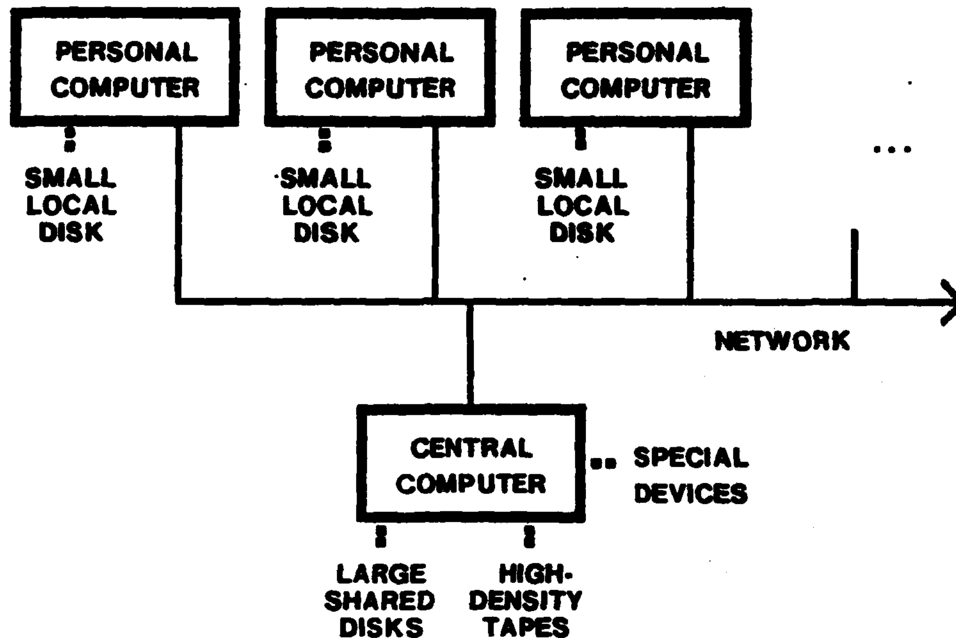


Figure 1.3. Personal Computer Network with Shared Expensive Devices

is often true for uniprocessor systems as well, where there may actually be parallel processing due to multiple I/O devices).

1.3. Problems Considered Here

The major problem considered here is that of concurrency control design, subject to two constraints. The first constraint is that the concurrency control should be as application independent as possible. The advantages of this are overwhelming, and are more fully discussed in Chapter 2.

The second constraint is one of efficiency. The architecture of Figure 1.3 is just an example; the concurrency control should apply to other architectures as well. A problem is that a particular concurrency control could be efficient for some architectures and not for others. Even given the architecture, there are many variables, such as network and memory bandwidths, probability of transaction conflict, average transaction and query size, etc. The second constraint, then, is that the concurrency control should be *general*: *all* concurrency controls, subject to certain explicit design criteria (such as application independence), should be *realizable* -- there should be *no* hidden design criteria. This results in a design that can be tailored for a specific environment by fixing the remaining design criteria.

Another problem considered here is that of query size. As noted above, queries can be arbitrarily large. But queries do not modify the database; it seems that it should be possible to run queries without any concurrency control interaction. In fact this is the case, and several solutions will be presented, but the solutions rely on multi-version objects. This then raises the problem of garbage collecting old versions, which is considered here as well. Multi-version objects are also of use in distributed systems for determining when cached copies of shared objects are "out-of-date", and they are generally useful for recovery purposes.

A related problem is one of transaction size. Even if most transactions are small, there may be some occasional large transactions (e.g., a transaction that physically reorganizes the database). Furthermore, poor database design can result in conceptually small transactions physically accessing large parts of the database. This is known as the *granularity* problem, and can be solved by hierarchical concurrency controls, such as that discussed in [Gray 78]. Another approach is to attempt to design the database so that large transactions are rare. This is the approach assumed here, and for this reason, earlier work related to one aspect of this problem -- design of dynamic index structures -- is presented in Appendix I. This is also a problem in design at higher levels: an example often used to illustrate a large transaction is one that raises all salaries in an employee file by 5%, say -- however, if salaries were not stored in absolute terms, but instead were stored as relative values, with a single data item giving the conversion, this apparently large transaction becomes very small. Furthermore, this design can be generalized if necessary to contain a number of conversion factors for different classes of employees. This example is mentioned only to show that the necessity for large transactions is not always clear. In any case, even if there are large transactions, in most applications they will be rare, and it is trivial to generalize non-hierarchical concurrency controls to simple hierarchical concurrency controls -- for example, an option can be added so that transactions can request that the entire database be locked. Finally, the effective use of hierarchical concurrency controls requires advance knowledge of the behavior of some transactions, contrary to the first constraint above. For these reasons hierarchical concurrency controls are not considered here.

1.4. Summary of this Work

The problems of transaction processing considered here have been described above. A goal of this thesis is to solve these problems in such a fashion that the designs will be usable in a wide variety of applications and systems. This problem is made more difficult by the fact that an initially acceptable design could later prove to be unacceptable if the design assumptions are violated. Furthermore, this difficulty is becoming ever larger as the varieties and uses of multiprocessor and computer network systems grow: due to the added complexity of these systems, first, there are in general many more alternative designs than in a uniprocessor system, and second, it is much more difficult to predict in advance how the alternatives will compare.

One well-known approach to this problem is information-hiding -- by isolating design decisions, and decomposing the system into modules that hide these decisions, a much more flexible design results. In Chapter 2 a design framework for transaction processing systems is developed based on information-hiding principles. This framework has two subsystems, a concurrency control and a memory manager, that control access to all shared data objects (it is assumed that in distributed systems the memory manager is decomposed into a number of local memory managers, and a global memory manager). Due to these subsystems, the framework can be used in a wide variety of systems, some examples of which are given. The properties the concurrency control must satisfy are also developed in Chapter 2 using a formal model that has sufficient detail to be immediately applicable in practice.

In Chapter 3 an overview of a system using the decomposition of Chapter 2 is presented. The design decisions that are common to various subsystems are discussed, and the communication protocols between subsystems are given.

Another approach to the generality problem may be called *correctness/policy separation*. When designing a module to solve a given problem, decisions are made at several levels, ranging from the level of fundamental correctness to the level of pure policy. For example, in the case of concurrency control, the module must be designed so that internal inconsistency of the database is never allowed due to interactions between concurrent transactions -- this is a fundamental correctness property. It is also desirable in some applications that when transactions conflict, the transaction with the earlier starting time be given priority -- clearly a policy. By separating these levels of decisions, and then applying information-hiding, a general design results. In Chapter 4 a design paradigm for concurrency controls is developed using *only* those assumptions that are necessary for correctness, application independence, and practicality. This paradigm has the property that regardless of the decisions made by any particular policy, the concurrency control will remain fundamentally correct.

Policies may be defined statically at design-time, or they may be defined dynamically by *policy modules* that are executed at run-time. This latter apporach makes it very easy to experiment with policies, since policies can be changed even while the system is in use. Furthermore, this makes possible a new area of research in concurrency control design: that of designing concurrency controls that dynamically adapt to system usage so as to optimize performance.

The property that the concurrency control remains fundamentally correct regardless of the policy is also useful for designing and maintaining policies, since the policy designer has a high degree of freedom. As an illustration of this freedom, in Chapter 5 a set of basic policies is developed in which all transactions are treated uniformly without priority: the result is 330 distinct policies. These policies should be considered only as a beginning in the study of policies, since in practice there are a variety of extensions that will prove

valuable. Two extensions that are often necessary, deadlock detection and queuing of requests, are described.

In Chapter 6 several designs for a global memory manager are developed. This subsystem supports multi-version objects, and is used to solve the granularity problem for queries by making it unnecessary for queries to interact with the concurrency control. A design for garbage collection, in which old versions of objects are deleted at the earliest point at which it can be guaranteed that they will never again be accessed, is also presented.

A complete transaction processing system using these designs was developed for the Cm* distributed multi-microprocessor. This system used a concurrency control in which all functions required by the paradigm were available, and the concurrency control used a policy module implementing all basic policies, any of which could be chosen at run-time. Record managers were replicated, and copies of shared data objects were cached. The throughput limitations of this system were investigated, and experiments were performed using several policies. This system and the experiments are described in Chapter 7. A generalization of the record manager used in this system is given in Appendix I, and algorithms developed for the concurrency control are given in Appendix II. These algorithms should apply directly to any transaction processing system using the framework of Chapter 2.

Chapter 8 contains conclusions and a discussion of further research.

1.5. Relationship to Previous Work

In early work on the concurrency control problem, the two-phase locking protocol was developed ([Eswaran et al 76], [Stearns et al 76]). An implicit assumption behind two-phase locking is that transactions should be controlled so as prevent aborts if at all possible. Starting from a different premise, that transactions should never wait for access to an object, a radically different *optimistic method* for concurrency control was developed in [Kung and Robinson 81]. As an example of the difference between the two approaches, in an optimistic method deadlock will never occur (since transactions never wait), and so deadlock detection is unnecessary; on the other hand, using an optimistic method, transactions are much more likely to be aborted. What the performance differences would be between the two approaches in any given application was unknown. This suggested the problem of designing a more general concurrency control that would be capable of both methods: in an application, experiments could easily be conducted (since both methods are fundamentally correct in the same sense), and the "best" method could then be used. This problem was the starting point for this thesis, generalized as follows: the general concurrency control should be capable of *any* method that satisfied certain explicit assumptions. Of these assumptions, the primary one is that the concurrency control must guarantee serializability (see Section 2.5); the remaining assumptions have to do with application independence and practicality (see Section 4.5).

No practical concurrency control can be designed without regard for the granularity problem,

but as discussed above, this seems to be a significant problem only for queries. This problem is nicely solved using multi-version objects, and is especially appropiate for the concurrency control design developed here, since serializability is guaranteed in an explicitly known order (see Section 4.5 for a discussion on why this is necessary). The use of multi-version objects in distributed database systems has previously been investigated in [Reed 78]. The major differences between their use by Reed and their use here are: (1) as used by Reed, version numbers (pseudo-times in Reed's terminology) are determined during the course of a transaction, whereas here, sequentially numbering the successful commits of transactions, the version numbers of the objects written by a transaction are the same as its commit number; (2) due to the distributed nature of version number assignment, Reed must largely avoid the garbage collection problem, and queries require concurrency control support, whereas here version numbers are managed by conceptually centralized entities, with the results that garbage collection algorithms can be developed, and that queries do not require concurrency control support.

The mapping of the transaction processing system framework of Chapter 2 to a distributed architecture was influenced by the Medusa operating system (a general-purpose operating system for Cm* -- see [Ousterhout et al 80]), and by the work of Garcia-Molina (see [Garcia-Molina 79]). If Medusa were extended to be a *database* operating system (in the sense of [Gray 78]), using the mapping of Chapter 2, the global memory manager and concurrency control would be seen as new utilities, and the local memory manager would be seen as a new type of kernel. An alternative design would be to include the concurrency control and global memory manager as part of a kernel (a copy of which runs on every node) -- however, Garcia-Molina simulated a variety of centralized and distributed concurrency controls, and found that the centralized concurrency controls in most cases gave better performance.

Finally, the literature for concurrency control has now grown very large, as can be seen from the recent survey [Bernstein and Goodman 81]. Although much of the previous work on concurrency control has had a strong influence here, as Bernstein and Goodman conclude, all the various designs can be seen as combinations and variations of a few basic techniques. A major difference of the approach here is that fundamental correctness has been separated from policy. This approach was motivated by the design of the Hydra operating system (see [Wulf et al 74]), and by the work of Everhart [Everhart 79].

2. Design Principles

An overall structure for transaction processing systems will now be described, based on information-hiding/data-independence principles. This is followed by a formal development of the notions of serializability and conflicts, forming a basis for concurrency control design.

2.1. On the Criteria to be Used in Decomposing (Transaction Processing) Systems into Modules

The title of this section refers, of course, to the paper by Parnas [Parnas 72]. As discussed there and elsewhere, one approach to system design is information-hiding, i.e., the decomposition of the system into modules based on difficult or changeable design decisions, with each module designed so as to hide one decision from the others. This approach has been demonstrated to have great advantages in decreasing system development and maintenance time, and in increasing confidence in system correctness. These advantages are widely believed to outweigh any resultant efficiency disadvantages, if in fact any such disadvantages do result.

The information-hiding principle, when applied to database design, has come to be known as *data-independence*, which for example Date defines as "the immunity of applications to change in storage structure and access strategy" [Date 77].

The most far-reaching decisions in database design are those of choice of data models. This is comparable to the more general problem of choice of data structures in programming. A problem in database design is that there are many data models that are variously most appropiate depending on the level at which the design is approached, ranging from the physical access level, where one would naturally like to think of a "data item" as a disk block or segment, to the user interface level, where one might think of a "data item" as a record of some type, or as a collection of records of the same type (e.g., a relation), or perhaps as some kind of entity suitable for display on a CRT.

This is a common problem of large systems, and can be solved by abstraction. In database design, separation of the design decisions on data models at three levels of abstraction -- the physical access (internal) level, the logical access (conceptual) level, and the user access (external) level -- has led to the so-called ANSI/SPARC three-level DBMS (database management system) architecture (see [Jardine 77] for a presentation and discussions of this architecture).

This architecture has the property that various different external data models may be simultaneously supported. Similarly, different internal data models may be supported. Thus, this architecture lends itself to data-independence -- the conceptual model provides a fixed interface between changeable application programs (i.e., external/conceptual mappings) and changeable storage-organization/access programs (i.e., conceptual/internal mappings).

In summary, this three-level architecture is shown in Figure 2.1. Here, a *user interface*

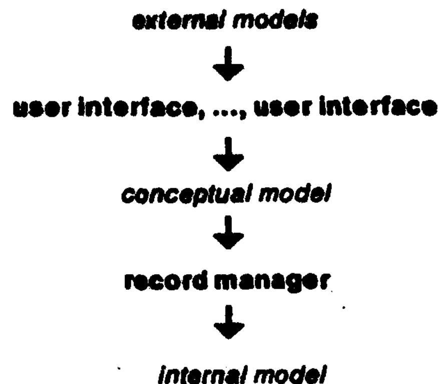


Figure 2.1. Three-Level Architecture

is a collection of modules providing a single external/conceptual mapping, and the *record manager* is a collection of modules that define the conceptual model and its mapping to physical storage.

2.2. Problems of the Three-Level Architecture

The three-level architecture seems to deal adequately with the problem of isolating application programs from storage structures and access strategies, providing one is willing to accept the essentially static nature of the conceptual model. However, it provides no such isolation at the physical access level (which could either be at the level of physical storage devices, or at the level of virtual storage as seen through a host operating system). Typically, the record manager has detailed knowledge of available storage and its characteristics, *in addition* to implementing concurrency control and recovery.

Apparently this has not been a significant problem in the past, as far as system correctness is concerned, since there are reliable transaction processing systems in existence. However, it has probably contributed significantly to the development time for these systems. Ideally, one would like to take existing record management software, or new software, and use it in a system without regard to the underlying machine architecture. Also, one might want to change record management policies as system usage changes (as part of system maintenance), implement a new record manager, or correct bugs in an existing record manager, without regard to possible interactions with concurrency control or recovery that could cause these subsystems to become incorrect.

These problems of the three-level architecture will become ever more significant given the increasingly complex hardware provided by multiprocessors, distributed processors, computer networks, and memory hierarchies. In fact, system correctness could very well become a significant problem.

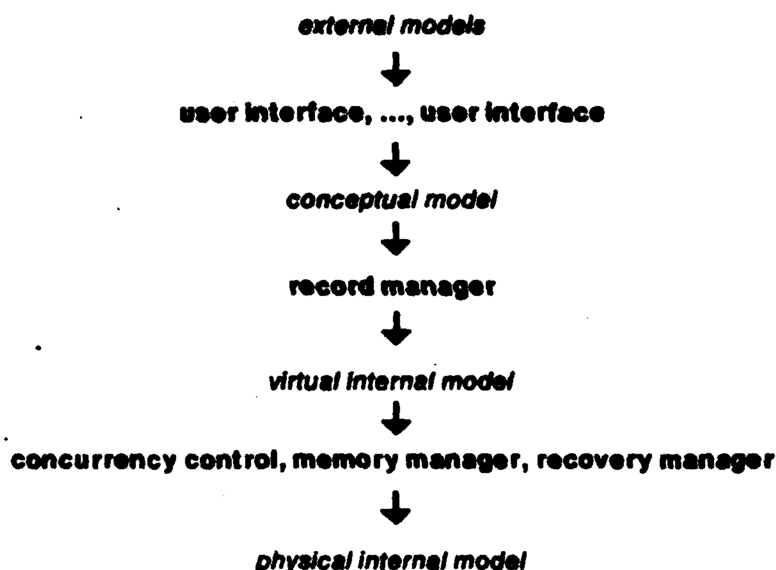


Figure 2.2. Four-Level Architecture

2.3. A Four-Level Architecture

Separating record management design into two levels -- the virtual and physical levels -- results in a general architecture like that of Figure 2.2. Here, the *physical internal model* refers to the hardware itself (possibly seen through a general-purpose operating system), and the *virtual internal model* refers to a data model based on a collection of abstract data objects, and in which all details of concurrency, memory hierarchies, and recovery are hidden.

The data objects supported at the virtual internal level could be of varying complexity, depending on the design. Some examples, in order of increasing complexity, are pages, segments, and records (see Section 3.2). Refering to Figure 2.2, the record manager is responsible for mapping the entities defined by the conceptual data model onto these objects, accessing an object using only the operations defined on that object; the memory manager is responsible for mapping these objects onto physical storage; the concurrency control is responsible for detecting and resolving conflicts (see Section 2.5); and the recovery manager is responsible for saving enough of the database state on highly reliable memory to allow recovery in case of failures.

The advantage of a four-level architecture such as this is that it solves these problems of the three-level architecture mentioned above, that is, it provides hardware-independence at the (new) record management level.

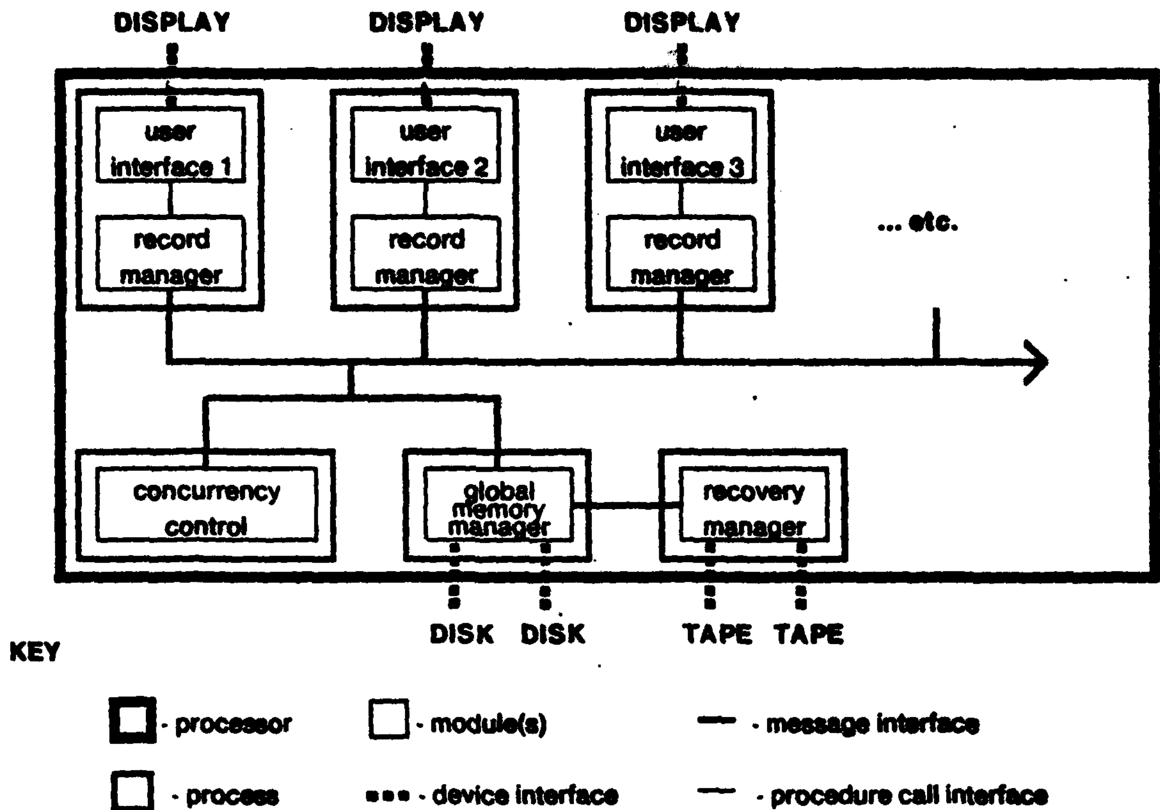


Figure 2.3. Centralized System

2.4. Example Systems

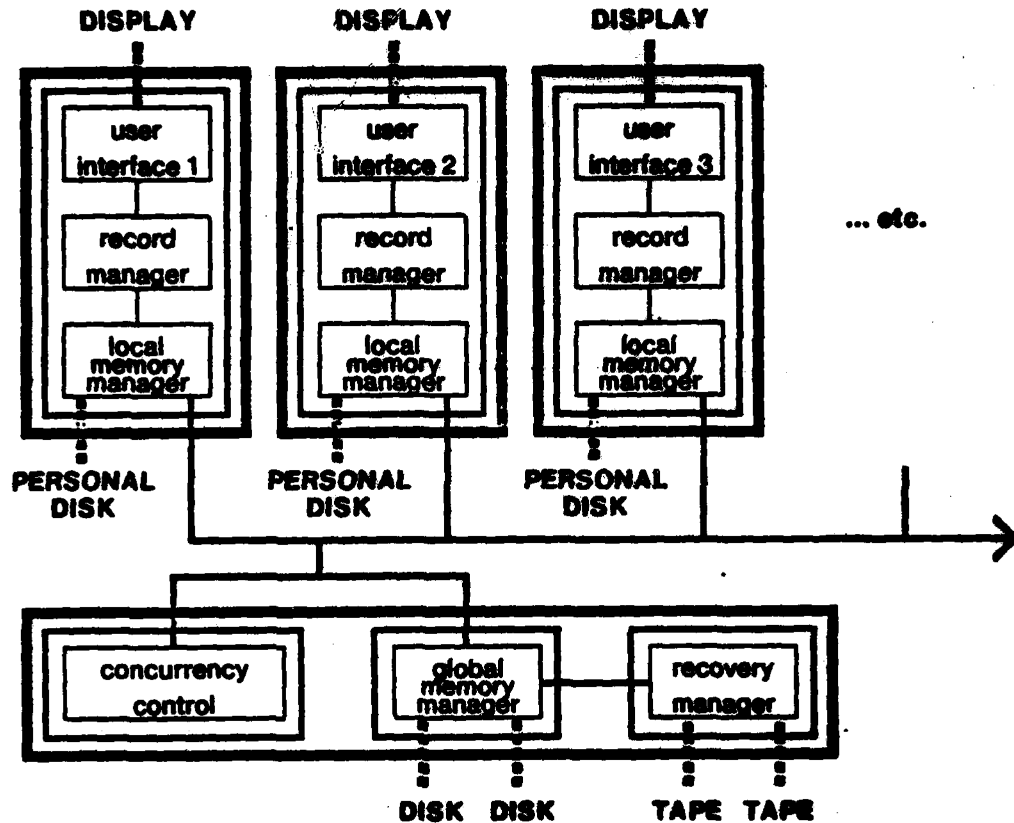
The four-level architecture can be mapped to a variety of architectures in a variety of ways. Three examples will now be given.

2.4.1. A Centralized System

The four-level architecture can be used in a straightforward way in a centralized system, as shown in Figure 2.3. Note that in this system, record managers can share code, and that the record managers, memory manager, and recovery manager can share buffer space. This could just as well be a tightly coupled *multiprocessor* system, that is, a multiprocessor system in which access to shared memory is equally inexpensive for all processors.

2.4.2. A Personal Computer Network System

One possible mapping of the four-level architecture to a personal computer network is shown in Figure 2.4. In this system there is a more complex memory hierarchy: there are a number of small local memories, and a large shared memory. One way to approach this problem is *functional distribution*. In this approach, access to a particular resource is



KEY

- processor
- module(s)
- message interface
- process
- device interface
- procedure call interface

Figure 2.4. Personal Computer Network System

confined to a resource manager that is physically close to the resource. In this case, the result is a number of *local* memory managers that are responsible for each local memory, and a *global* memory manager and recovery manager that are responsible for the shared memory (note that the global memory manager and recovery manger can share buffer space). One could imagine local recovery managers as well, but this is not dealt with here. Furthermore, considering the information required to detect conflicts (see below) at a resource, there is also a central concurrency control that "manages" this resource.

An alternative approach is distribution by *replication*. For example, if there were no shared memory, only local memories would be left. In such a case, there could be replicated

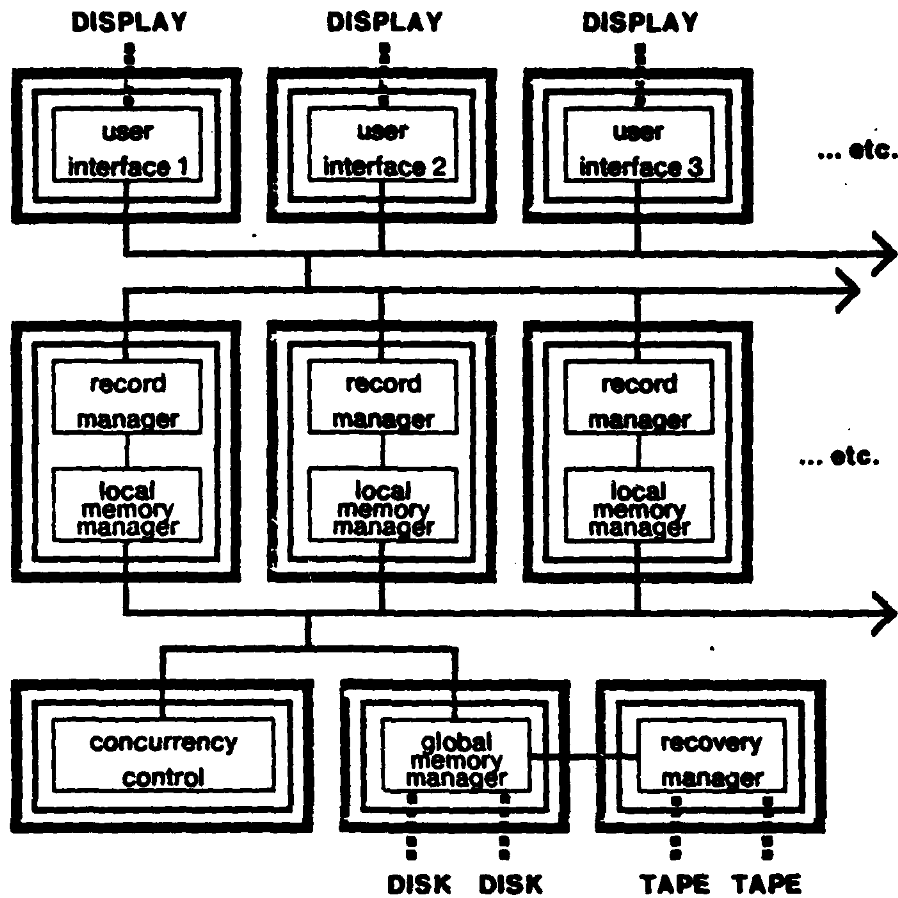


Figure 2.5. Multi-Microprocessor System

memory managers at each processor, each managing *all* local memories as if this were one large shared resource. The same approach could be applied to the concurrency control or to the recovery manager. Some disadvantages of this approach are: (1) the required process communication is much more extensive; (2) problems such as distributed deadlock seem very difficult to handle efficiently; and (3) system correctness is in general more in doubt, due to the added complexity of the system. In fact, in the case of concurrency control, simulation studies by Garcia-Molina (see [Garcia-Molina 79]) have shown that, in a variety of cases, centralized concurrency controls are much more efficient than comparable distributed concurrency controls.

2.4.3. A Multi-Microprocessor System

A system using the designs of Chapters 3-6 was implemented on Cm*, a multi-microprocessor system (see Chapter 7). A proposed decomposition for multi-microprocessor transaction processing systems is shown in Figure 2.5. This was the decomposition used in the implementation, except that the user interfaces were combined in a single master process that simulated the transactions produced by a number of users, and an extremely simple recovery manager was implemented as part of the global memory manager. The decomposition of Figure 2.5 is essentially the decomposition of Figure 2.4; however, address-space limitations have resulted in some further decomposition, as shown.

In the case that the concurrency control or global memory manager become system bottlenecks, it is possible to distribute these among several processors without sacrificing functional decomposition by partitioning the set of shared data objects (see Sections 4.6 and 6.5).

2.5. Serializability and Conflicts

In this final section the problem of concurrency control design is addressed. The goal here is to obtain a formal characterization of the kinds of interactions under concurrency that can lead to loss of consistency, assuming only that each transaction individually preserves consistency.

Let the following constants and sets be given:

- k, the number of transactions in the system;
- O, the set of object IDs;
- D, the set of object states (all possible data parts of an object);
- T, the set of transaction states, including, in particular, a halting state;
- **R**, **W**, read and write symbols (arbitrary, different constants).

First, versions of objects and database states will be defined.

Definition. A *version* is a triple $\langle o, v, d \rangle$, where $o \in O$, v is an integer, $0 \leq v \leq k$, and $d \in D$. In the triple $\langle o, v, d \rangle$, o, v, and d are called the *object ID*, *version number*, and *data*, respectively. A *database state* is a set of versions satisfying:

(1) for every $o \in O$, there is a version with a zero version number, $\langle o, 0, d \rangle$;

(2) for all object IDs o and version numbers v, there is at most one version with ID o and version number v.

Next, transaction steps and transactions are defined.

Definition. A *transaction step* is a sextuple $\langle i, j, C, P, R, W \rangle$, where i and j are integers, $1 \leq i \leq k$, and C, P, R, and W are any functions

$$C: T \rightarrow \{\mathbf{R}, \mathbf{W}\},$$

$$P: T \rightarrow O,$$
$$R: DxT \rightarrow T, \text{ and}$$
$$W: T \rightarrow DxT.$$

Here, i is called the *transaction number*, j is called the *sequence number*, and C, P, R, and W are called the *conditional function*, *parameter function*, *read function*, and *write function*, respectively. A *transaction* is a finite sequence of transaction steps, all with the same transaction number, and with unique, increasing sequence numbers.

In formalizing the notion of transaction systems, it is convenient to use a variable state vector $\langle s, t_1, t_2, t_3, ..., t_k \rangle$, where s is a database state and each t_i is a transaction state. Given the current value of the state vector and a transaction step $\langle i, j, C, P, R, W \rangle$, a new value of the state vector is produced as follows. First, if t_i is the halting state, the state vector is unchanged. Otherwise, C is applied to t_i. The result of C determines which of the following two cases apply.

Read case: $C(t_i)$ = R. In this case, R is then applied to $\langle d, t_i \rangle$, where d is the data of that version with object ID $P(t_i)$, and with the greatest version number less than or equal to i. The result of R is a transaction state, which is the new value of t_i in the state vector; the rest of the state vector is unchanged.

Write case: $C(t_i)$ = W. In this case, W is then applied to t_i, giving a data value d and a transaction state t. The new value of the state vector is derived by: (1) setting t_i to t; (2) modifying s, first by removing the version with object ID $P(t_i)$ and version number i from s if there is such a version, and then by adding the version $\langle P(t_i), i, d \rangle$ to s; and (3) leaving the rest of the state vector unchanged.

Next, serial and concurrent transaction systems are defined.

Definition. A *serial transaction system* is any sequence of transaction steps formed by appending k transactions, with transaction numbers 1, 2, 3, ..., k, in this order. A *concurrent transaction system* is any sequence of transaction steps formed by permuting the steps of a serial transaction system subject to the constraint that the sequence numbers for each transaction remain increasing for that transaction.

A serial or concurrent transaction system can be applied to an initial value ot the state vector by applying each step of the system in sequence, yielding a final value of the state vector. A *transaction history* of this process is a sequence, initially empty, formed by appending a quadruple for each step in the transaction system, with the exception of steps for transactions in the halting state. This takes place as follows. Let the current transaction step be $\langle i, j, C, P, R, W \rangle$. If t_i is the halting state, the transaction history is unchanged. Otherwise:

If $C(t_i)$ = R and $P(t_i)$ = o, append $\langle R, i, j, o \rangle$ to the history.

If $C(t_i)$ = W and $P(t_i)$ = o, append <W, i, j, o> to the history.

Now, consider the problem of preserving consistency. First, if each transaction individually preserves consistency, a serial transaction system clearly preserves consistency. Therefore, if a concurrent transaction system were somehow equivalent to a serial transaction system, i.e. *serializable*, it too would preserve consistency. In fact, it has been shown in [Kung and Papadimitriou 79] that this is the *weakest* such condition for a concurrent transaction system to preserve consistency if the consistency properties are not known (and of course, they are not known in an application-independent concurrency control). This leads to the following definition.

> **Defintion.** A concurrent transaction system is *serializable* (in the order 1, 2, ..., k) if, when applied to any initial value of the state vector, the final value is identical to the final value produced by applying the serial transaction system from which the concurrent transaction system was formed. A transaction history is *serializable* (in the order 1, 2, ..., k) if *all* concurrent transaction systems with this history are serializable.

This definition is somewhat different than that usually appearing in the literature, in that the serrializability order is assumed to be given. The question of whether this lack of generality in the definition above leads to any lack of generality in the concurrency control is taken up in Chapter 4 (the answer seems to be that it does not).

Now, conflicts are defined.

> **Definition.** Given a transaction history containing <R, i, j, o>, let <W, i', j', o> be that quadruple in the history with maximal i' and j', subject to $i' \leq i$, if such a quadruple exists. Then transactions i and i' *conflict* if $i' < i$ and <R, i, j, o> precedes <W, i', j', o> in the transaction history.

The result of this section is the following theorem.

> **Conflict Theorem.** Assume that D has more than one element, and that T has more than one non-halting state. Then a transaction history is serializable if and only if no two transactions conflict in the transaction history.

Proof.

($\Leftarrow$) Given a concurrent transaction system and an initial value of the state vector, note that (1) the state vector transition produced by a transaction step <i, j, C, P, R, W> depends only on the current value of t_i, and, in the read case, that previous transaction step <i', j', C', P', R', W'> with maximal i', j', $i' \leq i$, that was a write to object $P(t_i)$, if such a step exists; and (2) the only transaction state in the state vector changed by this transaction step is t_i. Therefore, if there are no conflicts in the transaction history, all state vector transitions in the concurrent transaction system will be the same as the state vector transitions in the serial transaction system from which the concurrent transaction system was formed.

(→) Given a transaction history with a conflict, a concurrent transaction system and an initial value of the state vector will be found such that the final value of the state vector is not the same as the final value produced by the corresponding serial transaction system. Since there is a conflict, the transaction history is of the form:

$$\ldots \langle R, i, j, o \rangle \ldots \langle W, i', j', o \rangle \ldots \quad (i' < i).$$

Let $D = \{x, y, \ldots\}$ and $T = \{halt, a, b, \ldots\}$. O has at least one element (namely o); let $O = \{o, p, q, \ldots\}$. Let the initial state vector be

$$\langle \{\langle o, 0, x\rangle, \langle p, 0, x\rangle, \langle q, 0, x\rangle, \ldots\}, a, a, a, \ldots \rangle,$$

that is, every object has one version with version number 0 and with data x, and every transaction is in state a. Let there be one transaction step in the concurrent transaction system for every quadruple in the transaction history, defining the conditional and parameter functions of each step so as to agree with the history. Now define every write step in the concurrent transaction system, except for the one corresponding to $\langle W, i', j', o\rangle$ above, as a write with data x and a transition to the current transaction state. Define the remaining write step as a write with data y and a transition to the current transaction state. Finally, define every read step as a transition to the current transaction state, except for the read step corresponding to $\langle R, i, j, o\rangle$ above; define the read function R of this read step as $R(x,a) = a$, $R(y,a) = b$. This concurrent transaction system is not serializable, since the final value of t_i is a, but serially the final value is b.∎

This simple and exact characterization of serializable transaction histories is possible primarily due to the inclusion of an explicit total ordering of transactions in the definition of serializability, and to the multi-version definition of objects. When these are omitted the characterization of serializable histories becomes, by comparison, highly complex -- in fact, the problem of determining if a transaction history is serializable in any order, even under a much simpler data model, has been shown to be NP-complete (see [Papadimitriou 79]).

Transaction histories are useful for concurrency control design since transaction histories formalize the information available to an application-independent concurrency control. Since the concurrency control is application-independent, it must not allow any transaction histories to develop that *could* have been produced by some non-serializable concurrent transaction system -- this was the motive for the definition of serializable transaction histories given above. Finally, the conflict theorem provides a simple way to test for non-serializable transaction histories. For the system described here, the test is actually somewhat simpler than might be expected from the above, since for each transaction, all reads precede all writes (to shared objects), and there is at most one write to a shared object. The conflict theorem will be applied in Chapter 4.

3. System Overview

In Chapters 4, 5, and 6, designs for the concurrency control and global memory manager subsystems of the four-level architecture will be developed. This chapter provides an overview of a complete system using this architecture. Global design decisions, such as communication protocols between subsystems, are discussed. It is assumed that the system is distributed, so that the memory manager has been decomposed into local and global memory managers, as described in Chapter 2. Although this overall design was developed in the course of the Cm* implementation (Chapter 7), it should apply to any transaction processing system using the four-level architecture. The following abbreviations will be used: *RcdM* - record manager, *LMM* - local memory manager, *GMM* - global memory manager, *CC* - concurrency control, *RcvM* - recovery manager.

3.1. Communication between Subsystems

It is convenient to have communication between LMMs and the GMM, LMMs and the CC, and between the GMM and the RcvM take place via messages -- in this case no other synchronization will prove necessary. The assumptions here are that there is a message buffer associated with each process; that sending a message to a process causes the message to be placed at the end of the message buffer for that process if possible, otherwise the sending process waits until it is possible to do so, with queueing of waiting processes; and that receiving a message removes the first message from the message buffer if there is one, otherwise the receiving process waits until there is a message to remove.

Each RcdM and LMM are part of the same process, and they share a common address space. Communication between the RcdMs and LMMs can take place by procedure calls.

3.2. Data Objects

At the virtual internal level the database consists of a collection of *data objects* (when the context is clear, simply *object*), each identified by a unique *ID*. A data object will be the unit of data transfer between local memories and shared memory.

For the Cm* system, *pages* (units of untyped storage of fixed size) were chosen as the data objects of the virtual internal level primarily for simplicity. There are only three operations defined on a page -- read, write, and delete -- in addition to the operation of creating a new page. A more advanced system could provide more complex objects, such as segments (units of untyped storage of variable size) and records (units of structured storage -- see Appendix I). One advantage of only using pages (or segments) is generality: no commitment is made to any particular data model. On the other hand, providing record objects at the virtual internal level could be far more efficient, and this approach would certainly be taken if record access hardware, such as logic-per-track disks, were available.

It is the responsibility of the RcdM to map the entities defined by the conceptual data model onto data objects in an efficient, flexible way. In the case that data objects are records of

some type, this will be trivial if the conceptual entities are records of the same type. The mapping becomes more complex as the difference between the conceptual model and the virtual internal model grow. In the case that data objects are pages and conceptual entities are records, various mappings can be achieved by organizing the database as a directed graph of pages, with certain *root* pages that are never created or deleted -- access to a record takes place by first accessing a root page, then following pointers to a page (or pages) containing the desired record. A simple example of this kind of mapping is to link a number of pages together linearly so as to form a sequential file; for a much more complex example, see Appendix I.

3.3. Read and Write Phases

As noted in Chapter 1, in general, any transaction may be aborted. Therefore, in order to avoid unnecessary transfer of data objects, all writes to the shared database will be buffered until the end of the transaction. This results in a *read phase*, in which the transaction is executed but does not write to the shared database, and then if the transaction is not aborted, a *write phase* in which all modified data objects are transferred to the shared database.

Some other reasons for choosing this approach are (1) it simplifies the GMM design; (2) the LMM cannot determine if a transaction will later modify an already modified object, without adding complexity to the RcdM/LMM interface; and (3) the object versions are not known until the end of the read-phase (see the next section).

3.4. Transaction Numbering and Versions

In Chapter 4, it will be seen that the concurrency control will guarantee that the system transaction history is always serializable. Furthermore, the serializability order, that is the transaction numbering, will be made explicit. Thus, the database can be thought of as a sequence of versions D_0, D_1, D_2, ..., where D_0 is the initial database, and D_i is the database after sequential execution of the transactions numbered 1, 2, 3, ..., i, in this order. Assuming transactions are executed sequentially, if each transaction actually wrote a new version D_i of the entire database, this new version would possibly be inconsistent until the transaction completed. But if version D_{i-1} were still available, queries that began before the transaction numbered i had completed could still "see" a consistent database by accessing this older version. Under concurrency, this approach can be simulated by having each transaction write new versions only of the objects it modifies. This scheme results in the database consisting of a collection of objects of one or more versions each, where the version number of an object is the same as the transaction number of the transaction that wrote the object.

This multi-version object scheme will be used here. This was the motive for defining versions of objects in the earlier formal development of serializability. The details of providing queries with a consistent view of the database without CC support, and of garbage collecting old versions, are given in Chapter 6. Transactions are sequentially numbered at

successful read-phase completions; the reasons for this are discussed in Chapter 4.

One objection sometimes raised to multi-version object schemes is that a large transaction that modifies the entire database, for example for reorganization purposes, will create a new version of every object, thus doubling the needed storage. However, in single-version schemes, how is recovery supported in this case? That is, what if as a result of hardware failure, software errors, or human mistakes, the database is "destroyed" by this large transaction? This is a real possibility, and one would hope that even in single-version object based systems an earlier version of the entire database were saved somewhere in this eventuality. In fact, this suggests the following solution for multi-version object based systems: as the large transaction runs, transfer the earlier version of each modified object to tape or other tertiary memory, and reclaim the secondary memory space. Note that the necessary mechanism could already be available as an automatic archival subsystem, or as part of the recovery subsystem.

3.5. Local Memory Managers

The LMM will have several responsibilities: managing a local cache of data objects, supporting the write-phase, hiding the GMM and CC from the RcdM, and providing a simple interface between the CC and the GMM. In the case that a local disk is available, the LMM could possibly participate with the RcdM in some recovery protocols, but this will not be considered here.

3.5.1. Cache Management

In order to avoid unnecessary transfer of data objects, the LMM will maintain copies of some of the objects that have previously been read (by any local transaction or query). Every read request to the GMM includes the version number of a local copy, if such exists. No object transfer is necessary if the local copy is the "correct" version (as determined by the GMM -- see Chapter 6).

If a particular transaction or query has already read an object, and there is still a copy, then no GMM communication is necessary at all. Whether or not an object has previously been read can be determined by marking the local copy.

In the computer network application (Section 2.4.2), in which a local disk is available to the LMM, at each node in the network that part of the database that is most often used at that node will migrate to that node. In particular, one would expect at least the upper levels of the database access structure (directories and indexes) to be present at each node, resulting in far fewer network object transfers than if caching were not used.

3.5.2. Write-Phase Support

As discussed above, an overall design decision is to write new versions of objects to the shared database only if it can be determined that the transaction that generates the new versions will not be aborted, which leads to the read-phase / write-phase protocol: during

the course of the transaction as seen by the RcdM (i.e. between *Tbegin* and *Tend* -- see below), no writes to shared memory occur; instead, a local copy is modified. Then, after the RcdM has called *Tend*, if the transaction is successful, the LMM writes all new versions to shared memory. Thus, the LMM must maintain copies of all objects written, created, or deleted, in order to perform the write-phase. In the case that there is not enough local memory for a particular transaction, it is easy to extend the GMM design presented here so that the LMM can request extra shared memory for its private use. In a centralized system this latter technique would always be used since the "LMM" would not have any local memory to manage.

Note: a deleted object is treated here as a new version of an object, i.e. in a fashion identical to a written object, primarily for simplicity.

3.5.3. Hiding Versions and Concurrency from the RcdM

Record management problems can be complex, and the complexity could become unmanageable if the existence of multi-version objects, read-phases and write-phases, and concurrency control had to be dealt with at the same level. It is the responsibility of the LMM to hide all of this from the RcdM, thus completing with the CC, GMM, and RcvM the support of a virtual internal model. From the point of view of the RcdM, the database consists of a collection of objects, of one version each, to which it has exclusive access.

The LMM can hide all of this from the RcdM by mapping RcdM object accesses to local copies (retrieving a shared copy if necessary), by sending the necessary information to the GMM to complete or abort a transaction, and by sending the CC the necessary information to detect conflicts.

3.5.4. GMM / CC Interface

When a transaction successfully completes, the CC will return a *transaction number*, which is just the current value of the *transaction number counter* (see Chapter 4). It is the responsibility of the LMM to supply the GMM with this number for version number use during the write-phase.

3.6. Summary of Subsystems and Interfaces

The following is a summary of the functions and interfaces of the various subsystems.

3.6.1. RECORD MANAGER

Functions. This subsytem defines the conceptual model and its mapping to the data objects of the virtual internal level. This includes the functions of definition of record structures; mapping of records, relations, etc., onto objects; creation and maintenance of indexes; efficient insertion, deletion, update, and retrieval of records; etc.

Interface. Varies, depending on the conceptual model provided by the RcdM.

3.6.2. LOCAL MEMORY MANAGER

Functions. Cache management; buffering until the end of a transaction objects written, created, or deleted; doing this in such a fashion that the facts that there are multiple versions of objects and that objects are shared is invisible to the record manager; provide a small, more controlled interface between the RcdM, GMM, and CC.

Interface.

Qbegin - begin a query. Invokes *MQbegin*.

Qread - make an object addressable. May invoke *Mread*, may cache object in local memory.

Qrelease - make an object non-addressable (free local memory for object).

Qend - end a query. Invokes *MQend*.

Tbegin - begin a transaction. Invokes *Cbegin* and *MTbegin*.

Tread - make an object read addressable. May invoke *Cread* or *Mread*, may cache object in local memory.

Tcreate - create an object. Invokes *Mcreate*, allocates local memory.

Twrite - make an object read/write addressable. May invoke *Mread*, *Cwrite* or *Mnew*, allocates local memory if necessary.

Trelease - make an object non-addressable (may free local memory for object).

Tdelete - delete an object. Invokes *Mnew*, allocates local memory if necessary.

Tabort - abort a transaction. Invokes *Cabort* and *Mabort*.

Tend - complete read-phase of a transaction. Invokes *Cvalid*; then if successful *Mwrites*, writes new versions to shared memory; finally *MTend* and *Cend*; otherwise invokes *Mabort*.

Tname - generate a unique name. Invokes *Mname*.

3.6.3. CONCURRENCY CONTROL

Functions. Detect and resolve possible conflicts so as to guarantee serializability; transaction numbering. Conflicts are detected by keeping track, for each transaction, of sets of objects (IDs) read and written; conflicts are resolved by having transactions wait or aborting transactions.

Interface.

Cbegin - begin a transaction.

Cabort - abort a transaction.

Cread, *Cwrite* - request for access to an object. Check for conflicts -- return decision or have transaction wait.

Cvalid - indicates end of read-phase for transaction. Final conflict check -- return decision or have transaction wait.

Cend - indicates end of write-phase.

3.6.4. GLOBAL MEMORY MANAGER

Functions. Maintaining object ID, version => physical address mappings; supplying each query with a consistent "snapshot" of the database; creation of new objects; garbage collection of old and deleted objects; generation of unique names.

Interface.

MTbegin, *MQbegin* - begin a transaction, query.

MTend, *MQend* - end a transaction, query.

Mabort - abort a transaction.

Mread - read an object: find correct version and return version number and address in shared memory.

Mnew - allocate space for new version of an object (including "deleted version").

Mcreate - create a new object (allocate space and return ID).

Mwrite - write an object (including "deleted version"): return address of shared memory allocated in *Mnew* or *Mcreate*.

Mname - generate and return a unique name.

3.6.5. RECOVERY MANAGER

Functions. During normal operation: write each new version (say), new values of the write-phase completion counter (see Chapter 6), other information useful for recovery from failures on highly reliable, inexpensive, write-once media. During recovery, find old versions in such a fashion so as to allow recovery of a previous consistent state.

Note. It should often be possible for the GMM to recover from non-disk system failures -- in general, this GMM-only recovery will be possible if the failure did not cause garbage to be written on the disk in "sensitive areas". Garbage can often be detected by redundancy techniques, e.g. checksums. In the simplest case, the RcvM could be used periodically to backup the entire contents of system disks, with no transactions allowed during this period. For example, in the Cm* system described in Chapter 7, a very simple RcvM was written as part of the GMM that saved the GMM object ID, version => address mapping on request, and the database itself could be backed up if desired by file transfer to another machine. A somewhat more complex use would be to periodically backup "snapshots" of the database, considering this entire procedure as one very large query -- transactions would still be allowed in this case. It is also possible to do this in a much more dynamic way, and to allow data-independent recovery at transaction granularity, by having the RcvM save each new version of each object as it is produced, and by saving each new value either of the write-phase completion list or the write-phase completion counter (see Chapter 6). The cost of this latter approach is an open problem, but would perhaps prove useful given large inexpensive write-once memories, such as video disks. Just as there are many policies for the CC, there also seem to be many RcvM policies. In any case, the problem of recovery is an important one, but is beyond the scope of this thesis: the design and use of the recovery manager will not be dealt with in any more detail here.

4. A General Paradigm for Concurrency Controls

In this chapter a general design paradigm for concurrency controls will be presented. First, it is necessary to discuss in more detail the assumptions made regarding the fashion in which the CC controls transactions.

4.1. Controlling Transactions

As a transaction runs, on the first *Tread* for a given object, the LMM will send the CC a *Cread* message for the object, and wait for a reply. This message is interpreted as a request for read access to the object. The CC must at this point decide whether to grant the transaction read access immediately, in which case a positive reply is sent, to abort the transaction, in which case a negative reply is sent, or to postpone the decision, in which case a reply, perhaps positive, perhaps negative, will be sent at some later time.

Similarly, on the first *Twrite* for a given object, the LMM will send the CC a *Cwrite* message for the object, and wait for a reply. This *Cwrite* message is interpreted as a request for read/write access to the object. Again, the CC replies with its decision, or postpones its decision.

Although the CC can abort a transaction by replying negatively to a request, it may also be the case that the CC will decide to abort a transaction at a point where the transaction is not waiting for a message from the CC. For example, the CC may decide to abort one transaction based on a request from another transaction. If it is possible to interrupt the transaction, the abort may be handled in this fashion. Timing problems can be handled by requiring the LMM to send an acknowledgement message to the CC, and by the CC marking the transaction as aborted, ignoring all requests from that transaction until the acknowledgement is received. Alternatively, the CC can mark the transaction as aborted, and send a negative reply on the next request from the transaction (for simplicity, the CC algorithms of Appendix III use this technique). In any case, the transaction is said to be *aborted* whenever the CC either replies negatively, interrupts the transaction, or marks the transaction as aborted, whichever occurs first.

If a transaction already has read or read/write access to some object, a request for the same kind of access to the same object is handled by immediately returning a positive reply (unless the transaction is aborted).

When a transaction "ends" (ends from the point of view of the RcdM), the LMM will send the CC a *Cvalid* message, and wait for a reply. Prior to this message, a transaction that has not been aborted is said to be *active*. This message is in essence a request to the CC for final approval, or *validation*, of the transaction. As discussed in Chapter 3, an overall design decision is send the GMM new versions only if the transaction that generated the new versions can be guaranteed not to be aborted. Since the LMM will begin the write-phase at this point if a positive reply is sent, the decision of the CC is in this sense final. When and if

the CC does return a positive reply, the transaction is said to be *validated*. Included with the reply is a *transaction number*, which the LMM will use as a version number for all objects written in the write-phase, and which the GMM will eventually use in *WPCL* processing (see Chpater 6). Aborted transactions are not numbered. A transaction number is generated simply by returning the value of an integer *transaction number counter*, for brevity named *TNC*, and then incrementing the counter. Let *TNC* be initially one.

At any point in time there will be a *validated transaction history* consisting of all reads and writes of validated transactions with transaction numbers 1, 2, 3, ..., *TNC*-1. *The function of the CC is to control transactions so as to guarantee that the validated transaction history is always serializable*. By the conflict theorem (see Section 2.5), this means that the validated transaction history must be kept conflict-free. There are two methods that can be used for guaranteeing this. First, transactions can be aborted -- an aborted transaction will not be part of the validated transaction history. For example, let *a* be a transaction in the validated transaction history: a correct concurrency control will in the future abort any (not yet validated) transaction *b* that conflicts with *a* (even though *b* has not yet been assigned a transaction number, it is known that if it ever is numbered, the transaction number of *a* will be less than the transaction number of *b*, so it makes sense to speak of a conflict between *a* and *b*). The reason for this is that if *b* conflicts with *a*, one of the two must be aborted -- however, *a* has already been given final approval, so *b* must be aborted. Second, in an attempt to avoid aborting transactions, transactions can be made to wait for read or read/write access. The idea is to rearrange reads and writes by postponing some of them so that the validated transaction history is kept conflict-free.

Suppose now that a *Cvalid* message is received from some transaction (that is not aborted). This transaction does not conflict with any previously validated transaction (or it *would* be aborted). How can the CC handle this request? As far as consistency of the database is concerned, the transaction can be validated immediately. Another possibility is for the CC to postpone the decision. At first this might seem pointless: if it is possible to validate the transaction, why not do so immediately? The reason is that cases may arise in which a transaction, if validated, causes a conflict with an active transaction, which means that the active transaction must then be aborted. However, if the validation of the transaction were postponed, it might be possible to eventually validate both transactions. A transaction for which validation has been postponed is said to be *pending*.

Finally, after a validated transaction completes its write-phase, the LMM will send the CC a *Cend* message. Such a transaction is then said to be *completed*. No decision is necessary at this point for the transaction that sent the message, since the final decision was made earlier when the transaction was validated -- rather, the purpose of this message is to inform the CC that the new versions written by the transaction may now be read by other transactions. In summary, the state transitions of a transaction are shown in Figure 4.1.

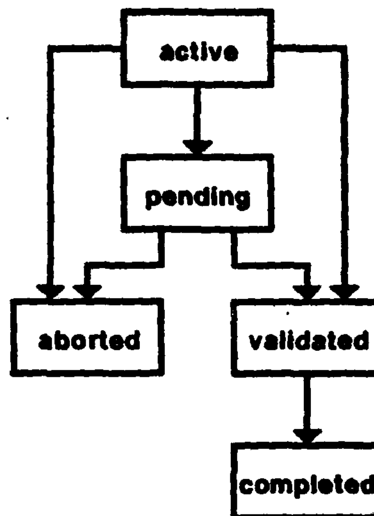


Figure 4.1. Transaction State Transitions

4.2. Correctness of the Concurrency Control

In Section 2.5, conflicts were defined in terms of *transaction histories*, and transaction histories were defined as a sequence of quadruples of the form <R, *i*, *j*, *o*> or <W, *i*, *j*, *o*>, where *i* is a transaction number, *j* is a sequence number, and *o* is an object ID. However, this formalism is unsuitable for the present purposes, since although the CC has information about the reads and writes of transactions, it has no information about their exact interleaving. For example, with respect to writes, the CC "knows" only that for any transaction, all writes take place after validation and before completion. Also, the transaction number of a transaction is not known until validation. In order to define a correct CC, a formalism describing the actions of the CC is necessary. With this in mind, a *CC history* is defined as a sequence of tuples of the following forms, where *a* is a transaction ID and *p* is an object ID:

<R, *a*, *p*>, meaning *a* is granted read access to *p*;
<W, *a*, *p*>, meaning *a* is granted write access to *p*;
<V, *a*>, meaning *a* is validated;
<A, *a*>, meaning *a* is aborted;
<C, *a*>, meaning the completion message from *a* is received.

The following predicates will be useful.

R(*a*, *p*): <R, *a*, *p*> appears in the CC history;
W(*a*, *p*): <W, *a*, *p*> appears in the CC history;

V(*a*): $\langle V, a\rangle$ appears in the CC history.

The CC history of an executing CC is maintained by appending the appropiate tuple as each of the above actions is taken by the CC. This is straightforward except for $\langle W, a, p\rangle$: when a transaction requests read/write access via *Cwrite* and a positive reply is returned, if *R*(*a*, *p*), then only $\langle W, a, p\rangle$ is appended; otherwise, $\langle R, a, p\rangle$ and $\langle W, a, p\rangle$ are both appended. With this notation, a correct CC history can be defined as follows.

Correctness. The CC history is correct if for all pairs of transaction IDs *a* and *b* in the history such that *R*(*a*, *p*) and *W*(*b*, *p*), one of the following cases holds:

C1. Not *V*(*a*) or not *V*(*b*);

C2. *V*(*a*) and *V*(*b*), and $\langle V, a\rangle$ precedes $\langle V, b\rangle$;

C3. *V*(*a*) and *V*(*b*), $\langle V, b\rangle$ precedes $\langle V, a\rangle$, and $\langle C, b\rangle$ precedes $\langle R, a, p\rangle$.

A correct CC is a CC for which the CC history is kept always correct. This correctness criterion is a straightforward application of the conflict theorem. In case C1, there is currently no conflict between *a* and *b*, since at least one of them has no writes to the current point. Next, if the transaction number of *a* is less than the transaction number of *b*, no conflict is possible between *a* and *b* with respect to a read of *a* and a write of *b* if all reads of *a* take place before any writes of *b*, as is the case in C2. Finally, if the transaction number of *b* is less than that of *a*, case C3 requires that *a* not be granted read access to the object in question until it can be guaranteed that *b* has written the new version of the object.

The correctness criterion is violated only in the case that for some transactions *a* and *b*, *R*(*a*, *p*), *W*(*b*, *p*), *V*(*a*), *V*(*b*), $\langle V, b\rangle$ precedes $\langle V, a\rangle$, and $\langle R, a, p\rangle$ precedes $\langle C, b\rangle$ (see Figure 4.2). In such a case a conflict is possible in the validated transaction history: transaction *a* may have read the most recent version of the object with ID *p* before the new version created by transaction *b* had been transferred. Thus, the validated transaction history is serializable if and only if it is conflict-free, and it can be guaranteed to be conflict-free if and only if the CC history satisfies the above correctness criterion.

4.3. The Paradigm

The correctness criterion defines correct CC histories in a static way: given a CC history, it can be determined if the CC history is correct. The problem now is to design the CC so that the CC history is kept always correct.

The independence of the CC from other modules and from applications means that the CC can make no predictions about the future accesses of transactions. In fact, as explained in Section 1.2, in most cases such predictions are impossible. Therefore, conditions possibly leading to a violation of the correctness criterion must be detected dynamically based on incoming access requests. In the absence of such conditions, access requests will always be granted. That is, the CC *will abort transactions or have them wait based only on current conditions that could possibly lead to violations of the correctness criterion.*

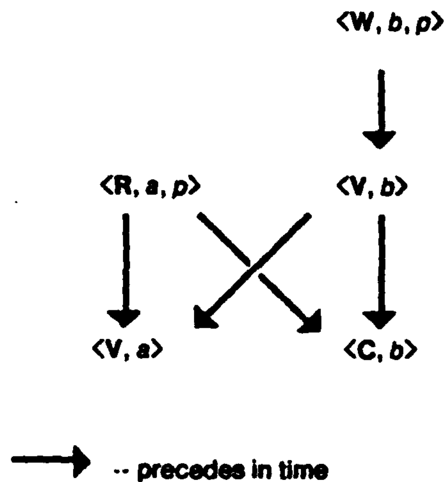


Figure 4.2. Only Violation of Correctness Criterion

The only condition that could possibly lead to a violation of the correctness criterion is that $R(a, p)$ and $W(b, p)$ for some transactions a and b. This condition is called a *possible conflict*, and can be detected by maintaining for each object ID p a *read set*, the set of transactions a for which $R(a, p)$, and a *write set*, the set of transactions a for which $W(a, p)$. Empty read or write sets need not be maintained; also, these sets need not be maintained for aborted transactions (due to C1). It will also be seen that they need not be maintained for completed transactions (this is due to the fact that transactions are numbered in validation order -- see Section 4.5 below).

The paradigm will be described by listing various options for handling access and validation requests. When $R(a, p)$ and $W(b, p)$ is detected for some transactions a and b, both active or pending, it is assumed that the CC records this fact for later reference. The alternative is to possibly (depending on the options selected) later check various read and write sets for intersection, which may become excessively time-consuming for a large number of transactions or for large read and write sets. When $R(a, p)$ and $W(b, p)$, a and b active or pending, this relation between a and b is written as $a \rightarrow b$.

The meaning of the relation $a \rightarrow b$ is that in order to validate both a and b, a must be validated before b (this is from C2 -- note that C3 does not apply since b is not completed). Depending on the options selected, it may arise that $a \rightarrow b$ and $b \rightarrow a$, in which case only one of the two transactions can be validated. Similarly, if $a \rightarrow b$, $b \rightarrow c$, and $c \rightarrow a$, only two of these three transactions can be validated. In an attempt to validate both a and b when $a \rightarrow b$ but not $b \rightarrow a$, and also in an attempt to avoid cyclic $\rightarrow$ conditions such as $a \rightarrow b$ and $b \rightarrow a$, an access or validation request may be postponed until one or more events have occurred. This is called *scheduling*. When the event or events have occurred, the access or validation request is re-analyzed as if newly arrived. It is assumed that *the*

goal of scheduling is to avoid aborting transactions. Therefore, from the correctness criterion, there are two types of events that can be used in scheduling: the validation of a transaction (C2), and the completion of a transaction (C3). The following notation will be used for scheduling.

$a \Rightarrow_V b$: a positive reply to the current access or validation request of transaction b will not be sent until transaction a is aborted or validated (transaction b may be aborted before then);

$a \Rightarrow_C b$: a positive reply to the current access request of transaction b will not be sent until transaction a is aborted or completed (transaction b may be aborted before then).

When a transaction requests access to an object p, it may be desirable for the CC to base its decision on the processing of this request not only on those transactions a for which $R(a, p)$ or $W(a, p)$, but also on those transactions for which access to p has been postponed (e.g., in order to queue requests). If a read or write request from transaction a for object p has been postponed, this is written as $RP(a, p)$ or $WP(a, p)$, respectively.

The paradigm follows. The generality of the paradigm is considered in Section 4.5.

Read request. Transaction a requests read access to p, no current read access.

R1 (aborted and completed transactions). Any aborted transaction can be ignored by C1, and any completed transaction can be ignored by C1 (if a is never validated) or C3 (if a is later validated).

R2 (postpone). For each active, pending, or validated transaction b with $W(b, p)$ or $WP(b, p)$ ($WP(b, p)$ is possible only for b active), do one of the following.

R2.1 (skip). Skip b in this step.

R2.2 (abort). Abort b (applicable only if b is not validated).

R2.3 ($\Rightarrow_C$). Schedule $b \Rightarrow_C a$ (thereby avoiding aborting a, aborting b, or $a \rightarrow b$ -- see R3 and R4 below).

If R2.3 was selected for any b, the access request has been postponed, and the CC history remains correct by C1. Later, the access request will be re-processed; for now, terminate.

R3 (abort). It is assumed now that the access request will not be postponed. If there is any validated transaction b with $W(b, p)$, neither C2 nor C3 can ever be true for a and b if the access request is now granted. Furthermore, although it does not violate the correctness criterion, if a is granted read access now, it may read inconsistent data, since the new version of p written by b may or may not be read by a. Therefore, if there is

any validated transaction b with $W(b, p)$, since a can never be validated, a should be aborted: abort a and terminate Otherwise, if there are any active or pending transactions with $W(b, p)$ or $WP(b, p)$, optionally abort a, and terminate.

R4 (grant). For each active or pending transaction b with $W(b, p)$, record $a \rightarrow b$. Then, grant a read access, and terminate.

Write request. Transaction a requests write access to p, no current write access.

W1 (aborted and completed transactions). Any aborted transaction can be ignored by C1, and any validated or completed transaction can be ignored by C1 (if a is never validated) or C2 (if a is later validated).

W2 (postpone). For each active or pending transaction b with $R(b, p)$ or $RP(b, p)$ ($RP(b, p)$ is possible only for b active), do one of the following.

W2.1 (skip). Skip b in this step.

W2.2 (abort). Abort b.

W2.3 ($\Rightarrow_V$). Schedule $b \Rightarrow_V a$ -- in this option, a waits for the validation of b at the current point, rather than possibly waiting at the validation point in V1.3 below.

If W2.3 was selected for any b, the access request has been postponed, and the CC history remains correct by C1. Later, the access request will be re-processed; for now, terminate.

W3 (abort). If there are any active or pending transactions b for which $R(b, p)$ or $RP(b, p)$, optionally abort a, and terminate.

W4 (grant). For each active or pending transaction b with $R(b, p)$, record $b \rightarrow a$. Then, grant a read access, and terminate.

Note that the write paradigm can be obtained from the read paradigm by interchanging R and W, $\Rightarrow_C$ and $\Rightarrow_V$, by reversing $\rightarrow$, by replacing "completed" with "validated or completed" in R1, and by removing the now inapplicable statement that a be aborted if there are any validated conflicting transactions in R3.

Read/write request. Transaction a requests read/write access to p, no current access.

In the processing of this request the set of possibly conflicting transactions are all those transactions b with $W(b, p)$, $WP(b, p)$, $R(b, p)$, or $RP(b, p)$. Again, any of these may optionally be aborted; a may optionally be postponed; a may optionally be aborted; or a may be granted read/write access. In the case that a is postponed, $\Rightarrow_C$ scheduling must be used for any transaction b with $W(b, p)$ or $WP(b, p)$, and $\Rightarrow_V$ scheduling for any

transaction *b* with $R(b, p)$ or $RP(b, p)$ (there is no harm in using both types of scheduling with respect to the same transaction -- in such a case $\Rightarrow_V$ is simply superfluous). As in a request for read access, if *a* is not postponed and there is a validated transaction *b* with $W(b, p)$, *a* should be aborted. Finally, if *a* is granted read/write access, $a \rightarrow b$ must be recorded for each transaction *b* with $W(b, p)$, and $b \rightarrow a$ must be recorded for each transaction *b* with $R(b, p)$ (both might be recorded with respect to the same transaction).

Validation request. Transaction *a* requests validation.

V1 (postpone). For each active or pending transaction *b* with $b \rightarrow a$, do one of the following.

V1.1 (skip). Skip *b* in this step.

V1.2 (abort). Abort *b*.

V1.3 ($\Rightarrow_V$). Schedule $b \Rightarrow_V a$ (thereby possibly avoiding aborting *a* or *b*).

If V1.3 was selected for any *b*, the validation request has been postponed, *a* is now pending, and the CC history remains correct by C1. Later, the validation request will be re-processed; for now, terminate.

V2 (abort). If there are any active or pending transactions *b* with $b \rightarrow a$, optionally abort *a*, and terminate.

V3 (validate). For each active or pending transaction *b* with $b \rightarrow a$, abort *b*. Then validate *a*, and terminate.

This completes the description of the CC paradigm. In the next section the question of how best to use the paradigm is considered.

4.4. Policies

The correctness criterion gives only those necessary and sufficient condtions for the serializability of the validated transaction history -- it does not, for example, rule out the case in which a transaction reads inconsistent data (see R3 above), although it does rule out the case in which such a transaction is ever validated. Nor does it rule out the case in which a transaction is never validated or aborted. In designing the paradigm for processing access requests above, the only criteria used were the correctness criterion, the independence of the CC, and the assumption that the purpose of scheduling is to avoid aborting transactions. No other criteria, such as "fairness", or the guaranteed eventual successful completion of transactions, were used. But in practice, first, it is necessary to choose one of the options provided by the paradigm; second, additional correctness properties such as guaranteed eventual successful completion may be important; and finally, it is desirable to choose options so as to optimize performance if possible. These problems will be dealt with here in

a *policy*, which is that part of the CC design that chooses the options as provided by the above paradigm.

Some additional correctness properties that may be important have to do with the two mechanisms that are used to control transactions: scheduling and aborting. In the case of scheduling, two well-known problems are deadlock (in which the union of the $\Rightarrow_V$ and $\Rightarrow_C$ relations is not a partial-ordering) and starvation (in which a transaction is repeatedly scheduled so that it waits potentially forever). Assuming aborted transactions are automatically restarted until they complete successfully (this may or may not be the case in an application), two analagous problems for aborting are cylic restart (in which a finite set of transactions repeatedly cause each other to be aborted so that none of the transactions is ever validated) and infinite restart (in which a transaction is repeatedly aborted due to possible conflict with a potentially infinite set of transactions). Note: there does not appear to be widespread agreement in the literature in terminology for these latter two problems. Whether or not any of these conditions are problems depends both on the policy and the application: if the policy never chooses a scheduling option, clearly deadlock and starvation are not problems. On the other hand, perhaps deadlock and starvation are possible, but in the application it is acceptable either to assume that (in the case that transactions are interactively generated) impatient users will abort their transactions, or to assume that the LMM will abort transactions on timeouts. This latter mechanism, which would often be used in a distributed environment, could perhaps make deadlock detection unnecessary. If deadlock detection is necessary, then there may be a policy question of how often to check for possible deadlock (see [Gray 78]).

All known solutions to the problems of cyclic and infinite restart involve some kind of priority scheme. The general idea is, first, to design the policy so that transactions with sufficiently high priority will never be aborted, and second, to give a transaction increasing priority as it becomes older or is repeatedly aborted. Of course, priority schemes can be based on performance criteria as well. Some of the many possible priority schemes are: (1) give increasing priority to transactions as they are aborted; (2) give increasing priority to transactions as their original starting time (that is, a starting time not changed by repeating a transaction due to a failure) becomes older -- these are the timestamp-based approaches (see the following section); (3) give priority to transactions that are generated interactively; (4) give priority to transactions that are part of some real-time process; (5) give priority to transactions that are for some reason expensive to retry (e.g., "big" transactions). Various priority schemes can also be combined.

The use of any priority scheme would involve extensions to the CC interface as presented in Appendix II, e.g., inclusion of a unique transaction ID, a starting time, or a transaction class as a parameter of *Cbegin*. Here, various *basic policies* that do not use any priority scheme will be studied. However, the extensions necessary to use a priority-based policy are simple and straightforward. All that is necessary is to include any information about transactions as

required by the policy as additional parameters to *Cbegin.*

Policies can be defined statically at design-time, or dynamically as *policy modules.* In the case that policies are defined statically various optimizations can be made -- for example, in some policies deadlock is impossible, and so deadlock detection may be omitted. On the other hand, the policy module approach may offer efficiency advantages in the case that the policy module is designed so as to be able to change policies at run-time. In the case that several policies are of interest, but their differences cannot be predicted, it becomes easy to experiment with these policies by using a policy module that provides all such policies. An example is the policy module used in the Cm* system (see Chapter 7), which provided 330 distinct basic policies (see Chapter 5).

4.5. Generality of the Paradigm

A great deal of previous research in concurrency control design can be viewed as policy design. It seems that some confusion has often resulted due to the lack of a clear separation of fundamental correctnes criteria (e.g., the correctness criterion above), additional correctness criteria (e.g., guaranteed eventual successful completion), and policy criteria (e.g., giving priority to transactions that are expensive to retry). Although the design problems may be listed independently, they may not be handled independently in the design itself. This has had the effect of making essentially similar concurrency controls seem superficially quite different. In fact, in a recent extensive survey of proposed concurrency controls [Bernstein and Goodman 81], it is concluded that "all practical concurrency control methods can be analyzed as combinations and variations of two basic synchronization techniques: two-phase locking and timestamp ordering."

Two-phase locking (due to [Eswaran et al 76]) is obtained from the above paradigm by selecting options R2.3 and W2.3 (or alternatively, V1.3 in place of W2.3) whenever possible (i.e., in the absence of deadlock). There are many variations of timestamp ordering concurrency controls, and its true nature is often obscured by combining the technique with two-phase locking types of policies. In what might be called "pure" timestamp ordering (no two-phase locking component), options R2.2 or R3 and W2.2 or W3 are always selected, with R2.2 or W2.2 selected if (referring to the paradigm) *a* has an earlier original starting time than *b*, and with R3 or W3 selected otherwise. Although the above claim of [Bernstein and Goodman 80] cannot be agreed with here (timestamp ordering seems more properly a policy priority scheme, and concurrency controls involving $a \rightarrow b$ options R4.1 and W4.1 are neglected), they do clearly point out that almost all proposed concurrency controls are essentially similar due to the fact that a common problem -- guaranteeing serializability -- is being solved.

A major advantage of the CC design here is the clear separation of basic correctness and policy. Since policy problems are quite complex in themselves, and subject to various solutions under different applications and environments, they should be handled separately

from the basic correctness problem of guaranteeing serializability. As an example of a complex policy problem, two-phase locking assumes that aborting transactions is expensive, and tries to avoid this whenever possible, but at the expense of scheduling. Yet, given the LMM description of Section 3.5, a transaction that will later be aborted is still possibly doing useful work in caching copies of objects. Such a transaction, when restarted, could quickly run to completion if very few object transfers were then necessary. So whether aborting or scheduling is more expensive is not obvious, and is most likely highly application dependent: a truly general concurrency control should provide for all kinds of policies, including those that select $a \rightarrow b$ options, such as [Kung and Robinson 81] and [Stearns and Rosenkrantz 81]. The advantages of separating basic correctness and policy, in system correctness and maintainability, have been discussed in a more general context in [Everhart 79].

The paradigm above is completely general, given the following conditions.

1. Access requests cannot be predicted in advance.
2. The goal of scheduling is to postpone transactions for as short a time as possible in order to attempt to prevent aborts.
3. A transaction, once validated, cannot be aborted, and the validated transaction history must be kept serializable *in validation order*.

Condition (1) is common to all application-independent concurrency controls. Condition (2) excludes highly heuristic scheduling techniques such as "wait 10 seconds and retry." Such techniques may be valuable in distributed systems, but are beyond the scope of this work. Condition (3) seems to be common to all practical concurrency controls. The notion of validating a transaction, or giving it final approval, is also often called *committing* a transaction (e.g., see [Gray 78] -- but also see the note below). This seems to be a necessary simplification to make the problem of concurrency control manageable. In fact, it is hard to imagine a system where one never knew for sure whether a transaction was completed -- the notion of a validation or commit point seems inescapable.

On the other hand, maintaining the validated transaction history serializable in validation order is an efficiency constraint. Note that serializability, as defined here, depends only on the ordering of transaction numbers, and not on the numbers themselves. Thus, if transaction *a* requests validation, and the current validated transaction history consists of the transactions numbered 1, 2, 3, ..., *n*, then *a* could conceivably be validated under transaction number 1.5, 2.5, 3.5, etc. However, this would require maintaining read and write sets for completed transactions, would probably prove excessively time-consuming, and could require (depending on the design) query validation as well. Therefore, such schemes are rejected here. It seems that in all existing or proposed concurrency controls in which transactions have a commit or validation step, the transaction history is maintained serializable in validation or commit order.

Note: the separate notions of *concurrency control* commit (or validation) and *recovery*

manager commit apparently have previously been confused -- in fact, in many systems, they are (accidentally) the same, probably due to the lack of separation of CC design from RcvM design. The CC commit point is described above. A RcvM commit point is quite different: assuming a memory hierarchy, a number of RcvM commit points at different levels can be defined as points at which, if memory at the given level of the hierarchy does not fail, the writes of transactions that have completed write-phases (say) to that level can be recovered. Here, it was earlier decided to send the GMM new versions only after (CC) validation, for efficiency and simplicity reasons, which results in CC commit points preceding RcvM commit points.

Finally, the paradigm by no means solves the general concurrency control problem, since the scope of concurrency control policy design is so large. For example, much of the research in distributed concurrency control design can be seen as the following problem: design the policy so that a decision regarding an access request for an object p can be made using only information that is present at the node where p is stored. Another large area of research in policy design, now made possible with the above general paradigm, is that of designing a policy module that selects the optimal type of concurrency control based on, say, performance monitoring or usage statistics. This is made possible by the above paradigm since the policy module need not be restricted to any one type of policy; whatever decisions are made by the policy module, the validated transaction history still remains serializable. For example, a policy module that selected options at random would still be a valid policy module, in terms of guaranteeing serializability.

4.6. Partitioning the Concurrency Control

In practice, it is often the case that possible conflicts between transactions arise rarely (see [Kung and Robinson 81] for a discussion of systems where this is likely to hold). In these cases, most of the work done by the CC is simply checking for each access request for some object that the read or write set for that object is empty, and after the request has been granted, updating the read or write set for that object. In the case that the CC forms a system bottleneck, this suggests the following scheme for introducing parallelism in the CC: partition the set of shared data objects in some fashion (for example, by mapping each object with ID *ID* into partition number *ID* MOD *n*, where there are *n* partitions), and use a separate process to manage the read and write sets for each partition. In a computer network application, if objects are partitioned based on the node where the transaction that created the object originated, the result is a type of primary site approach.

The remaining information managed by the concurrency control is transaction information: the status of each running transaction, the set of objects accessed for each transaction (these sets are used to update the proper read and write sets when the transaction completes or aborts), and all $\rightarrow$, $\Rightarrow_V$, and $\Rightarrow_C$ relations between these transactions. In the simplest case, this information can be managed by a single additional process, with a message interface between this process and the processes managing read and write sets.

Alternatively, several processes could be used to manage transaction information (for example, the same processes that manage read and write sets), and all transaction information could be stored in shared memory (in this case, access to transaction information would have to be synchronized, but designing the synchronization mechanism does not present any fundamentally new problems).

In the case of computer networks, a large number of schemes have been proposed in which transaction information is distributed over the network (typically, if a conflict develops between two transactions due to an access request for some object, this information is stored at the node where the object is stored). The use of a central transaction number counter can be avoided by determining the order in which transactions are possibly validated beforehand (by assigning timestamps at the beginning of transactions, for example -- if the timestamp of *a* is less than that of *b*, in order to validate both *a* and *b*, *a* must be validated before *b*). If multi-version objects are used, the same ordering must be realized in version numbers. The main problems with these approaches are that it is more difficult to be sure that the system is correct (the pre-determined ordering is often used in an attempt to cause identical decisions to be made at different nodes without communication, and so the correctness argument depends on a priority scheme), and often there are no resulting performance advantages (see [Garcia-Molina 79]). However, if the network is geographically distributed (with nodes in different cities, for example) *and* there is locality of reference (transactions almost always access objects stored at the node where the transaction originates), there are clear advantages to these approaches. In such cases, if the possible validation ordering is determined beforehand, the paradigm can be made to apply by adding the following restriction: $a \rightarrow b$, $a \Rightarrow_V b$, or $a \Rightarrow_C b$ can be chosen as an option *only* if *a* precedes *b* in the predetermined ordering.

5. Basic Policies

In this chapter a set of basic policies is defined. It will be seen that the design space for concurrency controls is much larger than has previously been recognized -- even in the simplest case in which all transactions are handled uniformly there is a large number of distinct policies. This set of policies should be considered only as a foundation for policy development, since in practice there are many valuable extensions. Two extensions to policies using scheduling, deadlock detection and queuing of requests, are described. Other extensions involve the introduction of priority schemes, as described in Chapter 4. As examples of how one might begin to develop a policy, possible philosophies behind several policies are discussed.

5.1. Definition of the Basic Policies

A basic policy is defined here as a policy in which all transactions are handled uniformly without the use of priorities. For each type of request, the paradigm of the previous chapter defines a set of transactions that may conflict with the transaction issuing the request: for a read request for p, all transactions b with $W(b, p)$ or $WP(b, p)$; for a write request for p, all transactions b with $R(b, p)$ or $RP(b, p)$; for a read/write request for p, all transactions b with $W(b, p)$, $WP(b, p)$, $R(b, p)$ or $RP(b, p)$; and for a validation request from a, all transactions b with $b \rightarrow a$. In each case, these transactions will be called here simply the conflicting transactions. Given a request, if the set of conflicting transactions is non-empty, the policy must be consulted. For a basic policy, *all conflicting transactions* are treated uniformly. A number of basic policies can be obtained by choosing one of the following options for each type of request (the "kill/die" terminology is taken from [Rosenkrantz et al 78]).

wait - have the requesting transaction wait on all conflicting transactions, using $\Rightarrow_V$ or $\Rightarrow_C$ scheduling as given by the paradigm.

kill - abort all conflicting transactions.

die - abort the requesting transaction.

grant - grant the access request (not an option for a validation request).

For a read/write request from a for p, if the *wait* or *kill* option is used, one might want to handle transactions b with $R(b, p)$ or $RP(b, p)$ but not $W(b, p)$ nor $WP(b, p)$ separately from transactions b with $W(b, p)$ or $WP(b, p)$. For example, transactions b with $R(b, p)$ or $RP(b, p)$ but not $W(b, p)$ nor $WP(b, p)$ could be ignored at this point, and possibly be waited on at the validation point if the possible conflict does not turn into a real conflict, while transactions with $W(b, p)$ or $WP(b, p)$ might be waited on at the current point. To handle these schemes, three sub-options may be added to the *wait* or *kill* options in the case of a read/write request, as follows.

read - wait on or abort only those transactions that have issued a conflicting read request but have not issued a conflicting write request.

write - wait on or abort only those transactions that have issued a conflicting write request.

all - wait on or abort all conflicting transactions.

Using these sub-options, 4x4x(2x3+2)x3 = 384 basic policies have been defined. However, there is some redundancy: if the *grant* option is never used, and the *all* sub-option is used for a read/write request, there will never be any conflicting transactions at the validation point, and so the validation option is unused. Eliminating this redundancy, 384 - 3x3x3x2 = 330 distinct basic policies have been defined. In the Cm* system described in Chapter 7, a policy module was used that provided all of these policies.

One might also consider policies in which active and pending or postponed and non-postponed transactions are differentiated to be basic policies, although these could also be considered priority schemes. In any case, sub-options to differentiate among classes of transactions can be added to the above scheme for basic policies, futher increasing the number of policies.

5.2. Deadlock Detection

Policies that use wait options may avoid deadlock by the use of a priority scheme, at the expense of increased probability of aborts (e.g., see [Rosenkrantz et al 78]). For the basic policies defined above, though, deadlock is possible. In this section the deadlock detection scheme used in the Cm* system will be described. Some alternatives to this scheme are to "not worry" about deadlock (relying on timeouts to abort transactions, for example), or to periodically check the wait relation for cycles (see [Gray 78]).

The scheme used in the Cm* system was to schedule $b \Rightarrow_V a$ or $b \Rightarrow_C a$ only if it was not the case that $a \Rightarrow^* b$, where $\Rightarrow^*$ is the transitive closure of the union of the $\Rightarrow_V$ and $\Rightarrow_C$ relations. If this could not be done, the requesting transaction was aborted. In this way the union of the $\Rightarrow_V$ and $\Rightarrow_C$ relations was maintained as a partial ordering. In order to determine if $a \Rightarrow^* b$, the following simple recursive procedure was used.

1. If $a \Rightarrow_V b$ or $a \Rightarrow_C b$, then $a \Rightarrow^* b$ -- return true.
2. For each transaction c such that $a \Rightarrow_V c$ or $a \Rightarrow_C c$: If $c \Rightarrow^* b$, then $a \Rightarrow^* b$ -- return true.
3. Otherwise, it is not the case that $a \Rightarrow^* b$ -- return false.

At this point a modification that may be made to the basic policy *wait* option above can be described. Consider the following example: a requests write access to p, and the request is postponed so that $WP(a, p)$; next, b requests read access to p, and $a \Rightarrow_C b$ is scheduled. Later, when the write request from a is re-processed, b will be a conflicting transaction since $RP(b, p)$, but scheduling $b \Rightarrow_V a$ would lead to deadlock, and so a is aborted. This does not seem to make sense in terms of a policy: if a really should be aborted due to the access request from b, why not abort a at the time the access request is received? For this reason,

in the Cm* policy module, when processing a request from a for p, all transactions b such that $a \Rightarrow_V b$ or $a \Rightarrow_C b$ and $RP(b, p)$ were removed from the set of conflicting transactions in the case that a wait option was selected. It should be clear by now that any modification such as this does not affect fundamental correctness -- this is one of the main strengths of the paradigm. In this case, the result is a queueing structure on objects in which, for each object, a number of readers can be waiting for a writer which can in turn be waiting on a number of readers, etc., on a first-come first-served basis. Of course, other schemes could be used. For example, a type of reader-priority scheme results if, on a read request from a for p, transactions b with $WP(b, p)$ are removed from the set of conflicting transactions before applying the *wait* option.

5.3. Some "Interesting" Policies

The two-phase locking policy is obtained by selecting the *wait* option for read and write requests, and the *wait all* option for read/write requests -- the validation option is redundant. As noted earlier, the goal of this policy is to avoid aborts if at all possible.

An optimistic policy is obtained by selecting the *grant* options for read, write, and read/write requests, and the *kill* option for validation requests. In this policy, transactions never wait, and conflicting transactions "race" to the finish: given a set of conflicting transactions, the transaction that first requests validation completes, and the conflicting transactions are aborted.

There are a variety of policies that lie between the two-phase locking and optimistic policies. In these policies, combinations of *wait*, *grant*, *die*, and *kill* options are used. For example, two-phase locking could be modified so as to grant a read request from a for p even if for some b, $W(b, p)$ -- although this introduces $a \rightarrow b$, a is allowed to proceed immediately, and it may still be possible to validate both transactions, having b wait on the validation of a when b requests validation if this case arises (and if this does not cause deadlock). The options for this policy would be: read - *grant*, write - *wait*, read/write - *wait all*, validation - *wait*.

Similarly, the optimistic policy could be modified so that the *wait* option is selected for a validation request from a with respect to all transactions $b \rightarrow a$ -- the philosophy behind this policy might be to retain the "never-wait" property of the optimistic policy for the read-phases of transaction, but to wait if necessary at the validation point in order to determine if possible conflicts turn into true conflicts. The options for this policy are: read, write, read/write - *grant*, validation - *wait*.

Finally, policies that select only *kill* or *die* options may seem uninteresting, but such policies could conceivably prove useful in some applications due to their extreme simplicity: for these policies, the $\Rightarrow_V$, $\Rightarrow_C$ and $\rightarrow$ relations are unused, and so need not be maintained.

6. Global Memory Managers

In this chapter the transaction support, query support, and garbage collection functions of GMMs will be described.

6.1. Memory Managment

The GMM must allocate and de-allocate storage space for objects, maintain the mapping between the virtual description (ID and version number) of an object and its physical address(es), and find certain versions of an object given its ID. Since none of these problems seem significantly new (in fact, several existing multi-version file systems solve these problems), discussion of such a multi-version object system will be omitted here.

6.2. Transaction Support

During the read-phase of a transaction, upon receiving *Mread*, the GMM must find the most recent version of the object requested. If the version number is different than that of the local copy (if any), the LMM will then read the new version of the object from shared memory. That the transaction "sees" a consistent database must be ensured by the CC. The GMM must also attempt to claim space in shared memory for new objects and new versions of objects, as requested.

During the write-phase, the GMM updates the mapping from virtual descriptions to physical addresses of each new version or new object as it is written. Then, upon receiving *MTend*, the GMM will update a *write-phase completion list* (*WPCL*), and possibly update a *write-phase completion counter* (*WPCC*). The *WPCC* is defined as the largest transaction number such that the corresponding transaction and all lesser-numbered transactions have completed their write-phases; the *WPCL* is defined as the list of transaction numbers greater than or equal to the *WPCC* of all such transactions that have completed their write-phases. This should be made clear by the following example.

> Assume that all transactions numbered 1093 and less have completed their write-phases, and that the transactions numbered 1095, 1096, and 1098 have also completed their write-phases. Then the *WPCC* is currently 1093, and the *WPCL* is:
>
> 1093, 1095, 1096, 1098.
>
> Continuing the example, upon receiving *MTend* for the transaction numbered 1100, the *WPCL* becomes:
>
> 1093, 1095, 1096, 1098, 1100,
>
> and the *WPCC* remains unchanged. However, upon receiving *MTend* for the transaction numbered 1094, the *WPCC* becomes 1096, and the *WPCL* becomes:
>
> 1096, 1098, 1100.

The *WPCC* and *WPCL* are of use in query support and garbage collection, as discussed below, and in recovery, as mentioned in Chapter 3.

6.3. Query Support

Suppose, as a query begins, the *WPCL* is:

1096, 1098, 1100.

At this point, a consistent version of the database can be observed, *without* any concurrency control, by accessing for each object ID the greatest version of the object that is less than or equal to 1096. This is the most recent version of the *entire* database that can be guaranteed to be consistent since, given an object ID, whether or not there may later be a version 1097, 1099, or a version greater than 1100 of this object, cannot now be determined.

By associating the current value of the *WPCC* with each query as its *query number* (*QN*) upon *Qbegin*, and thereafter sending that query the greatest version of each object requested that is less than or equal to its *QN*, queries will always observe a consistent database, without any CC support.

A possible problem with this scheme can be illustrated by the following example.

> A user executes a transaction, the transaction is successful and is numbered 1098, and when its write-phase completes the *WPCL* becomes:
>
> 1096, 1098, 1100,
>
> with a *WPCC* of 1096. But now, if the same user executes a query before the write-phase of the transaction numbered 1097 completes, the effect of the user's previous transaction (numbered 1098) will not be visible!

This problem arises only since write-phases are allowed to take place asynchronously, which is highly desirable for efficiency in the kinds of multiprocessor/network applications of concern here. If the above example represents a true problem, one solution is to restrict write-phases to be sequential in transaction-number order, which may be acceptable in a centralized system.

In de-centralized systems, though, other alternatives are more attractive. A scheme involving asynchronous notification of application programs of the occurrence of certain events (such as *WPCC* $\geq$ 1098) is feasible. Another solution is to allow queries to "pick" their own *QN* -- in the example above, the query could pick the transaction number of the completed transaction, 1098, as its *QN*. Then, the GMM could be designed so as to reply to *Qbegin*, but postpone the reply until the *WPCC* became greater than or equal to the *QN* of a given query, and the LMM could be designed so as to wait for such a reply. However, in order for garbage collection (see below) to be correct, queries must not be allowed to pick *QN*s less than the *WPCC*.

Finally, there is another alternative, in which the GMM associates a copy of the current

WPCL with each query as the query begins. In the example above, the list 1096, 1098, 1100 could be associated with the query. In this alternative the GMM finds for each read request the greatest version of the object that either (1) is less than the minimum transaction number in the associated *WPCL* copy, or (2) appears in the associated *WPCL* copy. This provides a consistent view of the database since, referring to the example above, the CC has guaranteed that there are no conflicts between the transactions numbered 1097 or 1099 and any lesser-numbered transaction.

6.4. Garbage Collection

Each time a new version of an object is created, the immediately lesser-numbered version of the object becomes potential garbage -- "potential" garbage since there may currently be queries executing that will need to access this version. Whether or not an object is "true" garbage can be determined by the current minimum value of all *QN*s, say, *min QN*. If this number is greater than or equal to the version number of the *new* version of an object, the preceding version can then be garbage-collected, since all current and future queries will now access versions of this object with version numbers greater than or equal to that of the new version.

Note: in the case that *WPCL copies* are associated with queries, then taking the *QN* of a query to be the minimum of its associated *WPCL* copy, the above reasoning still applies.

Garbage can be collected as soon as it is generated by maintaining a *garbage list* (*GL*) as follows. The gabage list is a list of (*version number*, *version set*) pairs, where a version set is a set of (*object ID*, *version number*) pairs -- i.e., each element of a version set refers to a particular version of a particular object. If a new version, with say version number *NV*, of the object with ID *ID* is written, and *OV* is the version number of the preceding version (assuming there is one), then the *GL* is updated by adding (*ID*, *OV*) to the version set in the *GL* associated with version number *NV* (creating a new version set if necessary). In the case that the new version is a deleted version, (*ID*, *NV*) is also added to this set. Potential garbage objects can now be collected as soon as they become true garbage by freeing all objects in the version sets of the garbage list associated with version numbers *NV* less than or equal to *min QN*, for each new value of *min QN*. Finally, *min QN* can be continuously updated by recalculation upon each query completion, or it could be periodically updated. An example is as follows.

Let the *GL* currently be

(1096, {(1,1095), (2,1001)}), (1097, {(1,1096), (4,998)}), (1098, {(3,1001)}),

and let *min QN* = 1095. Now, if version 1097 of the object with ID 5 is written, and the largest numbered version of this object was previously 1050, the *GL* becomes:

(1096, {(1,1095), (2,1001)}), (1097, {(1,1096), (4,998), (5,1050)}), (1098, {(3,1001)}).

Later, after all queries with *QN* = 1095 have completed, *min QN* increases, to 1096

say (i.e., assume there is still an uncompleted query with $QN = 1096$). Then, after garbage-collecting versions 1095 of object 1 and 1001 of object 2, the GL becomes:

$$(1097, \{(1,1096), (4,998), (5,1050)\}), (1098, \{(3,1001)\}).$$

6.5. Partitioning the Global Memory Manager

Parallelism can be introduced in the GMM by partitioning the set of shared data objects in some fashion (for example, if there are several secondary memory devices, if the constraint is made that all versions of an object must be stored on the same device, an object can be mapped to a partition corresponding to the device on which the object's versions are stored), and by using a process for each partition to manage storage allocation and the ID, version => physical address mapping for objects in that partition. Furthermore, the GL can be partitioned in the same fashion: if object p belongs to partition i, potential garbage versions of p are recorded in garbage list GLi, say. Each GL can be managed by a separate process, with a message interface between mapping and GL processes, or if the mapping and GL partition schemes are identical, the mapping processes can also perform garbage-collection. In order to balance free storage among the mapping processes, an additional process could be used to generate new object IDs, with IDs chosen in such a fashion that each newly created object maps to that partition containing the most free storage. Alternatively, for computer network applications for example, the LMM could always first try object creation via a mapping process that was "close" in the network, with other mapping processes used if this fails.

The *WPCL*, *WPCC*, and *QN* information are analagous to transaction information for the CC -- these structures can be managed by a single process, or by several processes accessing these structures in shared memory.

There does not seem to be any straight-forward way to distribute the *WPCL*, *WPCC*, and *QN* information in a way that would offer any performance advantages -- this is because these structures are all intimately connected with the central transaction number counter of the CC. In the types of proposed systems mentioned in Section 4.6 in which the use of a central transaction number counter is avoided by using timestamps or other schemes, these structures are simply omitted. The results are that queries must be controlled by the CC (however, by having queries access sufficiently old versions of the database, queries will almost never be aborted), and that there do not seem to be any algorithms for garbage collection other than heuristic techniques (for example, it might be assumed that any potential garbage version more than a day old could be deleted).

7. Transaction Processing on Cm*

In order to (1) develop algorithms for the concurrency control designs previously presented, (2) experimentally verify the correctness of these algorithms, (3) investigate the limitations of multiprocessor systems for transaction processing, and (4) demonstrate the usefulness of the policy module approach for policy experimentation on a complex system, a transaction processing system was implemented on Cm*/Medusa. In this chapter this system is described, the results of the experiments are presented, and some implications of these results are discussed. The concurrency control algorithms are given in Appendix II.

7.1. Overview of Cm*/Medusa

Cm*, a distributed multi-microprocessor designed and built at Carnegie-Mellon University (see [Swan et al 77]), currently consists of 50 *computer modules* (Cms) and five *communication controllers* (Kmaps), as shown in Figure 7.1. Each Cm consists of a DEC LSI-11 microprocessor, primary memory of 64K or 128K bytes, various devices, and a *local switch* (Slocal). The Slocal contains relocation tables that allow each memory reference to be mapped either to memory or devices on the associated LSI-11 bus (a *local reference*) or to be passed to the Kmap for the cluster (a *non-local* reference). Each Kmap is a microprogrammable microprocessor specially designed as a communication controller, and is responsible for mapping non-local memory references either to another Cm in the same

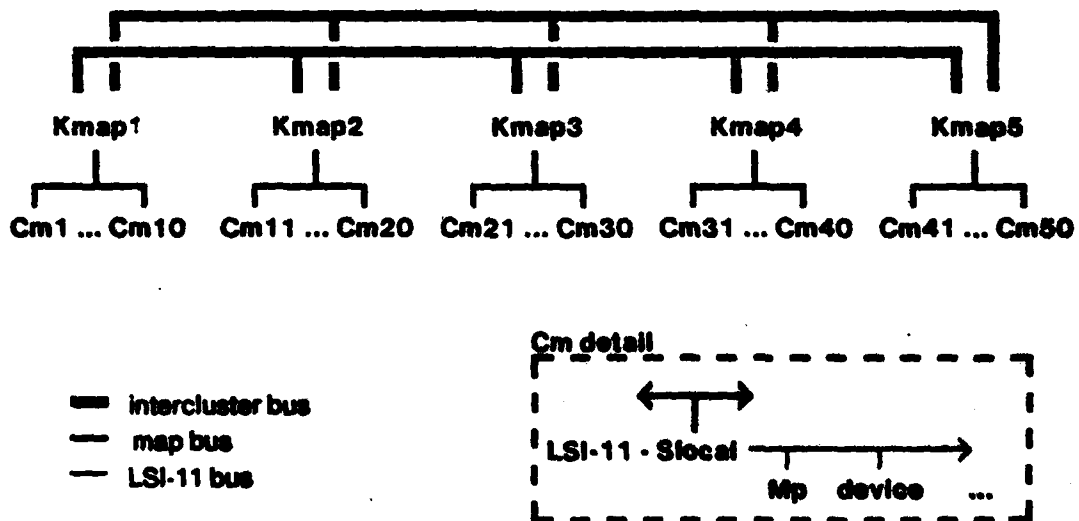


Figure 7.1. Cm* Architecture

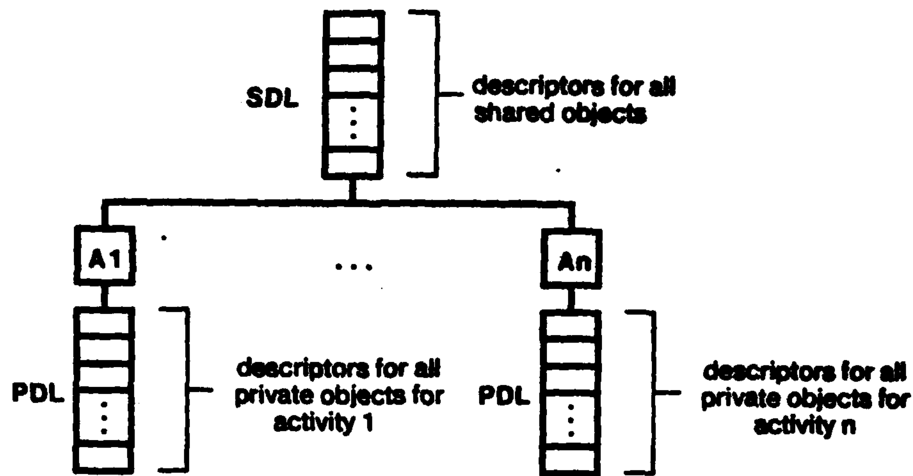


Figure 7.2. Medusa Task Force Structure

cluster (an *intracluster* reference), or through another Kmap to a Cm in a different cluster (an *intercluster* reference). However, because the Kmaps are microprogrammable, this mapping can take place in many different and complex ways. In particular, it is possible to microprogram key operating system communication primitives in the Kmaps.

Medusa (see [Outsterhout et al 80]) is one of two operating systems designed and implemented for Cm* (the other is StarOS -- for more detailed descriptions of Cm*, Medusa, and StarOS, along with a variety of information regarding Cm*-related research, see the research review [Jones and Gehringer 80]). The two primary uses of the Kmaps under Medusa are for message communication and address mapping -- these functions are implemented in the Kmap microcode. Message communication in Medusa takes place using objects called pipes, and is an extension of the Unix pipe mechanism (see [Ritchie and Thompson 74]). Here, it need only be noted that the extensions are such that the assumptions of Section 3.1 regarding communication between subsystems can be satisfied using mechanisms already provided.

Medusa provides a structure called a *task force* to implement operating system functions and user programs. A task force is a collection of *activities* (or processes), each of which can reference a distinct collection of private objects, such as code pages, and all of which can reference a single collection of shared objects, such as communication pipes or shared data pages. Access to an object is gained through a *descriptor list*; thus, for each task force there is a *shared descriptor list* (SDL), and for each activity there is a *private descriptor list* (PDL). This task force structure is shown in Figure 7.2. The mapping of an access to an object through a descriptor list is supported by the Kmap microcode; in particular,

descriptors are cached in the Kmap, so that access to a non-local object can usually proceed without the extra step of fetching a descriptor from a descriptor list. In the case that the object is local to the activity accessing the object, a simple access (i.e., an access produced by an LSI-11 instruction) can proceed directly through the Smap without involving the Kmap. Because of this, all code and private data pages are typically made local (if possible) for performance reasons.

Medusa supports various operating system functions through a collection of task forces called *utilities*. Utilities are special kinds of task forces in that, among other differences, a descriptor list of input pipes to the utilities, called the *utility descriptor list* (UDL), is stored in each Cm. Thus, any activity running on any Cm can invoke an operating system function by sending a message to the utility implementing that function through the appropiate pipe in the UDL. Of the utilities provided by Medusa, the only one that was used during the course of the experiments described below was the *file system* (other utilities were used during startup and after the completion of experiments). The file system utility handles all input and output devices, and provides a hierarchical file structure.

7.2. The Transaction Processing System

The transaction processing system was implemented as a single task force of eleven activities: a *master* activity, eight *transaction-processor* activities (TP1, TP2, ..., TP8), a CC activity, and a GMM activity. The SDL/PDL structure of this task force is shown in Figure 7.3. For the shared Cm memory system (see below), the SDL shown in the figure was extended to include 48 descriptors for shared Medusa (4096-byte) pages.

All experiments took place using a three-cluster partition of the system. In each case, all activities (including utility activities) were allocated their own Cm, and all code, stack, and data pages were local. Since Medusa did not support context swaps, activities were always resident in their respective Cms. The data objects supported at the virtual internal level were 512-byte pages.

The CC activity implemented all functions needed by the CC paradigm, and a policy module providing all basic policies was used, with deadlock detection and request queueing extensions as described in Section 5.2. At the start of each experiment a policy was chosen by sending the CC activity a message containing the options to be used by the policy module. For the policy experiments, the following four policies were used.

locking: read, write, read/write - *wait* (the validation option is redundant).
lock-opt: read - *grant*, write - *wait*, read/write - *wait all*, validation - *wait*.
opt-lock: read, write, read/write - *grant*, validation - *wait*.
optimistic: read, write, read/write - *grant*, validation - *kill*.

For the throughput experiments, the locking and optimistic policies were used.

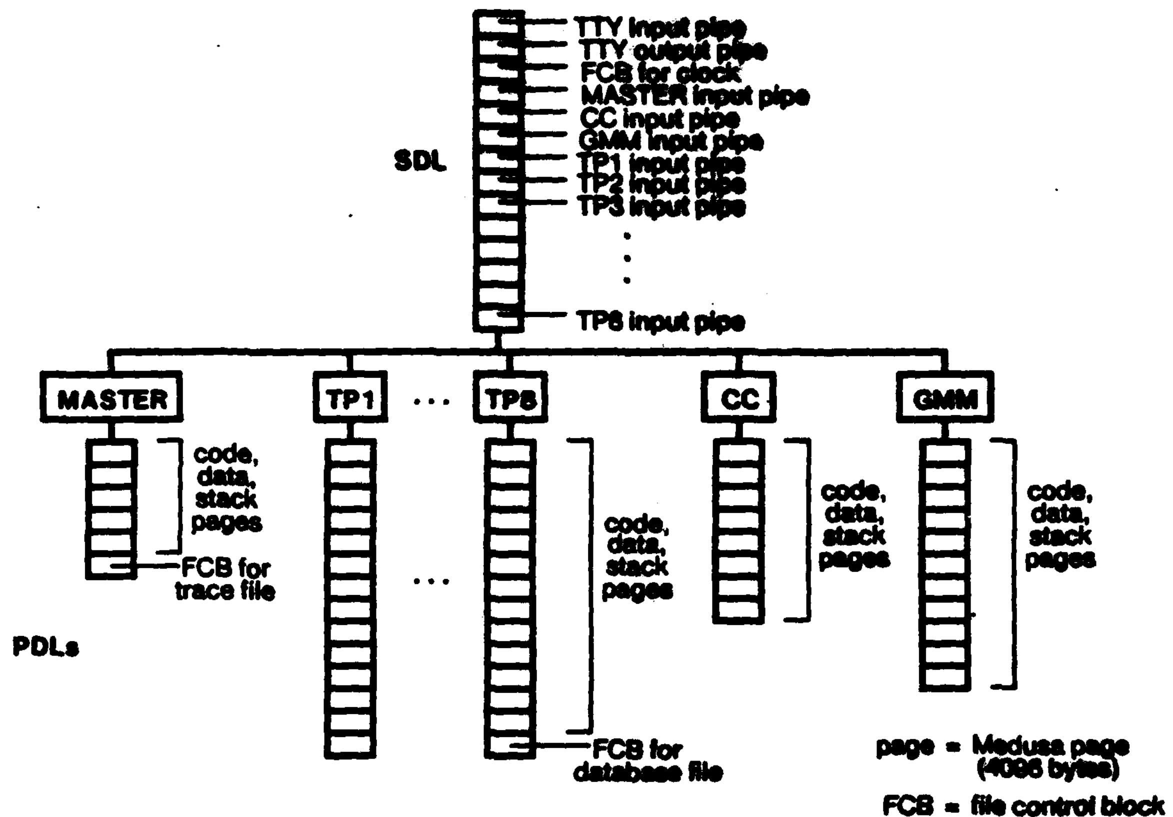


Figure 7.3. Transaction Processing System Structure

The GMM activity used the first scheme for query support of Chapter 6. The ID, version => address mapping used by the GMM, along with the *WPCL*, *GL*, *QN* table, etc., were stored in primary memory.

The structure of the transaction processor activities was as follows:

> transaction/query generator - generate random insertions, deletions, and queries.
>
> RcdM - implement a file structure based on a collection of 512-byte pages as shown in Figure 7.4 -- this RcdM attempted to optimize storage use by packing new records into existing record pages adjacent under the leaf page of the first index if possible.
>
> LMM - local memory manager -- all local memory that was available after allocation for code, data, and stack was used as LMM cache space, with a resulting maximum cache size of 42 512-byte pages (this space was also used for write-phase support), and the page-replacement policy was LRU (least recently used page replaced first).

The RcdM provided functions for creating and deleting relations, and for inserting, deleting, and finding tuples using the available indexes. The transaction/query generator generated

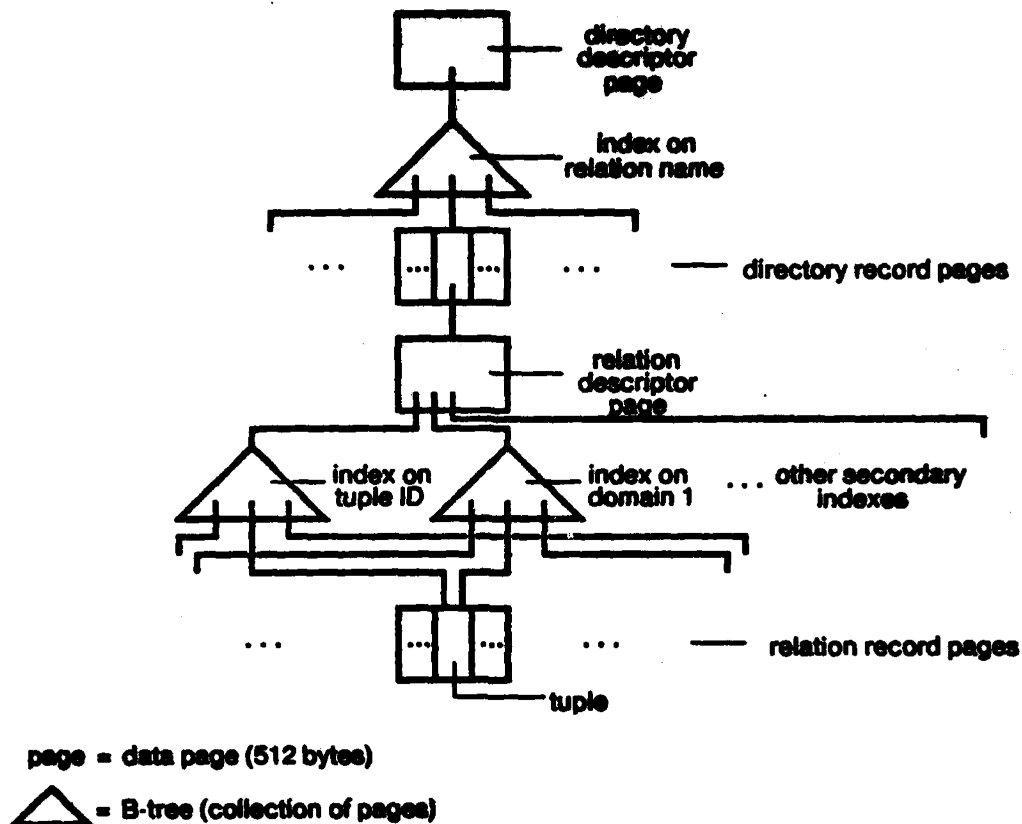


Figure 7.4. File Structure Used by Record Manager

and executed random insertions, deletions, or queries to find one tuple, based on messages from the master activity giving the relation name, operation, and miscellaneous additional parameters.

In all of the experiments, a previously created database consisting of 500 tuples in three relations was used. Although the RcdM supported variable length tuples, for the experiments only fixed size tuples were used. The three relations were as follows: relation A had 4 domains, of lengths 10, 10, 3, and 10, with indexes on tuple IDs and the first domain; relation B had 5 domains, of lengths 10, 10, 1, 2, and 10, with indexes on tuple IDs and the first domain; relation C had 3 domains, of lengths 10, 10, and 5, with indexes on tuple IDs and the first and second domains.

Tuple IDs were generated by incrementing a counter, and tuples were indexed on the reversed binary digits of their IDs for the reasons mentioned in Appendix 1. An insertion took place by randomly selecting one of the relations (any relation equally likely), getting a new tuple ID, and then using this value as the seed for a random string generator to fill

the contents of the tuple. A query to find one tuple first randomly selected a relation (any relation equally likely), then selected an index (any index for the relation equally likely), then selected an index value (any index value equally likely), and then found and retrieved the first tuple with a corresponding domain value greater than or equal to the key if there was such a tuple, otherwise the tuple with the maximal value for that domain was retrieved. A deletion proceeded in the same fashion as a query, except that the retrieved tuple was deleted. If a transaction failed due to a conflict, the master activity always immediately sent a message to the transaction processor activity to repeat the transaction. Below, this event is referred to as a *restart*.

Although this sytem used artificially generated transactions, it was based on an earlier "real" system (i.e., usable for applications). This system relied on the unique identification of tuples to support interactive examination and modification of the database *without* user interaction during the course of a query or transaction. Although the only transactions defined were insertions and deletions, and all defined queries were queries to find a single tuple (in various ways), the user was generally given the appearance of exclusive interactive access to the database by "remembering" the state of the user, in particular the contents and the unique identification of the most recently accessed tuple, between transactions and queries. Thus, the master activity was designed to simulate to a limited extent the behavior of a number of user interfaces of this type. In practice, the master activity would be replaced by a collection of user interface activities, as shown earlier in Figure 2.5.

In addition to driving the transaction processor activities, the master activity collected a trace of the experiment. In order to see what information was collected during a trace, part of a trace file is shown in Figure 7.5.

7.3. Maximum Throughput Experiments

As there are few multiprocessor transaction processing systems in existence, their limitations are of interest. Using the locking and optimistic policies, experiments were performed to investigate the maximum throughput as the number of transaction processor activities was increased. In these experiments 100 insertions or deletions were performed, either equally likely. Shared memory was accessed by random file access through the Medusa file system. The master activity, upon receiving a completion message from a transaction processor activity, always immediately sent a message to begin a new transaction (if there were any transactions left to perform). The observed throughputs are shown in Figure 7.6.

Separate experiments with the Medusa file system determined that the file system activity became a bottleneck for this system as the number of transaction processors increased. In order to see the effects of removing this bottleneck, a new system was developed in which the unused memory of four 128K Cms was used as shared memory, accessed through the SDL (the file system was still used by the master activity to read clock values and to write the trace file). Reads or writes of 512-byte pages were performed using a block move

```
10:06.46 TQG 1: insert 1105
10:06.61 TQG 1: read conflict has occurred, aborting
4 0 0
10:06.96 TQG 1: insert 1105
10:06.98 TQG 5: OK
11 7 4
10:07.05 TQG 5: wait 7
10:07.06 TQG 2: OK
20 15 4
10:07.11 TQG 2: wait 3
10:07.15 TQG 7: OK
10 2 5
10:07.18 TQG 7: wait 3
10:07.20 TQG 8: select A
10:07.23 TQG 8: insert 1106
10:07.35 TQG 3: select C
10:07.36 TQG 3: delete 13671
10:08.26 TQG 1: OK
13 3 3
10:08.30 TQG 1: wait 4
10:08.80 TQG 3: OK
11 6 4
10:08.83 TQG 3: wait 3
10:08.86 TQG 8: OK
11 8 3
```

Figure 7.5. Part of an Experiment's Trace

operation provided by the Medusa Kmap microcode. This system was not believed to be unrealistic, since it is possible to transfer data to disks at the maximal Kmap block move rate of approximately 300 512-byte blocks/second. The observed throughputs for this new system are shown in Figure 7.7.

These experiments clearly show that significant increases in throughput are possible for transaction processing using multiprocessor architectures, even when the database is highly shared. However, there are two limitations on the increases that can be achieved: shared memory bandwidth and transaction conflict.

With respect to shared memory bandwidth, using the Medusa file system, no increases in throughput could be achieved with more than four transaction processors. The bottleneck would have occurred even earlier if objects were not cached in local memory: in the case of the four transaction processor locking experiment using the file system, out of an average of 14.04 read requests to the GMM, an average of only 5.72 pages had to be read from shared memory, giving a "cache-hit" ratio of 59% (the meaning of cache-hit here is somewhat unique, in that once the LMM determines that a local copy is the correct version, no further messages to the GMM take place). This ratio was typical for all experiments.

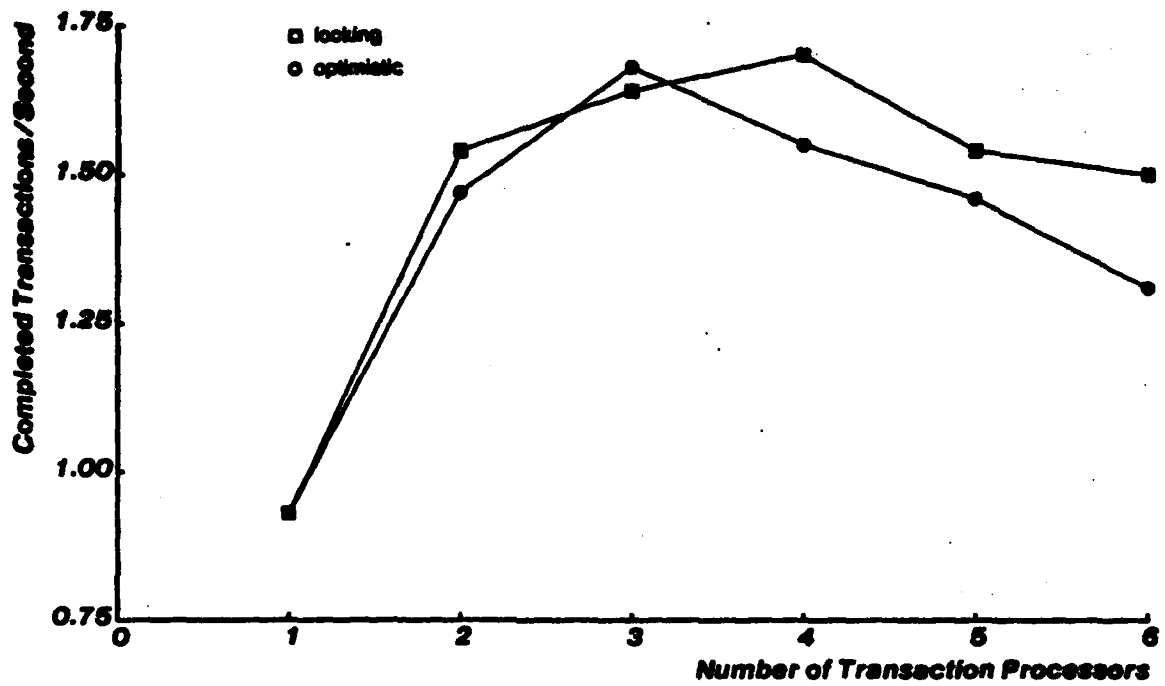


Figure 7.6. Throughput using Medusa File System

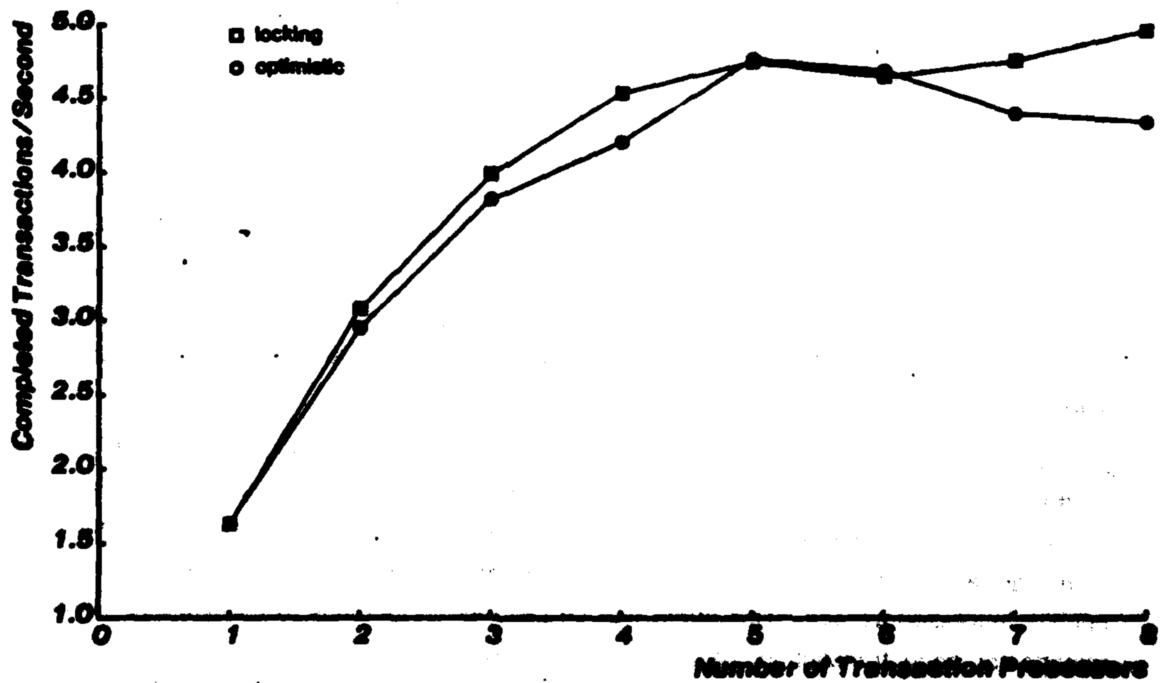


Figure 7.7. Throughput using Kmap Block Moves

Other approaches to reducing the shared memory bandwidth limitation rely on RcdM design. As noted above, the RcdM used for these experiments attempted to optimize storage by packing records into "nearby" record pages if possible (this RcdM was earlier developed on a system in which secondary storage was at a premium). The result is that for some transactions, a large number of pages were examined (for the four transaction processor locking experiment using the file system, the maximum number of read requests to the GMM for any one transaction was 23). A RcdM using a simpler record storage allocation scheme would encounter the shared memory bottleneck at a later point; however, storage would be utilized less effectively, and queries could be more expensive. As noted in Appendix I, there are many alternatives in RcdM design, and it is a field of on-going research.

The most direct way to approach this limitation, though, is to increase shared memory bandwidth. In the experiments using shared Cm memory, for example, a dedicated disk controller could be used on each shared memory Cm, with the primary memory of the Cm used as a large buffer. This example illustrates two techniques: provide more parallelism in the path to shared memory (multiple disk controllers), and use intermediate levels in the memory hierarchy (primary Cm memories).

The transaction conflict limitation is more difficult to avoid, since it has the effect of making additional parallelism useless: if there is a conflict between two transactions, they must (in the general case) proceed sequentially.

Again, RcdM design plays an important part. If records were not indexed under their reversed IDs in these experiments, there would have been conflicts between almost every set of concurrent insertions to the same relation. On the other hand, a RcdM using a simpler record storage allocation scheme could have had fewer conflicts since the read set sizes would have been smaller.

There is also a problem at the virtual internal level: if conceptual entities are mapped to larger internal entities, conflicts can occur between transactions that do not conceptually conflict. Given the framework of Chapter 2, the only solution is to decrease the granularity of the objects provided at the internal level. Other approaches rely on introducing application-dependence into the concurrency control so that larger classes of transaction histories are allowed (e.g., see [Kung & Papadimitriou 79]), but are beyond the scope of this work.

The effects of transaction conflict can be seen in Figures 7.6 and 7.7, particularly in the case of the optimistic policy -- for the optimistic policy, when a transaction is validated, all conflicting transactions are aborted. In these experiments, the degree of concurrency increased as the number of transaction processors increased, and so the probability of conflict increased as well. This effect can be seen as an increase in the average number of restarts for each transaction, as shown in Figures 7.8 and 7.9. Note that restarts occur much less using the locking policy, since a transaction is aborted only if scheduling its request would cause deadlock.

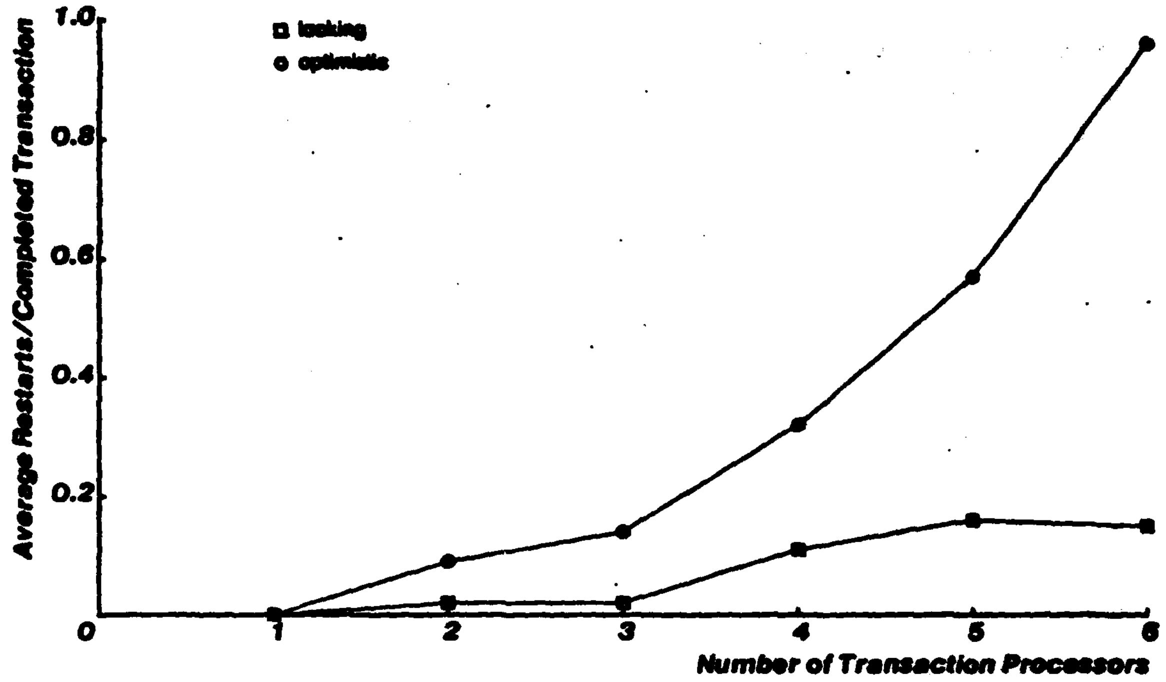


Figure 7.8. Number of Restarts using Medusa File System

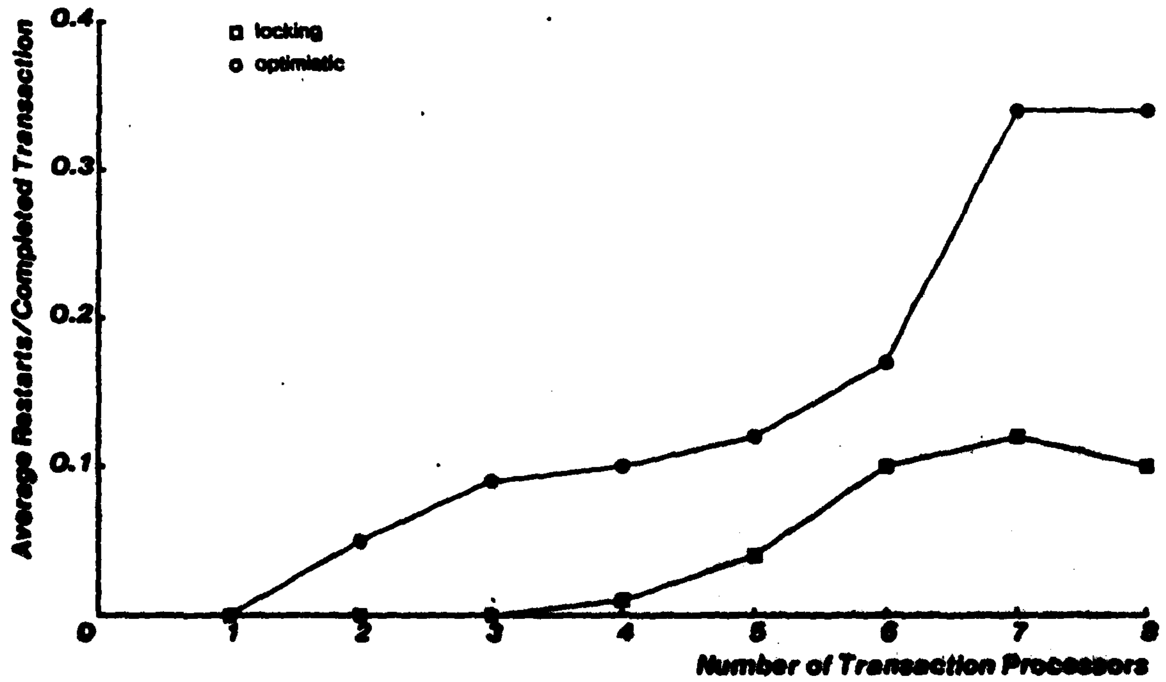


Figure 7.9. Number of Restarts using Kmap Block Moves

For the locking policy, increased transaction conflict results in increased waiting due to scheduling. However, as concurrency increases, there is also increased waiting on shared system resources (for this system, these are the file system, the Kmaps, and the master, GMM, and CC activities). The average execution times (including waiting times) for transactions that were successful on the first attempt under the locking and optimistic policies are shown in Figures 7.10 and 7.11. Since these curves are almost identical, and under the optimistic policy there is no waiting due to scheduling, it must be concluded that for this system waiting due to scheduling is negligible as compared to waiting on shared system resources. This explains why the locking policy generally gave higher throughput: since waiting due to scheduling is negligible, the effect of restarts dominates. In those three cases in which the optimistic policy gave slightly higher throughput, there was by chance a combination of less total work to perform using the optimistic policy (where total work was measured by the total number of shared page accesses) and a relatively small difference in restarts for the two policies. This can be seen by comparing Figures 7.12 and 7.13 with the previous figures.

7.4. Policy Experiments

Using a policy module that provides many policies, it is easy to investigate the performance of policies on a complex system. Using the locking, lock-opt, opt-lock, and optimistic policies, a *number* of experiments were conducted as a demonstration. The experiments were as follows.

1. The shared Cm memory system was used, with eight transaction processors.
2. 500 queries, insertions, or deletions were performed.
3. Transaction processor activities waited a randomly generated amount of time between transactions and queries, from 0 to 2 seconds.
4. The probability of a query was 1/2, and insertion and deletion probabilities were each 1/4.
5. Three different experiments with three different inital databases were conducted for each policy by varying the *initial seed for the random number generator.*

The results of these experiments are shown in Table A.

Since for the Cm* system the effect of waiting due to scheduling is negligible, the locking policy uniformly gave the best throughput, as expected. However, in the first set of experiments, the difference in transaction conflicts between the locking policy and the other policies was less than in the latter two sets of experiments. The result was that in this first set of experiments, both the optimistic and lock-opt policies gave better average response times than the locking policy, with the lock-opt policy giving the best response time -- the increased restarts were not enough to cancel out the decreased response time resulting from less scheduling. In the latter experiments the difference in restarts was greater, and

since response time includes the time taken by unsuccessful transactions, the locking policy gave the best response.

POLICY	RUN	NUMBER OF RESTARTS[A]	PAGE ACCESSES[AB]	EXEC. TIME FTS[CD]	EXEC. TIME FTF[CD]	EXEC. TIME RS[CD]	EXEC. TIME RF[CD]	RESPONSE TIME[AD]	THROUGHPUT[E]
locking	1	.02	3.38, 1.81	.60	.82	.78	F	.61	5.87
lock-opt	1	.03	3.31, 1.59	.55	.59	.78	.81	.57	5.77
opt-lock	1	.06	3.83, 1.86	.62	.74	.85	.89	.67	5.62
optimistic	1	.05	3.64, 1.64	.56	.64	.86	.41	.60	5.73
locking	2	.01	3.78, 1.71	.57	.74	.71	F	.58	5.85
lock-opt	2	.05	3.79, 1.71	.57	.59	.84	.79	.61	5.75
opt-lock	2	.07	4.06, 1.84	.60	.63	.88	.72	.65	5.62
optimistic	2	.05	3.54, 1.58	.55	.61	.90	.56	.59	5.69
locking	3	.02	4.06, 1.76	.58	.61	.80	.75	.59	5.73
lock-opt	3	.06	4.04, 1.67	.57	.65	.90	.62	.63	5.55
opt-lock	3	.07	4.11, 1.94	.58	.73	.78	.67	.63	5.53
optimistic	3	.06	4.00, 1.92	.60	.58	.77	.58	.66	5.51

[A] Averaged over all completed transactions or queries.

[B] shared pages read, shared pages written.

[C] FTS = first time success (including queries), FTF = first time failure, RS = restart success, RF = restart failure.

[D] Times in seconds.

[E] Completed transactions or queries/second.

[F] Did not occur.

Table A. Policy Experiments

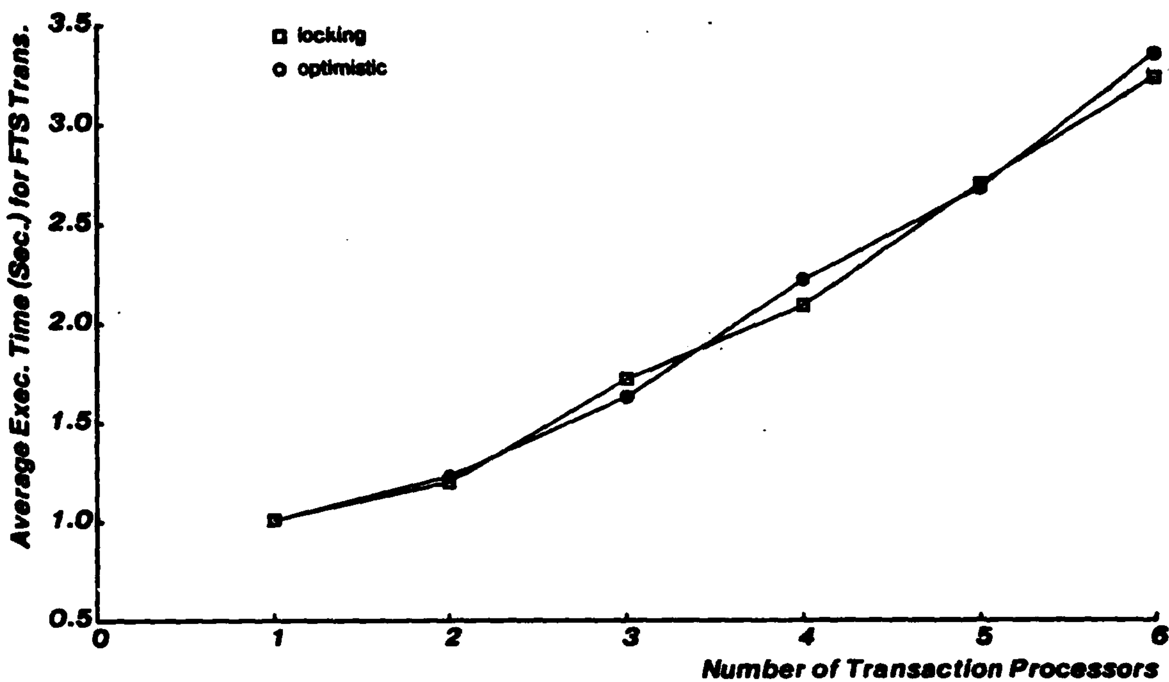


Figure 7.10. Execution Time using Medusa File System

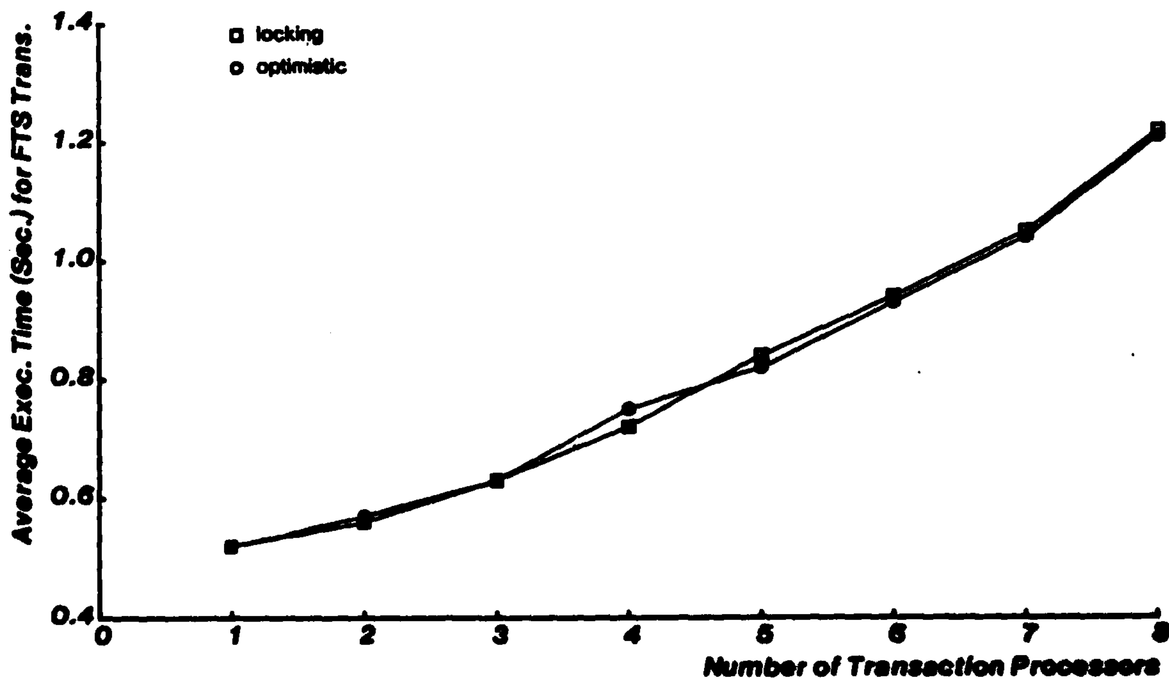


Figure 7.11. Execution Time using Kmap Block Moves

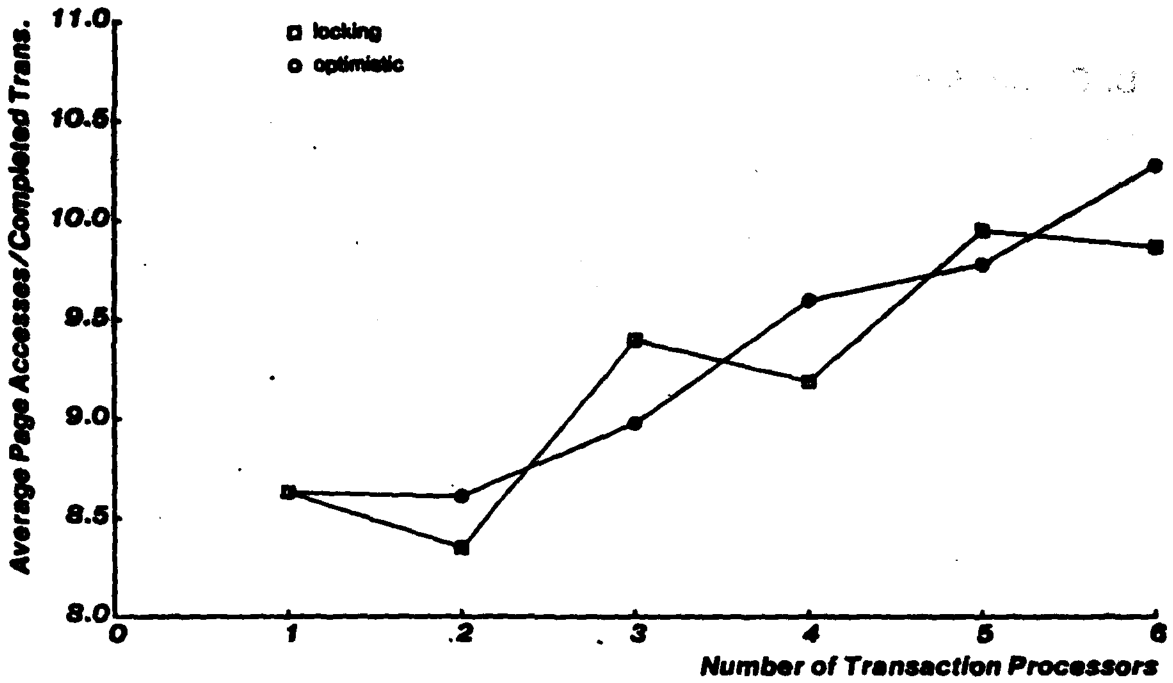


Figure 7.12. Number of Shared Page Accesses using Medusa File System

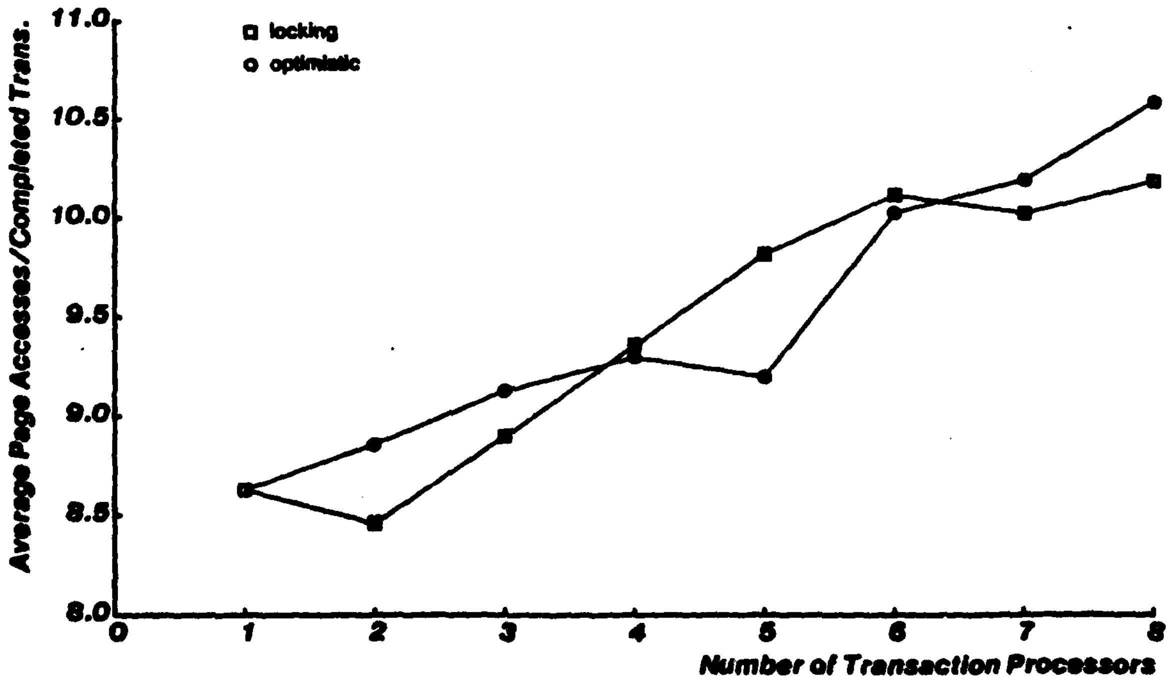


Figure 7.13. Number of Shared Page Accesses using Kmap Block Moves

8. Conclusion

Now that the four-level design framework, concurrency control paradigm, global memory manager designs, and Cm* experiments have been presented, various issues concerning this approach to transaction processing system design can be considered in more detail. In this concluding chapter, first, various of these issues are considered in turn. Next, conclusions drawn from the implementation experience are given, and implications of the Cm* experiments are discussed. Finally, some directions for future research are identified.

8.1. On the Use of Physical Pointers

Currently, it is common practice to use physical pointers to data objects in database record managers. The efficiency advantage is that, given a pointer to an object, the object can be retrieved without first looking up the physical address corresponding to the pointer. However, this advantage disappears as soon as a memory hierarchy is introduced. Using a memory hierarchy, an object can possibly be stored at several locations, and so some form of lookup is necessary in any case. It seems clear that memory hierarchies are necessary in all but centralized unshared database applications.

8.2. On Multi-Version Objects

Once it has been decided that objects will be referred to virtually, and that ID => address mappings will be maintained, there are clear advantages to extending these mappings to support multi-version objects. First, as observed in the introduction, it is common in transaction processing systems to have large queries. Without multi-version objects, queries can observe consistent database states only with concurrency control support. It is clearly undesirable to abort large queries, and so some policy giving priority to large queries must be used. Alternatively, using a hierarchical locking concurrency control, the query could begin by read-locking large portions of the database. In either case the result is that a large portion of the database cannot be modified by transactions while the large query is in progress. With multi-version objects, using one of the schemes for query support presented earlier, queries do not affect concurrent transactions (except perhaps by freezing garbage-collection -- see below). Some other advantages to the use of multi-version objects are that in distributed systems version numbers can be used to determine when copies are "out-of-date" (as in the Cm* system), and that versions (together with a write-phase-completion list, counter, or similar information) form a basis for recovery at the memory management level.

An objection that has been raised to multi-version object schemes is that extra storage is necessary. If old versions are quickly garbage-collected, this does not in general present a significant problem, and if version numbers are managed by conceptually centralized entities, such garbage-collection is possible: a garbage-collection algorithm was designed and implemented in which old versions are freed at the first point at which it can be guaranteed that no future transaction or query will access that version. It is still possible, though, for garbage-collection to be frozen for a long period of time due to the execution of a large

query. However, in single-version schemes, many transactions would be forced to wait for the completion of the large query, whereas in a multi-version scheme, all transactions can continue to run until storage is exhausted. Also, it is possible to transfer old versions to tertiary memory, as noted earlier in Section 3.4 -- this would allow the large query to continue to run (but at a much slower rate), and would free secondary memory space.

8.3. On General Concurrency Controls

Having removed query support as a concurrency control function, the problem of designing an efficient general-purpose concurrency control is greatly simplified. Nevertheless the question remains whether building record manager dependence into the concurrency control can greatly improve efficiency. In terms of run-time efficiency, this question can probably never be answered in a final way, since any modification of a transaction processing system's access structures could in principle introduce a variety of new specialized concurrency controls, all of which would have to be compared to many general concurrency controls. Furthermore, even given a demonstrably good specialized concurrency control for some access structure, there are currently no techniques for generalizing the concurrency control to the case in which the access structure is combined with other structures. For example, none of the special locking protocols developed for B-trees (see [Samadi 76], [Bayer and Scholnick 77], [Miller and Snyder 78], [Ellis 80], or [Lehman and Yao 81]) can be applied directly to the record manager used in the Cm* system -- although this record manager uses B-tree indexes, there is no global tree structure, since the index records of several B-trees can all point to the same tuple. So regardless of the run-time efficiency of specialized concurrency controls, there are clearly development and maintenance advantages for general concurrency controls.

Based on the Cm* experiments, it seems that general concurrency controls can provide enough concurrency to effectively utilize parallelism, giving significant increases in throughput. It is important to realize, though, that these results depend on the fact that a record manager was used in which conceptually small transactions were usually physically small as well, and in which conceptually non-conflicting transactions were usually physically non-conflicting. Since such properties are highly desirable for record managers in any case, one cannot seriously object to the fact that the efficiency of a general concurrency control depends on these properties. However, because the use of record managers with these properties is so important in the four-level architecture, earlier work on the problem of designing general index structures for such record managers is reported in Appendix I.

8.4. Implementation Experience

Two conclusions can be drawn from the Cm* transaction processing system implementation experience. First, the four-level framework really is valuable in the development of transaction processing systems. In the case of the Cm* system, the record manager was earlier developed on the DEC PDP10 architecture under an operating system (TOPS10)

completely different from Medusa, using a different concurrency control and a different memory manager (the concurrency control was an early implementation of the optimistic method, and the memory manager supported only single version objects). Luckily, dialects of a common language (BLISS -- see [Wulf et al 71]) were available on both systems, but this should often be the case for high-level languages. The only modifications necessary to the original record manager, other than those due to differences in the language dialects, were due to a lack of information-hiding in the original design -- the original record manager managed its own local page memory as a stack of pages and a few temporary pages. By replacing these pages with pointers to pages and by moving local memory management to the LMM, a more general design resulted. In fact, the modified record manager could be used on both systems, and was actually debugged and tested on the PDP10 system. Since the record manager was by far the most complex subsystem, this speeded development time immensely.

Second, if a concurrency control module providing all functions needed by the paradigm is implemented first, it is then easy to implement any particular general concurrency control as a policy module. In the Cm* system, the policy module providing all basic policies with deadlock detection and request queueing consisted of 72 lines of (BLISS) code. The module providing all functions needed by the paradigm can be thought of as a kernel in the HYDRA sense: the HYDRA operating system kernel was defined to be a set of facilities "which are both necessary and adequate for the construction of a large and interesting class of operating environments" [Wulf et al 74]. By replacing "a large and interesting class of operating environments" with "all general concurrency controls", a general concurrency control kernel is defined.

8.5. Implications of the Cm* Experiments

One conclusion that might be drawn from the Cm* experiments is that, since the effect of waiting due to scheduling was negligible, future investigations of concurrency controls should concentrate on locking-style policies. This conclusion is tentative at best: Cm* is currently a unique system, partly multiprocessor, partly computer network, and it is not at all clear that this result applies to dissimilar systems. Also, in these experiments individual processors were not multiprogrammed, but the expense of scheduling could be drastically increased if scheduling required context swaps. However, using a concurrency control policy module it is easy to perform initial experiments on *any* system to determine if this same situation holds, and so questions regarding the general applicability of this result seem somewhat unimportant. For example, it might be the case for some system that the effect of restarts was negligible (due to extremely high cache-hit ratios on restarts, say), but that waiting was expensive (due to context swaps, say) -- after having determined this, policy developers and maintainers for this system would simply concentrate on policies that avoid scheduling.

A more far-reaching conclusion can be drawn from the fact that the performance differences

among the various policies tested were insignificant when compared to the performance differences using the two types of shared memory. The implication is that long-term research in transaction processing system design should concentrate more on lower-level systems problems, such as increasing memory and communication bandwidths, than on concurrency control problems. A concurrency control policy module, on the other hand, can be seen as a maintenance and "tuning" tool that is most useful *after* a transaction processing system has been developed.

8.6. Further Research

Several areas for further research can be identified. First, here the case in which the concurrency control has near-minimal information about transactions has been examined: the concurrency control is informed only of the ID of each object as it is accessed, and whether it is a read or write access. It has been argued that this approach works well in most transaction processing systems (given good record manager design and concurrency-control-less queries), and that this approach has the advantage of separating concurrency control design from database design. Nevertheless there are many systems in which this approach is unacceptable. For example, some databases used for artificial intelligence applications consist of numerous highly interconnected objects, and currently it does not seem possible in these systems to maintain global consistency with small independent transactions. Also, in network database systems, it may be desirable to transfer function requests among nodes (e.g., "insert tuple T in relation R") instead of data. Although an access-driven concurrency control could be used at each node, a function-driven concurrency control could prove necessary at the global level. A problem for future research then, is the generalization of the policy approach to those cases in which additional information is available to the concurrency control.

Another issue that has not been explored here is the manner in which copies of objects are handled in distributed systems. The concurrency control design that has been developed here applies directly to the case in which an object and all of its copies are identified as a single object, and it also applies to the case in which each copy is considered to be a distinct object. In the latter case, though, the concurrency control can take advantage of the knowledge of which objects are copies. This approach has been important in the development of robust concurrency controls (e.g., the voting algorithms of [Thomas 79]). This can be seen as another example of a case in which it is desirable to make additional information available to the concurrency control.

Next, although the policy module approach can greatly reduce the need for performance analysis of concurrency controls, it certainly does not eliminate it. For example, in order to *automatically* switch to the optimal concurrency control method based on performance monitoring or usage statistics, a deeper understanding of the performance characteristics of alternative concurrency control methods is necessary.

Finally, a concurrency control can be used to maintain consistency while a transaction processing system is in operation, but a recovery subsystem is necessary to restore an inconsistent database to a consistent state. Thus, the concurrency control and recovery subsystem are in this sense equally important. Here, concurrency and recovery problems have been almost completely isolated through the read/write phase mechanism: in the read phase, no recovery support is necessary, since no shared object is modified; in the write phase, no shared object is read, and so no concurrency control interaction is necessary, *with the exception* of informing the concurrency control when the new versions of objects written in the write phase can be accessed by other transactions. It is in exactly this case that concurrency control and recovery interact. For example, it may be desirable for recovery reasons not to make new versions of objects accessible by other transactions until they have been written to duplexed disks, say. A solution is to assign the recovery subsystem responsibility for informing the concurrency control when new versions of objects are accessible. In any case, alternatives in communication between recovery subsystems and concurrency controls, and application of the policy approach to recovery subsystem design, are problems for future research.

Appendix I. Design of Record Managers

As noted in Chapter 1, in practice, most transactions are conceptually small. If the database is organized as a collection of pages and the record manager can be designed so that a conceptually small transaction is usually physically small as well (i.e., accesses a small number of pages), then concurrency control at the granularity of pages will be appropriate.

One problem in designing the database so that conceptually small transactions are usually physically small is that various search structures may be needed in order to support efficient queries, and these search structures will have to be periodically updated as transactions modify the database. In order to see how a record manager can be designed so that search structures are kept up to date while still keeping transactions physically small, in this appendix an outline of a record manager will be presented as an example. This record manager does not presuppose any particular data model; in practice, any given data model would be supported at a higher record management level. Note that this is just an example, and that many details have been omitted; there are many alternatives in record manager design, as presented in numerous textbooks (e.g., see [Wiederhold 77], [Date 77], or [Ullman 80]) and elsewhere; furthermore, this is a problem of on-going research.

Although there are many data models that can be used at the conceptual level (the most popular being relational, hierarchical, network, and entity-relationship), all of these data models can be realized as collections of files of records. A record of type ($type_0$, $type_1$, ..., $type_{N-1}$) is an element of $domain_0 \times domain_1 \times \ldots \times domain_{N-1}$, where each $type_i$ is some primitive type (e.g., integer, string, etc.), and each $domain_i$ is the set of all values of type $type_i$. A *file* is a set of records all of the same type. The various data models result from decisions on whether or not pointers to records or files are allowed as primitive types, and if they are allowed, restrictions on their use.

It is useful for a variety of reasons to have a means of referring to existing records without referring to their locations (at the virtual internal level), for use as record pointers, for example. Therefore, let each record have a unique *record ID*. These can be generated by the *Mname* facility of the GMM. The advantage of not referring explicitly to a record's location (in terms of keeping transactions small) is that records may be moved for storage allocation purposes without requiring a large number of pointer modifications.

The basic operations defined on a file are inserting a new record, deleting an old record given its ID, and retrieving a record given its ID (assume for simplicity that a record update is handled by a deletion of the record to be updated followed by an insertion of the updated record, however keeping the old record ID). The problems now are (1) to find space in some page for inserting a new record, (2) to reclaim the unused space after a record has been deleted, and (3) to find a record given its ID. One structure that solves all of these problems nicely is the *B-tree* [Bayer and McCreight 72], of which there are many variants (see [Comer 79] for a survey). Assuming that several records can be stored per page, it can be used in this case by (1) maintaining an index of record locations on record IDs, and (2)

using the B-tree storage allocation scheme for records, that is, treating pages containing records as the leaf pages of the B-tree structure. With respect to (2), this means that storage will be utilized effectively by sometimes redistributing records among adjacent pages (adjacent in record ID space), creating or deleting pages as necessary. In the case that several records cannot be stored per page, groups of pages can be linked together, and the group treated as one large page, simulating the simpler case.

The B-tree structure has the properties that storage is utilized effectively (there is a guaranteed minimum utilization of 50%, 66%, or more, depending on the variant, with typically much higher utilization), that few pages are read in finding a record given its ID (typically, 3 or 4, depending on page size and number of records), and that usually only one page is modified for an insertion or a deletion (in the case that several records can be stored per page).

Note on the generation of record IDs: in the case that there are many concurrent insertions, it is not desirable that newly generated record IDs be close in record ID space, since otherwise there would be many conflicts. If record IDs are generated by using the current time or by incrementing a counter, this could be a problem, and a scheme to handle it is to index the record with ID *ID* under *F(ID)*, where *F* is some "randomizing" function, and then later find the record by searching for *F(ID)*. A simple example of an *F* is $F(a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_0) = a_0 2^{n-1} + a_1 2^{n-2} + ... + a_{n-1}$, that is, reverse the binary digits of the record ID.

The above structure is all that is needed in many database applications. These include some network and hierarchical applications in which all records are found by starting from a root record and then following pointers. However, if a pointer to a record is not available, finding a record given its ID is not a particularly useful operation. In such applications (for example, relational databases), *secondary indexes* may be needed for query support.

For the record manager described below, secondary indexes of the following kind will be supported.

1. There are index records of the form

$$key_0, key_1, ..., key_{K-1}, record\ ID,$$

   where key_i is an element of a finite totally ordered set $domain_i$, K is a constant, and *record ID* refers to a record (in the file) with these values as some of its fields.

2. It is desired to retrieve records based on queries of the form

$$min_i \leq key_i \leq max_i, \qquad 0 \leq i \leq K-1,$$

   i.e., *range* queries.

The problem of designing a search structure for this type of secondary index that has some of the same properties as B-trees has been investigated in [Robinson 81]. The resulting

structure was named the *K-D-B-tree* since it combines properties of K-D-trees [Bentley 75] and B-trees. A survey of data structures for range searching appears in [Bentley and Friedman 79]. The K-D-B-tree structure will now be presented.

Define a *point* to be an element of $domain_0 \times domain_1 \times \ldots \times domain_{K-1}$, and a *region* to be the set of all points $(x_0, x_1, \ldots, x_{K-1})$ satisfying

$$min_i \leq x_i < max_i, \qquad 0 \leq i \leq K-1,$$

for some collection of min_i, $max_i \in domain_i$. Points can be represented most simply by storing the x_i, and regions by storing the min_i and max_i.

Below, it will be required that certain regions be disjoint, and that their union be a region -- thus the strict inequality on the right hand side of the region definition above. However, it will also be required that the union of certain regions be all of $domain_0 \times domain_1 \times \ldots \times domain_{K-1}$. It is therefore necessary to create for each domain a special element ∞_i, which is greater than all elements of $domain_i$, and to allow the max_i to assume these values. It is also convenient to define $-\infty_i$ as the minimum of $domain_i$.

Like B-trees, K-D-B-trees consist of a collection of pages and a variable *root ID* that gives the page ID of the root page. There are two types of pages in a K-D-B-tree.

1. Region pages: region pages contain a collection of (*region*, *page ID*) pairs.
2. Point pages: point pages contain a collection of (*point*, *record ID*) pairs, where *record ID* refers to a database record. The (*point*, *record ID*) pair is in fact an index record.

The following set of properties define the K-D-B-tree structure. The algorithm for range queries given below depends only on these properties, and the algorithms for insertions and deletions are designed so as to preserve these properties.

1. Considering each page as a node and each page ID in a region page as a node pointer, the resulting graph structure is a multi-way tree with root *root ID*. Furthermore, no region page contains a null pointer, and no region page is empty (note that this, together with the fact that point pages do not contain page IDs, means that the point pages are the leaf nodes of the tree).
2. The path length, in pages, from root page to leaf page is the same for all leaf pages.
3. In every region page, the regions in the page are disjoint, and their union is a region.
4. If the root page is a region page (it may not exist, or if there is only one page in the tree it will be a point page), the union of its regions is $domain_0 \times domain_1 \times \ldots \times domain_{K-1}$.

5. If (*region*, *child ID*) occurs in a region page, and the child page referred to by *child ID* is a region page, then the union of the regions in the child page is *region*.

6. Referring to (5), if the child page is a point page, then all the points in the page must be in *region*.

Figure A1 illustrates an example 2-D-B-tree.

A range query can be expressed by specifying a region, the *query region*. It is convenient to think of regions as a cross-product of intervals $I_0 \times I_1 \times \ldots \times I_{K-1}$. If some of the intervals of a query region are full domains, the query is a *partial range* query; if some of the intervals are points and the rest are full domains, the query is a *partial match* query; if all of the intervals are points, the query is an *exact match* query.

The algorithm to output all records satisfying a range query specified by *query region* is as follows.

Q1. If *root ID* is the null page ID, terminate. Otherwise, let *page* be the root page.

Q2. If *page* is a point page, then for each (*point*, *record ID*) pair in *page* with *point* a member of *query region*, retrieve and output the database record with ID *record ID*.

Q3. Otherwise, for each (*region*, *child ID*) pair in *page* such that the intersection of *region* and *query region* is non-empty, set *page* to be the page referred to by *child ID*, and recurse from (Q2).

Next, for insertions, it is necessary to define the *splitting of a region* along element x_i of $domain_i$. Let the region be $I_0 \times I_1 \times \ldots \times I_{K-1}$. If $x_i \notin I_i$, the region is not changed by splitting. Otherwise, let $I_i = [min_i, max_i)$; splitting the region results in two new regions:

1. $I_0 \times \ldots \times [min_i, x_i) \times \ldots \times I_{K-1}$,

2. $I_0 \times \ldots \times [x_i, max_i) \times \ldots \times I_{K-1}$.

Region (1) is called the *left* region and region (2) the *right* region. If $x_i \notin I_i$ since $x_i < min_i$, the region is said to lie to the *left* of x_i; if $x_i \geq max_i$, the region is said to lie to the *right* of x_i. A point $(y_0, y_1, \ldots, y_{K-1})$ is said to lie to the *left* of x_i if $y_i < x_i$, and to the *right* of x_i otherwise.

A point page is split along x_i by creating two new point pages, the *left page* and the *right page*; then transferring all the (*point*, *record ID*) pairs in the page to either the left or right page depending on whether *point* lies to the left or the right of x_i; and then deleting the old page.

A region page is split along x_i by creating two new region pages, again called the left page and the right page; filling these pages with regions derived from the old region page; and

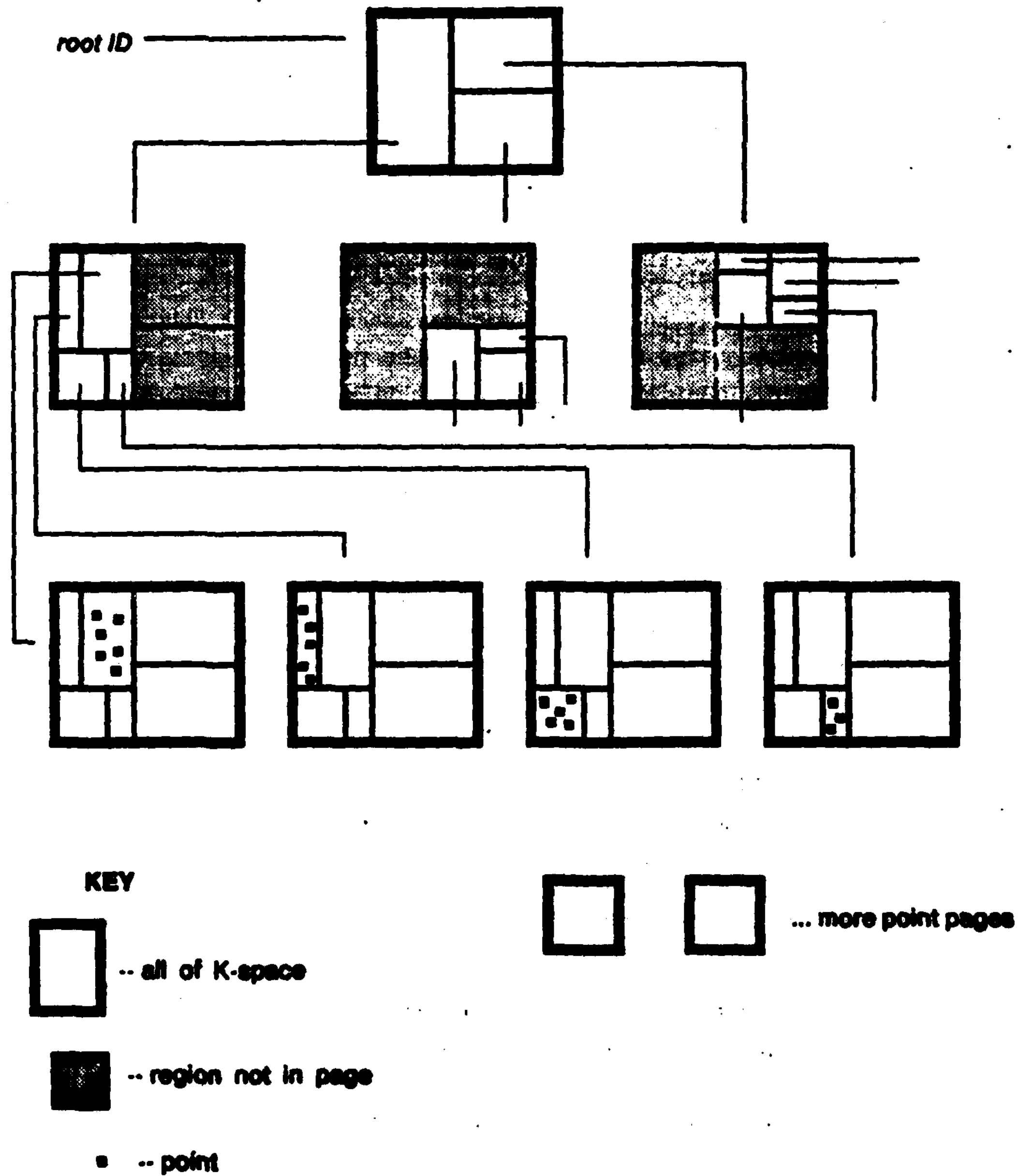


Figure A1. Example 2-D-B-Tree

then deleting the old page. This procedure takes place as follows. For each (*region, page ID*) in the old region page:

S1. If *region* lies to the left of x_i, add (*region*, *page ID*) to the left page.

S2. If *region* lies to the right of x_i, add (*region*, *page ID*) to the right page.

S3. Otherwise:

S3.1. Split the page referenced by *page ID* along x_i, resulting in pages with IDs *left ID* and *right ID*.

S3.2. Split *region* along x_i, resulting in regions *left region* and *right region*.

S3.3. Add (*left region*, *left ID*) to the left page, and (*right region*, *right ID*) to the right page.

Note that this procedure is recursive due to (S3.1).

The algorithm for inserting an index record (*point*, *record ID*) is as follows.

I1. If *root ID* is null, create a point page containing (*point*, *record ID*), set *root ID* to the ID of this page, and terminate.

I2. Otherwise, do an exact match query on *point*, which finds a point page that *point* should be added to if the K-D-B-tree structure is to be preserved. If *point* is already in the page, do something special (like generating an error, or modifying pointer fields in existing database records), and terminate.

I3. Add (*point*, *record ID*) to the point page. If the page does not overflow, terminate. Otherwise, let *page* be the point page.

I4. Let the ID of *page* be *old ID*. Pick a domain, $domain_i$, and an element x_i in this domain, such that *page* split along x_i will result in two pages that are not overfull (since the number of points or regions in *page* need only be decreased by one to avoid overflow, it is easy to see that this is always possible). Split *page* along x_i, giving left and right pages with IDs *left ID* and *right ID*.

I5. If *page* was the root page, go to (I6). Otherwise, let *page* be the parent page of *page* (this parent page was found during the exact match query step above). Replace (*region*, *old ID*) in *page* with (*left region*, *left ID*) and (*right region*, *right ID*), where *left region* and *right region* are obtained by splitting *region* along x_i. If this causes *page* to overflow, repeat from (I4); otherwise terminate.

I6. Create a new region page containing the regions

$$(domain_0 \times ... \times [-\infty_i, x_i) \times ... \times domain_{K-1}, left\ ID),$$

$$(domain_0 \times ... \times [x_i, \infty_i) \times ... \times domain_{K-1}, right\ ID),$$

and set *root ID* to its ID.

Variations of the above algorithm result from the way $domain_i$ and x_i are chosen in (I4). One way of choosing $domain_i$ is to do so cyclically, as follows. Store in each page a variable *splitting domain*, initialized to 0 in a root page when a new root page is created. When a page splits, an element of $domain_{splitting\ domain}$ is used, and the new pages have *splitting domain* set to (*splitting domain* + 1) MOD *K*. This method is analagous to the cyclic choice of domains in K-D-trees (see [Bentley 75]). Exceptions to this procedure, as well as other techniques for choosing $domain_i$ and x_i, are discussed in [Robinson 81].

Since the K-D-B-tree structure does not preclude empty point pages, and has no minimum storage utilization requirements, the basic deletion algorithm is very simple: find the index record (*point, record ID*) with an exact match query, and remove (*point, record ID*) from the point page.

Unless there are very few deletions, or by chance insertions take place that "fill in the holes" left by deletions, this basic deletion algorithm will be unacceptable due to the resulting low storage utilization. In B-tree algorithms, this problem is solved by what are here considered to be reorganization techniques. This reorganization takes place by redistributing index records among two or more adjacent sibling pages. The same type of reorganization can be performed on K-D-B-trees, providing the notion of adjacency can be generalized to more than one dimension.

One way to generalize adjacency is as follows: if the union of two or more regions is a region, the regions are said to be *joinable*. Using this property, an outline of the algorithm to "reorganize page *P*" is as follows (*P* could be an underfull point page produced by a deletion, or an underfull region page produced by previous reorganization).

1. Let *page* be the parent page of *P*, containing (*region, ID*), where *ID* refers to *P*.
2. Find ($region_1$, ID_1), ($region_2$, ID_2), ..., in *page* such that *region*, $region_1$, $region_2$, ..., are joinable (this is always possible -- in the worst case, this will be all the regions of *page*).
3. Catenate the pages with IDs *ID*, ID_1, ID_2, ..., and then repeatedly split this page and resulting pages until no page is overfull.
4. Replace (*region, ID*), ($region_1$, ID_1), ($region_2$, ID_2), ..., in *page* with the resulting new regions and page IDs.

5. If *page* is the root page, and it now contains only one pair (*region, ID*), delete *page* and set *root ID* to *ID*.

Another possible use of reorganization is during insertions, since step (S3) can leave empty or near-empty point pages. This should probably be done only at the point page level, since reorganization itself makes use of step (S3) when performed at higher levels. However, almost all pages are point pages (see the table below). Reorganization strategies for K-D-B-trees, and the performance of K-D-B-trees under reorganization, are subjects of current research.

A major difference between K-D-B-trees and B-trees with respect to insertions is step (S3), which forces pages at lower levels to split even though they are not overfull. An immediate question is how badly step (S3) affects performance, in terms of storage utilization and page accesses. Surprisingly, the performance of K-D-B-trees is quite good in spite of step (S3), even without reorganization. Table B shows the insertion characteristics of 2-D-B-trees and 3-D-B-trees, without reorganization, and with index records randomly generated, uniformly distributed in *K*-space. Details of these and other experiments appear in [Robinson 81].

Including various secondary indexes, as desired, an example of the resulting file structure is shown in Figure A2. A problem of future research is the optimal choice of multidimensional secondary indexes, given some kind of characterization of the "average" query.

K	PAGE SIZES[A]	SIZE	PAGES AT EACH LEVEL[B]	STORAGE UTILIZATION	PAGES ACCESSED/ INSERTION[C]
2	25, 42	20,000	1, 2, 40, 714	0.66	1.09, 3.36
		40,000	1; 4, 80, 1458	0.65	1.09, 4.00
		60,000	1, 7, 122, 2187	0.65	1.13, 4.00
		80,000	1, 9, 165, 2904	0.65	1.18, 4.00
		100,000	1, 12, 209, 3662	0.64	1.18, 4.00
3	36, 63	20,000	1, 20, 514	0.61	1.15, 2.92
		40,000	1, 2, 45, 1060	0.59	1.16, 3.30
		60,000	1, 2, 63, 1547	0.61	1.15, 4.01
		80,000	1, 4, 89, 2084	0.60	1.16, 4.01
		100,000	1, 4, 108, 2594	0.60	1.15, 4.00

[A] Page sizes = R, P, where R is maximum number of regions in a region page, P is maximum number of points in a point page.

[B] For example, "1, 20, 514" means 1 page at level 1 (root page), 20 pages at level 2, and 514 pages at level 3 (point pages).

[C] Pages accessed = W, R, where W is pages written, R is pages read, averaged over 20,000 insertions.

Table B. Growing K-D-B-Trees

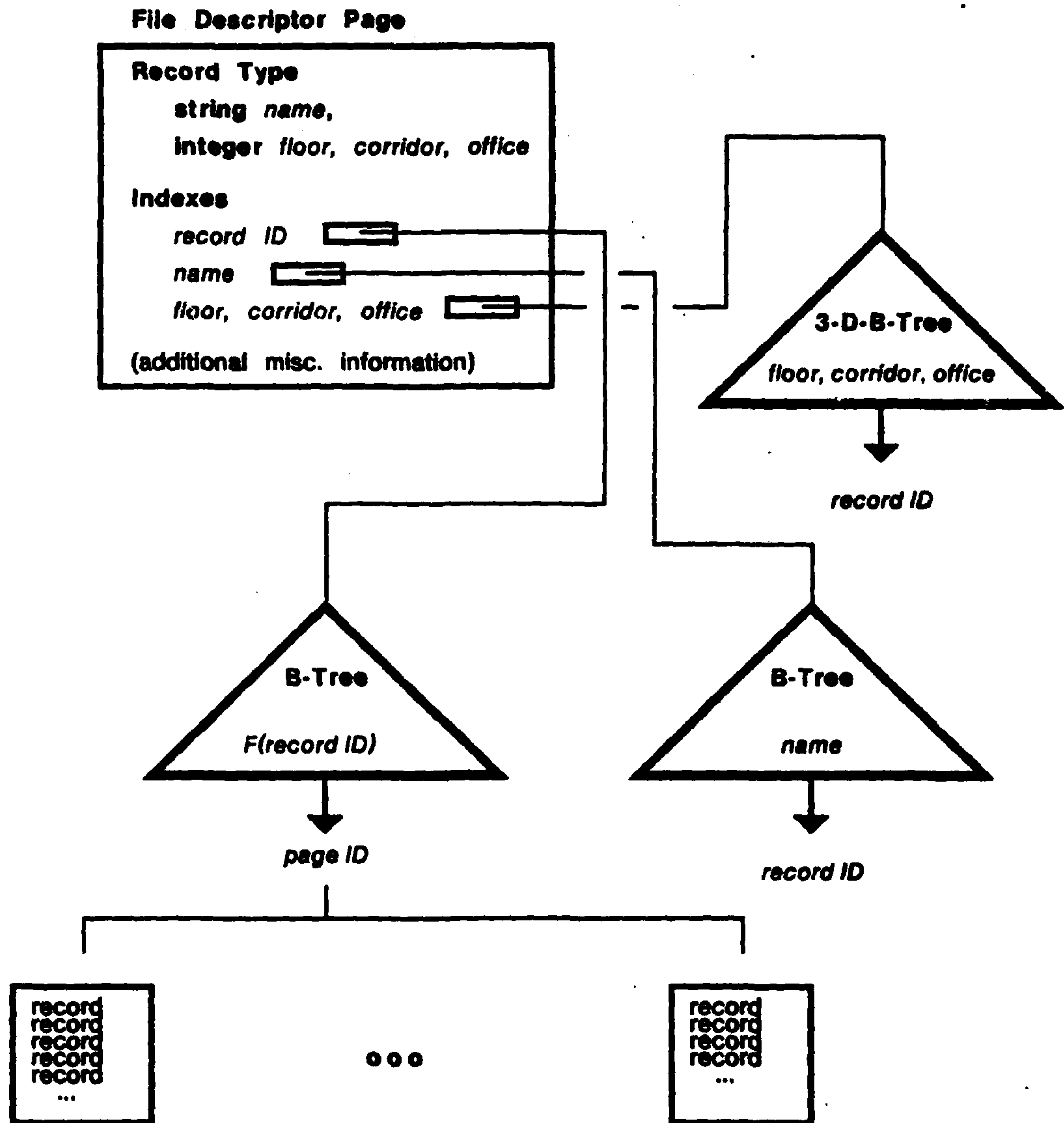


Figure A2. Example File Structure

By introducing record types that contain pointers to the descriptor pages of existing files, directories of files, etc., can be built, resulting in a hierarchical structure like that of the record manager used in the Cm* system, for example.

It might be thought that the introduction of new secondary indexes would require a reorganization of the file, that is, a large transaction, but even this can be avoided, assuming newly generated record IDs are monotonically increasing. With this assumption, one method for solving the problem is as follows.

1. Create the new secondary index, and index every record inserted to the file from this point on through this index, as usual. However, mark the index as not being up to date, so that queries will avoid its use.
2. Let *min ID* and *max ID* be the minimum and maximum record ID in the file at the time the new secondary index is created. Start a process that repeatedly finds the next record ID in the file, from *min ID* to *max ID*, and that indexes each corresponding record in the new secondary index, with each indexing operation implemented as a separate transaction.
3. When this process terminates, mark the index as being up to date.

The deletion of an existing secondary index does not cause significant concurrency problems, since once the pointer to the index is removed from the file descriptor page, all future transactions or queries cannot access the index. The process of deleting all pages of the secondary index can then be performed with transactions of a size chosen without regard to conflicts.

This concludes the outline of a possible record manager.

Appendix II. Concurrency Control Algorithms

In this appendix concurrency control algorithms are presented. In order to make these algorithms available to a wider audience, the Cm* CC subsystem was re-programmed in Pascal, following the original Bliss implementation fairly closely. The intent of the Pascal program below is to clearly show the logical nature of the algorithms, avoiding distracting complexity. Thus, sets of object IDs are declared as Pascal sets, even though the usual bit-vector representation would in practice be unacceptable (for example, 16-bit object IDs imply bit-vectors of length 64K bits to represent object ID sets); in practice, these sets would be represented as linked lists or stacks of object IDs (a linked list representation was used in the Cm* CC subsystem). Similarly, structures that are logically associatively accessed by object ID are declared as arrays indexed by object ID (these are RSet, RPSet, WSet, and WPSet below); in practice, only information for object IDs that were currently in use would be stored. In the Cm* CC subsystem, for example, information for any given object ID was accessed through a hash table, entries of which pointed to a linked list of object ID records with identical hash values. New object ID records were created as necessary, and when all transaction processor sets in a particular object ID record became empty, the record was deleted.

In order to test the Pascal implementation, terminal I/O was used in place of what had been message sending and receiving, and a procedure to print transaction records was added. All of this has been left intact, and as an aid to understanding the program, an example of the execution of the program follows the program listing.

Note: the Pascal variant used was IBM's Pascal/VS; the only occurring differences from Pascal as described in [Jensen and Wirth 74] are the "otherwise" construct in the procedure that reads an input line, and the terminal I/O initialization procedures.

The program and example follow.

```
{---------------------- CONCURRENCY CONTROL ----------------------}

program CC(input, output);

{ for illustrative purposes, transaction processors are named   }
{ '1', '2', ..., '9', and objects are named 'A', 'B', ..., 'Z'. }

const
   MinTPID = '1';  MaxTPID = '9';
   MinOID  = 'A';  MaxOID  = 'Z';

{---------------------------- TYPES ------------------------------}

type
   TPID = MinTPID .. MaxTPID;          { transaction processor ID }

   OID = MinOID .. MaxOID;             { object ID }

   TSet = set of TPID;                 { transaction set }

   OSet = set of OID;                  { object set }

   AccType = (none, R, RW, W, V);      { access types }

   DecType = (kill, wait, die, grant); { decisions }

   SubOpType = (rdrs, wrtrs, all);     { sub-options }

   StatType = (active, pending,        { status }
      validated, aborted, completed);

   MsgType = (Cbegin, Cread, Cwrite,      { messages }
      Cend, Cvalid, Cabort, Cpolicy, Clook, Cquit);

   TRec = record                          { transaction record }
      status: StatType;     { transaction status }
      access: AccType;      { type of most recent request }
      ObjID: OID;           { obj. ID for most recent access request }
      WaitCount: integer;   { number of trans.'s being waited on }
      PrecedeSet,           { set of trans.'s -> this trans. }
      VwaitSet,             { this trans. =>V set of trans.'s }
      CwaitSet,             { this trans. =>C set of trans.'s }
      ReferSet:             { set of trans.'s that refer in any }
         TSet;              {       fashion to this trans.      }
      ObjSet: OSet          { set of objects for which this trans. }
                            {        has requested access         }
      end;
```

```
{------------------------ GLOBAL VARIABLES ------------------------}

var
   TNC: integer;                    { transaction number counter }
   RunSet: TSet;                    { set of runnable transactions }
   CurrentTP: TPID;                 { current transaction processor ID }
   trans: array[TPID] of TRec;      { transaction records }
   RSet,                            { read sets }
   RPSet,                           { read postponed sets }
   WSet,                            { write sets }
   WPSet:                           { write postponed sets }
      array[OID] of TSet;


{------------------------ PM VARIABLES ------------------------}

   Roption,                         { read option }
   RWoption,                        { read/write option }
   Woption,                         { write option }
   Voption:                         { validation option }
      DecType;
   RWSubOption: SubOpType;          { read/write sub-option }
```

```
{-------------------------- INITIALIZATION -------------------------}

procedure init;
   var tp: TPID; id: OID;
   begin
   TNC := 1;
   RunSet := [];
   CurrentTP := MinTPID;
   for tp := MinTPID to MaxTPID do
      with trans[tp] do begin
         status := completed;
         access := none;
         ObjID := MinOID;
         WaitCount := 0;
         PrecedeSet := [];
         VwaitSet := [];
         CwaitSet := [];
         ReferSet := [];
         ObjSet := []
         end;
   for id := MinOID to MaxOID do begin
      RSet[id]  := [];
      RPSet[id] := [];
      WSet[id]  := [];
      WPSet[id] := []
      end
   end;
```

```
{--------------------- COMPUTE WAIT RELATION -----------------------}

{ determine if trans. on tp1 is waiting on trans. on tp2 }

function WaitingOn(tp1, tp2: TPID): boolean;
   label 1{return};
   var tp3: TPID;
   begin
   WaitingOn := false;
   with trans[tp2] do begin
      if tp1 in (VwaitSet + CwaitSet)
         then WaitingOn := true
         else for tp3 := MinTPID to MaxTPID do
            if tp3 in (VwaitSet + CwaitSet)
               then if WaitingOn(tp1, tp3)
                  then begin WaitingOn := true; goto 1{return} end
      end;
1:{return}
   end;


{--------------------------- SET POLICY ---------------------------}

procedure PMpolicy(Rop, RWop, Wop, Vop: DecType; RWSubOp: SubOpType);
   begin
   Roption := Rop;  RWoption := RWop;  Woption := Wop;  Voption := Vop;
   RWSubOption := RWSubOp;
   if Voption = grant then Voption := kill { grant is an illegal Vop }
   end;
```

```
{------------------------- POLICY MODULE ------------------------}

{ The following function decides how to handle a request from }
{ transaction processor tp.  ConflSet is the set of possibly  }
{ conflicting transactions, and WaitSet and AbortSet will be  }
{ set to the sets of transactions to wait on or abort.  The   }
{ result of the function is the decision for tp.              }

function PMdecide(tp: TPID; ConflSet: TSet;
                  var WaitSet, AbortSet: TSet): DecType;
  var
    tpi: TPID;  decision: DecType;
  begin
  with trans[tp] do begin
    case access of
      R:  decision := Roption;
      RW: decision := RWoption;
      W:  decision := Woption;
      V:  decision := Voption  end;
    if (access = RW) and ((RWoption = kill) or (RWoption = wait))
      then case RWSubOption of
        all:   ;
        rdrs:  ConflSet := ConflSet-(WSet[ObjID]+WPSet[ObjID]);
        wrtrs: ConflSet := ConflSet*(WSet[ObjID]+WPSet[ObjID])
        end;
    if decision = wait then begin { check if deadlock would result }
      if access ≠ V
        then { in order to allow queueing of requests }
          ConflSet := ConflSet -
            ((VwaitSet+CwaitSet) * RPSet[ObjID]);
      for tpi := MinTPID to MaxTPID do
        if tpi in ConflSet
          then if WaitingOn(tpi, tp)
            then decision := die { default victim is requestor }
      end;
    if ConflSet = [] then decision := grant;
    WaitSet := [];  AbortSet := [];
    case decision of
      grant: ;
      kill:  AbortSet := ConflSet;
      wait:  WaitSet := ConflSet;
      die:   ;   end
    end;
  PMdecide := decision
  end;
```

```
{------------------------ CC-TP COMMUNICATION ---------------------}

{ the following two procedures would in practice }
{ each send a message to a transaction processor }

procedure SendResult(tp: TPID; result: boolean);
   begin
   write('===> Message to TP', tp, ': ');
   if result then writeln('OK') else writeln('ABORT')
   end;

procedure SendTN(tp: TPID; tn: integer);
   begin writeln('===> Message to TP', tp, ': OK, TN = ', tn:1) end;


{ the following procedure would in practice        }
{ read a message from the CC input pipe.           }
{ Example terminal input:                          }
{   q       quit                                   }
{   b1      begin transaction on TP1               }
{   r2B     request for read access from TP2 for B }
{   pGGaGK  change policy to optimistic method     }

procedure GetMsg(var m: MsgType; var tp: TPID; var id: OID;
      var Rop, RWop, Wop, Vop: DecType; var RWSubOp: SubOpType);

   const MaxLnth = 6;
   var line: array[1..MaxLnth] of char; i: 1..MaxLnth; bad: boolean;

   procedure SetMsg(c: char);
      begin case c of
         'b': m:=Cbegin;  'r': m:=Cread;  'w': m:= Cwrite;
         'v': m:= Cvalid; 'e': m:=Cend;   'a': m:= Cabort;
         'p': m:= Cpolicy;  'l': m:= Clook;  'q': m:=Cquit;
           otherwise bad:=true  end end;
   procedure SetTPID(c: char);
      begin if (c ≥ MinTPID) and (c ≤ MaxTPID)
         then tp:=c else bad:=true end;
   procedure SetOID(c: char);
      begin if (c ≥ MinOID) and (c ≤ MaxOID)
         then id:=c else bad:=true end;
   procedure SetOp(c: char; var o: DecType);
      begin case c of
         'K': o:=kill;  'W': o:=wait;  'D': o:=die;  'G': o:= grant;
           otherwise bad := true  end end;
   procedure SetSubOp(c: char; var so: SubOpType);
      begin case c of
         'r': so:=rdrs;  'w': so:=wrtrs;  'a': so:=all;
           otherwise bad := true  end end;
```

```
begin

repeat
   writeln('(enter message)');
   read(line[1]);
   for i := 2 to MaxLnth do
      if eoln then line[i] := ' ' else read(line[i]);
   readln;
   bad := false;
   SetMsg(line[1]);
   if not bad then begin
      if m = Cpolicy
         then begin
            SetOp(line[2], Rop); SetOp(line[3], RWop);
            SetSubOp(line[4], RWSubOp); SetOp(line[5], Wop);
            SetOp(line[6], Vop) end
      else if m ≠ Cquit
         then begin
            SetTPID(line[2]);
            if (m=Cread) or (m=Cwrite) then SetOID(line[3]) end
      end;
   if bad then writeln('(bad input, try again)')
until not bad;

if m = Cpolicy
   then begin
      write('Policy change message: NEW POLICY =');
      for i := 2 to MaxLnth do write(' ', line[i]);
      writeln end
else if (m ≠ Cquit) and (m ≠ Clook)
   then begin
      write('Message from TP', tp, ': ');
      case m of
         Cbegin: writeln('BEGIN');  Cread: writeln('READ ',id);
         Cwrite: writeln('WRITE ',id);  Cvalid: writeln('VALIDATE');
         Cend: writeln('END');  Cabort: writeln('ABORT') end end
else if m = Cquit
   then writeln('(exit)')

end;
```

```
{------------------------ SCHEDULING ----------------------------}

{ have tp1 wait on tp2 }

procedure schedule(tp1, tp2: TPID);
   begin
   with trans[tp1] do begin
      if (access = R) or
       ((access = RW) and (tp2 in (WSet[ObjID]+WPSet[ObjID])))
         then
            trans[tp2].CwaitSet := trans[tp2].CwaitSet + [tp1]
         else
            trans[tp2].VwaitSet := trans[tp2].VwaitSet + [tp1];
      WaitCount := WaitCount+1;
      ReferSet := ReferSet + [tp2]
      end
   end;


{ postpone transaction }

procedure postpone(tp: TPID);
   begin
   with trans[tp] do begin
      if access = V
         then status := pending
         else begin
            if (access = R) or (access = RW)
               then RPSet[ObjID] := RPSet[ObjID] + [tp];
            if (access = W) or (access = RW)
               then WPSet[ObjID] := WPSet[ObjID] + [tp];
            ObjSet := ObjSet + [ObjID]
            end
      end
   end;
```

```
{------------------ MAINTAIN PRECEDE RELATION ---------------------}

procedure precede(tp1, tp2: TPID);
   begin
   with trans[tp1] do begin
      if (access = R) or (access = RW)
         then if tp2 in WSet[ObjID]
            then begin
               trans[tp2].PrecedeSet := trans[tp2].PrecedeSet + [tp1];
               ReferSet := ReferSet + [tp2]
               end;
      if (access = W) or (access = RW)
         then if tp2 in RSet[ObjID]
            then begin
               PrecedeSet := PrecedeSet + [tp2];
               trans[tp2].ReferSet := trans[tp2].ReferSet + [tp1]
               end
      end
   end;

{----------------------- GRANT A REQUEST --------------------------}

procedure GrantReq(tp: TPID);
   begin
   with trans[tp] do begin
      if (access = R) or (access = RW)
         then begin
            RPSet[ObjID] := RPSet[ObjID] - [tp];
            RSet[ObjID] := RSet[ObjID] + [tp]
            end;
      if (access = W) or (access = RW)
         then begin
            WPSet[ObjID] := WPSet[ObjID] - [tp];
            WSet[ObjID] := WSet[ObjID] + [tp]
            end;
      ObjSet := ObjSet + [ObjID]
      end;
   SendResult(tp, true)
   end;
```

```
{---------------------- ABORT A TRANSACTION -----------------------}

procedure abort(tp: TPID);
   var tpi: TPID; id: OID;
   begin
   with trans[tp] do begin
      status := aborted;
      for tpi := MinTPID to MaxTPID do
         begin
         if tpi in (VwaitSet + CwaitSet)
            then with trans[tpi] do begin
               WaitCount := WaitCount-1;
               if WaitCount = 0 then RunSet := RunSet + [tpi]
               end;
         if tpi in ReferSet then
            with trans[tpi] do begin
               PrecedeSet := PrecedeSet - [tp];
               VwaitSet := VwaitSet - [tp];
               CwaitSet := CwaitSet - [tp]
               end
         end;
      PrecedeSet := []; VwaitSet := []; CwaitSet := []; ReferSet := [];
      WaitCount := 0;
      for id := MinOID to MaxOID do
         if id in ObjSet then
            begin
            RSet[id] := RSet[id]-[tp];  RPSet[id] := RPSet[id]-[tp];
            WSet[id] := WSet[id]-[tp];  WPSet[id] := WPSet[id]-[tp]
            end;
      ObjSet := []
      end
   end;
```

```
{-------------------- VALIDATE A TRANSACTION ---------------------}

procedure validate(tp: TPID);
   var tpi: TPID; id: OID;
   begin
   with trans[tp] do begin
      status := validated;
      for tpi := MinTPID to MaxTPID do begin
         if tpi in VwaitSet
            then with trans[tpi] do begin
               WaitCount := WaitCount-1;
               if WaitCount = 0 then RunSet := RunSet + [tpi]
               end;
         if tpi in ReferSet
            then with trans[tpi] do begin
               PrecedeSet := PrecedeSet - [tp] end
         end;
      for id := MinOID to MaxOID do
         if id in ObjSet
            then RSet[id] := RSet[id] - [tp]
      end;
   SendTN(tp, TNC);
   TNC := TNC+1
   end;


{-------------------- COMPLETE A TRANSACTION ---------------------}

procedure complete(tp: TPID);
   var tpi: TPID; id: OID;
   begin
   with trans[tp] do begin
      status := completed;
      for tpi := MinTPID to MaxTPID do
         if tpi in CwaitSet
            then with trans[tpi] do begin
               WaitCount := WaitCount-1;
               if WaitCount = 0 then RunSet := RunSet + [tpi]
               end;
      PrecedeSet := []; VwaitSet := []; CwaitSet := []; ReferSet := [];
      WaitCount := 0;
      for id := MinOID to MaxOID do
         if id in ObjSet
            then WSet[id] := WSet[id] - [tp];
      ObjSet := []
      end
   end;
```

AD-A121 515 DESIGN OF CONCURRENCY CONTROLS FOR TRANSACTION PROCESSING SYSTEMS(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE J T ROBINSON 02 APR 82

2/2

UNCLASSIFIED N00014-76-C-0370 F/G 9/2 NL

END
DATE
FILMED
DTIC

```
{---------------------- PROCESS A REQUEST -------------------------}

procedure process(tp: TPID; ConflSet: TSet);
   var decision: DecType; WaitSet, AbortSet: TSet; tpi: TPID;
   begin

  { let the policy determine how to process this request }
   decision := PMdecide(tp, ConflSet, WaitSet, AbortSet);

  { check if the policy decided to abort a validated transaction,    }
  { or if the policy decided to grant the request even though there }
  { was a validated conflicting transaction                          }
   for tpi := MinTPID to MaxTPID do
      if trans[tpi].status = validated
         then if
            (tpi in AbortSet) or
            ( ((decision = grant) or (decision = kill)) and
              (tpi in ConflSet) )
            then begin decision := die; AbortSet := [] end;

  { abort transactions in AbortSet }
   for tpi := MinTPID to MaxTPID do
      if tpi in AbortSet then abort(tpi);

  { now process the request according to decision }
   case decision of

      die:  begin abort(tp); SendResult(tp, false) end;

      wait: begin
            for tpi := MinTPID to MaxTPID do
               if tpi in WaitSet then schedule(tp, tpi);
            postpone(tp)
            end;

      grant,
      kill: if trans[tp].access = V
               then validate(tp)
               else
                  begin
                  for tpi := MinTPID to MaxTPID do
                     if tpi in ConflSet then precede(tp, tpi);
                  GrantReq(tp)
                  end
      end
   end;
```

```
{---------------------- PROCESS CC REQUESTS -------------------------}

procedure CCprocess;
   var ConflSet: TSet;
   begin

  { repeatedly process requests while RunSet is non-empty }
   while RunSet ≠ [] do begin

      while not (CurrentTP in RunSet) do
         if (CurrentTP < MaxTPID)
            then CurrentTP := succ(CurrentTP)
            else CurrentTP := MinTPID;

      with trans[CurrentTP] do begin

        { determine set of possibly conflicting transactions }
         case access of
            R:  ConflSet := WSet[ObjID] + WPSet[ObjID];
            RW: ConflSet := RSet[ObjID] + RPSet[ObjID] +
                            WSet[ObjID] + WPSet[ObjID];
            W:  ConflSet := RSet[ObjID] + RPSet[ObjID];
            V:  ConflSet := PrecedeSet  end;

        { current transaction can't conflict with itself }
         ConflSet := ConflSet - [CurrentTP];

        { now process the request }
         if ConflSet = []
            then begin  { process here to save some time }
               if access = V
                  then validate(CurrentTP)
                  else GrantReq(CurrentTP)
               end
            else process(CurrentTP, ConflSet);

        { finished now with current transaction }
         RunSet := RunSet - [CurrentTP]

         end
      end
   end;
```

```
{------------------------- CC PROCEDURES --------------------------}

{ Suppose it was desired to use a timestamp-based policy.  Then   }
{ transaction records could be extended to include a time field,  }
{ and "trans[tp].time := <current time>" could be added to the    }
{ following procedure.  Similar modifications could be made for   }
{ any other policy based on information available at Cbegins.     }

procedure CCbegin(tp: TPID);
   begin
   trans[tp].status := active
   end;

procedure CCvalid(tp: TPID);
   begin
   with trans[tp] do begin
      if status = aborted
         then SendResult(tp, false)
         else begin
            access := V;
            RunSet := RunSet + [tp];
            CCprocess
            end
      end
   end;

procedure CCend(tp: TPID);
   begin
   complete(tp);
   CCprocess
   end;

procedure CCabort(tp: TPID);
   begin
   abort(tp);
   CCprocess
   end;
```

```
procedure CCread(tp: TPID; id: OID);
   begin
   with trans[tp] do begin
      if status = aborted
         then SendResult(tp, false)
      else if tp in RSet[id]
         then SendResult(tp, true)
      else
         begin
         access := R;
         ObjID := id;
         RunSet := RunSet + [tp];
         CCprocess
         end
      end
   end;

procedure CCwrite(tp: TPID; id: OID);
   begin
   with trans[tp] do begin
      if status = aborted
         then SendResult(tp, false)
      else if tp in WSet[id]
         then SendResult(tp, true)
      else
         begin
         if tp in RSet[id]
            then access := W
            else access := RW;
         ObjID := ID;
         RunSet := RunSet + [tp];
         CCprocess
         end
      end
   end;
```

```
{-------------------- LOOK AT TRANSACTION RECORD --------------------}

procedure look(tp: TPID);
   var id: OID;

   procedure WriteTSet(s: TSet);
      var tpi: TPID;
      begin
      for tpi := MinTPID to MaxTPID do
         if tpi in s then write(tpi)
      end;

   begin
   Writeln('TRANSACTION ',tp);
   with trans[tp] do begin

      write('   Status: ');
      case status of active: writeln('active');
       pending: writeln('pending'); validated: writeln('validated');
       aborted: writeln('aborted'); completed: writeln('completed') end;

      write('   Access: ');
      case access of none: writeln('none'); R: writeln('R');
       RW: writeln('RW'); W: writeln('W'); V: writeln('V') end;

      writeln('   Object ID: ', ObjID);

      writeln('   WaitCount: ', WaitCount:1);

      write('   PrecedeSet: '); WriteTSet(PrecedeSet); writeln;
      write('   VwaitSet: '); WriteTSet(VwaitSet); writeln;
      write('   CwaitSet: '); WriteTSet(CwaitSet); writeln;
      write('   ReferSet: '); WriteTSet(ReferSet); writeln;

      write('   ObjSet: ');
      for id := MinOID to MaxOID do
         if id in ObjSet then write(id);
      writeln

      end
   end;
```

```
{--------------------------- CC DRIVER ---------------------------}

procedure driver;
   var m: MsgType; tp: TPID; id: OID;
      Rop, RWop, Wop, Vop: DecType; RWSubOp: SubOpType;
   begin
   repeat
      GetMsg(m, tp, id, Rop, RWop, Wop, Vop, RWSubOp);
      case m of
         Cbegin:  CCbegin(tp);
         Cread:   CCread(tp, id);
         Cwrite:  CCwrite(tp, id);
         Cvalid:  CCvalid(tp);
         Cend:    CCend(tp);
         Cabort:  CCabort(tp);
         Cpolicy: PMpolicy(Rop, RWop, Wop, Vop, RWSubOp);
         Clock:   lock(tp);
         Cquit:   ;
         end
   until m = Cquit
   end;


{------------------------- PROGRAM BODY ---------------------------}

begin
   termin(input);  termout(output);        { initialize terminal i/o }
   init;
   PMpolicy(wait,wait,wait,wait,all);      { default to locking }
   writeln('(default locking policy in effect)');
   driver
end .
```

Example of Execution of CC Program

```
(default locking policy in effect)
(enter message)
b1
Message from TP1: BEGIN
(enter message)
w1A
Message from TP1: WRITE A
===> Message to TP1: OK
(enter message)
b2
Message from TP2: BEGIN
(enter message)
r2A
Message from TP2: READ A
(enter message)
b3
Message from TP3: BEGIN
(enter message)
r3A
Message from TP3: READ A
(enter message)
b4
Message from TP4: BEGIN
(enter message)
w4A
Message from TP4: WRITE A
(enter message)
l1
TRANSACTION 1
   Status: active
   Access: RW
   Object ID: A
   WaitCount: 0
   PrecedeSet:
   WwaitSet:
   CwaitSet: 234
   ReferSet:
   ObjSet: A
(enter message)
l2
TRANSACTION 2
   Status: active
   Access: R
   Object ID: A
   WaitCount: 1
   PrecedeSet:
   WwaitSet: 4
   CwaitSet:
   ReferSet: 1
```

```
   ObjSet: A
(enter message)
l4
TRANSACTION 4
   Status: active
   Access: RW
   Object ID: A
   WaitCount: 3
   PrecedeSet:
   VwaitSet:
   CwaitSet:
   ReferSet: 123
   ObjSet: A
(enter message)
v1
Message from TP1: VALIDATE
===> Message to TP1: OK, TN = 1
(enter message)
e1
Message from TP1: END
===> Message to TP2: OK
===> Message to TP3: OK
(enter message)
v2
Message from TP2: VALIDATE
===> Message to TP2: OK, TN = 2
(enter message)
v3
Message from TP3: VALIDATE
===> Message to TP3: OK, TN = 3
===> Message to TP4: OK
(enter message)
v4
Message from TP4: VALIDATE
===> Message to TP4: OK, TN = 4
(enter message)
e2
Message from TP2: END
(enter message)
e3
Message from TP3: END
(enter message)
e4
Message from TP4: END
(enter message)
b1
Message from TP1: BEGIN
(enter message)
b2
Message from TP2: BEGIN
(enter message)
b3
```

```
Message from TP3: BEGIN
(enter message)
r1A
Message from TP1: READ A
===> Message to TP1: OK
(enter message)
r2B
Message from TP2: READ B
===> Message to TP2: OK
(enter message)
r3C
Message from TP3: READ C
===> Message to TP3: OK
(enter message)
w1B
Message from TP1: WRITE B
(enter message)
w2C
Message from TP2: WRITE C
(enter message)
w3A
Message from TP3: WRITE A
===> Message to TP3: ABORT
===> Message to TP2: OK
(enter message)
v2
Message from TP2: VALIDATE
===> Message to TP2: OK, TN = 5
===> Message to TP1: OK
(enter message)
v1
Message from TP1: VALIDATE
===> Message to TP1: OK, TN = 6
(enter message)
e1
Message from TP1: END
(enter message)
e2
Message from TP2: END
(enter message)
pGGaGK
Policy change message: NEW POLICY = G G a G K
(enter message)
b1
Message from TP1: BEGIN
(enter message)
b2
Message from TP2: BEGIN
(enter message)
b3
Message from TP3: BEGIN
(enter message)
```

```
r1A
Message from TP1: READ A
===> Message to TP1: OK
(enter message)
r2B
Message from TP2: READ B
===> Message to TP2: OK
(enter message)
r3C
Message from TP3: READ C
===> Message to TP3: OK
(enter message)
w1B
Message from TP1: WRITE B
===> Message to TP1: OK
(enter message)
w2C
Message from TP2: WRITE C
===> Message to TP2: OK
(enter message)
w3A
Message from TP3: WRITE A
===> Message to TP3: OK
(enter message)
l1
TRANSACTION 1
   Status: active
   Access: RW
   Object ID: B
   WaitCount: 0
   PrecedeSet: 2
   WwaitSet:
   CwaitSet:
   DeferSet: 3
   ObjSet: AB
(enter message)
l2
TRANSACTION 2
   Status: active
   Access: RW
   Object ID: C
   WaitCount: 0
   PrecedeSet: 3
   WwaitSet:
   CwaitSet:
   DeferSet: 1
   ObjSet: BC
(enter message)
l3
TRANSACTION 3
   Status: active
   Access: RW
```

```
   Object ID: A
   WaitCount: 0
   PrecedeSet: 1
   VwaitSet:
   CwaitSet:
   ReferSet: 2
   ObjSet: AC
(enter message)
v2
Message from TP2: VALIDATE
===> Message to TP2: OK, TN = 7
(enter message)
v3
Message from TP3: VALIDATE
===> Message to TP3: ABORT
(enter message)
v1
Message from TP1: VALIDATE
===> Message to TP1: OK, TN = 8
(enter message)
e1
Message from TP1: END
(enter message)
e2
Message from TP2: END
(enter message)
q
(exit)
```

References

[Andler 79] Andler, S. Synchronization primitives and the verification of concurrent programs. In *Operating Systems*, Ed. D. Lanciaux, North Holland, New York, 1979, 67-99.

[Bayer and McCreight 72] Bayer, R. and McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (1972), 173-189.

[Bayer and Schkolnick 77] Bayer, R. and Schkolnick, M. Concurrency of operations on B-trees. *Acta Informatica* 9, 1 (1977), 1-21.

[Bentley 75] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Comm. ACM* 18, 9 (1975), 509-517.

[Bentley 79] Bentley, J. L. Multidimensional binary search trees in database applications. *IEEE Trans. Software Eng. SE-5*, 4 (1979), 333-340.

[Bentley and Friedman 79] Bentley, J. L. and Friedman, J. H. Data structures for range searching. *Computing Surveys* 11, 4 (1979), 397-409.

[Bernstein and Goodman 81] Bernstein, P. A. and Goodman, N. Concurrency control in distributed database systems. Computing Surveys 13, 2 (1981), 185-221.

[Comer 79] Comer, D. The ubiquitous B-tree. *Computing Surveys* 11, 2 (1979), 121-138.

[Date 77] Date, C. J. *An Introduction to Database Systems* (2nd Edition). Addison Wesley, Reading Massachusetts, 1977.

[Ellis 78] Ellis, C. S. Concurrent search and insertion in 2-3 trees. *Acta Informatica* 14, 1 (1980), 63-86.

[Eswaran et al 76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notions of consistency and predicate locks in a database system. *Comm. ACM* 19, 11 (November 1976), 624-633.

[Everhart 79] Everhart, C. F. The distinction between functional correctness and policy in parallel programs. Thesis proposal, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh Penn., May 1979.

[illegible] [illegible], W. Robust concurrency control for a [illegible] information system. Ph.D. Thesis (Report No. [illegible]), Laboratory for Computer Science, [illegible] Institute of Technology, December 1978.

[illegible] [illegible] J. K., [illegible], D. A., and Sindhu, P. S. Medusa: an experiment in distributed operating system structure. Comm. ACM 23, 2 (February 1980), 92-105.

[illegible] [illegible], C. H., Bernstein, P. A. and Rothnie, J. B. [illegible] problems related to database concurrency control. In *Conf. on Theoretical Computer Science*, Univ. Waterloo, 1977, [illegible].

[illegible] Papadimitriou, C. H. Serializability of concurrent updates. *Journal of the ACM* 26, 4 (October 1979), 631-653.

[illegible] Reed, D. P. Naming and synchronization in a decentralized computer system. Ph. D. Thesis (Report No. MIT/LCS/TR-[illegible]), Laboratory for Computer Science, Massachusetts Institute of Technology, September 1978.

[illegible] Ritchie, D. M. and Thompson, K. The Unix time-sharing system. Comm. *ACM* 17, 7 (July 1974), 365-375.

[illegible] Robinson, J. T. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM-SIGMOD 1981 Int. Conf. Management of Data*, May 1981, [illegible]

[illegible] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. II. System level concurrency control for distributed database systems. *ACM Trans. Database Systems* 3, 2 (June 1978), [illegible].

[illegible] Samadi, B. B-trees in a system with multiple users. *Information Processing Letters* 5, 4 (October 1976), 107-[illegible].

[illegible 81] Stearns, R. E. and Rosenkrantz, D. J. Distributed database concurrency controls using before-values. *Proc. ACM-SIGMOD 1981 Int. Conf. Management of Data*, May 1981, [illegible].

[Stearns et al 76] Stearns, R. E., Lewis, P. M. II and Rosenkrantz, D. J. Concurrency control for database systems. In *Proc. Seventh Symp. Foundations of Computer Science*, 1976, 19-32.

[Swan et al 77] Swan, R. J., Fuller, S. H., and Siewiorek, D. P. Cm* -- a modular multi-microprocessor. In *Proc. AFIPS 1977 NCC* (vol. 46), AFIPS Press, Arlington Virginia, 1977, 637-644.

[Thomas 79] Thomas, R. H. A majority consensus approach to concurrency control. *ACM Trans. Database Systems 4*, 2 (June 1979), 180-209.

[Ullman 80] Ullman, J. D. *Principles of Database Systems*. Computer Science Press, Potomac Maryland, 1980.

[Wiederhold 77] Wiederhold, G. *Database Design*. McGraw-Hill, New York, 1977.

[Wulf et al 71] Wulf, W. A., Russell, D. B., and Habermann, A. N. BLISS: a language for systems programming. *Comm. ACM 14*, 12 (December 1971), 780-790.

[Wulf et al 74] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: the kernel of a multiprocessor operating system. *Comm. ACM 17*, 6 (June 1974), 337-345.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CMU-CS-82-114	2. GOVT ACCESSION NO. ONR	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design of Concurrency Controls for Transaction Processing Systems		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John T. Robinson		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE April 2, 1982
		13. NUMBER OF PAGES 111
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-82-114	2. GOVT ACCESSION NO. ONR	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design of Concurrency Controls for Transaction Processing Systems		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John T. Robinson		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE April 2, 1982
		13. NUMBER OF PAGES 111
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)