Automation of Fine Concurrency in Object-Oriented Databases

Woochun Jun Le Gruenwald

School of Computer science
University of Oklahoma
Norman, OK 73019

Abstract

In this paper, we present a scheme to increase concurrency among methods in Object-Oriented Databases. We are concerned with instance access methods. Our work has the following characteristics. First, construction of commutativity relation among methods can be automated. Second, it provides more concurrency than read and write access modes on methods. Third, deadlocks due to lock escalation can be reduced. Finally, concurrency is increased further with the use of run-time information.
Key word: Concurrency, object-oriented database

1. Introduction

In object-oriented database systems (OODBS), a database is a collection of classes and instances called *objects*. Users can access objects by invoking methods in OODBS. To make sure atomicity of user interactions, the traditional transaction model can be used in OODBS. That is, users can access OODB by executing transactions, each of which is defined as a partially ordered set of method invocations on class or instance objects [1].

Concurrency control involves synchronization of multiple access to the database, so that the consistency of the database is maintained [2]. In OODBS, one of main concerns is to increase concurrency among methods so that more transactions can run in parallel. Usually, transactions are long in typical OODB applications. Thus, aborting a long transaction due to conflicts wastes system resources. Also, holding resources by a long transaction may delay other transactions. Commutativity is a widely used criterion to determine whether a method can run concurrently with those in progress on the same object. Two methods commute if their execution orders do not affect their results. Two methods conflict each other if they do not commute.

In general, there are two types of access to an OODB : instance access and class definition access [3]. An instance access consists of consultations and modifications of attribute values in an instance or a set of instances. A class definition access consists of consulting class definition, adding/deleting an attribute or a method, changing the implementation code of a method or changing the inheritance relationships between classes, etc.

In this paper, we present a scheme to increase concurrency among instance access methods. Our scheme has several important characteristics. First, it does not put the burden of determining commutativity for methods on application programmers. Second, it provides more concurrency than read and write access modes on methods. Third, we reduce deadlocks due to lock escalation, which is a main source of deadlocks [4]. Finally, it takes run-time information into consideration to improve concurrency.

The paper is organized as follows. In the next section, we review related studies and discuss their advantages and disadvantages. In section 3, we propose a scheme to increase concurrency among instance access methods. Finally, we give a conclusion in section 4.

2. Previous Work

Several techniques have been proposed to increase concurrency among instance access methods [1,5,6,7]. These techniques put a serious burden on application programmers by requiring them to specify method commutativity.

Recently in [4], the process of constructing commutativity relation from method contents is automated. It is based on the notion of *affected sets of attributes* [5]. That is, even if two methods conflict in terms of read or write operations, as long

as their access modes on individual attributes do not conflict, two methods can run in parallel. Commutativity of methods is determined at compile-time so that run-time commutativity checking is avoided. An Direct Access Vector (DAV) is constructed for each method, which is a vector whose field corresponds to each attribute defined in the class on which the method operates. Each value composing this vector denotes the most restrictive access mode used by the method when accessing the corresponding field. An access mode of an attribute can have one of three values, N (Null), R (Read) and W (Write) with N < R < W for their restrictiveness. Two methods commute if their corresponding DAVs commute, that is, their access modes are compatible for each attribute.

The above technique takes access mode information solely from the source code of a method and thus frees the user from determining commutativity relations. It also provides finer concurrency than mere read and write conflicts by examining attribute level. However, concurrency improvement offered is limited since run-time information on attributes is not taken into account.

3. Our Approach

3.1. Description

Our work exploits run-time information to further improve the scheme developed in [4]. Similar to [4], we need a two-phase pre-analysis which consists of two steps : 1) construction of DAV for each method and 2) construction of a commutativity table of methods. In each method, a break point is inserted by a programmer or a compiler when a conditional statement is encountered. Every method has a special break point called *first break point* before the first statement in the method. There are three kinds of DAVs in each method : 1) a DAV of the entire method as in [4] 2) a DAV of the first break point, which is a union of access modes of each attribute used by statements between the first break point and the next break point and access modes of each attribute used by statements that are executed regardless of execution paths. A union operation is equivalent to *max*, e.g., N + W = W and 3) a DAV of every other break point, which contains access modes of all attributes used by statements between this break point and the next break point (or end of the method).

For example, assume that we have three methods M1, M2 and M3 and a class Y with four attributes a_1, a_2, a_3 and a_4. A, A1, A2, and A3 are breakpoints of M1, B is a breakpoint of M2, and C, C1, and C2 are breakpoints of M3. A union operation denoted as '⊕' takes two arguments among N (null: no operation), R (Read), and W (Write) and selects the more restrictive one. Table 1 illustrates how the union operation works.

	N	R	W
N	N	R	W
R	R	R	W
W	W	W	W

Table 1. Union operation table

The contents and DAVs of each break point in the method are given below. Let DAV(x) represent DAV of a breakpoint x or a method x.

method M1
[A]
read a_1
If (a_1 > 100) then
[A1]
a_2 <= a_1
End if
read a_2
If (a_2 > 100) then
[A2]
a_3 <= a_2
End if
read a_3
If (a_3 > 100) then
[A3]

method M2
[B]
read a_1
read a_4
read a_4 <= a_1
return a_1
else
[C2]
read a_2
return a_2
end if

method M3
[C]
read a_1
If (a_1 > 100) then
[C1]

call M2
End if

The DAVs constructed for method M1 are :

DAV (A) = [R,R,R,N]
DAV (A1) = [R,W,N,N]; DAV (A2) = [N,R,W,N]
DAV (A3) = DAV (M2) = [R,N,N,W]
DAV (M1) = DAV (A) ⊕ DAV (A1) ⊕ DAV (A2) ⊕
DAV (A3) = [R,W,W,W]

Similarly, the DAVs for M2 and M3 are :

DAV (M2) = [R,N,N,W]; DAV (B) = [R,N,N,W]
DAV (M3) = [R,R,N,N]; DAV (C) = [R,N,N,N]
DAV (C1) = [R,N,N,N]; DAV (C2) = [N,R,N,N]

While in the scheme proposed in [4], the DAVs for the methods would be:

DAV (M1) = [R,W,W,W]; DAV (M2) = [R,N,N,W]
DAV (M3) = [R,R,N,N]

After the construction of the breakpoints' DAVs in all methods, we create a commutativity relation of methods in the form of a table. In this table, a lock requester's entries contain names of DAVs of all methods (denoted as M_i where Mi is the name of the method). For example, M_1 represents a DAV of the method M1, which is [R,W,W,W]. A lock holder's entries contain names of DAV of the each method (denoted as M_i), name of the DAV of the first break point (denoted as N) in Mi and names of the DAVs of other break points (denoted as N_i where $1 \leq i \leq$ number of breakpoints -1) in method Mi. For example, in method M1, M_1, A, A_1, A_2 and A_3 represent the following DAVs, [R,W,W,W], [R,R,R,N], [R,W,N,N], [N,R,W,N] and [R,N,N,W], respectively. Since we assume the worst case access mode for each attribute before execution, lock requesters always have the most restrictive access modes (i.e., DAVs of methods). But, after a method execution, a lock holder may have a less restrictive access mode (i.e., DAV of the first or of the other break points). Two break points commute if their corresponding DAVs commute. Two DAVs commute if, for every attribute, its access mode in the two DAVs commute. Fig. 1 and 2 gives the commutativity tables constructed in our scheme and in the scheme proposed in [4].

Our concurrency control is based on two-phase locking [8]. When a transaction invokes a method on an object, it gets a lock containing the DAV of the method. As the transaction meets a break point during run-time, the break point is recorded. After the method execution, the lock is changed from M_i to N, N_1,...N_n where N is the name of the DAV of the first break point in Mi and N_1...N_n are the names of the DAVs of other break points encountered during the method execution Mi. Since the union of DAVs of N, N_1,...N_n may be less restrictive than the DAV of M_i, this can give more concurrency to other transactions which request locks on the same object. For example, assume that a transaction T1 invokes a method M1 on instance i1 of class Y and has break points A, A_1, and A_2 after the execution of M1. Assume that another transaction T2 comes and invokes a method M2 on the same instance i1 while T1 still has a lock on i1. Applying our technique gives commutativity between M1 and M2 since a method M2 commutes with each of A, A_1 and A_2, by the commutativity table in Fig. 1. On the other hand, M1 and M2 do not commute by checking the commutativity table in Fig. 2 if the scheme in [4] is adopted. Note that O means commute, and X not commute.

		lock holders									
		M_1	A	A_1	A_2	A_3	M_2	M_3	C	C_1	C_2
lock requester	M_1	X	X	X	X	X	X	X	O	O	X
	M_2	X	O	O	O	X	X	O	O	O	O
	M_3	X	O	X	O	O	O	O	O	O	O

Figure 1. A commutativity table of our scheme

		lock holders		
		M1	M2	M3
lock requester	M1	X	X	X
	M2	X	X	O
	M3	X	O	O

Figure 2. A commutativity table for the scheme in [4]

3.2. How to reduce the number of break points?

In our work, a method may have many break points depending on the method's logic. This requires larger commutativity tables and also incurs much run-time overhead for lock changes and commutativity checking. Thus, we need a way to reduce the number of break points in a method in order to reduce this overhead. We propose some strategies to reduce the number of break points as follows.

• Breakpoint optimization strategy 1

We know that the union of DAVs encountered after a method execution is at least as restrictive as the DAV of the first break point. Thus, if a DAV of some break point in a method has equal or less restrictive DAV than the DAV of the first break point, we do not have to keep track of it and do not include it as a member of the commutativity table. For example, the DAV of the break point C1 of method M3 in the previous example is [R,N,N,N], which is the same as the DAV of the first break point C. Thus, we do not have to include the DAV of C1 in the commutativity table.

• Breakpoint optimization strategy 2

In this strategy, we define *the most restrictive access mode* (MRAM) for each method. MRAM can have one of two values, R (Read) or W (Write). A method *m* has MRAM = R if it is a read method. On the other hand, a method *m* has MRAM = W if there is at least one attribute with a W mode in method *m*. Also, we define *Access Mode Change Percentage* (AMCP), $0 \le AMCP \le 100$, for each break point. The AMCP of break point B_i in method *M* is defined as follows.

$$\text{AMCP of breakpoint } B_i = \frac{\textit{number of attributes in Bi whose access mode is MRAM}}{\textit{number of attributes in M whose access mode is MRAM}}$$

For example, if the DAV of B in a method M2 and DAV of M2 are defined as [R,R,W,N] and [W,W,W,R], respectively, then AMCP of B is 33% (=1/3) since MRAM is W and the number of attribute in B whose access mode = W (MRAM) is 1, and also the number of attributes in M2 whose access mode = W is 3.

To reduce the number of break points, we let a break point, say B, have a DAV and be an entry in the commutativity table only if AMCP of B is greater than P% where P ($0 \le P \le 100$) is defined as restrictive threshold. Otherwise, we perform the following operation and do not include the DAV of B in the commutativity table.

the DAV of the first break point = the DAV of the first breakpoint $\oplus$ *the DAV of B*

For example, consider the following DAVs of methods M1, M2, and M3.

method M1

DAV (M1) = [R,W,W,W]; DAV (A) = [R,R,R,N]
DAV (A1) = [R,W,N,N]; DAV (A2) = [N,R,W,N]
DAV (A3) = [N,R,R,R]

method M2: DAV (B) = [R,N,N,W]

method M3

DAV (M3) = [R,R,N,N]; DAV (C) = [R,N,N,N]
DAV (C1) = [R,N,N,N]; DAV (C2) = [R,R,N,N]

MRAM(M1) = W; MRAM(M2) = W; MRAM(M3) = R

AMCP of each break point in each method is as follows.

For M1: AMCP (A) = 0; AMCP (A1) = 33.3;
AMCP (A2) = 33.3; AMCP (A3) = 0
For M2: AMCP (B) = 100

For M3: AMCP (C) = 50; AMCP (C1) = 50;
AMCP (C2) = 100

Suppose we define P as 30%. Then, we have the following break points participating as entries in the commutativity table.

For M1: DAV (A) = [R,R,R,R]; DAV (A1)= [R,W,N,N];
DAV (A2) = [N,R,W,N]
For M2: DAV (B) = [R,N,N,W];
For M3: DAV (C) = [R,N,N,N]; DAV (C2) = [R,R,N,N]

Note that [A3] is added to the DAV of [A] and also [C1] is removed due to the breakpoint optimization strategy 1.

Strategy 1 is to eliminate any breakpoint which is not helpful to increase concurrency. Strategy 2 is to give trade-off between concurrency and run-time overhead. That is, the higher P is, the less run-time overhead is; but this results in less concurrency. On the other hand, with less P value, we can provide more concurrency at the expense of run-time overhead. We can apply strategy 1 to any method without further information such as the frequency of method invocations. But, it provides a limited form of concurrency. On the other hand, in strategy 2, we can change the level of concurrency for each method by adjusting its P value.

This optimization process is done at compilation time. Thus, we do not have to optimize break points for each method invocation, resulting in a reduction in run-time overhead. However, for OODBS whose schema is continuously evolving, the optimization incurs some overhead since method contents (also corresponding DAVs) may change frequently.

4. Conclusion

This paper presents a scheme to enhance concurrency among instance methods in OODBS. Due to space limit, we do not include a scheme to provide better concurrency among class definition access and concurrency between class definition access and instance access.

In this study, we do not consider class hierarchy, which is an important property in OODBS. Due to inheritance, a query-type access or class definition access may involve a class and all its subclasses. Currently, we are developing a scheme, which incorporates class hierarchy into our current work, aiming at less locking overhead on class hierarchy.

References

[1] D. Agrawal and A. E. Abbadi, “A Non-Restrictive Concurrency Control for Object-Oriented Databases”, Proc. of the 3rd Int. Conf. on Extending Data Base Technology, Vienna, Austria, pp. 469 - 482, 1992

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

[3] M. Cart and J. Ferrie, “Integrating Concurrency Control into an Object-oriented Database System”, 2nd Int. Conf. on Extending Data Base Technology, Venice, Italy, pp. 363 - 377, 1990

[4] C. Malta and J. Martinez, “Automating Fine Concurrency Control in Object-Oriented Database”, Proc. of the 9th Int. Conf. on Data Engineering. Vienna, Austria. pp. 253 - 260, 1993

[5] B. Badrinath and K. Ramamritham, “Synchronizing Transactions on Objects”, IEEE Transaction on Computers, Vol. 37, No. 5, pp. 541 - 547, 1988

[6] B. Badrinath and K. Ramamritham, “Semantic Based Concurrency Control: Beyond Commutativity”, ACM Transactions on Database Systems, Vol. 17, No. 1, pp. 163 - 199, 1992

[7] P. K. Chrysanthis, K. Raghuram, and K. Ramaritham, “Extracting Concurrency from Objects : A Methodology”, Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data, pp. 108 - 117, 1991

[8] K. P. Eswaren, J. P. Gray, R. A. Lorie, and I. L. Traiger, “The Notions of Consistency and Predicate Locks in a Database System”, Communications of the ACM. Vol. 19, No. 11, pp. 624 - 633, 1976