Applications of Static Analysis to Concurrency Control and Recovery in Objectbase Systems

by

Peter C.J. Graham

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba, Canada, 1994

ISBN 0-315-99046-5

Name _____

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science — SUBJECT TERM

0984 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

APPLICATIONS OF STATIC ANALYSIS TO CONCURRENCY CONTROL AND RECOVERY IN OBJECTBASE SYSTEMS

BY

PETER C.J. GRAHAM

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

This dissertation explores the use of statically derived information to improve concurrency control and recovery in objectbase systems. It surveys the relevant background material including existing objectbase systems and conventional concurrency control and recovery. The problem of providing concurrency control and recovery in objectbases supporting nested transactions is analyzed and types of static information which are useful to concurrency control and recovery are determined. Algorithms are proposed to derive the needed static information.

The problem of concurrency control in objectbases is decomposed into two simpler problems: intra-transaction concurency control and inter-transaction concurrency control and algorithms are developed for each. A novel concurrency control algorithm combining the algorithms for intra- and inter-transaction concurrency control which specifies serialization orders *à priori* is presented. The algorithm selects appropriate serialization orders based on its knowledge of intra- and inter-object relationships. This knowledge is based on static information derived at compile and object-instantiation time.

Static information derived at compile time is also used to decrease the overhead of ensuring recoverability during transaction processing. By exploiting knowledge of object method behaviours, both the amount of information which must be logged and the number of cache flushes which must be performed are significantly decreased. This decreases the number of corresponding I/O operations which are the primary source of overhead in recovery processing.

Numerous other results for both single and multi-version objectbases are also presented. The complexity, effectiveness, and correctness of the algorithms is discussed and conclusions are drawn on their applicability.

Acknowledgements

I am indebted to my advisor Ken Barker who using determination, ingenuity, motivation, and a strong foot managed to get me through my PhD. Thanks go to my wife Pat and daughter Lisa for their patience during the research for, and writing of, this dissertation and also to my parents who, although my father never lived to see it, I am sure were the only ones who never doubted I would finish. Finally, thanks to the members of the advanced database research group for their insights into, and frequent attacks on, my work.

> "I'd rather debate a question without settling it than settle a question without debating it." – Joseph Joubert (1754-1824)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Almost without exception, current database systems [Wie83, Dat86, EN89] attempt to accomplish concurrency control (CC) at runtime without the benefit of any semantic information [GM83, Wei88, FO89]. This means that the scheduler is reduced to *dynamically* analyzing primitive read and write operations as they occur [BK91, MM93] and making scheduling decisions based only on that information. This approach often leads to sub-optimal execution schedules and/or unreasonably expensive scheduling algorithms. Compile time analysis similar to that being used in optimizing and parallelizing compilers [ASU86, PW86, WB87] can be used to provide useful information concerning the available concurrency *within* transactions on an objectbase. Additional analysis performed when complex objects are created can provide information useful in enhancing concurrency *between* transactions. These analyses are referred to as *static analysis* and the information they derive is referred to as *static information*.

The thesis of this dissertation is; "If static information is used by the transaction scheduler(s) then cheaper and less restrictive concurrency control and recovery are achieved in objectbase systems."

1.1 The Problem

Existing concurrency control and recovery mechanisms were designed to be used in traditional database systems. The characteristics of such systems are very different from those of advanced database systems (including objectbases) where conventional mechanisms are inadequate. New concurrency control and recovery techniques must be found that are practical and effective if advanced database systems are to enjoy the same success as conventional ones.

One approach to improving concurrency control is to use static analysis to increase the information available to the runtime scheduler so it can make better scheduling decisions. The goals of applying static information are:

- To provide less costly means of concurrency control than conventional schemes.
- To provide less restrictive concurrency control.
- To do this *efficiently* by using static information in runtime schedulers.

It is important that the complexity of concurrency control be minimal for the user regardless of which mechanisms are used. The user should not have to specify any concurrency information. Instead, the systems software should derive the needed information. This information may be derived by two system components; the compiler and the runtime system. Current systems make use of information derived by the runtime system exclusively and fail to exploit all possible concurrency. To improve concurrency control additional, useful scheduling information must be derived. In objectbases, part of this information comes from the compiler and part from the runtime system.

Static transaction analysis will provide inferior performance compared to dynamic (run time) analysis if three assumptions are true. The first assumption is that the information which can be derived by static analysis is very limited. With advances in optimizing compiler techniques, this is no longer true. The second assumption is that the

analysis is to be done for transactions in conventional database systems. Distinct advantages can be achieved if the area of application is restricted to objectbases. The third assumption is that static analysis is used in isolation. Static information is extremely pessimistic since static analysis is forced to assume that conditional references *do* occur and to reflect this in the information produced. Thus, scheduling using static information *alone* will result in executions which are inferior to those produced using more accurate dynamic information. The benefits of static information are best realized by using it to optimize scheduling based on dynamic information.

Static analysis provides inexact but extensive information while dynamic analysis provides exact but limited information. Combining the two offers the potential for significantly improved scheduling performance.

Static analysis may also be applied to improve the efficiency of recovery in objectbases. The availability of statically derived semantic information can be exploited to decrease the overhead of logging. This may be done both by decreasing the number of log records written and by decreasing the amount of information in each log record.

Incorporating static information into scheduling and recovery is the subject of this dissertation. The specific problems associated with deriving and applying static information in objectbases are centered around three characteristics of the objectbase; the type of object sharing permitted, whether or not support for object versioning is provided, and the transaction nesting model (if any). These characteristics, the associated problems, and a related taxonomy of objectbases are discussed in chapter 3.

1.2 Motivation

Improvements in concurrency control to allow more concurrency and/or to decrease scheduling costs and other overheads are much sought-after developments in any database environment. Similarly, decreasing the overhead associated with ensuring recoverability is

highly desirable. Objectbase systems [ABD+89, GK88, Kim90, KGBW90, BM91, HPC93] are rapidly being accepted as the next logical step in database development and it is therefore reasonable to focus research efforts in concurrency control and recovery on objectbase systems.

1.2.1 The Need for Improved Concurrency Control and Recovery

Historically, transactions in database systems consisted of only a few data references and were correspondingly short-lived. Given these characteristics, conventional concurrency control and recovery techniques are appropriate.

Simple locking schemes, the basis of conventional concurrency control, provide adequate capabilities for short, simple transactions. As long as locks are held for a short time and are relatively cheap to obtain they are an appealing form of concurrency control. Originally, both of these conditions were met – short transactions meant locks were quickly freed and locking was normally done at the page level so there were comparatively few locks to obtain, thereby compensating for the relatively high cost of locking [Moh90]. The only long-lived transactions were typically large batch updates which were run over night to avoid locking out other database users.

Existing recovery techniques based on logging transaction operations are also adequate for conventional transactions. Simple, read and write operations can be logged effectively with limited overhead.

Current, traditional databases are growing and interactive access to them is increasing. Furthermore, they are also being distributed on a significant scale [BG81]. These factors make existing locking and logging schemes far less desirable. The large size of some databases means that overnight updates are often no longer possible. Interactivity further restricts the duration of time available to log operations and hold locks. Finally, distribution has introduced previously unseen delays due to network congestion, link or node failures, or simply the overhead of the distribution itself. These delays are

unacceptable when they result in undue increases to transaction execution times.

The greatest motivation to change concurrency control and recovery methods is evolving database technology and the needs of the applications using that technology. A typical application domain for *advanced database systems* is computer assisted design (CAD). This includes document preparation, software engineering, VLSI design and other, similar systems all of which involve multiple users interactively cooperating to do design. Other examples of advanced applications include multimedia systems and geographical information systems (GISs).

Computer assisted design applications [Gre91, BK91, GSW92, NRZ92, MRKN92, HHZ+92] share several common traits which require advanced concurrency control and recovery capabilities. These include:

- complex data structures
- distribution
- long lived transactions
- interactivity
- increased sharing
- cooperation between transactions

Support for complex data structures is required to attain data representation capabilities beyond those of the popular relational database model and the even more restrictive hierarchical and network database models. Many advanced applications manipulate data that has an inherent structure (often representing the structure of the real-world objects being described). A natural way to support complex structure is using abstract data types (ADTs) (such as those provided in Ada [U.S80], Alphard [Sha81], or CLU [LAB+81]), or more recently objects [RSC91, CY91a, CY91b, BEMP92]. The use of objects and the

nested structure that arises from declaring objects within other objects create challenging problems in concurrency control and recovery.

The provision of both complex data objects and distribution introduces the possibility of long-lived transactions. Complex data objects typically support more complex and hence long-lived operations. As described previously, distribution introduces the possibility of delays in transaction processing. Regardless of why transactions are long-lived, concurrency control must offer techniques which tolerate the associated long locking latency or, in the case of systems which are also highly interactive, avoid it.

Complex objects tend to be shared in complex ways. This means that simply analyzing read and write operations performed by those transactions is an ineffective way of managing the sharing and associated concurrency. Furthermore, with many users, sharing tends to increase and the possibility of multiple users sharing an object for concurrent update also increases[1]. Complex objects also introduce complex inter-object relationships which complicate the process of recovery.

A final motivation for studying and changing concurrency control is the development of parallel processing systems. The perceived need for cooperation between active transactions suggests that the programmers of such cooperative transactions [NRZ92] will have to deal with inter-transaction synchronization explicitly since it appears difficult to support such activity without user intervention. This is also true if *parallel* transactions are to be specified. Using this approach, database programmers are not sheltered from synchronization by using techniques based on serializability [Pap86]. Instead, a reasonable set of primitives for explicitly specifying how concurrent transactions should interact is provided. This is already being done by the programming language and parallel systems community in their efforts to develop effective parallel programming languages [Per87, GM88, Bur88]. If similar database programming languages can be developed then explicit synchronization may be specified relatively easily by the pro-

[1]This latter problem requires some advanced form of cooperation (i.e., synchronization) between the concurrent transactions.

grammer. Initial research into parallel database [HDV88] and query languages [CBW92] has already begun.

Explicit specification of synchronization is an appropriate long-term approach but other concurrency control mechanisms are needed to bridge the gap between what is currently available and what will eventually be made available. Static analysis of programs is being used successfully to permit the automatic "parallelization" of existing serial code. Similar *static analysis* can be applied in databases to achieve related results with serial transactions.

1.2.2 Why Static Analysis?

Researchers have started to address the concurrency control problems associated with advanced database systems based on persistent object stores and other data models. Many of the proposed solutions are radically different from conventional schemes and this makes them unfamiliar and difficult for existing database programmers to use. The use of static analysis represents minimal, if any, *programmer visible* changes to existing systems. Static analysis may be used to *transparently* improve the performance of concurrency control schemes with which programmers are already familiar. Static analysis also offers a base upon which new concurrency control schemes may be developed.

Unlike run-time scheduling methods, the costs associated with static analysis are incurred only once, prior to data access. This has two consequences. First, any overhead associated with scheduling that is handled by static analysis is effectively removed. Secondly, a comparatively large amount of time may be spent doing static analysis while the time spent doing runtime analysis must be strictly limited. This means that more extensive information can be derived using static analysis.

The application of static analysis to improve the efficiency of recovery is also transparent. As with concurrency control, it is undesirable to explicitly involve the transaction

programmer in the management of recovery. Using static analysis avoids such involvement.

Static analysis has thrived in the areas of programming languages and compilers. Many of the advances made, particularly in parallelizing compilers, are directly or indirectly applicable to transaction management. Other, related forms of static analysis may provide useful information which is unique to database systems.

1.3 Organization

The rest of this dissertation is organized as follows: Chapter 2 presents related work and background material. Chapter 3 details the problem of concurrency control in object-bases. In Chapter 4 the derivation and representation of the needed static information is described and in Chapter 5 the application of the information to transaction scheduling is presented. Chapter 6 addresses the problem of decreasing the overhead of recovery using static information. The dissertation concludes with a summary of the results presented and a discussion of future work in Chapter 7.

Chapter 2

Related Work

This chapter summarizes work related to that presented in this dissertation including; conventional concurrency control, existing objectbase systems, advanced concurrency control, and conventional recovery techniques..

2.1 Conventional Concurrency Control

A good understanding of existing techniques for concurrency control and their traditional uses is essential to understanding their limitations. The goal of this section is to survey concurrency control methods such as those discussed by Bernstein, *et al.* [BHG87]. It does not attempt to address the multitude of variations on each concurrency control "theme" that have appeared in the literature. When such a variation is relevant to the dissertation, it will be described as needed.

2.1.1 The Need for Concurrency Control

Concurrency control is not required in a system where the programmer is responsible for specifying program behaviours and their concurrency patterns. The programmer knows

transaction 1	transaction 2
temp:=read(X)	
	temp:=read(X)
	temp:=temp+100
	write(X,temp)
temp:=temp+200	
write(X,temp)	

The value of X after the concurrent execution of both transactions is the original value plus 200, not the the original value plus 300 as expected. The update made by transaction 2 has been "lost".

Figure 2.1: The Lost Update Problem

what concurrency will occur and can design programs to avoid anomalies (as is done in concurrent programming [GM88]). Unfortunately in database systems this cannot be done. Transactions are normally programmed by separate individuals who are unaware of one another's work. Furthermore, the set of transactions which is active at any given time changes dynamically. It thus falls on the database system itself to manage concurrency. An advantage of this *system-based* approach to concurrency control is that programming is easy since concurrency is transparent. A disadvantage is that the concurrency achieved is often sub-optimal.

Unconstrained concurrency in the execution of database transactions can lead to unexpected results not anticipated by the transaction programmer. Classic examples of concurrency anomalies include the "lost update" problem illustrated in Figure 2.1 and the "inconsistent retrieval" problem.

2.1.2 Forms of Concurrency Control

Concurrency control is commonly divided into two broad categories; optimistic and pessimistic.

Using optimistic concurrency control, transactions are allowed to execute concurrently without checking to ensure that the resulting execution will be valid until transaction completion. This allows incorrect executions to occur and thus places increased importance on recovery since the effects of invalid executions must be "undone" after they are detected. Incorrect transactions must be "rolled back" and the work of such transactions is lost.

Pessimistic concurrency control verifies that executions will be correct before they are allowed to occur. This has the effect of decreasing the complexity of recovery but in practice is achieved only at the price of increased overhead and significantly decreased *legitimate* concurrency [Moh90].

2.1.3 Serializability

The ultimate goal of concurrency control is to ensure that concurrent execution of transactions does not compromise the integrity of the database.

Definition 2.1 A *correctness criterion* is a specification of certain conditions which guarantee that database integrity is maintained. ■

By far the most widely used correctness criterion is serializability [Pap86] which ensures that concurrent executions will be equivalent to some serial execution of the transactions. Assuming that each serial transaction execution preserves database integrity, then it is guaranteed that any serializable concurrent execution also does.

Definition 2.2 A concurrent execution of the steps of a set of transactions is said to be *serializable* if it is equivalent to some serial execution of those transactions. ■

Serializability has the highly desirable characteristic of not requiring the database programmer to explicitly deal with concurrency. Simply writing correct serial transactions is sufficient because correct concurrent executions are assured. Unfortunately, practical implementations of serializability do not allow all valid concurrent executions of a set of transactions.

Various forms of serializability have been discussed in the literature. Most systems use *conflict serializability* (CSR). Since the only situation where concurrency can cause problems is when conflicting operations occur concurrently, conflict serializability serializes transactions only with respect to the conflicting operations, thereby allowing non-conflicting operations to occur concurrently.

A useful tool in discussing conflict serializability is the *serialization graph*.

Definition 2.3 A *serialization graph* is a directed acyclic graph $SG = (V, E)$ with a vertex $v \in V$ for each active transaction and a directed edge $e \in E$ from T_i to T_j iff an operation of T_i precedes and conflicts[1] with an operation of T_j. ■

The serialization graph records conflict orderings in transaction executions. Serialization is violated if two or more transactions execute operations resulting in conflict orderings which cause the corresponding serialization graph to become cyclic. This leads immediately to a test for serializability based on testing for cycles in the serialization graph.

Another form of serializability is known as *view serializability* (VSR). In this case, equivalence to a serial execution is achieved by guaranteeing that each transaction sees the same data (i.e., view) as it would in some serial execution.

The third major type of serializability is *final state serializability* (FSR) where transactions are allowed to execute concurrently in any way as long as the final state of the

[1] Two operations conflict when they both access the same data item and at least one operation is a write.

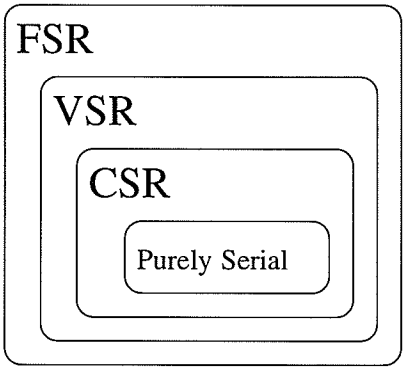

Figure 2.2: Executions Accepted by Conflict, View, and Final State Serializability

database is equivalent to that which would be produced by some serial execution of the transactions.

The relationship between the executions accepted by these three forms of serializability is illustrated in Figure 2.2. Conflict serializability allows fewer concurrent executions than view serializability which in turn allows fewer executions than final state serializability. Unfortunately, conflict serializability is the only form of the three which permits efficient implementation. The others are known to require algorithms which are NP-complete [Pap86].

2.1.4 Recovery

A key feature of database management systems is their ability to recover from system and/or transaction failures. The issues of concurrency control and recovery are closely related and the needs of recovery often affect decisions made about concurrency control and vice versa. Many classes of recoverable transactions have been defined. Only three classes are considered in this dissertation.

Transactions are said to be *recoverable* (RC) if they do not commit until all other transactions from which they read have also committed. This avoids the situation where transaction T_i reads a value produced by transaction T_j and then commits before T_j

does. If T_j subsequently aborts, then T_i has updated the database based on inconsistent information supplied by T_j.

A stronger recoverability condition results when a transaction can only read values that have been produced by a transaction which has *already* committed. Such transactions are said to *avoid cascading aborts* (ACA) since one transaction's abort will not cause another to abort.

Finally, a transaction is said to be *strict* (ST) if it neither reads nor rewrites a data value until a transaction that previously wrote the value completes. Most database systems enforce strictness.

2.1.5 Implementation of Concurrency Control

The two most common concurrency control algorithms in conventional database systems are two phase locking and timestamp ordering. There are many variations on both algorithms, but only the basic techniques are presented here.

Two Phase Locking

Two-phase locking (2PL) is the most widely used concurrency control algorithm. It guarantees that serialization errors will not occur because of the way locks are obtained. For conventional database systems where transactions are simple and short, two-phase locking provides effective concurrency control.

Two phase locking defines two types of locks on data items; read locks which must be obtained before reading data and write locks which must be obtained before writing. Multiple transactions are allowed to concurrently hold the same read lock, but if a transaction holds a write lock, no other transaction may hold the corresponding read or write lock. Transactions are required to obtain the appropriate locks for each data item they access and are restricted so that no locks may be freed until after all necessary locks have

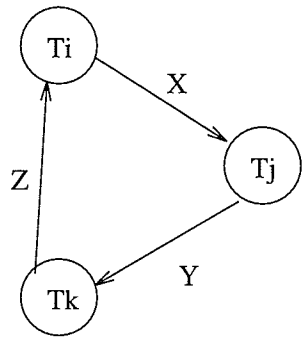

Figure 2.3: Cyclic Serialization Graph

been acquired (hence, "two phase"). By obtaining locks in this manner, serializability is guaranteed, but deadlock between transactions is possible.

To understand why 2PL guarantees serializability assume that a 2PL scheduler allows interleavings which violate serializability. This means that some collection of transactions which share data will interleave so that the corresponding serialization graph contains a cycle.

The following intuitive argument is offered in support of the claim. Consider, without loss of generality, the three transaction case. For an edge $T_i \rightarrow T_j$ to be induced in the serialization graph, two transactions must access the same shared data item in a conflicting way and T_i must access it first. Assume that T_i and T_j share only data item X, T_j and T_k share only data item Y and that T_k and T_i share only data item Z in a manner that causes a serialization error. This situation is depicted in Figure 2.3 where each arc in the figure is labeled with the name of the data item inducing the arc. Consider the lock and unlock operations that must occur to result in the serialization graph arcs shown. The arc from T_i to T_j implies that T_i must have locked X, then accessed it, and finally released it before T_j performs the same sequence on Y. Thus $Lock_i(X) \rightarrow Access_i(X) \rightarrow Rls_i(X)$, $Lock_j(Y) \rightarrow Access_j(Y) \rightarrow Rls_j(Y)$, and $Lock_k(Z) \rightarrow Access_k(Z) \rightarrow Rls_k(Z)$ (where $\rightarrow$ denotes the "happened-before" relation [Lam78]). Continuing this reasoning and focusing on the lock and release operations, the two-phase rule implies the result is the following

order of operations.

$$\underline{Lock_i(X,Z)} \rightarrow Rls_i(X,Z) \rightarrow Lock_j(X,Y) \rightarrow Rls_j(X,Y) \rightarrow$$

$$Lock_k(Y,Z) \rightarrow Rls_k(Y,Z) \rightarrow \underline{Lock_i(X,Z)}$$

This results in a cycle in the order of operations. Since this is impossible (an operation cannot occur before itself) 2PL cannot possibly create this serialization graph. More generally, 2PL cannot possibly create *any* cyclic serialization graph and therefore must enforce serializability. Bernstein, *et al.* [BHG87] provide a complete proof of the correctness of two-phase locking.

Timestamp Ordering

Timestamp ordering (TO) is not commonly used in commercial database systems. In timestamp ordering, a unique timestamp is generated for each transaction. The same timestamp is also associated with each of a transaction's operations. A TO scheduler ensures that two conflicting operations are processed in timestamp order.

It is easy to see that this scheduling scheme ensures serializability. Each edge $T_i \rightarrow T_j$ in a serialization graph implies that conflicting operations in T_i and T_j exist and that T_i's timestamp is less than that of T_j's. Now consider a cycle in the graph $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$. This implies that the timestamp of T_1 is less than itself, which is impossible. Since the serialization graph cannot contain cycles, serializability is ensured.

A TO scheduler immediately schedules operations that arrive for execution unless some other conflicting operation with a higher timestamp has already been scheduled. In this case, the transaction issuing the rejected operation is aborted and restarted (with a larger timestamp). To avoid aborting one read operation due to another, basic TO can be modified so that maximum timestamps are maintained for read and write operations

independently [BHG87]. Since reads do not conflict with other reads, a read operation need not be aborted due to the previous execution of another read with a larger timestamp. A reader must be aborted only if a writer with a higher timestamp has already executed. A writer must be aborted if either a reader or a writer with a higher timestamp value has already executed.

The TO implementation just described is obviously pessimistic. Another possible implementation of TO uses optimistic concurrency control. In this case, all operations proceed immediately regardless of timestamp value. A record of operations and their timestamps is maintained so that when a transaction attempts to commit, serializability can be verified [BHG87]. If necessary, some transaction(s) will then be rolled back.

2.1.6 Concurrency Control for Complex Data

Not all concurrency control protocols operate on primitive (unstructured) data items. In many applications, the data being operated on is more complex than just data items which may be read or written. In particular, a composite data item might be operated upon as a single atomic unit in one case and as its individual constituent components in another. Supporting the first form of access by obtaining multiple locks (one per component) is inefficient.

One approach to decreasing the number of locks which must be obtained is *multi-granularity* locking. In multi-granularity locking, locks are defined on complex items as well as simple ones and holding a lock for a complex data item is equivalent to holding the entire set of locks for its components. This is a primitive example of applying *semantic information* to the problem of reducing the overhead of concurrency control. In this case the semantic information exploited is the structure of data. Other approaches utilize semantic information about the processing performed on the data.

The implementation of multi-granularity locking typically includes the use of *intention* locks [BHG87]. If only conventional lock modes (Shared and eXclusive – corresponding to

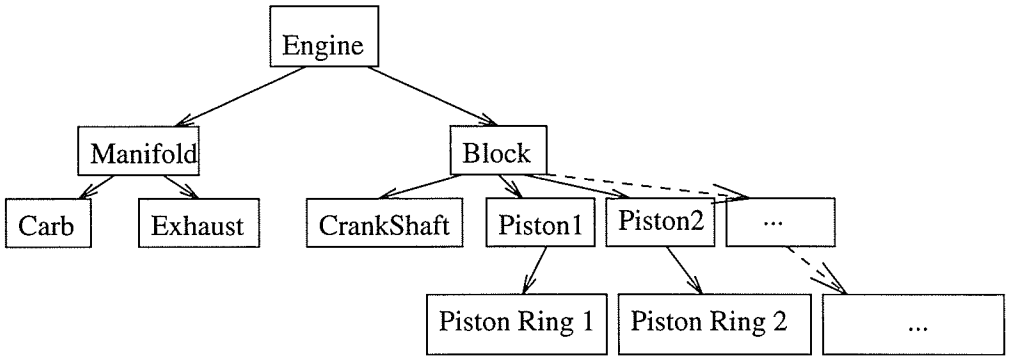

Figure 2.4: Hierarchically Structured Data

read and write access) are used with multi-granularity locks the overhead of concurrency control algorithms is unnecessarily high. The problem is that when a transaction wants to access a major component of a complex data object it must check all its subcomponents to see if another transaction is already accessing some part of the data object. Consider the example data structure shown in Figure 2.4[2]. If a transaction wants to access the engine as a whole it must check all subcomponents of the engine first to see if they are in use. If another transaction is accessing the second piston ring, a large amount of effort may be expended only to find that access cannot be granted to the requesting transaction.

Intention locks are set on the ancestors of data items actually being accessed. Both Intention-Shared (IS) and Intention-eXclusive (IX) locks may be set indicating that a subcomponent will be accessed in either a shared or exclusive fashion. A subsequent transaction attempting to access an object containing a sub-object which is already being accessed immediately finds the intention lock set and avoids the previously described overhead.

[2]Multi-granularity locking is normally discussed for a hierarchy which reflects the physical storage structure of the data (e.g., database/sector/relation/tuple/attribute) but is equally applicable to *logical* subdivision of the data.

2.2 Objectbase Systems

Database systems often provide a minimal data model consisting of data items upon which reads and writes are performed. Hierarchical or relational models are then defined over these basic data items to provide organization to the data and to make access to it easier. Existing concurrency control algorithms such as two-phase locking [EGLT76] and timestamp ordering [Tho79, BG81, KR81] perform adequately for these systems. New database applications, however, require more complex data models and associated concurrency control and recovery algorithms.

Attempts to use existing database technology in complex environments such as software development, engineering design, and groupware applications has highlighted the weaknesses of the traditional model and led to numerous proposals for advanced database systems [BK91]. Most efforts at building advanced database systems have been based on object-oriented database systems. *Objectbases* provide a powerful structuring mechanism for supporting complex data. An objectbase offers the best features of both objects (complex structure, encapsulation, inheritance, strong typing, and extensibility) and databases (persistence, data management, sharing, and access by declarative queries).

2.2.1 Objectbase Concepts

The concepts underlying objectbases have been discussed widely [ABD+89, Kim90, BM91, HPC93] but there is much disagreement and confusion over several key issues because different research areas have been focused on. This section provides an unbiased summary of the fundamentals underlying objectbases which will serve for subsequent discussion

Definition 2.4 An *objectbase* is a database supporting an object data model[3]. ■

An objectbase consists of a set of uniquely identified persistent objects that each contain structural and behavioural components. The structural component is a set of

[3]It is not a conventional database designed using object oriented techniques.

uniquely identified data items referred to as *attributes* whose values define the object's state. The behavioural component is a set of procedures, usually called *methods*, that are the only means of accessing the structural components and thereby modifying the object's state. Transactions in an objectbase thus consist of method invocations on one or more of the persistent objects.

Using objects for data structuring offers many advantages over the relational model.

- natural and consistent decomposition of an application
- data abstraction via encapsulation and information hiding
- enhanced type checking
- reusability via inheritance

Other advantages, specific to database applications, include:

- enhanced expressiveness (e.g., support for specifying entity-relationships)
- more natural modeling of data
- inclusion of database operations (i.e., methods) within the database providing enhanced access control to preserve privacy and integrity of data
- simplified data management
- natural support environment for active databases (e.g., triggers)

Several approaches to developing objectbases have been considered. Initial attempts to provide objectbase capabilities focused on extending the relational model by adding object oriented interfaces to relational database systems [BM91]. Unfortunately, the tuple model of storage does not provide full object capabilities. "Emulating" objects using relations thus decreases achievable performance and limits functionality.

An alternate approach adds persistent storage to object oriented programming systems. This approach divides the structure of an objectbase system into two parts; the first providing a persistent object storage facility and the second supporting high level access to the objects in the store[4]. This approach also has a shortcoming. Decisions

[4] Example systems implementing persistent object stores are discussed in the next section.

made concerning the design of the object store often affect (at least the efficiency of) the implementation of the higher level objectbase components.

Accessing Objects in an Objectbase

Existing objectbases typically support two means of accessing data. The first form of access is direct invocation of methods on specific objects. In this case, the object is identified at invocation time by specifying its unique object identifier (OID). The second form of access is using class-based *query* operations which invoke methods on all objects of a given class.

The benefits of query languages are that they are high-level and declarative in nature. This makes them easy for non-technical users since no knowledge of database structure is required. In conventional database systems, such queries are highly efficient because of indexes or similar structures defined on the data to enhance query speed. Indexes may also be applied to collections of objects and several prototype objectbase systems have defined indexes on objects and used them to enhance class query performance [MS86].

This dissertation considers only object access by OID.

Object Characteristics in Objectbases

Object characteristics include inheritance (possibly multiple inheritance), encapsulation, data abstraction, and part-of (*aggregation*) relationships. Encapsulation and the part-of relationship between objects play important roles in providing concurrency control in objectbase systems.

Encapsulation ensures that only object-local methods can access object attributes. This guarantees locality of effect and is a benefit in reasoning about the correctness of concurrent method invocations as well as the implementation of concurrency control as discussed in this dissertation.

Objectbase systems may support both shared and non-shared objects. Non-shared objects exist solely within other (compound) objects and are inaccessible independent of those objects. The common notion of the part-of relation is independent of whether or not non-shared objects are supported. Any object that *logically* occurs within an object (i.e., is either physically a part of it or is referenced by it) is considered a "part-of" that object. The existence of shared objects, makes possible the situation where an object is a part-of multiple objects. This is not normally permitted in object oriented languages, but is essential in objectbases where the sharing of data (i.e., objects) is fundamental[5].

The nesting of objects within other objects (physically or logically) has implications for concurrency control and recovery. Such nesting means that operations on objects are complex and therefore may be long-lived. This affects the way in which concurrency control must be provided thereby making concurrency control algorithms more complicated. This, in turn, affects recovery from transaction failures.

Other Objectbase Features

In design systems it is often necessary to support immutable objects and a versioning mechanism [MT86, Wie86, HPC93]. This permits multiple copies of an object to exist and possibly be accessed concurrently. Having multiple versions of an object also impacts concurrency control.

The issue of schema evolution is critical in objectbases. Conventional database systems have relatively static schemata and as such schema evolution is an infrequent operation. Because it is infrequent, schema evolution can be inefficient in traditional systems.[6] The applications using objectbases are typically design applications where changes to the contents and structure of the objectbase are frequent. Hence, schema evolution must

[5] One of the earliest stated goals of database systems was the elimination of redundancy and this implies that data that would otherwise be redundant instead be shared by (i.e., logically made a "part of") many objects.

[6] Schema evolution may even be an off-line process.

be efficiently and dynamically performed. Schema evolution may also affect concurrency control but is outside the scope of this dissertation.

2.2.2 Example Systems

The following discussion presents a representative sample of systems which implement complete objectbase systems. Familiarity with existing systems is important in appreciating their limitations and understanding the requirements for enhanced object base concurrency control and recovery. Prototype objectbase systems have been developed which support all the high level features expected of a complete objectbase system including such things as declarative queries and data management functions. Two important objectbase systems are ORION [KGBW90] and O_2 [VBD89, Deu90].

ORION is a full-featured, prototype objectbase system developed at MCC[7]. Three versions of ORION have been prototyped; a single user–single site version, a client/server version, and a "fully distributed" version. The features of the objectbase from a functional viewpoint are the same in all versions of ORION – distribution is transparent to users of ORION.

In addition to providing a (possibly distributed) object store, ORION provides a complete set of high-level database features including:

- declarative queries
- automatic query optimization
- composite objects and reference by OID
- transaction management and recovery
- version control and change notification
- schema evolution

[7] Microelectronics and Computer Technology Corporation, Austin, Texas

Query processing is based on queries applied to all objects belonging to a given class. Efficiency is ensured by the construction and maintenance of indexes on selected class attributes. Query optimization is based on statistics maintained by ORION, gathered while the system operates, which are used to determine the order in which sub-components of a query should be evaluated.

ORION supports both types of part-of relationships described previously. Queryable, shared objects are identified by an OID of the form {class_identifier,instance_identifier}. This naming structure provides fast access to the class schema and methods given an OID. Objects physically containing other private objects are referred to as "composite" objects. Objects contained within other objects are not addressable except by reference through their enclosing object.

Transaction management in ORION is based on object level locking for concurrency control and logging for recovery. Transactions are serializable and logical object locks are held until transaction completion. Physical page locks are also used, but are held only for the duration of an update to the page. Effectively, the logical locks enforce serializability while the physical locks ensure mutual exclusion. Early release of page locks limits the well-known *false sharing* problem [HP90]. The storage subsystem interacts with the lock manager to accomplish locking. Recovery support is limited to the maintenance of "before-images" in an UNDO log.

ORION has implemented various versioning and change control algorithms. It currently supports two kinds of versions; *transient* and *working* versions. Working versions correspond to the conventional notion of an immutable version of an object. Transient versions may be freely updated and under certain conditions may be promoted to working version status.

Schema evolution is supported in ORION using a deferred-update policy wherever possible. For example, if a class definition (part of the schema) is modified to remove an attribute, the instances of that class (its objects) are not immediately updated to remove

the attribute. Instead, access to the attribute is prevented via the schema change and the actual instances of the attribute are removed as the corresponding objects are referenced. Hence, information which is not needed is maintained in the object base for a potentially long period of time. When deferred-updating can be used though, the corresponding schema update operations are performed very quickly.

O_2 provides many of the same features as ORION. Specifically, it provides, persistent storage (based on the Wisconsin storage system [CDKK85]), declarative queries from within a high level interactive query language, reference by OID, transaction management (using a combination of two-phase locking on objectbase pages and optimistic techniques for class updates during schema evolution), and optional recovery control. O_2 provides language interfaces to the user which are heavily influenced by conventional database programming and query languages. It also defines a unique user interface generation tool which produces graphical software to access the objectbase. O_2 is designed as a client/server system.

Somewhat different approaches to objectbase systems are taken by the GemStone [MS86], Iris [WLH90], and Cactis [HK89] systems. A unique feature of GemStone is that queries over a collection of objects (i.e., instances of a class) are supported rather than only queries over *all* the objects of a given class. This more general form of query is useful, but a significant amount of effort is expended maintaining defined indexes to support such queries in an efficient way. Unlike ORION, GemStone uses an immediate update policy when doing schema evolution to ensure that no unnecessary data is stored in the objectbase. The price paid for this is that the schema update may be long lived and other users of the system are "locked" out while the update is in progress.

In the Iris objectbase, objects are typed (belong to a class) but contain no state. Instead, the state associated with an object (attribute values and methods) are modeled by functions. Stored, derived and external functions are supported. Stored functions correspond to the data resulting from the evaluation of the function (i.e., its "extent")

and are useful for frequently executed operations since they avoid recomputation of the function. Derived functions are specified strictly *within* Iris using functional expressions while external functions are specified and compiled outside of Iris making it impossible to derive any semantic information about their behaviour.

The Cactis OODBMS also supports the notion of stored functions (these correspond roughly to object methods in its data model). In most cases, such functions are limited to expressing how certain object attributes may be derived from others. Inter-object method invocation is not supported. Different objects are related to one another in the style of the entity-relationship model. Cactis is one of the few functional database systems which uses timestamps for concurrency control.

Other important research OODBMS's include ODE and Exodus which both tightly couple the database and the object programming language. ODE [AG89, ADG93] is a research object base system under development at AT&T Bell Labs based on an extension of C++ known as O++. ODE is very language oriented with the objectbase being defined, accessed, and managed using O++. It offers a wide variety of advanced features including inter-object triggers, object versioning, object-local consistency constraints, and the ability to define arbitrary sets of related objects and to iterate over them in a declarative way. Exodus [CDG+90, RCS92] is a similar system based on another extended version of C++ known as E. Exodus was developed to be an "extensible" database system – one which could support the addition of new fundamental data types. To support this, C++ was chosen as the database programming language thereby making Exodus an objectbase. Unlike ODE, E provides transparent rather than user-visible and explicit persistence.

2.3 Advanced Concurrency Control

This section discusses concurrency control in advanced database systems (including objectbases). A comprehensive overview of many of the techniques discussed in the following pages is given by Barghouti and Kaiser [BK91]. Ramamritham and Chrysanthis [RC92]

also provide valuable insight into the fundamental issues of concurrency control. The chief contribution of their paper is a taxonomy of correctness criteria which separates database consistency requirements from transaction correctness properties. This is important in considering concurrency control mechanisms other than serializability which provide both database consistency and transaction correctness in a single mechanism.

2.3.1 Improved Concurrency Control in Conventional Database Systems

Attempts have been made to improve concurrency control in conventional database systems. Efforts in this area may be divided into those which correspond to slight modifications to, or extensions of, existing techniques (e.g., [BG83, Moh90]) and those which introduce drastically different concurrency control algorithms (e.g., [Wei89b, DK90]). This section summarizes the results which may be applicable to improving concurrency control in objectbases.

Optimism

Some applications [MT86, BS91] benefit greatly from the use of optimistic algorithms for concurrency control. Whenever it can be determined *à priori* that the probability of conflicting operations in concurrent transactions is low, optimistic techniques should be considered. In this case, as long as other characteristics of the transactions (e.g., lack of support for roll back) do not prevent their use, optimistic concurrency control methods will normally provide higher performance. Herlihy proposes optimistic concurrency control techniques for abstract data types [Her90]. Due to the close relationship between abstract data types (ADTs) and objects, Herlihy's results are also applicable to objects.

Multi-Version Data

Bernstein and Goodman [BG83] discuss modified timestamp ordering and two phase locking algorithms which support multiple versions of data items. Their model supports *writes* to data items which result in the creation of new versions of those items. This permits late *read*s to overlap with subsequent operations thereby enhancing concurrency.

Multiversion timestamp ordering (MVTO) [Ree79] assigns each transaction T_i and its operations a unique timestamp, $ts(T_i)$. When the MVTO scheduler processes a read from transaction T_i, the version of x with the largest timestamp less than or equal to $ts(T_i)$ is read. A write operation by transaction T_i is rejected if the scheduler has already processed a read by T_j of a version written by T_k such that $ts(T_k) < ts(T_i) < ts(T_j)$. To ensure recoverability, T_i's commitment is delayed until all other transactions that wrote a version read by T_i have committed.

Multiversion two phase locking (MV2PL) [SR81] adds a new lock type, *certify* to existing read and write locks. Certify locks conflict with all other lock types. When a data item is first read by transaction T_i, the latest committed version is used. On subsequent reads, the latest version written by T_i is read. Once a write lock is obtained, T_i's writes create new versions as expected. When a transaction is ready to commit, its write locks are upgraded to certify locks. This upgrading is only allowed to take place once there are no outstanding read locks on the corresponding data items (since certify locks conflict with *all* others). T_i commits only when all the locks for versions it read (and did not write) have been upgraded to certify locks by the transactions which produced them.

These basic algorithms have been extended and refined by numerous researchers including, recently, Wu, *et al.* [WYC93] and Morzy [Mor93].

Structured Transactions

One disadvantage of conventional transactions is their monolithic structure. This impacts both concurrency control and recovery. Subdividing a transaction into parts and structuring it to match the *physical* structure of the data it operates on can significantly increase potential concurrency and simplify recovery. One approach to adding structure to transactions is the work on *nested transactions* by Moss[8] [Mos85] which has been refined by many other researchers (e.g., [Wei89c, HR93]).

Nested transactions provide two fundamental benefits; they allow concurrency within a transaction, and they support recovery at a finer granularity than the entire transaction.

Nested transactions divide an existing transaction into a parent transaction and one or more sub-transactions. The sub-transactions themselves can also be subdivided so nesting may be to an arbitrary depth. Each parent transaction controls the execution of its sub-transactions, specifying which sub-transactions execute concurrently and the recovery procedure(s) to be applied in the event of a sub-transaction failure. In objectbases, transaction nesting can be based on the natural, hierarchical subdivision of the data being operated on.

Two forms of transaction nesting have been proposed:

- closed nested transactions, and
- open nested transactions

Both closed and open nesting hide the updates to data items made by one sub-transaction from all other transactions until the sub-transaction in question *pre*-commits[9]. Thus, atomicity is provided at the sub-transaction level. Closed nesting further restricts access

[8] Earlier work on nested transactions was done by Reed[Ree78] but Moss' work is considered seminal.

[9] When a sub-transaction pre-commits, it is informing its parent transaction that it has completed its work. That work is not actually committed (i.e., made durable) until the root parent transaction commits.

so only other sub-transactions of the same parent (i.e., siblings and their descendants) are allowed to see changes after *sub*-transaction commitment. Other transactions see updates only after the top-level parent transaction successfully commits. The updates of open nested sub-transaction are visible to all other transactions following sub-transaction commitment. This may have the effect of permitting greater concurrency. If one sub-transaction aborts however, other sub-transactions which were run concurrently with it must also be aborted since they may have seen data that the failed sub-transaction wrote. This can result in cascading aborts. Thus, open nesting increases potential concurrency but complicates recovery.

Concurrency control for closed nesting based on extended two-phase locking has been defined and is described here. Enhanced concurrency between sub-transactions is provided by a lock inheritance mechanism. When a sub-transaction completes, the locks it holds are passed up to its parent who *retains* them. A retained lock may be subsequently obtained by a descendant of the transaction which retains it but by no other transaction. This process is known as *upward inheritance* of locks.

Moss [Mos85] defines four locking rules for concurrency control in nested transaction systems:

Rule 1 Transaction T may acquire a lock in exclusive (X)-mode if no other transaction holds the lock in X-mode or in shared (S)-mode and all transactions that retain the lock in X- or S-mode are ancestors of T.

Rule 2 Transaction T may acquire a lock in S-mode if no other transaction holds the lock in X-mode and all transactions that retain the lock in X-mode are ancestors of T.

Rule 3 When sub-transaction T commits, the parent of T inherits T's locks (both held and retained). After that, the parent retains the locks in the same mode in which T held or retained them.

Rule 4 When a transaction aborts, it releases all locks it holds or retains. If any of its ancestors hold or retain any of these locks they continue to do so.

These locking rules implement closed nesting since no transaction can see the uncommitted results of any other transaction.

Recent work extending nested transactions has modified the semantics of nested transactions to include downward inheritance of locks in addition to upwards inheritance [HR93]. In downwards inheritance, locks are passed from a parent transaction to some child without the parent retaining the lock. Supporting downward inheritance of locks permits certain desirable decompositions of transactions not possible with upwards inheritance. Downward inheritance requires open nesting with the corresponding benefits and limitations (downward inheritance permits greater concurrency while maintaining the *relatively* desirable recovery properties of open nested transactions.).

Other forms of structured transactions besides nested transactions have also been proposed. Kaiser and Pu [KP92] suggest a different form of transaction subdivision referred to as *split/join* transactions. A transaction split occurs dynamically and divides one transaction into two serializable transactions (*not* sub-transactions). Resources held by the first resulting transaction can then be released to be used by other transactions. When the second transaction can execute concurrently with other transactions using the resources released by the first, there is an increase in concurrency. Transactions are normally only split when it is known that concurrency will be increased sufficiently to offset the cost of performing the split.

Transactions can also be dynamically joined to produce a single transaction. After a join, the resulting transaction holds the resources of both its constituents. By combining splits and joins it is possible to transfer resources from one transaction to another.

While nested transactions are programmed statically, the expected use of split/join transactions is dynamic. When conflicts develop between running transactions, one or more may be dynamically split to resolve the conflict. The splitting must be done so that

the resulting transactions are serializable and the users of the transactions must accept a decreased level of atomicity. Since the unit of atomicity is the transaction, splitting a transaction into multiple transactions means that some of the resulting transactions may complete and commit while others do not. This is not the expected behaviour of the original transaction but may be acceptable in some situations (for example if the user associated with the original transaction explicitly requested the split). Simple tests based on statically produced readsets and writesets can be used to ensure that the transactions resulting from a split will be serializable.

Split/join transactions offer the same benefits as nested transactions in a different way.

Semantic Concurrency Control

Concurrency control based only on analyzing read and write operations at run time fails to exploit a significant amount of concurrency. For example, a certain execution of two transactions may not be conflict serializable but will leave the database in a consistent state. Having higher level, *semantic* information about the constituent transactions may permit the execution to be accepted. Semantic information can either be supplied by the database programmer or can be derived *automatically* by the compiler.

Early work on *semantic* concurrency control by Garcia-Molina [GM83] used semantic knowledge of the behaviour of transactions to improve scheduling in a distributed database system. The motivation for their work was investigating the potentially long communication delays which occur in distributed systems and how they affect lock-based concurrency control protocols. An important contribution of this work was the recognition that only a subset of all transactions (those referred to as *sensitive* transactions) will actually function incorrectly if they see inconsistent data. An example of a sensitive transaction is one which outputs results to the user based on inconsistent data. Non-sensitive transactions may interleave with other non-sensitive transactions.

Garcia-Molina suggests that transaction programmers should categorize their transactions according to their semantics and derive *compatibility sets* which group together transactions that can freely interleave. An algorithm for semantics-based concurrency control is described assuming that compatibility sets are available. It is observed that the complexity of analyzing transactions and specifying compatibility set information places a great burden on the programmer.

Farrag and Özsu describe a semantics based concurrency control algorithm [FO89]. They define a class of *relatively consistent* schedules that contain both serializable and non-serializable interleavings that never violate database consistency. The associated concurrency control algorithm enforces fewer restrictions on transaction interleavings than other semantics based algorithms.

Farrag and Özsu extend and modify the work of Garcia-Molina by easing the restrictions on which transactions are allowed to interleave freely. This is accomplished by replacing compatibility sets with *breakpoints* as a means of specifying legitimate interleavings. A breakpoint is a point in a transaction where it is "safe" to permit interleaving with other specified transactions. By not requiring that entire transactions be free to interleave with one another as required by Garcia-Molina's work, greater concurrency is potentially attainable.

More recent work in semantic concurrency control by Weihl [Wei88] defines *commutativity-based* concurrency control. Weihl's work is based on concurrently accessed abstract data types and extends naturally to objectbases.

Definition 2.5 Two operations O_1 and O_2 are *commutative* if the effect of performing O_1 followed by O_2 is the same as O_2 followed by O_1. ■

Two commutative operations may be performed in either order without affecting serializability[10].

[10] Commutativity, like any semantics-based algorithm, does not obviate the need to ensure mutual exclusion, only serializability.

For example, using commutativity allows executions which would not be allowed by conflict serializability. Assume there are two transactions:

$\mathbf{T_1}$: Write(X) Increment(Y)
$\mathbf{T_2}$: Write(X) Increment(Y)

and consider the interleaved execution described by the following history:

H: $W_1(\mathrm{X})\ W_2(\mathrm{X})\ I_2(\mathrm{Y})\ I_1(\mathrm{Y})$

If write (i.e., "$W()$") and increment (i.e., "$I()$") operations are treated as atomic steps and conflict serializability is used as the correctness criterion, this execution must be rejected since the resulting serialization graph will contain a cycle. If, however, commutativity is also considered, the history $W_1(\mathrm{X})\ W_2(\mathrm{X})\ I_2(\mathrm{Y})\ I_1(\mathrm{Y})$ is equivalent to $W_1(\mathrm{X})\ W_2(\mathrm{X})\ I_1(\mathrm{Y})\ I_2(\mathrm{Y})$ which is serializable and hence the execution is allowed.

Simple examples of commutative actions can be taken from commutative operations on numeric data (e.g., the increment operations above). Commutativity also applies to more complex data structures. For example, two insertions into a set are commutative operations as are inserting and deleting *different* elements.

Weihl [Wei88] describes two forms of commutativity; *forward* and *backward* commutativity. Forward commutativity is a symmetric relation which is thought to be more restrictive than the asymmetric backward commutativity relation. The terms "forward" and "backward" relate to recovery techniques. In forward commutativity, a forward recovery technique (based on, say, intentions lists) is required while in backward commutativity, backward recovery methods (e.g., undo lists) are acceptable.

Nakajima [Nak92] has introduced the notion of *generalized* commutativity which is defined to be the "union" of forward and backward commutativity. This offers the advantage of permitting more possible interleavings than either forward or backward commutativity alone. To accomplish this, Nakajima restricts the area of application to multi-version

objectbases. Multi-version objects support both forward and backward recovery methods concurrently thereby allowing forward and backward commutativity to be mixed. The method proposed is a useful extension of Weihl's commutativity relations for multi-version objects.

A fundamental disadvantage of commutativity based concurrency control is that it relies on the transaction programmer to specify commutativity tables. This complicates programming greatly and is inherently error prone. A better solution would be to determine commutativity automatically.

Lynch [Lyn83] introduced the concept of multi-level *atomicity* where flat transactions are decomposed into separate atomic units. Serializability is still enforced at the entire transaction level to ensure correctness but the decreased granularity of atomicity permits greater potential concurrency. A transaction is decomposed into a series of atomic sub-steps each separated by a breakpoint where the execution of atomic sub-steps from other transactions can interleave. The set of breakpoints is dependent upon the transaction which is to be interleaved. In this sense, semantic information is being used to enhance concurrency in much the same way as that described by Farrag and Özsu [FO89].

Transactions are pre-analyzed to group them into nested classes. At the lowest level of nesting each individual transaction occurs alone in its own class. Higher levels of nesting group related transaction classes of lower levels together to form new transaction classes. At the highest level there is a single class which consists of all sub-transactions (and, indirectly, all transactions). Transactions which are more closely related (by the transaction class hierarchy) are allowed to interleave at a finer level of atomicity. This simplifies the task of specifying breakpoints since they can be stated simply in terms of nesting level. Lynch observes that this approach does not describe all valid/useful interleavings.

Weikum [Wei91] and Weikum and Schek [WS92] discuss the use of open nested, multi-level transactions. Semantics are applied at each level of a collection of openly nested

sub-transactions. Intuitively, a transaction is implemented using operations at a high level of abstraction which are implemented using operations of lower level abstractions. At the bottom level are the actual read and write operations on data items in the database. Concurrency is supported at any/all levels of abstraction and as such concurrency control can be realized at any needed level.

Naturally, concurrency decisions made at higher levels must be supported by their implementations at lower levels. For example, two increment operations may be allowed to execute in an arbitrary order at a higher level (based on commutativity) but their underlying read-write implementations must not interleave or lost-updates may occur. That is, the scheduling decision made at the higher level assumes that increments are atomic actions and therefore the lower level must ensure that they are executed atomically (or at least in a way that is equivalent to atomic). Sub-transactions need not always be considered atomic. Higher level decisions can be made without assuming atomicity, but those decisions will be different from ones made assuming atomicity. This results in an open-nested model which in turn further increases potential concurrency. Unfortunately, like all open nesting models, it also makes recovery more difficult.

Weikum states:

> "The choice of levels involved in a multilevel transaction strategy reveals an inherent trade-off between increased concurrency and growing recovery costs." [Wei91, pp. 132]

This conclusion, like multi-level transactions themselves, relates directly to the nested objectbase environment and thus is of interest to the dissertation.

A recent objectbase concurrency control method that utilizes semantic information has been proposed by Muth, *et al.* [MRW+93]. In this work, commutativity information is applied to object methods rather than to abstract data types. This illustrates the relevance of commutativity to objectbases.

2.3.2 Concurrency Control in Objectbases

Concurrency control algorithms for objectbases are less developed than those for other database environments. Many objectbase systems use simple object locking schemes augmented with facilities required to support features of the object data model such as inheritance. Although objectbases have been identified as an enabling technology for advanced applications, advanced concurrency control algorithms for them have not been used in practice. The current "state of the art" in objectbase concurrency control is now discussed by first considering a typical prototype system which is then contrasted with research efforts in the area.

Concurrency Control in ORION

Concurrency control in ORION is complicated by the object-oriented data model it supports [GK88]. Concurrency control is done using locking protocols applied to three hierarchies: the class inheritance hierarchy, the conventional multi-granularity hierarchy, and the composite object (i.e., part-of) hierarchy. Locking is used to enforce serializability and is applied on an object by object basis. Recovery is managed using a mechanism based on conventional logging techniques.

The basic concurrency control protocol (which is modified by the other protocols to support the class hierarchy and composite objects) is an extension of multiple granularity locking with intention locks. The hierarchy upon which the locks and intention locks are obtained and freed is the class/object hierarchy. That is, if an object consists of a collection of other objects, then a transaction may access a sub-object by locking it only after having obtained an appropriate intention lock on the "parent" object(s).

The need for class-hierarchy locking arises because ORION supports dynamic schema evolution. It allows several kinds of updates to the schema to be performed while the objectbase is being actively accessed. For instance, super-class definitions can be modified in which case, concurrency control must prevent a class (or its instances) from being

updated during the super-class change. Another case requiring class hierarchy locking is when queries are posed against a class and all its sub-classes. Similar situations arise when dealing with composite objects and also necessitate changes to the multi-granularity locking protocol.

Lock conversion is supported in ORION in an effort to decrease the time during which transactions are blocked from accessing a complex object. A transaction is allowed to obtain a Shared lock on a complex object with the intent of updating a part of it and since the object is locally cached, the transaction can freely update its copy and only need upgrade to the eXclusive lock when it attempts to write the updated version of the object. This allows other transactions wishing only to have read access to the object to proceed while a potentially long-lived update is occurring.

The most severe restriction in ORION (and other objectbase prototype systems) is the decision to use object level locking. This significantly limits the potential concurrency exploitable in user generated transactions. Specifically, it precludes the possibility of two transactions concurrently accessing an object even if those transactions will not conflict. Therefore, locking on a per-object basis is too coarse grained.

Research in Objectbase Concurrency Control

Several theoretical treatments of concurrency control in object bases have appeared in the literature [RGN90, HH91, AE92, RE92]. These papers typically define a working model for the structure and behaviour of transactions on objectbases and then derive algorithms that provide concurrency control under the given models. These theoretical models make practical assumptions and are important to consider.

Rakow, Gu, and Neuhold [RGN90] define *object-oriented serializability* for open nested transactions in objectbases. The concurrency control algorithm described takes advantage of the semantics and nesting of operations to increase concurrency. Concurrency control is based on conflict analysis and commutativity is used to define conflicts (i.e., a conflict

is specified as occurring between two operations if they do not commute). It is assumed that commutativity tables for all methods of all objects are available. In this sense, their approach to concurrency control is based on operation semantics.

Operations are allowed to occur concurrently subject to conflicts and orderings required for serializability (arising due to related conflicts). Three types of "dependencies" are defined and used to determine orderings:

1. action dependencies
2. transaction dependencies
3. added dependencies

Action dependencies are those arising due to the expected semantics of sequential execution within a method (i.e., the compiler-theory definition of dependence) or because ordered method invocations within a method have a transaction dependency. Transaction dependencies are those which arise due to *related* conflicting operations. That is, if sub-transactions conflict and therefore impose an ordering, that ordering must be adhered to by the method(s) invoking the sub-transactions. Hence, such transaction dependences must be passed back to the invoking methods. Added dependencies refer to those which arise when two unrelated object methods invoke methods in a common object. If a *transitive* transaction dependence arises between the invoked methods in some third object, an added dependency is returned to each of the invoking methods to become an action dependence there. Added dependencies are equivalent to transaction dependencies but arise due to method invocations from apparently *unrelated* transactions.

Action dependencies incorporate all of the necessary orderings to ensure the serializability of objects. Hence concurrency control is maintained by scheduling in accordance with the partial order induced by the action dependencies. The derivation of the orderings required to ensure object serializability would be simpler and more efficient to implement in a closed nesting scheme since less ordering information has to be shared between transactions.

Transaction synchronization in objectbases has also been addressed by Hadzilacos and Hadzilacos [HH91]. This work includes a precisely defined model of an objectbase and the transactions executing on it. Unlike Rakow *et al.*, Hadzilacos and Hadzilacos define transaction management algorithms assuming that nested method invocations behave as *closed* nested transactions. This simplifies the process of ensuring object-oriented serializability.

Like Rakow *et al.*, a partial order based on commutativity is defined between the steps of a method. Since nesting is closed, this information alone can be used to ensure object-local serializability. Inter-object serializability (corresponding to "transaction" and "added" dependencies in Rakow's work) must be ensured separately to guarantee that the ordering of method invocations and their descendants is consistent. An algorithm is described that constructs a graph according to the method-local partial order and inter-object ordering information which will be acyclic only if the history used in constructing it is equivalent to a serial history of the same method invocations with the same nesting structure. Unfortunately, this algorithm corresponds to testing to see that methods in a concurrent execution see the same views as in a serial execution and hence is inherently inefficient.

Resende and El Abbadi [RE92] describe a graph testing concurrency control protocol for objectbases using the transaction model of Hadzilacos and Hadzilacos. The suggested optimistic protocol specifies how to construct a graph for each method invocation. The vertices in such a graph correspond to the *terminated* sub-transactions (method invocations) of the method in question. The vertices are added to the graph dynamically as each sub-transaction terminates along with those edges specified by the protocol. After each vertex is added, the scheduler tests for serializability by ensuring that the resulting graph is acyclic. The orderings induced by executed sub-transactions are recorded in the parent's graph. When a sub-transaction completes, its graph is incorporated into the graph of its parent. At that point, the consistency of the two graphs (and the executions they correspond to) is checked by verifying acyclicity. If the resulting graph is not acyclic,

then the sub-transaction's execution has caused the transaction's execution to become non-serializable and the transaction is aborted.

This is an optimistic method and it is not guaranteed that conflicts will be detected early and have a minimal affect. In the worst-case, a deeply nested transaction might be forced to abort only after executing its last sub-transaction. Two versions of Resende and El Abbadi's protocol are described; one which is order preserving for non-interleaved operations (which may be required in some applications) and one which is not.

Other work in concurrency control in objectbases has been done by Agrawal and El Abbadi [AE92]. They present a locking protocol for nested transactions in objectbases where a uniform treatment is given to both class and instance objects. An interesting aspect of their work is that conflicting operations can share locks on the same objects as long as the order between conflicting operations is maintained.

2.3.3 Objectbase Concurrency Control Using Static Analysis

Only recently has the idea of using statically derived information to enhance concurrency control in objectbases been investigated. Initial work using static analysis to permit finer granularity concurrency control at reasonable cost was undertaken by Graham, Zapp, and Barker [GZB92]. The primary result of this preliminary work was to show that data access information in an objectbase with nested method invocations could be summarized using bit strings which indicated which object attributes may be referenced by specific methods. This information can be statically derived and then used to determine whether or not concurrent method invocations can proceed without conflict.

Hakimzadeh and Perrizo [HP93] have applied similar static information to the problem of fine grained, object-local scheduling in objectbases. Their OC-ROLL (Object Centered ROLL) algorithm is an extension of the earlier ROLL algorithm [Per91]. In OC-ROLL object-local bit strings are maintained which reflect the object attributes being referenced. The high overhead normally associated with attribute level locking is avoided

by using the bit strings to predict when conflicts will occur. Effectively, the bit strings provide a form of summary information for methods which permits schedulers to avoid the overhead of obtaining one lock for each attribute. Local serializability is maintained at each object by detecting potential conflicts and scheduling accordingly. Global (inter-object) serializability is ensured by consistently scheduling conflicting operations at each object according to the order of arrival of transactions. This approach provides a simple means of guaranteeing global serializability A similar approach suggested by Zapp and Barker [ZB93a] permits global serializability to be ensured by algorithms which do not have this property. The relative costs and benefits of the two approaches merit further investigation.

Another related approach is that of Malta and Martinez [MM93]. They too use static analysis to predetermine which object attributes are referenced by each method. Rather than maintaining bit vectors though, they convert the bit vectors into specific lock modes using an algorithm based on finding the strongly connected components in a graph. Once the lock modes are defined, concurrency control is implemented through simple locking using the extended set of locks and an appropriate lock compatibility matrix. Unlike the work of Graham and Barker [GZB92], no attempt is made to support nested objects.

2.4 Fundamentals of Recovery

Concurrency control is a mechanism for ensuring the correctness of concurrent transaction executions in the *absence* of failures (i.e., transaction aborts and/or system crashes). Recovery is a mechanism for ensuring correctness given that failures may occur.

Automatic recovery from every conceivable form of failure in a database system is not feasible. For example, recovery from such catastrophic events as earthquakes and nuclear wars may be impossible. Furthermore, errors arising due to incorrect transaction input or errors made in coding transactions[11] cannot be detected and recovered from automat-

[11] If coding errors are possible, the fundamental assumption of serializability, that correct transaction

ically. There is no convenient way to deal with such errors without the specification and checking of extensive consistency constraints[12] which may themselves be flawed.

This leaves three types of failures that must be addressed by a database management system:

1. Detected Transaction Failures (i.e., aborts),
2. System Failures (e.g., operating system crashes), and
3. Media Failures (e.g., a damaged disk surface).

This dissertation focuses on providing automatic recovery for the first two failure types. Media failures are not addressed. It is assumed that these are handled through other, generally unrelated, mechanisms such as regular backups and disaster recovery mechanisms [BHG87].

2.4.1 The Recovery Problem

The database system must ensure that the results of every committed transaction are correctly reflected in the database and that none of the results of uncommitted transactions are. Furthermore, this must be guaranteed in the face of unpredictable system failures as well as transaction aborts. This requirement can be expressed functionally as the following two rules (paraphrased from [BHG87][pp.177–178]):

Rule 2.1 (*The Undo Rule*) If data item x's location in the stable database presently contains the last committed value of x, then that value must be saved elsewhere in stable storage before being overwritten by an uncommitted value. This permits the uncommitted change to be *undone* following a failure.

executions preserve database consistency, is violated.

[12] Whether it is even possible to define a set of consistency constraints for a real-world system that completely prevents transaction errors is doubtful.

Rule 2.2 (*The Redo Rule*) Before a transaction can commit, the value it wrote for each data item must be in stable storage so that any incomplete database updates can be *redone* following a failure.

The "stable storage" referred to in the undo and redo rules is any form of storage that persists over operating system/machine failures which may corrupt data stored in main memory.

To support recovery, information must be written to stable storage so that after a crash the database management system can determine if the database has been left in an inconsistent state. If it is inconsistent, the saved information must also support the database system in making the database consistent again. The database is consistent after recovery if its state is that which would have occurred if all updates of all committed transactions were made to the objectbase and no updates of uncommitted transactions were. The reason that no updates of uncommitted transactions may be included is to ensure that no partial transaction results are written (this satisfies the required atomicity property of transactions). The reason that the updates of all committed transactions must be included is to satisfy the durability property.

Many different recovery techniques have been suggested. These include schemes where neither, one, or both of *explicit* transaction redo and undo are required. Thus, the space of recovery algorithms may be subdivided into four *recovery classes*:

redo, undo – systems where recovery may require both redoing and undoing some transaction operations

redo, no-undo – systems where recovery may require redoing some transaction operations but never requires undoing any

no-redo, undo – systems where recovery never requires redoing any transaction operations but may require undoing some

no-redo, no-undo – systems where recovery requires neither redo nor undo (i.e., no recovery, *per se*, is necessary)

There are specific advantages and disadvantages to each class.

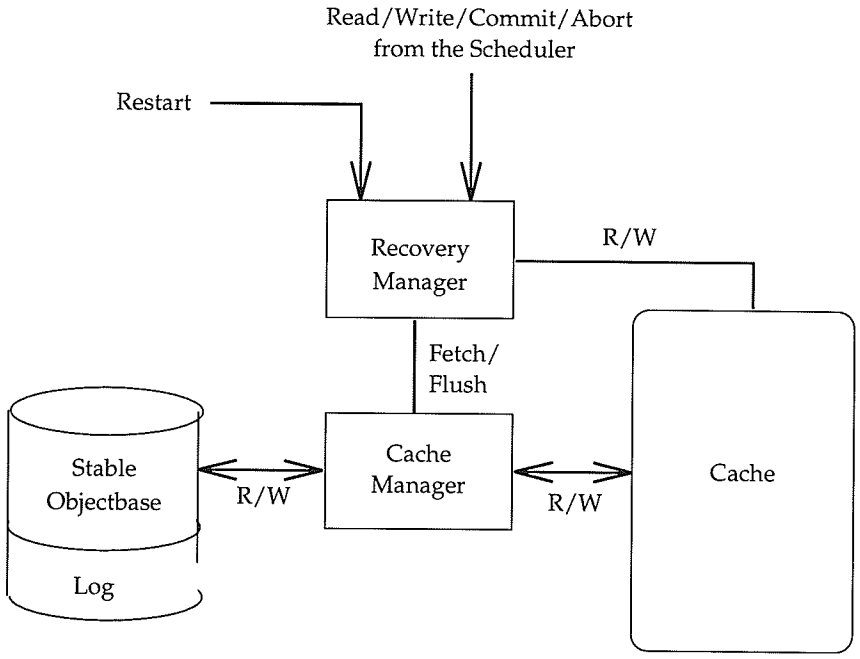

Figure 2.5: Recovery Architecture

2.4.2 Assumed Recovery Architecture

An architecture which focuses attention on specific recovery strategies and their costs is now presented. The recovery architecture shown in Figure 2.5 is used throughout the dissertation. The objectbase environment, *per se* does not affect this *high-level* structure.

The recovery manager of Figure 2.5 is responsible for processing operations ('read', 'write', 'abort', and 'commit') received from the scheduler and the 'restart' operation after a system failure. These operations are executed against an in-memory cache controlled by the cache manager. The recovery manager may interact with the cache manager by requesting explicit *fetches* of data into the cache and *flushes* of data out to the stable database and/or log. This control over the cache manager's functioning may be necessary to ensure the durability of committed transactions.

2.4.3 Analysis of Recovery Related Costs

Recovery processing is performed during both transaction execution and system restart (after a failure). Thus, recovery processing can be implemented as two separate components, one running during transaction processing and the other at restart. The desired performance characteristics of the two recovery system components varies between database applications. In some cases, it is acceptable to incur great expense during restart to minimize overhead during transaction processing. In others, spreading costs more uniformly across both normal and restart processing is more desirable. The overall cost of recovery is a function of the class of recovery algorithm chosen and the specific implementation strategies selected for the two recovery components.

During normal processing, recovery overhead is incurred both to save information which will facilitate later recovery and, in the case of transaction aborts, to apply that information to recover from the effects of aborted transactions. During restart processing, overhead is incurred only to perform recovery.

The overhead associated with recovery comes from three specific sources.

1. Output operations to write results to the database,
2. Output operations to write log information, and
3. The CPU overhead of recovery algorithm execution (for aborts and restart).

The dominant cost in transaction processing (including recovery) is disk I/O operations. This overhead is addressed, in part, in all database systems by caching data in main memory. The problem is that main memory is volatile and results stored in it may be lost during system or power failures. Thus, cache flushing and logging must be used to guard against lost results. Flushing and logging introduce disk access overhead but it is less than the overhead of accessing all data directly from disk (i.e., operating without a cache). It is primarily I/O overhead that must be minimized to achieve high performance support for recovery during transaction execution.

Basic Recovery Schemes and Definitions

A given recovery protocol must perform I/O operations during each phase of transaction execution; when it starts, while it executes, and when it completes. Recovery schemes will be analyzed by considering their I/O overhead during each of these phases. Some terminology is now defined and some common recovery schemes currently in use are described.

Definition 2.6 The *recovery log* (*"log"*) is a sequentially written structure on stable storage containing information required for the purpose of recovery. ■

Typically, information concerning transaction activity as well as copies of data from the database are stored in the log.

Definition 2.7 A *before-image* of some data item 'X' is a copy of the most recently committed value of 'X' stored in the log. ■

A before-image preserves the state of some data on stable storage so it may be freely modified elsewhere (e.g., in the database itself and/or the cache).

Definition 2.8 An *after-image* of some data item 'X' is a value of 'X' written by an active transaction stored in the log. ■

In-place updating occurs when transactions write, possibly uncommitted, data into the database and thereby overwrite the previous data. Logically, each data item in the database has a single location which is updated directly by all transactions that write the data item. In practice, the actual update of the data item *in* the database may be delayed due to caching. The possibility that there is uncommitted data in the database affects recovery.

The process of writing a before-image of a data item to the log before allowing the data item to be overwritten is known as *write-ahead logging* (WAL). Intuitively, the log record

(which preserves the before-image) is *written ahead* of the data item update(s). The use of in-place updating with write-ahead logging is the most popular recovery technique in commercial database systems [HR83].

When in-place updating is used, write-ahead logging is required to make undo possible. In this case, I/O overhead is incurred to write before-images of data to the log so, if a transaction which updates that data aborts, or otherwise fails to complete, the data may be restored to its state before the failed transaction began. Additional, *necessary*, I/O overhead is also incurred when a transaction commits because the updated data items (which may currently reside in the cache) must be written either to the stable database or to the log as after-images.

A log record indicating commitment must also be written[13]. Overhead during abort includes the writing of the log "abort" record and restoration of any before-images of data items written by the aborted transaction. Restart requires a scan of the log (on disk) to determine what processing is required and subsequent writes to the stable database to restore consistency via undos and/or redos.

An alternative to in-place updating (and possibly the use of logs) is the application of *shadowing* techniques. Rather than updating data directly in the database, updates are made to copies of the data and then, when the corresponding transaction successfully commits, the shadow copy replaces the original in a single atomic I/O operation. If the transaction aborts, the shadow copy is discarded.

Definition 2.9 A *shadow copy* of data item 'X' is a copy on stable storage, that contains the values for 'X' written by the transaction which "owns" the shadow copy. ■

The shadow copy always has a stable store location but may also be cached in memory for performance reasons. The chief advantage of shadowing is that when shadow copies

[13] Just as one indicating the start of a transaction must be written when the transaction begins.

of data are maintained there is no need to preserve before-images since the old values of data items are left intact. The cost of writing the before-image is therefore saved.

To allow different transactions to see different copies of various data items in the database, it is necessary to introduce a level of indirection in data access. This is provided by a structure called a *directory* which implements a logical mapping from data item names to the stored copies of the corresponding data items for each active transaction

At commit time, the shadow copies produced by a transaction must "replace" the corresponding original copies. For data which may be atomically written, this is a simple process. The original data is simply overwritten by the new data. Most transactions however, update large, possibly complex data structures that must be atomically written back to the database. Doing this safely requires care and is complicated when the writable data unit (normally a "page") contains data other than that which has been modified.

I/O costs associated with shadow paging include the need to create the shadow copy initially, to update the database when the corresponding transaction commits and to manage the transaction directories (which must be stored on stable storage). These are new costs not incurred by the update-in-place/WAL scheme. Savings in I/O overhead are made during transaction execution because there is no need to save before-images for undo since undo is not required. This also means that during restart recovery and abort there is no overhead incurred to perform undo. This is the fundamental benefit of shadowing techniques.

In an attempt to decrease the I/O overhead of logging *physical* write operations, logging logical has been suggested. *Logical logging* is the process of recording logical update operations rather than simple writes to facilitate recoverability [BHG87]. Since a single logical update operation (e.g., insert tuple 'X' into relation 'R' in a relational database) typically encompasses many separate writes, the *number* of log entries written is reduced. Thus, significant savings will be achieved.

The price paid for the decreased volume of log entries and the corresponding decrease

in I/O overhead during transaction execution is additional overhead during restart processing. Suppose a failure occurs after a transaction has committed and logically logged its actions but before the actual results of those actions are written from the cache to the stable database. Since the transaction has committed, redo must be performed to ensure that the transaction's effects are written to the stable database. This is no longer simply a matter of copying the updated data from the stable log to the stable database. Instead, the log must be examined to determine for which operations in the log results were not propagated to the database. All such operations must be re-executed (at additional cost) to make the database consistent again.

The use of logical logging also affects undo processing. If some of the effects of a transaction have been made in the stable database, then it must be possible to undo those effects even though logical logging is used. Two approaches to performing undo are possible; restoring pre-images or executing a compensating transaction. If the restoration of pre-images is used *all* the data which may be accessed by a logical operation must be pre-written to the log. To use compensating transactions, one such compensating transactions must be generated for each logical operation. Further, compensating transactions (used to implement undo) must be "idempotent".

Definition 2.10 An operation (undo, redo, or other) is *idempotent* if it may be performed multiple times and still produce the same result it would if executed only once. ■

Repeatedly writing a single updated value for some data item 'X' from the log to the stable database is clearly an idempotent operation – no matter how often you write it, the value is the same. On the other hand, a compensating transaction like "decrement-by-1" (which undoes the effects of the transaction "increment by 1") is not idempotent. Compensating transactions must either be idempotent or must be treated, during restart, like transactions themselves so that they provide atomicity of execution.

In some applications, it is desirable to attempt to reduce the cost of the recovery process during restart. This cost is very high if restart processing must use *all* log entries and the system operates without failure for long periods of time (the expected case). When a failure does occur, the cost of processing such a large log is high. One approach to addressing this problem is to periodically ensure that updates corresponding to "old" log information are reflected in the stable database. The old information can then be discarded from the log since it is no longer required during restart. This process is known as *checkpointing*.

Definition 2.11 *Checkpointing* is the process of ensuring that certain logged information is known to be reflected in the database. ■

By explicitly flushing specific data from the log and/or cache to the database during normal operation, the corresponding redo operations need not be performed during restart (and the corresponding log records may be discarded).

Three types of checkpointing are defined by Bernstein, *et al.* [BHG87].

commit consistent checkpointing – (also referred to as *transaction consistent checkpointing*) guarantees that the data checkpointed corresponds only to the results of committed transactions. No partial transaction results are checkpointed

cache consistent checkpointing – (also referred to as *action consistent checkpointing*) guarantees that data written to the cache are also checkpointed. This means that checkpointed data corresponds to completed actions, some of which may be actions from incomplete transactions.

fuzzy checkpointing – writes only a subset of the updated data in the cache. Normally this is the set of data items that have not been recently flushed. The set of values so flushed may correspond to either committed or uncommitted transactions.

To perform commit consistent checkpointing, it is necessary to stop transaction scheduling and allow the active transactions to complete prior to actually flushing *all* the cached data to the database. This introduces two fundamental sources of delay; the wait for transactions to complete, and the wait for the cache to be fully flushed. Such long latencies are undesirable because they interrupt normal transaction processing.

To address this latency, cache consistent checkpointing may be employed. This precludes the need to wait for the active transactions to complete before performing the checkpointing. To avoid mutual exclusion and other synchronization problems, the active transactions are "idled" during cache consistent checkpointing while the flushing is performed.

Finally, fuzzy checkpointing further reduces the required latency by flushing only that subset of the cached data which has not been recently flushed. If most of the updated information in the cache is regularly flushed by the default cache replacement algorithm then there will be little data to be checkpointed and thus the latency of checkpointing will be small. The data that will most often be flushed using fuzzy checkpointing is that which corresponds to "hot-spots" in the database. The flushing of this data is particularly important in reducing restart overhead since without such checkpointing, many updates of the hot spot data will be recorded in the log. Thus, log processing at restart must examine many entries and there is correspondingly high overhead.

A survey of important results concerning recovery in flat and nested transaction systems is now presented as a basis for future discussion. The survey is selective not exhaustive.

2.4.4 Recovery in Flat Transaction Systems

Research into recovery in conventional, flat transactions is mature. The fundamental concepts just outlined are described by Haerder and Reuter [HR83] and at length by Bernstein, Hadzilacos, and Goodman [BHG87]. The former provides a useful taxonomy

of recovery techniques and their respective characteristics. Such "classical" treatments of recovery are detailed but lack the rigor of a formal theory (such as that which benefits concurrency control).

A theory of recovery and reliability in database systems has been developed by Hadzilacos [Had88] which seeks to provide a foundation for formal discussion of concepts and algorithms related to recovery. The formalism presented treats recovery as an extension to correctness criteria for concurrency control which holds in the presence of commonly occurring failures. It cleanly divides recoverability into two components; ensuring that transactions are *recoverable* (as discussed in Section 2.1.4), and ensuring that they are *resilient*. Recoverable means that transactions do not depend on data written by transactions which may still abort. Resiliency means that all information necessary to recover from unexpected failures is written to disk prior to transaction commitment. This duality is used throughout this dissertation.

Recall that, most recovery schemes are based on in-place updating with write-ahead logging. Jhingran and Khedkar [JK92] have performed a detailed performance analysis for write-ahead logging in many database systems. Their analysis centers on the cost of recovery after a failure for various access models including uniform, varying read/write ratios, and hot spots. Their results clearly indicate the effect that recovery techniques have on the performance of the recovery process and thus the importance of choosing recovery techniques carefully.

Although recovery and concurrency control are often discussed separately in the literature, there is a close relationship between the two. That relationship is examined by Weihl [Wei89a] and by Alonso, Agrawal, and El Abbadi [AAE93]. This relationship is important because the concurrency control protocols introduced later in the dissertation must be recoverable if they are to be of practical use.

Weihl addresses the interaction between concurrency control protocols and recovery procedures for abstract data types. As with concurrency control, results concerning re-

covery which are obtained for abstract data types are often applicable to object systems with little or no modification. Weihl's work focuses on the relationship between recovery and concurrency control when *commutativity* is used as the conflict criterion. It is shown that the choice of recovery methods affects the potential concurrency available by restricting the applicable definition of commutativity. Specifically, *forward* commutativity can only be applied when recovery is based on deferred update (no undo) techniques while *backward* commutativity can only be applied when update in place (no redo) recovery strategies are used.

Alonso, *et al.* [AAE93] also consider the effects of concurrency control on recovery and vice versa. They expand on the work of Schek, Weikum, and Ye [SWY93] which defines *prefix reducibility* (or "PRED") by providing practical implementations of a class of executions equivalent to PRED. Prefix reducibility is a property of transaction executions which guarantees correctness both in terms of concurrency control and recovery. Thus, PRED is a class of correct concurrent transaction executions containing only executions which are both serializable and recoverable. Incorporating both serializability and recoverability into a single correctness criterion offers the distinct advantage of being able to compare different correctness-ensuring algorithms. This cannot be done with conventional definitions of correctness as was realized by Weihl [Wei89a].

Most recovery techniques are based on redo and undo. An entirely different approach to recoverability is the use of *compensating transactions*. Rather than preserving state information to ensure that transactions may be undone, the use of compensating transactions permits arbitrary updates to be made freely by transactions which may subsequently abort (perhaps even after temporarily committing). A compensating transaction may be run at any time to compensate for the effects of an aborting transaction.

Korth, Levy, and Silberschatz [KLS90] address the subject of recovery through the use of compensating transactions. They advocate the use of compensating transactions in long-duration and/or nested transactions as a means of recovery management in the

presence of early externalization (i.e., making uncommitted results available to other transactions). This permits recovery without aborts (either simple or cascading). The chief drawback of this approach to recovery is the requirement on the part of the transaction programmer to specify the compensating transactions.

Finally, Moss, Griffith, and Graham [MGG86] have explored the relevance of "abstraction" in recovery management. Abstraction refers to the implementation of sequences of low-level, physical operations as higher-level, logical ones. Although their work is limited to a read/write access model, they provide intuition into the behaviour of nested object systems. Their model deals with multi-level operations and introduces the notion of *layered serializability* and *layered atomicity*. Their application of abstraction to recovery is limited to transaction aborts and does not address system failures.

2.4.5 Recovery in Nested Transaction Systems

Nested transactions are a fundamental part of objectbase concurrency control, so it is important to review recovery techniques for nested transactions.. This section surveys important work in this area.

Providing recovery for nested transactions using logs is discussed by Moss [Mos87]. It is shown that simple extensions to existing redo/undo logging suffice to support recovery for nested transactions. The only significant change to conventional logging techniques is required to permit the undo of a transaction *and* all its sub-transactions at once. This may be accomplished by chaining sub-transactions onto the parent transaction's chain of undo operations. This approach works for both transaction aborts and system failures, however during system failures, *complete* user transactions are always aborted. No attempt is made to permit committed sub-transactions to survive a crash unless they are a part of a committed user transaction.

Haerder and Rothermel [HR87] also discuss the refinements which must be made to conventional (i.e., flat transaction) recovery schemes to support nested transactions.

Their focus is on cooperating sub-transactions using *conversational* interfaces between calling and called sub-transactions. In this model, a single sub-transaction may be invoked and return results multiple times. Like open nesting, this makes recovery more difficult since the scope of sub-transactions which may have seen uncommitted data is difficult to determine. This model is not of interest in the dissertation.

The use of checkpoints to decrease the granularity of recovery in nested transactions is also discussed by Haerder and Rothermel for both conventional and conversational call interfaces. A set of criteria that determine which sub-transactions require UNDO relative to a given checkpoint is then presented. These criteria are summarized as follows:

- Sub-transactions committed prior to the checkpoint are correct and require no processing during recovery.
- Sub-transactions committed (or active) after the checkpoint are rolled back.
- Sub-transactions started before the checkpoint and committed prior to the start of recovery are preserved and may be accepted or rejected by their parent.
- Sub-transactions started before the checkpoint and *not* committed before the start of recovery are rolled back in their entirety.

Finally, Rothermel and Mohan [RM89] present a recovery method for nested transactions based on write-ahead logging. This method is known as ARIES/NT which is an extension of ARIES [MHL+89] that supports nested transactions. Their work attempts to support a wide variety of nested transaction models and includes specific data structures and algorithms used in supporting and delivering recovery. The data structures and algorithms presented are effectively the same as those for flat transactions (in ARIES) except that the tree structure of the sub-transaction nest is incorporated into the chain of transaction log records. Naturally, the algorithms which process this chain are modified to process the tree structure.

Chapter 3

The Object Base Concurrency Control Problem

Before discussing the problems associated with concurrency control in objectbases it is necessary to provide a framework. The first section of this chapter presents a high level overview of the object model used and assumptions made. This is followed by a simple taxonomy of objectbase characteristics which affect concurrency control. The specific focus of the dissertation is described in relation to this taxonomy. Next, formal definitions of objectbase and concurrency control concepts are provided. This is followed by a discussion of concurrency in objectbases which outlines specific requirements which must be met by effective concurrency control solutions. Finally, the limitations of existing object base concurrency control schemes and how to improve them using statically derived semantic information are presented.

3.1 Assumed Environment

While some of the ideas espoused in the dissertation have applicability in many types of database systems, discussion is limited to objectbase systems. This decision and the ones

made concerning the object model provide a manageable but still realistic environment in which to apply static analysis.

3.1.1 Object Model Overview

An objectbase consists of a set of uniquely identified persistent objects that each contain structural and behavioural components. The structural component is a set of uniquely identified data items referred to as *attributes* whose values define the object's state. The behavioural component is a set of procedures, called *methods*, that are the only means of accessing the structural component and thereby modifying the object's state. In what follows, the j^{th} attribute of object O_i is denoted a_{ij}. Similarly, an object's method(s) are identified using the notation m_{ij}.

Definition 3.1 An *object* $O_i = (\mathcal{S}_i, \mathcal{B}_i)$ where:

1. i is the unique identifier of the object,
2. $\mathcal{S}_i$ is the object's structure composed of attributes such that $\forall a_{ij}, a_{ik}(j \neq k) \in \mathcal{S}_i$, $a_{ij} \neq a_{ik}$,
3. $\mathcal{B}_i$ is the object's behaviour composed of methods such that $\forall m_{ij}, m_{ik}(j \neq k) \in \mathcal{B}_i$, $m_{ij} \neq m_{ik}$. ■

Point (1) assigns a unique name to each object. Point (2) specifies the attributes of the object and Point (3) specifies the methods of the object.

The steps performed by a given object method are logically divided into those which access an object's local attributes (*local steps*) and those steps which "access" non-local attributes via method invocations on other objects (*message steps*). Methods which only access object attributes are distinguished and will be referred to as *leaf methods* reflecting their position in the tree-like structure induced on method executions by the dynamic call sequence.

The sets of attributes and methods are specified in *class definitions* which are used to instantiate objects. It is assumed that objects are instantiated offline using techniques not addressed in the dissertation.

Definition 3.2 An *objectbase* $\mathcal{OB} = \{O_1, O_2, ..., O_k\}$ is a set of uniquely identified objects upon which method invocations may be made. ■

Objects within the objectbase interact with one another by method invocations.

Objectbase users execute transactions on objects by invoking methods that manipulate their attributes. An invocation of method k on object j made by a transaction T^i is denoted m^i_{jk}. A subsequent invocation of method r in object q from within m^i_{jk} is denoted m^i_{qr}. Thus, the superscript specifies the *originating* user transaction for each object method execution arising from it.

Transactions are submitted against the objectbase by *multiple, concurrent* users. A transaction submitted by a user consists of a single object method invocation. A method may, of course, invoke many other methods to accomplish work of arbitrary complexity. By restricting user transactions in this way all operations performed on the objectbase are made *statically* available for analysis. User Transactions are denoted UT^i.

Methods invoked by user transactions and the methods they invoke[1] are each executed atomically as *nested atomic transactions* [Mos85, HR93]. The collection of nested method invocations, from the first user initiated invocation to the final method invoked, forms a *transaction tree* [Elm92] (or possibly a *transaction lattice*). It is assumed that all methods are correctly formed and execute on a consistent objectbase so a user transaction, in the absence of other user transactions, will always produce a new consistent objectbase.

The operations in conventional databases are reads and writes of data items. In object bases, the objects are the data, and the operations on them are the methods provided in

[1] Method invocations may be either direct or indirect through an arbitrarily long sequence of other invocations.

their class definitions. The notion of conflict between reads and writes is well understood. The following definition of conflicting methods (i.e., object operations) is provided.

Definition 3.3 Two methods in a class (or object) *conflict* if they contain steps which access attributes in a conflicting manner. If two methods m_{ij} and m_{ik} conflict then this is denoted $m_{ij} © m_{ik}$. ■

Due to encapsulation, conflicts between methods in different objects are impossible.

In what follows, the set of all the operations (i.e., direct or indirect method invocations) of user transaction T^i is denoted OS^i (the transaction's Operation Set). The transaction's termination condition is denoted by $N^i \in \{Commit, Abort\}$.

Definition 3.4 A *nested object transaction* is a partial order $T^i = (\Sigma^i, \prec^i)$ where:

1. $\Sigma^i = OS^i \cup \{N^i\}$,
2. for any two $m^i_{jk}, m^i_{jl} \in OS^i$ which conflict, either $m^i_{jk} \prec^i m^i_{jl}$ or $m^i_{jl} \prec^i m^i_{jk}$,
3. $\forall m^i_{jk} \in OS^i, m^i_{jk} \prec^i N^i$,
4. the termination conditions of all $m^i_{jk} \in OS^i$ are consistent and equal to N^i. ■

Point (1) enumerates the operations performed by the transaction. Point (2) ensures that operations which conflict are ordered. Point (3) guarantees that all transaction operations occur before its termination and Point (4) ensures that all of a transaction's sub-transactions either commit or abort with their parents.

This dissertation adopts the convention of indicating method invocation using a procedure call syntax. Therefore, a method invocation (including one within a user transaction) can accept several parameters and, where appropriate, optionally return explicit results (e.g., $\{R_1, R_2\} = m^i_{jk}(arg1, arg2, ...)$). This is not intended to be functionally significant but is merely the most convenient and familiar notation available.

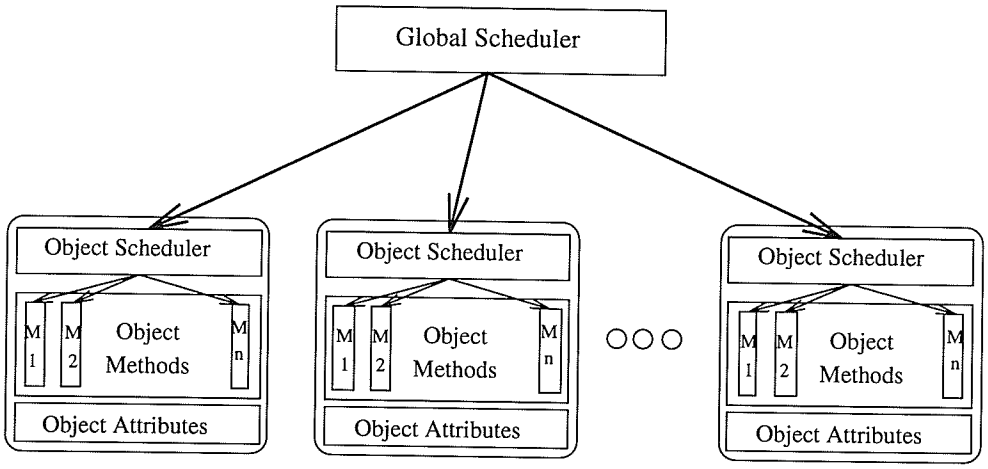

Figure 3.1: Two Level Scheduling Architecture

Where necessary, a specific execution order for a set of method invocations (or other operations) will be specified using the notation "$\rightarrow$"[2] (e.g., $m^i_{rl} \rightarrow m^j_{sm} \rightarrow m^j_{tn}$). This should not be confused with serialization order which, when necessary, will be indicated using the symbol "$\Rightarrow$".

Finally, it is assumed that each object schedules its own method invocations in co-operation with a global scheduler as suggested by Hadzilacos and Hadzilacos [HH91] and developed by Zapp and Barker [ZB93c, ZB93a]. Each object scheduler only ensures the serializability of concurrent method invocations executed at that object. The global scheduler is responsible for ensuring inter-object serializability. Scheduling this way permits the use of different, but compatible concurrency control mechanisms at each object. This permits concurrency control to be tailored to a particular object based on the semantics of its methods. This two-level scheduling architecture is illustrated in Figure 3.1.

[2]This reflects Lamport's *happens-before* [Lam78] relation.

3.1.2 Object Model Restrictions

In this section limiting assumptions of the object model are detailed and justifications for the assumptions are provided.

The popularity of object oriented programming languages (OOPLs) has influenced peoples' view of object oriented computation. The fundamental concepts of object orientation have, to some extent, been lost in the syntax and semantics of particular OOPLs. This section attempts to circumvent any confusion which may arise in this way.

The objects referred to in this dissertation are "vanilla flavoured". They embody the key requirements of objects without including unnecessary extensions. Discussing such simple objects offers two benefits; first, the problem is simplified by the elimination of unnecessary special cases and second, the work is more generally applicable because it does not cater to particular features of a given OOPL.

The key requirements of object oriented programming are:

Encapsulation – Access to an object's data is restricted to the methods of that object. This provides the desirable properties of data abstraction, information hiding and locality of effect. Encapsulation directly affects concurrency control.

Inheritance – This is the key feature that differentiates object oriented programming from the use of abstract data types. The ability of a class to inherit both structure and behaviour from its super classes is key to code re-use and directly affects concurrency control.

Polymorphism – Polymorphism takes various forms. In its simplest form, *ad-hoc* polymorphism provides the ability to overload function names and operators thereby permitting the re-use of user-meaningful names in different contexts. Ad-hoc polymorphism does not affect concurrency control. More generally, polymorphism permits objects to be treated as if they were of their declared class or any super class thereof. This form of polymorphism affects concurrency control.

This dissertation assumes that encapsulation is *strictly* enforced. Thus only object methods can access object data. This seems obvious, but not all OOPLs adhere to the principle. For example, C++ provides the `friend` mechanism which permits closely related classes to share information in a controlled (and hopefully limited) way. While `friends` are a useful extension to the basic OO concept, they are not strictly required and complicate the static analysis which is fundamental to this dissertation. Although it may be possible to support `friends`, the problem of doing so is not addressed.

Inheritance, fully supported. Both single and multiple inheritance are considered in this dissertation but inheritance is considered to be a static process. This means that when a class is defined, its constituent attributes and methods are fully specified. Either they are declared explicitly within the class or are inherited (logically *into* the class) from existing superclasses.

A question related to the implementation of polymorphism is whether or not method invocations are bound to object methods statically or dynamically. In static binding, the specific method referred to by a method invocation is determined at compile time while in dynamic binding this association is only made at run time. Dynamic binding is required to support full polymorphism.

Dynamic binding permits a collection of objects instantiated from classes derived from a common base-class to be manipulated using common code as if they were all objects of the common base class. References to method names which are overloaded between classes and a common superclass are resolved (i.e., dynamically bound to) at run time based on the actual type of the object the method is invoked on. Dynamic binding is supported but may affect the efficiency of the algorithms presented later.

The focus of the research presented in this dissertation is concurrency control for the *method executions* performed in response to user transactions. There are other concurrency control related operations which are not addressed in the dissertation (e.g., schema/class evolution). While these are important issues, they are beyond the scope of

Symbol	Meaning
SV	Single Version Objects
MV	Multi Version Objects
NT	Nested Transactions
FT	Flat Transactions
NS	No Object Sharing
S	Full Object Sharing

Table 3.1: Legend for Figure 3.2

the dissertation.

Finally, in order to ensure the efficiency of the compile time analysis of methods (which will be described in Chapter 4) it is assumed that methods are specified using only *structured* [ASU86] control flow primitives[3].

3.2 A Taxonomy of Objectbase Characteristics Affecting Concurrency Control

There are three important characteristic properties of objectbases related to concurrency control. These are illustrated in Figure 3.2 (a legend for which appears in Table 3.1) which defines a taxonomy of objectbase characteristics affecting concurrency control. The three characteristics defining the taxonomy are: support for nested transactions, object sharing, and support for multi-version objects. This defines eight fundamental objectbase concurrency "models" (Table 3.2) where static analysis may be applicable to enhancing concurrency.

[3]The presence of unstructured control flow such as that arising from `GOTO` statements complicates static analysis to the point that it would detract from the main thesis.

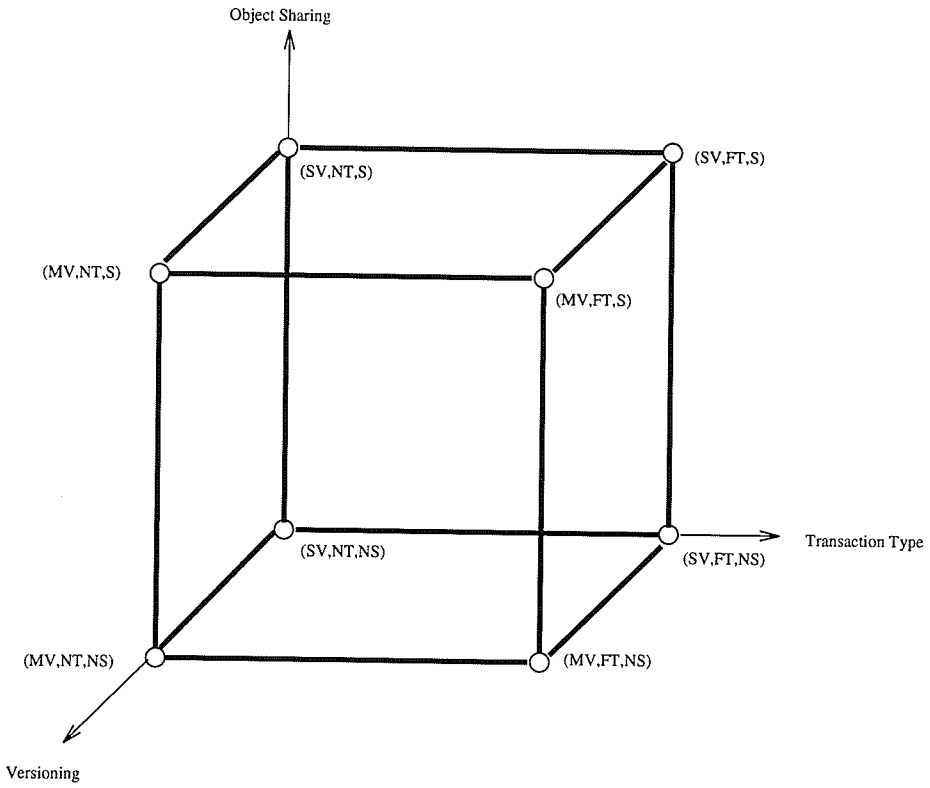

Figure 3.2: Taxonomy of Objectbase Characteristics

Model Name	Description
(SV, FT, NS)	Single Version, Flat Transactions, Non-shared Objects
(SV, FT, S)	Single Version, Flat Transactions, Shared Objects
(SV, NT, NS)	Single Version, Nested Transactions, Non-shared Objects
(SV, NT, S)	Single Version, Nested Transactions, Shared Objects
(MV, FT, NS)	Multi Version, Flat Transactions, Non-shared Objects
(MV, FT, S)	Multi Version, Flat Transactions, Shared Objects
(MV, NT, NS)	Multi Version, Nested Transactions, Non-shared Objects
(MV, NT, S)	Multi Version, Nested Transactions, Shared Objects

Table 3.2: Objectbase Concurrency Models

The nested transaction concept described by Moss [Mos85] can be extended to objectbases to permit greater concurrency. Rather than having the transaction programmer *explicitly* specify sub-transactions, in an objectbase the sub-transaction structure can be *automatically* determined to reflect the inherent object calling structure. When an object calls another object, each method execution can be performed as a sub-transaction of the method from which it is invoked[4]. Supporting nested transactions in this way provides the normal benefits of conventional nested transactions in an objectbase system; (1) sub-transactions may be executed concurrently with one another and/or their parent, and (2) recovery overhead is decreased since only failed sub-transactions need to be re-executed.

Another important characteristic of objectbase systems that affects concurrency is object sharing. Concurrency control protocols are simpler if an object can only be referenced from within a *single* object. Conceptually, references from multiple objects constitute object sharing where the sub-object is a child (or *part–*) of the objects referencing it. If object sharing is not permitted then two method invocations on distinct objects which

[4]This can also be done when methods are invoked in non-nested objects, as will be seen later.

are not themselves related by a parent/child relationship, access disjoint sets of objects. Such method invocations can be executed concurrently without restriction.

Bernstein and Goodman [BG83] describe a technique using *multiple versions* of data items to enhance concurrency in conventional database systems by allowing late reads to execute concurrently with other reads and writes. Multi-version concurrency control can also be extended for application in objectbase systems where it can permit greater concurrency than in conventional database systems. The resulting multi-version object concurrency control protocols are typically optimistic.

The emphasis in this dissertation is on problems in the classes (SV, NT, S) and (SV, NT, NS). Results in the classes (MV, FT, NS) and (MV, NT, NS) will also be presented.

3.3 Formal Definitions

This section provides formal definitions for concepts referred to throughout the dissertation. It extends the earlier definitions given in Section 3.1.1. These definitions provide the basis for exact descriptions of algorithms and ensure that the meaning of the results presented will be clear.

3.3.1 Fundamental Concepts

The formal definition and description of such fundamental concepts as objects and the classes from which they are derived is now presented. These definitions capture relevant aspects of class specifications which are required to perform static analysis. They are not specific to concurrency control.

Definition 3.5 The objectbase *class library* $\mathcal{C} = \{C_1, C_2, ..., C_n\}$. ■

The objectbase class library enumerates all of the class definitions available for instantiating objects. It defines the types/classes of all objects in the objectbase.

Definition 3.6 The objectbase *primitive types* $\mathcal{PT}$ = { `int, float, char, ...` }. ■

The primitive types of the objectbase are a set of builtin types whose semantics are well-defined and understood by the class compiler. The types specified in Definition 3.6 are only examples. The actual types may vary from implementation to implementation.

Definition 3.7 A *class* $C_i = (A_i, M_i)$ where:

1. i is the unique class identifier.
2. A_i is the set of attributes that the class defines such that $\forall a_{ij}, a_{ik}(j \neq k) \in A_i,\ a_{ij} \neq a_{ik}$,
3. M_i is the set of methods that the class defines such that $\forall m_{ij}, m_{ik}(j \neq k) \in M_i,\ m_{ij} \neq m_{ik}$. ■

Point (1) uniquely identifies the class within the objectbase class library. Point (2) enumerates the attributes defined by the class and Point (3) enumerates the methods. Note that the notation for object and class methods (m_{ij}) and for object and class attributes (a_{ij}) has been overloaded. The correct use of the notation should be clear from context. In situations where ambiguity is possible, explicit clarification is provided.

Informally, a class defines a new type which may subsequently be used as a basis for the instantiation of objects of that type. Such a type definition consists of the specification of the constituent data elements and available operations of the type. Each object instantiated from the class inherits the attributes and methods of the class.

Definition 3.8 An *attribute* $a_{ij} \in A_i$ is of type T_j where:

1. j is the unique attribute identifier in C_i,

2. $T_j \in (\mathcal{C} \cup \mathcal{PT})$. ■

Point (1) specifies the attribute's name while Point (2) specifies the type (either a user defined class or a primitive type) of the attribute.

A similar definition applies when dealing with parameters.

Definition 3.9 A *parameter* P_i is of type T_i where:

1. i is the unique parameter identifier,
2. $T_j \in (\mathcal{C} \cup \mathcal{PT})$. ■

Both input and output parameters (in Definition 3.11) adhere to this definition.

Definition 3.10 A *step* S_{ij_k} in method m_{ij} is one of:

1. an expression involving a method's attributes and/or input parameters,
2. an assignment to a method's attributes or output parameters,
3. a method invocation, or
4. a control structure (e.g., `if`, `while`, etc) which may contain other steps. ■

Definition 3.11 A *method* $m_{ij} \in M_i$ is an ordered triple (I_{ij}, O_{ij}, S_{ij}) where:

1. j is the unique method identifier in C_i.
2. $I_{ij} = (IP_1, IP_2, ..., IP_{r_j})$,
3. $O_{ij} = (OP_1, OP_2, ..., OP_{s_j})$,
4. $S_{ij} = (S_{ij_1}, S_{ij_2}, ..., S_{ij_{t_j}})$. ■

Point (1) specifies the method's name. Point (2) identifies the set of *input parameters* of the method and Point (3) identifies the set of *output parameters* (i.e., return values) of the method. Finally, Point (4) identifies the sequence of executable statements (referred to as *steps*) of the method.

A method corresponds to the common notion of a function which accepts parameters, returns results, and operates by changing only the object's attributes and/or by calling methods in the same or other objects.

A transaction model is also required to capture the execution concurrency that this dissertation seeks to control. As previously described, nested method invocations give rise to nested transactions. *Nested object transactions* have already been defined (Definition 3.4). The execution of a particular method m_{ij} as a transaction is referred to as an *Object Transaction* and is denoted OT_{ij}.

A useful concept in reasoning about transaction executions is that of a *history* (sometimes referred to as a *schedule*) which collects information concerning multiple, concurrent transaction executions and the ordering of their (especially, conflicting) operations. Concurrent transaction executions can be shown to be correct if the history resulting from a concurrent execution is "equivalent" to that of a serial execution of the same transactions. This is the normal approach used to prove the serializability (and hence, correctness) of concurrent transaction executions.

A history is a partial order of the executions of transaction operations where the ordering relation must include all pairs of operations that conflict.

Definition 3.12 A *history* H, of a set of transactions $\mathcal{T} = \{T^1, T^2, \ldots, T^n\}$, is a partial order $(\Sigma_T, \prec_T)$ where:

1. $\Sigma_T = \bigcup_{i=1}^{n} \Sigma^i$.
2. $\prec_T \supseteq \bigcup_{i=1}^{n} \prec^i$.
3. For any two conflicting operations $o^i_{kr}, o^j_{ks} \in \Sigma_T$, either $o^i_{kr} \prec_T o^j_{ks}$, or $o^j_{ks} \prec_T o^i_{kr}$. ■

Point (1) collects all operations of all transactions in the history while Point (2) collects the ordering relationships. Point (3) ensures that conflicting operations *between* different transactions are also included in the partial order. This definition enables us to reason

about transaction operation execution sequences using histories and the ordering relation $\prec_T$. For example, given that the initial state of an object O_k, prior to the execution of the transaction's in T is s_k, then three method invocations executing in the sequence; $O^i_{kr} \prec O^j_{ks} \prec O^l_{kt}$ (i, j, l not necessarily distinct) will move O_k into the state $s_{k'}$. This can be expressed using a sequence of state transition functions such that $s_{k'} = o^l_{kt}(o^j_{ks}(o^i_{kr}(s_k)))$. State transition functions are a convenient way to reason about method executions and their effects on objects.

3.3.2 Ordering Relations in Method Specifications

This section discusses the expected execution ordering relations *within* a given class method. It is assumed that classes are specified using a sequential programming language and as such, the expected execution order on the statements in a method is serial (reflecting the semantics of the language). Our definition of intra-method correctness is therefore equivalence to serial execution (i.e., serializability). Any concurrency introduced into a method execution must not violate serializability.

Following the approach of Hadzilacos and Hadzilacos [HH91], the steps of a method are divided into two sets: the *local steps*, and the *message steps*.

Definition 3.13 The *local steps*, $LS(m_{ij})$, of a method are those which operate on object attributes (and possibly method-execution local variables). ■

Definition 3.14 The *message steps*, $MS(m_{ij})$, of a method are those which correspond to method invocations. ■

The set of all steps S_{ij_x} in a method m_{ij} is denoted $STEPS(m_{ij}) = LS(m_{ij}) \cup MS(m_{ij})$.

Definition 3.15 Within method m_{ij}, all steps, $STEPS(m_{ij})$, are related by a total order $(STEPS(m_{ij}), \prec_s)$. The *expected serial execution order*, $\prec_s$, orders every pair of

steps in $STEPS(m_{ij})$ in the order they would execute in a serial execution of the method. ■

A serial method execution fails to exploit potential inter-step concurrency including message step (i.e., sub-transaction) concurrency. To capture the potential inter-step concurrency, the following partial order is introduced.

Definition 3.16 All steps $STEPS(m_{ij})$ are also related by a less restrictive *partial* order $(STEPS(m_{ij}), \prec)$ induced by the *data dependencies* between the steps. ■

To preserve correctness, any valid concurrent execution of the steps in a method must respect $\prec$. This is equivalent to saying that any valid execution ordering is serializable. The use of dependencies provides a means of statically determining which execution order(s) of the steps within a method will be serializable.

Two types of steps are distinguished and used in the definition of data dependencies as they apply to object methods and in the definition of $\prec$. These are *accessor* and *mutator* steps.

Definition 3.17 An *accessor* step is defined to be either a local step which reads an attribute value or a message step which uses an attribute value as an input parameter. ■

Definition 3.18 A *mutator* step is defined to be either a local step which assigns to an attribute value or a message step which specifies an attribute as one of its output parameters. ■

Three basic forms of data dependence are defined:

Definition 3.19 *True dependence* $(S_{ij_x} \delta S_{ij_y})$ occurs between two steps if S_{ij_x} is a mutator step which precedes (according to $\prec_s$) an accessor step, S_{ij_y}, and S_{ij_x} updates and S_{ij_y} accesses the same object attribute. ■

Definition 3.20 *Anti dependence* ($S_{ij_x}\overline{\delta}S_{ij_y}$) occurs between two steps if S_{ij_x} is an accessor step which precedes (according to $\prec_s$) a mutator step, S_{ij_y}, and S_{ij_x} accesses and S_{ij_y} updates the same object attribute. ■

Definition 3.21 *Output dependence* ($S_{ij_x}\delta^o S_{ij_y}$) occurs between two steps if S_{ij_x} is a mutator step which precedes (according to $\prec_s$) a mutator step, S_{ij_y}, and both S_{ij_x} and S_{ij_y} update the same object attribute. ■

In certain cases the form of dependence(s) is not important. This gives rise to the following definitions:

Definition 3.22 An *arbitrary direct* dependence ($\delta^?$) between steps in a method occurs if there is a true, anti, or output dependency between the steps. ■

Definition 3.23 *Arbitrary, indirect data dependence* (δ^*) is the transitive closure of the arbitrary direct dependence relation ($\delta^?$). ■

Two steps S_{ij_x} and S_{ij_y} are arbitrarily, indirectly dependent (subsequently referred to simply as "indirectly dependent") if there is a chain of arbitrary direct dependencies between them (i.e., $S_{ij_x}\delta^* S_{ij_y} \equiv S_{ij_x}\delta^? S_{ij_{s_1}}\delta^? S_{ij_{s_2}}\delta^? ... \delta^? S_{ij_{s_l}}\delta^? S_{ij_y}$ for some $l \geq 0$).

If two steps S_{ij_x} and S_{ij_y} are related by $S_{ij_x}\delta^* S_{ij_y}$ then $S_{ij_x}, S_{ij_y} \in OPS(m_{ij})$ are related by the partial order $(OPS(m_{ij}), \prec)$.

3.3.3 Definitions Related to Concurrency Control

To perform concurrency control it is necessary to identify the object attributes read and written by each object method. The read and write sets of individual method steps are captured in the following definitions:

Definition 3.24 The *readset* of method step S_{ij_k} in method m_{ij} is $RS(S_{ij_k}) = \{a_{ix} \mid a_{ix}$ is read by $S_{ij_k}\}$. ■

Definition 3.25 The *writeset* of method step S_{ij_k} in method m_{ij} is $WS(S_{ij_k}) = \{a_{ix} \mid a_{ix}$ is written by $S_{ij_k}\}$. ■

Membership in the read (write) sets for local steps is obvious. For message steps, the read set contains those attributes used as input parameters in the corresponding method invocation while the write set contains those attributes used as output parameters.

The read and write reference sets of a method are conservatively defined as:

Definition 3.26 The *read reference set* of method m_{ij} is $\mathcal{RR}(m_{ij}) = \bigcup RS(S_{ij_n})_{1 \leq n \leq t_j}$[5]. ■

Definition 3.27 The *write reference set* of method m_{ij} is $\mathcal{WR}(m_{ij}) = \bigcup WS(S_{ij_n})_{1 \leq n \leq t_j}$. ■

To perform concurrency control it is also necessary to know which objects are logically a *part-of* other objects (i.e., those objects which are referenced from a given object). This information determines whether or not two method invocations on distinct objects may conflict due to access to a third *shared* object. The following definitions are presented to address this possibility.

Definition 3.28 A message step within a method *directly invokes* the object method specified in the message step. ■

Definition 3.29 A message step within a method may *indirectly invoke* those object methods which are directly or indirectly invoked by the object method it directly invokes. ■

[5] t_j comes from Definition 3.11

Definition 3.30 The *extent* of a message step consists of all object methods that may be directly or indirectly invoked by its execution. ■

Definition 3.31 A message step's *reachable set* consists of those objects containing one or more methods in the extent of the message step. ■

These definitions are extended to apply to entire methods:

Definition 3.32 A method *directly invokes* the object methods which may be directly invoked by the message steps contained in it. ■

Definition 3.33 A method *indirectly invokes* those object methods which may be directly or indirectly invoked by the object methods it directly invokes. ■

Definition 3.34 The *extent* of a method consists of all object methods that may be directly or indirectly invoked by the method's execution. ■

Definition 3.35 A method's *reachable set* those objects which contain one or more methods in the extent of the method. ■

For notational convenience, sets with the same names as the preceding definitions are assumed to exist. For example, $DirectlyInvokes(m_{ij})$ is the set of methods directly invoked by m_{ij}.

3.4 Objectbase Concurrency

Recall that a transaction is a method execution and a user transaction is a method execution that is invoked *directly* by a user. User transactions, transactions, and subtransactions are all method executions. Therefore, the terms method execution and transaction (referring to any form of transaction) will be used interchangeably.

Concurrency in transaction execution against an objectbase can occur in several forms. Three obvious levels of potential concurrency are identified.

Definition 3.36 *Coarse-grained concurrency* arises due to the availability of multiple, concurrent user transactions. ■

Definition 3.37 *Medium-grained concurrency* results from the concurrent execution of sub-transactions (invoked by concurrent message steps). ■

Definition 3.38 *Fine-grained concurrency* occurs between non-message steps within a method. ■

In this dissertation only medium and coarse granularity concurrency are addressed.

An objectbase concurrency control mechanism should support concurrent method executions that arise both between and within multiple user transactions whenever correctness constraints make it possible.

Definition 3.39 *Inter-transaction concurrency* occurs between different user transactions and thus corresponds to exploitation of coarse-grained concurrency. ■

Definition 3.40 *Intra-transaction concurrency* occurs between the sub transactions of a single user transaction and corresponds to exploitation of medium-grained concurrency. ■

To maximize concurrency both intra- and inter-transaction concurrency must be considered.

In the following sections it is assumed that, for each method, dependence analysis has been performed to derive the partial order $(STEPS(m_{ij}), \prec)$. Thus, for any two steps S_{ij_x} and S_{ij_y} in a method, it is known whether $S_{ij_x} \prec S_{ij_y}$, $S_{ij_y} \prec S_{ij_x}$, or $S_{ij_x} \not\prec S_{ij_y}$[6].

[6]The notation $S_{ij_x} \not\prec S_{ij_y}$ implies that S_{ij_x} and S_{ij_y} are not related by $\prec$.

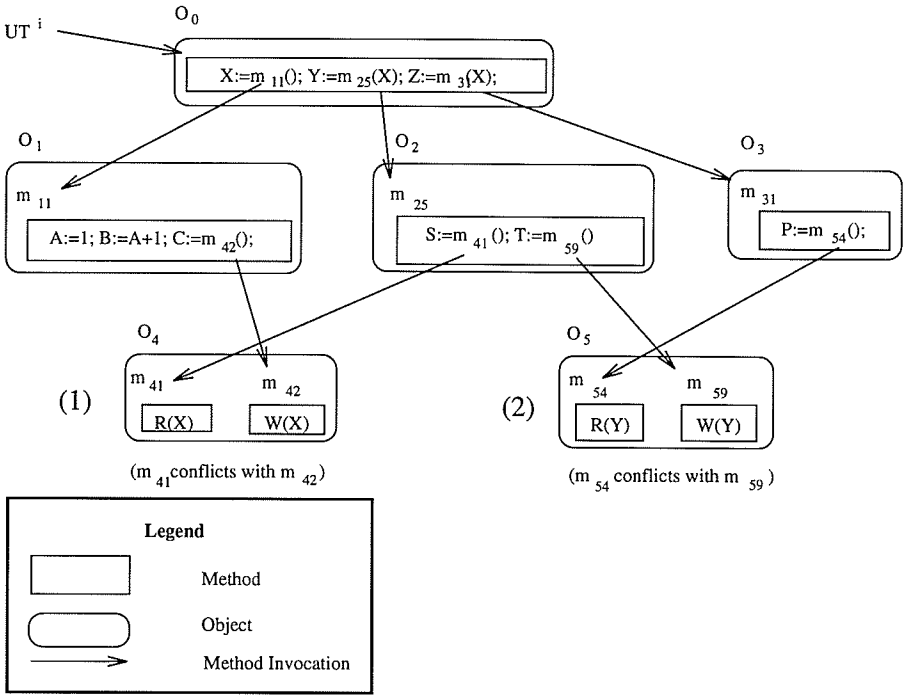

Figure 3.3: Intra-Transaction Concurrency Example

3.4.1 Intra-Transaction Concurrency

Trivially, intra-transaction concurrency can only occur within non-leaf methods. A method containing at least a single message step may permit concurrency. Concurrency can occur between a transaction and one or more of its sub-transactions or between the sub-transactions alone.

All local steps which are independent of a given message step may be executed concurrently with it. This observation permits specific parts of the local execution of a transaction to be "overlapped" with the execution of its independent sub-transactions.

Method local dependency between two message steps precludes all related sub-transaction concurrency. Such sub-transactions must be executed sequentially in an order consistent with $\prec$ to ensure equivalence to serial execution. The lack of such dependencies between message steps does not guarantee that the corresponding sub-transactions can safely be

executed concurrently. If object sharing is permitted, two independent message steps may invoke methods on the same object and if those methods access object attributes in a conflicting manner then uncontrolled concurrency is unsafe. This problem can be generalized as follows; "Two independent message steps within a method cannot be allowed to execute concurrently if their reachable sets of objects are non-disjoint and if at any object in the intersection of their reachable sets the corresponding method invocations (given in the related extents) access object attributes in a conflicting manner."

Figure 3.3 illustrates the fundamental cases which may arise in intra-transaction concurrency. The two cases of interest are marked '(1)' and '(2)' in the figure and occur when objects are shared.

In case '(1)' although object O_4 is shared indirectly by method invocations made from user transaction UT^i there is no possibility of conflict since the second method invocation in UT^i is true dependent on the first (based on accesses to attribute X) and therefore will not be executed until after it has completed. Thus, method-local dependencies preclude the need to check for conflicts occurring in shared objects at lower levels.

In case '(2)' however there is no dependency-induced ordering between the method invocations on O_2 and O_3. To maximize concurrency (and with only method-local information) these invocations should be made concurrently. The conflict that occurs at object O_5 must be detected so that the system can guarantee that the method invocation arising from O_2 can be executed before that arising from O_3. This is the execution order that must be satisfied to ensure serializability.

It is possible for multiple objects to be shared at various levels of calls arising from a single user transaction. Whenever related, concurrent method invocations (i.e., those arising from the same user transaction) occur at *any* object the appropriate serialization order must be enforced. Thus, for example, conflicting method executions at objects O_3 and O_6 (in Figure 3.4) must be scheduled so their execution order is the same as that of their ancestors in the original user transaction. It is possible that no dependency exists

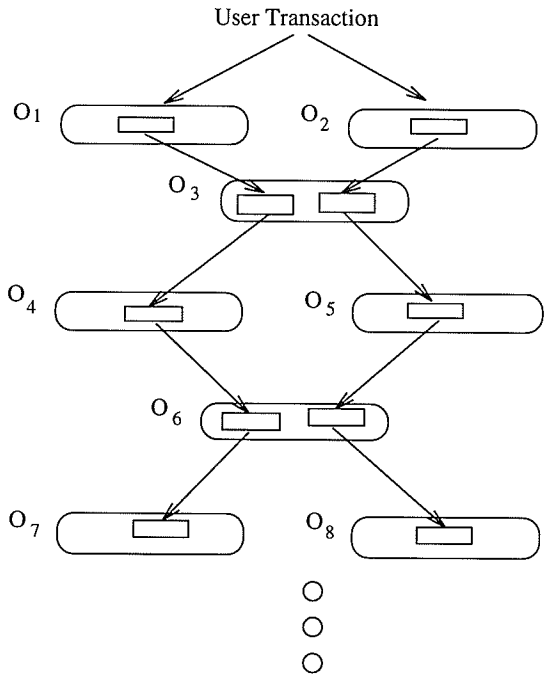

Figure 3.4: Multiple Shared Objects and Intra-Transaction Concurrency

between the relevant method invocations in the user transaction but that a conflict occurs at one or more shared objects. In this case, it is still necessary to enforce the serialization order determined by the order of method invocations in the user transaction consistently at all shared objects.

3.4.2 Inter-Transaction Concurrency

A fundamental difference between method executions arising from within and between transactions is that those between transactions never have a prescribed order of execution pre-determined. Serializability at this level, as in conventional database systems, simply prescribes that execution be equivalent to *some* serial execution. While this does permit greater potential concurrency it also makes the problem of controlling concurrency more difficult and one that necessitates involvement of a run time scheduler.

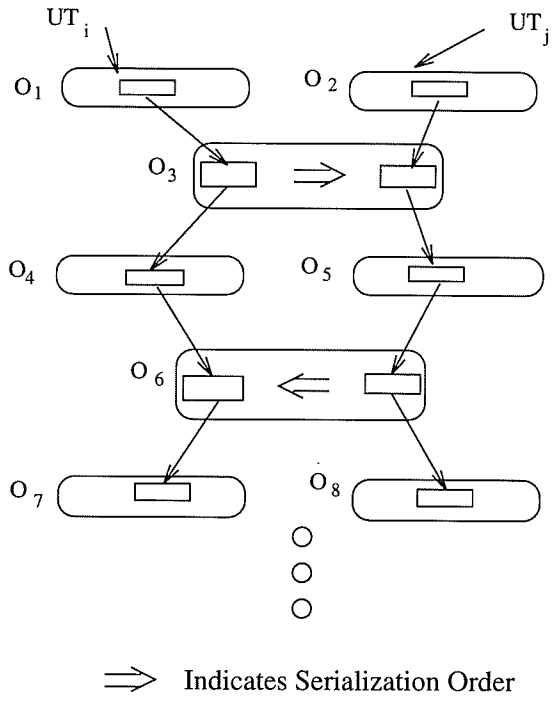

Figure 3.5: Multiple Shared Objects and Inter-Transaction Concurrency

When concurrent method invocations OT_{ij} and OT_{ik} originating from *different* user transactions execute at object O_i they can be serialized so that their execution is equivalent to either $OT_{ij} \Rightarrow OT_{ik}$ or $OT_{ik} \Rightarrow OT_{ij}$. The increased difficulty in providing concurrency control arises because the serialization order selected at the first object shared between two user transactions must be adhered to at *all* objects so shared.

Consider the situation illustrated in Figure 3.5. This represents a serialization error since at O_3, UT_i is serialized before UT_j while at O_6, UT_j is serialized before UT_i. Any correct concurrency control algorithm must ensure that the serialization orders at O_3 and O_6 are consistent.

Consistent serialization orderings across objects can be achieved using either static or dynamic local atomicity properties as described by Weihl [Wei89b]. Algorithms employing static techniques offer the advantage of pre-determined ordering of conflicting

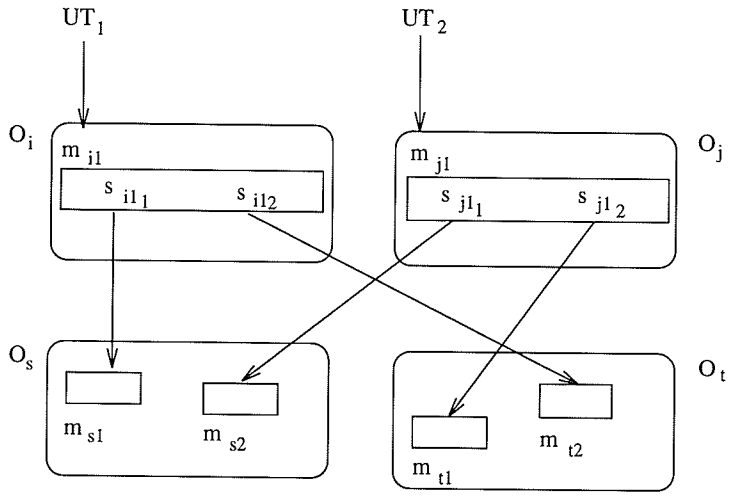

Figure 3.6: Inter-Transaction Concurrency and Potential Deadlock

operations but only at the cost of potentially decreased concurrency. Timestamp and order-preserving methods fall into this class. Algorithms based on dynamic techniques do not restrict concurrency *à priori* but are prone to deadlock.

An example of how deadlock can arise can be seen in Figure 3.6. If a two-phase locking scheme is used on object attributes then deadlock may occur as follows. Assume that m_{s1} and m_{s2} conflict as do m_{t1} and m_{t2}. If at O_s, m_{s1} obtains the necessary execution lock on behalf of UT_1 and at O_t, m_{t1} obtains its lock on behalf of UT_2 then deadlock will result. UT_2 will be forced to wait at O_s and UT_1 will be forced to wait at O_t and neither can proceed due to locks held by the other. Note that deadlocks correspond to serialization errors when using two-phase locking.

As with intra-transaction concurrency, method local dependency precludes all sub-transaction concurrency resulting from two dependent message steps within a method. This can be seen by once again referring to Figure 3.6. As previously described, for the deadlock/serialization-error to occur, message steps S_{i1_1} and S_{i1_2} and S_{j1_1} and S_{j1_2} must be independent so they can execute concurrently. If dependencies exist, for instance, between S_{i1_1} and S_{i1_2} then a serialization error cannot occur. Note however that deadlock

will still occur in this situation. Not all deadlocks reflect serialization errors.

3.4.3 Multi-Version Concurrency

Preventing concurrency between conflicting operations, even in the least restrictive way possible, still fails to address the issue of supporting long-lived transactions. The only way to permit concurrent conflicting access is through the use of optimistic concurrency control protocols. Unfortunately, in the conventional optimistic scenario all but one conflicting transaction must be rolled back. This results in the loss of the associated work and is unacceptable in a CAD environment. A significant step towards addressing this issue is achieved using multiversion objects to support optimism and reconciliation to avoid roll back [GB94].

To discuss multi-version objects the definition of an object is extended to include a unique version identifier which distinguishes different versions of the same object.

Definition 3.41 An *object version* $V_{ij} = (S_{ij}, B_i)$ where:

1. i is the object identifier,
2. j is the version identifier,
3. S_{ij} is the object version's structure composed of attributes such that $\forall a_{ij}, a_{ik}(j \neq k) \in S_{ij},\ a_{ij} \neq a_{ik}$,
4. B_i is the object version's behaviour composed of methods such that $\forall m_{ij}, m_{ik}(j \neq k) \in B_i,\ m_{ij} \neq m_{ik}$. ■

Points (1) and (2) uniquely identify a version and the object to which it belongs. Point (3) specifies the attributes of the object version and Point (4) specifies the methods of the object.

To avoid the high cost of transaction roll back and re-execution when conflicting operations optimistically update their own versions of an object it must be possible to

"correct" the execution of the transaction which would be rolled back. The correction process is referred to as *reconciliation*.

Given two conflicting methods m_{i1} and m_{i2} in a class and a serialization order, $m_{i1} \Rightarrow m_{i2}$, a reconciliation procedure reconciles (i.e., corrects) the results produced by an execution of m_{i2} so that they reflect the results which would have been obtained if m_{i2} had operated on a version of the object derived from the version which results from m_{i1}'s commitment. For reconciliation to be worthwhile, it must be possible to statically generate these reconciliation procedures and the cost of applying them must be less than that of roll back and re-execution of one of the conflicting transactions.

3.5 Applying Static Information to Concurrency Control

Several problems exist with conventional approaches to providing object base concurrency control. Concurrency control mechanisms in existing objectbases normally support only coarse granularity concurrency based on object-level locking [GK88, HK89, KGBW90, Deu90, WLH90, HCL+90]. Additionally, no attempt is made to exploit semantic information about objects or the inherent inter-object structure in the objectbase to optimize concurrency control. Furthermore, despite the potentially drastic variation in behaviour of different objects, normally a *single* concurrency control policy is applied to *all* objects.

Fully exploiting the available concurrency in objectbases requires a nested transaction approach (to permit the exploitation of medium-granularity concurrency). For complex objects this results in a significant increase in concurrency. Rather than force the objectbase programmer to explicitly provide code to support nested transactions, simple static analysis can be used to automate the process.

Conventional run-time methods, based on a transaction's dynamic read/write behaviour alone, fail to exploit semantic information that can be derived by the method compiler and other, pre-runtime, components of the objectbase system. Such static anal-

ysis can be applied to derive semantic information which can be used to improve the level of concurrency obtainable in several ways. Among other applications, it can be used to increase the level of medium granularity concurrency and also to further decrease the granularity of concurrency beyond that provided by a nested transaction scheme. Static analysis can be applied to virtually any program component but throughout the dissertation, application on a *per-method basis* is assumed unless explicitly stated otherwise.

The availability of semantic information is also the basis for permitting *different*, object-specific concurrency control protocols to be used at each object. This leads to a two-level concurrency control paradigm where object-local concurrency control is provided by potentially different but globally compatible protocols constrained by the inter-object concurrency control policy. This follows the approach of Hadzilacos and Hadzilacos [HH91] and Zapp and Barker [ZB93c, ZB93a].

In terms of the formalism introduced earlier in this chapter, concurrency control algorithms must examine the extents of two methods and check for conflicts between method executions occurring in the intersection of those extents. For example, in Figure 3.7 the shaded area denotes the intersection of the extents of the method invocations made from statements S_{ij_x} and S_{ij_y}. If any of the method executions in the intersection conflict then so do the executions arising from S_{ij_x} and S_{ij_y}. The example illustrates intra-transaction concurrency but the concept also applies to inter-transaction concurrency. The somewhat vague notion of conflicts arising in extent intersections will now be formalized.

It must be possible to statically determine if two concurrent method executions m_{i1} and m_{i2} *could* result in a conflict which would produce incorrect results. Such a conflict can arise in two possible ways:

- The sets of object attributes referenced by the two methods may be non-disjoint (indicating possible local conflict).
- The sets of objects containing methods invoked by m_{i1} and m_{i2} may be non-disjoint (indicating possible non-local conflict).

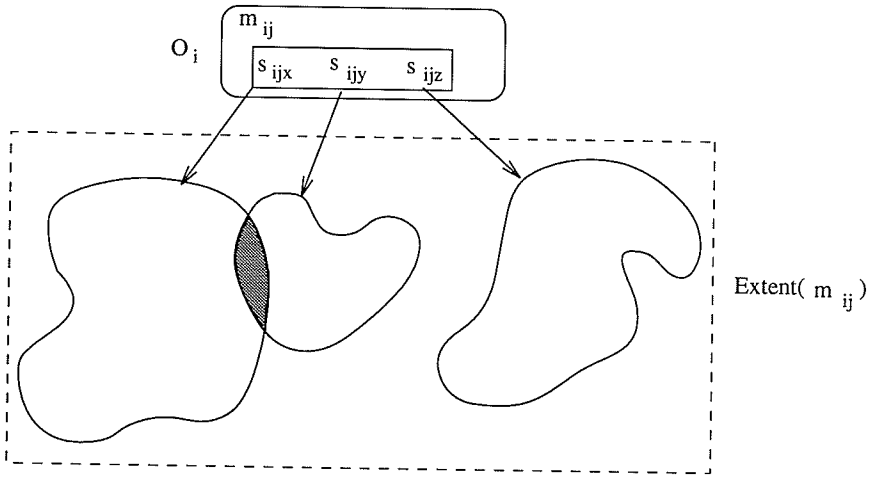

Figure 3.7: Extent Intersection

Attribute access information is captured by the definitions of the read reference set, $\mathcal{RR}$, (Definition 3.26) and the write reference set, $\mathcal{WR}$, (Definition 3.27) of a method. It is simple to also construct sets that summarize the method reference information.

Definition 3.42 The *direct method reference set* of method m_{ij} is:

$\mathcal{MR}(m_{ij}) = \{\ m_{kl} \mid m_{kl}$ is a method invoked by some message step $S_{ij_k} \in m_{ij}\ \}$ ■

Definition 3.43 The *transitive method reference set* of method m_{ij} is:

$\mathcal{MR}^*(m_{ij}) = \mathcal{MR}(m_{ij}) \bigcup_{p \in \mathcal{MR}(m_{ij})} \mathcal{MR}^*(p)$ ■

The set $\mathcal{MR}^*$ represents the, possibly indirect, method references by enumerating all *methods* invoked and is equivalent to the methods *extent* (Definition 3.34).

The sets $\mathcal{RR}$, $\mathcal{WR}$, $\mathcal{MR}$, and $\mathcal{MR}^*$ are collectively called a method's *reference* sets. Using reference sets, conflict between two methods (m_{i1} and m_{i2}) can be conservatively detected using the test:

$$(\mathcal{RR}(m_{i1}) \cap \mathcal{WR}(m_{i2}) \neq \phi) \wedge (\mathcal{WR}(m_{i1}) \cap \mathcal{RR}(m_{i2}) \neq \phi) \wedge$$

$$(\mathcal{WR}(m_{i1}) \cap \mathcal{WR}(m_{i2}) \neq \phi) \wedge (\mathcal{MR}^*(m_{i1}) \cap \mathcal{MR}^*(m_{i2}) \neq \phi)$$

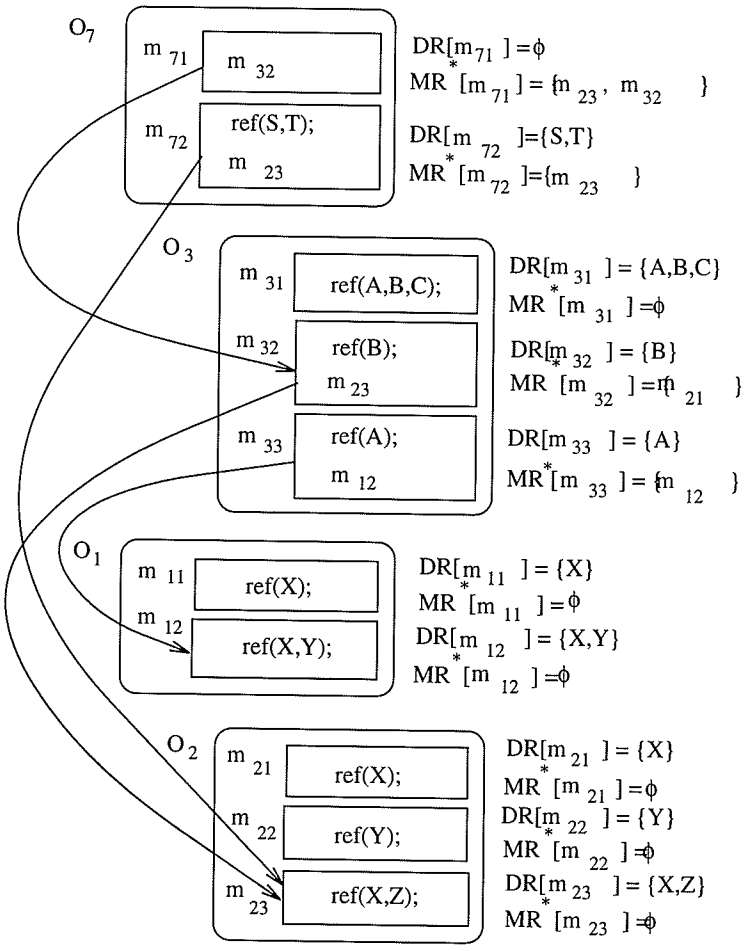

Figure 3.8: Reference Set Example

Using this notation, any method with $\mathcal{MR}(m_{ij}) = \phi$ is a *leaf method*.

An example of the reference sets of the methods in some sample objects is given in Figure 3.8. For the sake of simplicity, only a single *data* reference set ($\mathcal{DR} = \mathcal{RR} \cup \mathcal{WR}$) is shown. The $\mathcal{DR}$ sets for each method contain the set of object attributes that those methods "reference". Each object is depicted as a box containing abstractions of the steps in the object's methods. The arcs in the diagram indicate method invocation paths and the $\mathcal{DR}$ and $\mathcal{MR}^*$ sets for each method in each object are shown. Thus, for example, the $\mathcal{MR}^*$ set for m_{71} consists of m_{32} which is called directly and m_{23} which is called indirectly through m_{32}. The $\mathcal{MR}^*$ set for m_{11} is empty making it a leaf method.

The same information used to detect conflicting method executions provides the basis for determining and producing reconciliation procedures. If two methods in an object do not conflict, then their executions, arising from two transactions, can be serialized in any order. Furthermore, if both method's are read-only (with respect to object attributes referenced) then no new object version need be produced when the corresponding transactions are committed. If one or both of the methods update object attributes then the reconciled object must contain the updated attribute values produced by both method executions.

If two methods conflict then reconciliation is more complex since at least part of the execution of the transaction serializing last must be re-executed. The part which must be re-executed consists of those statements which depend on values written by the conflicting method execution which serializes first. This is a subset of the entire method which can be determined automatically at compile time using static analysis.

Chapter 4

Deriving and Representing Static Information

To use static information to enhance concurrency in objectbases, it must be possible to efficiently derive and represent the information in a compact, usable form. This chapter begins by describing the required static information. It then describes simple set and directed graph representations for the information, some of which are preserved and stored with the relevant class/object(s) for use in scheduling method executions at run time. Finally algorithms which extract the static information from (1) method specifications at compile time and, (2) inter-object relationships determined at object instantiation time are presented.

4.1 The Required Static Information

The basis for the static information needed (Section 3.5) can be directly determined by analyzing class methods using efficient *syntax directed* [ASU86] techniques and by analyzing the structure of objects within the objectbase.

The representations chosen for the static information must be capable of encoding three types of information:

1. **Control flow information** – detailing which sections of code are executed in what order and under what conditions.
2. **Method invocation information** – detailing the calling sequence between objects.
3. **Attribute reference information** – detailing the order in which attributes are referenced in a method and their read/write relationships.

Control flow information is needed to determine the serializability of method executions. A concurrent method execution is deemed correct if it is equivalent to a serial execution of the same method given the same initial state and identical input parameters. This equivalence is judged by comparing the execution orders of the concurrent and serial executions. The total order $\prec_s$ of a serial execution and the partial order $\prec$ (determined by dependency analysis) are both affected by control flow.

Method invocation information is required to determine each method's extent and reachable set of objects. At compile time, this information cannot be determined. Implicit in the class specifications is information detailing only the *classes* of objects each method will invoke methods on. The exact objects which a given *object* method will invoke methods on, are unknown until all the relevant objects have been instantiated. At compile time, the *form* of a method's extent can be determined but only once the objects are instantiated can the *contents* of the extent be determined.

Attribute reference information is required to determine when methods conflict. This is because conflict is based on common attributes being referenced in conflicting ways by two methods of an object. Attribute reference information is also required to produce the dependence relations needed for various purposes in enhancing concurrency. Dependence information, like control flow information, is fundamental to determining serializability since the partial order $\prec$ is derived using it. Dependence is also the basis for generating reconciliation procedures for optimistic, multi-version, object concurrency control.

```
IF (Expr1) THEN
      Sx
ELSE
      WHILE (Expr2) DO
            Sy
      ENDWHILE
ENDIF
```

Figure 4.1: Example Code containing Control Flow Statements

4.2 Representing Static Information

The needed static information can be represented using attribute reference sets and control flow, method invocation, and data dependence graphs. This section formalizes these concepts and discusses them as they relate to concurrency control. The data structures described are developed for use on a per-method basis although they may have wider applicability.

4.2.1 Representing Control Flow Information

Definition 4.1 A *basic block* is a code segment with a single entry point that is devoid of branches with the possible exception of the final statement in the block. ■

Intuitively, a basic block is a straightline segment of code where, if the first statement is executed, all subsequent statements in the block are also executed. As a first step towards representing the necessary static information, each method specification will be divided into its constituent basic blocks using the conventional "leader" algorithm [ASU86].

Definition 4.2 The *control flow graph* of a method m_{ij} is a directed graph $CFG(m_{ij}) = (V, E)$ with a vertex $v_x \in V$ for each basic block, B_x, in m_{ij} and a directed edge $e \in E$ from vertex v_x to vertex v_y iff method execution may pass directly from B_x to B_y. ■

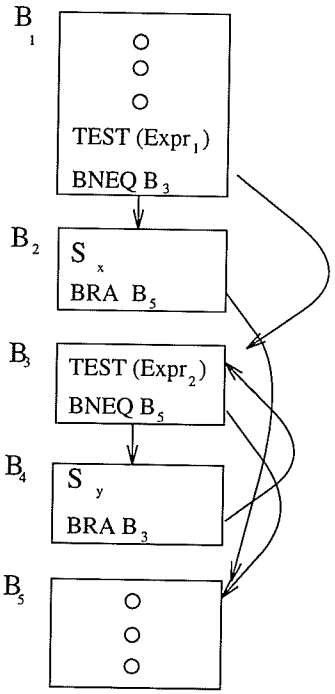

Figure 4.2: Control Flow Graph for Code in Figure 4.1

For conventional imperative programming languages, the detection of control flow is easily accomplished in a simple, syntax-directed fashion. An example control flow graph for the code sequence shown in Figure 4.1 is given in Figure 4.2. The statements within each basic block which are unrelated to control flow are abstracted away.

The details of how basic blocks are encoded is largely irrelevant to the dissertation. Significant aspects are discussed in Section 4.2.3. From this point forward, it is assumed that the control flow graph of each method in every class definition is available for analysis.

4.2.2 Representing Method Invocation Information

Conventional inter-procedural analysis uses the *call graph* [Ryd79, CCHK90] to record the procedure calling relationships between the procedures in a program.

Definition 4.3 The *call graph* of a program is a directed graph $CG = (V, E)$ containing a vertex $v_i \in V$ for each procedure in the program and a directed edge $e \in E$ from vertex v_i to vertex v_j iff procedure v_i calls procedure v_j. ■

All method invocations are made from within other methods so any data structure summarizing method calls must be associated with each class method. The call graph of definition 4.3 does not suit the objectbase environment because not all relevant methods are known at compile time. Inter-object calls occur and it is impossible to know at compile time specifically which objects are referred to in the method invocations. Thus, the equivalent of a call graph (referred to as the *object call graph*) cannot be built until object instantiation time. A formal definition of the object call graph and discussion of its instantiation time construction will be provided shortly.

The primary application of method call information is to determine if two method executions may conflict at some object due to (possibly indirect) invocations of conflicting methods on that object. Analysis of the object call graph (Definition 4.5) can make this determination. In certain cases however, information derived strictly at compile time can also be useful. If the classes of methods invoked by two method executions are distinct then there cannot be a conflict. This is obviously true since any object belongs only to a single class. Checking the class call graph (Definition 4.4) at compile time and recording the result is a useful optimization since it can preclude the need for run-time checking.

The construction of the class and object call graphs is only done for method invocations made on objects which exist as distinct entities having unique object identifiers. This does not address all object method invocations. There are two "degenerate" cases. The first case occurs when the object containing the method being invoked is a physical part of the object invoking it. It is then assumed that the sub-object is instantiated at compile time within the enclosing object as an *atomic* attribute. The atomicity assumption precludes intra-transaction concurrency between objects and such sub-objects contained within them. The second case where method call information is not generated is for calls

to object-local methods. Such calls are assumed to be inline expanded and thus treated as straightline code rather than method invocations. This too precludes intra-transaction concurrency. These assumptions are not necessary but simplify the presentation of the algorithms discussed later by eliminating these special cases.

The construction of the class call graph assumes that no class definition can be made which references objects belonging to an undefined class. This declaration before use rule is common in all strongly statically typed languages.

Definition 4.4 The *class call graph* of method m_{ij} (in *class* C_i) is a directed graph $CCG(m_{ij}) = (V, E)$ containing a root vertex $v_{root} \in V$ corresponding to m_{ij} and directed edges $e \in E$ from v_{root} to those vertices $v_x \in V$ which are the root nodes of the class call graphs corresponding to the methods invoked directly by m_{ij}. ■

The class call trees that result from the statically determined calling behaviour of class methods effectively chains the class definitions to reflect their calling relationships. This means that a class call graph $CCG(m_{ij})$ will be a sub-graph of some other class call graph $CCG(m_{xy})$ iff m_{xy} calls m_{ij} (directly or indirectly).

The object call graph is similar to the class call graph except it threads actual objects and is constructed when objects are instantiated not at compile time. For the sake of simplicity, the definition before use rule applied to the construction of the compile time call structure is also assumed for the run time structure. This restriction, which prevents dynamically created objects, can be relaxed.

Definition 4.5 The *object call graph* of method m_{ij} (in *object* O_i) is a directed graph $OCG(m_{ij}) = (V, E)$ containing a root vertex $v_{root} \in V$ corresponding to m_{ij} and directed edges $e \in E$ from v_{root} to those vertices $v_x \in V$ which are the root nodes of the object call graphs corresponding to the methods invoked directly by m_{ij}. ■

For discussion in this dissertation, the "child-of" relationship is extended to include objects "called by" another object. Thus, the object call graph records the extended child-of relationship for objects in the objectbase.

Traversing the class (respectively, object) call graph permits the construction of sets of referenced class (object) methods for the corresponding class (object). Discussion of such sets is postponed until Section 4.3.2.

4.2.3 Representing Attribute Reference and Dependence Information

The definitions of data dependence in a sequential method specification were presented in Section 3.3.2. Attribute reference information is used together with control flow information to permit the construction of dependence graphs which reflect the dependence relations. In addition to indicating how attribute reference information is stored, this section will also discuss how the information can be threaded to represent data dependences.

Recall that basic blocks contain a sequence of non-control flow steps possibly followed by a single step affecting control flow. The non-control flow steps in a basic block are of three types:

1. Assignment steps
2. I/O steps
3. Method invocation steps

Each of these types of steps may involve several attribute references. To minimize unnecessary dependencies, it is desirable to reduce these steps to their simplest possible forms. Input/output steps and method invocations are non-decomposable. A complex assignment step however, should be decomposed[1] into a number of simpler, and hopefully independent, assignment steps. For example the assignment step shown on the left

[1]Such decomposition is done prior to the division of each method into its basic blocks.

Original Step	Decomposed Steps

`x := (a+b)/(c-d)+e*a;`	`t1 := a+b;`
	`t2 := c-d;`
	`t3 := e*a`
	`t4 := t1/t2 ;`
	`x := t4+t3 ;`

Table 4.1: Decomposition of Complex Assignments

of Table 4.1 might be replaced with the sequence of assignment steps shown on the right. Note that some of the resulting "sub"-assignments are independent of one another and therefore could be executed concurrently (subject to efficiency and other considerations). Concurrency between local (assignments and input/output) steps is not considered, but concurrency between local and message (method invocation) steps is addressed (this is parent/sub-transaction concurrency). This decomposition may permit earlier scheduling of message steps (i.e., sub-transactions) and thereby increase concurrency.

Each basic block consists of these minimal, decomposed steps (Definition 3.11). It is assumed that dead-code elimination [ASU86] has been performed on the resulting basic blocks. This ensures that only the useful steps in a method contribute to the construction of its attribute reference information and ultimately may prevent the detection of non-existent conflicts. For each such step S_{ij_k} a read set $RS(S_{ij_k})$ and write set $WS(S_{ij_k})$ of object attributes are maintained. Similarly, for each basic block, BB_x, in m_{ij}, a read set ($RS(BB_x)$) and write set ($WS(BB_x)$) are maintained. The construction of the read and write sets for each basic block is captured by the following equations (where $STEPS(BB_x)$ is the set of steps in basic block BB_x).

$$RS(BB_x) = \bigcup_{k \in STEPS(BB_x)} RS(S_{ij_k})$$

$$WS(BB_x) = \bigcup_{k \in STEPS(BB_x)} WS(S_{ij_k})$$

These read and write sets are used to derive the inter-step and inter-basic block dependencies.

Definition 4.6 The *dependency graph* of a basic block BB_k in method m_{ij} is a directed graph $DG(BB_k) = (V, E)$ containing vertices $v_x \in V$ for each step S_{ij_x} in the basic block and a directed edge $e \in E$ from v_s to v_t iff $S_{ij_s} \delta^? S_{ij_t}$. ■

Applying this definition, a dependence graph within each basic block can be constructed which threads the steps together in dependence order (i.e., reflecting the partial order $\prec$) using the information in the per-step read and write sets and the definitions of dependence given in Section 3.3.2.

The read and write set information for basic blocks must be combined to provide summary information for each method. A simple but very conservative approach is to simply union the read (write) sets for all the constituent basic blocks. This process is captured by the following equations (where $BasicBlocks(m_{ij})$ is the set of basic blocks within m_{ij}).

$$RS(m_{ij}) = \bigcup_{x \in BasicBlocks(m_{ij})} RS(BB_x)$$

$$WS(m_{ij}) = \bigcup_{x \in BasicBlocks(m_{ij})} WS(BB_x)$$

The sets $RS(m_{ij})$ and $WS(m_{ij})$ are equivalent to $RR(m_{ij})$ and $WR(m_{ij})$ from Definitions 3.26 and 3.27, respectively, but are constructed differently.

In the presence of run time conditionals, these conservative definitions provide only *worst case* access information. In some cases this is sufficient so $RS(m_{ij})$ and $WS(m_{ij})$ are maintained. Greater accuracy can be achieved if information for each possible control flow path through the method can be statically derived, stored, and then selected appropriately at run time. The construction of this access information depends on the availability of control flow information.

It is only practical to select attribute reference information for a control flow path at runtime if the appropriate path can be determined *prior to* method execution. It must be possible to determine the appropriate control flow path based only on *existing* attribute values and input parameters which are available immediately before method execution. It must also be possible to enumerate each control flow path and the condition under which it is executed.

Unlike the conventional definition of an execution path through a routine (provided by the software engineering community [Som89][§21.2]) the control flow paths needed for this dissertation are only concerned with whether or not a basic block is visited, not how often it is visited. This means that the basic blocks in a sub-graph of the control flow graph corresponding to a loop structure need only be recorded *once* in any control flow path that visits the sub-graph. Therefore, the number of control flow paths in a method is simply 2^k where k is the number of control structures in the method[2].

Definition 4.7 A *control flow path* CFP_x through method m_{ij} is a sequence of basic blocks $< BB_{k_0}, BB_{k_1}, \ldots, BB_{k_{n-2}}, BB_{k_{n-1}} >$ where BB_{k_0} is the entry node in the control flow graph for m_{ij}, $BB_{k_{n-1}}$ is generally an exit node of the graph, and where there is an edge in the control flow graph for each pair of basic blocks BB_{k_l} and $BB_{k_{l+1}}$, $0 \leq l \leq n-2$. ■

Intuitively, this definition says that a control flow path is a sequence of basic blocks representing a valid execution path (according to the semantics of the source language) through a method from its entry to some point (normally its exit). There are many such control flow paths possible in any given method containing conditionals and/or loops. All the paths from method entry to any method exit are enumerated.

[2] For simplicity, only well-structured (e.g., `IF` and `WHILE`) control structures are considered. Unstructured control statements (e.g., `GOTO`s) are ignored as are more complex control structures (e.g., `CASE`) which can be decomposed into simple conditionals and loops.

Definition 4.8 The *control flow paths* of method m_{ij} are enumerated by a set $CFPs(m_{ij})$ of all possible control flow paths from the entry node to some exit node of m_{ij}. ■

Each basic block in the control flow graph of a method is executed under certain control flow conditions[3].

Definition 4.9 The *control flow condition* of a basic block BB_i is a boolean expression denoted $CFC(BB_i)$ involving attributes and/or method input parameters and/or method-local variables which must, at run time, evaluate to true if BB_i is to be executed. ■

Control flow conditions involving only attribute values which are *unmodified* by the execution of the method and/or method input parameters may be efficiently evaluated prior to method execution. This permits the scheduler to perform the evaluation and select an appropriate set of reference information based on the runtime value of the control flow condition.

A control flow path is executed only if the logical conjunction of the conditions of its constituent basic blocks is true. Thus, the execution condition for a control flow path is defined as follows:

Definition 4.10 The *control flow condition* of a control flow path CFP_i is a boolean expression which is the logical conjunction of the conditions of its constituent basic blocks: $CFC(CFP_i) = \bigvee_{\forall x \in CFP_i} CFC(BB_x)$ ■

Complex control flow conditions such as those containing method invocations or input statements must, conservatively, be assumed to always evaluate to true. This means that *all* attribute references made by basic blocks having such complex control flow conditions must be considered in conflict determination. By extrapolation, all attribute references

[3]Including, possibly none, in which case the block is always executed.

made by all basic blocks on control flow paths containing a basic block having a complex control flow condition must be considered in conflict determination.

Definition 4.11 The *control flow conditions* of method m_{ij} (in class C_i) are enumerated by a set $CFCs(m_{ij})$ containing a control flow condition $CFC(CFP_i)$ for each control flow path CFP_i in m_{ij}. ■

Read and write sets for individual control flow paths are also defined.

$$RS(CFP_i) = \bigcup_{x \in BasicBlocks(CFP_i)} RS(BB_x)$$

$$WS(CFP_i) = \bigcup_{x \in BasicBlocks(CFP_i)} WS(BB_x)$$

Attribute References to Objects in Collections

Discussion to this point has been limited to individual object attributes. In many cases, objects are grouped into structured collections such as arrays, lists, etc. It is advantageous to deal with such collections of objects as a whole or to treat their constituent objects specially when addressing concurrency control issues.

Conventional dependence analysis [Ban88, Wol89, ZC90] is focused on loop structures in which array variables are accessed. This is because the majority of the parallelism in a scientific program is found within loops operating upon arrays. Dependence analysis in objectbases is focused on *method*, not loop executions (recall that the dissertation is concerned only with medium-grain concurrency). Thus, loop based dependence analysis is required only when loops contain method invocations on object arrays. In this case, method invocations which are to distinct object elements in distinct loop iterations must be detected. When this happens, the method executions can be scheduled concurrently thereby exploiting intra-transaction concurrency.

```
MYCLASS  myobjects[100];
int      i,x,y;

y = 27;
for (i=0;i<100;i++) {
        x = f(y);
        myobjects[i].func(x, &y);
}
```

Figure 4.3: Parameter based Dependence

Three potential problems arise when considering method invocations on object elements within object arrays. First, access to distinct object elements does not guarantee freedom from conflicts at indirectly invoked objects if object sharing is permitted (as discussed in Section 3.4.1). Secondly, conventional dependence analyzers typically give up on loops containing subroutine calls since determining their *global* behaviour is too complex. This cannot be done when dealing with arrays of objects since every operation on the object elements is accomplished via a method invocation (i.e., a "subroutine" call). Fortunately, the encapsulation property of objects means that method invocations have *no* global side-effects[4]. Thirdly, since method invocations are parameterized, method invocations on *distinct* object elements may still need to be executed sequentially due to dependencies between *parameter* values. Consider the example code shown in Figure 4.3 where each iteration of the loop invokes a method on a distinct object array element but concurrency between iterations is still impossible because of the reference parameter y.

Conventional loop-based data dependence techniques can determine (albeit, sometimes imperfectly) whether data accesses within loops will conflict from iteration to iteration. As just illustrated, it is also necessary to be able to tell if there are dependences which arise due to the output parameters of one call being used as input parameters in another. This can be easily accomplished by integrating scalar and array dependence

[4] Aside from those arising due to shared sub-objects.

```
float       array[100],x;
int         i;

x = 27.4;
for (i=0;i¡100;i++) {
            array[i]:=array[i]+x;
            x:=array[i];
}
```

Figure 4.4: Integrated Scalar and Array Dependence

analysis. If an operation on an array element depends on a scalar and that scalar in turn depends on an array element, then integrating dependence analysis will detect the dependence[5]. This is no different than integrating scalar and array dependencies to detect interrelated dependencies in conventional code. Consider, for instance, the example shown in Figure 4.4 where the inter-iteration dependence involving x parallels that of y in the example of Figure 4.3.

Thus, the dependence graphs for scalar and array variables must be integrated into one, treating each array element as an individual data item. Furthermore, the construction of the dependence graphs must be based on the definitions in Section 3.3.2.

A sufficient condition for permitting the concurrent execution of method invocations (subject to shared sub-objects) within different iterations of a loop is that all dependences are *loop independent*.

Definition 4.12 A *loop-independent* dependence is one which involves only statements within a single iteration of the loop. ■

Definition 4.13 A *loop-carried* dependence is one involving statements in different iterations of the loop. ■

[5]This is the essence of the extended data dependence definitions given earlier.

One technique for expressing dependence in loops uses the *dependence distance* [ZC90] between two dependent references to an object array element in a loop. This is done by calculating the difference of the two subscript expressions involved. Thus for, references to a general d-dimensional object array, the following definition of dependence distance applies.

Definition 4.14 In general, the *data dependence distance vector* (or just *distance vector*) is defined to be $\Psi = (\psi_1, \psi_2, ..., \psi_d)$ where $\psi_k \in \mathbf{N}$. $S_v \delta_\Psi S_w$ is written when $S_v[i_1, ..., i_d] \delta^? S_w[j_1, ..., j_d]$ such that $\psi_k = i_k - j_k$ for $1 \leq k \leq d$. ■

Considering 1-dimensional object arrays, for simplicity, some concurrency may be possible in the presence of loop carried dependence as long as the minimum dependence distance between any two dependent steps in different iterations is greater than one. In this case, a subset of the loop iterations of size less than or equal to the minimum dependence distance can be concurrently scheduled. For example, if there are 100 loop iterations and a minimum dependence distance of 3 then iterations 1, 2, and 3 can execute concurrently and once they have completed, iterations 4, 5, and 6 can execute concurrently and so on. Even this limit on concurrency can be relaxed. Since iteration i+3 is dependent only on iteration i and since the concurrent method executions may take varying times to complete it may be worthwhile to permit, for example, iteration 4 to proceed once iteration 1 has completed although iterations 2 and 3 have not finished.

Recall that in scientific programming the fundamental data structure is the array. The well understood semantics of loops and operations on arrays within them has been exploited to permit dependence analysis for optimization/parallelization purposes. In an objectbase environment there is less focus on arrays as fundamental data structures. While important, other structures such as lists, queues, and indexes are equally, or perhaps more, important. Therefore, it would be beneficial to apply a modified form of dependence analysis to these structures. No work has appeared in the literature on this topic primarily because these structures are typically user defined rather than a builtin

part of the programming language. Thus, their semantics are less well understood and may be unpredictable.

Given a formal definition of a list structure, for example, and restricted operations on the list, it may be possible to extend dependence analysis to deal with the operations performed on these lists. Class specifications provide such formal definitions and this suggests that an objectbase environment may be ideal for exploiting dependence analysis on user defined classes. This requires the modification of (at least) the class compiler's behaviour based on the class definitions to be optimized. This interesting area of research is beyond the scope of the dissertation.

Without the in-depth analysis and dynamic compiler modifications just suggested, the standard limitations of conventional dependence analysis still apply. For example, the presence of pointers introduces the possibility of *aliasing*. Since pointer operations are not well defined, it is not always possible to determine what a pointer points at. In such cases conservative assumptions must be made. One benefit of the encapsulation properties of objects is that global variables (i.e., object attributes) do not present as great a problem as they do in conventional programming languages. Since their scope is limited, so too is their locality of effect – only the methods of the object can be affected by a change to an object attribute. This dissertation treats pointers conservatively.

Attribute References and Multi-Version Objects

Special attribute reference information is needed for reconciling concurrently produced versions of an object. To understand what new information is needed, reconciliation must be discussed. This requires some new definitions related to multi-version object concurrency control. In particular, certain special versions of objects must be identified.

Definition 4.15 The *latest committed version* (LCV) of an object is the version of an object which was created by the last *committed* transaction which updated the object. ■

Definition 4.16 Each transaction receives its own *active version* of an object when it first references the object. The active version is a copy of the latest committed version of the object which the transaction may subsequently modify. ■

Active versions are *derived* from the LCV of an object and the version from which an active version is derived is referred to as the *base version* of the object for the corresponding transaction. An active version becomes the new LCV when its corresponding transaction successfully commits.

Reconciliation is required when two transactions T^i and T^j execute concurrently at some object. When the first (say T^i) commits, its version becomes the LCV and, by serializability, the one which T^j should be using as its base version. Later, when T^j attempts to commit, its version of the object must be reconciled with the new *LCV* (which was T^i's active version) to compensate for the stale attribute values it used in its computation.

The stale attribute values used are those attributes which T^j read and which T^i wrote. Each transaction's execution on an object takes the form of a method execution. Thus, reconciliation is between pairs of object methods. For each class, for each pair of methods within the class, and for each possible serialization/commit order, a set of stale attributes must be determined.

Definition 4.17 Given two class methods m_{ix} and m_{iy} and the serialization order $m_{ix} \Rightarrow m_{iy}$, the set of stale attributes read by m_{iy} is

$$StaleAttrs(m_{ix}, m_{iy}, m_{ix} \Rightarrow m_{iy}) = RS(m_{iy}) \cap WS(m_{ix})$$

■

The set *StaleAttrs* specifies the stale attributes that were read. It does not specify what subset of method m_{iy}'s operations must be re-executed to achieve reconciliation (i.e., what the reconciliation procedure is). This information must also be represented.

A method specification is represented as a control flow graph of basic blocks each containing statements. A reconciliation procedure is simply a subset of a method and thus, its representation is a subset of the control flow graph (and its basic blocks).

4.3 Deriving Static Information

Some of the algorithms used for deriving the needed static information are well-understood. They are presented here for completeness with only the slight modifications required to produce the desired representations. It is assumed throughout this section that conventional compile-time optimizations, such as removing all loop-invariant code and eliminating unreachable code, which can affect the efficiency and/or results of the algorithms described have been performed.

4.3.1 Deriving Control Flow Information

The derivation of control flow information begins with the determination of basic blocks from a method specification. Algorithm 4.1 gives pseudo-code for determining basic blocks. The algorithm is taken from Aho, Sethi, and Ullman [ASU86][pp.529] where it appears as Algorithm 9.1. This algorithm assumes that the method specification being operated on has been decomposed into simple assignment steps and conditional and unconditional branches which reflect the control structure given in the original specification.

Algorithm 4.1 *Partitioning into Basic Blocks*

1. Determine the set of *leaders*, the first statements of basic blocks. The rules used are:
   (a) The first statement in a method is a leader
   (b) Any statement that is the target of a conditional or unconditional branch (arising as a result of control structures) is a leader
   (c) Any statement that immediately follows a branch (conditional or not) is a leader
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the method.

End of Algorithm

Once the basic blocks have been determined, the control flow graph linking the basic blocks can be constructed. This process is illustrated in Algorithm 4.2.

Algorithm 4.2 *CFG Construction*

```
INPUT : The basic blocks of a method;
OUTPUT : The basic blocks linked to form the CFG;

BBqueue : Queue of Basic Blocks;

FOREACH basic block in the method DO
    add the basic block as a vertex in the CFG;
EntryNode ← Basic Block containing the first method step;
insert EntryNode into BBqueue;
WHILE (BBqueue not empty) DO
    /* process blocks adding edges to CFG as appropriate */
    CurrentBlock ← remove(BBqueue);
    SWITCH (LastStmt in CurrentBlock) IN
    CASE unconditional branch:
        add an edge from CurrentBlock to block referenced in branch;
        insert referenced block into BBqueue;
    CASE conditional branch:
        add an edge from CurrentBlock to block referenced in branch;
        insert referenced block into BBqueue;
        add an edge from CurrentBlock to following block;
        insert following block into BBqueue;
    OTHERWISE :
        IF (CurrentBlock is not an exit block) THEN
            add an edge from CurrentBlock to following block;
            insert following block into BBqueue;
    ENDSWITCH ;
```

End of Algorithm

Where needed, the single entry node of the control flow graph of m_{ij} will be denoted $CFG(m_{ij}).EntryNode$.

4.3.2 Deriving Method Invocation Information

Method invocation information is captured in the class call graph (CCG) and the object call graph (OCG) associated with each class and object method. The class call graph

is conceptually built at class compile time while the object call graph is conceptually constructed at object instantiation time. Method invocation information is physically stored in the class method reference set (CMRS) and the object method reference set (OMRS), respectively.

Definition 4.18 The *class method reference set* for method m_{ij} (in C_i) is a set $CMRS(m_{ij}) = \{m_{c_1d_1}, m_{c_2d_2}, ..., m_{c_nd_n},\}$ of methods referenced directly or indirectly by m_{ij} as determined at class compile time. ■

Definition 4.19 The *object method reference set* for method m_{ij} (in O_i) is a set $OMRS(m_{ij}) = \{m_{c_1d_1}, m_{c_2d_2}, ..., m_{c_nd_n},\}$ of methods referenced directly or indirectly by m_{ij} as determined at object instantiation time. ■

Constructing the CMRS

The CMRS of a method specification is associated with the compiled method specification in the class library and is composed of a *set* of "pointers" to methods in other class definitions within the library. The construction of the set is a straightforward syntax-directed process given the assumption of definition before use. It is described in Algorithm 4.3.

Algorithm 4.3 *Constructing the CMRS for* m_{ij}

INPUT : method specification for m_{ij};
OUTPUT : $CMRS(m_{ij})$;

MethodRef : pointer to class method specification;
MethodRefs[m_{ij}] : **SET OF** pointers to class method specifications;

MethodRefs[m_{ij}] $\leftarrow \emptyset$;
FOREACH step in the method specification **DO**
    IF (the step is a message step) **THEN**
        MethodRef $\leftarrow$ pointer to the method being invoked;
        MethodRefs[m_{ij}] $\leftarrow$ MethodRefs[m_{ij}] $\cup$ MethodRef $\cup$ MethodRefs[MethodRef];

End of Algorithm

Since definition before use is assumed, all class method specifications pointed to by elements in the set 'MethodRefs' already contain a complete 'MethodRefs' set of their own. Thus, by construction, each 'MethodRefs' set contains all direct *and indirect* methods invoked.

Constructing the OMRS

Unlike the construction of the CMRS, construction of the OMRS cannot occur until the specific objects in question are instantiated. When an object is created, the object methods its methods will "reference" may be explicitly specified (instantiation time binding) or they may not be known (run time binding). In the former case, definition before use applies and the OMRS can be constructed immediately.

Run time binding is more problematic. If the set of objects being referenced is unknown at instantiation time, construction of the OMRS cannot be completed. In this case, the number of direct method references in the OMRS *is* known[6] but not all the elements of the set can be added to it. The construction of the set can only be completed once the corresponding method has actually been executed (possibly many times) to dynamically instantiate all the objects it may reference. Until this has happened the reference information based on the OMRS is incomplete and concurrency control using it will be incorrect. For this reason, objects binding to other objects at run time must be initially marked "incomplete". This will allow the transaction scheduler to use other, default, methods which conservatively ensure serializability until the OMRS and information based on it are complete.

The construction of the OMRS is described in Algorithm 4.4. The algorithm is invoked whenever a method invocation from the method in question is made. This permits the algorithm to be invoked either when an object is instantiated or when the relevant method executes and references a new object. Two counts are maintained for each object method

[6]The number of edges in the OCG is the same as the number of edges for the same method in the CCG.

which are used to determine when the OMRS is fully defined – the number of method invocations which have been made (call this 'NumObjectMethodRefs') and how many are expected (call this 'NumExpectedObjectMethodRefs'). When the two counts are equal, the OMRS is complete.

Algorithm 4.4 *Constructing the OMRS for* m_{ij}

INPUT MethodRef : pointer to an object "method";

MethodRefs[m_{ij}] : **SET OF** pointers to object "methods";

IF (This is the first time we have been invoked for this object method) **THEN**
    MethodRefs[m_{ij}] $\leftarrow$ { };
IF (MethodRef $\notin$ MethodRefs[m_{ij}]) **THEN**
    /* A new object method reference */
    MethodRefs[m_{ij}] $\leftarrow$ MethodRefs[m_{ij}] $\cup$ MethodRef $\cup$ MethodRefs[MethodRef];
    NumObjectMethodRefs++;

End of Algorithm

4.3.3 Deriving Attribute Reference and Dependence Information

Dependence information (Section 3.3.2) is based on object attributes and, possibly, method parameters and/or method-local variables being read and written. This can occur in local steps or via parameters used in message steps. To determine dependences, read and write sets for various program segments must be computed. This is trivial for assignment steps within basic blocks since they have been decomposed into simple calculations involving only a single monadic or dyadic operator. In the following, $LHS(S_{ij_k})$ refers to the attribute specified on the left hand side of assignment step S_{ij_k}, $RHS(S_{ij_k})$ refers to the attribute specified on the right hand side of the assignment step, and $Operands(RHS(S_{ij_k})$ refers to *all* the attributes specified on the right hand side of the assignment step.

$$WS(S_{ij_k}) = LHS(S_{ij_k})$$

$$RS(S_{ij_k}) = \begin{cases} RHS(S_{ij_k}) & \text{if monadic operation} \\ Operands(RHS(S_{ij_k})) & \text{if dyadic operation} \end{cases}$$

For example, if S_{ij_k} = "$X \leftarrow Y + Z$" then $WS(S_{ij_k}) = \{X\}$ and $RS(S_{ij_k}) = \{Y, Z\}$. For input steps, the readset is empty and the writeset contains all input attributes while for output steps, the writeset is empty and the readset consists of all output attributes. Finally, for message steps, the writeset contains all arguments corresponding to output parameters of the method being invoked and the readset contains all arguments corresponding to input parameters of the method being invoked.

Given $RS(S_{ij_k})$ and $WS(S_{ij_k})$, the derivation of $RS(BB_i)$, $WS(BB_i)$, and $RS(m_{ij})$, $WS(m_{ij})$ are immediate from their definitions (Section 4.2.3).

Dependence within Basic Blocks

Dependence in basic blocks is represented by threading the constituent steps to form a dependence graph. Since basic blocks are straightline code segments, the expected execution order of statements is sequential. This makes construction of the graph trivial (see Algorithm 4.5).

Algorithm 4.5 *Constructing the True Dependence Graph for a Basic Block*

INPUT : method specification for m_{ij};
OUTPUT : $DG(m_{ij})$;

RefList : list of < attribute, definition > pairs

RefList $\leftarrow \emptyset$;
add each step in the basic block as a vertex in the dependence graph;
FOREACH step S_{ij_k} in the basic block **DO**
    /* Handle attributes read */
    FOREACH attribute $a_{ip} \in RS(S_{ij_k})$ **DO**
        IF (a_{ip} is in RefList) **THEN**
            Source $\leftarrow S_{ij_x}$ where $< a_{ip}, S_{ij_x} >$ is in RefList;
            create an edge in the DG from Source to S_{ij_k};
    /* Handle attributes written */
    FOREACH attribute $Vj \in WS(S_{ij_k})$ **DO**
        IF (a_{ip} is not in RefList) **THEN**
            add the pair $< a_{ip}, S_{ij_k} >$ to RefList;
        ELSE
            update the pair $< a_{ip}, * >$ to be $< a_{ip}, S_{ij_k} >$ in RefList;

End of Algorithm

This algorithm produces a dependence graph which reflects *true* dependence within basic blocks. Algorithms to thread the statements within a basic block to form anti, output, or arbitrary dependence graphs are straightforward adaptations of Algorithm 4.5.

Another useful construction is the inverse dependence graph. That is, a graph which links uses of variables to their definitions rather than vice versa. The construction of this graph is also a simple modification of the process given for constructing the true dependence graph. In fact, the inverse graph can be produced from the corresponding dependence graph directly. The notation $DG(BB_i)$ is used to refer to the arbitrary dependence graph for basic block BB_i. The inverse dependence graph will be referred to as $IDG(BB_i)$.

Definition 4.20 The *inverse dependence graph* of a basic block, BB_i, is a directed acyclic graph $IDG(BB_i) = (V, E)$ with vertices $v \in V$ for each vertex in $DG(BB_i)$ and edges $e = (v_x, v_y) \in E$ iff there are corresponding edges $e = (v_y, v_x)$ in $DG(BB_i)$. ■

Dependence for Object Arrays Accessed in Loops

The presence of code segments which contain loops complicates the construction of the dependence and inverse dependence graphs for a class method. In constructing these graphs, sub-graphs of the CFG corresponding to loops are taken to be single steps with respect to the method itself. Within a *compound* step (i.e., one containing the basic blocks corresponding to a loop) there may also be other steps including other compound steps. This introduces a nested step structure (see Figure 4.5).

Dependence analysis is performed on both the method and on each compound step. Conservative assumptions about attributes accesed by compound steps are made when determining the dependences between those compound steps and other steps in their containing "scope". This often precludes the possibility of concurrency between steps inside a compound step and those outside it. This restriction does not seriously affect overall concurrency though since the focus is on medium-granularity concurrency. Loops

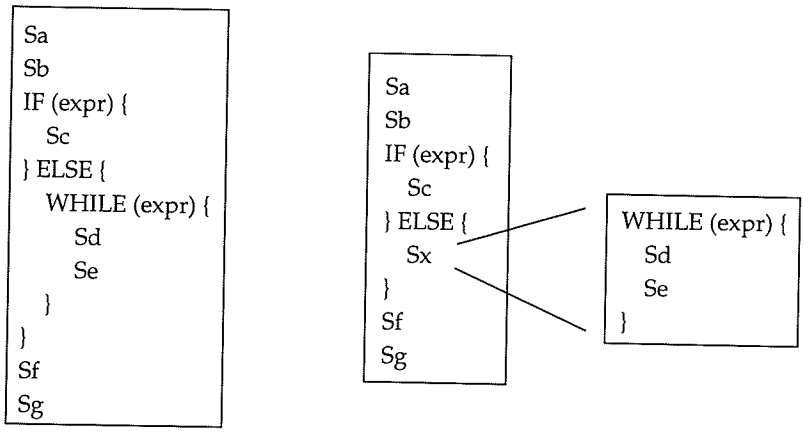

Figure 4.5: Illustration of Nested Steps

which do not contain message steps are executed serially anyway. Those which do contain message steps allow concurrency between the corresponding method executions subject to dependences within the compound step. In most cases, the greatest advantage is obtained by providing concurrency between loop iterations rather than between individual loop iterations and steps outside the loop.

For the purpose of dependence analysis, a method is viewed as a sequence of simple and complex steps. That is, all non-nested loops occurring in the method are replaced by compound steps. Each compound step is assumed to access *all* attributes that the corresponding loop could access. The determination of dependencies between compound and other steps is based on attributes accessed. Thus, the construction of the dependence and inverse dependence graphs of a method given in the following definitions are conservative and correct.

Definition 4.21 The *dependence graph* of a method m_{ij} is a directed graph $DG(m_{ij}) = (V, E)$ containing vertices $v_x \in V$ for each step (simple or compound), S_{ij_x}, in the method and a directed edge $e \in E$ from v_s to v_t iff $S_{ij_s} \delta^? S_{ij_t}$. ■

Definition 4.22 The *inverse dependence graph* of a method m_{ij}, is a directed acyclic

graph $IDG(m_{ij}) = (V, E)$ with vertices $v \in V$ for each vertex in $DG(m_{ij})$ and edges $e = (v_x, v_y) \in E$ iff there are corresponding edges $e = (v_y, v_x)$ in $DG(m_{ij})$. ■

Storing simple dependence graphs for compound steps is insufficient since they do not capture the inter-iteration dependencies which must be known to permit concurrency between message steps in loop bodies. A dependence graph can be constructed for the steps *within* the loop body in the same way as one is for the steps within a method. This permits dependence analysis between steps within a single iteration of the loop. Inter-iteration dependences must also be determined and recorded for use in providing concurrency control.

An initial discussion of dependence in accessing object arrays within loops was presented in Section 4.2.3. The information that needs to be captured (in addition to the inter-step dependencies) is a list of invalid dependence distances. Such a list may contain no entries thereby implying that there are no loop-carried dependences and hence each iteration is independent of all others. It may also contain one or more distances that must be avoided. Given a list of invalid distances, code generation may be performed to permit concurrency between loop iterations at all but the invalid distances.

Definition 4.23 The *invalid dependence distance list* for a compound step S_{ij_k}, is an ordered sequence of integer values, $IDDL(S_{ij_k})$, where for each value, n, in the list, there is some scalar object or object array element referenced within the loop in some iteration(s) I and $I + n$. ■

When dependence distances cannot be accurately calculated, conservative assumptions are made. Conservative results will often be made *by construction* in the case of nested loops. A nested loop is treated as a compound step in the loop containing it. Since the compound step is conservatively assumed to reference all the attributes it may reference, there are likely to be dependences between the executions of the compound step in *all* iterations of the outer loop. This tends to result in the serial execution of the iterations of

outer loops. Only the inner loops iterations are likely to be executed concurrently. This is not a significant drawback though since in loop nests most of the computation is done in the inner loops anyway.

Control Flow Conditions

Most often the step immediately preceding a conditional branch is the one which sets the condition codes (CC) thereby determining whether or not the branch is taken. Whether or not it is the one preceding the conditional branch, a specific step, S_{SetsCC}, is identified as the one which sets the condition codes to determine the branch. The details of identifying this step are omitted.

The condition under which a conditional branch is taken can be expressed as the subgraph of the inverse dependence graph (IDG) rooted at step S_{SetsCC} in terms of the values input to the corresponding basic block[7].

Control flow conditions for basic blocks within a method are combined in sequences by logical conjunction while they are combined where multiple control flow paths meet at a single basic block by logical disjunction. Figure 4.6 illustrates the process of constructing the control flow conditions using conjunctions and disjunctions as required. In the example, the control flow condition (CFC) for BB4 is the "AND" of the CFCs for BB2 and BB1 since BB4 is only reached by executing through them. The CFC for BB6 is the "OR" of the CFCs for BB5 and BB3 since BB6 may be reached from either one. Determining the control flow condition under which each basic block is executed involves a single walk of the control flow graph as is shown later in Algorithm 4.7.

The example shown avoids two complications which can arise in constructing the control flow conditions for basic blocks. The first occurs when the variables specified in conditions are, or depend on, values which are read at runtime. In this case, the condition cannot be evaluated prior to method execution (when the input occurs). The solution to

[7]This assumes that the graph is annotated with the operations performed at each node.

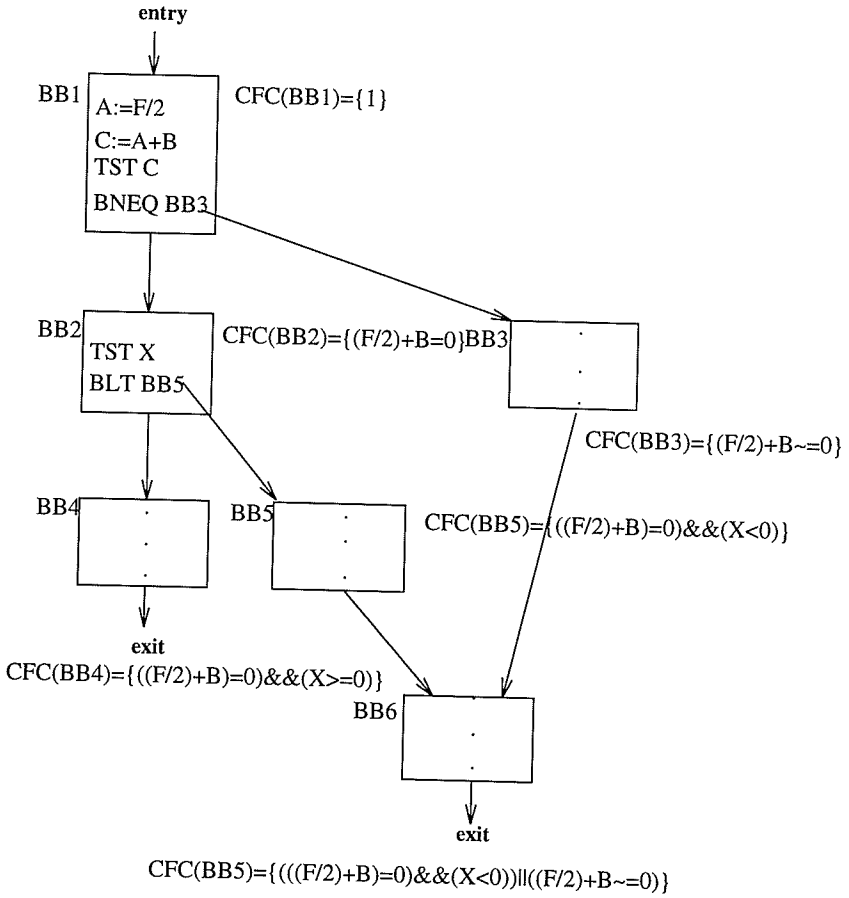

Figure 4.6: Constructing the Control Flow Conditions for Basic Blocks

this problem is to *mark* the corresponding basic block as "non-checkable". Any control flow path that contains such a basic block will also be marked non-checkable.

The second complication arises when variables specified in conditions are directly or indirectly affected by computations in basic blocks which may execute before the one whose condition is being computed. If the expressions which compute each such variable are identical along every control flow path from the entry node to the basic block being analyzed then those expressions can be incorporated into the control flow condition for the basic block. If however, the expressions vary based on the control flow path taken to the basic block in question then the control flow condition for the basic block must

also be marked not checkable. The determination of the equivalence of such expressions is a simple modification of the conventional dataflow analysis problem of determining common sub-expressions [ASU86].

In practice, this determination need not be made. The goal of run-time condition checking is to enhance concurrency control. To be effective doing this, the cost of the checking must be low (less than the benefit obtained by increased concurrency). Thus, conditions which require the evaluation of code from across basic blocks are generally too complex to merit pre-computation except in the largest method specifications. While it is important to determine the empirical limits for the application of run-time condition checking, the development of the needed heuristics is beyond the scope of the dissertation. For this reason, any run time checking which would require cross-basic block computation is not done and the corresponding basic blocks will conservatively be marked non-checkable.

Algorithm 4.6 specifies how to generate the branch conditions for method m_{ij}, $(BC(m_{ij}))$, given the inverse dependence graph for the method and a distinguished step which determines whether the branch is taken or not. The algorithm is somewhat simplified for presentation in that it assumes all dyadic operations. A more complex, but fundamentally equivalent, algorithm performs the same function in the presence of n–ary operations.

Algorithm 4.6 *Generating Branch Conditions*

INPUT : $IDG(m_{ij})$;
INPUT : $DistinguishedStep \in STEPS(m_{ij})$;
OUTPUT : $BC(m_{ij})$;

NonCheckable : **BOOLEAN** ← **FALSE** ; /* checkability flag */
BoolExpr : **STRING**;

STRING FUNCTION BuildExpr(Graph, Node);
    IF (Node is a leaf in Graph) **THEN**
        IF (Node.Vble is an input variable) **THEN**
            NonCheckable ← **TRUE**;
            RETURN FALSE;
        RETURN Node.Vble; /* variable name */
    ELSE

```
            IF (Node.Operation is a method call) THEN
                NonCheckable ← TRUE;
                RETURN FALSE;
            RETURN BuildExpr(Graph, Node.LeftChild) CONCAT
            Node.Operation CONCAT BuildExpr(Graph, Node.LeftChild);
ENDFUNCTION;

BoolExpr ← "(";
/* Walk the inverse graph from the distinguished step to the */
/* basic block's "inputs", depth-first, constructing a boolean expression. */
BoolExpr ← BoolExpr CONCAT BuildExpr(IDG, DistinguishedStep);
BoolExpr ← BoolExpr CONCAT ")";
/* Determine the branch condition based on the expression just derived */
/* and the conditional branch specified in the final step in the block */
SWITCH branch type of final step in basic block IN
CASE BEQL:
    BoolExpr ← BoolExpr CONCAT "= 0";
CASE BNEQ:
    BoolExpr ← BoolExpr CONCAT "≠ 0";
CASE BLT:
    BoolExpr ← BoolExpr CONCAT "< 0";
CASE BLE:
    BoolExpr ← BoolExpr CONCAT "≤ 0";
CASE BGT:
    BoolExpr ← BoolExpr CONCAT "> 0";
CASE BGE:
    BoolExpr ← BoolExpr CONCAT "≥ 0";
ENDSWITCH

IF (NonCheckable) THEN
    BC(basic block) ← FALSE; /* mark basic block non-checkable */
ELSE
    BC(basic block) ← BoolExpr; /* mark basic block checkable */
```

End of Algorithm

To determine the control flow conditions under which each basic block will be executed, a walk of the control flow graph for the corresponding method must be made from its entry node along all possible control flow paths. The control flow condition for a basic block is the logical disjunction of the control flow conditions of the basic blocks from which it may be reached, modified where necessary, to reflect a predecessor block that only reaches the basic block conditionally.

The construction of the control flow conditions for *all* basic blocks in a method are computed using a graph walk technique in a single pass (Algorithm 4.7).

Algorithm 4.7 *Constructing Control Flow Conditions for Basic Blocks*

INPUT : $CFG(m_{ij})$;
OUTPUT : $CFC(BB_x) \forall BB_x \in m_{ij}$;

BBqueue : Queue of Basic Blocks to be processed; /* strict FIFO */
CurrBB,OtherBB : Basic Block pointers;
BrCond : STRING; /* containing a boolean expression (a branch condition) */

Initialize $CFC(BB_x) \leftarrow True$ for all basic blocks;
CurrBB $\leftarrow CFG(m_{ij}).EntryNode$;
mark CurrBB as visited;
insert CurrBB onto BBqueue; /* CFC(CurrBB) $\leftarrow True$ */
WHILE (BBqueue is not empty) **DO**
    /* Pull a processed BB off the queue and for each BB directly reachable */
    /* from it, compute its control flow condition and add it to the queue */
    /* if it may be directly reached from another BB whose control flow */
    /* condition has not yet been incorporated into this BB. */
    CurrBB $\leftarrow$ remove(BBqueue);
    SWITCH LastStmt in CurrBB **IN**
    CASE unconditional branch:

        /* we always flow to the specified BB */
        /* simply inherit control flow condition */
        OtherBB $\leftarrow$ BB referred to in unconditional branch;
        CFC(OtherBB) $\leftarrow$ CFC(OtherBB) $\vee$ CFC(CurrBB);
        IF (OtherBB is not marked as visited) **THEN**
            mark OtherBB as visited;
            insert OtherBB onto BBqueue;
    CASE conditional branch:
        /* we either fall through to the next BB (and simply */
        /* inherit the control flow condition) ... */
        OtherBB $\leftarrow$ BB following this one;
        CFC(OtherBB) $\leftarrow$ CFC(OtherBB) $\vee$ CFC(CurrBB);
        IF (OtherBB is not marked as visited) **THEN**
            mark OtherBB as visited;
            insert OtherBB onto BBqueue;
        /* ... or we flow to the specified BB (and adjust */
        /* the control flow condition accordingly) */
        OtherBB $\leftarrow$ BB referred to in conditional branch;
        BrCond $\leftarrow$ branch condition from CurrBB to OtherBB;
        CFC(OtherBB) $\leftarrow$ CFC(OtherBB) $\vee$ (CFC(CurrBB) $\wedge$ BrCond);
        IF (OtherBB is not marked as visited) **THEN**

```
            mark OtherBB as visited;
            insert OtherBB onto BBqueue;
OTHERWISE :
    /* not a branch so we fall through to the next BB */
    /* simply inherit control flow condition */
    OtherBB ← BB following this one;
    CFC(OtherBB) ← CFC(OtherBB) ∨ CFC(CurrBB);
    IF (OtherBB is not marked as visited) THEN
        mark OtherBB as visited;
        insert OtherBB onto BBqueue;
ENDSWITCH
```

End of Algorithm

The derived control flow conditions enumerate all possible control flow paths *and* their control flow conditions. Using this information at runtime, concurrency control algorithms are more effective. The information is more accurate than using conservative, compile-time estimates of method behaviour alone.

Control Flow Paths

The determination of the control flow paths through a method is specified by Algorithm 4.8. The control flow condition of a control flow path is simply the control flow condition of its exit node[8]. This is because the condition of each node in a control flow path is a logical restriction on the condition of the node preceding it in the path. Clearly, the most restrictive condition is that of the last node (the exit node) on the path. If that node is to be executed, its control flow condition must be met and since its control flow condition logically "contains" the control flow conditions of its predecessors the exit node's control flow condition should be the control flow condition of the control flow path.

To simplify the identification of the control flow paths of a method, Algorithm 4.8 begins by removing all the back edges of the corresponding control flow graph. Such edges arise only due to the presence of loops and their removal does not affect the correctness

[8] This refers to the control flow conditions for the nodes as calculated for the specific control flow path in question not the more general (and conservative) conditions for the entire method execution.

of the process. Once back edges are removed, the identification of the control flow paths and their execution conditions degenerates to finding all the paths from the entry node to an exit node of the resulting DAG.

Algorithm 4.8 *Construction of the Set of Method Control Flow Paths and their Conditions*

INPUT : $CFG(m_{ij})$;
OUTPUT : $CFPs(m_{ij})$

BBqueue : Queue of Basic Blocks to be processed; /* strict FIFO */
CurrBB, ChildBB : Basic Block pointers;

/* Eliminate all back edges in CFG - using standard dominators algo.*/

/* Perform a depth-first traversal of the control flow graph */
/* combining relevant branch conditions, recording control flow */
/* conditions and path information. If an exit node, record the path */
/* and CFC for the block in a set of CFPs for the method. */
CFPs(m_{ij}) $\leftarrow \emptyset$; /* initially no control flow paths */

CurrBB $\leftarrow CFG(M_{ij}).EntryNode$;
CFC(CurrBB) $\leftarrow$ **TRUE**; /* always execute */
Path(CurrBB) $\leftarrow$ CurrBB;
insert CurrBB into BBqueue;
WHILE (BBqueue is not empty) **DO**
    CurrBB $\leftarrow$ remove(BBqueue);
    IF (CurrBB is an exit node) **THEN**
        add < Path(CurrBB),CFC(CurrBB) > to CFPs(m_{ij});
    ELSE
        FOREACH (child basic block, ChildBB, of CurrBB) **DO**
            Path(ChildBB) $\leftarrow$ Path(CurrBB) || ChildBB;
            BrCond $\leftarrow$ branch condition from CurrBB to ChildBB;
            CFC(ChildBB) $\leftarrow$ (CFC(CurrBB) $\wedge$ BrCond);
            insert ChildBB into BBqueue;

End of Algorithm

Chapter 5

Concurrency Control Using Static Information

This chapter discusses the application of statically derived information at run time to provide concurrency control. Effective concurrency control has two aspects; low overhead and high potential concurrency. The use of static information addresses these requirements by avoiding concurrency control where it is unnecessary (when conflicts will not occur) and by exploiting semantic information to avoid unnecessary restrictions on potential concurrency.

The chapter consists of five sections discussing run-time support, intra-transaction concurrency control, inter-transaction concurrency control, combined concurrency control, and multi-version concurrency control, respectively.

5.1 Run Time Support

Certain functions must be performed by the run time environment of the objectbase system to support concurrency control. These include the maintenance of the OMRS and support for compile time specified concurrency.

5.1.1 Maintaining the OMRS

The first requirement of the run time system is the maintenance of the OMRS for each object method. The process of updating the OMRS for a method is described in Algorithm 4.4. The construction of the OMRS requires the run time environment to invoke this algorithm whenever a method invocation is made. Method invocations are easily detected thereby permitting the algorithm to be invoked as required. If all the invocations made by the method for which the OMRS is being constructed have already occurred (as judged by the equivalence of the current and expected counts of method invocations) then the algorithm is not invoked.

Algorithm 4.4 can also be invoked when an object is dynamically created and then referenced for the first time. Similarly, when an object reference is changed (so that a new object is pointed to) the algorithm can be re-run to *update* the reference information. Because the definition before use rule applies to the new object being referenced, in this case, the process of updating the reference sets is a simple set union.

The dynamic construction and maintenance of method reference sets is complicated by referenced sub-objects which have incomplete OMRS information. By transitivity, any super-object thereof also has incomplete reference sets. When such a sub-object's sets are updated, the sets of all its parent objects must also be updated. In this case, the overhead of maintaining the reference sets is greater than in other situations. Fortunately, the need to update super-object reference sets occurs when new objects are dynamically allocated. This is uncommon after object creation since updates to data can often be done *within* existing objects without the need to dynamically allocate and reference new objects. Updating super-object reference sets requires the maintenance of inverse part-of links. These are easily constructed at the same time each part-of link is created.

Throughout this chapter, the method reference sets are presented as sets. This is not a practical implementation strategy though since it is necessary to support sets which could theoretically contain an element for every method in every object in the objectbase

(a very large set). The method reference "sets" are actually stored as ordered lists of $< object, method >$ pairs. This permits the efficient implementation of set union and intersection operations using merging techniques. These operations have linear time complexity in the number of set elements *present* not the number of set elements possible. In most cases the number of set elements present will be smaller than the number possible by many orders of magnitude.

The object method reference sets are used to determine when sub-transaction execution may result in conflicting operations at other objects. Applying method reference information in a straightforward way requires the comparison of the method reference sets for each pair of concurrent message steps *at run-time*. To enhance clarity, this is how conflict detection is presented in the algorithms that follow. There is, however, an obvious optimization which may be applied to decrease the run time overhead of using the method reference information.

For methods whose object method reference sets are complete, it is easy to pre-calculate the intersections of the reference sets of all possible pairs of local message steps. This avoids the need to perform the intersections at run-time when providing *intra*-transaction concurrency control. The pre-calculated intersection information is stored in *message step conflict sets*.

Definition 5.1 The *message step conflict set* for message steps S_{ij_k} and S_{ij_l} of method m_{ij} is a possibly empty set of method pairs, $MSCS(S_{ij_k}, S_{ij_l}) = \{< m_{x_1r_1}, m_{x_1s_1} > , < m_{x_2r_2}, m_{x_2s_2} >, ..., < m_{x_qr_q}, m_{x_qs_q} >\}$ where for each O_{x_n}, $m_{x_nr_n} \in extent(S_{ij_k})$, $m_{x_ns_n} \in extent(S_{ij_l})$, and $m_{x_nr_n} © m_{x_ns_n}$. ■

5.1.2 Supporting Compile Time Concurrency

A second requirement of the run time system is to provide support for compile time specified concurrency. To accomplish this, three concurrency primitives; "par," "meet",

and "sync" are provided. The method compiler generates code which calls these primitives at run time to accomplish the desired concurrency without the run time overhead of detecting the concurrency. The syntax of these concurrency primitives is as follows:

- **par** $Label_1, Label_2, ..., Label_N$
- **meet** $Label_s$
- $Label_s$: **sync** $Number$

A call to "par" creates separate threads[1] of control which begin execution at each of the N labels specified as arguments. Logically, the current thread (the one executing "par") terminates after creating the N new threads. For efficiency reasons, "par" need not create a new thread for the final label but may instead branch to it and begin executing the code itself.

To collect the results of the concurrently executed threads (i.e., sub-transactions) "meet" and "sync" are used. The final statement executed in each thread is a "meet" which specifies that the thread should terminate and synchronize (i.e., "meet") with other thread(s) at the sync point specified as it argument. At each sync point, a "sync" primitive waits until a specified number of threads have met at the sync point. The number of threads to synchronize is specified as the argument to "sync".

5.2 Intra-Transaction Concurrency Control

This section addresses the problem of providing effective concurrency control between the sub-transactions of a single user transaction. It ignores the issue of inter-transaction concurrency altogether. The algorithm presented is thus incapable of providing full concurrency control in an objectbase unless all user transactions are executed serially. This is not practical, so the algorithm is designed to be integrated with one for inter-transaction concurrency control presented in Section 5.3.

Any sub-transaction concurrency introduced in a method (i.e., transaction) execution must be correct. The correctness criterion used throughout this dissertation is serializability [Pap86].

[1] The threads are assumed to execute within a single address space so no explicit transfer of intermediate results between threads is required.

In conventional, flat databases, there is no concept of sub-transactions and thus, conflict serializability is classically defined *between* transactions. To assess the correctness of intra-transaction concurrency in objectbases, *intra-transaction serializability* must be used. The following definition uses history equivalence to reason about the serializability of concurrent sub-transaction executions.

Definition 5.2 A concurrent execution of the set of sub-transactions $\{T^{i_1}, T^{i_2}, ..., T^{i_n}\}$ of a transaction T^i is *intra-transaction serializable* iff its history is equivalent to *the* serial history of transaction T^i. ■

The expected serial execution order of T^i is well understood; method invocations are made in the order they are encountered and since method execution is synchronous, they complete before subsequent method steps are executed. This means the expected serial processing order is depth-first in the OCG. Thus, when two method invocations from T^i result in conflicting method executions at some object, the expected order of the executions is pre-determined. In any concurrent execution, the order of conflicting method executions must be the same as in the serial execution.

In a nested transaction model both open and closed nesting are possible. The difference between open and closed nesting is whether updates made by sub-transactions are visible to other *user* transactions prior to the parent user transaction's commitment. Since intra-transaction concurrency control is concerned with the execution of a *single* user transaction in isolation there is no distinction between open and closed nesting.

5.2.1 Using the OMRS for Intra-Transaction Concurrency Control

A user transaction is a method invocation on some object. Intra-transaction concurrency control thus focuses on the operation of the corresponding object scheduler. The responsibility for guaranteeing serializability falls on the root transaction's object scheduler (rather than the object schedulers for the sub-transactions) because the root transaction is the least common ancestor (lca) of *all* potentially conflicting sub-transactions. By the definition of the OMRS, it is therefore the *only* scheduler having sufficient information to detect *all* possible conflicts. This idea is illustrated in Figure 5.1. The potential conflict at O_8 could be detected by the object scheduler

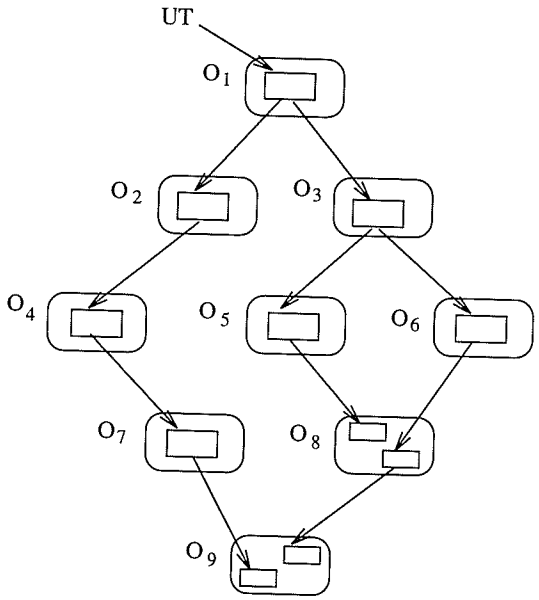

Figure 5.1: Least Common Ancestors

for O_3. The potential conflict at O_9 however can only be detected by the scheduler at O_1 which corresponds to the root (user) transaction.

To provide concurrency control efficiently, the object scheduler must examine the extents of the invoked method's message steps to determine whether conflicting method executions may occur at any sub-object(s). If none may, then the method's message steps can be scheduled for concurrent execution subject only to method-local dependence relationships. When conflicting method executions are possible, rather than delaying the message steps giving rise to all but one method execution, the root object scheduler explicitly communicates with the object scheduler(s) where conflicts may occur to ensure that method invocations will not be scheduled in a non-serializable order. This approach offers the benefit of preserving sub-transaction concurrency which would otherwise be lost. Consider the calling sequence in Figure 5.2 where blocking UT^2 due to the conflict at O_x would result in lost sub-transaction concurrency with UT^1 between O_w and (O_y and O_z).

Testing for Message Step Conflict

The algorithms presented in this chapter require the ability to test if two message steps can

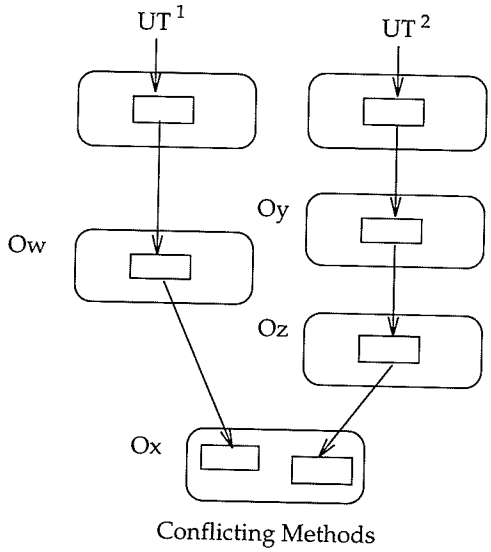

Figure 5.2: Sub Transaction Concurrency

lead to conflicting method executions at some object. Algorithm 5.1 provides this ability. It accepts two message steps as input and returns a sequence of conflicting method execution pairs. Each pair is returned in the execution order which must be respected to guarantee serializability. If no conflicts may arise, the sequence is empty.

Algorithm 5.1 *Determine Message Step Conflicts*

FUNCTION $MSConflicts(S_{ij_k}, S_{rs_t})$ **RETURNS List of Method Pairs** ;
INPUT S_{ij_k} : First message step ;
INPUT S_{rs_t} : Second message step ;
OUTPUT : Ordered pairs of conflicting methods ;
ConflictSequence : List of Method Pairs;

ConflictSequence $\leftarrow \emptyset$;
IF ($MSCS(S_{ij_k}, S_{rs_t})$ exists and is complete) **THEN**
    RETURN ($MSCS(S_{ij_k}, S_{rs_t})$);
FOREACH $(O_x \mid \exists m_{xv} \in \{OMRS(DirectlyInvokes(S_{ij_k})) \cap OMRS(DirectlyInvokes(S_{rs_t}))\})$ **DO**
    FOREACH ($m_{xy} \in OMRS(DirectlyInvokes(S_{ij_k}))$) **DO**
        FOREACH ($m_{xz} \in OMRS(DirectlyInvokes(S_{rs_t}))$) **DO**
            IF (m_{xy} © m_{xz}) **THEN**
                add the pair $< m_{xy}, m_{xz} >$ to ConflictSequence;
RETURN (ConflictSequence);

End of Algorithm

The Intra-Transaction Concurrency Control Algorithm

An algorithm for intra-transaction concurrency control is a specification of the behaviour of an object-scheduler (refer to Figure 3.1). Such an object scheduler may receive three types of messages; method invocation requests (which may correspond to either a user transaction or a sub-transaction), method execution ordering messages sent by other object schedulers, and method commit messages which are generated by each method upon termination.

When an object scheduler receives a method invocation request, its behaviour depends on whether the request corresponds to a user transaction or a sub-transaction. When processing a user transaction, m_{ij}, the object scheduler must invoke '$MSConflicts$' on each pair of message steps in m_{ij} to determine if conflicting method executions may arise. If conflicts may occur then the scheduler sends messages to the object(s) involved, specifying the necessary execution orderings to ensure serializability. The method's steps are then executed subject to the local dependencies specified by $DG(m_{ij})$.

When an object scheduler receives a method invocation corresponding to a sub-transaction, it executes the corresponding sub-transaction subject to any orderings prescribed by the root user transaction scheduler. Such orderings are recorded in the scheduler's *local scheduling graph* as they are received from the root transaction scheduler (prior to sub-transaction execution). Method invocations which must wait for other, unexecuted, method invocations are queued. Those invocations for which all orderings in the LSG have already been satisfied are executed. Sub-transaction method steps are also scheduled subject to their method local dependencies.

Definition 5.3 The *local scheduling graph* for object O_i is a directed acyclic graph $LSG(O_i) = (V, E)$ with vertices $v_x \in V$ for each conflicting method invocation on O_i and edges $e \in E$ from v_r to v_s iff v_r and v_s conflict, v_r has not yet been executed, and the serialization order $v_r \Rightarrow v_s$ is required. ■

When a method commit message is received for an execution of m_{vw}, the scheduler must remove any edges in the local scheduling graph which have m_{vw} as their source. Once m_{vw} is removed, any method invocation m_{xy}, queued only because of an edge $m_{vw} \rightarrow m_{xy}$, is available for execution. One such invocation, if it exists, is immediately selected for execution.

Since the OMRS for each object is built conservatively (assuming that all conditional message steps will be executed) the ordering messages sent to object schedulers are also conservative.

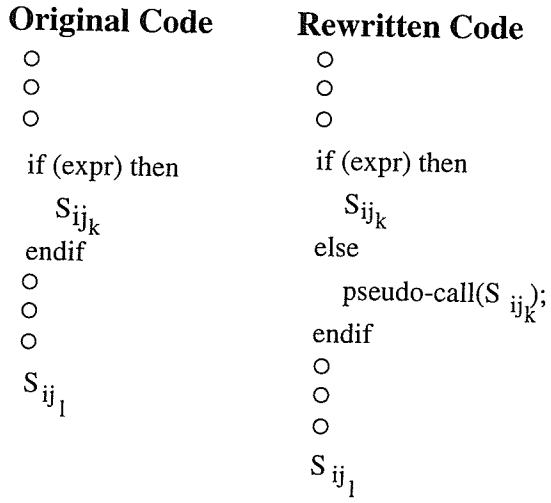

Figure 5.3: Example of a Pseudo-Call

This may result in orderings specified in *LSG*s which can never be satisfied. For example, if a conditional message step S_{ij_k} precedes another concurrent message step S_{ij_l} and they may result in conflicting method executions at O_x, then $LSG(O_x)$ will contain an edge $m_{xy} \rightarrow m_{xz}$. If S_{ij_k} is not executed, then m_{xz} will block forever at O_x waiting for the completion of m_{xy} which will never be executed. To avoid this problem, the method compiler must generate pseudo-calls to methods which are made whenever control flow conditions preclude the execution of a real call. The original and resulting rewritten code sequence are shown in Figure 5.3.

An object scheduler processes a pseudo-call in the same manner as it would any other call with two exceptions. First, the scheduler does not actually invoke the method. Second, the scheduler does not queue pseudo-calls due to conflicts with other executing or queued method invocations. Instead it immediately deletes any *LSG* edges which have as their source node the invocation identified in the pseudo-call and it also deletes the *LSG* node corresponding to the invocation in the pseudo call.

A consequence of the need for pseudo-calls is that message overhead is incurred for every conditional method invocation even if it is not made. Only when conflict freedom can be determined at compile time can this overhead be avoided. The overhead of a few pseudo-calls is minimal. It is only when conditional method invocations occurring in loops are not taken that the overhead may be significant.

For simplicity, the details of handling pseudo-calls are not presented in Algorithm 5.2.

Algorithm 5.2 *Intra-Transaction Concurrency Control*

PROCEDURE *ObjectScheduler*(Msg); /* for object O_i */

INPUT : Msg;

ConflictList : LIST-OF-PAIRS;

SWITCH (Msg) **IN**
CASE user transaction m^t_{ij}:
  FOREACH (pair of message steps $S_{ij_k} \prec_s S_{ij_l}$) **DO**
    ConflictList $\leftarrow$ $MSConflicts(S_{ij_k}, S_{ij_l})$;
    FOREACH ($< m_{xy}, m_{xz} >$) in ConflictList) **DO**
      sendto(O_x,"Order m^t_{xy} before m^t_{xz}");
  /* Only schedule after sending ordering messages */
  schedule the steps of m^t_{ij} according to $DG(m_{ij})$;
CASE sub-transaction m^t_{ij}:
  IF (there is an edge $m^t_{ix} \rightarrow m^t_{ij}$ in LSG) **THEN**
    queue m^t_{ij} for later execution;
  ELSE
    schedule m^t_{ij} according to $DG(m_{ij})$;
CASE ordering-information $< m^t_{iy}, m^t_{iz} >$:
  add vertices m^t_{iy} and m^t_{iz} to LSG if necessary;
  add the edge $m^t_{iy} \rightarrow m^t_{iz}$ to LSG;
CASE method commit m^t_{ij}:
  remove all edges of the form $m^t_{ij} \rightarrow m^t_{ix}$ from LSG;
  remove any unconnected vertices from LSG;
  IF (LSG was changed) **THEN**
    IF ($\exists$ a queued m^s_{iq} not in LSG) **THEN**
      schedule m^s_{iq} according to $DG(m_{iq})$;
ENDSWITCH ;

End of Algorithm

Correctness

The correctness criterion for intra-transaction concurrency control is intra-transaction serializability (Definition 5.2). It must be shown that Algorithm 5.2 ensures intra-transaction serializability.

Theorem 5.1 *Algorithm 5.2 ensures the intra-transaction serializability of each user transaction,* $UT^a = m_{ij}$.

Proof:

Consider concurrency between the steps of method m_{ij}. Any execution of the method respects $\prec$ locally because the steps are scheduled according to $DG(m_{ij})$. Thus, any concurrent execution of the steps of the method will produce an object-local history (one involving only operations at that object) which is equivalent to an object-local serial history. Specifically, all locally conflicting operations will be ordered consistently in the two histories. Thus, any concurrency arising *within* m_{ij} is serializable.

Conflicts may also occur when locally independent message steps are executed concurrently giving rise to concurrent sub-transactions which conflict. Due to encapsulation, sub-transactions executing on different objects cannot conflict with each other. Only when message steps result in concurrent method executions at the same sub-object can conflicts occur. To ensure intra-transaction serializability, conflicting method invocations must execute in the same order as in a serial execution of m_{ij}.

For every pair of concurrently executed message steps S_{ij_k} and S_{ij_l} for which $S_{ij_k} \prec_s S_{ij_l}$ the function $MSConflicts(S_{ij_k}, S_{ij_l})$ is called and returns all pairs of conflicting method executions $< m_{xy}, m_{xz} >$ that may arise. Each pair is ordered so that m_{xy} is the method invocation which arises from S_{ij_k} and m_{xz} is the invocation which arises from S_{ij_l}. Since $S_{ij_k} \prec_s S_{ij_l}, m_{xy} \prec_s m_{xz}$. Algorithm 5.2 sends messages to all objects O_x specifying that m^t_{xy} must execute before m^t_{xz}. The object scheduler for O_x records the orderings in its LSG. Only *after* the ordering messages have been sent are sub-transaction executions permitted. Assuming reliable, ordered, message delivery this ensures that for every object accessed by more than one sub-transaction of m_{ij}, the corresponding object scheduler already has the required execution ordering in its LSG.

Since each object scheduler orders conflicting sub-transaction method invocations according to its LSG, all the steps of sub-transaction m^t_{xy} will precede those of m^t_{xz}, in any history, as required. This includes conflicting steps. Thus, intra-transaction serializability is ensured. ■

Complexity

The time complexity of intra-transaction concurrency control is determined by the total number of method invocations made (directly or indirectly) by, the number of conflicting method executions made by, and the number of message steps in, the user transaction being scheduled.

The number of method invocations made in executing a user transaction determines the size of the relevant OMRS. This in turn determines the worst case complexity of executing Algorithm 5.1. An implementation of the algorithm using the ordered list of pairs data structure described in Section 5.1.1 has complexity which is constant if the appropriate $MSCS$ is available and linear in the number of pairs in the list if it is not. A conservative upper bound, N, on the number of pairs in the list is the cardinality of the OMRS. Thus, the complexity of Algorithm 5.1 is $O(N)$ in the worst case but only $O(1)$ for existing, fully defined, objects. If the number of message steps in the user transaction is M then the complexity of checking for sub-transaction conflicts is $O(M \cdot N)$ for new objects and $O(M)$ for existing ones. Since objects are accessed many times after they are created, the complexity of testing all message steps for conflicts is taken to be $O(M)$. Assuming good programming practices, the size of each method will be small and the number of methods invoked (M) will be correspondingly small.

The number of conflicting method invocations arising from the execution of the user transaction determines the number of scheduling messages sent by Algorithm 5.2. This number depends on the aggregation structure of the object(s) being operated on by the user transaction. If there is a high degree of object sharing among those objects then conflicting methods executions are likely otherwise they are unlikely. In any case, the number of conflicting method executions is bounded by the number of method invocations made (i.e., $O(N)$). In the average case, however, the expected percentage of conflicting executions (as compared to invocations made) will be small.

The cost of detecting message step conflicts is $O(M)$ and the cost of dealing with them is $O(N)$. Thus, the worst-case time complexity of intra-transaction concurrency control using OMRSs is $O(M + N)$.

It is difficult to do meaningful comparisons between this result and the time complexity of locking schemes. Object (attribute) level locking incurs overhead each time an object (attribute) is accessed. Suppose that N_o object (N_a attribute) references occur during the execution of a user transaction. Since objects are referenced by making method invocations on them, $N_o = N$. Furthermore, $N_a >> N$ since few methods access only a single object attribute. Thus, it is certainly true that object locking schemes have time complexity $\Omega(N)$. There is no way to determine, however, except by empirical testing when $\Omega(N) > O(N + M)$.

Evaluation

An analysis of computational complexity, by itself, is insufficient to judge the value of a concurrency control algorithm. It fails to consider the cost of each individual concurrency control operation which, for simple objects, may be as significant as the asymptotic complexity of the concurrency control algorithm. More importantly, it does not consider the benefit of enhanced concurrency which may outweigh increased overhead.

In accessing N objects, object locking requires that locks be acquired and freed at least N times. The cost of acquiring and freeing locks is known to be high [Moh90] (especially when it involves making a system call to a host operating system). On the other hand, the operations performed by Algorithm 5.1 are relatively inexpensive. For existing objects, a simple lookup operation (to retrieve the stored $MSCS$) is all that is required. For new objects (the uncommon case), a merge to determine potential conflicts consists of at most N *simple* operations each of which is a comparison of object identifiers. The operations involved in messaging (which may be expensive due to system calls) are necessary only when conflicts may actually occur. If no conflicts can occur, no messaging overhead is incurred (N is a *very* conservative estimate of the number of messages sent in Algorithm 5.2). Thus, the overhead of intra-transaction concurrency control for most object methods is small.

The level of concurrency attained using Algorithm 5.2 is very high compared to flat transactions. The concurrency achievable is potentially better than, but at least similar to that, offered by other nested transaction schemes. Increased concurrency is achieved in those situations where sub-transactions execute non-conflicting methods on an object. Overall, the concurrency attained will be higher than that provided by nested, object-level locking but lower than that provided by nested, *attribute-level* locking.

The fundamental benefit of intra-transaction concurrency control is decreased overhead. This is particularly true when the optimizations to Algorithm 5.2 presented in Section 5.2.2 are considered. Some increased concurrency is also possible depending on the semantics of the methods in each object.

5.2.2 Compile Time Enhancements

Two enhancements to the intra-transaction concurrency control algorithm may be easily provided at compile time:

1. embedding inter-step dependence information in the method code, and
2. conflict detection at compile time.

Method steps are scheduled by Algorithm 5.2 "according to $DG(m_{ij})$" (i.e., according to method-local dependencies). These dependencies are known at compile time and can be incorporated into the method code so that the object schedulers do not have to analyze $DG(m_{ij})$ at run time. It is also possible to determine at compile time that certain pairs of message steps are conflict-free. By making this information available to the object scheduler at run time, its overhead is decreased since it does not have to check for conflicts between those method invocations. The obvious benefit of these compile time enhancements is that run time overhead is decreased by their use.

Embedding Dependence Information in Method Code

At compile time, it is possible to explicitly encode the inter-step dependence information for each method within its code. This decreases overhead because the scheduler need not analyze the method's dependence graph at run time. Concurrency *within* methods is specified at compile time and supported at run time by the concurrency primitives; "par", "meet", and "sync" discussed in Section 5.1.2.

To support medium granularity concurrency, a method is subdivided into a series of "concurrent components", each of which consists of a sequence of (at least one) message steps and the dependent local steps which surround them. Steps within concurrent components are executed serially but independent concurrent components are executed concurrently. Explicit concurrency primitives are inserted between concurrent components to enforce any orderings required by dependencies between them.

By their construction, concurrent components maximize concurrency between the sub-transactions and their parent transaction.

Algorithm 5.3 *Partition into Concurrent Components and Emit Augmented Code*

INPUT : IDG – the inverse dependency graph
OUTPUT : Code augmented with concurrency primitives (backwards)

METHOD : Depth first search of IDG forking processes at each node with outdegree > 1 and emitting the correct augmented code as we go.

```
Nd : node in IDG;
CurrTree, TreeCnt, CallNo : Integer;

CallNo ← 1;
TreeCnt ← number of nodes in IDG with indegree = 0;
emit("exit");
IF (TreeCnt > 1) THEN
      emit("END: sync "TreeCnt);
/* Graph may in fact be a forest so ... */
CurrTree ← 1;
FOREACH node Nd with indegree = 0 DO
      emit("meet END");
      DoTree(Nd);
      emit("p"CurrTree":");
      CurrTree ← CurrTree +1;
IF (TreeCnt > 1) THEN
      emit("par p"1, "p"2, ..., "p"TreeCnt);
emit("entry");
```

End of Algorithm

Algorithm 5.3 performs the partitioning of a method's code into *concurrent components* and also emits the necessary concurrency primitives required to manage their execution. It makes extensive use of the subroutine *DoTree* to process each tree in the IDG rooted at an entry node. The *DoTree* subroutine is specified in Algorithm 5.4.

Algorithm 5.4 *DoTree*

```
RECURSIVE PROCEDURE DoTree(Root: node in IDG);
INPUT : a node in the IDG;
OUTPUT : augmented method code;

myCallNo, numSuccs, currSucc : INTEGER;
SuccNode : node in IDG;

myCallNo ← CallNo;
CallNo ← CallNo +1;
emit(Root);
IF (Root has outdegree = 0) THEN RETURN;
IF (Root has outdegree = 1) THEN
      SuccNode ← successor of Root;
      DoTree(SuccNode);
ELSE
      numSuccs ← number of successors of Root;
      emit"e"myCallNo": sync "numSuccs;
      currSucc ← 1;
      FOREACH SuccNode a successor of Root DO
            emit("meet e"myCallNo);
            DoTree(SuccNode);
            emit("p"myCallNo"."currSucc":");
            currSucc ← currSucc +1;
      emit("par p"myCallNo"."1, "p"myCallNo"."2, ..., "p"myCallNo"."TreeCnt);
```

End of Algorithm

Figure 5.4 presents an example dependence graph and the augmented code generated for it by Algorithm 5.3. Consider the use of the concurrency primitives illustrated in the example. The execution of S_6 is immediately preceded by the sync point $e1$: where the two threads (determined by the dependency graph) preceding S_6 meet. The threads (corresponding to the concurrent components containing, respectively, S_1, S_2, S_f, S_3 and S_4, S_5, S_g) each conclude with a "meet e1" primitive. The threads, are started by the first "par" primitive in the augmented code.

The code emitted by Algorithm 5.3 precludes the need to check for conflicts between the method executions arising from certain pairs of message steps. Certainly, message steps within the same concurrent component need not be tested since they are executed

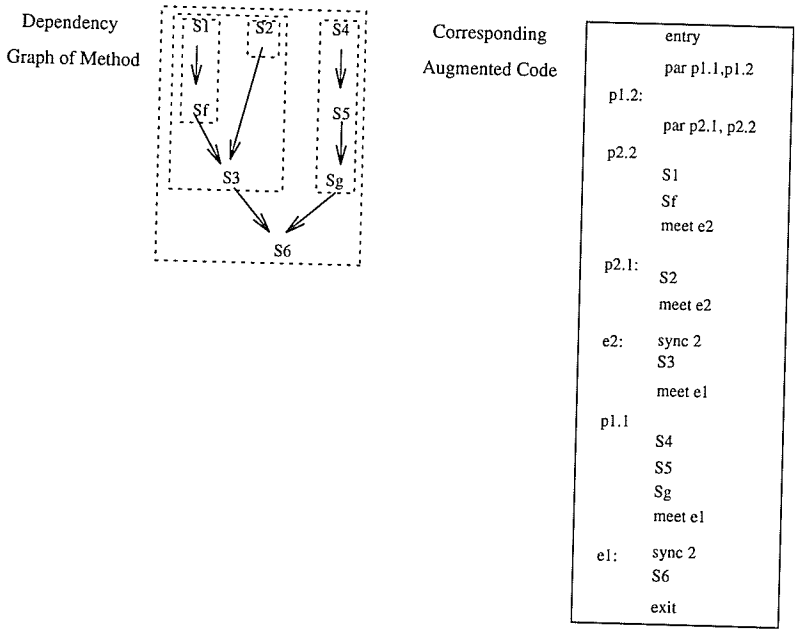

Figure 5.4: Example Dependence Graph and Augmented Code

serially. More generally, message steps which are related by $\prec$ need not be tested. This follows directly from the discussion presented in Chapter 3 which illustrated that methods arising from dependent message steps never conflict because they are never executed concurrently (case (1), Figure 3.3). Thus, in addition to eliminating the need to analyze the dependence graph at run time, partitioning into concurrent components also allows Algorithm 5.2 to be rewritten so that fewer calls to $MSConflict$ are necessary[2].

Conflict Detection at Compile Time

Algorithm 5.2 detects conflicts at run time. It may be enhanced, as just described (Algorithm 5.3), to detect conflicts only for those message step pairs which may be executed concurrently. It is possible to detect some (possibly all) of these conflicts at compile time by analysing CMRSs. The resulting information can be stored for use later

[2]This can be accomplished implicitly by incorporating compile time conflict detection into the algorithm.

by the object schedulers. This further decreases the run time overhead of intra-transaction concurrency control since the detection of conflicts is performed at compile time not run time. Potential conflicts which cannot be ruled out at compile time must still be checked for at run time.

At compile time, each pair of message steps from *different* concurrent components (as determined in Algorithm 5.3) can be tested for potential sub-transaction conflicts. Only message steps from different concurrent components need be tested since message steps within the same concurrent component are executed serially. Testing for conflicts is accomplished using Algorithm 5.1 as at run time except that the revised algorithm operates using the CMRS rather than the OMRS. The results of compile time conflict detection are summarized as *class message step conflict sets* (or logically as method conflict tables).

Definition 5.4 The *class message step conflict set* for message steps S_{ij_k} and S_{ij_l} of method m_{ij} is a possibly empty set of method pairs, $CMSCS(S_{ij_k}, S_{ij_l}) = \{< m_{x_1 r_1}, m_{x_1 s_1} > , < m_{x_2 r_2}, m_{x_2 s_2} >, ..., < m_{x_q r_q}, m_{x_q s_q} >\}$ where for each O_{x_n}, $m_{x_n r_n} \in extent(S_{ij_k})$, $m_{x_n s_n} \in extent(S_{ij_l})$, and $m_{x_n r_n} \copyright m_{x_n s_n}$. ■

The $CMSCS$ is a subset of the corresponding $MSCS$ restricted to contain those method pairs which may be derived using only compile time information.

Consider when message steps from independent concurrent components will be conflict free. If the executions of S_{ij_k} and S_{ij_l} can never reference a method from the same class, then they are certainly free of conflicts since they cannot access a sub-object in common. This can be determined by checking for an empty intersection of the corresponding methods' CMRSs.

For sub-objects which are declared as a physical *part of* the object (rather than simply being referenced by the object), compile time conflict testing can be more precise. The OMRS restricted to the contained sub-objects exists and can be used. In this case, if

message steps S_{ij_k} and S_{ij_l} execute at a common object in the OMRS subsets and if those methods conflict, then the message steps conflict.

Performing this analysis divides all pairs of message steps into three sets; those which may conflict (according to compile time information), those which do not conflict (according to compile time information), and those for which conflicts, if any, can only be detected at run time.

For message steps S_{ij_k} and S_{ij_l}, $CMSCS(S_{ij_k}, S_{ij_l}) = \emptyset$ if S_{ij_k} and S_{ij_l} are from the same concurrent component (or otherwise locally dependent) or if they can be shown to be conflict free at compile time. $CMSCS(S_{ij_k}, S_{ij_l}) = \{$ the set of conflicting method pairs $\}$ if S_{ij_k} and S_{ij_l} are locally independent and it can be shown at compile time that they conflict. Finally, $CMSCS(S_{ij_k}, S_{ij_l}) = UNDEFINED$ if it cannot be determined at compile time whether S_{ij_k} and S_{ij_l} conflict. The root object scheduler need not invoke $MSConflicts$ for those pairs of message steps for which the CMSCS is defined.

5.2.3 Object Arrays and Loops

Significant intra-transaction concurrency may arise when an object "contains" an array of objects as one of its attributes. In this case, the object's methods must process the object array by invoking methods on the array elements (which are objects). Each method invocation results in a sub-transaction and because object arrays may contain *many* individual objects they may account for a large percentage of the potential sub-transaction concurrency.

The bulk of the processing of object arrays is performed using loops. For each compound step, S_{ij_k}, corresponding to a loop, the invalid dependence distance list, $IDDL(S_{ij_k})$, is created at compile time (see Definition 4.23). This information provides the basis upon which code re-writing may be applied to loop structures in class methods. The goal of re-writing is to support the sub-transaction concurrency arising within loops. This is necessary because the simple dependence techniques discussed for straightline

code cannot deal with the iterative structures which are used to process object arrays.

Not all loops deal with object arrays. If a loop contains no message steps (invoking methods on scalar objects or elements of object arrays) then it is simply executed serially since there is no sub-transaction concurrency to exploit. This degenerate case is subsequently ignored.

For those loops which contain message steps, there are two sources of concurrency; inter-iteration concurrency, and intra-iteration concurrency. Inter-iteration concurrency occurs when different loop iterations are independent and their bodies may therefore be executed concurrently. Intra-iteration concurrency occurs within each loop iteration when message steps in the loop body are independent of one another and thus can be executed concurrently. It is possible to combine intra-iteration concurrency with inter-iteration concurrency to achieve greater overall concurrency.

Inter-Iteration Concurrency

For loops containing message steps there may be a subset of the steps in the loop body which are independent of the message steps. These statements may induce loop-carried dependences which should not affect the concurrency of the independent message steps. This is undesirable so the first stage in re-writing loops containing message steps is *loop splitting*. In loop splitting, a loop is replaced by two loops with identical bounds one of which contains all the message steps and dependent non-message steps in the body of the original loop and the other which contains the remaining, non-message steps, if there are any. The latter loop contains no message steps and is therefore processed serially. More importantly, the latter loop no longer induces any unnecessary dependences in the loop containing the message steps. From this point forward, only split loops containing message steps are considered.

If, for a loop corresponding to a compound step S_{ij_k}, $1 \in IDDL(S_{ij_k})$, then there can be no inter-iteration concurrency because in any given iteration n there is a step which

```
FOR i:= 0 to HiBound DO
        ObjArray[i];    /* a reference to an object array element */
```

Figure 5.5: Sample Loop Containing an Object Array Reference

```
FOR i:= 0 to HiBound BY MaxDepDist DO
        PARFOR j:=0 to MaxDepDist-1 DO
                ObjArray[i*MaxDepDist+j];    /* reference the object array element */
```

Figure 5.6: Rewritten Loop Containing an Object Array Reference

either depends on a step in iteration $n-1$ or there is a step in iteration n on which a step in iteration $n+1$ depends. Although inter-iteration concurrency is impossible, intra-iteration concurrency may still be possible.

Let $MaxDepDist = min(IDDL(S_{ijk})) - 1$ be the largest valid dependence distance. A loop having this largest valid dependence distance might look like the one shown in Figure 5.5[3]. This loop may be rewritten as the two loop nest shown in Figure 5.6 (the code shown assumes that $MaxDepDist$ divides $hiBound$ evenly) where the innermost loop is free from loop-carried dependencies and thus may have all its iterations executed concurrently. This is illustrated by the use of the `PARFOR` syntax in Figure 5.6. Any `FOR` loop for which $IDDL(S_{ijk})$ is empty has no loop carried dependences and may be replaced *in its entirety* by the equivalent `PARFOR` loop.

The class compiler can generate code which replaces `PARFOR` statements with the corresponding *unravelled* [Pol88] method steps. This is possible because the value of $MaxDepDist$ is known at compile time and thus the `PARFOR` loop has constant lower and upper bounds. Knowing the bounds at compile time permits unravelling. The unravelling

[3]For convenience, it is assumed that arrays are subscripted from zero to some upper bound. If this is not the case, the loop may be easily "normalized" [ZC90] to meet the requirement.

```
FOR i:= 0 to HiBound BY MaxDepDist DO
        PARFOR j:=0 to MaxDepDist-1 DO
                ObjArray[i*MaxDepDist+j];
```

... becomes (assuming MaxDepDist=3) ...

```
FOR i:= 0 to HiBound BY MaxDepDist DO
        ObjArray[i*3];
        ObjArray[i*3+1];
        ObjArray[i*3+2];

FirstLeftOver=(MaxDepDist*(HiBound DIV MaxDepDist));
FOR i:=0 to HiBound - FirstLeftOver DO
        ObjArray[FirstLeftOver+i];
```

Figure 5.7: Unravelling Example

process is illustrated in Figure 5.7 where a maximum valid dependence distance of 3 is assumed.

In the case of loops which are free of inter-iteration dependences but do not have compile-time evaluatable bounds, this type of unravelling cannot be done. Unravelling is still possible however by selecting a "reasonable" value for $MaxDepDist$. Ideally $MaxDepDist$ would be chosen to be the range of the loop $(HiBound - LoBound + 1)$ but this value is not known until run-time. A guess at the value of $MaxDepDist$ is made and the code shown in Figure 5.7 is used. The first loop is executed in parallel and the second loop is executed serially to finish up the iterations missed in the original loop. The trick is to choose $MaxDepDist$ so that for unknown loop bounds the work done in the first loop is maximally concurrent and as few iterations as possible remain afterwards. Thus, a large value for $MaxDepDist$ must be chosen to attain greater concurrency in the first loop but a small value should be chosen to minimize the number of left over iterations which will be executed serially by the second loop.

A simple heuristic for choosing a value for $MaxDepDist$ is based on the size of the

object arrays operated on in the loop being re-written (often there will be only a single object array). $OptMaxDepDist$ is selected as the size of the *smallest* object array accessed. This is an optimistic estimate since the loop may not process all of the elements in the array. Choosing $MaxDepDist = MAX(\sqrt{(OptMaxDepDist)}, 2)$ ensures that some concurrency (at least 2 iterations worth) is always exploited. An entire array (of size $OptMaxDepDist$) is accessed in $\sqrt{(OptMaxDepDist)}$ steps of $\sqrt{(OptMaxDepDist)}$ iterations each. Subsets of the array are accessed in fewer steps of the same size. The maximum number of iterations ever executed serially is $\sqrt{(OptMaxDepDist)} - 1$. This provides an effective compromise in maximizing and minimizing the value of $MaxDepDist$.

Loop dependence analysis, loop splitting, and unravelling also apply to multi-dimensional object arrays but the details are straightforward and would not significantly enhance the dissertation. Thus, they are not discussed.

Intra-Iteration Concurrency

Concurrency within each iteration of innermost loops may also be achieved by dividing the code of the loop body into concurrent components. This is accomplished by applying Algorithm 5.3 to the required loop body (instead of the mainline code) using the IDG corresponding to the loop body (instead of the mainline). If `PARFOR` loops are unravelled prior to determining intra-iteration concurrency then no special processing is required for them. In this case, the intra-iteration concurrency that arises within their enclosing loop(s) will automatically be detected.

Caveats

The code resulting from loop re-writing does not explicitly result in concurrency. All it does is identify concurrent components. At run-time, message steps in different concurrent components must still be checked for potential conflicts at common sub-objects.

A possible problem is that multiple iterations from one transaction may invoke the

same object method. In this case, the ordering information sent to the object must identify the loop iterations from which the sub-transactions arose to distinguish between them. This problem can be handled by the class compiler which can easily augment a method call with a serial number within the method identifying the relevant loop iteration. This problem only arises if the resulting sub-transactions at the common object may execute concurrently. Also, if dependencies exist between the original message steps, there is no problem (in fact the steps will be in the same concurrent component).

5.3 Inter-Transaction Concurrency Control

Algorithms for managing intra-transaction concurrency are insufficient for implementing concurrency control in the presence of concurrent user transactions. Inter-transaction concurrency control must also be provided.

Due to the unpredictable interactions of *unrelated* transactions, inter-transaction concurrency control requires more run time involvement in scheduling than intra-transaction concurrency control does. The compile-time optimizations discussed in the previous section are not practical (or, in some cases, even possible) when dealing with inter-transaction concurrency. With multiple, concurrent, user transactions the definition of serializability also becomes more complicated. Correctness is achieved if a concurrent execution is equivalent to *any* serial execution. In this respect, *inter-transaction serializability* is similar to conventional serializability.

Definition 5.5 An execution of a set of transactions $\mathcal{T} = \{T^1, T^2, ..., T^n\}$ is *inter-transaction serializable* iff its history is equivalent to some serial history of the transactions. ■

Since inter-transaction concurrency control is concerned only with the execution of user transactions without sub-transactions, transaction nesting is not an issue. Concurrency due to nesting is supported by intra-transaction concurrency control.

5.3.1 Supervised Inter-Transaction Concurrency Control

Supervised inter-transaction concurrency control is jointly provided by the global scheduler and the object schedulers (see Figure 3.1). The global scheduler is charged with ensuring that inter-object serializability is respected. The object schedulers enforce serializability locally at their respective objects, in part, by adhering to specific orders dictated by the global scheduler.

Supervised inter-transaction concurrency control is an extension of the work of Zapp and Barker [ZB93c, ZB93a, ZB93b] who describe a model, architecture, and concurrency control algorithm for closed nested transactions in objectbases which extends the work of Hadzilacos and Hadzilacos [HH91].

Zapp and Barker's algorithm provides object-local serialization of transaction operations (using attribute-level, two-phase locking) and global, inter-object serialization of transactions (based on cycle detection in a dynamically constructed graph of object local orderings – the DAX). Serialization conflicts occur when object-local schedulers select contradictory serialization orders (possibly transitively) for the operations of two or more transactions. Such conflicts are detected by the global scheduler when the object schedulers report their selected serialization orders.

When a new ordering is received from an object scheduler, the global scheduler *conditionally* adds the appropriate arc to the DAX. A check for acyclicity is then performed and if a cycle is found, a conflict is detected. The conditional edge is then removed, and the corresponding transaction is rolled back[4]. Thus, only valid serialization orders are permanently recorded in the DAX. These orders are enforced at all objects by blocking transactions at the global scheduler[5]. The appropriate node and arcs in the DAX are removed when a transaction commits or aborts.

[4] With nested transactions, forcing *entire* transactions to be rolled back incurs unnecessary overhead.

[5] Blocking entire transactions rather than just their conflicting components (i.e., sub-transactions) decreases potential concurrency (Figure 5.2 and related discussion).

Zapp and Barker's algorithm may be improved by applying static analysis to determine, *à priori*, when transaction conflicts may occur. Conflicts can be detected by the global scheduler using Algorithm 5.5 which replaces Algorithm 5.1 and determines conflicts between unrelated user transactions rather than between message steps within a user transaction. The global scheduler can then instruct the object schedulers to follow specific serialization orders for the operations of the conflicting transactions. This avoids unnecessary roll-backs since serialization orders are prescribed and thus, contradictory, object-local, serialization orders are never chosen. It also enhances concurrency because only conflicting operations are postponed to ensure serializability (independent operations which follow a blocked operation may still execute).

Algorithm 5.5 *Determine User Transaction Conflicts*

FUNCTION $UTConflicts(m_{ij}^{t_1}, m_{rs}^{t_2})$ **RETURNS Set of Method Pairs** ;

INPUT $m_{ij}^{t_1}$: First user transaction ;
INPUT $m_{rs}^{t_2}$: Second user transaction ;
OUTPUT ordered pairs of conflicting methods;

ConflictSet : Set of Method Pairs;

ConflictSet $\leftarrow \emptyset$;
FOREACH $(O_x \mid \exists m_{xv} \in \{OMRS(m_{ij}) \cap OMRS(m_{rs})\})$ **DO**
  FOREACH $(m_{xy} \in OMRS(m_{ij}))$ **DO**
    FOREACH $(m_{xz} \in OMRS(m_{rs}))$ **DO**
      IF $(m_{xy} \copyright m_{xz})$ **THEN**
        ConflictSet $\leftarrow$ ConflictSet $\cup < m_{xy}^{t_1}, m_{xz}^{t_2} >$;
RETURN (ConflictSequence);

End of Algorithm

New transactions that do not conflict with active transactions may be scheduled immediately. Those that may conflict require checking to ensure that they will not cause deadlock and to determine an appropriate serialization order with other transactions at those objects where conflicts may occur[6].

[6]These turn out to be related problems.

Determining possible serialization violations is done by *conditionally* adding arcs to the DAX based on statically determined conflict locations and then performing cycle detection as in the original algorithm when a new object-local serialization order is determined. This process is called *speculative* DAX checking. As long as there are relatively few objects at which conflicts may occur (the expected case), exhaustive testing of all possible serialization orders is feasible. If this is not the case, heuristics may be applied.

One distinct benefit of scheduling *à priori* to avoid serialization errors is the impossibility of deadlock. Locking is *not* used in Algorithm 5.6 as it is in the original algorithm. Object schedulers schedule methods based on method conflicts, allowing non-conflicting method executions to proceed concurrently and imposing an order on conflicting ones. This is subject to inter-object serialization orders imposed by the global scheduler to avoid rollback, etc. Conflicting operations are ordered so their corresponding transactions are serializable. Since serializability precludes cycles in the DAX, the "circular wait" condition required for deadlock cannot occur. Thus, deadlock is not possible.

When the global scheduler decides that a prescribed serialization order is beneficial, it must ensure that the selected order is adhered to at those objects where the transactions involved conflict. This requires the global scheduler to communicate with the object schedulers[7].

In the simplest case, a prescribed serialization order will indicate that the new transaction must be serialized after an existing one. This is easily enforced since the global scheduler can send ordering information *before* actually making the relevant object method invocations. A more interesting situation arises when the new transaction must serialize before an already active transaction. In this case it is possible that the global scheduler will send a message prescribing an order that can no longer be met (because a conflicting transaction operation has already been performed). In this case, *partial* rollback (of only the *known* conflicting operations) may still be required. If the conflicting active trans-

[7]Such communication occurs in the original algorithm but is not used to specify serialization orders.

action has not yet executed the conflicting operation and the object schedulers always process incoming serialization-order messages before scheduling new local operations then no serialization error occurs and the desired ordering is achieved.

Algorithm 5.6 *Supervised Inter-Transaction Concurrency Control*

GLOBAL DAX : Directed Graph of UT serialization orders;
GLOBAL TransSet : Set of Active Transactions;
GLOBAL ConflictSet : Set of Conflicting Method Pairs;

RECURSIVE FUNCTION UpdateDAX(UT^i,LocalDAX,List) **RETURNS BOOLEAN**;
IF (List is Nil) **THEN**
    RETURN(**TRUE**);
vertex ← First element on List;
List ← List without first element;
/* Try first serialization order: $UT^i \Rightarrow$ vertex */
insert edge $UT^i \rightarrow vertex$ into LocalDAX;
IF (CycleFree(LocalDAX, UT^i, vertex)) **THEN**
    IF (UpdateDAX(UT^i,LocalDAX,List) = **TRUE**) **THEN**
        RETURN (TRUE);
remove edge $UT^i \rightarrow vertex$ from LocalDAX;
/* Try second serialization: $UT^i \Rightarrow$ vertex */
insert edge $vertex \rightarrow UT^i$ into LocalDAX;
IF (**NOT** CycleFree(LocalDAX, vertex, UT^i)) **THEN**
    RETURN (**FALSE**);
RETURN (UpdateDAX(UT^i,LocalDAX,List));
ENDFUNCTION

PROCEDURE GlobalScheduler(UT^i); /* Begin by checking if any active Transactions have finished */
FOREACH ($UT^j \in$ TransSet that has completed) **DO**
    TransSet ← TransSet $-\{UT^j\}$;
    Remove vertex UT^j and any adjacent edges from DAX;
/* Now check UT^i for conflicts with active transactions */
ConflictSet ← $\emptyset$;
FOREACH (active transaction $UT^j \in$ TransSet) **DO**
    ConflictSet ← ConflictSet $\cup$ $UTConflicts(UT^i, UT^j)$;
IF (ConflictSet == $\emptyset$) **THEN**
    TransSet ← TransSet $\cup$ $\{UT^i\}$;
    Issue UT^i;
ELSE
    ConflictingTransactions ← NIL;
    FOREACH ($UT^j \mid \exists m^j_{xy} \in$ ConflictSet) **DO**
        add UT^j to ConflictingTransactions if not already present;
    LocalDAX ← DAX;
    Add node UT^i to LocalDAX;
    IF (UpdateDAX(UT^i,LocalDAX,ConflictingTransactions) = **FALSE**) **THEN**

```
        abort(UT^i); /* or a sub-transaction thereof */
        return;
    DAX ← LocalDAX;
    FOREACH (pair < m^i_xy, m^j_xz > ∈ ConflictSet) DO
        send order of UT^i and UT^j in DAX to O_x;
    TransSet ← TransSet ∪ {UT^i};
    Issue UT^i;
ENDPROCEDURE
```

End of Algorithm

Algorithm 5.6 provides a specification of the behaviour of the global scheduler in providing inter-transaction concurrency control. For convenience, the algorithm is described such that it uses local copies of the DAX (LocalDAX) for speculative DAX checking. This is merely a convenience for presentation. The actual checking can be performed without making copies of the DAX.

Checking for cycles is performed by the routine *CycleFree* which accepts three parameters: the localDAX, the user transaction being scheduled (UT^i), and the *vertex* to which the speculative edge from UT^i points. *CycleFree* then checks for cycles in localDAX which include the edge from UT^i to *vertex* using conventional algorithms.

The object scheduler algorithm is not specified. Object schedulers simply schedule method invocations in the order determined by the global scheduler (if one is determined) and in any arbitrary order otherwise.

The routine *UpdateDAX* called in Algorithm 5.6 is responsible for selecting appropriate serialization orders for the new transaction relative to existing, active transactions and for updating the DAX to reflect the orderings determined. There are a number of possible algorithms for *UpdateDAX*. A simple, greedy approach which tries all possible serialization orders and adopts the first valid one is presented in Algorithm 5.6.

A simple improvement to the purely greedy approach is possible. When the global scheduler decides on an order which serializes a new transaction *before* an active one, rather than waiting to determine that a conflicting order has been chosen, the object

schedulers might be required to respond to the selected order indicating whether or not it is acceptable. This still involves only one additional message per conflicting object and precludes the need for later rollbacks if an alternative serialization order is possible. The greedy algorithm simply selects the *first* valid serialization order and may therefore inadvertently choose an order which can no longer be satisfied due to indirectly conflicting orders chosen by the object schedulers involved. A good heuristic for choosing a serialization order is to try to serialize new transactions *after* existing ones first. This approximates timestamp order at conflicting objects but does not result in the blockage of *all* transaction operations when a conflict occurs with a transaction having an earlier "timestamp".

Correctness

Correctness is illustrated by showing that inter-transaction serializability is ensured by Algorithm 5.6. This is done by considering the histories which can be produced by executions permitted by the algorithm.

Theorem 5.2 *Algorithm 5.6 (the global scheduler) ensures the inter-transaction serializability of each user transaction, $UT^a = m_{ij}$, submitted to it.*

Proof:

It must be shown that inter-transaction serialization orders are consistent at all objects. Since non-conflicting operations need not be serialized, only conflicting operations are considered. All conflicting operations (method executions) are detected by Algorithm 5.6 through calls to $UTConflicts$ and assigned relative serialization orders. These orders are respected at the objects where conflicts occur because each object scheduler follows the orders prescribed by the global scheduler and the global scheduler sends all ordering messages before scheduling the operations of a user transaction. Therefore, it need only be shown that the global scheduler selects a *correct* serialization order for conflicting objects.

Given a direct conflict between the user transaction being scheduled and an existing one, the serialization order is insignificant. The global scheduler will choose one or the other and ensure that it is enforced at all objects where the user transactions conflict. Thus, inter-transaction serializability is correctly provided for the two transaction case.

It is also possible that *indirect* (or *transitive*) conflicts may occur. For example, UT^a may conflict with UT^b at O_i, while UT^b conflicts with UT^c at O_j and UT^c conflicts with UT^a at object O_k. In this case, a correct serialization order for all the user transactions must be selected. The number of transactions and objects involved in such an indirect conflict may be arbitrarily large.

The definition of inter-transaction serializability states that any correct concurrent execution must be equivalent to some serial execution of the same set of transactions. Thus, for a given serialization order, $UT^a \Rightarrow UT^b \Rightarrow UT^c$, all operations of UT^a which conflict with operations of UT^b must precede those operations of UT^b in a correct history. Similarly, all operations of UT^b which conflict with operations of UT^c must precede those operations of UT^c in a correct history, etc. To see that this will be so, the construction of the DAX must be considered.

Before any transaction is executed, the DAX is empty. When the first transaction is scheduled, there are no conflicts, so it is allowed to execute once a node for it has been added to the DAX (and, in Algorithm 5.6, its transaction identifier has been recorded in $TransSet$). At this point, the DAX is clearly acyclic.

When another transaction UT^i arrives to be scheduled its conflicts with *all* active transactions are detected. If there are none, then it is allowed to execute after the DAX and $TransSet$ have been updated. At this point, the DAX is also acyclic. For each active transaction, UT^j, that UT^i conflicts with, a serialization order must be determined. Either $UT^i \Rightarrow UT^j$ or $UT^j \Rightarrow UT^i$ or UT^i is aborted. The function $UpdateDAX()$ implements the selection of serialization orders. It greedily selects the first combination of serialization orders between UT^i and those active transactions it conflicts with which

is serializable. It tests a particular serialization order by speculatively inserting the appropriate edge into $LocalDAX$ and checking for cycles. By testing for cycles at each step, transitive serialization errors are also detected.

Finally, when a transaction completes, it is removed from $TransSet$ and the corresponding vertex and any arcs attached to it are removed from the DAX.

The DAX specifies the serialization orders between the active transactions. It is always acyclic and contains a vertex for every active transaction and edges reflecting every serialization order between those transactions. All object schedulers execute conflicting operations in the order recorded in the DAX. Since the DAX is acyclic, the serialization order it determines is valid. All conflicting operations are ordered consistently at all objects in the history of any concurrent execution. Because each object scheduler follows the orders prescribed by the global scheduler, if $UT^a \Rightarrow UT^b$ then all operations of UT^a which directly or indirectly conflict with operations of UT^b at any object will execute before the conflicting operations of UT^b (i.e., all the operations of UT^a precede the operations of UT^b they conflict with in any history). Hence, the set of active transactions is serializable (in the order specified in the DAX). Thus, inter-transaction serializability is ensured. ■

Complexity

The complexity of Algorithm 5.6 is determined by two components; the complexity of determining conflicts and the complexity of finding a serializable execution order in the presence of conflicts.

The complexity of Algorithm 5.5 (for determining conflicts) is $O(N)$, where N is the size of the largest OMRS[8]. Algorithm 5.5 is invoked once for each active transaction. Thus, if there are K active transactions, then the complexity of detecting conflicts is $O(N \cdot K)$.

[8]This assumes an implementation like that discussed for the implementation of Algorithm 5.1.

The complexity of finding a valid serialization order is dominated by the complexity of cycle detection in a directed graph. The number of vertices in the graph is $K+1$ and the number of edges, E, depends on the number of conflicts between the active transactions. Cycle testing in a directed graph has complexity $O(E)$ which in the worst case is $O(K^2)$ (an edge to and from every other vertex). Normally the DAX graphs will have relatively low connectivity so the average case complexity should be much better.

The cycle detection algorithm must be run once for each active transaction that conflicts with the transaction being scheduled (say L times). Thus, the overall complexity of finding a valid serialization order is $O(L \cdot E)$.

Evaluation

Once again, because of the dynamic nature of the transaction conflict relations, the complexity analysis only provides extremely pessimistic, worst case estimates of efficiency.

The dominant costs in Algorithm 5.6 are cycle checking and conflict detection. Zapp and Barker's algorithm checks for cycles every time a new order is sent from an object-local scheduler. Algorithm 5.6 only invokes the cycle-check algorithm for conflicting operations. Since, in most cases, only a small subset of all operations will actually conflict, Algorithm 5.6 incurs far less overhead for cycle checking than Zapp and Barker's algorithm. To achieve this benefit, conflict detection must be performed and its overhead incurred.

Conflict detection incurs a *small* constant overhead for each method invocation made (directly or indirectly) by the transaction being scheduled. Zapp and Barker's algorithm incurs a *large*, worst-case $O(E)$, overhead for each method invocation. Since conflicting operations represent only a small fraction of all operations, Algorithm 5.6 should provide a significant performance increase in most cases.

The real cost of both cycle checking and conflict detection may be further decreased using techniques which will be discussed shortly.

Freedom from Deadlock

Another advantage of Algorithm 5.6 is that it provides freedom from deadlocks. The four necessary conditions for deadlock [SG94] (as applied to concurrency in object bases) are:

1. **Mutual exclusion:** Access to at least one object is exclusive so other concurrent transactions must wait.
2. **Hold and wait:** At least one transaction must exist which has exclusive access to an object and is waiting to obtain access to other objects held by other transactions.
3. **No preemption:** Access to objects cannot be taken away from one transaction by another.
4. **Circular Wait:** There exists a set of waiting transactions $\{T^0, T^1, ..., T^n\}$ such that T^0 is waiting to access an object T^1 has exclusive access to, T^1 is waiting to access an object T^2 has exclusive access to, ..., and T^n is waiting to access an object T^0 has exclusive access to.

Theorem 5.3 *Algorithm 5.6 ensures freedom from deadlocks.*

Proof:

If it can be shown that any one of the four necessary conditions for deadlock does not apply, then by definition, deadlock is impossible [SG94]. By the nature of Algorithm 5.6 the "circular wait" condition is precluded.

Only conflicting operations achieve mutually exclusive access to objects so deadlocks must arise only due to conflicting operations. Assume that deadlock is possible and consider how a set of waiting transactions $\{T^0, T^1, ..., T^n\}$ such that T^0 is waiting to access an object T^1 has exclusive access to, ..., and T^n is waiting to access an object T^0 has exclusive access to may form. T^0 must be executing a method on some O_0 which has invoked a method on some O_1 where T^1 is already executing a conflicting method which has invoked a method on some O_2 where T^2 is already executing a conflicting method which ... which has invoked a method on some O_n where T^n is already executing a conflicting method which has invoked a method on O_0 which conflicts with the method being

executed by T^0 (This reasoning also extends to concurrently executed sub-transactions.). By Theorem 5.2, Algorithm 5.6 ensures inter-transaction serializability. The constructed cycle implies that $T^0 \Rightarrow T^1 \Rightarrow ... \Rightarrow T^n \Rightarrow T^0$. Since this violates serializability and therefore cannot be produced by Algorithm 5.6 a contradiction results and, thus, Algorithm 5.6 ensures freedom from deadlocks. ■

5.3.2 Improved Supervised Inter-Transaction Concurrency Control

Algorithm 5.6 may be made more efficient by decreasing the cost of its most expensive functions; conflict detection and cycle testing.

Decreasing the Overhead of Conflict Detection

The frequent calls to $UTConficts()$ in Algorithm 5.6 are a costly part of its execution. It is possible to avoid this significant overhead by caching method reference information for the active transactions in a data structure local to the global scheduler. A single OMRS for all the active transactions can be dynamically maintained and then compared to the OMRS of a new transaction to detect conflicts.

Maintaining a "combined" OMRS is complicated by the possibility that two active transactions may invoke the same object method. Thus duplicate elements in the combined OMRS are possible. The information that more than one transaction is referencing a particular method is important and must not be discarded. Thus, a simple set cannot be used to store the combined reference information.

A multi-set[9] summarizing transaction accesses to objects permits multiple, identical elements. The combined OMRS for the active transactions can be represented as a multi-set.

[9] A set where a single element may occur multiple times.

Initial COMRS	$\{m_{11}, m_{13}, m_{42}, m_{72}\}$
New Transaction's OMRS	$+ \{m_{42}, m_{87}\}$
Final COMRS	$= \{m_{12}, m_{13}, m_{42}, m_{42}, m_{72}, m_{87}\}$

Figure 5.8: Adding an OMRS to the COMRS

Definition 5.6 The *combined OMRS* (COMRS) is a multi-set containing elements representing object methods such that, for each active transaction accessing some method m_{ix}, the element corresponding to m_{ix} occurs in the multi-set once. ■

To maintain the COMRS so it accurately represents the current method references for the active transactions, steps must be taken when transactions are scheduled and when they terminate. When a transaction is scheduled, its OMRS (a degenerate multi-set where each element occurs only once) must be "added" to the COMRS. When a transaction commits or aborts, its OMRS must be "subtracted" from the COMRS. The "addition" of multi-sets is illustrated in Figure 5.8.

When a new transaction begins, its OMRS may be intersected with the COMRS and the result determines whether or not the transaction may conflict with any of the active transactions. This does not, however, determine *where* a conflict will occur. To determine this, a data structure must be maintained which captures *both* the set of objects each transaction accesses and the set of transactions which access each object. A multi-set alone does not provide this capability.

Since the needed data structure will be queried and updated frequently, it must be time efficient. This goal is complicated because the set of objects accessed by any transaction is a *sparse* subset of the objects in the objectbase and since different transactions may access vastly different sets of objects, the sets are likely to be nearly disjoint for each pair of transactions.

The process of determining *which* transactions conflict with a new transaction requires

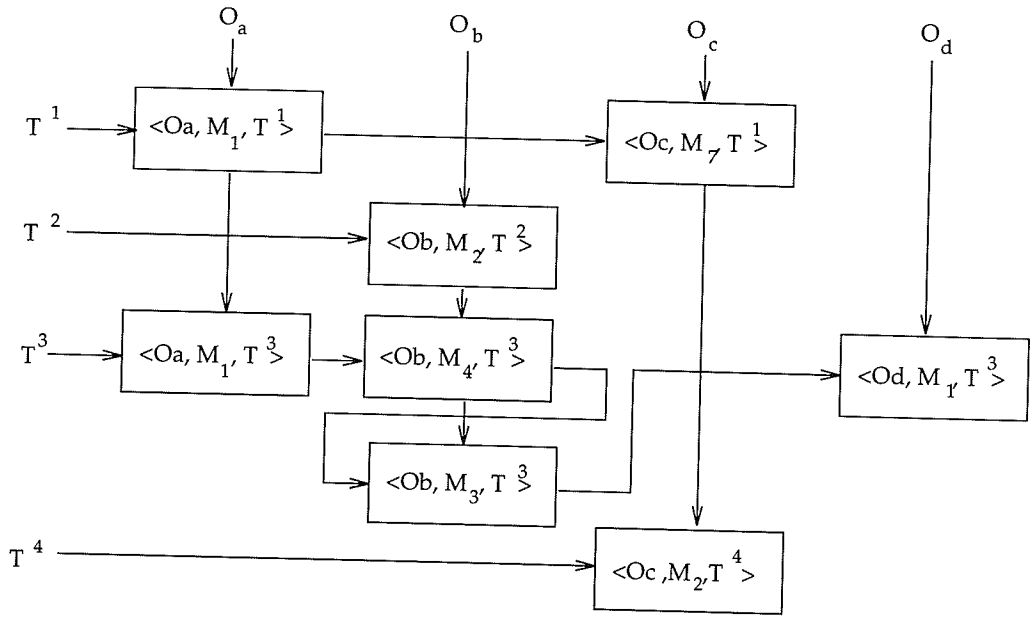

Figure 5.9: Scheduling Data Structure

knowledge of which objects are in conflict and, for each such object, which other transactions are accessing it. These requirements and the required multi-set operations may be met by a simple scheduling structure – the active transaction reference list (ATRL). An example ATRL is shown in Figure 5.9.

Data Structure 5.1 The *active transaction reference list* is an ordered, doubly-linked list of $< object\_id, method\_id, transaction\_id >$ triples threaded by object and transaction identifiers that enumerates which objects are accessed by each transaction and which transactions access each object.

An implementation of the COMRS as a singly-linked list ordered by *method_id* within *object_id* (the COMRL) may be dynamically maintained and used to detect conflicts quickly. The COMRL is equivalent to the multi-set built directly from the ATRL by unioning its "rows". Thus, the ATRL need only be used once conflict has been detected to determine the actual conflicts. Testing for conflicts is done using the COMRL.

The ATRL suggests an efficient representation strategy for *all* method reference sets. Since such sets are typically very sparse and come from large domains, the use of an ordered linked list structure is an appropriate and efficient set representation. Thus, OMRSs are assumed to be implemented as OMRLs and the COMRS is implemented as the COMRL.

Data Structure 5.2 The *object method reference list* is an ordered link list representation of the corresponding OMRS.

Data Structure 5.3 The *combined object method reference list* is an ordered link list representation of the corresponding COMRS.

Conflict checking as applied in Algorithm 5.6 is now done as an initial test performed by intersecting the OMRS for the new transaction and the COMRS (implemented as an *object_id* merge match of the OMRL for the new transaction and the COMRL). If no conflicts are detected, the ATRL is updated with the information from the new transaction's OMRL and execution of the transaction begins. If conflicts are detected, then the $ConflictSet$ is built from the results of the merge match and information contained in the ATRL by calling $UTConflicts$ (Algorithm 5.7).

The merge-match returns the *object_ids* of those objects accessed by both the new transaction and at least one active transaction. It must be determined which active transactions access objects accessed by the new transaction to determine if their accesses (i.e., method invocations on the object) conflict with the method invocation of the new transaction.

$UTConflicts$ is called only if the new transaction accesses object(s) in common with the active transactions. Unlike Algorithm 5.5 this algorithm is called only once with the ATRL, OMRL, and a "set" of shared objects as input. It returns a set of conflicting method pairs just as the original algorithm did. Thus, Algorithm 5.6 must be modified

to do the initial conflict test as described above and to make a single call to the new $UTConflicts$ rather than looping and calling it for each active transaction.

Algorithm 5.7 *Determine User Transaction Conflicts using ATRL*

FUNCTION $UTConflicts(ATRL, OMRL, ObjList)$ **RETURNS Set of Method Pairs** ;

INPUT ATRL : Active Transaction Reference List ;
INPUT OMRL : Object Method Reference List of new transaction T^i;
INPUT ObjSet : Set of *object_ids* of shared objects ;
OUTPUT : Ordered list of conflicting method pairs;

ConflictSet : Set of Method Pairs ;
Curr : pointer to OMRL elements ;

ConflictSet $\leftarrow \emptyset$;
Curr $\leftarrow$ OMRL;
FOREACH ($O_x \in ObjSet$) **DO**
    /* skip over irrelevant entries in OMRL */
    WHILE ($Curr \rightarrow object\_id \neq O_x$) **DO**
        Curr $\leftarrow$ next element on list;
    /* merge match of OMRL with column of ATRL corresponding to O_x */
    WHILE ($Curr \rightarrow object\_id = O_x$) **DO**
        Let $m_{xy} \leftarrow Curr \rightarrow \{object\_id, method\_id\}$;
        FOREACH ($m^{at}_{xz} \in$ column of ATRL corresponding to O_x) **DO**
            IF ($m_{xy} \copyright m_{xz}$) **THEN**
                ConflictSet $\leftarrow$ ConflictSet $\cup < m^i_{xy}, m^{at}_{xz} >$;
RETURN (ConflictSequence);

End of Algorithm

Decreasing the Overhead of Cycle Testing

The cost of cycle testing is directly related to the size of the graph ($LocalDAX$) which must be checked. Abstractly, checking for cycles is a reachability problem. To determine which vertices are reachable from a given vertex all outgoing paths from the vertex must be followed and the vertices visited, enumerated. To find all cycles in a graph, this process must be done for each vertex in the graph. Once the set of reachable vertices is known for vertex V_i it can be checked to see if it contains V_i as an element. If so, then V_i is reachable from V_i and there is a cycle.

It is possible to decrease the size of the graph by restricting $LocalDAX$ to contain only vertices reachable from the transactions with which UT^i conflicts rather than vertices corresponding to *all* active transactions. This decreases the size of the graph and with it the cost of cycle detection. Another way of looking at this is that the test for acyclicity performed in Algorithm 5.6 is simpler than the general problem of detecting arbitrary cycles in a DAG. When testing for cycles in $LocalDAX$ it is known that any cycle *must* contain the vertex, V_T, which corresponds to the transaction being scheduled. Furthermore, it must also contain the "speculative" edge just added to the graph and the vertex it points to. This is true because the DAX from which $LocalDAX$ is built is always acyclic. By restricting cycle testing to start at V_T and include the speculative edge, overhead is reduced compared to the general cycle detection problem. This makes sense since the only nodes visited in the graph walk are those which are reachable from V_T via the speculative edge.

Algorithm 5.8 *Detect Cycles in LocalDAX including the edge* $V_{source} \rightarrow V_{sink}$

FUNCTION $CycleFree(LocalDAX, V_{source}, V_{sink})$ **RETURNS BOOLEAN** ;

INPUT : $LocalDAX$: ;
INPUT : V_{source} : ;
INPUT : V_{sink} : ;
OUTPUT : boolean indicating if cycles are present or not;

VertexQueue : queue of vertices in $LocalDAX$;
ReachableVertices : set of vertices in $LocalDAX$;
CurrVertex : vertex in $LocalDAX$;
ChildVertex : vertex in $LocalDAX$;

/* Breadth first search of graph from V_{sink} looking for V_{source} */
ReachableVertices $\leftarrow \emptyset$;
Mark all nodes in $LocalDAX$ as unvisited;
Mark V_{source} as visited;
Insert V_{sink} in VertexQueue;
WHILE (VertexQueue not empty) **DO**
    CurrVertex $\leftarrow$ remove(VertexQueue);
    IF (CurrVertex is marked visited) **THEN**
        RETURN (FALSE);
    Mark CurrVertex as visited;
    FOREACH (ChildVertex | $\exists$ an edge CurrVertex $\rightarrow$ ChildVertex) **DO**

```
                Insert ChildVertex in VertexQueue;
    RETURN (TRUE);
```

End of Algorithm

Algorithm 5.8 performs cycle checking as required by Algorithm 5.6. It scans only those vertices reachable from the vertex corresponding to the new transaction and containing the arc just added. Furthermore, as soon as *any* cycle is detected, the algorithm terminates.

5.4 Full Object Concurrency Control

Practical concurrency control in an objectbase system requires that both intra- and inter-transaction concurrency control be provided. The methods discussed in the previous two sections may be easily combined to provide full object concurrency control.

5.4.1 Integrating Intra and Inter-Transaction Concurrency Control

A straightforward way to provide full object concurrency control is to combine Algorithms 5.2 and 5.6. As stated in Section 5.2, the algorithm for intra-transaction concurrency control was designed for use with the one presented for inter-transaction concurrency control. As such, the algorithms are compatible and may be directly combined.

Theorem 5.4 *Algorithm 5.2 and Algorithm 5.6 are compatible and, together, provide full object concurrency control.*

Proof:

Both algorithms send messages to object schedulers specifying required execution orders and both are correct when executed by themselves. All possible concurrency is

covered by one algorithm or the other. Thus, it is sufficient to show that neither algorithm interferes with the other to show that they work together correctly.

The only way for the schedulers to interfere with one another is by specifying, directly or indirectly, contradictory execution orders. The global scheduler (for inter-transaction concurrency) specifies execution orders between operations from *different* user transactions. The root object scheduler (for intra-transaction concurrency) specifies execution orders between operation from the *same* user transaction.

An arbitrary ordering message generated for intra-transaction concurrency control has the form – $m^t_{ij} \Rightarrow m^t_{ik}$. An arbitrary ordering message generated for inter-transaction concurrency control has the form – $m^t_{ij} \Rightarrow m^s_{ik}$. A directly contradictory ordering message reverses the order of the method specifications in the original message. (e.g., $m^t_{ij} \Rightarrow m^t_{ik}$ is contradicted by $m^t_{ik} \Rightarrow m^t_{ij}$.) Neither algorithm contradicts messages of its own. If Algorithm 5.2 generates the message $m^t_{ij} \Rightarrow m^t_{ik}$ then Algorithm 5.6 cannot generate the contradictory message because all messages it generates specify *different* transaction identifiers (since it deals only with *inter-* transaction concurrency). The converse also holds. Thus, the two schedulers cannot possibly send messages which specify directly or indirectly conflicting method execution orders at an object. ■

To combine the two algorithms, no change is required to the specification of the global scheduler given in Algorithm 5.6. The only change required to the object scheduler specified in Algorithm 5.2 is to have it follow orders specified by the global scheduler. This requires a simple extension of the algorithm to support method invocation specifications which include transaction identifiers.

Transaction Nesting

The full object concurrency control algorithm produced by combining the algorithms for intra- and inter-transaction concurrency control provides an *open* nesting model by default. This is true since once the operations of a transaction have executed at an object,

the (possibly conflicting) operations of other transactions are free to execute immediately regardless of whether or not the preceding transaction has committed. This, of course, maximizes the potential concurrency but complicates recovery.

With simple modifications to Algorithms 5.6 and 5.2 it is also possible to achieve a *closed* nesting model. To provide closed nested transactions, each object scheduler must delay the execution of transaction operations until after the transaction(s) associated with preceding conflicting operations at the object have terminated. Only the global scheduler knows when user transactions have committed or aborted and thus, it must be involved in ensuring closed nesting.

A simple pre-commit protocol (e.g., Zapp and Barker [ZB93a]) provides the basis for closed nesting. When a sub-transaction executing at an object completes, it *pre-commits* rather than committing. Its parent transaction is informed of the pre-commit and the parent treats it as a commit for the purpose of scheduling additional operations from the *same* transaction. The local object scheduler however does not consider the sub-transaction to be complete and therefore ensures that no conflicting operations from *other* transactions are permitted to execute.

When the global scheduler receives a pre-commit message corresponding to a *user* transaction, it begins a proper commit of the user transaction by sending a commit request to the objects accessed by the user transaction. When an object scheduler receives a request to commit from its parent (global or object) scheduler, it in turn sends commit messages to the object schedulers corresponding to its sub-transactions. A transaction is committed only after all its sub-transactions have successfully committed. After a transaction commits, operations from other user transactions are permitted to proceed.

The pre-commits logically propagate up the method invocation tree while the commits propagate down.

5.4.2 Further Enhancing Concurrency

Due to the static generation of attribute reference sets, the information they contain is necessarily conservative. In the presence of run-time conditionals, the sets overspecify the set of attributes which are actually referenced by an execution of the corresponding method. Since the attribute reference sets are used to determine conflict relations, those relations are unnecessarily pessimistic for objects having methods containing run-time conditionals.

In Section 4.3.3 the derivation of control flow paths and control flow conditions for methods was discussed. The control flow paths through a method were enumerated as $CFPs(m_{ij})$ and each such path was marked as either "checkable" or "non-checkable" depending on whether or not the corresponding control flow conditions were based only on object attributes available prior to method execution. This information permits the methods of a class/object to be divided into three groups; those which contain a single control flow path (i.e., straightline code), those which contain only checkable control flow paths, and those which contain at least one non-checkable control flow path.

For methods which contain multiple, checkable control flow paths it is possible to test the relevant object attributes prior to scheduling to determine which control flow path will be taken. Once this is known, the corresponding set of attribute accesses can be used for determining conflicts. Since this set is a *subset* of the conservative estimate based on execution of all control flow paths, fewer conflicts with other concurrently executing methods will be detected (no "false" conflicts will be detected).

The ability to dynamically determine a control flow path and select the appropriate attribute reference set accordingly, does not affect the determination of execution and hence serialization orders by the global (root-object) scheduler for inter (respectively, intra) transaction concurrency control. This is because even if nested method invocations are checkable, it is not possible to perform the checking of the attribute values which determine the control flow path taken since they may not yet exist. Thus, execution

orders must be determined using only the conservative information which assumes all control flow paths are followed. This information may be summarized in a class-method conflict table for efficient checking of the "©" relation in Algorithms 5.6 and 5.2.

The execution orders determined using conservative information result in ordering messages being sent to object schedulers. The ordering messages explicitly specify *potentially* conflicting method execution pairs. If the corresponding object scheduler can determine, using CFP information, that the method executions will *not* conflict, then it is free to re-order their execution in contradiction to the ordering prescribed by the global (or root object) scheduler[10].

The control flow path information is used in each object scheduler in scheduling method invocations made on the object. For each edge $m_{ij}^{T_1} \rightarrow m_{ik}^{T_2}$ (T_1 possibly equal to T_2) in the LSG for the object, special processing is required (beyond that specified in Algorithm 5.2) to exploit CFP information. The processing needed is detailed in Algorithm 5.9.

Algorithm 5.9 *Using CFP information in Object Schedulers to Enhance Concurrency*

/* Assuming the edge $m_{ij}^{T_1} \rightarrow m_{ik}^{T_2}$ in LSG is the only one having $m_{ik}^{T_2}$ as its sink */
IF ($m_{ik}^{T_2}$ has arrived but $m_{ij}^{T_1}$ hasn't arrived) **THEN**
    IF (m_{ik} is checkable) **THEN**
        Determine CFP in m_{ik} and re-evaluate conflicts using the exact attribute reference information;
    IF (no conflict) **THEN**
        Remove edge from LSG and Let $m_{ik}^{T_2}$ execute immediately;
IF ($m_{ij}^{T_1}$ arrives while $m_{ik}^{T_2}$ is waiting) **THEN**
    Let $m_{ij}^{T_1}$ execute immediately;
    IF (either/both of m_{ij} and m_{ik} are checkable) **THEN**
        Determine CFP(s) and re-evaluate conflicts using the exact attribute reference information;
    IF (no conflict) **THEN**
        Remove edge from LSG and Let $m_{ik}^{T_2}$ execute immediately;

End of Algorithm

[10] Since non-conflicting operations do not affect serializability.

Care must be taken in determining the control flow path taken by a method execution to avoid accessing stale attribute values. Fortunately, if any currently executing method may write an attribute value required to determine the control flow path taken, then Algorithm 5.9 will not be executed This is because more than one method invocation must have $m_{ik}^{T_2}$ as its sink[11]. Thus, CFP testing is effectively postponed until all conflicting method executions complete.

The use of CFP testing clearly increases the concurrency achievable in transaction execution.

5.4.3 Heuristics to Minimize Overhead

There are a variety of heuristic techniques which may be applied to decrease overhead generally or in specific, but frequently occurring, cases. This section discusses two such possible techniques; caching inter-object conflict information, and the special case of concurrency control for simple objects.

Caching Conflict Information

It is possible to improve the performance of inter-transaction concurrency control using method reference sets. There are two reasons why inter-transaction conflict sets are not stored. First, there are many possible inter-object method pairs and this makes the storage of all possible conflict sets impractical. Second, many of the possible inter-object method pairs will never be invoked as concurrent user transactions and therefore should not be stored. Unfortunately, it is difficult to determine *à priori* which inter-object method pairs will be invoked concurrently.

The behaviour of concurrent user transactions is typically unpredictable but not irregular over time. This suggests that over a long duration, many identical pairs of trans-

[11] The values of the attributes used in determining control flow conditions are *read* by the method and therefore the method conflicts with any other method that *writes* any of the attributes.

actions may execute concurrently. When this is true, it is beneficial to cache the resulting conflict sets for future use. Such caching may take place in-memory and thus be volatile or on disk and thus persistent. Regardless, a fixed size area may be pre-allocated for caching conflict sets and maintained on a Least Recently Used (LRU) basis. Fast access can be provided by a hashing scheme using the *object_id*s and *method_id*s of the transactions to form a key.

Dealing with Simple Objects

The advantage of providing concurrency control using method reference information is best realized when accessing complex objects. Very simple (especially, non-nested) objects offer little (or no) sub-transaction concurrency to exploit. For this reason, it makes sense to consider simpler mechanisms for concurrency control for very small objects. A reasonable metric for judging whether or not it is worth applying method reference (i.e., *static*) information is the number of levels of sub-objects. Maintaining this information is easy and testing a scalar integer value is a small overhead in choosing a concurrency control strategy. Obviously, non-nested objects (having zero levels of sub-objects) do not require method reference techniques. Choosing a specific threshold value of the nesting depth to determine whether to use method reference information will depend on empirical testing.

A special case of "simple" objects are those which contain objects which are private to the enclosing object (i.e., cannot be accessed/shared by other objects). This dissertation precluded discussion of concurrency between such sub-transactions for the sake of simplicity. Such concurrency is, however, possible and requires only intra-transaction concurrency control. Because the only way to access the sub-objects is through their *single* parent, inter-transaction concurrency control need only be applied to the parent object. This suggests that "private" objects are desirable for simplified concurrency control. Of course, this conflicts with the goal of data sharing.

5.5 Multi-Version Concurrency Control

This section addresses the problem of providing concurrency control in multi-version objectbases. The approach taken is to develop optimistic protocols for transaction management and use *reconciliation* to recover from "conflicts" after they happen. The generation of reconciliation procedures is based on static information.

Recall from Section 4.2.3 that in the multi-version object model described, a new version of an object is derived from the LCV of the object for each user transaction. All method invocations made by a user transaction on an object are made on the transaction's active version. Thus, concurrent user transactions each operate on their own version of an object and conflicts, as they are commonly known, do not occur. This, however, does not prevent lost updates or the reading of stale attributes. To address these problems, when a user transaction commits, its active versions are *reconciled* with any versions of the objects committed since the user transaction derived its active versions. The reconciled versions become the last committed versions which are then used to derive new versions of the objects for subsequent transactions.

5.5.1 Reconciliation

Reconciliation may be required when two method invocations on an object execute concurrently. In this case, the object, O_i, is initially in some state s_i and two methods ($f = m^s_{ix}$ and $g = m^t_{iy}$) concurrently perform state transforming functions on their local copies of the object. If the methods conflict then the versions of O_i that f and g produce must be *reconciled* to produce a new valid version. Figure 5.10 illustrates the situation which gives rise to the need for reconciliation.

Reconciliation is most beneficial when the number of attributes involved in the conflict between two method executions is small compared to the total number of attributes referenced. In this case, only a small percentage of the accessed attributes have been

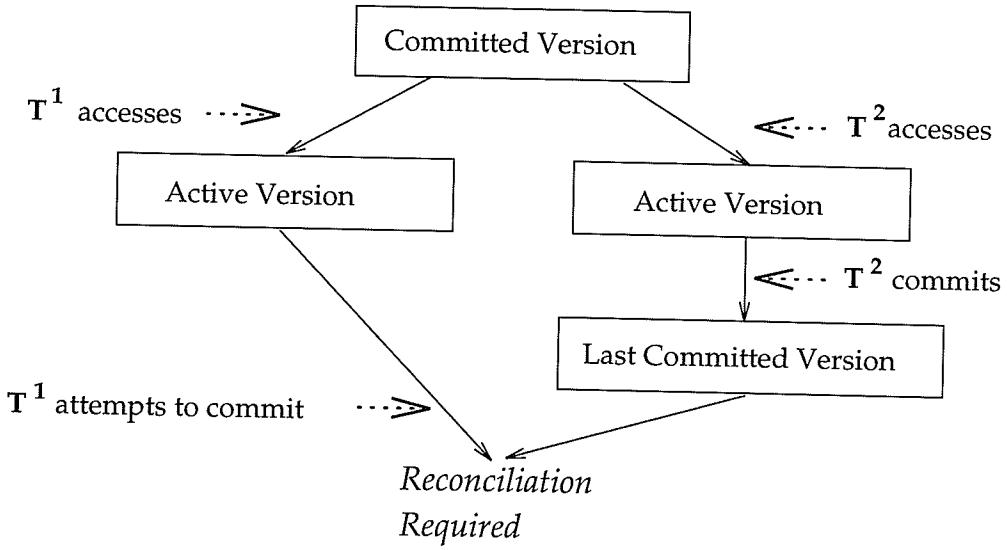

Figure 5.10: Reconciliation

incorrectly updated and roll back would result in a great deal of the method's work being unnecessarily lost. By reconciling rather than rolling back, this work is not lost[12] and a savings in execution overhead is achieved. The reconciliation process will totally preserve that part of a method execution's work which is non-conflicting and will re-execute only the conflicting part. In no case will reconciliation ever be more expensive than roll back and re-execution.

Dependences in method executions are respected. Thus, concurrent method executions on an object arising from *dependent* message steps from the same transaction do not occur. This means that the execution order of f and g is irrelevant as long as it is serializable. To be correct, the new object state after the concurrent execution of f and g must be either $f(g(s_i))$ or $g(f(s_i))$. There is, however, no guarantee that an uncontrolled concurrent execution of f and g will produce a final object state equivalent to either. In general, transaction T^s which executes f will produce a version of O_i in state $f(s_i)$ while transaction T^t (executing g) will produce a version of O_i having state $g(s_i)$ and these

[12] Additionally, the work of other method executions from the same transaction is not lost.

versions may not be "compatible" (i.e., $g(s_i) \neq f(s_i) \neq f(g(s_i)) \neq g(f(s_i))$). To know when concurrent executions will be incompatible it must be known when the method executions conflict.

The definition of conflict can be extended to versioned objects as follows:

Definition 5.7 *Version conflict* occurs when two versions of an object are produced by conflicting method executions on active versions of an object which are both derived (possibly indirectly) from the same base version. ■

When two concurrent method invocations are made against an object O_i, two active versions (O_i' and O_i'') are created and these may be conflicting versions. Four possible "Reconciliation Cases" arise, each of which must be resolved to ensure object consistency.

Reconciliation Case 1 – If f^s and g^t are both read-only transactions with respect to attributes at O_i, then no conflict occurred. Since both active versions are unchanged no new version needs to be committed.

Reconciliation Case 2 – If f^s is read-only at O_i but g^t is not, then the new object state must be set to $g^t(s_i)$ and similarly if g^t is read-only but f^s is not, then the new object state must be set to $f^s(s_i)$. The additional consequence of this is that transaction T^s must be serialized before T^t in the first case and T^t must be serialized before T^s in the second.

Reconciliation Case 3 – If f^s and f^t conflict and f^s is a costly operation[13] but g^t is not then set the new object state to be $f^s(s_i)$ and simply re-execute g^t. Similarly if g^t is a costly operation but f^s is not then set the new object state to be $g^t(s_i)$ and re-execute f^s.

Reconciliation Case 4 – If f^s and f^t conflict and both f^s and g^t are costly operations then apply a reconciliation procedure to combine the effects of f^s and g^t to produce a single new version having the consistent state $s_i' = f^s(g^t(s_i))$ or $g^t(f^s(s_i))$ whichever is cheaper to compute. The selection of reconciliation function determines the serialization order of transactions T^s and T^t.

Reconciliation Cases 3 and 4 require that action be taken if the two methods actually conflict. To determine conflict in these situations, the read set ($RS(m_{ik})$) and write set ($WS(m_{ik})$) of each method m_{ik} are used in the expected way.

[13] As judged by a statically derived estimate of the average number of instructions executed.

Condition 1: $WS(f^s) \bigcap WS(g^t) = \phi$
Condition 2: $WS(f^s) \bigcap RS(g^t) = \phi$
Condition 3: $RS(f^s) \bigcap WS(g^t) = \phi$

Figure 5.11: Intersection Conditions

Consider two method invocations f^s and g^t on O_i and the intersection conditions given in Figure 5.11. If these three intersection conditions hold then the updates made to the attributes of an object by the execution of the methods are distinct and independent so the resulting object versions do not conflict. This means that both versions can be committed and the transactions may be serialized in either order. In this case, only simple reconciliation (described later) is required unless both transactions are read-only in which case, *no* reconciliation is needed.

Intersection Condition 1 tests to ensure that the updates are to distinct object attributes. The subsequent conditions test to ensure that no transaction reads a stale attribute value (i.e., one which is reassigned by the other transaction). If only the first condition and *one* of the last two hold, then the Read-Write conflict may lead to an incorrect execution depending on the serialization order chosen. If the appropriate serialization order (which has the writer following the reader) is followed, then no error occurs despite the intersection of the read and write sets. For example, if conditions 1 and 2 hold but 3 does not, then there is no conflict between transactions T^s and T^t at object O_i provided that the transactions are serialized in the order $T^s \rightarrow T^t$. This is because an attribute read by f^s which is written by g^t is *not* stale. Since T^t serializes after T^s, f^s is expected to read the *old* value of the attribute from the base version of O_i.

When these conditions (1,2,3 or 1,2 or 1,3) hold and the appropriate serialization order is followed (for 1,2 and 1,3), it is possible to easily combine the effects of the two transactions using simple reconciliation.

Definition 5.8 *Simple reconciliation* is the process of reconciling f^s and g^t by setting

the new object state on an attribute by attribute basis as follows assuming that the base version of O_i used by f^s and g^t are the same.

If s_i is the original object state and s_i' is the new, consistent object state, then for each attribute a_{ix}:

$$s_i'(a_{ix}) = \begin{cases} f^s(s_i)(a_{ix}) & \text{if } a_i \in WS(f^s) \\ g^t(s_i)(a_{ix}) & \text{if } a_i \in WS(g^t) \\ s_i(a_{ix}) & \text{otherwise} \end{cases}$$

where the notation $f^s(s_i)$ indicates the state of object O_i produced by executing method f from transaction T^s on the initial object state s_i. ■

Complex reconciliation is required whenever two costly method executions produce *conflicting* versions (based on violation of the intersection conditions of Figure 5.11). Although several method invocations may execute concurrently at the same object, only the two invocation case is dealt with. This is all that is required because commitment is a non-concurrent process (i.e., no two concurrent, conflicting transactions may commit at the same time).

By considering the desired serialization order of the two transactions executing the conflicting methods (if there is one) and which of the three intersection conditions (Figure 5.11) apply, a *best* commit (and therefore, final serialization) order can be determined.

The results of the method execution which serializes (and commits) first are correct without modification but those of the other method execution must be changed so that they reflect the results which would have been produced had the initial state of the object used been that produced by the the first method's execution. Which object attributes must be recalculated to achieve this effect can be easily determined. Furthermore, it is known that those attributes must be recalculated according to the execution semantics of the *second* method. Thus, if the operations m_{ij}^s and m_{ik}^t are to be serialized in the

order $T^s \rightarrow T^t$ and the single intersection condition $WS(m^s_{ij}) \cap RS(m^t_{ik}) = \phi$ is violated then any computation performed in m^t_{ik} which relies on an attribute value in $WS(m^s_{ij}) \cap RS(m^t_{ik})$ must be re-executed using the state $m^s_{ij}(s_i)$ for input. Complex reconciliation can be viewed as a *partial* re-execution of one of the methods involved.

Generating Reconciliation Procedures

The process of reconciliation is accomplished by the invocation of statically generated *reconciliation procedures*. Given a pair of conflicting method executions and a serialization order, a specific reconciliation procedure must be invoked at commit time to ensure that the correct object state is produced.

The reconciliation procedures described do not attempt to exploit *high level* semantics as in Weihl's work on commutativity [Wei88] or in Weikum's work on recovery using compensating transactions [Wei91]. It is the need to understand high level program semantics that makes the automatic generation of commutativity tables and compensating transactions difficult. Each reconciliation procedure is a subset of one of the methods being reconciled. The determination of the required subset is accomplished by exploiting data dependence [Ban88, ZC90] techniques.

Some additional definitions are required to discuss the generation of reconciliation procedures.

Definition 5.9 Given two conflicting operations m^s_{ij} and m^t_{ik} to be serialized in the order $T^s \rightarrow T^t$, the *conflict set* for reconciliation specifies which attributes of O_i must be re-read from the state $m^s_{ij}(S_i)$ by the partial re-execution of m^t_{ik} and is defined as: $\mathcal{CS}(m^s_{ij}, m^t_{ik}) = WS(m^s_{ij}) \cap RS(m^t_{ik})$ ■

Definition 5.10 A computation step $s_p \in m^t_{ik}$ reads a *critical input attribute* a_{ix} if it accesses it for reading, if $a_{ix} \in \mathcal{CS}(m^s_{ij}, m^t_{ik})$, and if there is no output dependence $s_q \delta^{o*}_{a_{ix}} s_p$ for any $s_q \in m^t_{ik}$. ■

The dependence graph of a method $DG(m_{ik})$ effectively abstracts the computation performed by the method. Each reconciliation procedure can be similarly abstracted as a sub-graph of the dependence graph of the method. For each pair of methods m_{ij} and m_{ik} in object O_i two reconciliation procedures $reconcile(m_{ij}, m_{ik})$ and $reconcile(m_{ik}, m_{ij})$ must be generated, one for each possible serialization order ($m_{ij} \Rightarrow m_{ik}$ and $m_{ik} \Rightarrow m_{ij}$).

Definition 5.11 A *reconciliation procedure* $reconcile(m_{ij}, m_{ik})$ is abstracted by the sub-graph of $DG(m_{ik})$ consisting of those vertices which are dependent on the critical input attributes in $\mathcal{CS}(m_{ij}, m_{ik})$ and the edges which connect those vertices. ■

The automatic construction of an actual reconciliation procedure given its corresponding dependence sub-graph substitutes the appropriate computation steps from m_{ik} for each vertex in the sub-graph and emits the resulting code in an order given by a topological sort of the computation steps which is consistent with the partial order determined by the edges of the sub-graph (i.e., consistent with $\prec_\delta$).

To help clarify the construction of reconciliation procedures a simple example is given in Figure 5.12. A class specification with two methods "IncUpper" and "DecLower" are shown as are the unravelled forms of the two methods. The conflict set ($CS(IncUpper, DecLower, IncUpper \Rightarrow DecLower)$) and required reconciliation procedure are shown assuming that two concurrent transactions execute these methods and that a conflict occurs and requires that the execution of "DecLower" serialize after the execution of "IncUpper". The example has no inter-step dependencies and hence, the dependence graphs are not shown. Thus, the reconciliation procedure consists simply of those statements in "DecLower" which read the diagonal elements of "a" written by "IncUpper".

```
CLASS C;

ATTRIBUTE a: ARRAY[1..3,1..3] OF INTEGER;

METHOD IncUpper()
i,j : INTEGER:
BEGIN
FOR i:=1 TO 3 DO
    FOR j:=1 TO i DO
        a[i,j]++;
END

METHOD DecLower()
i,j : INTEGER;
BEGIN
FOR i:=1 TO 3 DO
    FOR j:=i TO 3 DO
        a[i,j]--;
END
```

Original Class Specification

```
METHOD IncUpper()
BEGIN
a[1,1]++;
a[2,1]++;
a[2,2]++;
a[3,1]++;
a[3,2]++;
a[3,3]++;
a[4,1]++;
a[4,2]++;
a[4,3]++;
a[4,4]++;
END
```

Unravelled Code for IncUpper

```
METHOD DecLower()
BEGIN
a[1,1]--;
a[1,2]--;
a[1,3]--;
a[1,4]--;
a[2,2]--;
a[2,3]--;
a[2,4]--;
a[3,3]--;
a[3,4]--;
a[4,4]--;
END
```

Unravelled Code for DecLower

CS(IncUpper,DecLower,IncUpper=>DecLower) = { a[1,1], a[2,2], a[3,3], a[4,4] }

```
METHOD ReconcileIncUpper-DecLower()
BEGIN
a[1,1]--;
a[2,2]--;
a[3,3]--;
a[4,4]--;
END
```

Reconciliation Procedure

Figure 5.12: Example Reconciliation Procedure

5.5.2 Multi-Version Concurrency Control with Flat Transactions

Discussing conflicts and the reconciliation process as if they were applicable to exactly two *active* transactions attempting to commit *simultaneously* is unrealistic. It is unacceptable to postpone the commitment of one transaction while waiting for another to complete (especially with potentially long-lived transactions). Thus, in Algorithm 5.10 reconciliation is applied in a way which is compatible with a single transaction committing at a time.

The Algorithm

Algorithm 5.10 *Multiversion Concurrency Control using Reconciliation*

PROCEDURE Version_Transaction_Scheduler (T^i) : **RETURNS** (result);

INPUT T^i : transaction to be executed;
OUTPUTresult : boolean; /* TRUE/commit or FALSE/abort */

$\mathcal{BS}^i \leftarrow$ create set of objects read/written by T^i;
FORALL $O_k \in \mathcal{BS}^i$ **DO**
    derive an *active* object $O_k^{i\prime}$ from the LCV of O_k;
execute T^i: mapping each operation on O_l to an operation on $O_l^{i\prime}$

IF (T^i aborts) **THEN**
    abort T^i or a sub-transaction thereof
    exit (FALSE);
IF (T^i is prepared–to–commit) **THEN**
    FORALL $O_k \in \mathcal{BS}^i$ **DO**
        lock (O_k);
        IF (O_k's LCV is not the base version of $O_k^{i\prime}$) **THEN**
            /* A new LCV exists which resulted from a *single* method */
            /* execution operating on the same base version as O'_k */
            IF (case 2 and $O'_k = LCV$) **THEN**
                /* reconciliation is needed since we read stale data */
                complex_reconcile($LCV(O_k), O'_k$) ;
            ELIF (case 3) **THEN**
                /* reconciliation is cheapest operation so do it */
                complex_reconcile($LCV(O_k), O'_k$);
            ELIF (case 4) **THEN**
                IF (conditions in Figure 5.11 are met for the
                serialization order $T^{LCV} \rightarrow T^i$ at O_k) **THEN**
                    simple_reconcile($LCV(O_k), O'_k$;

ELSE
complex_reconcile$(LCV(O_k), O'_k)$;
ELSE ; /* this version is OK */
BEGIN_ATOMIC;
FORALL $O_k \in \mathcal{BS}^i$ **DO**
commit $O_k^{i'}$ as O_k if $O_k^{i'}$ was updated;
unlock (O_k);
END_ATOMIC;
EXIT (TRUE);

End of Algorithm

Correctness

The correctness of Algorithm 5.10 is predicated upon the execution being equivalent to some serial execution – the standard definition of serializability.

If no concurrent execution of methods occurs at any object, then the multi-version execution is equivalent to a non-multi-version serial execution and is therefore correct. This is true because the commitment order is serial at every object. Since commitment involves generating a new LCV, which will be used by subsequent transactions, and because of the transactions' atomicity and durability properties, it is guaranteed that a subsequent method execution will see only correct, committed object state information when it derives its active version.

When two concurrent method executions occur at an object, one will finish and commit first. According to the algorithm presented, this determines a serialization order which is equivalent to the commit order[14]. The committing transaction produces a new LCV. In any serial execution, it is this version of the object which must be used as the base version for method invocations arising from any transaction serializing after the one which produced the new LCV. Since concurrency has occurred, the second method execution may have seen stale data from the previous LCV (its base version). For concurrent execution to be correct, the second method execution must execute as if it had read the

[14]This need not be the case as will be seen shortly.

new LCV and used it as its base version.

If the concurrent method executions do not conflict given the serialization order determined by their transaction's commitment order then the second method execution can commit freely without reconciliation and since it did not conflict with the first, it can safely generate a new LCV. If both method executions were read-only then no new LCV need be written by either committed transaction. If only the second method execution writes attributes, then it simply commits creating a single new LCV which is correct and consistent. If the first method execution writes attributes then, since there is no conflict between the method executions, it must have written data that cannot affect the execution of the second method execution. This is the simple reconciliation case. The second transaction cannot be allowed to commit and write its version back as the new LCV since this would not correctly reflect the updates made to object attributes by the first method execution (i.e., the lost update problem). This is corrected by creating a new LCV which consists of the updated attributes produced by *both* method executions. Effectively, the updates of the first transaction are propagated into the version produced by the second transaction and then that version is committed.

Finally, if the method executions do conflict then complex reconciliation must be applied. To produce a correct execution, the result of complex reconciliation must be a version of the object in a state equivalent to that which would have resulted from the serial execution of the two method executions in the order of their transactions' serialization. This is exactly the effect provided by executing a reconciliation procedure constructed in the fashion described earlier. Since any operation within the second method which depends on an attribute written by the first is re-executed as a part of the reconciliation procedure and since the execution order of statements within the reconciliation procedure adheres to the serial semantics (i.e., dependencies) of the original method, correctness is assured.

Inter-Object Serializability

The algorithm just presented ensures only *object-local* serializability. It is, of course, possible that conflicting serialization orders may be chosen at different objects thereby making the corresponding transaction executions non-serializable. To address this problem, a global scheduler (such as that described in Section 5.3.1) may be used which dynamically builds and maintains a graph of inter-transaction serialization orders and verifies that serialization orders selected locally at each object are in fact globally serializable.

Simply checking for directly conflicting serialization orders (e.g., $T^1 \Rightarrow T^2$ at O_i and $T^2 \Rightarrow T^1$ at O_j) is insufficient, serialization orders may *indirectly* conflict. To detect this, the graph of serialization orders must be checked for cycles of arbitrary length. The method (i.e., global) scheduler of Zapp and Barker [ZB93c, ZB93a, ZB93b] performs this checking every time a serialization order is reported by an object scheduler. This allows conflicts to be detected early so that one of the transactions involved may be aborted. Given reconciliation, the communication overhead between the global scheduler and the object schedulers, may be decreased by only reporting serialization orders when sub-transactions pre-commit.

When the global scheduler determines that a *user* transaction is ready to commit, it must verify that all of the object versions it produced (via pre-committed sub-transactions) may be committed. This is true for each version, only if the LCV of the object is the same as the base version from which the version to be committed was derived. If not, then at least one other transaction has committed and the committing object version must be reconciled. Due to the need to provide inter-object serializability it may not be possible to commit the new transaction's version as the new LCV since this may affect the serialization order. In such cases, either an alternative commit point must somehow be chosen or the committing transaction must be aborted.

Changing the Commit Order

Rather than aborting a transaction, in some cases it may be possible to change the commit order at the object(s) where conflict occurs. Doing this requires the cooperation of the global and object schedulers. The global scheduler must determine that the problem exists and ask the object scheduler to determine the possible commit order changes that could correct the problem. Given this information, the global scheduler can determine which order, if any, will permit successful transaction commitment. If one is found, then the global scheduler must explicitly request it during commit processing.

To determine that a change in commit order is worthwhile, three conditions must be met:

1. the object scheduler must track who wrote each version of an object so that it can report back possible serialization order changes,
2. the object scheduler must preserve all previous LCVs which are being used as the base version of uncommitted transactions, and
3. the global scheduler must be able to explicitly associate a DAX edge with a particular conflict at a given object.

When it is determined that a transaction has committed a new version too soon (based on an inter-object conflict) it may be desirable to commit a later transaction before it. This can be done by appropriate complex reconciliation in the new version (if necessary) and by simple reconciliation in those versions already committed but now logically serialized *after* the new one. The simple reconciliation serves to propagate the results of those transactions pushed back in the serialization order to those after them. Further consideration of the details of changing the commit order is beyond the scope of the dissertation.

5.5.3 Multi-Version Concurrency Control with Nested Transactions

Ignoring nested transactions when using multi-version concurrency control does not decrease concurrency (since all method invocations are optimistically executed on their own

versions of objects) but it does impact recovery. Nested transactions have the effect of decreasing the granularity of recovery. In systems where "abort and re-execute" is used for recovery, this means that only certain sub-transactions need be aborted and re-executed. Similarly, if sub-transactions are provided in a system supporting reconciliation then only certain sub-transaction invocations need to be reconciled.

In conventional (read/write model) databases, the relationships between sub-transactions are straightforward and thus, it is easy to tell if the re-execution of a sub-transaction may require the re-execution of other sub-transactions. This happens when the results produced by a re-executed sub-transaction are expected to be read by other sub-transactions. Since the results of the original sub-transaction were made visible to the other sub-transactions of the user transaction prior to user transaction commit, those other sub-transactions executed using stale data. Thus, a form of cascading aborts limited to the scope of a single user transaction results.

Wherever cascading aborts may occur, it is also possible that cascading reconciliation may be required. The problem of supporting nested transactions for multi-version concurrency control with reconciliation is how to determine which sub-transactions must be reconciled when a given sub-transaction fails. This information may be determined, conservatively, using static information.

Due to the encapsulation property of objects, the only way that a sub-transaction may affect the execution of other sub-transactions on different objects is via its interface. These affects may be divided into two categories;

downward effects – those effects caused by method invocations made by the sub-transaction being reconciled, and

upward effects – those effects caused by changes to values returned by reconciled method invocations which may affect the flow of control or method invocations of the parent transaction

Upward and downward effects determine which sub-transactions require reconciliation. Upward effects may cause additional effects at object methods not directly beneath or above (in terms of the method invocation hierarchy) the failed sub-transaction.

The determination of the upward and downward effects of a failed sub-transaction is a clear area for the application of static analysis. Discussion of this determination is beyond the scope of the dissertation.

Chapter 6

Recovery Using Static Information

This chapter discusses the implementation of object base recovery using static information to enhance performance.

6.1 Recovery in Object Bases

The problem of recovery in objectbases is not significantly different from recovery in other systems which support nested transactions. The same basic failure modes arise; (1) aborts of transactions and/or sub-transactions, and (2) system failures. These failures must be dealt with and much of the same information must be logged to facilitate this (e.g., beginning and end of transactions, the operations they perform, and the relationships between them). The defining characteristics of objects (e.g., inheritance, encapsulation, etc.) that distinguish them from other environments that support nested transactions have little or no effect on recovery issues. This applies to both physical and logical logging.

6.1.1 Recovery Using Physical Logging

Recovery based on the use of physical logging (i.e., logging of writes to object attributes) is straightforward. At this level of abstraction, transactions in objectbases, as described in earlier chapters, are simply nested transactions and the work done on recovery in nested transaction systems (e.g., [Mos87, HR87, RM89]) applies.

The focus in the dissertation is on in-place updating rather than shadow techniques. This permits easy comparison of the algorithms presented with the bulk of the related work which also focuses on in-place updating. Both physical and logical logging are considered.

In general, the recovery log must contain the following information to support recovery from aborts and/or system failures in objectbases:

- Before-images of object data (unless shadowing techniques are used)
- After-images of updated object data (unless logical logging is used)
- Transaction start, commit, and abort information (including the relationship between sub-transactions of the same user transaction)
- Checkpoint information if checkpointing is used to limit restart overhead

When using in-place updating with write-ahead logging, before-images must be written to the log before updates can take place. The information contained in each resulting log record must contain the attribute identifier and the *committed* value of the attribute at the start of the transaction for which the before-image is being written and the transaction identifier of the transaction. The committed value may be found either in the cache (if it was recently written there) or in the stable objectbase itself. The format of a before-image log record is "$BI(T^k, a_{ij}, oldvalue)$" where T^k identifies the transaction whose execution required the generation of the before-image log record, a_{ij} is the attribute whose before-image is to be preserved, and *oldvalue* is the committed value of the attribute (i.e., the before-image itself).

In physical logging, the results of every write operation performed by a transaction must be recorded in the log. These values are the after-images produced by the write operations. Since writes may be performed on the same attributes by multiple transactions, it is also necessary to be able to determine which transaction produced a given after-image. This leads to after-image log records having the format "$AI(T^k, a_{ij}, value)$" where T^k is the transaction doing the write, a_{ij} is the attribute being updated, and *value* is the value being written to a_{ij}. One such log record is written for each update operation performed by an active transaction.

It must be known during recovery, whether a given transaction committed, aborted, or is still active. This is needed, for example, during restart processing when the effects of all *committed* transactions must be preserved while the effects of all *aborted* or *active* transactions must be discarded. To accomplish this, the log must contain information that delimits the extent of each transaction's processing and an indication of whether it committed or aborted if it completed. This can be accomplished by writing log records that indicate the beginning ("$BT(T^i)$") and end ("$ET(T^i, Commit)$" or "$ET(T^i, Abort)$") of each user transaction and sub-transaction[1].

The expected commit semantics of nested transactions [Mos85] dictate that committed results of *sub*-transactions are not durable except in the context of a committed parent transaction. By extrapolation then, during recovery, the effects of committed sub-transactions must be discarded unless the sub-transaction's user transaction committed. This means that the parent-child relationship between transactions must also be recorded in the log.

To associate a child transaction with its parent, a log entry must be made for each method invocation (corresponding to a sub-transaction creation) performed by a transaction. Such a log record has the form "$MI(T^i, T^j)$" where T^i is the transaction making the method invocation and T^j is the sub-transaction which results. Because of the pre-

[1] "$ET(T^i, Commit)$" records are written for sub-transactions at pre-commit time.

$$
\begin{array}{c}
BT(UT^1) \\
BI(X, 27) \\
BI(Y, 444) \\
BI(Z, \text{“}fred\text{”}) \\
AI(Y, 222) \\
MI(UT^1, ST^{1.1}) \\
BT(T^{1.1}) \\
BI(Q, 23.45) \\
AI(Q, 1.99) \\
ET(T^{1.1}, Commit) \\
AI(Y, 888) \\
AI(Z, \text{“}tina\text{”}) \\
MI(UT^1, T^{1.2}) \\
BT(T^{1.2}) \\
BI(B, True) \\
AI(X, 0) \\
AI(B, True) \\
ET(T^{1.2}, Commit) \\
ET(UT^1, Commit)
\end{array}
$$

Figure 6.1: Example Objectbase Log Segment

commit protocol associated with the use of nested transactions, all sub-transactions will complete (pre-committing or aborting) prior to the completion of their parent transaction and thus, in the log, the "BT" and "ET" records of a parent transaction will completely enclose all the log records corresponding to the operations of the sub-transactions. Log records for operations from other, non-conflicting, transactions may be interleaved with the sub-transaction's log records.

In the example log shown in Figure 6.1, for clarity, user transactions are denoted UT^i while sub-transactions are denoted $T^{i.j}$. For simplicity, only the operations of a single user transaction are shown in the figure so there is no interleaving between operations arising from different user transactions. Note, however, that there is interleaving between the operations of the concurrently executed sub-transactions of UT^1. Specifically, the update to X by UT^1 is done concurrently with the execution of sub-transaction $ST^{1.2}$ and the log reflects the interleaving of these (*non-conflicting*) operations since "$AI(X, 0)$" occurs in the

log within the bounds established by "$ST(T^{1.2})$" and "$ET(T^{1.2}, Commit)$". The example log represents a serializable but not serial execution. Despite this, *all* the operations of UT^1 and its sub-transactions are enclosed within "$BT(UT^1)$" and "$ET(UT^1, Commit)$".

The processing of these log record types during abort and restart will be described later in the specification of "reference" recovery algorithms.

6.1.2 Recovery Using Logical Logging

Objectbases provide a natural environment for the application of logical logging. Each method invocation on an object corresponds to a logical (i.e., object) operation. Given the current state of an object, logical logging of the invoked method's identifier and its input parameters provides sufficient information to allow redo of the operation. Given the potential complexity of object methods (a single method invocation may make many attribute updates), logical logging will significantly decrease the amount of log information which must be written. This decreases the corresponding I/O overhead associated with writing the log.

Logical logging requires only simple changes to the physical log records just discussed. There are three fundamental differences. First, there is no need to write after-images to the log since redo is implemented by re-execution of the logical operations recorded in the log. Second, the before-images which are written to the log consist not of individual data items and their values but of the entire object state (i.e., *all* attribute values). Finally, the method invocations are logged together with their input parameter values.

Using logical logging undo requires the restoration of an object's before-image. Redo requires the re-execution of committed transaction operations on the object's before-image given the same input parameter values. These operations may be easily implemented during restart processing.

6.1.3 Environmental Effects

The environment in which objectbases are expected to be used affects the design of recovery protocols. In the advanced software systems that objectbases are designed to support, a fundamental requirement is performance. Interactive users expect their applications to run quickly. The existing, non-objectbase, implementations of such applications, provide high-performance by sacrificing many of the benefits of objectbases including automatic recovery.

Transaction aborts are relatively frequent compared to system crashes. Given the emphasis on transaction performance, the recovery scheme chosen should minimize the cost of recovering from transaction aborts even at the expense of added overhead when recovering from system failures. During the actual process of recovery, extra overhead is acceptable because it provides the benefit of the successful completion of committed operations despite failures.

6.2 Implementing Recovery Using Static Information

Recovery mechanisms for objectbase systems which exploit static information must concentrate on limiting I/O overhead (cache flushing and log writing) during transaction execution. The overhead of flushing and logging may be limited by either decreasing the *apparent* overhead or by limiting the amount of information written to disk. Both approaches are feasible and are described below.

To simplify the initial presentation of the algorithms in this chapter, the cache is assumed to operate on a per-attribute basis. In practice, real cache managers move *fixed size* units of storage between the objectbase or log and the cache. Normally the unit of transfer is the *page* (or *block*) and its size is determined by the I/O hardware's unit of *atomic* transfer. Since objects and/or their attributes may be larger or smaller than a single page, problems arise that affect the efficiency of both concurrency control and

recovery (e.g., false sharing [HP90]). For simplicity, these problems are ignored, and it is assumed that the unit of transfer between the cache and stable storage is the *object attribute* which has no particular fixed size. This assumption permits concentration on the necessary issues without undue concern over specific implementation details.

6.2.1 Reference Recovery Algorithms

Reference recovery algorithms are now presented which provide unoptimized recovery for transactions in objectbase systems. Both undo and redo are assumed to be required so that the algorithms are as general as possible. Choosing a recovery strategy that requires both undo and redo also minimizes the amount of overhead introduced during normal transaction processing. This is in accordance with the expressed desire to minimize such overhead at the expense of increased cost during restart processing. The reference algorithms will be improved in various ways in later sections by the application of static information. Algorithms for undo and redo are also presented.

Extending the nomenclature of Bernstein, *et al.* [BHG87], seven recovery manager ("RM") routines are defined that are called by the transaction scheduler to accomplish a transaction's I/O related operations. The recovery manager interface to the cache manager is defined by the following functions:

RM_Start – called when a new (user or sub) transaction begins execution

RM_Read – called when a transaction executes a read of an attribute

RM_Write – called when a transaction executes a write to an attribute

RM_SubTrans – called when a transaction starts a sub-transaction (i.e., at the point of the method invocation)

RM_Commit – called when a transaction commits

RM_Abort – called when a transaction aborts

RM_Restart – called to restore the objectbase to a consistent state when the system is recovering after a failure

The processing performed by these routines is now discussed. Certain subroutines are used throughout the algorithm specifications. They are the following:

Cache_Read – Reads the specified attribute from its location in the cache and returns its value.

Cache_Write – Writes the specified value to the specified attribute's location in the cache.

Log_Write – Writes the specified information to the log.

Fetch – Fetches the specified attribute from the objectbase into the cache.

Flush – Flushes the specified attribute from the cache to the objectbase.

Normal Processing

When a transaction begins, all that has to be done is to write a "beginning of transaction" record to the log. Once this has been done, the transaction may issue read, write, sub-transaction, commit, and/or abort operations and the corresponding log entries will be sequenced properly in the log.

Algorithm 6.1 *Reference RM_Start Algorithm*

INPUT T^k : Transaction being started;

Log_Write("$BT(T^k)$");

End of Algorithm

During the processing of transaction read and write operations on object attributes, the only processing required to support recovery is the logging of the after-images of written attributes. Additional processing is also required to manage cache loading and flushing.

When a read of an attribute is issued for some transaction, the requested data is simply returned if it is found in the cache. If it is not found in the cache, the cache manager fetches the data and it is then returned from the cache. The need to fetch a

new attribute into the cache means that a cache replacement may be performed as a side-effect of the reading of an attribute value.

Algorithm 6.2 *Reference RM_Read Algorithm*

INPUT i, j : Identifier of attribute to be read;
OUTPUT a_{ij} : Value of attribute read;

IF (a_{ij} is not cache resident) **THEN**
    Fetch(a_{ij});
RETURN (Cache_Read(a_{ij}));

End of Algorithm

When a write of attribute a_{ij} is issued for transaction T^k, an after-image must be written to the log. If the data item to be written is not found in the cache, it must be loaded (thus, a write may also result in cache replacement). An after-image record ("$AI(T^k, a_{ij}, value)$") is then written to the log and the cache is updated with the new value for a_{ij}.

Algorithm 6.3 *Reference RM_Write Algorithm*

INPUT T^k : Transaction issuing the write;
INPUT a_{ij} : Attribute to be written;
INPUT Value : value to be written;

IF (a_{ij} is not cache resident) **THEN**
    Fetch(a_{ij});
Log_Write("$AI(T^k, a_{ij}, Value)$");
Cache_Write(a_{ij},Value);
RETURN

End of Algorithm

In addition to issuing reads and writes, in a nested transaction environment, a transaction may also create sub-transactions. In an objectbase system, this is done by invoking methods on objects. When a transaction makes a method invocation to create a sub-transaction, the RM_SubTrans routine is invoked on its behalf. This routine records the necessary method invocation (MI) record in the log.

Algorithm 6.4 *Reference RM_SubTrans Algorithm*

INPUT T^i : Transaction making the method invocation;
INPUT T^j : Transaction created as a result of the invocation;

Log_Write("$MI(T^i, T^j)$");
RETURN

End of Algorithm

Commit Processing

When a transaction is ready to commit, an end of transaction record indicating commitment must be written to the log.

Algorithm 6.5 *Reference RM_Commit Algorithm*

INPUT T^k : Transaction being committed;

Log_Write("$ET(T^k, Commit)$");

End of Algorithm

Abort Processing

When a *sub*-transaction aborts, it may be possible to re-execute the sub-transaction rather than actually aborting it [Mos85]. The mechanisms by which this is attempted are beyond the scope of the dissertation. Therefore, it is assumed that if a sub-transaction aborts, its enclosing user-transaction must be aborted and thus the dissertation focuses on user transaction aborts only.

When a *user* transaction aborts, recovery processing must be performed. No redo processing is required, but the objectbase system must ensure that any updates performed by the user transaction or any of its sub-transactions are undone. Thus, abort processing consists of writing an abort record in the log for each sub-transaction and the user transaction itself. This must be done in such a way that each pair of "BT" and "ET"

records encloses all of the operations of the corresponding transaction[2] so that the correct before-images will always be restored. Once this has been done, the undo algorithm is invoked to complete the abort.

Algorithm 6.6 *Reference RM_Abort Algorithm*

INPUT T^k : Transaction being aborted;

FOREACH (direct or indirect sub-transaction ST of T^k in depth-first order) **DO**
Log_Write("$ET(ST, Abort)$");

/* now it is safe to write the end of transaction record for the parent */
Log_Write("$ET(T^k, Abort)$");
undo(T^k);

End of Algorithm

Undo is accomplished in the following manner. The log is scanned backwards from the user transaction abort record to the user transaction start record. Because the "BT" and "ET" records enclose all operations of the transaction to be aborted, (including those of its sub-transactions), this is the entire subset of the log records that must be scanned. During the backward scan, log entries recording before-images generated due to the execution of the user-transaction to be aborted (or its sub-transactions), must be processed to restore the before-images they contain. This process must also be applied ("recursively") to the sub-transactions created by the execution of the transaction being undone. This processing is detailed in Algorithm 6.7.

Algorithm 6.7 *Reference Transaction Undo Algorithm*

PROCEDURE undo(T^k);

INPUT T^k : Transaction to be undone;

Posn ← EndOfLog;
/* skip over irrelevant records logged after the abort of T^k */
WHILE ($LogRecord(Posn) \neq ET(T^k, Abort)$) **DO**

[2]This is easily accomplished by generating sub-transaction abort log records in a depth-first manner.

```
    Posn ← previous record in log;
/* process before-image records between BT(T^k) and ET(T^k, Abort) */
WHILE (LogRecord(Posn) ≠ BT(T^k)) DO
    IF (LogRecord(Posn) = BI(T^k, a_ij, value)) THEN
        a_ij ← value;
    IF (LogRecord(Posn) = MI(T^k, T^l)) THEN
        /* T^l is a sub-transaction of T^k */
        undo(T^l);
    Posn ← previous record in log;
```

End of Algorithm

Algorithm 6.7 processes sub-transactions by recursively calling itself. This is clearly impractical since it will result in multiple passes over the log entries on disk (resulting in high I/O overhead). This can be avoided by explicitly chaining the BI records for a user transaction and its sub-transactions together in the log. If this is done, a single sequential scan of the log between the user transaction "BT" and "ET" records will suffice. This approach is used in ARIES/NT [RM89].

The process of recovering from abort using this undo procedure is idempotent so no special steps are needed to ensure that a system failure during the undo will be recovered from properly.

Restart Processing

After a system failure, recovery processing must both undo the actions of uncommitted transactions (and redo the actions of committed transactions) which were (not) reflected in the stable objectbase.

In the first phase of restart processing, the log must be scanned to determine which transactions must have their effects undone and which must have their effects redone. Once the set of transactions to undo and redo is determined, the actual processing of the undos and redos may be performed during a second phase.

Algorithm 6.8 *Reference RM_Restart Algorithm*

```
UndoList : list of transactions to be undone;
RedoList : list of transactions to be redone;

/* determine which transactions must be undone and redone */
FOREACH (log record from last to first) DO
    IF (log record is ET(T^i, Abort)) THEN
        add T^i to UndoList;
    IF (log record is ET(T^i, Commit)) THEN
        add T^i to RedoList;
    IF (log record is BT(T^i)) THEN
        IF (no corresponding ET(T^i) has been seen) THEN
            /* transaction was active so undo it */
            add T^i to UndoList;
/* perform undos */
FOREACH (transaction T^k on UndoList in reverse order of their initiation) DO
    undo(T^k);
/* perform redos */
FOREACH (transaction T^k on RedoList in order of their initiation) DO
    redo(T^k);
```

End of Algorithm

The undo processing of the user transactions during restart is the same as that described for abort processing. Redo processing of the transactions identified for redo is performed as follows. The log is scanned forward from the the user transaction start record to the user transaction commit record (which must exist for transactions being redone). Once again, the "TS" and "TE" records delimit the subset of the log records that must be scanned. During the forward scan, log records which record after-images of the committed transaction to be redone must be processed to reapply the after-images they contain. This process must also be applied ("recursively") to the sub-transactions created by the execution of the committed transaction. This processing is detailed in Algorithm 6.9.

Algorithm 6.9 *Reference Transaction Redo Algorithm*

PROCEDURE redo(T^k);

INPUT T^k : Transaction to be redone;

Posn ← EndOfLog;
/* skip over records logged after the start of T^k */
WHILE ($LogRecord(Posn) \neq BT(T^k, Commit)$) **DO**
    Posn ← previous record in log;
/* process before-image records between $BT(T^k)$ and $ET(T^k, Commit)$ */
WHILE ($LogRecord(Posn) \neq ET(T^k)$) **DO**
    IF ($LogRecord(Posn) = AI(T^k, a_{ij}, value)$) **THEN**
        a_{ij} ← value;
    IF ($LogRecord(Posn) = MI(T^k, T^l)$) **THEN**
        /* T^l is a sub-transaction of T^k */
        redo(T^i);
    Posn ← next record in log;

End of Algorithm

Again, this algorithm makes unrealistic recursive calls which may be replaced by the use of chained log entries in a practical implementation.

A number of optimizations to these basic algorithms, made possible by static analysis, are now presented.

6.2.2 Reducing the Number of Cache Flushes and After Image Log Records

Cached data is often written to stable storage unnecessarily often because the cache manager must, when it runs out of available cache space, flush some attribute(s) to disk. The problem is it makes the decision on which attribute(s) to flush without any *semantic* information about their future usefulness. Since the cache manager cannot dynamically examine the executing code to determine which attribute(s) will be needed in the future, it resorts to using some heuristic cache replacement algorithm (such as FIFO, LFU, or LRU [Mil92]) to make the decision. Whenever an attribute, which will subsequently be accessed again, is selected for replacement unnecessary I/O overhead may be incurred.

It is impractical to attempt to determine, at replacement time, which attribute should be selected for replacement. It is easier to ensure, with high probability, that there will be available cache space when new attributes must be loaded. This can be done by applying static information to determine when cached attributes are no longer needed so they may be flushed to make room for future attributes in the cache.

For a given transaction, static analysis can determine when the updates to each attribute have been completed. This information may then be used to provide "*cache hints*" to the cache manager at run-time though the use of augmented method code. Augmenting the code to deliver these cache hints is the job of the class method compiler. It must generate code to send messages (via the recovery manager) to the cache manager according to a pre-agreed interface which is supported by the objectbase runtime system.

The class method compiler must determine the *last writes* performed by a transaction to each attribute and generate code to inform the cache manager at run time when a last write occurs. The detection of last writes should be done after any unravelling is performed in support of concurrent execution. This helps to ensure that the cache hints are sent at the earliest possible time.

Loops consisting only of local steps are treated as single statements for the purpose of determining last writes. No distinction is made between writes to attributes in different iterations of non-unravelled loops. If a last write occurs in any iteration of the loop, the augmented code delivers the cache hint message to the cache manager immediately following the completion of the loop.

Definition 6.1 The *last writes* performed on an attribute a_{ij} by method m_{ik} is a set $LWs(a_{ij}, m_{ik}) = \{S_{ik_{x_1}}, S_{ik_{x_2}}, ..., S_{ik_{x_n}}\}$ where each step $S_{ik_{x_p}}$ is the last step to update a_{ij} on some control flow path through m_{ik}. ■

Conceptually, the process of determining the last writes made by a method is straightforward. A simple dataflow analysis can determine which writes to a given attribute will

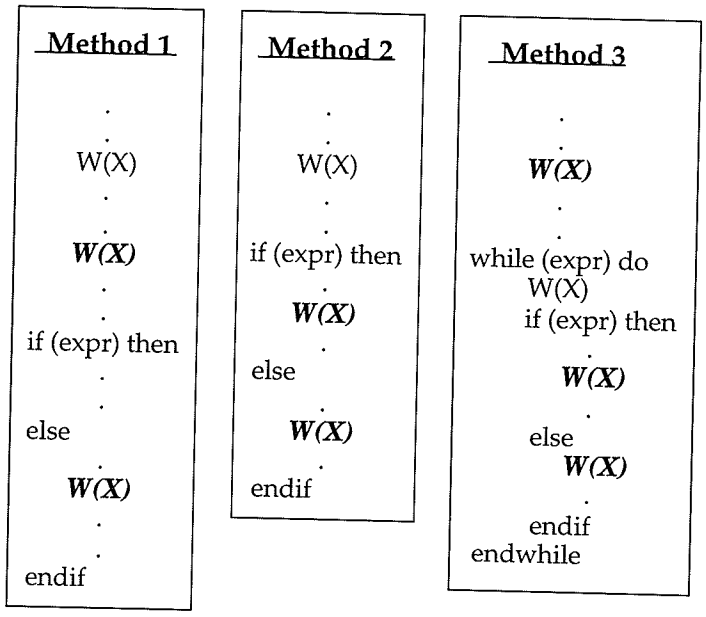

Figure 6.2: Last Writes of 'X' in Three Methods

be the final ones along the various control flow paths through the method[3]. Logically, a backwards search can be performed from each exit node to determine the *last* write(s) performed along the control flow paths leading to it. The set of last writes for a given object attribute is then the union of the last writes determined along each control flow path restricted to the attribute in question. In practice, however, this determination is hampered by the existence of *conditional* last writes which are contained in conditionally executed code. An example of the last writes of an attribute 'X' in three different methods is shown in Figure 6.2 (the last writes are shown in bold face type).

There are three fundamental "last write cases" which must be dealt with.

Last-Write-Case 1 : unconditional last write

Last-Write-Case 2 : conditional last write with no alternative

Last-Write-Case 3 : conditional last write with alternative

[3]This is similar to determining "live variables" [ASU86].

If the write of 'X' nearest an exit node of the method is not contained in any conditionally executed segment of code then it is a definite last write of 'X' (this scenario is not illustrated in the example). This corresponds to Last-Write-Case 1 (the "unconditional last write").

If the write of 'X' nearest an exit node of the method is contained in conditional code which does not provide an alternative (e.g., an ELSE clause) containing another possible last write, then it is a potential last write of 'X'[4]. Whether or not it will be the last write can only be determined at run time when it is known whether the conditional code is executed. If the conditional code is executed then the potential last write contained in it is the last write. If the code is not executed then some preceding potential last write is the last write. The alternate last write must be found by searching for it in the statements preceding the conditional. This situation is illustrated by 'Method 1' in the example and corresponds to Last-Write-Case 2.

If the two writes of 'X' nearest an exit node of the method are found, one in each half of an IF-THEN-ELSE construct, then one or the other will be the last write in any given execution of the method along any control flow path leading to that exit node. Since only one will ever be executed at runtime, both can be considered to be final writes. This is illustrated in 'Method 2' in the example and corresponds to Last-Write-Case 3.

A variety of combinations of control structures and placements of writes can lead to many different possible last write scenarios (see, for example, 'Method 3' in the example) but they all fall into one of the three previously described last write cases (e.g., 'Method 3' is Last-Write-Case 2.).

The determination of the last writes of an attribute a_{ij} made by method m_{ik} (including conditional last writes) is described in Algorithm 6.10. The details of the recursion required to follow all possible control flow paths from a given exit node in the CFG is

[4]Example constructs that might produce this situation include an IF-THEN-ELSE with a last write in only one of the THEN and ELSE parts, an IF-THEN, or a WHILE loop that may iterate zero times.

abstracted away.

Algorithm 6.10 *Determining Last Writes by Method* m_{ik}

INPUT $CFPs(m_{ik})$;

LstWrite : set of statements in m_{ik};

FOREACH (attribute a_{ij} in O_i) **DO**
  $LWs(a_{ij}, m_{ik}) \leftarrow \emptyset$; /* Initially no last writer */
  FOREACH (exit node $N \in CFPs(m_{ik})$) **DO**
    /* Scan back along CFP to find last writer of a_{ij} */
    LstWrite $\leftarrow \emptyset$;
    WHILE (nodes left to examine) **DO**
      /* 'step' is the current step being examined */
      IF (step is an unconditional $W(a_{ij})$) **THEN**
        /* found definite last write */
        LstWrite $\leftarrow$ LstWrite $\cup$ step ;
        BREAK out of while loop ;
      IF (step is a conditional $W(a_{ij})$) **THEN**
        LstWrite $\leftarrow$ LstWrite $\cup$ step;
      trace back along CFP to previous step ;
  $LWs(a_{ij}, m_{ik}) \leftarrow LWs(a_{ij}, m_{ik}) \cup LstWrite$;

End of Algorithm

Once the last writes of a method have been determined, the method code must be augmented to make use of the information by generating cache hint messages. This process is now described.

For unconditional last writes, the method code can be immediately augmented to include the sending of the cache hint message immediately after the last write. As shown, however, a last write may be conditional. In this case, it may be assumed that the potential last write is *not* a last write and a default message can be sent to the cache manager when the method completes. This may be thought of as a conservative (or pessimistic) approach. Alternatively, an optimistic approach might be applied which always assumes the potential last write is a last write and sends the cache hint message immediately. This makes sense under all but the heaviest workloads assuming a *lazy* approach to flushing cached attributes is taken. Lazy flushing will be discussed shortly.

T^k
...
$W(a_{ij})$
$Send(CacheHint(T^k, a_{ij}))$
...
exit

Figure 6.3: Augmented Code for Unconditional Last Write

Another approach to handling conditional last writes is to rewrite the method code more extensively so that the *correct* cache hint is sent as soon as it is known what it should be. Method rewriting to support this is straightforward for the three last write cases described previously.

Immediately following an unconditional last write, code to send a "message" to the cache manager informing it of the attribute which has just been written for the last time and the transaction which did the write is inserted into the method code. Following the notation used for log records, such a cache hint message has the form "$CacheHint(T^k, a_{ij})$" where T^k identifies the writing transaction and a_{ij} identifies the attribute written. An example of the augmented code for dealing with an unconditional last write is shown in Figure 6.3.

When there is a conditional last write with alternative, the cache hint message is generated in the code following *both* conditional last writes. The branch taken at runtime will determine the actual last write and, immediately after the write is executed, the cache hint will be sent. An example of the augmented code for dealing with a conditional last write with alternative is shown in Figure 6.4.

When a conditional last write with no alternative is encountered the code must be augmented so that the cache hint is sent after the conditional write if it is executed or *as soon as* it is determined that the conditional write will not be executed. In this, latter, case the hint is sent on behalf of a preceding, potential last write. Thus, the

T^k
...
if (expr) **then**
...
$W(a_{ij})$
$Send(CacheHint(T^k, a_{ij}))$
...
else
...
$W(a_{ij})$
$Send(CacheHint(T^k, a_{ij}))$
...
endif
...
exit

Figure 6.4: Augmented Code for Conditional Last Write with Alternative

method must be rewritten to include code to send the cache hint message after the last conditional write (which will be sent if the code is executed) and also code to send the cache hint message as part of a new "ELSE" part of the conditional statement enclosing the unexecuted potential last write. The new "ELSE" part will be executed only if the conditional code is not executed. An example of the augmented code for dealing with a conditional last write with no alternative is shown in Figure 6.5.

It is also possible to further rewrite the method code to move the conditional tests which determine whether or not conditional last writes will occur earlier in the code. If this can be done and it is determined that the conditional code will *not* be executed, then the default cache hint may be sent earlier. This optimization is not addressed in the dissertation.

Runtime Processing of Cache Hints

When the cache manager receives a hint from an executing method (i.e., transaction) that it has performed its last write of an attribute, it must verify that no other active

T^k
...
if (expr) **then**
...
$W(a_{ij})$
$Send(CacheHint(T^k, a_{ij}))$
...
else /* NEW CODE */
$Send(CacheHint(T^k, a_{ij}))$ /* NEW CODE */
endif
...
exit

Figure 6.5: Augmented Code for Conditional Last Write with No Alternative

transaction requires the attribute before deciding to "nominate" the attribute for cache flushing. The set of active transactions, and the conflicting object methods they invoke is maintained in the active transaction reference list (ATRL) (defined in Section 5.3.2). As described, the ATRL tracks conflicting method invocations from different transactions. If another active transaction wants to access a recently written object attribute then it will appear in the ATRL since its access operation will conflict with the last write.

If examination of the ATRL indicates that there are no active transactions requiring the updated attribute, then that attribute is a good choice for subsequent cache replacement. At this point, since the attribute is unneeded by the active transactions, it may be immediately flushed from the cache. This has the effect of decreasing the overhead during a subsequent restart but means that new transactions may be less likely to find needed data in the cache. This technique should be used only if efficiency in restart processing is deemed more important than efficiency during transaction execution.

Alternatively, a *lazy* approach where the attributes are not immediately flushed may be taken. The advantage to the lazy approach is that if a new transaction begins executing which needs the attribute it may still find it in the cache. An easy way to manage this is to have the cache manager maintain two lists of cached, dirty attributes. The first being

a list of the attributes which are still being actively accessed by some transaction(s). The second being a list of the attributes which have been written and are simply awaiting flushing. When replacement is required, space is created by flushing attributes first from the second list if it is non-empty.

If, despite the use of static information, replacement is still required, conventional schemes (such as LRU) may still be applied. In this situation, choosing the least recently used attribute for replacement is reasonable and maintaining LRU ordering information is easily and efficiently done [Mil92].

The early "flushing" of cached attributes results only in changes to the behaviour of the cache manager and does not affect the reference algorithms presented earlier.

Extensions

The idea of determining which attributes are no longer needed in the cache need not be applied solely to *dirty* attributes. In some cases, attributes will be read into the cache and not updated. If these attributes appear to be unneeded (based on analysis of the active transactions) they too should be considered prime candidates for replacement. In fact, unneeded "clean" attributes should be selected for replacement before dirty ones because no flushing is required to make the attribute's space available for re-use.

To implement this, the cache manager may maintain an additional list of clean attributes which are no longer needed. At replacement time, space should be taken from this list first, then from the list of unneeded dirty attributes, and only from the list of needed attributes as a last resort. It is possible to make the cache space occupied by unneeded clean attributes available immediately once it is known that they are unneeded but the use of a lazy approach offers the same benefit as for unneeded dirty attributes.

The availability of last writes information can also be applied to decrease the overhead of supporting recovery during transaction processing in other ways. An additional benefit of last-writes information is the ability to bypass the logging of all after-images except the

last one from each transaction. In this case logging overhead is significantly reduced. This "limited logging" of after-images can be supported by modifying the cache manager so that it only logs write operations (i.e., after-images) when it receives cache hint messages that the last write has been performed by some transaction. This approach provides a simple implementation within the described recovery architecture.

No modifications to the reference algorithms are required to support this extension except that the RM_Write algorithm only writes an after-image record to the log if the write is a last write of some attribute. The overhead during transaction execution is decreased because fewer after-image log records have to be written. During restart, redo processing will also exhibit less overhead because fewer AI (after-image) log records have to be processed since fewer were written. The modified RM_Write algorithm is detailed in Algorithm 6.11.

Algorithm 6.11 *RM_Write Algorithm with Last Writes*

INPUT T^k : Transaction issuing the write;
INPUT a_{ij} : Attribute to be written;
INPUT Value : value to be written;
INPUT LastWrite : flag indicating if this is a last write;

IF (a_{ij} is not cache resident) **THEN**
    fetch a_{ij} from the stable objectbase into the cache;
IF (LastWrite) **THEN**
    Write after-image record ($AI(T^k, a_{ij}, Value)$) to the log.
write Value into a_{ij} in the cache
RETURN

End of Algorithm

Dealing with Page-Based Caching

It has been assumed throughout this section that transfers to and from the cache were performed on individual object attributes. As stated, this is an unrealistic assumption. In a practical implementation, cache I/O operations will be performed on a per-page basis

and a given page may contain many attributes[5]. Only minor modifications are required to support *page-based* cache operations.

Having multiple attributes in a single page affects when that page should be considered for replacement[6]. If all the accesses to some attribute a_{ij} in a given page are completed, the page may still need to remain resident in the cache due to anticipated future accesses by active transactions to other attributes in the same page.

The possibility of multiple attributes per page does not affect the compile time generation of last-write hint messages to the cache manager. What changes is the behaviour of the cache manager upon receiving such a hint. Suppose transaction T^i has just executed its last write on attribute a_{ij} and sent $CacheHint(T^i, a_{ij})$ to the cache manager. Instead of checking only for other active transactions which want to access a_{ij} before deciding to flush it, the cache manager must check for active transactions which want to access *any* attribute in the same page as a_{ij}.

The chief requirement of handling page-based cache operations is the ability to determine page membership (i.e., which attributes are co-located in each page). Such determination is easily made prior to run time. Object sizes and the layout of attributes in them are known at compile time and object addresses are determined at object instantiation time. Given a fixed page size, this information can be used to easily determine which attributes reside in each page. The objectbase system can preserve this information in a form which is convenient for the cache manager to use at run time.

[5] Since only physical logging is considered in this section, the possibility of updating very large attributes (i.e., instances of classes) which exceed a single page in size is ignored.

[6] It also affects other aspects of recovery. For example, when using recovery techniques that *force* cached attributes back to the stable database to ensure durability, uncommitted updates of other attributes may also be flushed because they reside in the same page.

6.2.3 Reducing the Latency of Log Operations

Executing a given computation as a transaction normally results in increased overhead and correspondingly slower response times compared to executing it alone. A fundamental cause of the slower response time is that the recovery related operations of transaction execution (e.g., writing log entries) block on writes. This ensures that the values written are copied to either the objectbase or the log by the time control returns to the invoker. Blocking introduces delays in transaction execution which can normally be avoided. In conventional systems, this delay is hidden by the availability of other transactions to execute. Thus, although transaction execution time is increased, throughput is not reduced. In the advanced systems which will make use of objectbases, transaction execution times are more important. Thus blocking is highly undesirable.

Non-blocking writes may be used up to the point of commitment. Object data may be written *asynchronously* to the objectbase or log but there will be no guarantee that a transaction write will be reflected in stable storage before the transaction continues. This is not a problem though because *partial* results of transactions are not needed for recovery. If a transaction fails to commit, the partial work it did must be discarded anyway. Only when a transaction is about to commit is it necessary to ensure that *all* updates have been written back to the objectbase or log. Blocking once, at commit time, is preferable to blocking on every write. Using non-blocking (asynchronous) writes decreases the *apparent* overhead of recovery by decreasing the latency of transaction update operations. In reality all that has been gained is improved response time.

The order of execution of asynchronous writes is not guaranteed. Static information is needed to enable the use of asynchronous writes because dependent writes must be executed in a prescribed order. Asynchronous writes will be presented initially without considering out of order writes. The application of static analysis to ensure the order of dependent asynchronous writes will be addressed later.

To implement asynchronous writes, the recovery system must keep track of which

outstanding write operations have successfully completed. This information can be maintained in volatile memory and used at commit time to ensure that all necessary writes have completed. If a system crash results in the loss of this information, there is no problem because the information concerns only uncommitted transactions which will be undone anyway.

The concept of asynchronous I/O is well understood and descriptions of it may be found in undergraduate operating systems texts (e.g., Milenkovic [Mil92] or Leffler, *et al.* [LMKQ89]). Asynchronous output generally involves using a special "WriteAsync" system call which is passed a unique "identifier" value used to associate the output operation with the system call that generated it. When the I/O system later notifies the issuer of the write of the its completion, the identifier value is supplied. This makes it clear which of a number of possibly outstanding writes has completed. It is assumed that this notification is accomplished via an asynchronously invoked subroutine in the writer code (i.e., a call to a handler routine).

One way to use the facility just described to provide the needed asynchronous write capability is to pass the address of a queue entry to the I/O subsystem as the identifier value. The queue whose entry addresses are passed is a queue of outstanding asynchronous writes issued on behalf of a particular transaction. Before issuing an asynchronous write, the cache manager (which issues *all* writes to stable storage) adds a new entry onto the queue. The contents of the entry may be any information of interest to the cache manager concerning the write operation (e.g., the transaction identifier of the transaction on whose behalf the operation is being performed, the attribute being written, and the value to write)[7]. The address of the new queue entry is then passed to the I/O system as the identifier value.

When an I/O operation completes, the I/O system invokes the write completion handler in the cache manager and passes it a return code and the unique identifier specified

[7]The contents of the queue entry are unimportant to the process of keeping track of write completions.

when the asynchronous write was issued. The handler verifies that the operation was successful (by examining the return code) and if so, removes the corresponding entry from the queue of outstanding writes. It cannot be assumed that asynchronous writes will be processed in the order of their issue. To do so would unnecessarily constrain the behaviour (and thereby the efficiency) of the I/O system. Thus, the queue must be implemented so that deletions from the middle are feasible.

In the event that the handler determines that the write operation failed, it can attempt to re-execute the write. Assuming that the information needed to re-schedule the write (data, attribute identifier, etc.) is contained in the queue entry, the handler has all it needs to know to issue the write again. When it does so, instead of allocating a new queue entry it simply re-uses the existing one. A count of the number of retry attempts can be kept in the queue entry so that the handler can abort retrying after a specified number of failures. Should this happen, whatever steps would be taken for write failure in the absence of asynchronous writes must be performed by the handler. In addition to this normal failure processing, the queue corresponding to the, now aborted, transaction whose write failed should be destroyed.

At commit time, after all writes have been scheduled, the cache manager must wait until the queue corresponding to the committing transaction is empty before it knows that all data is safely on stable storage. Only once this has happened can the corresponding transaction be allowed to commit. This can be done by having the handler check if it is deleting the last queue entry and, if so, inform the cache manager that the queue is empty. This too, can be implemented by a "callback" mechanism. To avoid unnecessary callbacks to the cache manager (when the queue becomes empty but more writes are still to be performed for the transaction) cooperation between the cache manager and handler is required. The cache manager can write a flag in the queue data structure to indicate when it is interested in knowing that the queue is empty. It will set this flag when a transaction is prepared to commit. Only when that flag is set will the handler actually inform the cache manager when the queue empties.

Asynchronous writes may be applied to both logging and objectbase updates. The application of asynchronous writes to the logging of after-images is a straightforward modification of the RM_Write algorithm. The modified algorithm is presented in Algorithm 6.12. The corresponding handler routine which is invoked when an asynchronous write completes is detailed in Algorithm 6.13. It is possible to combine last writes and asynchronous writing even though Algorithm 6.12 does not do so.

Algorithm 6.12 *RM_Write Algorithm with Asynchronous Writes*

INPUT T^k : Transaction issuing the write;
INPUT a_{ij} : Attribute to be written;
INPUT Value : value to be written;

QNaddr : address of queue node;

IF (a_{ij} is not cache resident) **THEN**
    Fetch(a_{ij});

QNaddr ← NEW(queue node);
initialize queue node appropriately;
link queue node onto head of queue;
Log_Write("$AI(T^k, a_{ij}, Value)$") *asynchronously* passing QNaddr as the identifier value;
Cache_Write(a_{ij},Value);
IF (T^k is prepared to commit) **THEN**
    mark queue so handler will inform cache manager when it is empty;
RETURN

End of Algorithm

When doing objectbase updates, it is the cache manager which decides, via its replacement algorithm, when an attribute will be written from the cache back to the objectbase. If replacement is only performed on demand (i.e., when space is needed immediately in the cache for new data) then asynchronous writes are of no benefit. If unneeded attributes in the cache are replaced *before* the cache is full, asynchronous writing is beneficial.

Algorithm 6.13 *Asynchronous Write Completion Handler*

INPUT retcode : Return Code from the write (success or failure);
INPUT idval : identifier value containing address of queue element;

IF (retcode == success) **THEN**
    dequeue entry pointed to by idval;
    IF ((queue is marked) **AND** (queue is empty)) **THEN**
        issue callback to cache manager indicating queue empty
ELSE
    IF (retry count exceeded) **THEN**
        give up on write and call cache manager to abort transaction;
    ELSE
        increment retry count;
        asynchronously rewrite after-image record (described in queue entry) to the log passing idval as the identifier value;

End of Algorithm

The desire to avoid the delay of a cache flush when a new item must be brought into a full cache conflicts with the goal of preserving old data in the cache as long as possible in the hopes that it can be re-used. One way to allow a balance to be struck between these goals is to keep "unneeded" data in the cache for some period of time but write it out before the cache fills. This can be accomplished by defining a minimum number of empty and unneeded-but-clean pages which must be maintained. When the threshold is reached, asynchronous writes of unneeded, dirty pages begins. If all goes well, the asynchronous writes complete before the cache actually fills up. The pages to be written asynchronously can be selected according to LRU order. To achieve optimal performance using this scheme, tuning based on specific workload is required.

Out of Order Asynchronous Writes

When synchronous writes are used, it is impossible for writes issued in a particular order to be written to disk in any other order. This is not true when asynchronous writes are used. In many cases, the order of asynchronous writes is unimportant but in others, certain sequences of writes must be performed in a specific order to ensure

correctness. For example, the last write to a given attribute performed by a particular transaction must be the one reflected in the objectbase. The use of asynchronous writes must not cause two writes from one transaction to be reordered if one of them is the transaction's last write (i.e., last write semantics must be preserved). It is the cache manager's responsibility to ensure this.

The problem of ensuring that the last write of an attribute is the one reflected in the stable objectbase can be addressed by partially ordering outstanding synchronous writes on a per-attribute basis according to their last write dependencies. The last write dependencies are determined by restricting $DG(m_{ij})$ to the output operations on a given attribute[8]. When a new asynchronous write is being scheduled, the RM_Write routine can check if any other dependent writes of the attribute have not yet completed. If this is the case, the new operation can be queued for later execution when the preceding write(s) have completed. This still allows independent writes to the same or different attributes to proceed concurrently. Furthermore, where synchronous writes are required, the synchronicity is ensured by the objectbase *without* forcing transactions to block.

There is no problem ensuring appropriate write orders when asynchronous writes are combined with last writes because only one write per attribute is ever performed. This seems to suggest that asynchronous writes are of limited usefulness when combined with last writes optimizations. This is not the case. Asynchronous writes combined with last writes is useful because it allows the concurrent processing of last writes for *different* attributes.

6.2.4 Reducing the Amount of Log Information Written

The overhead of physical logging can be reduced by using logical logging. In conventional databases, this means logging operations performed on larger units than individual data items (e.g., tuples or records) instead of logging each write to an attribute in the "larger

[8]This is an application of static information.

unit". In objectbases, this means logging method invocations. Since a single method invocation may encompass many operations on individual attributes, fewer disk writes are performed using logical logging and the I/O overhead is correspondingly decreased.

In objectbases, considering object method invocations to be logical operations is straightforward and leads to a natural decomposition of the writes into groups of related operations. Furthermore, since partial results from a method are not of interest, logging only after successful method execution means that no overhead is incurred to save the results of non-recoverable operations. By logging object operations (method invocations) rather than the results of executing them, significant overhead is saved during transaction execution. The price paid is the need to re-execute any committed operations whose results were not flushed to disk prior to a system failure.

As discussed earlier, logical logging in objectbases requires the logging of method invocations, their input parameters, and the state of the object being operated on prior to the invocation. Logging the method invocations and their input parameters can be accomplished using extended MI log records. The format of an extended method invocation log record is "$MI(T^i, T^j, Parms)$" where T^i is the transaction making the method invocation, T^j is the sub-transaction which results, and $Parms$ is the set of input parameters provided to T^j by T^i. These extended MI log records can be written by the cache manager during sub-transaction creation. The extended RM_SubTrans algorithm is shown in Algorithm 6.14.

Algorithm 6.14 *Extended RM_SubTrans Algorithm*

INPUT T^i : Transaction making the method invocation;
INPUT T^j : Transaction created as a result of the invocation;
INPUT $Parms$: Input parameters provided to T^j;

Log_Write("$MI(T^i, T^j, Parms)$");
RETURN

End of Algorithm

The before-images of the attributes to be updated by a logical operation must be

logged prior to the execution of each logical operation[9]. In objectbases, this can be problematic due to the size and complexity of objects. It may be impractical to log the entire object state especially for the execution of short methods. This can be addressed by using static information to decrease the amount of information which must be written in each before-image log record.

The amount of information in a before-image log record may be reduced by limiting what object attributes are included in it. Object attributes which cannot be updated by the transaction to be executed need not be preserved for undo purposes. Thus, only those attributes which may be updated need to be in the before-image log record and the set of such attributes can be determined using static analysis. Only those object attributes specified in the writeset of the method (i.e., $WS(M_{ij})$) need be written in the before-image. In many cases, this set of attributes will be significantly smaller than the total set of object attributes. Writing fewer attributes reduces the I/O overhead of logging before-images.

The logging of before-images is the responsibility of the RM_Start routine. The extended version of this routine is given in Algorithm 6.15.

Algorithm 6.15 *Extended RM_Start Algorithm*

INPUT T^k : Transaction being started (it executes m_{ij});

Value : an attribute value;

Log_Write("$BT(T^k)$");
FOREACH (attribute $a_{ij} \in WS(M_{ij})$) **DO**
    Fetch(a_{ij});
    Value $\leftarrow$ Cache_Read(a_{ij});
    Log_Write("$BI(T^k, a_{ij}, Value)$");

End of Algorithm

[9] Attribute values for the before-images must be taken from the most recently committed transactions even if those values are still stored in the cache.

An even smaller set of attributes which must be written as part of a before-image can be determined if the input parameter values to the method are considered in determining which attributes may be updated. Different write sets for each method may be maintained which are selected at run time based on input parameter value(s) to the method. Discussion of the derivation and application of these sets is not considered.

After a system crash, the restart procedure must redo the committed transactions that appear in the log (i.e., those which *may* not have been successfully flushed to the objectbase because they occurred after the last checkpoint). To accomplish this, the operations of committed transactions recorded in the log must be re-executed. If no attempt is made to preserve the pre-committed results of the sub-transactions of an uncommitted user transaction then redo may be performed on a user transaction basis. That is, only committed *user* transactions found in the log need be re-executed. Assuming that the transactions are re-executed in the same serialization order in which they originally executed. they will operate on the same objectbase data, invoke the same sub-transactions, and produce the same results as they did originally. The modified restart processing required to support logical logging is detailed in Algorithm 6.16. No new transactions are processed until restart has completed.

The processing of undo operations is unchanged in the algorithm despite the fact that the before-images used in the undo store only partial object states. As each undo is performed, the before-images of all attributes which may have been overwritten by uncommitted data are restored. By the time all necessary transactions are undone in the reverse order from which they were initiated, *all* possibly incorrect attributes have had their proper before-images restored in the cache. Eventually, these will be flushed to the stable objectbase.

Algorithm 6.16 *Logical Restart*

RedoList : list of transactions to be redone;

/* determine which transactions must be undone and redone */
FOREACH (log record from last to first) **DO**
    IF (log record is $ET(T^i, Abort)$) **THEN**
        undo(T^k); /* done here to ensure correct order */
    IF (log record is $ET(T^i, Commit)$) **THEN**
        IF (T^i is a user transaction) **THEN**
            add T^i to RedoList;
    IF (log record is $BT(T^i)$) **THEN**
        IF (no corresponding ET has been seen) **THEN**
            /* transaction was active so undo it */
            undo(T^k);
/* The undos have restored the before-images from the last checkpoint */
/* Now perform redos by re-executing the committed logical operations */
FOREACH (transaction T^k on RedoList in order of their initiation) **DO**
    re-execute T^k; /* this is a user transaction */

End of Algorithm

6.2.5 Recovery in Multi-Version Objectbases

Ideally, no transactions need be aborted when using multi-version objectbases with reconciliation for concurrency control (Section 5.5.1). Conflicting method executions (i.e., transactions) are simply reconciled. In practice, however, given the algorithm for multi-version object concurrency control described in Section 5.5 and knowing that in some cases, transactions *choose* to abort themselves, recovery processing must take aborts into account.

A fundamental advantage of multi-version objectbases is that uncommitted results need never be present in the objectbase. Active transactions operate on their own versions of objects which occupy physically distinct space from other versions in the objectbase. If a system failure occurs prior to transaction commitment, undo is accomplished by simply reclaiming any space allocated to the active version. This also applies to dealing with aborts. Thus, although aborts are possible, there is only minimal undo processing associated with them. Another consequence of multi-version objects is that before-image

records never have to be logged. This results in decreased recovery related I/O overhead during transaction processing.

The lack of undo processing is reminiscent of shadow-based recovery techniques and recovery processing for multi-version objects is very similar to shadow paging. The updated "pages" are the updated active object versions and these must be written back to the object base atomically when a transaction commits or simply discarded when it aborts.

A two-phase commit strategy may be used to ensure that all the new object versions associated with a committing transaction are successfully written to the objectbase. At commit time, the list of object versions created by the transaction is written atomically to stable storage in an area reserved for this purpose. This is the first phase of commitment. Then, the new object versions are made the LCVs for their corresponding objects one by one. This involves updating some sort of directory structure to install each new version as the "top" (last committed) element on what is logically a stack of object versions. As each version is successfully installed, it is removed from the stable store list of versions to be installed. When the list is empty, it too is removed from stable storage. Installing new object versions is the second phase of commitment.

Should a failure occur during the first phase of the commitment process, the objectbase remains consistent because all the effects of the, as yet, uncommitted transaction will be lost. If it occurs during the second phase, sufficient information is on stable storage to permit the commitment to continue upon restart. Note also that restart is idempotent since the operation of installing an object version can be easily made idempotent.

Commitment, as just presented, seems to preclude the possibility of avoiding the cost of flushing updated versions at commit time. This is not necessarily the case. An object version need not be flushed to stable store at commitment time. If the updates made in creating a new version are logged as after-image records, then the log will contain sufficient information to reconstruct an object version that was committed but not entirely flushed

to the stable objectbase. The logging and processing of after-images is the same as in non-versioned objectbases.

Keeping the last committed version of an object in cache is highly desirable if another transaction will access the object. Requiring another active version to be derived for the new transaction. This involves copying the LCV in its entirety and such copying is done orders of magnitude faster in the cache than on disk.

Chapter 7

Conclusions and Future Work

This dissertation has presented new algorithms for concurrency control and recovery in objectbases. The algorithms described all exploit some form of statically derived information to achieve their results and each algorithm improves on existing algorithms in some way. Thus, the thesis of this dissertation, that "static information can be used to enhance concurrency control and recovery", has been demonstrated.

7.1 Contributions

This dissertation has contributed in numerous ways beginning with the idea of applying dependency analysis to concurrency control and recovery. Previous applications of static analysis were extremely limited and were either applied in a limited fashion or required manual intervention. Greater benefits are achieved when all aspects of static analysis are considered and applied automatically as done in the dissertation.

A fundamental contribution to the application of static analysis to providing concurrency control in objectbases is the decomposition of the problem into intra-transaction and inter-transaction concurrency. Doing this simplifies the process of reasoning about concurrency *vis a vis* static analysis and thereby simplifies the development of related

concurrency control algorithms. The algorithms for intra- and inter-transaction concurrency control developed in the dissertation are combined to form a single algorithm for "full objectbase concurrency control"[1].

Existing concurrency control algorithms determine serialization orders either dynamically (e.g., two-phase locking) or statically (e.g., timestamp ordering). The algorithm presented combines static and dynamic information by determining serialization orders dynamically using static information. For sub-transactions of a user transaction, the serialization order of conflicting sub-transactions is determined *à priori* based on static information. Serialization orders between user transactions are determined dynamically but are explicitly selected (using static information) rather than using arbitrary orderings produced as a side-effect of the concurrency control algorithm used (as happens with two-phase locking).

In the algorithm, overhead is incurred for conflicting operations only. This is in clear contrast to existing concurrency control protocols which incur some overhead for all operations. Conflicts are determined and serialization orders are chosen by the global scheduler and the root object schedulers. Messages are then sent between these schedulers and those sub-transaction object schedulers where conflicts may occur. This exchange of messages occurs only for conflicting method executions but still guarantees serializable executions at each object.

In addition to limiting the situations in which concurrency control overhead is incurred, this approach offers two other advantages. First, it is deadlock free. Freedom from deadlocks has historically been a characteristic of only order-preserving concurrency control algorithms. The algorithm presented is not order preserving[2] but is still deadlock free. Second, concurrency between sub-transactions whose children conflict is permitted (see Figure 5.2). Other objectbase concurrency control protocols typically require

[1] It is this combined algorithm that is now discussed.

[2] at least in the accepted sense of the phrase

such sub-transactions to execute serially [RGN90, HH91]. Thus, the presented algorithm may permit significantly more concurrency depending on the aggregation structure of the objects being operated on.

Many objectbase concurrency control algorithms are based on object-level locking (or variants thereof). This approach provides far coarser grained concurrency than that provided by the dissertation algorithm. By determining conflicts on a per-method basis, non-conflicting method invocations on a single object may execute concurrently. This increase potential concurrency depending on the attribute access patterns of the methods within an object.

A final contribution to objectbase concurrency control is the idea of optimizing conflict detection by considering control flow paths. Automatically determining which of a number of possible control flow paths will be followed in a method execution permits the sets of attributes accessed to be more accurately estimated and thereby decreases the likeliehood of detecting false conflicts. Fewer conflicts means that more method executions may take place concurrently and thus, both transaction throughput and transaction response time may improve.

Contributions to enhancing recovery through the use of static information are also presented in the dissertation. Foremost among these is the idea of determining which writes to attributes are the *last* writes performed by a given method. Having this information allows the number of disk writes to be decreased thereby decreasing the overhead of ensuring recoverability during transaction processing. Last writes information can be applied to both limiting cache flushing and to decreasing logging.

By knowing when the last write for a method execution has occurred, the cache manager can make an intelligent choice when replacement is required. It can choose to replace pages which it *knows* are no longer needed. This is a significant improvement over selecting a page using a conventional replacement algorithm which has no insight into the potential future behaviour of the active transactions.

When logging after images of update operations for redo purposes, all writes to an object attribute by a method are normally logged. This is unnecessary since only the last write must be durable (the results of non last writes are always discarded). The writing of such useless after images introduces unnecessary overhead which will be very high if methods make many updates to each attribute. By logging only the last write to each attribute, this overhead is avoided.

A recovery protocol which exploits logical logging is also presented. Objectbases offer a natural environment for the application of logical logging where each method invocation is logged as a single logical operation. Such logging drastically decreases the number of log records which must be written. Treating method invocations as logical operations implies that the data being operated on are the objects themselves and this can introduce problems.

Before-images are logged to permit undo and thus, it must be possible to reconstruct the previous state of an object from its logged before-image. The amount of information (i.e. number of attributes) in an object may be quite large and logging it may thus be expensive. This is particulaly true if a given method invocation only updates a small subset of an object's attributes. To address this problem, static information can be applied to determine the set of attributes which may be updated by a given method execution and only this subset need be logged in a before-image. This makes logical logging in objectbases practical.

7.2 Future Work

The work presented in this dissertation is a first application of static analysis to concurency control and recovery. Much work remains to be done in this area including the elimination of certain explicit and implicit simplifying assumptions in the work presented, extensions to the work, and the application of static analysis in entirely new ways.

Additionally, there is a need to develop a testbed environment to permit the implementation of the algorithms to empirically determine their effectiveness for various kinds of applications and to compare them against existing algorithms.

A number of simplifying assumptions were made throughout the dissertation, many of which may be removed. This is the first form of future work.

Concurrency between a method and other methods in the same object and between a method and the methods of physically contained sub-objects was precluded in the dissertation. There is nothing special about these methods and there is no reason, beyond simplified presentation, to preclude such concurrency. The techniques presented in the dissertation may be easily extended to accomodate concurrency within an object and between an object and its sub-objects.

Another simplifying assumption made was that the run time checking of control flow conditions involving attributes calculated by code spanning multiple basic blocks in a method was impractical. In many, but not all, cases this is true. In large methods, it may be desirable to consider the runtime checking of such complex control flow conditions. Larger methods may access correspondingly larger sets of attributes and accuracy in determining the exact sets accessed is important. The difficulty in evaluating such conditions is two-fold. First, the expressions themselves may be difficult to compute because they require a significant percentage of a method's computation to be performed to calculate the needed values. Nothing can be done about this and such conditions should be marked non-checkable. Second, due to complex control flow, it may appear difficult to determine what the value of an attribute involved in the condition is. Existing dataflow techniques may be applied to solve this problem which is simply the "common sub-expressions" problem [ASU86].

Certain simplifying assumptions were also made in the discussion of multi-version objectbase concurrency control. Primary among these was that the object versions to be reconciled came from the same base version. This is an undesirable assumption to make

in a practical implementation because it precludes new transactions starting if they make conflicting accesses to an object for which there is an outstanding active transaction whose base version is not the current LCV. In systems where access to objects is not close to uniformly distributed, transactions accessing "hot spot" objects will be delayed as in pessimistic protocols[3]. Support for different base versions is an open area for future research.

Many extensions to the algorithms presented in the previous chapters were also suggested. These provide obvious areas for future work. One ripe area for immediate future work is in developing heuristics for choosing "good" serialization orders in Algorithm 5.6. The algorithm, as presented, applies a greedy approach to determining a serialization order. A simple heuristic where, given a choice, newer transactions are serilized after older ones was described. Other heuristics which may consider other, possibly longer-term, goals are also possible.

Static analysis was always performed on a per-method basis (with the exception of the discussion of control flow paths). Determining conflict on a per-method basis increases the potential concurrency over object-level locking but is inferior to attribute-level locking. Application at a finer granularity would permit greater potential concurrency but it is not immediately obvious how to do this in general. One possible extension is to optimize conflict detection by examining method input parameter values. If, based on input parameter values available at run time, it can be determined that certain attributes will not be accessed, then the attribute reference sets used in conflict determination can be "pruned" and the chance of conflict with other methods will be correspondingly decreased. This is similar to determining control flow paths at runtime and is useful for both concurrency control, as described, and recovery[4].

When considering multi-version concurrency control, support for nested objects is

[3] Such delays do not result in delays any greater than what would be incurred using object-level locking.

[4] Determining restricted write sets can be used to limit the amount of information which must be recorded in before-images.

desirable for recovery purposes. Thus, future work should include the investigation of reconciliation for nested multi-version objects. As described in the dissertation, this must focus on determining the upward and downward effects of failed sub transactions. A starting point will be the definition of a taxonomy of effects and how they are propogated. A solution to the general problem appears to be difficult due to the possibility of sub-transactions sharing objects and thus, the possibly far-reaching effects of a failed sub-transaction.

Many additional applications of static analysis to concurrency control and recovery which could not be discussed in the dissertation are also feasible. Possibilities include the automatic determination of commutativity through the use of static symbolic analysis, the use of different object-local concurrency control protocols (or at least different object-local definitions of conflict) on a per-class basis, and the application of dependency analysis techniques to user-defined container classes. There are undoubtedly many other possible applications as well.

Bibliography

[AAE93] G. Alonso, D. Agrawal, and A. El Abbadi. A Unified Implementation of Concurrency Control and Recovery. Technical report, University of California at Santa Barbara, 1993.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40 – 57, 1989.

[ADG93] R. Agrawal, S. Dar, and N.H. Gehani. The O++ Database Programming Language: Implementation and Experience. In *Proceedings of the 9th IEEE Conference on Data Engineering*, 1993.

[AE92] D. Agrawal and A. El Abbadi. A Non-Restrictive Concurrency Control for Object Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the 3rd International Conference on Extending Database Technology*, pages 316 – 319. Springer-Verlag, March 1992.

[AG89] R. Agrawal and N.H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings of the ACM, SIGMOD Conference on the Management of Data*, 1989.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic, 1988.

[BEMP92] K. Barker, M. Evans, R. McFadyen, and K. Periyasamy. A Formal Ontological Object-Oriented Model. Technical Report TR 92-2, University of Manitoba, Dept. of Computer Science, 1992.

[BG81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185 – 221, 1981.

[BG83] P.A. Bernstein and N. Goodman. Multiversion Concurrency Control – Theory and Algorithms. *ACM Transactions on Database Systems*, 8(4):465 – 483, 1983.

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

[BK91] N.S. Barghouti and G.E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[BM91] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4):33 – 47, 1991.

[BS91] D.F. Bacon and R.E. Strom. Optimistic Parallelization of Communicating Sequential Processes. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 155 – 166, 1991.

[Bur88] A. Burns. *Programming in Occam 2.* Addison-Wesley, 1988.

[CBW92] C. Chachaty, P. Borla-Salamet, and M. Ward. An Approach to the Design of a Parallel Query Language. In *Proceedings of Parallel Architectures Europe*, 1992.

[CCHK90] C.D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, 16(4):483 – 487, April 1990.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenburg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases.* Morgan-Kaufmann, 1990.

[CDKK85] H.T. Chou, D.J. Dewitt, R.H. Katz, and A.C. Klug. Design and Implementation of the WisconsinStorage System. *Software – Practice and Experience*, 15(10), 1985.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, 1991.

[Dat86] C.J. Date. *An Introduction to Database Systems, 4ed.* Addison Wesley, 1986.

[Deu90] O. Deux *et al.* The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91 – 108, 1990.

[DK90] P. Dasgupta and Z.M. Kedem. The Five Color Concurrency Control Protocol: Non-Two-Phase Locking in General Databases. *ACM Transactions on Database Systems*, 15(2):281 – 307, June 1990.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624 – 632, November 1976.

[Elm92] A.K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[EN89] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1989.

[FO89] A.A. Farrag and M.T. Özsu. Using Semantic Knowledge to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503 – 525, December 1989.

[GB94] P.C.J. Graham and K. Barker. Effective Optimistic Concurrency Control in Multiversion Object Bases. In *Proceedings of the International Symposium on Object Oriented Methodologies and Systems*, Palermo, Italy, September 1994. *Accepted.*

[GK88] J.F. Garza and W. Kim. Transaction Management in an Object-Oriented Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37 – 45. ACM, 1988.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[GM88] N. Gehani and A.D. McGettrick, editors. *Concurrent Programming.* Addison-Wesley, 1988.

[Gre91] S. Greenberg, editor. *Computer-supported Cooperative Work and Groupware.* Academic Press, 1991.

[GSW92] I. Greif, R. Seliger, and W. Weihl. A Case Study of CES: A Distributed Collaborative Editting System Implemented in Argus. *IEEE Transactions on Software Engineering*, 18(9):827 – 839, 1992.

[GZB92] P.C.J. Graham, M.E. Zapp, and K. Barker. Applying Method Data Dependence to Transactions in Object Bases. Technical Report TR 92-7, University of Manitoba, Dept. of Computer Science, 1992.

[Had88] V. Hadzilacos. A Theory of Reliability in Database Systems. *Journal of the Association for Computing Machinery*, pages 121–145, January 1988.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, and E. Shekita. Starburst Mid-Flight:

As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–159, 1990.

[HDV88] B. Hart, S. Danforth, and P. Valduriez. Parallelizing FAD: A Database Programming Language. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, 1988.

[Her90] M. Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems*, 15(1):96 – 124, March 1990.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2 – 24, 1991.

[HHZ+92] S. Heiler, S. Haradhvala, S. Zdonik, B. Blaustein, and Arnon Rosenthal. A Flexible Framework for Transaction Management in Engineering Environments. In *Database Transaction Models for Advanced Applications*, chapter 4. Morgan Kaufmann, 1992.

[HK89] S.E. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems*, 14(3):291 – 321, 1989.

[HP90] J.L. Hennessey and D.A. Patterson, editors. *Computer Architecture A Quantitative Approach.* Morgann Kaufmann, 1990.

[HP93] H. Hakimzadeh and W. Perrizo. Fine Granularity Locking for Object-Oriented Databases. In *Proceedings of the First Annual Mid-Continent Information Systems Conference*, pages 41 – 44, May 1993.

[HPC93] A.R. Hurson, S.H. Pakzad, and J.-B. Cheng. Object-Oriented Database Management Systems: Evolution and Performance Issues. *IEEE Computer*, 26(2):48 – 60, 1993.

[HR83] T. Häerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, pages 287–317, December 1983.

[HR87] T. Häerder and K. Rothermel. Concepts for Transaction Recovery in Nested Transactions. In *Proceedings of the ACM Conference on Management of Data*, pages 239–248, May 1987.

[HR93] T. Härder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39 – 74, January 1993.

[JK92] A. Jhingran and P. Khedkar. Analysis of Recovery in a Database System Using a Write-Ahead Logging Protocol. In *Proceedings of the ACM-SIGMOD*

1992 Conference on Management of Data, pages 175–184, San Diego, California, 1992.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109 – 124, 1990.

[Kim90] W. Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327 – 341, 1990.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th Very Large DataBase Conference*, pages 95–106, Brisbane, Australia, 1990.

[KP92] G.E. Kaiser and C. Pu. Dynamic Rstructuring of Transactions. In *Database Transaction Models for Advanced Applications*, chapter 8. Morgan Kaufmann, 1992.

[KR81] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213 – 226, 1981.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *Lecture Notes in Computer Science 114: CLU Reference Manual.* Springer-Verlag, 1981.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[LMKQ89] S.J. Leffler, M.K.. McKusick, M.J.. Karels, and J.S. Quarterman. *The Design and Iplementation of the 4.3 BSD UNIX Operating System.* Addison Wesley, 1989.

[Lyn83] N.A. Lynch. Multilevel Atomicity – A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484 – 502, 1983.

[MGG86] J. E. B. Moss, N.D. Griffith, and M.H. Graham. Abstraction in Recovery Management. In *Proceedings of the ACM-SIGMOD 1986 Conference on Management of Data*, pages 72–83, 1986.

[MHL+89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. A Transaction Recovery Method Supporting Fine Granularity Locking and Partial Rollbacks using Write-Ahead Logging. Technical Report IBM-RJ6649, IBM Almaden Research Center, January 1989.

[Mil92] M. Milenkovic. *Operating Systems Concepts and Design.* McGraw Hill, second edition, 1992.

[MM93] C. Malta and J. Martinez. Automating Fine Concurrency Control in Object-Oriented Databases. In *Proceedings of the International Conference on Data Engineering*, pages 253 – 260, 1993.

[Moh90] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proceedings of the 16th Very Large DataBase Conference*, pages 1 – 14, 1990.

[Mor93] T. Morzy. The Correctness of Concurrency Control for Multiversion Database Systems with Limited Number of Versions. *IEEE*, pages 595 – 604, 1993.

[Mos85] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[Mos87] J. E. B. Moss. Log-Based Recovery for Nested Transactions. In *Proceedings of the 13th Very Large DataBase Conference*, pages 427–432, Brighton, 1987.

[MRKN92] P. Muth, T.C. Rakow, W. Klas, and E.J. Neuhold. A Transaction Model for an Open Publication Environment. In *Database Transaction Models for Advanced Applications*, chapter 6. Morgan Kaufmann, 1992.

[MRW+93] P. Muth, T.C. Rakow, G. Weikum, P. Brossler, and C. Hasse. Semantic Concurrency Control in Object-Oriented Database Systems. In *Proceedings of the International Conference on Data Engineering*, 1993.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Proceedings of the International Workshop on Object-Oriented Databases*, pages 171 – 182, September 1986.

[MT86] S.J. Mullender and A.S. Tanenbaum. A Distributed File Service Based on Optimistic Concurrency Control. Technical Report CS-R8507, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[Nak92] T. Nakajima. Commutativity Based Concurrency Control and Recovery for Multiversion Objects. In *Preproceedings of the International Workshop on Distributed Object Management*, pages 101 – 119, 1992.

[NRZ92] M.H. Nodine, S. Ramaswamy, and S.B. Zdonik. A Cooperative Transaction Model for Design Databases. In *Database Transaction Models for Advanced Applications*, chapter 3. Morgan Kaufmann, 1992.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[Per87] R.H. Perrott. *Parallel Programming.* Addison-Wesley, 1987.

[Per91] W. Perrizo. Request Order Linked List (ROLL): A Concurrency Control Object for Centralized and Distributed Database Systems. In *Proceedings of the International Conference on Data Engineering*, pages 278 – 285, 1991.

[Pol88] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic, 1988.

[PW86] D.A. Padua and M.J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communication of the ACM*, 29(12):1184 – 1201, December 1986.

[RC92] K. Ramamritham and P.K. Chrysanthis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In *Preproceedings of the International Workshop on Distributed Object Management*, pages 120 – 140, 1992.

[RCS92] J.E. Richardson, M.J. Carey, and D.T. Schuh. The Design of the E Programming Language. Technical Report TR824, Computer Science Dept. U. Wisconsin, Madison, 1992.

[RE92] R.F. Resende and A. El Abbadi. A Graph Testing Concurrency Control Protocol for Object Bases. In *Proceedings of the 4th International Conference on Computing and Information*, pages 316 – 319, 1992.

[Ree78] R. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Dept. of Computer Science, Massachusetts Institute of Technology, 1978.

[Ree79] D.P. Reed. Implementing Atomic Actions. In *Proceedings of the Seventh Symposium on Operating System Principles*, December 1979.

[RGN90] T.C. Rakow, J. Gu, and E.J. Neuhold. Serializability in Object-Oriented Database Systems. In *Proceedings of the International Conference on Data Engineering*, pages 112 – 120, 1990.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proceedings of the 15th Very Large DataBase Conference*, pages 337–346, Amsterdam, 1989.

[RSC91] C.V. Ramamoorty, P.C. Sheu, and P. Chillakanti. Object-Oriented Approach: Systems, Programming, Languages, and Applications. In E. Nahouraii and F. Petry, editors, *Object-Oriented Databases*, pages 21 – 38. IEEE Computer Society Press, 1991.

[Ryd79] B. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 5(3):216 – 225, 1979.

[SG94] A. Silberschatz and P.B. Galvin. *Operating System Concepts.* Addison-Wesley, 1994.

[Sha81] M. Shaw, editor. *ALPHARD Form and Content.* Springer-Verlag, 1981.

[Som89] I. Sommerville. *Software Engineering.* Addison-Wesley, 1989.

[SR81] R.E. Stearns and D.J. Rosenkrantz. Distributed Database Concurrency Control Using Before-Values. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pages 74 – 83, 1981.

[SWY93] H.J. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. In *Proceedings of he ACM Symposium on Principle of Database Systems*, pages 300 – 311, June 1993.

[Tho79] R.H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180 – 209, June 1979.

[U.S80] U.S. Department of Defense. *Lecture Notes in Computer Science 106: The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: an Overview. In *Proceedings of the 15th Conference on Very Large Database Systems*, pages 357 – 366, 1989.

[WB87] M. Wolfe and U. Banerjee. Data Dependence and Its Application to Parallel Processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[Wei88] W.E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488 – 1505, 1988.

[Wei89a] W. E. Weihl. The Impact of Recovery on Concurrency Control. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 259–269, 1989.

[Wei89b] W.E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions of Programming Languages and Systems*, 11(2):249 – 282, 1989.

[Wei89c] W.E. Weihl. Theory of Nested Transactions. In S. Mullender, editor, *Distributed Systems*, chapter 12, pages 237 – 262. ACM Press, 1989.

[Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132 – 180, March 1991.

[Wie83] G. Wiederhold. *Database Design.* McGraw Hill, 1983.

[Wie86] D. Wiebe. A Distributed Repository for Immutable Persistent Objects. In *Proceedings of OOPSLA '86*, pages 453 – 465, 1986.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implemntation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63 – 75, 1990.

[Wol89] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.

[WS92] G. Weikum and H.–J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann, 1992.

[WYC93] K-L. Wu, P.S. Yu, and M-S. Chen. Dynamic Finite Versioning: An Effective Versioning Approach to Concurrent Transaction and Query Processing. *IEEE*, pages 577 – 586, 1993.

[ZB93a] M.E. Zapp and K. Barker. Modular Concurrency Control Algorithms for Object Bases. In *International Symposium on Applied Computing: Research and Applications in Software Engineering, Databases, and Distributed Systems*, pages 28–36, Monterrey, Mexico, October 1993.

[ZB93b] M.E. Zapp and K. Barker. On Concurrency Control in Object Bases. In *Proceedings of the First Annual Mid-Continent Information Systems Conference (MISC'93)*, pages 91–97, Fargo, USA, May 1993.

[ZB93c] M.E. Zapp and K. Barker. The Serializability of Transaction in Object Bases. In *Proceedings of the International Conference on Computers and Information*, pages 428–432, Sudbury, Canada, May 1993.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison Wesley, 1990.