# The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System

Michael Young, Avadis Tevanian, Richard Rashid, David Golub,
Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black and Robert Baron
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Appeared in Proceedings of the 11th Operating Systems Principles,
November, 1987

## Abstract

Mach is a multiprocessor operating system being implemented at Carnegie-Mellon University. An important component of the Mach design is the use of memory objects which can be managed either by the kernel or by user programs through a message interface. This feature allows applications such as transaction management systems to participate in decisions regarding secondary storage management and page replacement.

This paper explores the goals, design and implementation of Mach and its external memory management facility. The relationship between memory and communication in Mach is examined as it relates to overall performance, applicability of Mach to new multiprocessor architectures, and the structure of application programs.

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 N00039-85-C-1034. The views expressed are those of the authors alone.

## 1. Introduction

In late 1984, we began implementation of an operating system called Mach. Our goals for Mach were:

- an object oriented interface with a small number of basic system objects,
- support for both distributed computing and multiprocessing,
- portability to a wide range of multiprocessor and uniprocessor architectures,
- compatibility with Berkeley UNIX, and
- performance comparable to commercial UNIX offerings.

Most of these early goals have been met. The underlying Mach kernel is based on five interrelated abstractions; operations on Mach objects are invoked through message passing. Mach runs on the majority of workstations and mainframes within the Department of Computer Science, and supports projects in distributed computing and parallel processing such as the Camelot distributed transaction processing system [21], the Agora parallel speech understanding system [3] and a parallel implementation of OPS5 [7]. Mach has already been ported to more than a dozen computer systems including ten members of the VAX family of uniprocessors and multiprocessors[1], the IBM RT PC, the SUN 3, the 16-processor Encore MultiMax , and the 26-processor Sequent Balance 21000. Mach is binary compatible with Berkeley UNIX 4.3bsd and has been shown to outperform 4.3bsd in several benchmarks of overall system performance [1].

A key and unusual element in the design of Mach is the notion that communication (in the form of message passing) and virtual memory can play complementary roles, not only in the organization of distributed and parallel applications, but in the implementation of the operating system kernel itself. Mach uses memory-mapping techniques to make the passing of large messages on a tightly coupled multiprocessor or uniprocessor more efficient. In addition, Mach implements virtual memory by mapping process addresses onto memory objects which are represented as communication channels and accessed via messages. The advantages gained by Mach in treating memory and communication as duals in this way include:

- increased flexibility in memory management available to user programs,
- a better match between Mach facilities and both tightly and loosely coupled multiprocessors, and
- improved performance.

In this paper we describe the relationship between memory and communication in Mach. In particular, we examine the design and implementation of key Mach memory management operations, how Mach memory objects can be managed outside the Mach kernel by application programs and the overall performance of the Mach operating system.

## 2. Early Work in Virtual Memory/Message Integration

The design of Mach owes a great deal to a previous system developed at CMU called Accent [15]. A central feature of Accent was the integration of virtual memory and communication. Large amounts of data could be transmitted between processes in Accent with extremely high performance through its use of memory-mapping techniques. This allowed client and server processes to exchange potentially huge data objects, such as large files, without concern for the traditional data copying costs of message passing.

---

[1] The VAX 11/750, 11/780, 11/785, 8200, 8300, 8600, 8650, 8800, MicroVAX I and MicroVAX II are supported, including support for QBUS, UNIBUS, MASSBUS and BIBUS devices. Several experimental VAXen are also in use including a VAX 11/784 (four processor 780), 11/787 (two processor 785) and 8204 (four processor 8200).

In effect, Accent carried into the domain of message-passing systems the notion that I/O can be performed through virtual memory management. It supported a *single level store* in which primary memory acted as a cache of secondary storage. Filesystem data and runtime allocated storage were both implemented as disk-based data objects. Copies of large messages were managed using shadow paging techniques. Other systems of the time, such as the IBM System 38 [6] and Apollo Aegis [13], also used the single level store approach, but limited its application to the management of files.

For the operating system designer, a single level store can be very attractive. It can simplify the construction of application programs by allowing programmers to map a file into the address space of a process. This often encourages the replacement of state-laden libraries of I/O routines (*e.g.*, the UNIX standard I/O package) with conceptually simpler programming language constructs such as arrays and records. A single level store can also make programs more efficient. File data can be read directly into the pages of physical memory used to implement the virtual address space of a program rather than into intermediate buffers managed by the operating system. Because physical memory is used to cache secondary storage, repeated references to the same data can often be made without corresponding disk transfers.

Accent was successful in demonstrating the utility of combining memory mapping with message passing. At its peak, Accent ran on over 150 workstations at CMU and served as the base for a number of experiments in distributed transaction processing [20], distributed sensor networks [8], distributed filesystems [12], and process migration [24].

Accent was unsuccessful, however, in surviving the introduction of new hardware architectures and was never able to efficiently support the large body of UNIX software used within the academic community [16]. In addition, from the point of view of a system designer, the Accent style of message/memory integration lacked symmetry. Accent allowed communication to be managed using memory-mapping techniques, but the notion of a virtual memory object was highly specialized and the management of such an object was largely reserved to the operating system itself. Late in the life of Accent this issue was partially addressed by the implementation of *imaginary segments* [24] which could be provided by user-state processes, but such objects did not have the flexibility or performance of kernel data objects.

## 3. The Mach Design

The Mach design grew out of an attempt to adapt Accent from its role as a network operating system for a uniprocessor to a new environment that supported multiprocessors and uniprocessors connected on high speed networks. Its history led to a design that provided both the message passing prevalent in Accent and new support for parallel processing and shared memory.

There are four basic abstractions that Mach inherited (although substantially changed) from Accent: *task*, *thread*, *port* and *message*. Their primary purpose is to provide control over program execution, internal program virtual memory management and interprocess communication. In addition, Mach provides a fifth abstraction called the *memory object* around which secondary storage management is structured. It is the Mach memory object abstraction that most sets it apart from Accent and that gives Mach the ability to efficiently manage system services such as network paging and filesystem support outside the kernel.

### 3.1. Execution Control Primitives

Program execution in Mach is controlled through the use of tasks and threads. A *task* is the basic unit of resource allocation. It includes a paged virtual address space and protected access to system resources such as processors and communication capabilities. The *thread* is the basic unit of computation. It is a lightweight process operating within a task; its only physical attribute is its processing state (*e.g.*, program counter and registers). All threads within a task share the address space and capabilities of that task.

### 3.2. Inter-Process Communication

Inter-process communication (IPC) in Mach is defined in terms of *ports* and *messages*. These constructs provide for location independence, security and data type tagging.

A *port* is a communication channel. Logically, a port is a finite length queue for messages protected by the kernel. Access to a port is granted by receiving a message containing a port capability (to either send or receive messages). A port may have any number of senders but only one receiver.

A *message* consists of a fixed length header and a variable-size collection of typed data objects. Messages may contain port capabilities or imbedded pointers as long as they are properly typed. A single message may transfer up to the entire address space of a task.

---

**msg_send(message, option, timeout)**
*Send a message to the destination specified in the message header.*

**msg_receive(message, option, timeout)**
*Receive a message from the port specified in the message header, or the default group of ports.*

**msg_rpc(message, option, rcv_size, send_timeout, receive_timeout)**
*Send a message, then receive a reply.*

---

**Table 3-1:** Primitive Message Operations

---

The fundamental primitive operations on ports are those to send and receive messages. These primitives are listed Table 3-1. Other than these primitives and a few functions that return the identity of the calling task or thread, **all** Mach facilities are expressed in terms of remote procedure calls on ports.

The Mach kernel can itself be considered a task with multiple threads of control. The kernel task acts as a server which in turn implements tasks and threads. The act of creating a task or thread returns send access rights to a port that represents the new task or thread and that can be used to manipulate it. Messages sent to such a port result in operations being performed on the object it represents. Ports used in this way can be thought of as though they were capabilities to objects in an object-oriented system [10]. The act of sending a message (and perhaps receiving a reply) corresponds to a cross-domain procedure call in a capability-based system such as Hydra [23] or StarOS [11].

The indirection provided by message passing allows objects to be arbitrarily placed in the network without regard to programming details. For example, a thread can suspend another thread by sending a suspend message to the port representing that other thread even if the request is initiated on another node in a network. It is thus possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine is more a function of its servers than its kernel.

Tasks allocate ports to represent their own objects or to perform communication. A task may also deallocate its rights to a port. When the receive rights to a port are destroyed, that port is destroyed and tasks holding send rights are notified. Table 3-2 summarizes the operations available to manage port rights and control message reception.

**port_allocate(task, port)**
*Allocate a new port.*

**port_deallocate(task, port)**
*Deallocate the task's rights to this port.*

**port_enable(task, port)**
*Add this port to the task's default group of ports for msg_receive.*

**port_disable(task, port)**
*Remove this port from the task's default group of ports for msg_receive.*

**port_messages(task, ports, ports_count)**
*Return an array of enabled ports on which messages are currently queued.*

**port_status(task, port, unrestricted, num_msgs, backlog, receiver, owner)**
*Return status information about this port.*

**port_set_backlog(task, port, backlog)**
*Limit the number of messages that can be waiting on this port.*

**Table 3-2:** Port Operations

## 3.3. Virtual Memory Management

A task's address space consists of an ordered collection of valid memory regions. Tasks may allocate memory regions anywhere within the virtual address space defined by the underlying hardware[2]. The only restriction imposed by Mach is that regions must be aligned on system page boundaries. The system page size is a boot time parameter and can be any multiple of the hardware page size.

Mach supports read/write sharing of memory among tasks of common ancestry through inheritance. When a child task is created, its address space may share (read/write) or copy any region of its parent's address space. As in Accent, copy-on-write sharing is used to efficiently perform virtual memory copying both during task creation and during message transfer.

Table 3-3 summarizes the full set of virtual memory operations that can be performed on a task.

## 3.4. External Memory Management

An important part of the Mach strategy was a reworking of the basic concept of secondary storage. Instead of basing secondary storage around a kernel-supplied file system (as was done in Accent and Aegis), Mach treats secondary storage objects in the same way as other server-provided resources accessible through message passing. This form of *external memory management* allows the advantages of a single level store to be made available to ordinary user-state servers.

The Mach external memory management interface is based on the the Mach *memory object*. Like other abstract objects in the Mach environment, a memory object is represented by a port. Unlike other Mach objects, the memory

[2]For example, an RT PC task can address a full 4 gigabytes of memory under Mach while a VAX task is limited to at most 2 gigabytes of user address space by the hardware.

**vm_allocate(task, address, size, anywhere)**
*Allocate new virtual memory at the specified address or anywhere space can be found (filled-zero on demand).*

**vm_deallocate(task, address, size)**
*Deallocate a range of addresses, making them no longer valid.*

**vm_inherit(task, address, size, inheritance)**
*Specify how this range should be inherited in child tasks.*

**vm_protect(task, address, size, set_max, protection)**
*Set the protection attribute of this address range.*

**vm_read(task, address, size, data, data_count)**
*Read the contents of this task's address space.*

**vm_write(task, address, count, data, data_count)**
*Write the contents of this task's address space.*

**vm_copy(task, src_addr, count, dst_addr)**
*Copy a range of memory from one address to another.*

**vm_regions(task, address, size, elements, elements_count)**
*Return a description of this task's address space.*

**vm_statistics(task, vm_stats)**
*Return statistics about this task's use of virtual memory.*

**Table 3-3:** Virtual Memory Operations

object is not provided solely by the Mach kernel, but can be created and serviced by a user-level data manager task.

A memory object is an abstract object representing a collection of data bytes on which several operations (*e.g.*, read, write) are defined. The data manager is entirely responsible for the initial values of this data and the permanent storage of the data if necessary. The Mach kernel makes no assumptions about the purpose of the memory object.

In order to make memory object data available to tasks in the form of physical memory, the Mach kernel acts as a cache manager for the contents of the memory object. When a page fault occurs for which the kernel does not currently have a valid cached resident page, a remote procedure call is made on the memory object requesting that data. When the cache is full (*i.e.*, all physical pages contain other valid data), the kernel must choose some cached page to replace. If the data in that page was modified while it was in physical memory, that data must be flushed; again, a remote procedure call is made on the memory object. Similarly, when all references to a memory object in all task address maps are relinquished, the kernel releases the cached pages for that object for use by other data, cleaning them as necessary.

For historical reasons, the external memory management interface has been expressed in terms of kernel activity, namely paging. As a result, the term *paging object* is often used to refer to a memory object, and the term *pager* is frequently used to describe the data manager task that implements a memory object.

### 3.4.1. Detailed Description

The interface between data manager tasks and the Mach kernel consists of three parts:

- Calls made by an application program to cause a memory object to be mapped into its address space. Table 3-4 shows this extension to Table 3-3.
- Calls made by the kernel on the data manager. Table 3-5 summarizes this interface.
- Calls made by the data manager on the Mach kernel to control use of its memory object. Table 3-6 summarizes these operations.

As in other Mach interfaces, these calls are implemented using IPC; the first argument to each call is the port to which the request is sent, and represents the object to be affected by the operation.

---

**vm_allocate_with_pager(task, address, size, anywhere, memory_object, offset)**
*Allocate a region of memory at the specified address. The specified memory object provides the initial data values and receives changes.*

---

**Table 3-4:** Application to Kernel Interface

---

A memory object may be mapped into the address space of an application task by exercising the *vm_allocate_with_pager* primitive, specifying that memory object (a port). A single memory object may be mapped in more than once, possibly in different tasks.

The memory region specified by *address* in the *vm_allocate_with_pager* call will be mapped to the specified *offset* in the memory object. The offset into the memory object is not required to align on system page boundaries; however, the Mach kernel will only guarantee consistency among mappings with similar page alignment.

---

**pager_init(memory_object, pager_request_port, pager_name)**
*Initialize a memory object.*

**pager_data_request(memory_object, pager_request_port, offset, length, desired_access)**
*Requests data from an external data manager.*

**pager_data_write(memory_object, offset, data, data_count)**
*Writes data back to a memory object.*

**pager_data_unlock(memory_object, pager_request_port, offset, length, desired_access)**
*Requests that data be unlocked.*

**pager_create(old_memory_object, new_memory_object, new_request_port, new_name)**
*Accept responsibility for a kernel-created memory object.*

---

**Table 3-5:** Kernel to Data Manager Interface

---

When asked to map a memory object for the first time, the kernel responds by making a *pager_init* call on the memory object. Included in this message are:

- a *pager request* port that the data manager may use to make cache management requests of the Mach kernel;
- a *pager name* port that the kernel will use to identify this memory object to other tasks in the description returned by *vm_regions* calls[3].

The Mach kernel holds send rights to the memory object port, and both send and receive rights on the pager request and pager name ports.

If a memory object is mapped into the address space of more than one task on different hosts (with independent Mach kernels), the data manager will receive an initialization call from each kernel. For identification purposes, the pager request port is specified in future operations made by the kernel.

---

[3]The memory object and request ports cannot be used for this purpose, as access to those ports allows complete access to the data and management functions.

**pager_data_provided(pager_request_port, offset, data, data_count, lock_value)**
*Supplies the kernel with the data contents of a region of a memory object.*

**pager_data_lock(pager_request_port, offset, length, lock_value)**
*Restricts cache access to the specified data.*

**pager_flush_request(pager_request_port, offset, length)**
*Forces cached data to be invalidated.*

**pager_clean_request(pager_request_port, offset, length)**
*Forces cached data to be written back to the memory object.*

**pager_cache(pager_request_port, may_cache_object)**
*Tells the kernel whether it may retain cached data from the memory object even after all references to it have been removed.*

**pager_data_unavailable(pager_request_port, offset, size)**
*Notifies kernel that no data exists for that region of a memory object.*

**Table 3-6:** Data Manager to Kernel Interface

In order to process a cache miss (*i.e.*, page fault), the kernel issues a *pager_data_request* call specifying the range (usually a single page) desired and the pager request port to which the data should be returned.

To clean dirty pages, the kernel performs a *pager_data_write* call specifying the location in the memory object, and including the data to be written. When the data manager no longer needs the data (*e.g.*, it has been successfully written to secondary storage), it is expected to use the *vm_deallocate* call to release the cache resources.

These remote procedure calls made by the Mach kernel are asynchronous; the calls do not have explicit return arguments and the kernel does not wait for acknowledgement.

A data manager passes data for a memory object to the kernel by using the *pager_data_provided* call. This call specifies the location of the data within the memory object, and includes the memory object data. It is usually made in response to a *pager_data_request* call made to the data manager by the kernel.

Typical data managers will only provide data upon demand (when processing *pager_data_request* calls); however, advanced data mangers may provide more data than requested. The Mach kernel can only handle integral multiples of the system page size in any one call and partial pages are discarded.

Since the data manager may have external constraints on the consistency of its memory object, the Mach interface provides some functions to control caching; these calls are made using the pager request port provided at initialization time.

A *pager_flush_request* call causes the kernel to invalidate its cached copy of the data in question, writing back modifications if necessary. A *pager_clean_request* call asks the kernel to write back modifications, but allows the kernel to continue to use the cached data. The kernel uses the *pager_data_write* call in response, just as when it initiates a cache replacement.

A data manager may restrict the use of cached data by issuing a *pager_data_lock* request, specifying the types of access (any combination of read, write, and execute) that must be prevented. For example, a data manager may wish to temporarily allow read-only access to cached data. The locking on a page may later be changed as deemed necessary by the data manager. [To avoid race conditions, the *pager_data_provided* call also includes an initial lock value.]

When a user task requires greater access to cached data than the data manager has permitted (*e.g.*, a write fault on a page made read-only by a *pager_data_lock* call), the kernel issues a *pager_data_unlock* call. The data manager is expected to respond by changing the locking on that data when it is able to do so.

When no references to a memory object remain, and all modifications have been written back to the memory object, the kernel deallocates its rights to the three ports associated with that memory object. The data manager receives notification of the destruction of the request and name ports, at which time it can perform appropriate shutdown.

In order to attain better cache performance, a data manager may permit the data for a memory object to be cached even after all application address map references are gone by calling *pager_cache*. Permitting such caching is in no way binding; the kernel may choose to relinquish its access to the memory object ports as it deems necessary for its cache management. A data manager may later rescind its permission to cache the memory object.

The Mach kernel itself creates memory objects to provide backing storage for zero-filled memory created by *vm_allocate*[4]. The kernel allocates a port to represent this memory object, and passes it to a *default pager* task, that is known to the kernel at system initialization time[5], in a *pager_create* call. This call is similar in form to *pager_init*; however, it cannot be made on the memory object port itself, but on a port provided by the default pager.

Since these kernel-created objects have no initial memory, the default pager may not have data to provide in response to a request. In this case, it must perform a *pager_data_unavailable* call to indicate that the page should be zero-filled[6].

## 4. Using Memory Objects

This section briefly describes two sample data managers and their applications. The first is a filesystem with a read/copy-on-write interface, which uses the minimal subset of the memory management interface. The second is an excerpt from the operation of a consistent network shared memory service.

### 4.1. A Minimal Filesystem

An example of a service which minimally uses the Mach external interface is a filesystem server which provides read-whole-file/write-whole-file functionality. Although it is simple, this style of interface has been used in actual servers [12, 19] and should be considered a serious example.

An application might use this filesystem as follows:

[4]The same mechanism is used for *shadow objects* that contain changes to copy-on-write data.

[5]The default pager will be described in more detail in a later section.

[6]When shadowing, the data is instead copied from the original.

```
char *file_data;
int i, file_size;
extern float rand();    /* random in [0,1) */

/* Read the file -- ignore errors */
fs_read_file("filename", &file_data, file_size);

/* Randomly change contents */
for (i = 0; i < file_size; i++)
    file_data[(int)(file_size*rand())]++;

/* Write back some results -- ignore errors */
fs_write_file("filename", file_data, file_size/2);

/* Throw away working copy */
vm_deallocate(task_self(), file_data, file_size);
```

Note that the *fs_read_file* call returns new virtual memory as a result. This memory is copy-on-write in the application's address space; other applications will consistently see the original file contents while the random changes are being made. The application must explicitly store back its changes.

To process the *fs_read_file* request, the filesystem server creates a memory object and maps it into **its own** address space. It then returns that memory region through the IPC mechanism so that it will be mapped copy-on-write in the client's address space.[7]

```
return_t fs_read_file(name, data, size)
    string_t name;
    char **data;
    int *size;
{
    port_t     new_object;

    /* Allocate a memory object (a port), */
    /* and accept request */
    port_allocate(task_self(), &new_object);
    port_enable(task_self(), new_object);

    /* Perform file lookup, find current file size,*/
    /* record association of file to new_object */
    ...

    /* Map the memory object into our address space*/
    vm_allocate_with_pager(task_self(), data, *size,
                                TRUE, new_object, 0);

    return(success);
}
```

When the *vm_allocate_with_pager* call is performed, the kernel will issue a *pager_init* call. The filesystem must receive this message at some time, and should record the pager request port contained therein.

When the application first uses a page of the data, it generates a page fault. To fill that fault, the kernel issues a *pager_data_request* for that page. To fulfill this request, the data manager responds with a *pager_data_provided* call.

[7] If the client were to map the memory object into its address space using *vm_allocate_with_pager*, the client would not see a copy-on-write version of the data, but would have read/write access to the memory object.

```
void pager_data_request(memory_object, pager_request,
        offset, size, access)
    port_t memory_object;
    port_t pager_request;
    vm_offset_t offset;
    vm_size_t size;
    vm_prot_t access;
{
    char *data;

    /* Allocate disk buffer */
    vm_allocate(task_self(), &data, size);

    /* Lookup memory_object; find actual disk data*/
    disk_read(disk_address(memory_object, offset),
                            data, size);

    /* Return the data with no locking */
    pager_data_provided(pager_request, offset, data,
                            size, VM_PROT_NONE);

    /* Deallocate disk buffer */
    vm_deallocate(task_self(), data, size);
}
```

The filesystem will never receive any *pager_data_write* or *pager_data_unlock* requests. After the application deallocates its memory copy of the file, the filesystem will receive a port death message for the pager request port. It can then release its data structures and resources for this file.

```
void port_death(request_port)
    port_t request_port;
{
    port_t memory_object;
    char *data;
    int size;

    /* Find the associated memory object */
    lookup_by_request_port(request_port,
                &memory_object, &data, &size);

    /* Release resources */
    port_deallocate(task_self(), memory_object);
    vm_deallocate(task_self(), data, size);
}
```

## 4.2. Consistent Network Shared Memory Excerpt

In this subsection we describe how the memory management interface might be used to implement a region of shared memory between two clients on different hosts.

In order to use the shared memory region, a client must first contact a data manager which provides shared memory service. In our example, the first client has made a request for a shared memory region not in use by any other client. The shared memory server creates a memory object (*i.e.*, allocates a port) to refer to this region and returns that memory object, X, to the first client.

The second client, running on a different host, later makes a request for the same shared memory region[8]. The shared memory server finds the memory object, X, and returns it to the second client.

---

[8]How it specifies that region (e.g., by name or by use of another capability) is not important to the example.

Each client maps the memory object X into its address space, using *vm_allocate_with_pager*. As each kernel processes that call, it makes a *pager_init* call to the memory object X. The shared memory server records each use of X, and the pager request and name ports for those uses. Note that the Mach kernel does not await a reply from the shared memory server, but does perform the *pager_init* call before allowing the *vm_allocate_with_pager* call to complete. Also note that while there is only one memory object port (X), there are distinct request and name ports for each kernel. In this example, the shared memory server may be located on either of the clients' hosts, or on yet another host.

In the second frame, each client takes a read fault on the same page of the shared memory region. The same page may be mapped to different addresses in the two clients. Each kernel makes a *pager_data_request* on X, specifying its own pager request port for X. The shared memory server replies using the *pager_data_provided* call on the appropriate request port. Since the request only requires read access, the shared memory server applies a write lock on the data as it is returned[9]. The shared memory server must also record all of the uses (i.e. pager request ports) of this page.

In the final frame, one client attempts to write on one of the pages which both clients have been reading. Since the writer's kernel already has the data page to satisfy the fault, it makes a *pager_data_unlock* call on X, asking that write permission be granted. Before allowing write permission, the shared memory server must invalidate all of the other uses of this page; it does so using the *pager_flush_request* call. Once all readers have been invalidated, the server grants write access to the first client using *pager_data_lock* with no lock.

## 5. Implementation Details

Four basic data structures are used within the Mach kernel to implement the external memory management interface: *address maps*, *memory object* structures, *resident page* structures, and a set of *pageout queues*.

### 5.1. Address Maps

As in Accent, a task *address map* is a directory mapping each of many valid address ranges to a memory object and offset within that memory object. Additional information stored for each range includes protection and inheritance information.

To account for sharing through inheritance, Mach address maps are two-level. A task address space consists of one top-level address map; instead of references to memory objects directly, address map entries refer to second-level *sharing maps*. Changes in per-task attributes, such as protection and inheritance, are stored in the top-level map. Changes to the virtual memory represented by a map entry are reflected in the sharing map; for example, a *vm_write* operation into a region shared by more than one task would take place in the sharing map referenced by all of their task maps.

It is then the second-level sharing maps that refer to memory object structures. As an optimization, top-level maps may contain direct references to memory object structures if no sharing has taken place.

When a *vm_allocate_with_pager* call is processed, the Mach kernel looks up the given memory object port, attempting to find an associated internal memory object structure; if none exists, a new internal structure is created, and the *pager_init* call performed. Once a memory object structure is found, it is inserted into the (top-level) task

---

[9] The choice to prevent writing is made here to simplify the example. It may be more practical to allow the first client write access, and then to revoke it later.

address map. Note that the sharing semantics are different than in the inheritance case in that no sharing map is involved; an attempt to *vm_write* one mapping of the memory object would merely replace that mapping, rather than reflecting it in other tasks that have also mapped that memory object.

### 5.2. Virtual Memory Object Structures

An internal *memory object* structure is kept for each memory object used in an address map (or for which the data manager has advised that caching is permitted). Components of this structure include the ports used to refer to the memory object, its size, the number of address map references to the object, and whether the kernel is permitted to cache the memory object when no address map references remain.

A list of resident page structures is attached to the object in order to expediently release the pages associated with an object when it is destroyed.

### 5.3. Resident Memory Structures

Each *resident page* structure corresponds to a page of physical memory, and vice versa. The resident page structure records the memory object and offset into the object, along with the access permitted to that page by the data manager. Reference and modification information provided by the hardware is also saved here. An interface providing fast resident page lookup by memory object and offset (*virtual to physical table*) is implemented as a hash table chained through the resident page structures.

### 5.4. Page Replacement Queues

Page replacement uses several *pageout queues* linked through the resident page structures. An *active* queue contains all of the pages currently in use, in least-recently-used order. An *inactive* queue is used to hold pages being prepared for pageout. Pages not caching any data are kept on a *free* queue.

### 5.5. Fault Handling

The Mach page fault handler is the hub of the Mach virtual memory system. The kernel fault handler is called when the hardware tries to reference a page for which there no valid mapping or for which there is a protection violation. The fault handler has several responsibilities:

- *validity and protection* - The kernel determines whether the faulting thread has the desired access to the address, by performing a lookup in its task's address map. This lookup also results in a memory object and offset.
- *page lookup* - The kernel attempts to find an entry for a cached page in the *virtual to physical* hash table; if the page is not present, the kernel must request the data from the data manager.
- *copy-on-write* - Once the page has been located, the kernel determines if a copy-on-write operation is needed. If the task desires write permission and the page has not yet been copied, then a new page is created as a copy of the original. If necessary, the kernel also creates a new shadow object.
- *hardware validation* - Finally, the kernel informs the hardware physical map module of the new virtual to physical mapping.

With the exception of the hardware validation, all of these steps are implemented in a machine-independent fashion.

# 6. The Problems of External Memory Management

## 6.1. Types of Memory Failures

While the functionality of external memory management can be a powerful tool in the hands of a careful application, it can also raise several robustness and security problems if improperly used. Some of the problems are:

- *Data manager doesn't return data*. Threads may now become blocked waiting for data supplied by another user task, which does not respond promptly. The tight interconnection between IPC and memory management makes it difficult (or merely expensive) to determine whether the source of any memory is hostile prior to attempting to access that memory.
- *Data manager fails to free flushed data*. A data manager may wreak havok with the pageout process by failing to promptly release memory following pageout of dirty pages.
- *Data manager floods the cache*. This is rather similar in nature to a data manager which fails to free data, but is easier to detect and prevent.
- *Data manager changes data*. A malicious data manager may change the value of its data on each cache refresh. While this is an advantage for shared memory applications, it is a serious problem to applications which receive (virtual copied) data in a message.
- *Data manager backs its own data*. Deadlock may occur if a data manager becomes blocked in a page fault waiting for data which it provides.

Fortunately, there are several techniques that minimize these potential problems:

- A task may use the *vm_regions* call to obtain information about the makeup of its address space. While this enables a task to avoid deadlock on memory it provides, it does not prevent two or more tasks from deadlocking on memory provided by the others.
- The use of multiple threads to handle data requests also aids in deadlock prevention; one thread within a task may service a data request for another thread in that task.
- Server tasks which cannot tolerate changing data, or which must ensure that all of the necessary data is available, may use a separate thread to copy that data to safe memory before proceeding.

## 6.2. Handling Memory Failures

### 6.2.1. Analogy to Communication Failure

The potential problems associated with external data managers are strongly analogous to communication failure. This is actually not surprising since external data managers are implemented using communication. Solutions to communication failure problems are applicable to external data manager failure.

A task's address space becomes populated with memory objects in two basic ways: explicit allocation (*vm_allocate* or *vm_allocate_with_pager*) or receipt of data in an IPC message. In the first case, the source of the data is considered trusted (either the default pager, or a data manager known to the requestor). Use of the data received in a message can be viewed as attempting to receive that data explicitly; the only differences are the time and the granularity of the failure. The same options provided for communications failure may be applied to memory failures: a *timeout* period may be specified, after which a memory request is aborted; a *notification* message may be sent so that another thread can perform recovery; *wait* until the request is filled. Aborting a memory request after a timeout may involve providing (zero-filled) memory backed by the default pager, or termination of the waiting thread.

Destruction of a memory object by the data manager (i.e. deallocation of the memory object port) results in failure modes similar to destruction of a port: *notification* of those tasks previously having access to that object, and *abortion* of those requests in progress.

#### 6.2.2. The Default Pager

The default pager manages backing storage for memory objects created by the kernel in any of several ways: explicit allocation by user tasks (*vm_allocate*); shadow memory objects; temporary memory objects for data being paged out. Unlike other data managers, it is a trusted system component.

When cached data is written back to a data manager, those pages are temporarily moved to a new memory object which is destroyed when the data manager relinquishes that memory. If the data manager does not process and release the data within an adequate period of time, the data may then be paged out to the default pager. In this way, the kernel is protected from starvation by errant data managers.

Because the interface to the default pager is identical to other external data managers, there are no fundamental assumptions made about the nature of secondary storage. For example, the default pager may use local disk, network services, or some combination of the two. Furthermore, a new default pager may be debugged as a regular data manager.

#### 6.2.3. Reserved Memory Pool

In order to perform pageout operations, the kernel must send a message; to send a message, the kernel may need to allocate additional memory. To avoid deadlock, the kernel must reserve some amount of memory for itself to do pageout. Furthermore, data manager tasks may need to allocate memory in order to manage a pageout request. The kernel must allow for at least enough memory for the default pager to process a pageout, and for performance reasons wants to be able to provide enough memory for reasonable data manager tasks.

Many other operating systems have met with this problem. Often these systems solve the problem by using inside knowledge of the algorithms used to manage secondary storage. Unfortunately, the Mach kernel has no such knowledge; it must rely on a conservative estimate of the amount of memory required by the default pager to perform a pageout operation.

## 7. Multiprocessor Issues

As faster networks and more loosely connected multiprocessors are built, the distinctions between the use of shared memory and message passing as communication mechanisms are becoming blurred. There are three major types of multiple instruction, multiple data stream (MIMD) multiprocessors which have gained commercial acceptance:

1. uniform memory access multiprocessors (*UMAs*) with fully shared memory and nearly uniform memory access times for all data;
2. non-uniform memory access multiprocessors (*NUMAs*) with shared memory in which an individual CPU can access some memory rapidly and other memory more slowly; and
3. no remote memory access (*NORMAs*), message-based multiprocessors which communicate over an internal or external network.

UMA systems are the most prevalent in the commercial marketplace. They are the easiest of the three types to use because they are the most similar to traditional multiprogrammed uniprocessors. Shared memory is typically provided on some kind of shared bus with individual CPUs accessing the bus through a cache. Cache contents are synchronized. Write operations by one CPU will either update or flush the appropriate cache blocks visible to other CPUs. Access times appear nearly uniform, although, depending on the architecture, cache flushing can result in non-uniform memory access times for some algorithms. Examples of UMAs are the Encore MultiMax, Sequent

Balance, VAX 8300 and VAX 8800.

Some of the earliest multiprocessors were examples of NUMAs, including CMUs C.mmp [23] and CM* [11]. One problem with UMAs is that they often rely on a shared global bus that limits the maximum number of processors and memory units which can be attached. NUMAs typically avoid this problem of scale by associating a local memory with each CPU. This memory can be accessed by other CPUs, but only at a time cost greater than that for local memory access. The typical CPU-to-CPU interconnect is a communication switch with several levels of internal switching nodes. While such switches can be made to accommodate large numbers of communicating processors, they add the characteristic NUMA remote memory access delay. The difficulties in keeping cache contents consistent through such switches have led most NUMA designers to (1) not provide cache memory, (2) only allow caching for non-shared memory or (3) provide instruction-level cache control to be used by smart compilers. The BBN Butterfly [2] is an example of a commercial NUMA. Communication between CPUs in the Butterfly is by means of a *Butterfly Switch,* the complexity of which increases only as the logarithm of the number of processors. The Butterfly makes no use of cache memory and remote access times are roughly 10 times greater than local access times. Because NUMA architectures allow greater numbers of CPUs to be put in a single machine, many experimental very large multiprocessors are NUMAs; the 512-processor IBM RP3 is a recent example.

NORMAs are the easiest of the multiprocessors to build and, in a sense, have existed for as long as local area networking has been available. The typical NORMA consists of a number of CPUs interconnected on a high speed bus or network. Interconnection topology depends on the network technology used. The Intel HyperCube, for example, uses a multi-level cube structure with 10MHz Ethernet as the connecting technology. NORMAs are distinguished from NUMAs in that they provide no hardware supplied mechanism for remote memory access. Typically NORMAs can be much larger but also much more loosely coupled than NUMAs. For example, on the HyperCube, remote communication times are measured in the hundreds of microseconds versus five microseconds for a Butterfly and considerably less than one microsecond (on average) for a MultiMax.

All three types of multiprocessors can be made to support message passing or shared memory. Although some manufacturers [4, 9] have provided hardware support for message mechanisms, implementations of message communication on uniprocessors and tightly coupled multiprocessors typically use internal semaphores and data copy operations. It is possible to implement copy-on-reference [24] and read/write sharing [13, 14] of information in a network environment without explicit hardware support.

Just as Accent demonstrated that copy-on-write could be used for message passing in a uniprocessor, Li at Yale showed that a modified Apollo Aegis kernel could support applications which required read/write sharing of virtual memory data structures on a 10MHz token ring [14]. Network read/write sharing is accomplished using software techniques which parallel the hardware management of consistent caches in a multiprocessor. Cache blocks (in this case physical memory pages mapped by the memory mapping hardware of the Apollo workstations) are retrieved and cached as necessary from global memory. Multiple read accesses with no writers are permitted but only one writer can be allowed to modify a page of data at a time. An attempt by a reader to write previously read data causes a memory fault which converts a reader into a writer and invalidates all other page caches. A subsequent attempt to read by another workstation will cause the writer to revert to reader status (at least until the next write is performed). The efficiency of algorithms that use this form of network shared memory depends on the extent to which they exhibit read/write locality in their page references. Kai Li showed that multiple processors which seldom read and write the same data at the same time can conveniently use this approach.

Mach provides a collection of operating system primitives defined such that a programmer has the option of choosing to use either shared memory or message-based communication as the basis for the implementation of a multithreaded application. Depending on the desired style of the application and the kind of multiprocessor or network available, message passing can be implemented in terms of memory management primitives or memory management can be implemented in terms of communication primitives.

## 8. Applications

The implications of message/memory duality in Mach extend beyond the issue of multiprocessor support. There is evidence that the efficient emulation of operating system environments such as UNIX can be achieved using the Mach primitives. Process migration can be supported more effectively by relying on network copy-on-reference of process data. The interaction of message and memory primitives can also be an important tool in the design and implementation of transaction and database management facilities. Work is also being done to allow AI knowledge structures to be built, maintained and accessed using these techniques.

### 8.1. Emulating Operating System Environments

During the 1960's the notion of a virtual machine base for operating systems development became commercially popular. That popularity faded in the research community due to the complexities of truly virtualizing a wide range of devices and hardware functions. Alternative systems such as UNIX [18] were developed which provided a simple set of basic abstractions which could be implemented on a range of architectures.

Today, the concept of an extensible operating system is once again gaining acceptance -- this time as the solution to the unconstrained growth of UNIX. During the last 20 years operating systems and their environments have undergone dramatic expansion in size, scope and complexity. The Alto operating system [22], a workstation operating system of the early 70's, occupied approximately 19K bytes, including code and data. A typical UNIX implemention on a modern workstation such as a MicroVAX or SUN can consume over 1.5 megabytes of storage before a single user program is executed.

This increase in complexity has been fueled by changing needs and technology which have resulted in UNIX being modified to provide a staggering number of new and different mechanisms for managing objects and resources. UNIX has become a dumping ground for new features and facilities because key requirements for many of these new facilities can be found only in kernel state:

- easy access to user process state,
- access to virtual and physical memory objects,
- device access,
- the ability to run several processes which communicate through shared memory, and
- efficient context switching and scheduling.

One of the goals of Mach is to provide an extensible kernel basis upon which operating system environments such as UNIX can be built.

Emulation of UNIX-like system environments can be implemented using a variety of techniques. Generic UNIX system calls can be implemented outside of the Mach kernel in libraries and server tasks. For example, UNIX filesystem I/O can be emulated by a library package that maps open and close calls to a filesystem server task. An open call would result in the file being mapped into memory. Subsequent read and write calls would operate directly on virtual memory. The filesystem server task would operate as an external pager, managing the virtual memory corresponding to the file. Shared process state information can be passed on to child processes using inherited shared memory.

### 8.2. Copy-On-Reference Task Migration

One of the thorniest problems of task migration is the handling of large address spaces. Edward Zayas showed that migration could be performed efficiently using copy-on-reference techniques [24]. The task migration service can create a memory object to represent a region of the original task's address space, and map that region into the new task's address space on the remote host. The kernel managing the remote host treats page faults on the newly-migrated task by making paging requests on that memory object, just as it does for other tasks.

The generality of the external memory management allows for a wide variety of migration strategies. To reduce faulting overhead, the migration manager may provide some data in advance for tasks with predictable access patterns. This pre-paging can proceed while the newly-migrated task begins to run. Alternatively, the migration manager can respond to requests on demand for unpredictable or excessively large tasks.

### 8.3. Database Management: Camelot

Camelot is a transaction processing system being implemented on Mach [21]. Camelot provides support for distributed transactions on user-defined objects. In Camelot, servers maintain permanent objects in virtual memory backed by the Camelot disk manager. Camelot uses the write-ahead logging technique to implement permanent, failure-atomic transactions. When the disk manager receives a *pager_flush_request* from the kernel, it verifies that the proper log records have been written before writing the specified pages to disk.

Transaction systems reap many benefits from the ability to manage memory objects:

- Camelot clients can access data easily and quickly by mapping memory objects into their virtual address spaces.
- Camelot clients do not have to implement their own page replacement algorithms.
- Clients need not reserve fixed sized physical memory buffers. The amount of physical memory devoted to each client varies dynamically with overall system load.
- Recoverable data can be written directly to permanent backing storage without first being written to temporary paging storage.

By using an external interface, Camelot can benefit from these advantages without having to modify the operating system to provide specialized support [5]. Mach manages the physical memory cache while the Camelot disk manager guarantees that the write-ahead log protocol is followed.

### 8.4. AI Knowledge Bases: Agora

A parallel can be made between the potential uses of external pagers in transaction and database management and their use in the implementation of AI knowledge structures. The speech research group at CMU is currently engaged in a project to build a distributed speech understanding system called Agora [3]. This work is being done on Mach and currently makes use both of Mach memory sharing and message passing.

Both communication and memory sharing are used to implement a shared blackboard structure in which hypotheses are placed and evaluated by multiple cooperating agents. The blackboard physically resides on a multiprocessor host. All accesses to the blackboard are through a procedural interface that determines if shared memory or communication must be used. Agents use shared memory to directly modify the blackboard. Message passing is used between loosely coupled components of the system that collect data, perform low level signal processing, and display results. The total system distributes itself among a collection of machines consisting of a VAX 8200 with four CPUs and a number of MicroVAX IIs and IBM RT PCs interconnected via several Ethernets and IBM token rings.

## 9. Performance

One of the key benefits to Mach's external memory management is that it allows user provided objects to take advantage of the same kind of physical memory caching which traditionally has only been available to kernel-supplied secondary storage. A user program can, for example, create a memory object which is used to represent a data array and provide access to that array to many other programs through a server message interface. The clients of such a service would only have to exchange a single message with the server to get access to the array and, if other clients had already referenced the data of the array, the physical memory cache of the array would be directly accessible to the client with no further message traffic.

A large, well-managed memory cache of this kind can be a powerful aid to improving the performance of an operating system. For example, traditional UNIX implementations manage a cache of recently accessed file data blocks. This cache, which is normally 10% of physical memory in a Berkeley UNIX system, is accessed by user programs through read and write kernel-to-user and user-to-kernel copy operations. In contrast, Mach uses the bulk of its physical memory as a cache of secondary storage data pages. The effect of this kind of caching on the performance of UNIX and its traditional suite of application programs is dramatic. Compilation of a small program cached in memory on a SUN 3/160 running Mach is twice as fast as when running the more conventional SunOS 3.2 operating system [17]. In a large system compilation, the total number of I/O operations can be reduced by a factor of 10 [1].

## 10. Status

All of the Mach facilities described in this paper have been implemented -- with the external memory management facility the most recent and most experimental addition. A production version of Mach is in use on over 200 workstations and large timesharing systems within the CMU Department of Computer Science. It is also being used on dozens of machines outside the Department, including systems at BBN, the IBM Hawthorne Laboratory, the CMU Software Engineering Institute and the Los Alamos Research Laboratory.

As of this writing, Mach runs on more than a dozen computer systems including the VAX family of uniprocessors and multiprocessors, the IBM RT PC, the SUN 3, the 16-processor Encore MultiMax, and the 26-processor Sequent Balance 21000. Work is proceeding on implementations for several other computer systems.

Versions of Mach for all machine types are built from the same source with machine specific code isolated to machine dependent directories[10]. All VAXen run the same binary image of the kernel. Mach currently supports the X window manager and is binary compatible with 4.3bsd UNIX on VAX architecture machines.

The paging interface component is still under development:

- *Default Pager*. The default pager is currently running as a separate kernel-state task which uses the memory object interface. To manage secondary storage, it uses Unix inodes and the Unix buffer pool. Future plans include eliminating use of the buffer pool, and instead allowing that memory to be used for the global virtual memory cache.
- *User Pager Tasks*. The implementation already supports both pagein and pageout requests being handled by user-state pager tasks; however, cache consistency primitives are not fully implemented.
- *Special Conditions*. Currently, initialization and termination are implemented as described; however, handling of pager failures is not yet done.

---

[10] Some support for manufacturer specific UNIX features has been kept in machine independent files to accomodate binary compatibility with various manufacturers' non-4.3bsd UNIX environments.

Mach is being released externally to interested researchers. The second release of Mach was made in April, 1987. The next release is scheduled for the end of October, 1987.

## References


[1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[2] BBN Laboratories.
*Butterfly Parallel Processor Overview*.
Technical Report, BBN Computer Company, Cambridge, MA, June, 1985.

[3] Bisiani, R., Alleva, F., Forin, A. and Lerner, R.
Agora: A Distributed System Architecture for Speech Recognition.
In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, April, 1986.

[4] ELXSI Computer, Inc.
*System Programmer's Reference Manual*
ELXSI Computer, Inc., 1983.

[5] Eppinger, J.L., and Spector, A.Z.
*Virtual Memory Management for Recoverable Objects in the TABS Prototype*.
Technical Report CMU-CS-85-163, Carnegie-Mellon University, December, 1985.

[6] French, R.E., R.W. Collins and L.W. Loen.
System/38 Machine Storage Management.
*IBM System/38 Technical Developments, IBM General Systems Division* :63-66, 1978.

[7] Gupta, A.
*Parallel Production Systems*.
PhD thesis, Carnegie Mellon University, May, 1986.

[8] Hornig, D.A.
*Automatic Partitioning and Scheduling on a Network of Personal Computers*.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, November, 1984.

[9] Kahn, K.C. et al.
iMAX: A Multiprocessor Operating System for an Object-Based Computer.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 127-136. ACM, December, 1981.

[10] Jones, A.K.
The Object Model: A Conceptual Tool for Structuring Systems.
*Operating Systems: An Advanced Course.*
Springer-Verlag, 1978, pages 7-16.

[11] Jones, A.K., Chansler, R.J., Durham, I.E., Schwans, K., and Vegdahl, S.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings of the 7th Symposium on Operating System Principles*, pages 117-129. ACM, December, 1979.

[12] Jones, M.B., Rashid, R.F., and Thompson, M.R.
*Sesame: The Spice File System*.
Internal Document, Department of Computer Science, Carnegie-Mellon University, October, 1982.

[13] Leach, P.L., P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf.
The Architecture of an Integrated Local Network.
*IEEE Journal on Selected Areas in Communications* SAC-1(5):842-857, November, 1983.

[14] Li, K. and Hudak, P.
Memory Coherence in Shared Virtual Memory Systems.
In *5th Symposium on Principles of Distributed Computing*. 1986.

[15] Rashid, R.F. and Robertson, G.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the 8th Symposium on Operating System Principles*, pages 64-75. December, 1981.

[16] Rashid, R.F.
From RIG to Accent to Mach: The Evolution of a Network Operating System.
In *Proceedings of the ACM/IEEE Computer Society 1986 Fall Joint Computer Conference*. ACM, November, 1986.

[17] Rashid, R.F., Tevanian, A., Young, M.W., Golub, D.B., Baron, R.V., Black, D.L., Bolosky, W., and Chew, J.J.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, October, 1987.

[18] Ritchie, D.M. and Thompson, K.
The Unix Time-Sharing System.
*Communications of the ACM* 17(7):365-375, July, 1974.

[19] Satyanarayanan, M., et.al.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 35-50. ACM, December, 1985.

[20] Spector, A.Z., Butcher, J., Daniels, D.S., Duchamp, D.J., Eppinger, J.L., Fineman, C.E., Heddaya, A., Schwarz, P.M.
Support for Distributed Transactions in the TABS Prototype.
In *Proceedings of the 4th Symposium on Reliability In Distributed Software and Database Systems*. October, 1984.
Also available as Carnegie-Mellon Report CMU-CS-84-132, July 1984.

[21] Spector, A.Z.
Distributed Transaction Processing and the Camelot System.
In Yakup Paker (editor), *NATO Advanced Study Institute - Computer and Systems Sciences*. Volume : *Distributed Operating Systems: Theory and Practice*. Springer-Verlag, 1987.
Also available as Carnegie-Mellon Report CMU-CS-87-100, January 1987.

[22] Thacker, C.P., et al.
Alto: A personal computer.
In *Computer Structures: Readings and Examples*. McGraw-Hill, 1980.
Edited by D. Siewiorek, C.G. Bell, and A. Newell, second edition.

[23] Wulf, W.A., Levin, R., and Harbison, S.P.
*Hydra/C.mmp: An Experimental Computer System.*
McGraw-Hill, 1981.

[24] Zayas, E.R.
*The Use of Copy-On-Reference in a Process Migration System*.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, January, 1987.

# Table of Contents

# List of Tables