A Parallel Execution Model for Database Transactions

Javam C. Machado
Dep. de Computação, Univ. Federal do Ceará
CP 12166 - Fortaleza, Ce - 60455-760 - Brazil
javam@lia.ufc.br

Christine Collet
LSR-IMAG, University of Grenoble
BP 53 38041 Grenoble cedex 9, France
Christine.Collet@imag.fr

Abstract

This paper explains our approach for optimizing the execution of object-oriented database transactions. We provide for parallel execution of methods inside atomic transactions. Our approach enhances parallel method execution without user-defined control structures. Optimization is based on method compatibility which is automatically determined during the method compilation phase. Optimized execution plans are efficiently built when transactions are compiled. An execution plan is then used for generating new transaction code where compatible methods are scheduled for parallel execution.

Keywords: Object-Oriented Databases, Parallel Databases, Transaction processing

1. Introduction

This paper presents the approach we propose for providing parallel method execution without user-defined control structures. The approach is completely transparent to method definition and is efficient since optimization is performed at compilation time. Moreover our approach is likely to have a better performance as we do not have to create a transaction each time two methods are concurrently executed.

In our approach, optimizing transaction execution means providing implicit parallelism between methods of a transaction. The new transaction execution model we propose supports synchronous and asynchronous method execution. This is accomplished by creating an optimized transaction execution plan each time a transaction is compiled and using this plan to transform the transaction code.

Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1-4, 1997

The paper is organized as follows. Section 2 gives the main characteristics of the O2 database system. Section 3 briefly explains our approach for optimizing transaction execution based on method compatibility. Section 4 concentrates on the resulting method compatibility relationship. Section 5 shows how this relationship is used for building transaction execution plans with parallelism. Section 6 describes related work and finally, a brief conclusion and a short presentation of the prototype implementing our proposals are given in Section 7.

2. O2 DBMS Background

This section reviews the main characteristics of the O2 System. The reader may find further details in [1, 6]. Figure 1 shows how an O2 application is organized.

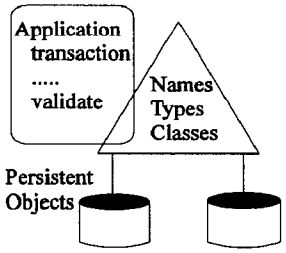

Figure 1: *Organization of an O2 application*

An O2 *schema* is a set of definitions. A schema consists of a set of classes related by inheritance links and/or composition links, types, functions and applications. A class definition consists of a type definition and a set of methods. A type is defined recursively from atomic types (integer, Boolean, char, string, real, and bits) or classes and constructors (tuple, list and set).

An O2 base groups together objects and values which have been created in compliance with a schema. An object is an instance of a class and has an identity, a value and a behavior defined by its methods. The type of the value is the one given by the class definition and methods are those of the

class. Independently from objects, O2 also manages values which are instances of a type. In the remainder of this paper, the entity concept is used for referencing either objects or values.

By default, entities created during program execution are not persistent. To become persistent, an entity must be directly or indirectly attached to a *name*, i.e., a persistent root belonging to a schema. The entity attached to the root is stored in one of the database associated with the schema, the one "active" when the attachment is done.

Methods are defined using O2C, which allows to express manipulations on persistent entities as well as non-persistent entities. Persistent entities are manipulated only inside transactions. The query language of the system, O2SQL, can be used to express Boolean expressions on entities as well as SQL queries on collections. O2SQL queries can be executed either in an interactive mode or as part of the body of a method or a program.

An O2 application is a set of programs and belongs to a schema. When an application starts up, it opens a read-only transaction. Programs of the application executed within this transaction can manipulate transient entities (apply methods, modify values, etc.) but are restricted to read-only access to persistent entities. A (read-write) transaction must be initiated before updating persistent entities in order to ensure consistency.

Transactions in O2 are atomic transactions. Commands such as transaction, validate, commit and abort are provided for managing transactions. Actually the O2 transaction model does not support nested transactions. Any time a **transaction** command is issued inside an existing transaction, it is ignored. There is only one transaction at a time in a running O2 application. Transactions send messages to transient or persistent objects to execute their methods.

3. Overview of the Approach

In our approach, two kinds of information are extracted at compilation time for defining compatibility between methods. First, we consider method access sets over attributes and persistent roots defined in the schema the methods belong to. A method may "read" (access) or "write" (update) attributes and persistent entities. As a method may concern entities of different types, it is possible to build two access sets describing the set of entities that can be accessed or modified during method processing, respectively.

Let us consider that methods m_1 and m_2 are defined as follows:

Code of the method m_1

```
o2 Class_X x;
int a=12, b;
   x = new Class_X;
   Root += set(x);
   x->m1a(a);
   b=x->m2a;
```

Code of the method m_2:

```
o2 set(Class_X) y;
o2 Class_X x;
int b;
   y = Root;
   for x in y do {
         b=x->m2a;
         ...
      }
```

In the source above, Root is a persistent collection. m_{1a} modifies the attribute A of x, while m_{2a} reads the attribute B of x.

For building the access sets of m_1, we use the *environment* of the method which is a space of definitions that can be shared by its components. The environment of a method is composed of definitions of attributes and roots of the schema the method belongs to. Others kind of information is not interesting for our purposes. Local variables may directly or indirectly denote objects or persistent roots. They can be read or even modified by the method action. In that case, we are interested only in the messages sent from the method.

The environments of methods m_1 and m_2 can be represented as the set of names {Root, ...}. Method m_1 modifies the persistent collection Root and method m_2 accesses objects of Root. The access sets of m_1 are: read=∅ and write={Root} and those of m_2 are: read={Root} and write=∅. Considering the access sets of both methods, we determine that there is a conflict: m_1 "writes" on Root and m_2 "reads" Root.

If now we assume the access sets for m_1 and m_2 as: read={Root} and write=∅ then we cannot say that the methods are compatible as we have to check that method m_{1a} is compatible with m_{2a} and that m_{2a} is compatible with itself. Given the information on m_{1a} and m_{2a}, it is clear that they are compatible. Even if they manipulate the same object, they do not modify a common attribute of this object.

4. Method Compatibility

The parallel execution of methods inside transactions relies on compatibility relation between methods. Compatible methods can be scheduled for parallel execution and still have their results *equivalent* to the results of a sequential execution of the same methods. This section explains how we build the compatibility relation over the methods of a schema. The reader may find further details in [19].

4.1 Notation

Partial function:

Let *Idf* be the set of Identifiers and *Type* = *{class, method, root, type, atomic, complex}*. Considering $id \in Idf$ and $t \in Type$, *t/i* is a partial function that belongs to the set of functions ENV=*Idf* $\rightarrow$*Type*:

t/id = λy. if $y = id$ then t else the function is not defined.

Domain of a function *t/id*:

Def(t/id) = { x | t/id(x) is defined}

Attributes and methods of a class:

The definition of an attribute of a class C is of the form *a:t* where *a* is an identifier and *t* is an atomic type or a complex type. Such a definition can be represented as a function *t/C.a*. For example, the function *atomic/Person.name* describes that *name* is an attribute of class *Person* and the function *complex/Person.address* describes the fact that *address* is an attribute of class *Person* with a complex type. A component attribute of *address* can also be described as a function, e.g., *atomic/Person.address.zip*. Given the definition of a class C, the attributes specified in this definition are given by:

$$\alpha^{C}_{Att} = \bigcup_{i=1}^{n} t_i \,/\, C.a_i, t_i \in \{atomic, complex\} \wedge a_i \in Idf$$

The definition of a method m of a class C can be represented as a function *method/C.m*. Given the definition of a class C having n methods m_i, the set of methods of C is represented as:

$$\alpha^{C}_{Meth} = \bigcup_{i=1}^{n} method \,/\, C.m_i, m_i \in Idf$$

Schema:

A schema is a set of definitions of classes, types, functions, applications and persistent roots. Given a schema *S*, it is possible to build α^S, a set of partial functions, each of them describing an element of the schema. For example, if *S* contains the definition of a class *Person*, the *class/Person* function belongs to α^S and describes the fact that *Person* is a class. Given an identifier $id \in Idf$, *class/Person(id)* tells us if *id* denotes the class *Person* or not. Also if the persistent root *ThePersons* defined as ThePersons: set(Person) belongs to *S* then the function *root/ThePersons* $\in \alpha^S$. In the following we give only the sets we are interested in for defining the compatibility of methods. Of course, we assume that we have α^S.

$a^S_{Class} = \{c \in a^S \mid c(id) = class\}$ is the set of class definitions.

$\alpha^S_{Meth} = \{m \in \alpha^S \mid \exists c \in \alpha^S_{Class} \wedge m \in \alpha^C_{Meth}\}$ is the set of methods of schema.

$\alpha^S_{Root} = \{pr \in \alpha^S \mid pr(id) = root\}$ is the set of persistent roots.

4.2 Primitive access sets of methods

For defining the primitive access sets of a method, we consider the O2C expressions and statements used in its body. The compilation phase of a method *m* of a class *C* in a schema *S* includes a new phase which consists in determining which attributes of an instance of *C* are accessed or modified. This phase also detects which persistent root of *S* are manipulated. Let *mode* be a function returning the access mode over an entity[1]:

mode : Idf $\rightarrow$ *{na, read, write}*

Definition 1 *Let S be a schema. The primitive access sets of a method in S is given by (m denotes a method of S, i.e.,* $m \in Def(\alpha^S_{Meth})$*):*

$$read_m = \{n \in Def(\alpha^C_{Att} \cup \alpha^S_{Root}) \mid mode(n) = read\}$$

$$write_m = \{n \in Def(\alpha^C_{Att} \cup \alpha^S_{Root}) \mid mode(n) = write\}$$

The following function β_m associates to a method *m* its primitive access sets:

$$\beta_m : m \rightarrow read_m \times write_m$$

This function belongs to the set of functions $Idf \rightarrow 2^{Idf} \times 2^{Idf}$ and $\beta_m[1] = read_m$, $\beta_m[2] = write_m$. The primitive access sets of methods β_S of a schema *S* are:

$$\beta_S = \bigcup \beta_{m_i}, m_i \in Def(\alpha^S_{Meth})$$

The β_S function may be represented as a table. For example, Table 1 gives the access sets of some methods of classes C_1 and C_2 belonging to a schema example.

	read	write	ext
$C_1.p$	$\varnothing$	$\varnothing$	
$C_1.r$	$\varnothing$	$\{C_1.at_4\}$	
$C_2.q$	$\{C_2.at_2\}$	$\varnothing$	

[1] *na* means "no access".

$C_2.s$	$\{C_2.at_1\}$	$\{C_2.at_3\}$	
$C_2.t$	$\varnothing$	$\{C_2.at_1\}$	
$C_2.u$	$\{C_2.at_5\}$	$\varnothing$	

Table 1: *Primitive access sets of some methods*

An important aspect in building access sets of a method relates to its arguments. Arguments having atomic types are considered as local variables. If an argument represents an instance of the class to which the method belongs, we consider it as *self* and therefore, we take it into account when building the primitive access sets.

Another important aspect relates to late binding. A method defined in a class can later be redefined in its subclasses (overriding). Since method code can depend on the actual class of objects, the binding between a method name and its code has to be carried out at run time. This makes impossible to know, at compilation time, which code will be executed exactly. Therefore, in building primitive access sets, we choose to keep the most restrictive access over the entities used in different codes associated to the same method name. Let us consider a class C_1 which has a method m_1 and a class C_2, subclass of C_1. Method m_1 is inherited by C_2 and is redefined in C_2 (cf. Figure 2). Assume that method $C_1.m_1$ "reads" root1 while method $C_2.m_1$ modifies it. The access sets of $C_1.m_1$ and $C_2.m_1$ are *read* = {root1} and write = {root1}. This is clearly a pessimistic approach, but it makes sure compatibility of methods is correct even though methods are inherited.

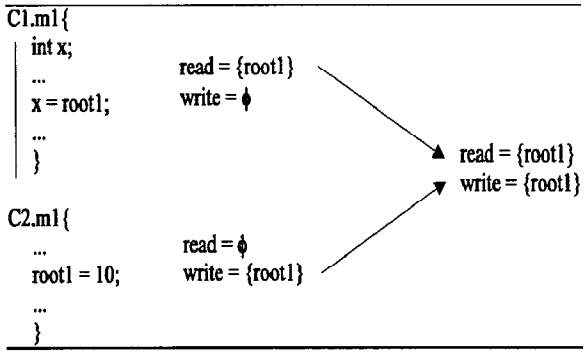

Figure 2: *Method overriding*

4.3 Extended access sets

In the primitive access sets of a method m of a class C, we consider only attributes of C. However, m may have indirectly some effects on attributes of the same class or of other classes. More generally, if a method m_1 calls another method m_2 of the same class or of another class, then the access sets of m_1 takes in account the access sets of m_2 which in turns takes into account the sets of any called method. In order to get the final access sets of a method m of a schema, we consider the access sets and the control graph of m.

Figure 3 shows that the control flow among methods possibly called during the execution of a method p can be represented as a directed acyclic graph, so-called a *control graph*. In such a graph, nodes represent methods (more precisely method identifiers) and edges represent the control flow. The root of the graph is the method of interest. The control structure is denoted by $\rightarrow$ and the semantics of $C_1.m_1 \rightarrow C_2.m_2$ is that the execution of $C_2.m_2$ should sequentially follow $C_1.m_1$. This means that method m_1 of class C_1 calls method m_2 of class C_2. In Figure 3, we have $C_1.p \rightarrow C_2.q$ and $C_1.p \rightarrow C_1.r$ which means that method p of class C_1 calls method q of class C_2 and method r of class C_1.

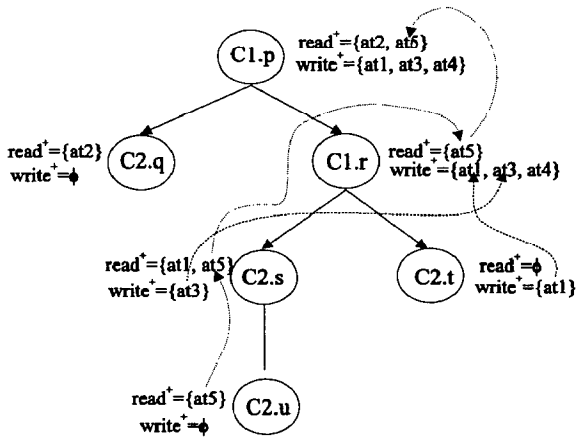

Figure 3: *Building the extended access sets of methods using control graphs*

Also in Figure 3 we show how we extend the access sets of method p of class C_1 considering the control graph of p and assuming we have the primitive access sets of Table 1. Method u does not call methods and its primitive access sets are also its extended access sets. The construction of the extended access sets of $C_2.s$ adds $C_2.at_5$ to $read_s$. Extending access sets of $C_1.r$ takes into account the extended access sets of $C_2.s$ and $C_2.t$. It first adds $C_2.at_1$ and $C_2.at_5$ to $read_r$, and $C_2.at_3$ to $write_r$. Then, proceeding $C_2.t$, it adds $C_2.at_1$ to $write_r$. Finally, it removes $C_2.at_1$ from $read_r$. After extending access sets, Table 1 has the configuration given in Table 2.

	read	write	ext
$C_1.p$	$\{C_2.at_2, C_2.at_5\}$	$\{C_2.at_1, C_2.at_3, C_1.at_4\}$	√
$C_1.r$	$\{C_2.at_5\}$	$\{C_2.at_1, C_2.at_3, C_1.at_4\}$	√
$C_1.q$	$\{C_2.at_2\}$	$\varnothing$	√
$C_2.s$	$\{C_2.at_1, C_2.at_5\}$	$\{C_2.at_3\}$	√
$C_2.t$	$\varnothing$	$\{C_2.at_1\}$	√
$C_2.u$	$\{C_2.at_5\}$	$\varnothing$	√

Table 2: *Extended access sets of some methods*

Let us consider the compilation phase of a method m and let us assume that this method has been compiled and that its primitive access sets have been calculated: $\beta_S(m)$ is defined. Given the control graph of method m, the following algorithm is applied (at the end of compilation phase) for extending the access sets of m and of possibly other methods.

Algorithm Access_set_extension
Input: A control graph of a method m, β_S
Ouput: Extended access sets of m

```
extend(node, root, βS) {
if ( node = nil ) then
    root.ext = true );
else
    while node ≠ nil do {
        extend(succ(node), node, βS);
        βS(root)[1] = βS(root)[1] ∪ βS(node)[1];
        βS(root)[2] = βS(root)[2] ∪ βS(node)[2];
        node = alt(node);
    };
    βS(root)[1] = βS(root)[1] - βS(root)[2];
}
```

In order to make a difference hereafter between a primitive access set and its corresponding extended access set we use the following:

- m denotes a method of a schema S, i.e., $m \in Def(\alpha^S_{Meth})$;
- $write^+_m$ is the extended write access set of m;
- $read^+_m$ is the extended read access set of m

4.4 Compatibility

The extended access sets of a method give us information on the way the method interacts directly or not on attributes of classes and persistent roots. Based on those sets, compatibility relation is defined for all methods of a schema. Two methods m_1 and m_2 of a schema S are compatible if they are not in conflict, i.e., if an element of the extended write access set of one of the methods does not belong to extended access sets of the other.

Definition 2 *The compatibility relation between methods of a schema is given by:*

$$\forall m_1, m_2 \in \alpha^S_{Meth},\ Mcompatible(m_1, m_2) \Leftrightarrow$$
$$read^+_{m1} \cap write^+_{m2} = \varnothing \wedge$$
$$write^+_{m1} \cap read^+_{m2} = \varnothing \wedge$$
$$write^+_{m1} \cap write^+_{m2} = \varnothing$$

The *Mcompatible* relationship can be materialized as a matrix. Figure 4 gives the matrix built for methods in Figure 3. One entry in line *l* and column *c* gives the result of *Mcompatible(l,c)*.

	C_1.p	C_1.r	C_2.q	C_2.s	C_2.t	C_2.u
C_1.p	false	false	true	false	false	true
C_1.r	false	false	true	false	false	true
C_2.q	true	true	true	true	true	true
C_2.s	false	false	true	false	false	true
C_2.t	false	false	true	false	false	true
C_2.u	true	true	true	true	true	true

Figure 4: *Example of a Mcompatibility matrix*

5. Parallelizing O2 Transactions

Our approach transforms the definition of an O2 transaction into an *execution plan* where methods are scheduled for parallel execution. This plan is build based on the order of methods inside transactions and on compatibility relation. Compatible methods are scheduled for parallel execution while scheduling follows definition order for methods that have to be sequentially executed. Later on the plan is used to generate a new transaction definition code.

5.1 Execution plan

Each transaction is logically represented in terms of a *graph of methods*. The vertices are methods and the edges represent the execution order relation. For instance, if a method m_j immediately follows a method m_i in a transaction definition, than there is edge from m_i to m_j in the transaction graph. Consider a transaction and his graph given in figure 5.

```
   o2 C1 a;
   o2 C2 b;
(1)  transaction;
(2) b→u;
(3)   b→t;
(4)   b→q;
(5)   if ( condition ) {
(6)      a→p;
(7)      b→s; }
(8)  validate;
```

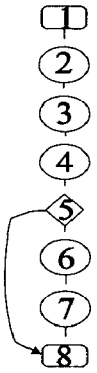

Figure 5: Transaction graph

There are three different types of vertices: (1) transactional, (2) conditional and (3) message. Transactional and conditional vertices always follow definition order and will not be moved in the graph. Message vertices are considered for transformation if the methods they represent are compatible.

Before transformation, the order between message vertices is totally defined so that each vertex has only one incoming and an outgoing edge. Graph transformation changes the relation between vertices, in a way that a vertex may have several incoming and outgoing edges. When a vertex has a number of outgoing edges, it starts a *set of sibling vertices* and is said to be a *spawn vertex*. Each sibling vertex has only one outgoing edge that arrives in a vertex called *barrier*. We call a *bloc* a set of vertices with a spawn vertex followed by a set of siblings and finished by a barrier vertex.

Graph transformation algorithm creates blocs of vertices inside transaction graphs. It follows a transaction graph and performs two main steps:

1. When consecutive message vertices represent compatible methods, they become siblings. This transformation is made until one of the following condition holds:
   - Next vertex is a transaction or conditional one.
   - The method of the next message vertex is not compatible with *one* of the actual siblings.
2. A barrier *vertex* is always introduced after siblings.

Considering the transaction graph of the figure 5, transformation would result in a new graph showed in figure 6. Vertices 2, 3 and 4 become siblings since they are consecutive and their methods are compatible. They are followed by a synchronization vertex right before vertex 5 which is a conditional vertex. On the other hand, vertices 6 and 7 are consecutive but their methods are not compatible, so the parent/child relation for those vertices is the same as before.

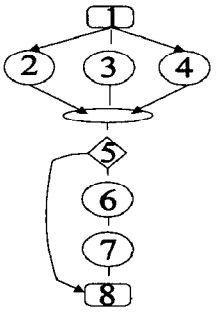

Figure 6: *Graph of a transaction after transformation*

5.2 Code Generation

We use the execution plan to generate a new O2 transaction code that will schedule for parallel execution of methods and will produce the same results as the transaction sequential code [19]. We generate new code mainly for blocs of vertices. The rest of the transaction code remains as it has been defined. So methods can be sequentially or concurrently executed in transactions.

Spawn and barrier vertices define the boundaries for parallel execution inside transactions. As a result parallel execution of methods will never transpose transaction boundaries. We assure that at the beginning and at the end (abort or validate) of a transaction there will be only a thread of control inside transactions. Using transactions as synchronization boundaries for parallel execution of methods implies that we would never have two transactions of the same application running in parallel. So all methods of a transaction have to finish execution before a new transaction is set up.

5.2.1 Spawning and Synchronizing threads

The code generation phase transforms sibling vertices of a bloc in system calls that create as many threads of control as the number of vertices. Each method of the bloc will be schedule for parallel execution in a new thread of control. Threads are asynchronously spawned and each one is sequentially executed although full parallelism is accomplished inside the bloc.

There is no need for synchronization while parallel threads are performed since they are completely independent one of the other. This is assured by the compatibility relation between methods which is the basis for bloc construction. Each thread will execute the method code until the end of method and control will be transferred back to the transaction code.

Thread creation calls have several formal parameters that allow identification of the method

to be executed and transfer of input and output method parameters. Threads are also uniquely identified. Thread id will be further used for synchronization purpose.

Barrier vertices generate synchronization code for parallel threads. We use a primitive called barrier that synchronizes all the running threads at the end of a bloc. After all methods have been executed, only one thread will be active and will continue the transaction execution.

In the following, we give the code generated by the execution plan of the figure 6. Note the system calls to create threads to execute methods $C_2.u$, $C_2.t$ and $C_2.q$. Right after these calls transaction issues a barrier to synchronize the spawned threads.

```
    o2 C1 a;
    o2 C2 b;
(1) transaction;
(2) lwp_create(Th1, b→u, 0, 0);
(3) lwp_create(Th2, b→t, 0, 0);
(4) lwp_create(Th3, b→q, 0, 0);
    barrier(Th1, Th2, Th3);
(5) if ( condition ) {
(6) a→p;
(7) b→s; }
(8) validate;
```

6. Related Work

Parallelism is not a new issue in database systems. There is a lot of work that deals with *parallel query processing* in the framework of relational systems [10, 22]. Several prototypes have already been built [5, 7, 11, 16, 25]. The issue in those approaches is the execution of a SQL query in parallel. Two forms of parallelism have been proposed: *intra-operation* and *intra-query* parallelism [22]. With object-oriented databases, new transaction models have been proposed. *The nested transaction model* also allows for exploiting parallelism in the context of database transactions [21]. Parallel execution of a transaction and its sub-transactions is said to be *vertical* while parallelism between two transactions of the same level is called *horizontal* [17]. Several prototypes offer a way of specifying parallelism between or inside nested transactions [12, 13]. Those approaches differ from the one described in this paper mainly in the way parallelism is specified. We generate parallelism by code transformation while they leave to programmers the responsibility of describing and controlling parallelism inside transactions. Besides our approach works in a classic transaction model.

The notion of *compatibility* has been used in the context of abstract data types mainly for concurrence control over shared objects [3, 23]. Compatibility between operations is usually defined as part of the abstract data type specification. Concurrence control algorithms used operation semantic information supplied in the type specification for dynamically defining compatibility between operations. Most of the compatibility operation checking procedure was made at runtime since the purpose was to augment object accessibility in concurrent transactions.

Commutativity and *recoverability* have lately been proposed as the natural successors for operation compatibility in the context of concurrent transactions [4, 9, 26]. Although they allow more concurrence than compatibility, these approaches are usually based on the functionality of transaction managers in order to validate operations in the correct order or rollback operations in case of invalid transaction history. We cannot use them since methods do not have a transaction semantics. Furthermore, we have decided to do as much as possible at compilation time in order to have less overhead at execution time.

In [20] we described a parallel execution model for a database programming language called Peplom. This model is based on the notion of actors [2] and supports both parallelism between objects and between methods inside objects. Asynchronous message passage is the basis for parallelism between objects while synchronization techniques make sure the mutual exclusion of parallel execution of methods belonging to the same object.

In [8] we proposed a model for optimizing active rules with parallelism. Asynchronous rule execution has been provided in [14, 15, 18] by the means of decoupled transactions. However, those early works differ from the approach we describe in [8] in various aspects. In our approach parallel execution of rules takes place inside the triggering transaction and therefore is semantically different from decoupled transactions. Nested transactions are used in [18] to concurrently execute rules as sub-transactions of the triggering transaction. Transaction creation has to be specified within the rule definition explicitly in [14, 18] or using composite and transaction events in [15]. In [24], rule associations define execution order between rules. Tree kinds of associations are proposed to programmers for specifying a precedence relationship between two rules, a rule and a set of rules, or a set of rules and a rule. Rules in a set can be executed in parallel. The strategy used for parallel execution of methods in a set is based on the control flow among the methods. Briefly speaking, two methods which are not involved in more than one association are said to be independent. If two methods/statements are independent of each other they are executed in parallel.

7. Conclusion

In this paper we defined our approach for exploring parallelism in object-oriented database outside of the context of SQL queries. We proposed a technique for parallelizing transactions in a flat classical transaction model where a transaction is a sequence of methods. The technique is based on compatibility relation of methods which is automatically determined at method compilation phase. Intra-transaction parallelism is accomplished by transforming transaction definition code in order to execute methods in parallel.

We implemented a prototype for the intra-transaction parallelization model using the O2 object-oriented database system, which we described in this paper. The prototype introduces parallel execution inside O2 transactions through creation and synchronization of threads of control inside an O2 client running an application. We extended the O2 DBMS with two new modules:

- The *compatibility table generator* which is responsible for calculating compatibility relation between methods of a schema. It also manages the compatibility relation table as O2 named object.
- The *transaction* parallelizer which transforms O2 transaction definition code for scheduling methods for parallel execution.

Our prototype runs in a monoprocessor Unix-like workstation and supports virtual parallelism.

We already have some results on applying the intra-transaction parallelization model on a nested transaction model [19]. In this context, we combine our technique of parallelizing method execution inside transactions with the power of inter-transaction parallelism of the nested transaction model. Further investigation is still on the way and the results will be published later.

8. References

[1] M. Adiba; C. Collet. *Objets et Bases de Données - Le SGBD O2*. Hermes, 1993

[2] Actors: A Model of Concurrent Computation in Distributed Systems. The MIT Press, 1986.

[3] B.R.Badrinath; K.Ramamritham. Synchronizing transactions on objects. *IEEE Transactions on Computer*, 37(5):541-547, May 1988.

[4] B.R. Badrinath; K.Ramamritham. Semantic-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163-199, March 1992.

[5] B. Bergsten; M. Couprie; M. Lopez. DBS3: A parallel database system for shared store. *In Proc of the 2nd Int. Conf. on Parallel and Distributed Information Systems*, pages 260-262, San Diego, Jan 1993.

[6] F. Bancilhon; C. Delobel; P. Kanellakis, editors. *Building an Object-Oriented Database System - The History of O2*. Morgan Kaufmann, 1992.

[7] G. Copeland; W. Allexander; E. Boughter; T. Keller. Data placement in Bubba. *In Proc of the 1988 ACM SIGMOD Int. Conf. on Management of Data*, pages 99-108, Chicago.May 1988.

[8] C. Collet; J.C. Machado. Optimization of active rules with parallelism. In *Proc of the Int. Workshop on Active and Real Time Databases*, Skövde, June 1995.

[9] P.K.Chrysanthis; S.Raghuram; K.Ramamritham. Extracting concurrency from objects: A methodology. In *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, pages 108-117, Denver, June 1991. ACM Press.

[10] D.J. DeWitt; J. Gray. Parallel database systems: The future of high Performance database systems. *Communications of the ACM*, 35(6):85-98, June 1992.

[11] D.J. DeWitt; S. Ghandeharadizeh; D. Scheiner; A. Bricher; H.I. Hsiao; R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):112-124, Mar 1990.

[12] P. Dasgupta; R.J. LeBlanc Jr.; W.F. Appelbe. The Clouds Distributed Operating System. *In Proc of the 8th Int. Conf. on Distributed Computing Systems*, pages 2-9, San Jose, June 1988, IEEE.

[13] J.L. Eppinger; L.B. Mummert; A.Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*, 1991, Morgan.

[14] S. Gatziu; A. Geppert; K.R. Dittrich. Integrating active concepts into an object-oriented database system. In *Proc. of the 3rd Int. Workshop on Database Programming Languages: Bulk Types & Persistent Data*, pages 399-415, Nafplion, 1991. Morgan.

[15] N. Gehani; H.V. Jagadish; O. Shmueli. Event specification in an active object-oriented database. In *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, pages 81-90, San Diego, 1992.

[16] G. Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120-135, Feb 1994.

[17] T. Härder; K. Rothermel. Concurrency control issues in nested transaction. *The VLDB Journal*, 2(1):39-74, Jan 1993.

[18] M. Hsu; R. Ladin; D. McCarthy. An execution model for active database management systems. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases*, pages 171-179, June, 1988.

[19] J.C. Machado. Parallélisme et Transactions dans les Bases de Données à Objets, PhD Thesis, Université de Grenoble, 1995.

[20] J.C. Machado; C. Collet; B. Defude; P. Dechamboux; M. Adiba. The parallel PEPLOM execution model. In *proc. of 31st Annual ACM Southeast Conference*, Birmingham, Apr 1993.

[21] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[22] E. Omiecinski. Parallel relational database systems. In *Modern Database Systems*, W. Kim, ed., pages 494-512, ACM Press, 1995.

[23] M. Roesler; W. Burkhard. Concurrency control scheme for shared objects : A peephole approach based on semantics. In *Proc. of the 7th Int. Conf. on Distributed Computing Systems*, pages 224-231, Berlin, September 1987.

[24] S.Y.W. Su; R. Jawadi; P. Cherukuri; Q. Li; R. Nartey. OSAK*.KBMS/P: A parallel, active, object-oriented knowledge base server. Technical Report TR94-031, University of Florida, Gainesville, 1994.

[25] M. Stonebraker; R. Katz; D. Patterson; J. Ousterhout. The design of XPRS. *In Proc of the 14th Int. Conf. on Very Large Data Bases*, pages 318-330, Los Angeles, Aug 1988.

[26] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computer*, 37(12):1488-1505, Dec 1988.