Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach

Alessandro Forin, Joseph Barrera, Michael Young, Richard Rashid

August 1988

CMU-CS-88-165

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

A shorter version of this report will appear in the 1988 Winter USENIX conference, San Diego, January 1989

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract number N00039-87-C-0251. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Abstract

This report describes the design, implementation and performance evaluation of a virtual shared memory server for the Mach operating system. The server provides unrestricted sharing of read-write memory between tasks running on either strongly coupled or loosely coupled architectures, and any mixture thereof. A number of memory coherency algorithms have been implemented and evaluated, including a new distributed algorithm that is shown to outperform centralized ones. Some of the features of the server include support for machines with multiple page sizes, for heterogeneous shared memory, and for fault tolerance. Extensive performance measures of applications are presented, and the intrinsic costs evaluated.

1. Introduction

Shared memory multiprocessors are becoming increasingly available, and with them a faster way to program applications and system services via the use of shared memory. Currently, the major limitation in using shared memory is that it is not extensible network-wise, and therefore is not suited for building distributed applications and services. Example uses of a distributed shared memory facility include operating system services such as file systems and process migration, distributed databases, parallel languages like Ada or Multilisp, and systems for parallel and distributed programming [11, 2, 10]. More motivation for a distributed shared memory facility comes from the increasing interest that hardware designers show in non-shared memory multiprocessors: the Nectar project [1] at CMU for instance uses fast fiber optic links. This will reduce the end-to-end time to send a 1 kilobyte data packet from the tens of milliseconds range of the current ethernet to the tens of microseconds range of the fiber. Fast communication makes distributed shared memory an appealing complement to message passing.

The Mach virtual memory system allows the user to create memory objects that are managed by user-defined processes (external pagers) [13]. An external pager is a process responsible for providing data in response to page faults (pagein) and backing storage for page cleaning (page-out) requests. This is precisely the function of the in-kernel disk pager. The only difference is that the user-specified pager task can manage the data in more creative ways than the designer of the in-kernel pager may have envisioned. This paper describes the design, implementation, and performance evaluation of one such memory server which provides a shared memory semantics for the objects it manages. The server provides unrestricted sharing of read-write memory between tasks running either on the same machine or on different machines. In the first case, all processors have direct access to common physical memory (architectures with Uniform Memory Access time (UMA) or Non-Uniform Memory Access time (NUMA)) and the server provides a flexible management of shared memory. In the second case, processors do not have any way to access common memory (architectures with No Remote Memory Access (NORMA)) and the server provides it in software, migrating and replicating virtual memory pages between processors as needed.

To understand the properties of a distributed shared memory facility the performance characteristics of the server itself and of some application programs have been evaluated. To measure the effects of different page management policies in the server, a number of algorithms have been implemented and evaluated, including a new distributed algorithm that outperforms centralized ones. Some of the features of the algorithms described include support for machines with differing page sizes, for heterogeneous processors, and for fault tolerance. The algorithms service page faults on multiple machines by migrating pages that must be shared, scheduling conflicting or overlapping requests appropriately, tagging and translating memory pages across incompatible processors and keeping a duplicate copy in case of machine crashes. The experiments with application programs were designed under the assumption that the amount of information that is communicated in each synchronization operation is the key factor. Applications at the extreme of the spectrum were selected for testing.

2. Shared Memory Within a Machine

The first goal of the server is to provide sharing of read/write memory between tasks allocated on the same machine. This overcomes the constraint of the standard Mach memory inheritance mechanism that the shared memory must have been allocated by some common ancestor, as well as a security check in the implementation of the Unix exec(2) system call that deallocates all of the task's address space. The server provides the user with a call to create *memory objects*, logical pieces of memory that are identified by ports. A memory object can be used by a thread in a call to the *vm_map()* kernel primitive, which maps some portion of the object into the task's address space at some virtual address. Note that since a port can only be transmitted in a message, memory objects are entities protected by the kernel. Note also that access to ports can be transmitted over the network, and therefore the *vm_map()* primitive allows for networked shared memory.

From user to pager

create_memory_object(initial size)
RPC, Creates a new memory object and returns the associated port.

memory_object_replicate(object)
RPC, When using the distributed pager, create a local copy of the memory object.

memory_object_tag(tag, page range)
RPC, When using heterogeneous processors, assign a type tag to a portion of the memory object.

From user to kernel

vm_map(task, memory_object, address range, attributes)
RPC, Maps an object into a task's address space.

vm_deallocate(task, address range)
RPC, Removes all mappings for the given address range.

From kernel to server

memory_object_init(pager, control_port)
MSG, Contact the pager of an object which is mapped for the first time, for initial handshake.

memory_object_data_request(page range, protection)
MSG, Request for a (range of) page which the kernel does not have in its cache.

memory_object_data_unlock(page range, protection)
MSG, Requires more access permissions for a page.

memory_object_data_write(page range, pages)
MSG, Pageout of dirty pages from main memory.

memory_object_lock_completed(page range)
MSG, Completion of the requested paging operation.

memory_object_terminate()
Notification of removal from cache.

From server to kernel

memory_object_set_attributes(attributes)
MSG, Confirms availability completing initial handshake, specifies initial attributes.

memory_object_data_provided(page range, pages)
MSG, Provides page(s) data to the cache.

memory_object_data_unavailable(page range)
MSG, Zero fill page(s).

memory_object_lock_request(object, request, reply_port)
MSG, Cache control request, e.g. page flush or granting of write access.

Figure 1: Summary Of The External Pager Interface

The thread can access the memory normally, and the kernel delegates the paging duties to the user-level memory manager (external pager) that is responsible for the memory object. This is done via an asynchronous message protocol between the pager and the kernel which is described in more detail in [13]. The external pager interface allows pagers to control the managing of main memory by the kernel, so that main memory effectively acts as a common cache for memory objects. The various operations have the flavor of cache control functions: when a thread first accesses a page it takes a page fault and the kernel sends to the pager a *memory_object_data_request()* message to request the missing page, which is similar to a cache miss. The server provides the page in a *memory_object_data_provided()* message. Other messages allow a pager to request a page flush or specify the caching and copy policies for the object. Figure 1 informally lists the messages and remote procedure calls defined by the external pager interface and by the virtual shared memory server.

3. A Simple Algorithm

The shared memory server has been structured in an object-oriented fashion, so that it is possible to have memory objects with different behaviors. When a memory object is mapped by tasks on multiple machines, the pager needs to manage multiple copies of memory pages in some coherent way. The various management policies for memory objects are provided by different implementations of a common set of operations: an implementation is called *fault scheduler* in the following, because the goal of the module is to schedule read and write faults on different kernels in the best way, just like ordinary schedulers schedule the execution order of various threads. One of the many reasons for this choice is to allow experimentation with various algorithms and heuristics. At object creation time, a user can choose which specific scheduling policy will be applied to the new object, or rely on the default one. All the algorithms we describe maintain *strict memory coherence* on the objects they manage. There is no stale data because at any given time each page exists in only one version.

This Section describes a very simple scheduler that provides centralized, single page-size memory objects. There is only one pager task for each memory object, but different objects might be allocated to different pager tasks to reduce service contention. Since Mach IPC is location transparent, the location of the pager task is also transparent to the client kernels. A later Section will describe how this algorithm is modified to allow distributed, coordinated management of a single object between separate pagers on different machines. Ownership of a page is transferred among kernels on demand: the *owner* of the page is the kernel that currently has write access to the page. When no kernel has write access to a page the scheduler itself is the owner, multiple kernels are allowed to have read-only copies of the page. The simple scheduler's algorithm is an automaton with four per-page states, which correspond to the four conditions in which a page can be:

- **Read**: There are no writers, there may be readers with a copy, the server has a valid copy. This is the initial state.
- **Write**: There is one writer, there are no readers and no one is queued waiting, the server does not have a valid copy.
- **ReadWait**: There is one writer, some readers are waiting, the server does not have a valid copy and has asked the current owner to return the page to the server.
- **WriteWait**: There is one writer, some writers are queued waiting, there may be readers waiting, the server does not have a valid copy and has asked the current owner to return the page to the server.

Transitions between states are driven by the requests that are made by client kernels. In practice, not all requests make sense in all states. For instance, a kernel will not pageout a page that has not been modified.[1] The server accepts four input message types (requests), which the scheduler handles in three procedures:

[1] Security issues have not been addressed directly in the server. Rather, it is assumed that other servers, for example a name service integrated with an authentication service, will do the necessary verifications before handing a memory object port to a user. An object might then have different ports associated with it, one for read-only access and one for read-write access. Note then that it is possible to prevent a user from impersonating a kernel by having a secure server handle the object's port directly, and never permitting to unsecure tasks direct access to the port. These and other issues are addressed in a forthcoming document [14]

- *read_fault()*: a kernel requests read access to a page (memory_object_data_request).
- *write_fault()*: a kernel requests write access to a page and either needs a fresh copy of the page (memory_object_data_request) or does not (memory_object_data_unlock).
- *pageout()*: a kernel flushes out the page to the server (memory_object_data_write and memory_object_lock_completed).

These three functions do all the necessary work. A pseudo code description of how they operate on a page appears in Figures 2, 3 and 4. It can be assumed that all procedures keep the page locked and that messages are processed in the order of arrival. This pseudo code will be used again later to describe the distributed algorithm. The remaining procedures are either for initialization, termination, or recovery from kernel crashes. The pseudo code indicates that writers are queued in FIFO order, while readers do not need to be ordered. Writers take precedence over readers. Other, possibly more complicated policies might be needed. It is possible, for example, to introduce a simple test to prevent writers from causing starvation of readers. Sections 3.1 will expand on the queueing strategies. If we ignored fault tolerance issues the algorithms would differ only in a minor way: the server can dispose of a page once it is sent to a writer. This and other optimizations can be easily applied in the case the server runs without the (local) support of a permanent storage server, which is the case of a diskless workstation.

```
read_fault(page, kernel)
     switch ( page->state ) {
     case Read:
             memory_object_data_provided(kernel)
             break
     case Write:
             page->state = ReadWait
             memory_object_lock_request(page->owner, FLUSH(page), owner_self)
             break
     default: /* just enqueue */
     }
     set_add(page->readers, kernel)
```

Figure 2: Handling of Read Faults

An example will help clarify the following discussion. Since all the tasks on one machine use the same copy of the memory object's pages (cached copy, possibly mapped into the various address spaces with different protections), we can pretend there is a single task per machine. Let us assume that a thread makes a read access to a page. The page is not in the cache, hence the kernel sends a *memory_object_data_request()* message to the pager. If the page is in *Read* state (the initial state), the server immediately sends the page in a *memory_object_data_provided()* message, with read-only protection. If the thread makes a subsequent write access, the kernel sends a *memory_object_data_unlock()* message to request a protection upgrade which will be granted in a *memory_object_lock_request()* message, unless the page has changed state in the meantime. If the page is not in *Read* state, the kernel's request is enqueued and possibly the current writer is asked to page out the page via a *memory_object_lock_request()* message. When the page is actually paged out, the *pageout* procedure dequeues the next write access request and satisfies it, or satisfies all read requests at once.

```
   write_fault(page, kernel)
        switch ( page->state ) {
        case Read:
                set_remove( page->readers, kernel)
                forall( readers )
(1)                     memory_object_lock_request( reader, FLUSH(page), owner_self )
                page->readers = empty_set
(2)
                page->state = Write
                page->owner = kernel
                if (needs_data)
                        memory_object_data_provided( page->owner )
                else
                        memory_object_data_unlock( page->owner )
                break
        case Write:
                memory_object_lock_request( page->owner, FLUSH(page), owner_self )
                /* fall through */
        case WriteWait:
        case ReadWait:
                page->state = WriteWait
                enqueue( kernel, page->writers )
        }
```

Figure 3: Handling of Write Faults

```
   pageout(page, kernel, data)
(3)     switch( page->state ) {
        case Read:
                return  /* never happens */
        case Write:
                save(data)  /* true pageout */
                page->state = Read
                page->owner = owner_self
                break
        case WriteWait:
(4)
                save(data)
                page->owner = dequeue( page->writers )
                memory_object_data_provided( page->owner)
                if (!page->writers)
                        if (page->readers)
                                page->state = ReadWait
                        else
                                page->state = Write
                if (page->readers || page->writers) {
                        deschedule_myself()
                        memory_object_lock_request( page->owner, FLUSH(page), owner_self)
(5)
                }
                break;
        case ReadWait:
                save(data)
                forall(readers)
                        memory_object_data_provided(reader)
                page->state = Read
(6)             page->owner = owner_self
        }
```

Figure 4: Handling of Pageout Requests

3.1. Multiple Page Sizes

The simple scheduler described above can only be used by machines with the same page size, an unpleasant restriction. Moreover, in Mach the size of a virtual page can be changed and set even on a per-machine basis. Transforming a single page size scheduler into a multiple page size scheduler is not immediate. Our multiple page size scheduler uses internally an arbitrary page size (scheduler page size) and solves the problem by two means:

- for requests *smaller* than the scheduler page size, the request is rounded up to the scheduler page size, and
- for requests *larger* than the scheduler page size, the request is fulfilled by multiple scheduler pages (shipped all at once), after appropriate synchronization.

Synchronization is accomplished via a queueing mechanism. It is necessary to avoid both false contention and descheduling of kernels until absolutely necessary, and to satisfy requests as quickly as possible while maintaining fairness. When the scheduler receives a request from a kernel, it may take one of the following actions:

1. Satisfy the request immediately.
2. Deschedule some writers and enqueue the request.
3. Simply enqueue the request.

The first is the case when there are no writers on any of the data that the kernel requests. For a read request, the scheduler can simply add the kernel to the set of readers of each scheduler-page; if the request is a write request, then the scheduler deschedules all readers of any scheduler-page in the writer's request range before scheduling the writer. In the second case, the scheduler finds that there are writers on some of the requested data, but none of them have yet been descheduled. The scheduler deschedules the writers, and the request is queued.

In the third case, the scheduler finds descheduled writers on some of the requested data, indicating that other requests are already waiting for those scheduler-pages. In this case, the scheduler does not deschedule the rest of the writers because the requesting kernel is not yet ready to use their pages; the request is simply enqueued. When a descheduled writer sends a confirmation (a *memory_object_lock_completed()* message), the scheduler finds the request that was awaiting it. If the confirmation was the last one that the request was waiting for, then the scheduler satisfies the request (as in case 1 above) and checks to see if there are any more requests that might be satisfied as well.

The data structures used for queueing readers and writers allow most operations to occur in constant time, while some (such as determining whether an incoming request can be immediately satisfied) take time proportional to the number of scheduler pages in the request. Each waiting client is represented by a record containing the identity of the requestor, a reference counter, and a pointer to a linked list of requests that follow. The reference counter is used to quickly test if the request can be satisfied. When the request follows other requests the counter represents the number of requests pointing to it; otherwise it is used to represent the number of outstanding descheduling acknowledgements. For each scheduler page there is also a pointer to the request waiting for an acknowledgement from the writer of the page, and a pointer to the last request waiting for the page. These pointers are set to nil if no such request exists.

3.2. Heterogeneous Processors

Parallel programs that use a distributed shared memory facility should not be constrained to run on a uniform set of processors. Such a constraint is undesirable because as the number of machines available at a given site increases one typically observes an increased variation in their types as well. Unfortunately, interfacing heterogeneous processors not only creates the problem of potentially different page sizes, but also raises the issue of different machine representations of data objects. This problem goes beyond the *byte order* problem, since different processors are free to assign any given meaning to any given sequence of bits. A clear example is the case of floating point numbers.

A more difficult set of problems arises from *software* data types. Modern programming languages allow higher level types to be built on top of hardware types, for instance in composing record structures with diverse component types. Quite often, the language definition does not specify how these types should be mapped to the hardware types, and the compiler is free to define this mapping as appropriate. A well known consequence is that the different fields of a record in the C language may be allocated at different offsets by different compilers, sometimes even among compilers for the same machine architecture. Finally, some languages use types that do not have any correspondent hardware type. Lisp systems, for instance, often use runtime data tags to mark a collection of bits as the representative of some data type (see [12] for a recent analysis). Only a few processors implement some form of data tagging in hardware.

Solving the heterogeneity problem is difficult because it requires that the server has knowledge of the application's data types. This leads to undesirable close links with the application's runtime system and programming language [2]. On the other hand, the problem can be separated in two sub-problems: hardware data types (e.g. integers) and software data types (e.g. C records). A general purpose server solves the problems for the first class of types, and can be extended to cope with the second class of types. Quite simply, our server assigns a type tag to each segment of a paging object and makes the appropriate translation (if necessary) when sending data from that segment to a kernel. The interface with the application program is defined by the *memory_object_tag_data()* RPC from the client to the pager that assigns a type tag to a segment. This operation is typically used by a dynamic memory allocator to fragment shared memory in typed segments, each segment containing only data of the given type. The standard Unix BSD malloc(2) memory allocator for C was modified to allocate typed data, as exemplified in Figure 5. Although different types cannot be mixed in a structure, one can always resort to a level of indirection, building records that only contain pointers to data.

```
extern char
        *tmalloc( type_tag, num_elements )
enum { t_int8, t_int16, t_int32, t_float32, ... } type_tag;
unsigned long int num_elements;

#define malloc_short(n) (short*)tmalloc( t_int16, n)
...
```

Figure 5: A Typed *malloc()*

All type tags and machine types must be known to the server in advance, hence each server is able to deal with a limited set of machine and data types. The server refuses type tags or machine types that it does not know how to handle. This limitation is not very restrictive: since the server is a user level process it can be modified quite easily to account for new data or machine types. A dynamic solution requires the use of runtime type descriptors that the server uses for data translation. This approach is described in [5], and solves the problem of software data types as well. It is certainly possible to extend our server in this way.

Finally, note that an approach similar to the one used for data translation might be used for other problems. Some approaches to the implementation of shared libraries require the use of a dinamic linker. Dinamic linking could be done using lazy-evaluation, only linking those pages of code that are actually accessed by the program when they are faulted in. A similar case arises with a secure program loader, which must check that the executable image has not been tampered with. A distributed object system might also use similar techniques while mapping objects into the program's address space.

4. A Distributed Algorithm

The motivations for a distributed algorithm are manyfold. A centralized server is a solution that does not scale up. When many kernels share many memory objects serviced by the same pager the availability of each object decreases, because the pager becomes the bottleneck where all requests pile up. Even when few kernels are involved, the location of the server is important because local and remote messages might have very different costs. A distributed solution that can allocate any number of servers on any number of machines is more usable. In this way the sharing of memory between tasks located on the same (multi)processor is decoupled from unrelated events on other machines. A careful analysis of the external pager protocol [13] also reveals one inefficiency: transferring ownership of a page from one kernel to another requires four messages (requesting the page, obtaining it, receiving the end-of-transfer message, shipping it to the right kernel), while only two messages are strictly needed (request the page transfer, ship it from one kernel to the other). Rather than modifying the external pager interface to handle this case, we have designed and implemented a distributed paging server which exploits this and various other opportunities for reducing network traffic.

The approach taken is simple: *treat each remote server just like another kernel, and apply the algorithm of the centralized case*. The reader may wish to go back to Figures 2, 3 and 4 and review the algorithm substituting the word "kernel" with "client", which now means either a kernel or (more likely) a fellow server. A pager will now accept a *memory_object_lock_request()* message just like a Mach kernel does and treat it as a fault notification, invoking read_fault() or write_fault() as appropriate. A *memory_object_data_provided()* message is handled by the pageout() procedure.

Note now that the notion of the "owner" that each pager has does not need to be exact at all times. It is quite possible, actually highly desirable, that a pager be able to ask a second pager to transfer a page directly to a third one who needs it, without handling the page directly. We call this optimization **forwarding**, to catch both the positive effect of avoiding one message hop, and the (minor) negative effect of producing a new type of activity: the act of forwarding a mis-directed page fault message to the correct destination. Implementing forwarding requires relatively simple changes to the centralized algorithm.

Figures 6 and 7 illustrate the changes and additions to the pseudo code of Figures 2, 3 and 4 to implement forwarding. A pager creates a local copy of a memory object when a user asks for it. The initial state of all pages in this case is the *Write* state, and the owner is the pager from which the object has been copied. Of course, no real copy is actually done. Note that it is possible to copy from another

```
(1)     memory_object_lock_request( reader, FLUSH(page),
                is_server(page->owner) ? kernel : owner_self)

(2)     if (page->owner != owner_self) {
                memory_object_lock_request( page->owner, WRITE_FAULT(page), owner_self)
                enqueue(page->writers, kernel)
                page->state = WriteWait
                return
        }

(3)     if (kernel != page->owner && !hinted(page))
                page->owner = kernel
        hinted(page) = FALSE

(4)     if (!page->writers) {
                page->owner = owner_self
                goto ReadWait
        }

(5)     if (is_server(page->owner))
                page_state = WriteWait  /* pretend */

(6)     if (!is_server(kernel))
                page->owner = owner_self
```

Figure 6: Modifications to the Distributed Scheduler
to Implement Forwarding of Page Faults

copy, and that the pager does not need to have complete knowledge of all the kernels involved. The handling of read faults does not change. While handling write faults, at line (1) all readers are informed of who the new owner is, if it is a different pager. At line (2), a check is added to see whether the true owner actually is another pager, in which case the fault is queued and the state of the page modified accordingly. In the pageout() procedure at line (3) it is necessary to handle the case where the pager has incorrect information about the true owner. Note that the pager might have received a *hint* about who will eventually become the owner because it forwarded a write fault. At line (5) it is necessary to handle specially the case when a page is given to a server queued for writing, while having other readers waiting. The immediate request to have the page back pretends that there are writers queued anyway, to prevent the race that would otherwise arise. Line (4) jumps to the correct code in case the last writer had actually been serviced. Line (6) handles the fact that if the pager only receives read-only access to the page it does not become the owner of the page.

Two new procedures, described in Figure 7, are used to check whether a page fault must be forwarded and to handle invalidations of read-only pages. A *memory_object_lock_request()* message is handled first by the page_fault() procedure, which forwards it if necessary. The fault is definitely not forwarded if the pager has ownership of the page, or the pager has already asked the current owner for write access to the page (state WriteWait), or if the pager has (state Read) or is about to have (state ReadWait) a read-only copy of the page and the fault is a read fault. In other words, a fault is only forwarded to another server when the pager has no current interest in the page whatsoever. An invalidation of a read-only page is generated at lines (1) and (7) if the reader is a server, and is handled in the invalidate_page() procedure. This is the only new message type needed.

Forwarding creates problems for a closed form analysis, since the effect of forwarding of both page locations (page faults) and invalidations (page flush) are difficult to model. Our claim is that in actual use one will typically see only the two extreme cases: pages that are frequently accessed in write mode by many parties, and pages that are accessed infrequently, most likely in read mode. Even if a page is

```
    invalidate_page(page, owner)
         if (page->state != Read)
                 return  /* sanity check */
         forall (readers)
(7)              memory_object_lock_request(reader, FLUSH(page), owner)
         page->state = Write;
         page->owner = owner;

    page_fault( page, who, fault_type)
         if ((page->owner == owner_self) ||
             !is_server(page->owner) ||
             (page->state == WriteWait) ||
             ((fault_type == READ) && (page->state != Write))) {
                 if (fault_type == READ) read_fault(page, who)
                 else write_fault(page, who)
                 return
         }
         /* Forward */
         send_page_fault(owner,who,page)
         if (fault_type == WRITE) {
                 page->owner = who
                 hinted(page) = TRUE
         }
```

Figure 7: Additions to the Distributed Scheduler to Implement Forwarding of Page Faults

accessed infrequently, it is hard to generate a faulting sequence that produces many forwarding messages. This claim is supported by the experience with actual application programs. Infrequently accessed pages do not affect performance. The bottlenecks derive very easily from the opposite case. Our analysis shows that the expected number of remote messages required to service a N-party page fault for the distributed pager is

- 3N-4 initially, and
- 2N-1 or
- 2N at steady state

depending on boundary conditions. To get the total number of messages in the distributed scheduler one must add a total of 2N-2 local messages between pagers and the kernels they service. For comparison, any centralized algorithm that maintains strict memory coherence must use at least 4N remote messages and no local messages. In the case of the simple scheduler this figure is 5N messages. Since the cost of local messages is often much less than the cost of remote messages, the distributed pager clearly outperforms the centralized one. The performance evaluation results, reported in Section 7 confirm this analysis.

4.1. Example

When a thread first maps a memory object in its address space the kernel contacts the server but does not require it to send any data yet. It is only when a thread touches a memory location within the address range where the object is mapped that a fault is generated. The faulting thread is stopped, and a message is sent to the pager to request data to service the fault. When the scheduling algorithm in the server has the necessary data available the page is sent to the kernel which maps it for the faulting thread which can then continue execution. In case the page is not immediately available at the server, a message is sent to the kernel that currently owns the page, asking to page it out to the server. In the case of the distributed algorithm, this may imply some more processing, since the "kernel" is actually another server.

It is interesting to consider one example that shows the effects of forwarding page faults among distributed servers. Let us assume that N servers (each one serving one or more kernels) all take repeated page faults on the same page, which is the *hotspot* case that makes distributed shared memory perform the worst. Initially, all servers refer to the memory object's pages from the same one (say server 1). Therefore N-1 requests are sent to server 1. The server first services its local fault(s), then ships the page to server 2 (say) which becomes (in server's 1 opinion) the new owner. The next fault request is then forwarded by server 1 to server 2, the next to server 3 and so on, to server N-1. When all faults have been forwarded and served, the situation is such that servers 1, N-1 and N all know that the page is located at server N, while every other server *i* believes the page is at server *i+1*. When all servers take the next page fault only 2 requests are sent to the owner, and any other request *i* is queued at server *i+1* waiting for *i+1* itself to be served first.

```
S1      S2 ->   S3 ->   S4 ->   ...     Sn-1 -> Sn
|                                               ^
-------------------------------------------------
```

Figure 8: Steady State Behavior for a N-Party Write Hotspot

This situation is depicted in Figure 8 and can repeat itself. Our experiments show that indeed in a write-hotspot the system oscillates between two configurations of this type, never entering the initial state again. There is a worst case that could surface: an isolated page fault triggers a number of forwarding messages. This number is N-2, since always at least two servers know exactly where the page is: the owner and the one who sent the page to it. In the example, this would happen if server 2 alone takes a fault after the first N faults are served. After a worst case fault all servers know exactly where the page is, and therefore the system goes back to the initial state.

5. Fault Tolerance

A network memory server must be prepared to handle machine crashes and network partitioning without deadlocking. Once a crash has been detected, the server must either make user programs aware of the problem (for example signaling a memory error), or attempt to recover from the problem one way or another. Whatever action the server takes will not provide *application-level* fault tolerance since the crash could leave memory inconsistent from the application's point of view. This happens, for instance, when a kernel crashes and some shared memory lock was held by a thread running on that processor.

The centralized schedulers provide a mechanism for surviving kernel crashes whereby memory availability is preserved despite a failure of the current owner of a page. This avoids the alternative of making the whole object permanently unavailable. Assuming the current writer crashes (or for any reason is not capable of communicating with the server any more) the server reverts to the latest copy it has of the page, which is the one that was sent to the writer when it asked for write permission. Fault tolerance mechanisms for the distributed scheduler have not yet been implemented, and they will need to face the problems of network partitioning as well.

Failure of a kernel only needs to be detected when the server needs a page back from it. The overhead of a fault tolerance guard can therefore be quite limited, about 1% of our servers' time when heavily used.

6. Related Work

Forwarding creates a new need: the need of forwarding page faults to the current owner of a page. Li [7] looked at the problem of locating a page and provided various algorithms to solve it, and analyzed their costs. Our distributed algorithm must be compared against the "Distributed Manager Algorithm 2.6", with the optimizations indicated at pages 61-63 that page invalidations are sent in a divide-and-conquer fashion. Note however that in Li’s algorithms all operations are RPC, hence requiring twice as many messages and unnecessary serialization. Li also evaluates the use of broadcast messages and proves that they could benefit some of his algorithms, under the assumption that their cost is the same as a direct message. Note that in our algorithm the use of broadcasts would be detrimental to performance, since it brings back the system to the initial state and away from the most favorable situation. The idea of propagating invalidations in a divide-and-conquer fashion is, in our system, much more effective than broadcasts. In this paper it was only assumed that the underlying architecture provides efficient point-to-point communication, with quasi-uniform cost. The cost of sending a message to N recipients is therefore greater than or equal to N times the cost of a message to a single recipient.

Cheriton [3] has recently extended the V kernel to support user-level data and caching servers, which can be used to provide distributed shared memory. His facility has many similarities with Mach’s external pager facility, although it is described in terms of file abstractions rather than memory object abstractions. The implementation uses a scheme analogous to the simple scheduler presented above, but might add considerable extra message traffic by polling and forcing page flushes every T-milliseconds to provide *T-consistent* files for transaction support.

Fleisch [4] has extended the Locus kernel to provide distributed shared memory, with a SystemV interface. The scheme he describes seems geared to maintaining consistency at the segment rather than page level. A report on the implementation work will be necessary to better evaluate his approach.

Our work is concerned with Operating System level distributed shared memory, where it is implemented as shared pages of *virtual memory*. Other approaches to user-level shared memory objects are possible, for example providing shared *data structures* as in the Agora [2] system. Other references can be found in [2].

7. Performance Evaluation

The performance of the server was evaluated along a number of dimensions. Fundamental are the average times to service a fault, in both cases of single machine and multi-machine applications. These are affected by the various special features of the server. The centralized and distributed cases were compared, using ad-hoc programs that exercise the hotspot behavior. Our measures show two overall results: the distributed algorithm is more efficient than the centralized one, and none of the special features we introduced has an unacceptable impact on performance. The major bottleneck in the test configuration (token ring workstations) is the network latency, which accounts for about 98% of the elapsed times. The server was instrumented in two ways: keeping track of the number and type of faults it services (per object and per page), and collecting extensive traces of the message activity. These data were obtained via a remote procedure call by other processes, with minimum perturbation.

7.1. Basic Costs

The most common use of the server is in sharing memory within a single machine. In this case, a fault on a missing page (cache-fill) requires two local messages, for a total cost of 1.5ms on a IBM RT-APC. A protection fault also requires two messages but no memory mapping, for a cost of 1.1ms. A pageout operation requires two receive messages and the deallocation of data, which is not a system call but a RPC to the kernel and involves two messages[2]. The total cost is then 2.5ms. Since system time is by far the dominant factor (93%) in all cases, schedulers do not show significant differences in the handling of local faults. Table 1 summarizes the most important costs.

Memory use is an important factor for characterizing the performance of a program, although our primary concern was speed rather than space. The server allocates memory in a sparse fashion only when a kernel demands it, and then replaces each page as it is paged out by a kernel. This not only reduces the memory usage for a large and sparse object, but also removes from the critical path the copying of data (just switch a pointer) and the deallocation of memory (two messages) which can be done in batches. To quantify these improvements, the hotspot cycle time for the distributed case for the simple scheduler was reduced by this strategy from 7.8ms/fault to 5.5ms/fault, including memory deallocations. Memory deallocation can be devoted to a separate thread, which reduces the fault time to approximately 4.2ms/fault. Memory saving depends on the actual use, and is very effective for some applications.

Parameter	**Measured Cost**
Zero-fill Fault	**1.5ms/fault**
Protection Fault	**1.1ms/fault**
Hotspot Cycle	**4.2ms/cycle**
Multiple Page Size Overhead	**0.2ms/fault max**
Avg Messages, centralized hotspot case	**5.0/fault (all remote)**
Avg Messages, distributed hotspot case	**4.1/fault (2.0 remote)**
Forwarded Faults	**10% (hotspot)**
System Time	**93%**

Table 1: Costs Of The Server.

7.2. Costs of The Algorithms

The multiple page size scheduler adds some overhead to the fault times, primarily because more server pages might be needed to cover a kernel's page fault. In most cases, a small range of page sizes will be used, but even with an unlikely ratio maximum/minimum page size of eight the overhead over the basic fault times is only 0.2ms. If necessary, however, the algorithm can be tuned further for larger page size ranges.

Various experiments were performed on the distributed scheduler, the most interesting one being the case of an hotspot page. This is demonstrated by a simple program that repeatedly increments the same memory location, replicated on various machines. The measures show that on average each server

[2] As noted later, deallocation of memory is done by a separate thread, which means that for evaluating the *latency* of the server a value of 1.5ms must be used

received requests for read/write/protection faults in an equal amount, as expected. This also means that the user program was interrupted 60% of the times when attempting to write the location (write or protection fault), and 30% when reading from it (read fault). The average number of messages per fault is the single most important figure: on average, each server handled 4.1 messages per fault. Half these messages are received and half sent. On average, 2.1 messages are local (interactions with the local kernel) and 2.0 are remote (interactions with other servers). This nicely confirms the estimates presented in Section 4. Remote messages are extremely more expensive than local ones: an average 98% overhead was observed in the test system, equally divided among the local Mach network server, the remote one, and the TCP/IP transfer protocol.

The results indicate that the distributed algorithm makes the page available in a fair fashion, in the sense that among homogeneous processors the page is made available for an equal amount of time to all kernels: the number of operations for all programs were the same within a factor of 2%. If processors of different speed are used, the time during which a page is available does not change (it is bound by the network latency): using a processor two times as fast on our RTs exactly doubles the number of operations in the user programs. Other measures indicate that during the experiment each server handled about 58% local faults and 42% remote faults, including a 10% of requests that are forwarded. The total number of faults was the same (within 2%) on all servers. Each server requested or provided the page the same number of times (within 3%), including the case of a mix of slow and fast processors.

Machine	Int16/32	Float32	Float64
Sun 4/260 (*)	0.8	1.0	1.1
Vax 8800	1.5	2.3	3.7
IBM RT	1.9	2.4	2.5
Sun 3/280	1.9	2.5	2.9
μVax-III	2.8	4.6	6.8
Sun 3/160	3.0	4.8	4.6
Vax 785	4.4	7.6	10.9
Encore (*)	4.9	12.5	14.3
μVaxII	6.1	10.4	14.5
Vax 8200	9.1	15.3	27.9

Table 2: Overhead of Data Translations
(in milliseconds per 4kbytes).

For the heterogeneity problem, only those machine types that are more or less implied by the definition of the C language were chosen for implementation, which means integers of various sizes and floating point numbers. Many other data types map obviously onto these types. Support for software types is not provided. For floating point numbers, the two formats that are most often used on our machines (Vax-D and IEEE-754) were selected. Both short (32 bits) and long (64 bits) forms were considered. Table 2 shows the overheads measured on the server on a wide variety of machines. The times reported are those necessary to convert 4kbyte of data, but note that some machines use larger page sizes. There is no other basic overhead beyond a simple test of whether conversion is necessary or not. Starred entries in the table indicate machines for which a Mach External Pager kernel is not yet available. In these cases, a synthetic test was run to time the critical code. Note that the translation process is very much dependent on the processor type because the availability of special byteswap instructions can speed it up considerably. This is demonstrated by the entry for the IBM RT.

Assuming that the server's (multi)processor has spare cycles, it is possible to eliminate the type conversion overhead at the expense of increased memory usage. The server can keep multiple copies of each segment, one per machine type, and pre-translates it when a page is received. Translation is done in parallel by a separate thread, which works in a pipelined fashion with the main thread that services faults. We have not yet implemented this optimization.

In the centralized servers, the indicated overhead is paid each time a page is sent to a kernel, as added time for executing the *memory_object_data_provided()* operation. This means that both read and write faults are affected, for machines that are not of the same general type as the object's creator. There is no overhead for protection faults, or for identical or compatible machines like the case of two Vaxen or the case of a Sun and an IBM RT. Note, however, that in some configurations the overhead of byte-swapping and floating point conversion sum up: In the worst case of a centralized server running on an IBM RT and serving a Vax and an Encore, swapping is required both before and after the floating point conversion. Again, the distributed server performs much better since translation is merged in the page replication process: The server that *receives* a page from a machine-incompatible other server translates it before forwarding it to the Mach kernel. In this case, no more than one translation is ever required, and read or write faults do not require any translation at all when the server has a valid local copy of the page.

7.3. Application Programs

Intuitively, the performance gain from the use of memory sharing techniques comes from the large amounts of information that can be transferred with no cost between parallel activities in each synchronization operation. Below a certain threshold, on a uniprocessor the integration of scheduling and data transfer provided by a kernel optimized for message passing is apparent and wins over the simple busy-waiting scheme of spin-locks. The effect must be visible in the networked case, where spin-locks are more expensive. This was the idea that guided the choice of applications for testing the server. This hypothesis only partially contradicts the suggestion that the *locality* of references would completely dominate performance of a distributed shared memory program.

In the networked shared memory case, all the tasks running on the same machine produce a single load on the pager, and the advantage of one of them obtaining a page that will then be used by other tasks is not apparent. This non-measurable gain was eliminated from the experiments and only one task was allocated per machine even if this is clearly unfair to the pager.

All programs have been developed for a uniform shared memory multiprocessor, and were not modified in any way to get better distributed performance. In the matrix multiplication case, the problem is decomposed so that each machine computes all the elements of some row in the output matrix. In this way it is easy to compute large matrices with few processors. The Shortest Path program is a parallel version of a sequential algorithm which shows Nlog(N) complexity for planar graphs [6]. The program evaluates in parallel the possible extensions to the most promising paths, and each activity only looks in the neighborhood of a point and queues the new extensions to other activities. The other two programs have been used for architectural simulations, on the assumption that they are representatives of a large class of parallel programs. Mp3d is a particle simulator [8] and LocusRoute is a parallel VLSI router [9].

The experiments were performed on machines under standard multi-user operating conditions, including any necessary disk paging. Measures were taken of elapsed and per-thread CPU times. Table 3 shows the results of executing the programs on a small group of IBM RTs on a token ring. The network latency dominates performance, and only the matrix multiplication case shows linear speedup. All programs are known to demonstrate linear speedups on a bus-based shared memory multiprocessor with a small number of processors.

Program	1 Machine	2 Machines	3 Machines
Matrix 128x128	29	15	10
Matrix 256x256	241	122	80
ShortestPath	60	60	40
LocusRoute	277	333	397
Mp3d	8.6	16.1	23.0

Table 3: Execution Times For Some Application Programs

One important factor affecting the performance of an application that uses dynamically managed shared memory is the memory allocation algorithm used. Li described a scheme for memory allocation derived from Knuth's FirstFit scheme. A quick comparison was made with a different one, a descendant of Knuth's FreeList algorithm. Such an allocator is currently used, in a sequential version, by the standard Berkeley BSD Unix distribution. A parallel version was easily created by associating a semaphore to each free list, whereby requests for memory blocks of different sizes proceed completely in parallel. It is much more difficult to make the FirstFit scheme more parallel.

The measurements show that not only does the FreeList algorithm use less memory (1/4 on average) than the FirstFit one, but that it is about 20-30% faster even in the sequential case. Other measurements indicate that a two level memory allocation strategy is very effective in reducing shared memory contention. The simple solution of allocating and deallocating memory in batches for blocks of the most frequently used size often suffices to eliminate the most obvious bottlenecks.

8. Conclusions

A user-level memory server for Mach and the algorithms it uses for dealing with issues like heterogeneity, multiple page sizes, distributed service and fault tolerance was described. The server shows very good performance under all tests, and the distributed algorithm is effective in reducing communication over the (potentially slow) communication medium. Results with application programs are dominated by the network latency, but still optimal in some cases. It is conjectured that the amount of data exchanged between synchronization points is the main indicator to consider when deciding between the use of distributed shared memory and message passing in a parallel application. There is definitely space for more research work: a number of extensions and optimizations can be attempted using more sophisticated caching strategies and heuristics in servicing fault requests.

Besides final user applications (e.g. scientific applications, window managers, etc.) there are a number of operating system utilities that can be built using shared memory, knowing that it is now a resource that is available network-wise. I/O between processes can be modeled as the transfer of ownership of some shared memory buffer. In this way, a process (the producer) can allocate a buffer, fill it with data, and then notify the other process (consumer) that the buffer is available by enqueuing it in, for example, a circular queue. A good case in point is implementation of the Streams abstraction at the user level. Supporting distributed databases with distributed shared memory also becomes more simple. An example of how to structure a file system using the external pager facility was illustrated in [13], and the Camelot system [11] uses the facility to provide distributed atomic transactions. Finally, all parallel languages that assume a shared memory model will port easily on a distributed shared memory system, although they will require some tuning to obtain the best performance.

Acknowledgements

We would like to thank David Black and Roberto Bisiani for their invaluable help in reviewing earlier drafts of this paper.

References

[1] Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., Steenkiste, P. A.
The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.
April, 1989.
To appear in the *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*.

[2] Bisiani, R. and Forin, A.
Multilanguage Parallel Programming of Heterogeneous Machines.
IEEE Transactions on Computers , August, 1988.

[3] Cheriton, D.
Unified Management of Memory and File Caching Using the V Virtual Memory System.
Tech. Report STAN-CS-88-1192, Stanford University, Computer Science Department, 1988.

[4] Fleisch, B. D.
Distributed Shared Memory in a Loosely Coupled Distributed System.
In *Compcon Spring 1988*. IEEE, San Francisco, CA, February, 1988.

[5] Forin, A., Bisiani, R., Correrini, F.
Parallel Processing with Agora.
Tech. Report CMU-CS-87-183, Carnegie-Mellon University, Computer Science Department, December, 1987.

[6] Johnson, D.B.
Efficient Algorithms For Shortest Path Is Sparse Networks.
JACM 24(1):1-13, January, 1977.

[7] Kai Li.
Shared Virtual Memory on Loosely Coupled Multiprocessors.
PhD thesis, Yale, September, 1986.

[8] McDonald, J.
A Direct Particle Simulation Method for Hypersonic Rarified Flow on a Shared Memory Multiprocessor.
Final Project Report CS411, Stanford University, Computer Science Department, March, 1988.

[9] Rose, J..
LocusRoute: A Parallel Global Router for Standard Cells.
In *Conf. on Design Automation*, pages 189-195. June, 1988.

[10] Sobek, S., Azam, M., Browne, J.C.
Architectural and Language Independent Parallel Programming: A Feasibility Demonstration.
In *International Conference on Parallel Programming*. IEEE, Chicago, August, 1988.

[11] Spector, A.
Distributed Transaction Processing and the Camelot System.
Distributed Operating Systems: Theory and Practice.
Springer-Verlag., 1987.

[12] Steenkiste, P.
Tags and Type Checking in LISP: Hardware and Software Approaches.
In *Second Intl. Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-II*. ACM-SIGPLAN, Palo Alto, CA, October, 1987.

[13] Young, M., Tevenian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.
In *11th Symposium on Operating Systems Principles*. ACM, November, 1987.

[14] Young, M.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, Carnegie-Mellon University, 1989.
In preparation.

Table of Contents

List of Figures

List of Tables