Object–Based Consistency in a Distributed Shared Memory*

Thilo Kielmann and Guido Wirtz
Dept. of Electrical Engineering and Computer Science,
Univ. of Siegen, Germany
{kielmann,guido}@informatik.uni-siegen.de

Abstract

Distributed shared memory (DSM) systems combine the advantages of shared memory multiprocessors and distributed systems. The value of DSM systems primarily depends on the expressiveness of their consistency model and on implementation efficiency which are contradicting goals. In this paper, we sketch existing DSM consistency models and develop a model which allows to efficiently deal with shared objects.

Keywords: Parallel Programming, Distributed Shared Memory, Object Orientation, Class Library

1 Introduction

Distributed shared memory (DSM) systems have attracted considerable research efforts recently, since they combine the ease of programming of shared–memory multiprocessors with the high availability and good price/performance ratio of distributed computing systems [3, 9]. Despite the fact that object orientation has been established as state–of–the–art software–engineering methodology in sequential programming, only few work so far deals with object–based DSM. In this paper, we will briefly survey existing consistency models for distributed shared memories and approaches for shared–object systems. Then we will introduce our approach of providing an object–based DSM in the form of a C++ class library and illustrate its use by two simple examples.

2 Existing Consistency Models

The work in [10] provides a comprehensive survey of existing consistency models and their impacts on ease–of–use and implementation efficiency. We will now briefly sketch the core concepts in order to motivate the needs of object–based DSM systems.

We can distinguish between two types of DSM consistency, depending on whether or not programmers have to explicitly program with synchronization variables. The most stringent consistency model is called *strict consistency* and can be defined by the following condition [10]: *Any read to a memory location x returns the value stored by the most recent write operation to x.* This model provides the perfect illusion of a shared memory. Unfortunately, this model can not be implemented because communication delays with messages exchanged between hosts and the lack of a global clock make it not feasible to decide which write operation might be "the most recent one" at any given time of a computation.

Fortunately, this strictness of consistency is usually not needed for concurrent applications. A widely used consistency model dates back to [7] and is called *seqential consistency.* It can be defined by the following condition [10]: *The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.* With this definition, any interleaving of concurrent operations is acceptable, but all processes must observe the same sequence of memory references. This style of consistency is widely used, e.g. in shared–memory programming with synchronization based on semaphores.

Implementations of sequential consistency and other flavours of consistency models without explicit use of synchronization variables have to cope with automatized updates of shared memory regions to all processors which might access the memory. Hence, this approach sacrifices implementation performance for programming ease. But because runtime efficiency is the most important goal of parallel computing, alternative consistency models have to be investigated.

In contrast to the models introduced so far, *en-*

*Proc. SIPAR Workshop on Parallel and Distributed Computing, Biel–Bienne, Switzerland, October 6, 1995

try consistency [2] and the related *release consistency* model require the programmer to explicitly use synchronization variables like e.g. locks. With these models, all accesses to shared memory have to be grouped into critical sections where programmers have to use *acquire* and *release* operations at the start and the end of each critical section, respectively. Entry consistency furthermore requires that every shared variable has to be associated with some synchronization variable. When an acquire is done on a synchronization variable, only those shared variables guarded by that synchronization variable are made consistent. With entry consistency, acquire operations can provide exclusive as well as non–exclusive access in order to provide more flexibility.

Because entry consistency performes memory updates only for those shared variables actually in use, it requires minimal communication overhead. The drawback of this approach is that memory accesses have to be enclosed by accesses to synchronization variables which introduces additional programming efforts.

3 Existing Shared–Object Models

In classical (sequential) object–based programming [8], objects are instances of abstract data types. Essentially, objects contain an internal state which is protected by and can only be accessed via the interface routines (called methods or operations) defined by the specification of their corresponding type. One representative of systems providing objects shared by distributed memory computers is Objective Linda [6]. Here, the Linda model [4] is based on objects as instances of abstract data types. Hence, Objective Linda provides shared memory units called *object spaces* in which objects can be stored and later retrieved using predicates defined by their interface rather than by memory addresses. This kind of associative object store is a powerful model for coordinating concurrent processes, but it is not really a DSM model.

Another well-known system providing shared data objects is the Orca programming language [1]. Here, shared data is encapsulated in passive objects, and all data inside objects may only be accessed by a set of indivisible operations defined by the object's type. So, the interface operations execute under mutual exclusion. Furthermore, these operations can be protected by a so–called *guard* which blocks execution until a certain predicate on the objects becomes true. This way, object consistency is completely under control of the Orca runtime system. Mutual exclusive execution allows programmers to implement problems using *mutual exclusion synchronization* where only one process at a time can be active in a critical section. Guards enable *condition synchronization* where one process blocks until another one unblocks it. Both kinds of synchronization are essential for concurrent programming and can be found e.g. in the classical bounded–buffer problem in which only one process at a time is allowed to modify the buffer.Additionally, consumers have to wait until producers made the buffer non–empty and producers have to wait until consumers made the buffer non–full.

4 Providing Object–Based Consistency in a DSM

In order to raise acceptance by programmers of concurrent systems, models and tools should be closely related to already existing programming languages and systems. For this reason, we prefer not to invent a new programming language (like Orca) or to propose a completely new coordination model like Linda. Instead, we prefer the approach of providing a distributed shared memory abstraction in form of a class library for C++ which reduces the amount of new concepts to be introduced in order to write concurrent programs.

As outlined above, a consistency model has to provide mechanisms for mutual exclusion synchronization as well as for condition synchronization. The entry consistency model introduced above requires minimal communication overhead and enables mutual exclusion synchronization by introducing critical sections. Therefore, we base our model on entry consistency. As objects are obviously well-suited units of protection, our model automatically associates a lock variable with each object to be placed in the DSM. More specifically, we introduce a common superclass called *DSM_object* for all classes of objects which shall be placed in DSM. So, the lock variable becomes an integral part of every DSM object. Processes willing to access DSM objects have to execute acquire and release operations on the objects before and after the body of an interface routine. Fortunately, acquire and release can be made part of the implementation of the DSM objects themselves such that client objects can completely abstract from consistency and synchronization issues. Our model allows for *write_acquire* and *read_acquire* operations which grant exclusive or

non-exclusive object access, respectively.

Mutual exclusion synchronization, however is not sufficient for suitably implementing condition synchronization problems. The only way of realizing condition synchronization under this assumption are busy–waiting schemes in which processes periodically acquire, test, and release shared objects. But this kind of implementation is far from being efficient at runtime. Hence, mechanisms for blocked waiting on conditions are necessary.

The introduction of additional synchronization objects like semaphores or condition variables which are independent from the object locking mechanism neither provide a way to implement condition synchronization. This can easily be seen because processes waiting for a certain condition to hold true on an object have to release their lock before they are blocked because otherwise no other process ever could modify the object in order to fulfill the condition. The process of releasing the object lock, blocking wait, and re-acquiring the lock cannot be performed atomically with independent synchronization objects. Hence, race conditions can violate correct behaviour of a program.

Alternatively, shared objects with mutually exclusively executing operations resemble the classical monitor [5] concept known from operating systems. Monitors provide condition variables which are directly combined with the mutual exclusion mechanism of routines. Hence, by additionally providing monitor objects, we are able to safely implement condition synchronization problems. Furthermore, monitor objects are more flexible than Orca objects because condition variables can block routine execution at any time and not only at the beginning. It can be shown that the implementation of monitor objects is simply an extension of *DSM_objects* which removes non–exclusive locks and instead realizes synchronization with condition variables which atomically handle blocking wait and passing over of the object locks.

To summarize, our DSM system provides the following two classes from which application objects can be derived by the inheritance mechanism of C++.

1. *DSM_object* supporting the operations:
   - *write_acquire:* locks an object exclusively and makes it available on the processor executing the operation; waits until the lock can be obtained w.r.t. other locking operations on the object.
   - *read_acquire:* locks an object non–exclusively and makes it available on the processor executing the operation; waits until the lock can be obtained w.r.t. other locking operations on the object. Non–exclusively locked objects can not be modified.
   - *release:* releases the last acquire operation on the object.
2. *Monitor* supporting the operations:
   - *lock:* locks the monitor object exclusively and makes it available on the processor executing the operation; waits until the lock can be obtained w.r.t. other locking operations on the object.
   - *unlock:* releases the last lock operation on the monitor object.
   - *signal(cond_var):* wakes up one process waiting at the specified condition variable of this monitor if such a process exists. The executing process must already have locked the monitor. The executing process is blocked, the lock is passed to the awakened process and returned to the executing process on its resumption.
   - *wait(cond_var):* blocks the executing process until another one performes a *signal* operation on the same condition variable and this process will be selected for resumption. The executing process must already have locked the monitor. The lock is passed to the next process becoming active w.r.t. this monitor object and retained to the executing process on its resumption.

5 Two Examples

We will now illustrate our model by two examples which show how concurrent programs can be implemented with our DSM class library. The first example the implementation of which we sketch is the traditional reader/writer problem which uses the standard *DSM_object* class.

```
class rw : public DSM_object {
public:
itemtype do_read {
  itemtype result;
  read_acquire();
  // result = ...
  release();
}
void do_write (itemtype i) {
  write_acquire();
```

```
  // store i ...
  release();
}
}
```

The second example sketches the implementation of a bounded buffer based on a monitor object.

```
class bb : public monitor {
private:
  condition nonempty,nonfull;
public:
void put (itemtype i) {
  lock();
  if is_full() wait(nonfull);
  // store i ...
  signal(nonempty);
  unlock();
}
itemtype get () {
  itemtype result;
  lock();
  if is_empty() wait(nonempty);
  // result = ...
  signal(nonfull);
  unlock();
}
}
```

6 Conclusion

In this paper, we showed that our approach allows the realization of a comfortable, object–based distributed shared memory system without the introduction of a new programming language, solely implemented as a class library for the C++ programming language. An implementation of the library is currently under way.

References

[1] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A Language for Distributed Programming. *SIGPLAN Notices*, 25(5):17–24, 1990.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. IEEE COMPCON Conference*, pages 528–537. IEEE, 1993.

[3] Bernd Freisleben and Thilo Kielmann. Approaches to Support Parallel Programming on Workstation Clusters: A Survey. Informatik-Bericht Nr. 95-01, University of Siegen, Dept. of Electrical Engineering and Computer Science, Siegen, Germany, 1995.

[4] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[5] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549 – 557, 1974.

[6] Thilo Kielmann. Object–Oriented Distributed Programming with Objective Linda. In *Proc. First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russia, 1995.

[7] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[8] Bertrand Meyer. *Object–oriented Software Construction*. Prentice Hall, New York, 1988.

[9] Jelica Protić, Milo Tomašević, and Veljko Milutinović. A Survey of Distributed Shared Memory Systems. In *Proc. 28th Hawaii International Conference on System Sciences*, pages 74–84, Maui, HA, 1995.

[10] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.