Supporting Fine Concurrency in Object-Oriented Databases

Woochun Jun and 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. 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, concurrency is increased further with the use of dynamic information.

1 Introduction

In Object-Oriented Data Base Systems (OODBS), one of main concerns is to increase concurrency among methods so that more transactions can run in parallel. Since OODB allows arbitrary user-defined methods rather than simple reads and writes, we may increase concurrency by exploiting semantics of methods. 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 the results of the methods. Two methods conflict each other if they do not commute. In this paper, we present a scheme to increase concurrency among methods. Our scheme has the 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. Finally, it takes dynamic information into consideration to improve concurrency.

The paper is organized as follows. In the next section, we present previous studies and discuss their advantages and disadvantages. In section 3, we propose a scheme to increase concurrency. The paper concludes with further work with in section 4..

2 Previous Work

Several techniques have been proposed to increase concurrency among methods[1,2,3]. Usually, these techniques require application programmers to specify commutativity of methods. A process of commutativity relation construction is a serious burden on application programmers. Recently in [4], they automate the process of constructing commutativity relation from method contents. It is based on the notion of *affected sets of attributes* [1]. 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.

In their work, commutativity of methods is determined at compile-time so that run-time commutativity checking is avoided. They make use of the concept of Direct Access Vector (DAV) 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. Access mode information is syntactically extracted from the source code of the method at compile-time. Two methods commute if their corresponding DAVs commute. In turn, two DAVs commute if their access modes are compatible for each attribute. This commutativity relation is defined in the form of a table. This approach can provide finer concurrency than mere read and write conflicts. However, this technique does not increase much concurrency since it is too conservative. That is, it does not use any dynamic information when a method is executed on an object.

3 Our Approach

Our work improves the scheme developed in [4] to provide better concurrency. Our scheme has two objectives. First, we still automate the process of

commutativity relation construction. Second, we increase concurrency among methods by exploiting run-time information.

Our work is based on [5,6]. In their work, by pre-analysis, data sets for transactions can be obtained where a data set is a set of data items that a transaction might access. Since the presence of control structures within a transaction reduces the set of data actually used, each decision point (conditional statement) makes the transaction access a subset of its data set. Thus, as the transaction meets a decision point, its data subset, which is usually much smaller than its complete data set, is formed. Using this subset, data conflicts between the transaction and other transactions requesting data at this point can be reduced.

In our work, we need a two-phase pre-analysis : 1) construction of DAV of each break point and 2) construction of a commutativity table of DAVs. In each method, a break point is inserted by a programmer when a new subset of data can be obtained by executing a conditional statement. At compile time, the DAV of each break point is constructed. The construction of a DAV of a break point in a method has three steps : 1) construct an *initial* DAV of each break point where each value of the initial DAV denotes the most restrictive access mode of each attribute used by statements between this break point and the next break point (or end of the method). An access mode can have three values, N (Null), R (Read) or W (Write). 2) construct a method tree and an *intermediate* DAV of each break point. In an intermediate DAV, each value denotes the access mode used by statements from the beginning of the method but before the next break point though an execution path. Also, a method tree shows relationships among break points. For any two break points, say Bi and Bj, Bi is an ancestor of Bj if Bj can be reached from Bi through a conditional statement execution. In this case, Bj is called a descendent of Bi. 3) construct a *final* DAV of each method. A final DAV represents join of access modes of each attribute used by statements before this break point and the worst case access mode of each attribute used by statements from this break point to the end of the method. Thus, the final DAV of the first break point in a method represents the worst case access mode of attributes in the method.

After the construction of final DAVs of break points, we construct a commutativity relation of methods in a form of a table. In a commutativity table, a lock requester's entry contains the name of the first break point in each method. A lock holder's entry contains the name of all break point in each method. Two methods commute if their *final* DAVs commute. Two *final* DAVs commute if the access modes of each attribute commute.

When a transaction invokes a method on a data object, it gets a lock containing the first break point in the method. As the transaction meets a break point during run-time, the lock is changed to contain the name of the current break point which may have less restrictive final DAV. Thus, it can give more concurrency to other transactions which request locks on same data object at this point.

4 Further Work

In our work, a method may have many break points depending on the method's logic. This requires bigger commutativity tables and also incurs much run-time overhead for lock change. Thus, we need to devise a way to reduce the number of break points in a method in order to reduce space overhead and time overhead for lock changes during run-time.

References

[1] B. R. Badrinath and krithi Ramamritham, "Synchronizing Transactions on Objects", IEEE Transaction on Computers, Vol. 37, No. 5, May 1988, pp. 541 - 547.

[2] B. R. Badrinath and Krithi Ramamritham, "Semantic-Based Concurrency Control : Beyond Commutativity", ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992, pp. 163 - 199.

[3] Panos K. Chrysanthis, S. Raghuram and Krthi Ramaritham, "Extracting Concurrency from Objects : A Methodology", Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 117.

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

[5] D. Hong, T. Johnson and S. Charavathy, "Real-Time Transaction Scheduling : A Cost-Conscious Approach", Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data, 1993, pp. 197 - 206.

[6] S. Chakravathy, D. Hong and T. Johnson, "Real-Time Transaction Scheduling : A Framework for Synthesizing Static and Dynamic Factors", Tech. Report no. 008, Dept. of CIS, Univ. of Florida, 1994.