Distributed Shared Object Memory

Paulo Guedes Miguel Castro

INESC
R. Alves Redol 9, 1000 Lisboa, Portugal
{pjg,miguel}@inesc.inesc.pt

1 Introduction

This paper describes the goals, programming model and design of DiSOM, a software-based distributed shared memory system for a multicomputer composed of heterogeneous nodes connected by a high-speed network. A typical configuration is a cluster of tens of high-performance workstations and shared-memory multiprocessors of two or three different architectures, each with a processing power of a few hundred MIPS and several hundred kilobytes of memory, and connected by a high-speed interconnect such as ATM.

Programs in DiSOM are written using a shared-memory multiprocessor model where synchronization objects are associated with data items. Programs use a threads library to start new threads, possibly at a specified node, and to synchronize between concurrent threads. Programs must call the synchronization primitives explicitly, as they would in a shared-memory multiprocessor. The system traps these calls and uses the information to drive both distributed synchronization and the memory coherence protocol. DiSOM uses the *entry consistency* [Bershad 91] memory model to ensure coherence. This model guarantees memory consistency, as long as an access to a data item is enclosed between an acquire and a release on the synchronization object associated with the data item. Stronger consistency models, such as release consistency and processor consistency, may also be supported.

Page-based distributed shared memory systems use the virtual memory page as the unit of sharing and coherence, but the granularity of logical objects does not match the page size. Hence, false sharing occurs when several independently shared data items are placed in the same page. These systems use page invalidation techniques to detect and classify accesses to shared data items, which avoids the need for compiler support to do it. On the other hand, page faults can introduce a significant overhead. In DiSOM, the granularity of sharing and coherence is defined by arbitrarily sized language-level data items called objects. Since the system's unit of sharing matches the logical data partition, the programmer can decompose the application data in independently shared units, virtually eliminating false sharing. Accesses to a shared data item are detected by the calls to the synchronization primitive associated with the data item, thus avoiding more complex mechanisms for their detection, at the expense of a more complex programming model.

One of our goals is to allow programmers to customize the actions taken when the synchronization object associated with a data item is acquired or released. The default

mechanism in DiSOM just updates the bits of the data item ensuring the translations needed due to heterogeneity and without interpretation. This default procedure can be overwritten by the programmer who can specify which other objects should also be transferred and made consistent. This can be used to perform prefetching and to express situations where the memory representation of the object is not contiguous, as is the case of complex objects which are in fact represented as a graph of sub-objects.

Another of our goals is to support heterogeneity. Memory can be shared between machines with different data representations, different alignments of data types and different compilers. We achieve this because our data items are typed memory regions, as opposed to raw strings of bits, thus allowing the system to convert them between machines.

A major aspect of DiSOM is the definition of the relationship between objects and synchronization in a natural and non-intrusive way so that it is acceptable to programmers and can be easily introduced in existing programs. We associate with each object a vector of routines that contain the implementation of the synchronization primitives and routines to pack and unpack the object's state to and from messages. These routines may be redefined on a per-object basis allowing the use of application specific knowledge to prefetch data or relax the memory model even further. In our C++ prototype this is easily achieved with inheritance.

2 Related work

Much of the previous work in distributed shared memory has focused on the definition of weakly-consistent protocols for memory coherence and their implementation both in hardware and software. The Dash multiprocessor [Lenoski 90] implemented release consistency in hardware. Munin [Bennett 90] provided a software implementation of release consistency and supported different memory consistency protocols for different categories of shared data objects. Memo [Keleher 92] introduced lazy release consistency. Midway [Bershad 91] introduced the notion of entry consistency. In Midway, programs are annotated with a set of keywords to specify which variables are shared and use function calls to associate them with at least one synchronization object.

We started from the same consistency model as Midway but adopted a different programming model. We allow the programmer to redefine class-specific actions to be taken when locks are acquired and released. This allows the programmer to piggyback semantic information on the synchronization messages, thus reducing the total number of messages. Unlike Midway, DiSOM has type information associated with each object and uses this information to cope with heterogeneity. Another difference is that we do not require a single program image across all machines. Our implementation is also totally independent from any other system.

Several distributed object systems supported a single object address space and object sharing in a distributed system [Jul 88,Marques 89], which is a form of distributed object memory. In these systems, object sharing was mainly supported by a function-shipping policy based on remote procedure calls. Additionally, they provided support for limited forms of data-shipping, allowing the programmer to explicitly move objects and replicate read-only objects. Clouds [Ramachandran 89] uses distributed shared memory techniques to ensure the coherence of the memory segments that compose an object. Contrary to our language defined objects, these segments are coarse grained, typically larger than a page.

Orca [Bal 89] provides a distributed implementation of the shared data-object model

where shared variables are accessed through mutually exclusive operations on the objects that encapsulate the data. We also encapsulate data items inside objects but we use a more conventional programming model where operations are not guaranteed to be atomic, the programmer has to explicitly use synchronization primitives to guarantee program correctness.

3 Object based entry consistency

A memory coherence protocol is best described as a contract between the system and the application program. If a program satisfies the contract's requirements the system guarantees that the program views a sequentially consistent memory [Lamport 79]. DiSOM uses the *entry consistency* memory coherence protocol introduced by Midway [Bershad 91]. The contract imposed by entry consistency on the programmer forces each shared data item to have an associated synchronization variable and all accesses to the shared data item to be enclosed inside acquire/release operations on that synchronization variable.

In Midway, all shared data must be explicitly labeled as shared and must be explicitly associated with at least one synchronization variable. The association between synchronization variables and shared data items is a many-to-many binary association. The association is dynamic and may change during the program's execution.

In DiSOM the programming model is slightly different. All instances of the special class `sharedObject` or classes derived from `sharedObject` are potentially shared and have a lock attached to them. In fact lazy evaluation techniques are used so that only objects on which synchronization primitives are called need to have an associated lock.

The `sharedObject` class defines the basic Concurrent Read Exclusive Write (CREW) lock. The lock is accessed through acquire and release operations defined in the object's class. In order to update the state of an object a thread must acquire the object's lock for writing and then release it to allow others to observe the update. As an example consider the increment of a shared counter variable, written in C++:

```
class sharedObject {
public:
   virtual void acqRead(void);
   virtual void relRead(void);
   virtual void acqWrite(void);
   virtual void relWrite(void);
};
class Counter : public sharedObject {
    int cnt;
public:
    Counter(void) { cnt = 0; }
    void inc(void) { cnt++; }
    int value(void) { return cnt; }
};

// Declaration of shared variable count;
Counter count;

// Incrementing the shared counter
count.acqWrite();
count.inc();
```

```
count.relWrite();

// Reading the shared counter
count.acqRead();
int value = count.value();
count.relRead();
```

The inspection of the state of a shared object requires a thread to acquire the object's associated lock for reading or writing. This ensures that the most recent update is observed. If the operation on the object's state is read-only the lock should be acquired for reading to allow maximum efficiency and concurrency.

DiSOM does not impose the use of external synchronization, internal synchronization may also be used. The synchronization primitives may be called inside the class methods around the sections of code where the instance data is accessed.

```
class Counter : public sharedObject {
    int cnt;
public:
    Counter(void) { cnt = 0; }
    void inc(void) { acqWrite(); cnt++; relWrite(); }
    int value(void) { acqRead(); int v = cnt; relRead(); return v; }
};
// Declaration of shared variable count;
Counter count;

// Incrementing the shared counter
count.inc();

// Reading the shared counter
int value = count.value();
```

A thread can inspect a given object without acquiring its associated lock, but there is no guarantee that the inspected state will correspond to the last value written. This is useful mostly for read-only global data that is replicated at thread creation time and does not change afterwards. Some algorithms may also tolerate limited incoherence, provided the application program synchronizes frequently enough to bound the degree of incoherence.

Class `Counter` in the example above could also redefine the implementation of the synchronization primitives that acquire and release the locks, or could define new synchronization routines that internally call the more basic primitives. This provides ample flexibility to the programmer to write the programs in a simple and intuitive way and yet have access to the low-level mechanisms to improve performance.

DiSOM also supports the locking of parts of an object. For example, our `matrix` class supports locking of individual rows. This is achieved by associating a lock object with the virtual memory that composes the row (or rows) of the matrix. When that lock is acquired the memory associated with it (i.e. the row of the matrix) is made coherent. The representation of the object in virtual memory is separated from the sharing patterns of its instance variables. Objects (matrices in this case) continue to be represented in memory as contiguous arrays and still benefit from the advantages of being described as objects.

4 Implementation

DiSOM is composed of a set of communicating threads. Several threads can exist in an address space (*context*) and several contexts can exist in any of the computing nodes. Each context may have its own program image, but all program images are linked to the DiSOM library. This library is implemented as a C++ class library on top of UNIX. The class hierarchy is designed for extensibility and flexibility, allowing the mix and match of system classes to define new object sharing policies. This section provides a high level overview of DiSOM's implementation.

4.1 A single shared object address space

In DiSOM there is not a single shared virtual memory address space spanning all contexts. Instead a form of pointer swizzling [Wilson 92] is used to offer the abstraction of a single shared object address space. Objects are uniquely identified by a *global identifier* which is composed of a *context identifier* and an *object identifier*. The context identifier uniquely identifies the context where the object was created (called the *home context*) and is implemented by concatenating the address of the machine where the home context executes with the port number used to communicate with the context. The object identifier is the virtual address of the object in the home context.

Programs always manipulate objects using language level references (virtual memory pointers in C++). When an object is created its only identifier is the language level reference. When a reference is exported for the first time, a global identifier is associated with the object using an *object dictionary*. The language level reference is swizzled into the global identifier whenever the object reference is exported.

A context importing a given reference for the first time allocates space for the object and swizzles the global identifier into the pointer to the allocated copy. The pointer and the global identifier are inserted in the object dictionary. The dictionary is used to swizzle the global identifier into the object pointer whenever the same reference is reimported or exported.

A context imports an object's reference from a remote context in the following situations: it receives the object as a parameter in the remote thread creation primitive; the object is global; a reference to it was obtained from the name server; or the object's reference was part of an accessible object's instance data. In all these situations the reference export and import procedures are applied. In the special case of global data a valid copy of the object's state is transferred when a remote thread is created.

Our solution significantly differs from [Zhou 92] where the shared virtual address space is similar in all contexts, although it may start at a different virtual address at each context. Our virtual address spaces are totally different from context to context and depend only on the objects that are lazily mapped on each context. Hence, a shared object only uses virtual memory space in the contexts that import its reference and there is no need to coordinate contexts in order to obtain a uniform virtual memory address space.

We also support the notion of domains – sets of homogeneous machines sharing a single virtual memory address space. When transferring object references between machines in the same domain pointer swizzling is turned off. Domains are mainly handled by the system administrator who groups machines to form domains.

4.2 Memory coherence protocol

In a multicomputer system based on a cluster of workstations it is important to minimize the number of messages used by the memory coherence protocol. The entry consistency memory coherence protocol lends itself to this kind of optimization. When implemented with an update based policy the memory coherence protocol messages can be piggybacked on the CREW lock algorithm's messages (described in the next section). An update to the state of an object is transferred to a context only when one of the threads running in that context calls `acqWrite` or `acqRead` following a call of `relWrite` by a thread in another context. Thus the total number of messages generated in the system is only determined by the number of synchronization interactions between contexts.

The object state transfer relies on two methods defined by each class to determine what to transfer and how to transfer. These methods, *pack* and *unpack*, receive a buffer and pack or unpack the object state onto and from the buffer. If these methods are left undefined the compiler generates a default implementation which transfers the value of the object's instance data[1]. The programmer can also define these methods to inhibit or modify the transfer of part of the instance variables.

4.3 Concurrent Read Exclusive Write locks

DiSOM's entry consistency is closely associated with the distributed CREW locks. Our CREW lock algorithm is a variation of Li's dynamic distributed manager with distributed copy sets [Li 86]. It implements CREW lock semantics in the synchronization of multiple threads existing in multiple contexts in the distributed system. We support the execution of several threads in each context. This allows us to explore parallelism inside a context when the machines support it (e.g. shared memory multiprocessors) and to overlap computation and communication.

The algorithm is based on two types of tokens, the *write token* and the *read token*, which are associated with each lock. The system oscillates between two states ensuring CREW semantics – either one context holds the write token exclusively and no read token exists or several contexts hold read tokens concurrently and no write token exists. There is also the notion of lock *owner*. The owner is either the context holding the write token or the last context to hold a write token. Ownership is dynamic, to keep track of the owner we use a forwarding pointer scheme [Fowler 85]. The forwarding pointer identifies the context believed to own the lock. A write token can only be obtained from the owner context but a read token can be obtained from any context holding a read token.

A context must hold the write token for an `acqWrite` performed by any thread executing in the context to complete. An `acqRead` will complete if the context holds either the read or the write token. When a thread calls `acqWrite` or `acqRead` and the enclosing context does not hold the needed token, a message is sent requesting the token and the thread blocks waiting for the reply. The message is sent along the forwarding pointer chain. The forwarding stops when the message reaches the owner, a context holding an appropriate type of token or a context which is waiting for any type of token for the same lock. In the later case, the algorithm proceeds as before after the local request is satisfied.

When the lock owner receives a request for a write token, it waits until the lock is released by local threads and then sends a message transferring ownership, the copy set

[1] Currently we generate these methods by hand.

and the object state to the requester. Note that if the owner holds a write token the copy set is empty. The owner then sets the forwarding pointer to point to the requester and the requester sets the forwarding pointer to himself. The requester will then send messages to all the contexts in the copy set, invalidating their read tokens, and wait for the replies. Each context which receives an invalidate message will do the same, thus invalidation of read tokens proceeds in a distributed divide and conquer fashion. When a context receives an invalidate message, it receives the identifier of the new owner and sets the forwarding pointer accordingly. The new owner then becomes the holder of the write token. The new owner can reacquire the lock repeatedly without interacting with other contexts as long as it holds the write token.

When a read token holder receives a request for a read token the requesting context is inserted in the *copy set* and a reply is sent to it. The reply includes the read token, the object state and the forwarding pointer. Thus all readers have their forwarding pointers set correctly. If the owner receives a read token request and it holds a write token, it proceeds as above but first converts its write token into a read token. The requesting context can reacquire the lock for reading without interacting with other contexts as long as it holds the read token.

The forwarding pointer is also updated when a context forwards a token request message – the forwarding pointer is set to the identifier of the requester. This is used to reduce the length of the forwarding pointer chain.

5 Summary

We presented the goals, programming model and design of DiSOM, a software-based distributed shared object memory system. It is designed to run on a multicomputer composed of heterogeneous nodes connected by a high-speed network. One of our goals is to explore the redefinition by the programmer of the actions to be taken when locks are acquired and released in an entry-consistent shared memory. This provides ample flexibility for optimization while keeping the programming model simple and easily understandable. Another goal is to support heterogeneity by having a type description of the shared data items.

DiSOM is currently running on a network of SPARCStations running SunOS. A port to i386 PCs running Mach3.0 and the BSD Single Server is almost complete. Parallelized versions of matrix multiplication and Successive Over Relaxation (SOR) are being used to test the system and obtain perform measurements. Results based on a non-optimized early version of the system seem to confirm our expectations. We hope to provide soon meaningful performance figures for a few test programs.

References

[Bal 89] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A Distributed Implementation of the Shared Data-Object Model. In *Proceedings of the First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, Ft. Lauderdale FL (USA), Oct 1989. Usenix.

[Bennett 90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In

Proceedings of the 17th Annual Symposium on Computer Architecture, pages 125–134, May 1990.

[Bershad 91] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[Fowler 85] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, Department of Computer Science, University of Washington, December 1985. Available as Technical Report 85-12-1.

[Jul 88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Keleher 92] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. 19th Int. Symposium on Comp. Architecture*, pages 13–21, Gold Coast (Australia), May 1992.

[Lamport 79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[Lenoski 90] D. Lenoski, J. Laudon, K. Gharachorloo, A Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–160, New York, May 1990. IEEE.

[Li 86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 229–239, August 1986.

[Marques 89] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA 89*, New Orleans, 2-6th October 1989.

[Ramachandran 89] Umakishore Ramachandran and M. Yousef A. Khalidi. An Implementation of Distributed Shared Memory. In *Workshop on Experience with Building Distributed and Multiprocessor Systems*, Fort Lauderdale, FL, October 5-6 1989.

[Wilson 92] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[Zhou 92] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heretogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5), September 1992.