Virtual memory support for distributed computing environments using a shared data object model

To cite this article: F Huang et al 1995 Distrib. Syst. Engng. 2 202

View the article online for updates and enhancements.
Virtual memory support for distributed computing environments using a shared data object model*

Feng Huang†, Jean Bacon‡ and Glenford Mapp§

† Division of Computer Science, University of St Andrews, St Andrews KY16 9SS, UK
(E-mail address: fh@dcs.st-and.ac.uk)
‡ Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, UK
(E-mail address: jmb@cl.cam.ac.uk)
§ Olivetti Research Laboratory, 24a Trumpington Street, Cambridge CB2 1QA, UK
(E-mail address: gem@cam-orl.co.uk)

Received 17 July 1995

Abstract. Conventional storage management systems provide one interface for accessing memory segments and another for accessing secondary storage objects. This hinders application programming and affects overall system performance due to mandatory data copying and user/kernel boundary crossings, which in the microkernel case may involve context switches. Memory-mapping techniques may be used to provide programmers with a unified view of the storage system. This paper extends such techniques to support a shared data object model for distributed computing environments in which good support for coherence and synchronization is essential. The approach is based on a microkernel, typed memory objects, and integrated coherence control. A microkernel architecture is used to support multiple coherence protocols and the addition of new protocols. Memory objects are typed and applications can choose the most suitable protocols for different types of object to avoid protocol mismatch. Low-level coherence control is integrated with high-level concurrency control so that the number of messages required to maintain memory coherence is reduced and system-wide synchronization is realized without severely impacting the system performance. These features together contribute a novel approach to the support for flexible coherence under application control.

1. Introduction

The conventional approach to main memory and storage management is based on a two-level store architecture, which provides one interface defined by programming languages to access memory segments and another defined by file systems to access persistent data residing in secondary storage. The mapping between the two types of data is usually done in part by the file system and in part by explicit user translation code. The explicit user translation code has to be written and included in each program. This includes the code concerned with the explicit movement of data between main and backing store and the code required to change the representation of the data for long term preservation and restoration. The quality and productivity of software development is impaired by the mapping [5]. Moreover, operating system performance is compromised because of mandatory data copying and unnecessary user/kernel boundary crossings, which in the microkernel case may involve context switches [11, 26]. If a database system is implemented on top of this architecture, double paging [27] may cause resources to be used inefficiently and may lead to the double paging anomaly [14], where a significant increase in the number of page faults occurs with an increase in buffer pool size without a corresponding increase in physical memory.

Memory mapping [26] provides a uniform view of both volatile and persistent data. It also provides better performance for accessing persistent data because mandatory information copying and unnecessary user/kernel boundary crossings (or context switches in microkernels) are avoided. Double paging is also eliminated. Memory-mapped file abstractions have been provided by many single machine operating systems, such as Unix, Mach and Windows NT. In distributed environments, objects may be mapped simul-
stantaneously on multiple networked sites. This presents challenges in the area of coherence and concurrency control. Many protocols have been developed to tackle the system-wide cache coherence problem both in distributed shared memory systems (DSM) and in distributed file systems with client caching. These protocols can be used for memory-mapped object systems but a single, system-provided protocol cannot be expected to perform well for all applications. For example, optimistic concurrency control has minimal overheads but it is only suitable for systems where contention is unlikely [6]. Pessimistic concurrency control, on the other hand, involves higher overheads but it is more suitable for systems where a high contention rate is likely. Even in pessimistic concurrency control, different coherence protocols are suitable for different applications. Two kinds of commonly used protocols, namely write-invalidate and write-update, are considered here. Suppose that the page is the unit of coherence control. The list of nodes that have a valid cached copy of an object page for read is called the copy set of the page. The node that has the most recent write access to an object page hence has the up-to-date page contents is called the owner of the page. In write-invalidate protocols, when a process attempts to write to an object page, it has to instruct the copy set of the page to invalidate their caches before it can go ahead. Any subsequent read accesses from nodes other than the owner will cause a copy of the page to be shipped from the owner. In the write-update protocols, each modification to an object page is written through to the copy set. In the case that the copy set is relatively stable and the page is accessed many times by each node, the write-update protocols will perform better. For example, if the size of the copy set is $n$, for each write access, $n$ RPC calls are required in the write-update protocols while $2n$ RPC calls are needed in the write-invalidate protocols if the page is accessed again by all the nodes in the copy set after the updating. On the other hand, if the copy set changes dynamically, the write-invalidate protocols are expected to perform better since there is no need to ship data to those nodes that are in the copy set but do not use the data in the near future. This leads to a conclusion that support for multiple coherence protocols and adding new protocols is desirable in a general distributed environment.

This paper examines the issues involved in supporting a shared data object model for distributed computing environments. It proposes a coherent, memory-mapped object system (COMMOS) [16] architecture to integrate main memory with secondary memory and local memory with remote memory. The primary aim of COMMOS is to provide a shared virtual address space as a global abstraction nor to provide an object database management system although both can be implemented as value-adding services of COMMOS. COMMOS is to provide a general flexible architectural framework for a shared data object model, which is able to accommodate multiple coherence protocols to meet different applications' requirements and to easily add new coherence protocols to extend the system.

The approach is based on a microkernel [1, 13, 20], typed memory objects, and integrated coherence control. The major advantages of microkernels are openness and the ability to enforce modularity behind memory protection boundaries. It is easy to implement, install and debug new services, since adding or changing a service does not mean stopping the system and rebooting with a new kernel. The COMMOS architecture distinguishes itself from other microkernel-based systems by clearly separating the coherence server from the external pager [1, 13, 20] and providing typed memory objects. This separation makes it easier to provide multiple coherence protocols and to add new protocols. The support for typed memory objects allows different coherence protocols and paging algorithms to be applied to different types of memory object. Based on these approaches, applications can advise the system of the most suitable coherence protocols and paging algorithms to be used to avoid protocol mismatch. Also, low-level coherence control is integrated with high-level concurrency control. This reduces the number of messages required to maintain object coherence and realizes system-wide synchronization without a centralized concurrency control mechanism.

The rest of this paper is organized as follows. Section 2 briefly reviews related research and compares the major issues addressed with those of COMMOS. Section 3 describes the COMMOS architecture, section 4 discusses the design issues, section 5 presents a prototype implementation and section 6 reports the performance measurements. Section 7 describes a prototype persistent C++ class library as a value-adding client.

2. Related work

Apollo Domain [22] was one of the earliest systems to assure coherence of shared memory-mapped objects in a local area network of workstations. It had the same goal as COMMOS to provide a coherent integrated virtual memory in distributed systems and had the notion of type but it did not make use of this property to provide clients with the flexibility to choose the way objects are managed.

 Mach [13], Chorus [1] and the V system [15] are microkernel-based systems. Lightweight processes and efficient message passing are supported and memory-mapped secondary storage objects are managed by external pagers. V++ takes a step further to allow applications to control the usage of physical memory. Coherence is achieved through the construction of new external pagers in user space. Each external pager usually supports only one coherence protocol. COMMOS borrows many ideas from these systems, especially in the construction of the low-level memory management layer. However COMMOS distinguishes itself by clearly separating the coherence server from the external pager. The separation makes it easier to support multiple coherence protocols and the addition of new protocols. The provision of typed memory objects allows clients to choose the most suitable protocols for different types of object.

Clouds [4], Choices [28] and Spring [20] are organized using an object-oriented approach. All components of the operating system are treated as objects. In Clouds and Spring, there is no explicit interprocess communication mechanism and all communications are carried out via
object invocations. Similar to other microkernel-based systems, coherence control is implemented by external pagers and only one protocol is provided. Again COMMOS is different from these systems in that it supports multiple coherence protocols and the addition of new protocols. Further research in Clouds [3] proposes to support multiple coherence protocols in a runtime library and applications can choose whatever protocols they require by explicitly invoking library calls to maintain memory coherence. In COMMOS, clients need not be concerned with explicit invocation of coherence control operations.

Some DSM systems have attempted to tackle the problem of coherence protocol mismatch. Munix [8] supports multiple coherence protocols. Each shared variable declaration is annotated by a predefined access pattern and the system chooses a coherence protocol for it. Quarks [9] supports multiple coherence protocols on a per-page basis and is implemented as a runtime library. COMMOS is different because it is an architecture framework to support shared data objects and it provides applications with the freedom to choose different coherence protocols and system developers with the flexibility to add new protocols. Also, low-level coherence control is integrated with high-level concurrency control in COMMOS so that the number of messages required to maintain memory coherence is reduced and no centralized locking mechanism is needed.

Object database management systems (ODBMS) [21, 7, 18, 10] also provide support for persistent shared objects. Although ODBMS can be provided as a value-adding service for COMMOS, they are mainly concerned with higher level issues such as database management, language-level objects, multiple-language support, and friendly application interfaces. COMMOS is concerned with the operating system support level.

3. COMMOS architecture

3.1. Objects and types

Objects in COMMOS are defined as logical entities that can be mapped into contiguous regions of a virtual address space. They can be shared by multiple processes on different network nodes. When an object is mapped, it can be read or written by simply reading or writing an address location within the address space corresponding to the offset of the byte in the object. The use of the term 'object' here does not imply any sophisticated concept used in object-oriented programming languages. It is solely an array of uninterpreted bytes, or more precisely, an array of pages, possibly associated with some backing secondary storage, which can be used to contain and enclose language-level objects. Class code can be bound to it by an object-oriented programming language runtime library.

Each object is an instance of a type. Here type means different logical data abstractions that can be managed in different ways, such as executable code, data, stack, file and other persistent object types. This provides the flexibility for implementing a number of aspects of virtual memory management and persistent object management, such as paging algorithms and object coherence protocols.

3.2. Integrated coherence control

One way to ensure object coherence in distributed systems is to allow multiple readers to read a specific object fragment at the same time but writers must have exclusive access [6]. This can be assured by a pessimistic or optimistic concurrency control mechanism. Pessimistic concurrency control with a locking interface is employed in the current implementation of COMMOS. However, optimistic concurrency control, such as that used in the Warp coherence mechanism [2], is also possible. In either case, high-level concurrency control is integrated with low-level coherence control so that the number of messages required to maintain memory coherence is reduced and system-wide synchronization is achieved without a centralized concurrency control mechanism.

3.3. COMMOS layers

Memory management in COMMOS is divided into two layers: the basic virtual memory management (VMM) layer and the object management (OM) layer (see figure 1).

Above these layers, there may be other value-adding clients, such as persistent programming languages, database management systems, and persistent shared virtual memory.

The VMM layer manages the local memory and is further divided into the machine-dependent and machine-independent parts. The machine-dependent part is concerned with managing the memory management unit (MMU) hardware and catches all page faults. The machine-independent part is concerned with managing address maps, satisfying the page faults for zero-filled objects and persistent programming languages, database management systems, and persistent shared virtual memory.

Objects are named by globally unique and location independent names that can be used to refer to objects from anywhere in the distributed system. A name can be a fixed-length bit pattern identifier or a path name such as that used in NFS. The access control list (ACL) method has been chosen to protect objects from unauthorized client accesses.

Object Management Layer

Database Management System

Persistent Shared Virtual Memory

Figure 1. The COMMOS Layers.

† An object fragment is part of an object, which may be mapped into a higher-level logical data structure, such as a language-level object or a record.
4. Design issues

4.1. Granularity
The granularity of memory, at which coherence is maintained, is an important design issue. The larger the granularity, the greater the contention will be. To reduce memory contention, a small granularity is desirable. In a typical network environment, due to the overhead of the software protocols, the transmission of large packets that may contain thousands of bytes is not much more expensive than the transmission of small ones. Therefore, large granularity is expected to improve the network throughput. In the emerging ATM network environments, network throughput improves while the packet size increases to a certain point [12, 19]. Beyond that point (8K Bytes for IP in the environment used in [12]), the throughput drops because segmentation and reassembly takes more time. Meanwhile, a page represents the smallest memory unit on which protection can be enforced by the memory management hardware and existing page fault schemes can be used. A page-based approach therefore seems natural and is adopted in COMMOS. Locks to variable length granularity are supported in the public interface but they are translated into the underlying page-based mechanism. Some researchers have started exploring memory management techniques to support fine-grained page size and to accommodate mixed page sizes in virtual memory [24]. When these techniques become available, hardware support can be used for variable length granularity.

4.2. Remote interprocess communication
When objects are mapped on multiple network nodes, one-to-many communication is needed to maintain object coherence. For example, one-to-many communication is needed for invalidating the copy set in write-invalidate protocols and for propagating updates in write-update protocols. Using the traditional one-to-one RPC to carry out one-to-many communication would introduce significant delay. This problem is exacerbated by the fact that an RPC to a dead or unreachable computer must time out before the connection is declared broken and the next computer is tried. A simple broadcast is not advisable because all nodes in the system have to process each broadcast request. Multicast is useful since it involves only the interested nodes. However, raw multicast is difficult to use and not all network interfaces or drivers support multicast.

MultiRPC [29] retains one-to-one RPC semantics while overlapping the computation and communication overheads at each of the destinations by sending a single request to multiple servers and awaiting their individual responses. In a system without multicast support, the actual transmission is done sequentially. However, the resultant concurrent processing by the servers results in a significant increase in efficiency over a sequence of standard RPC calls. The client is responsible for supplying a handler routine for any server operation that is used in a MultiRPC call. This handler is called by the runtime as each individual response arrives; it is used both for providing individual server return codes to the client and for giving the client control.
over the continuation or termination of the MultiRPC call. MultiRPC guarantees source ordering.

In the absence of network and machine crashes, it guarantees that a request (or response) will be processed exactly once; otherwise, it guarantees that it will be processed at most once. Both communication failure and site failure are detected although they are not distinguishable for the caller. MultiRPC can make use of multicast if the underlying system has such support. This mechanism is adopted as the remote interprocess communication tool in COMMOS.

4.3. Deadlock and prevention

The programming interface provides users with facilities to acquire and release locks for object fragments. A parameter is used to indicate whether the caller is to be blocked if a requested lock cannot be granted immediately. This supports higher level concurrency control such as two-phase locking with deadlock detection for transaction support.

When a lock for an object fragment that consists of multiple pages is requested, the corresponding page locks are requested in incremental order to avoid the possibility of deadlock. Deadlock could occur, however, if locks are requested for small object fragments that reside on the same page. The possibilities are self-deadlock and inter-thread deadlock.

The self-deadlock of a thread is illustrated in figure 3(a). After successfully obtaining a read lock for a small fragment A of object Z, a thread requests a write lock for another fragment B in the same page as A. Since the underlying locking mechanism actually locks the whole page, this thread would never get the write lock for B before it releases the read lock for A. This may occur with any conflicting locks; read/write and write/write. Self-deadlock can be prevented by keeping a list of threads that hold a lock in the page lock mechanism and checking before a lock request is denied. When a write lock for a page is requested, if the caller thread holds and is the only one who holds a read lock of the page, the write lock is granted. If a read lock for a page is requested and the calling thread holds the write lock for the page, the read lock is granted.

Threads that request fine-grained locks on the same page can result in inter-thread deadlocks. Figure 3(b), shows an example. After successfully obtaining read locks for fragments A and B, thread T and T1 request write locks for fragments C and D. Because there are two holders of the read lock for page N their requests can not be satisfied unless one of them releases the read lock. A deadlock occurs. A simple way to prevent inter-thread deadlock is to use the non-blocking mode of lock request and spin-wait for a certain amount of time. If the lock is still not granted then release all the locks held, e.g. abort a transaction.

4.4. Coherence protocols

Two groups of coherence protocol, write-invalidate and write-update, are supported in the current design. They are derived from well known protocols and the modifications made in this work do not change their correctness. The locking mechanism is integrated with the coherence control mechanism. When a copy of a page is requested to satisfy a page fault, the owner CoherSvr has to obtain a local read lock for the page before it can send a copy of the page to the faulting node. In the write-update protocols, in addition, when the owner multicasts an update, the CoherSvr that receives the update has to wait for all the reading threads to complete their current reading sessions before it can actually update the local cache. This assures the multiple-reader/single-writer constraint in distributed environments. System wide concurrency control is achieved without a global locking mechanism. Also in
the write-update protocol, releasing a write lock to a page triggers the POM to multicast the up-to-date state of the page to the copy set.

The write-invalidate protocols, namely the centralized-control protocol and the distributed-control protocol, are derived from those used in the IVY shared virtual memory system [23] but differ in several ways. First, coherence control in IVY is not integrated with concurrency control. It is prone to thrashing. For example, if two nodes compete for write access to a single page, it may be transferred back and forth at such a high rate that no real work can get done. By integrating low-level coherence control with high-level concurrency control, this ping-pong effect can be avoided in COMMOS because the page is only transferred to another node when a write lock on the page is released. Hence, the number of messages required to maintain object coherence is reduced. Second, COMMOS supports persistent data object sharing and the clients can choose different protocols for different applications while IVY provides a flat shared virtual address space without persistence and its clients have no flexibility to apply different protocols to different applications. Third, COMMOS employs parallel MultiRPC for one-to-many communications while IVY uses only simple RPC.

The implementation of the write-update protocols is simplified by integrating low-level coherence control with the high-level locking mechanism. The reason is that the standard kernel-level page fault handling is unsuited to the task of processing updates in the way that the write-update option requires. Also the number of messages required to maintain object coherence is reduced because the multiple updates to a page are grouped in a single message to propagate to the copy set when the write lock is released.

4.5. Coherence mechanism

All coherence protocols discussed above are supported by a set of CoherMgr and CoherSvrs. The CoherMgr provides POMs with facilities to open and close shared persistent objects, to write modifications back to the storage server and to obtain page data or write permission. It also provides CoherSvrs with facilities to update the owner information on the CoherMgr and to forward POMs’ requests. The CoherSvr provides POMs with facilities to obtain page data or write permission. It also provides the CoherMgr and other CoherSvrs with facilities to forward POMs’ requests, to invalidate or update object pages. New coherence protocols can be added by modifying the implementation of the existing CoherMgr and CoherSvr or by adding new sets of CoherMgr and CoherSvrs.

5. Implementation

The prototype system consists of three MVME147 (MC68030) machines running the Wanda [6] microkernel operating system and a DEC 3100 workstation running Ultrix 4.3 as shown in figure 4.
MultiRPC has been ported onto Wanda and is running over the UDP/IP communication protocols without multicast support. TCP/UDP/IP is implemented on Wanda using a user space process called the internet server. The POM and CoherSvr are implemented on Wanda. The CoherMgr is implemented on Ultrix and a storage server (StorSvr) emulator using NFS as the backing store is implemented to provide the persistent storage for the prototype system. They communicate using MultiRPC over a non-dedicated Ethernet. Both write-invalidate protocols are implemented.

Objects are named by NFS path names. Each object is represented by an entry in an object table, which contains information about the object such as its name and type and which paging algorithm and coherence protocol to apply to the object. This object table is mapped into the address spaces of POMs so that it can be used to interact with the user processes. The Wanda event mechanism is used to notify a POM of events occurring on an object that require its attention. The POM examines the state of the object and gets the necessary information to deal with the event, then returns to user space. After serving the event, it returns the results to the threads waiting on the event and unblocks them. When a POM is invoked to service a page fault, it checks to see what coherence protocol is to be applied. If this field is not nil, the POM gets the coherence information of the object page and acts accordingly. After the page data or the write permission is obtained, it sets the new coherence information, such as the access type, the owner and the copy set of the object page.

A process consists of various objects namely a code object, a data object, a bas (uninitialized data) object, a stack object, an environ object, and other objects created or mapped by the process. Information about where and how an object is mapped is contained in a process map, which is mapped read-only into the address space of the process as part of its initialisation. Each entry in the table, known as a map_entry, contains the name of the object, the type of the object, the starting address and the length of the object, the access rights of the user, the index of the map_entry in the process map and the number of threads accessing the object in the same address space [25]. User processes are managed by a privileged user process called the process server (ProcSvr).

The CoherMgr can be implemented on the same node as the StorSvr or on any ordinary node. In the prototype implementation it is implemented as lightweight threads on Ultrix sharing a process address space with the StorSvr. The light-weight process (LWP) package provided with RPC2 [29] is used to support multiple lightweight threads. The prototype CoherSvr is implemented as lightweight threads sharing an address space with the prototype POM. However there is no conceptual constraint in the architecture which would prevent the CoherSvr from being implemented as a separate user-level process. In fact, if there exist multiple POMs for different object types which share the services provided by a CoherSvr, the CoherSvr is better implemented as a separate process.

### Table 1. Basic activities.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalidating a local page</td>
<td>0.42</td>
</tr>
<tr>
<td>Servicing a page fault</td>
<td>0.76</td>
</tr>
<tr>
<td>A null RPC from Unix to Wanda</td>
<td>12.1</td>
</tr>
<tr>
<td>A null RPC from Wanda to Unix</td>
<td>13.7</td>
</tr>
<tr>
<td>A null RPC from Wanda to Wanda</td>
<td>16.8</td>
</tr>
<tr>
<td>Reading a page (1K bytes) from an NFS file</td>
<td>16.96</td>
</tr>
<tr>
<td>Fetching a page from the StorSvr</td>
<td>31.5</td>
</tr>
</tbody>
</table>

### 6. Performance

Some performance data is reported in this section. The main purpose is to analyse the cost for basic operations in the system and to demonstrate the feasibility of the COMMOS approach. All the results are averages of more than 10 000 iterations of the operation. The timings for basic activities are shown in table 1. The first two entries give the cost of the VMM layer. The figure for servicing a page fault does not include the time spent by the POM to fetch a remote page. The next three entries give the communication cost and the sixth entry gives the file access cost. The last entry gives the cost for the POM to fetch a page directly from the StorSvr without a coherence guarantee (it includes an RPC from Wanda to Unix and a read access to an NFS file).

The time to fetch a remote page to service a read fault is shown in table 2. When the CoherMgr is used as a relay node, it means that the faulting node makes its request to the CoherMgr. In the centralized-control protocol, when the owner of the page requested is another Wanda machine, an operation is invoked to write the modified data back to the secondary store. Such a nested RPC is counted as 2 RPCs in this section. In the distributed-control protocol, in order to avoid multilevel nested RPC in the case that the request has to be forwarded more than once, a separate RPC is used in the CoherSvr to either send the page data to the faulting node or to forward the request to the probable owner known by the node.

The time to fetch a page to service a write fault is shown in table 3. It can be seen that the time taken to service a write fault when there are one or two nodes in the copy set is almost the same. This provides further evidence that the use of MultiRPC improves the performance of one-to-many communications significantly. When the owner of the page requested is another Wanda machine, an operation is invoked to write the modified data back to the secondary store. This explains why the time to service a write fault is longer than the time to service a read fault in the same situation. Because the CoherMgr runs on a Unix system there is a degree of unpredictability in any timings which involve it. Efforts have been made to reduce the unpredictability: first by ensuring that all the measurements are taken in the middle of the night and that there are no other user processes running on the machine thus reducing unpredictability of process scheduling; second by ensuring that the StorSvr closes the file at the end of every page fetch in order to eliminate the unpredictability of file buffer...
Virtual memory support for distributed computing environments

Table 2. Fetching a page for read.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Owner</th>
<th>Relay node</th>
<th>Number of RPC</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized-Control</td>
<td>CoherMgr (Unix)</td>
<td>CoherMgr (Unix)</td>
<td>1</td>
<td>32.9</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>CoherMgr (Unix)</td>
<td>2</td>
<td>40.6</td>
</tr>
<tr>
<td>Distributed-Control</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>1</td>
<td>32.4</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>CoherMgr (Unix)</td>
<td>3</td>
<td>55.3</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>—</td>
<td>1</td>
<td>31.8</td>
</tr>
</tbody>
</table>

Table 3. Fetching a page for write.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Owner</th>
<th>Relay Node</th>
<th>Copy Set</th>
<th>Number of RPC</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized-Control</td>
<td>CoherMgr (Unix)</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>1</td>
<td>32.8</td>
</tr>
<tr>
<td></td>
<td>CoherMgr (Unix)</td>
<td>CoherMgr (Unix)</td>
<td>2</td>
<td>2</td>
<td>56.3</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>2</td>
<td>57.5</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>3</td>
<td>123.6</td>
</tr>
<tr>
<td>Distributed-Control</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>32.6</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>4</td>
<td>99.0</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>CoherMgr (Unix)</td>
<td>—</td>
<td>1</td>
<td>155.7</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>65.4</td>
</tr>
<tr>
<td></td>
<td>CoherSvr (Wanda)</td>
<td>—</td>
<td>—</td>
<td>3</td>
<td>99.9</td>
</tr>
</tbody>
</table>

management. However, there were still system daemon processes running when the measurements were taken. The fifth and the eighth entries almost certainly represent Unix process rescheduling effects.

There are several reasons for these relatively high values. The internet server, the POM and the CoherSvr all run in user space. This means more context switches are involved than in monolithic kernel systems where all the system services run in the kernel. Second, while the architecture is aiming for future high speed and wide address space environments, the prototype is implemented on slow machines and a slow network. Third, as described above, file caching is not used in the measurements to avoid unpredictability. It should also be noticed that the prototype has not been specially tuned for optimal performance.

By comparing the figures of tables 2 and 3 with those of table 1, it can be seen that the major cost is communication and NFS file access. The overhead for servicing multiple coherence protocols at the same time is not significant. As more powerful processors and faster networks become widely used, in conjunction with the use of a faster storage system, COMMOS will become a viable approach to build distributed applications.

7. Persistent C++: an example of the value-adding layer

As an example of the value-adding layer, a prototype C++ class library, which makes use of the object mapping and locking mechanisms provided by COMMOS to support distributed persistent programming, has been built. A GNU g++ 2.3.3 cross compiler has been configured and installed on mips Ulrix to generate executable code for Wanda. Tested code fragments illustrating the implementation can be found in [17].

In C++ [30], an object can be created by the operator new and destroyed by the operator delete. The new operator attempts to create an object of the object type to which it is applied and returns a pointer to the object created. It will call the function operator new() to obtain storage. The first argument must be of type size_t, an implementation-dependent integral type defined in the standard header <stddef.h>. An object created by the new operator exists until it is explicitly destroyed by the delete operator. The delete operator may be applied only to a pointer returned by new or to zero.

It is possible to take over memory management for a class by defining operator new() and operator delete(). This remains possible and is even more useful for a class that is the base for many derived classes. This feature makes supporting distributed persistent programming on COMMOS very straightforward. What should be done is to define a base class for persistent object classes and implement the operators new and delete to map and unmap persistent object states.

After defining the operators new and delete for the base class of all persistent object classes, application programmers can define their own persistent classes as derived classes of the base class and use operators new and delete to map and unmap object states.

No modification to compilers is required. The class library implementors and the end users need not be concerned about the location of objects and data movement between main memory and backing store.

8. Conclusions

The major contribution of this work is exploring the architectural support for flexible coherence and persistence for distributed systems. The architecture supports multiple coherence protocols and allows clients to choose the
most suitable protocols for their applications. It also supports the addition of other coherence protocols by the addition of new coherence managers and servers. Through the COMMOS architecture, main memory has been integrated with secondary storage and local memory with remote memory in distributed computing environments while the system is kept open to meet different requirements for different applications. The prototype implementation and performance measurements have demonstrated the practicability and feasibility of this approach.

The current design of COMMOS enables clients to advise the system to apply different coherence protocols to different types of object. More flexible schemes, such as individual object coherence and adaptive coherence may be desirable and can easily be added.

Exploitation of the functionality of the system to support high-level applications, such as distributed persistent programming languages, distributed databases and computer-supported cooperative work will be the ultimate demonstration of the feasibility of the approach.

Acknowledgement

We would like to thank Ken Moody, Sai-Lai Lo, Zhixue Wu and other members of the Opera group at Cambridge for useful discussions and practical assistance during the course of this research.

References

[9] Carter J B, Khondekar D and Kamb L 1995 Distributed shared memory: where we are and where we should be headed Proc. 5th Workshop on Hot Topics in Operating Systems 1995
[11] Dean R W and Armand F 1992 Data movement in kernelled systems (School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA)
[12] Dhawan M, Kaly K and Overstreet C M 1994 Performance evaluation of TCP/IP/UDP/IP over ATM Networks Technical Reports TR-94-23 (Computer Science Department, Old Dominion University, Norfolk
VA 23529-0162, USA)
[18] Objectivity Inc. 1994 Objectivity/DB
[23] Kai Li 1986 Shared virtual memory on loosely coupled multiprocessors PhD Thesis Yale University Department of Computer Science, New Haven, CT 06520, USA (Also available as Technical Report YALEU/DCS/RR-492)
Virtual memory support for distributed computing environments


