Integrating an Entry Consistency Memory Model and Concurrent Object-Oriented Programming *

Antonio J. Nebro, Ernesto Pimentel and José M. Troya

Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga
E.T.S.I en Informática. Campus de Teatinos.
E29071- Málaga (SPAIN)
e-mail: {antonio, ernesto, troya}@lcc.uma.es
Phone: +34 5 2133310
Fax: +34 5 2131397

Abstract. Entry consistency is a weak memory consistency model that makes possible the efficient implementation of distributed shared memory (DSM) languages and systems. In this paper we study a way an entry consistency based memory model can be integrated into concurrent object-oriented languages. One problem to be solved is how to satisfy the entry consistency requirements when objects include synchronization constraints. We propose a solution based on the definition of a programming model with the following characteristics: (1) the establishment of ownership relations among objects, (2) a distinction between command and query operations, and (3) a lazy wait synchronization mechanism. Preliminary results show that significant speed-ups can be obtained.

1 Introduction

Object orientation provides a high modeling power and a set of techniques for the development and maintenance of software that benefit its reusability. On the other hand, concurrency allows the exploitation of parallel architectures to obtain speedups when solving a given problem, or to solve larger problems. Concurrent object-oriented languages are attractive because they combine the advantages of both paradigms. A number of concurrent object-oriented programming models have been proposed, but none of them have been widely accepted. Several reasons have been argued to explain this lack of agreement [5]. A general opinion is that implementation of these languages is inefficient [3] [4] because object communication is based on some kind of message passing, making communications among local objects more expensive in comparison to the invocation costs in sequential languages. A related problem is the high cost of remote invocations in distributed systems.

* This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC94-0930-C02-01.

The problem of obtaining good parallel performance requires cost reductions in both local and remote invocations. We focus mainly on the latter issue. The purpose of our work is to study the applicability of distributed shared memory (DSM) schemes [6] to concurrent object-oriented languages to reduce the costs of remote invocations when they are implemented in distributed systems. In particular, we want to study what requirements the object model has to fulfill to allow object replication and its consequences for the programming model. Distributed shared memory schemes are popular because, by using replication and migration techniques, they allow the performance of language and system implementations to be increased. The choice of a memory consistency model is a trade-off between increasing concurrency and programming model complexity. We choose the entry consistency memory model [2], a weak memory model that associates a synchronization variable to each shared data unit. We also propose a programming model that fits into the non-orthogonal, uniform and integrated category of Papathomas's classification of concurrent object-oriented languages [7]. Examples of languages of this category are those based on actors [1]. The main advantage of these languages is that they contribute to reducing the complexity problem, because concurrent programs only have one kind of component, the object, which has a well defined structure and behaviour.

The rest of this paper is organized as follows. In Section 2, the programming model we propose is described. Preliminary results are presented in Section 3. Finally, some conclusions and future research are outlined.

2 A Programming Model based on Entry Consistency

We propose a programming model characterized by an object and a concurrency model. An object is an entity having three basic components: a state, an interface and a set of synchronization constraints. The state can only be accessed through the operations defined in the interface. Synchronization constraints prevent the acceptance of operations when these are not allowed. Objects may invoke operations on other objects, create more objects and modify their internal state because of an operation is performed. A parallel program is composed of a collection of objects (some of which can be replicated) that communicate among themselves invoking the operations defined in their interfaces. Objects cannot simultaneously execute more than one operation that can modify its state.

Objects establish among themselves owner-owned relations. An object can only invoke the operations of another object if the latter is owned or acquired by the former. The ownership can be exclusive or non-exclusive. An exclusive ownership permits the owner to access the acquired object in mutual exclusion, while non-exclusive ownership allows the same acquired object to be accessed by several owners. If an owned object is replicated, the owners can access it in parallel. When an owner object is not going to access an owned object again, it releases the object. The rules that govern on object's ownership follow a protocol satisfying the entry consistency requirements, with the following exception: the owned object must be temporarily released when an invocation cannot be solved

because of a synchronization constraint. Therefore, a different object can access it to change the state variables and allow the original owner to resume its activity.

We distinguish two kinds of operations: queries and commands. A *query* is an operation which returns some information about the object, without modifying its state. A *command* is an operation that modifies the state of the object, without returning information about it. Thus, queries imply pure read accesses and commands imply pure write accesses. If an object is replicated or resides in a multiprocessor system, several queries can be performed in parallel, without altering the basic object model, because this fact is transparent.

The concurrency model is based on the different behaviour of query and command operations. Command invocations are asynchronous, so the owner object can continue its execution in parallel with the owned object. If the owner object invokes several commands sequentially, these must be accepted by the owned object in the same order. Query invocations are synchronous, so the owner must wait for the response of the owned object. If the owner object has issued one or more command invocations before a query, this query will be accepted when all the commands have been performed. Query invocation leads the owner and the owned objects to synchronize, following a lazy wait mechanism [5].

The query-command distinction has the problem of eliminating bi-directional communication between objects. Let us consider the specification of a typical bounded buffer class, using a C++ like notation:

```
ConcurrentObject IntBuffer
  {  int buffer[DIM] ;
     int in, out, count ;
   public:
     IntBuffer() ;
     int head(); bool is_full(); bool is_empty(); int free_slots(); // Queries
     void put(int); void delete();                                  // Commands
   constraints:
     disable put when (count == DIM);
     disable delete, head when (count == 0); } ;
```

This class has an interface composed of four query operations and two command operations. In order to ensure a correct manipulation of objects it is necessary to own them before being used, and release them after. Thus, we can define a `hold` construct to allow the ownership of objects. For example, the effects of a `get` operation could be obtained as in Figure 1 (left).

```
IntBuffer buf ;
hold buf {
   item = buf.head() ;
   buf.delete() ;
} /* hold */
```

```
IntBuffer buf ;
hold buf {
   if (buf.free_slots > n)
      for (int i = 1; i++; i<n)
         buf.put(i) ;
} /* hold */
```

```
IntBuffer buf ;
hold buf {
   if (!buf.is_empty)
      sum += buf.head() ;
} /* hold */
```

Fig. 1. The `hold` Construct

When an object α_1 invokes a `hold` over another object α_2, it must be guaranteed that α_1 can access the most current state of α_2, and this state cannot be changed by another object while α_1 is executing the instructions inside the `hold`. An exception to this rule is given by the non-acceptance by α_2 of an operation issued by α_1 that is disabled by a synchronization constraint.

The `hold` construct does not need to specify the kind of ownership. A compiler could examine the instructions inside it and decide if the ownership is exclusive or non-exclusive depending whether all the invoked operations are queries or not. For example, the code in Figure 1 (center) establishes an exclusive ownership, while the code in Figure 1 (right) establishes a non-exclusive ownership.

3 Implementation and Performance

The current implementation has been coded in C++ and consists of a runtime system and a set of base classes that must be inherited by the classes of the distributed programs. It runs on SUN UltraSPARC workstations on top of two networks: a 10 Mb/sec ethernet and a 155 Mb/sec ATM network. We have used the Solaris 2 thread package, assigning a thread to each object. The protocol used for object replication is based on that of the Midway system [2].

In this section we present the performance of two parallel programs: matrix multiply, which allows us to measure the impact of replicating large size objects, and a branch and bound algorithm for the resolution of the Traveling Salesman Problem (TSP), which replicates an integer object.

MM is a simple matrix multiply program which multiplies two square matrices of integers. The parallel algorithm is based on dividing the result matrix in 4^N square submatrices, and computing them in parallel. We present the results of multiplying matrices with a size ranging from 400 to 1000.

ETH	2 Nodes	4 Nodes	8 Nodes	ATM	2 Nodes	4 Nodes	8 Nodes
400	1.59	2.57	2.90	**400**	1.63	3.09	4.57
600	1.72	3.05	4.03	**600**	1.77	3.44	6.10
800	1.85	3.34	4.71	**800**	1.88	3.68	6.68
1000	1.91	3.52	5.27	**1000**	1.94	3.79	7.12

Table 1. Speed-ups of parallel matrix multiply

BB is a parallel program that solves the TSP using a branch and bound algorithm. The performance of the BB program is given in Figure 2. The experiments have been carried out with a 100-city problem instance.

4 Conclusions and Future Work

We have presented a concurrent object-oriented programming model based on object replication. It is based on entry consistency, a weak consistency memory

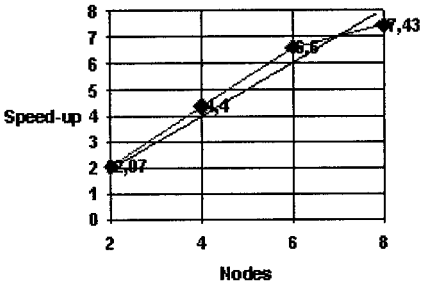

Fig. 2. Speed-up for a 100-city TSP Instance

model which allows an increase in performance but requires explicit annotations of programs. We conclude that the requirements of entry consistency can be held by a concurrent object-oriented programming model in an elegant way, by the establishment of ownership relations among objects, the distinction between commands and queries, and a lazy wait synchronization mechanism. Preliminary results, using matrix multiply and branch and bound algorithms, have been presented.

There are several lines of future work. The first one is to add an exception handling mechanism to the object model, to make it more usable. Another interesting aspect to be considered is enhancing the expressive power of the model by including the possibility of conditional acquirements, and to allow the ownership of several objects at one time.

References

1. Agha, G: "Actors: A Model of Concurrent Computation in Distributed Systems". The MIT Press. 1987.
2. Bershard, B. N., Zekauskas, M. J. "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors". Tech. Report CMU-CS-91-170. 1991.
3. Karamcheti, V., Chien, A.: "Concert-Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware". In Proceedings of Supercomputing'93, Portland, Oregon. November 1993.
4. Matsuoka, S., Taura, K., Yonezawa, A.: "Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages". OOPSLA'93.
5. Meyer, B: "Systematic Concurrent Object-Oriented Programming". Communications of the ACM, vol. 36, no. 9. September 1993.
6. Nitzberg, B., Lo, V.: "Distributed Shared Memory: A Survey of Issues and Algorithms". IEEE Computer, vol 24. August 1991.
7. Papathomas, M.: "Concurrency Issues in Object-Oriented Languages". Tech. Rep. Centre Universitaire Informatique. University of Geneva, D. Tsichritzis, Ed., 1989. 1992.