THE UNIVERSITY OF CALGARY

Efficient Data Passing in Distributed Systems

by

Paul Robert Milligan

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JANUARY, 1992

ISBN 0-315-75264-5

THE UNIVERSITY OF CALGARY

Faculty of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Efficient Data Passing in Distributed Systems," submitted by Paul Robert Milligan in partial fulfillment of the requirements for the degree of Master of Science.

R. Vasudevan, Supervisor
Computer Science

Brian Unger
Computer Science

Paul Kwok
Computer Science

Laurence E. Turner
Electrical Engineering

Date January 29, 1992

Abstract

Simple data-passing modes are proposed for interprocess communication (IPC) in distributed systems. The modes are *duplicate*, *move* and *dynamic-share*. IPC incorporating these data-passing modes can be implemented efficiently on networks of shared memory multiprocessors by taking advantage of memory management hardware.

The motivation for conducting this research is a desire for a data-passing IPC model that is competitive in terms of performance with memory-sharing IPC models. However, the data-passing mode approach also enriches the semantics of data-passing IPC because the intended access to the memory containing the passed data can be specified.

The primary problem addressed by this research is how to define data-passing IPC operation semantics for distributed systems so that they can be implemented efficiently. Efficient implementation includes minimising data copying.

A new synchronous message-passing IPC model called Regions is presented. The purpose of Regions is to provide IPC operations that support the duplicate, move and dynamic-share data-passing modes. Regions supports data-passing between separate processes where memory is not shared by default but where memory sharing can be established dynamically.

Regions has been implemented on Sun 3 workstations and a BBN Butterfly multiprocessor. These implementations are used to measure and analyse the elapsed time performance of the IPC operations.

We conclude that the semantics and efficiency of data-passing IPC operations can be significantly improved by supporting the data-passing modes.

Acknowledgements

I thank my supervisor, Vasu, for his unconditional support. If there is any significant contribution in this thesis it is due to his patience and his commitment to clarity, simplicity and quality. It has been a pleasure and an honour to work with him.

Graham Birtwistle also deserves my thanks for allowing Vasu to supervise me while on leave by performing the local administrative tasks of interrim supervisor.

I thank my wife, Myreille, for her love, support and understanding. She has had to share my attention with this thesis from the moment we met. Myreille and the child within her provided the incentive I needed to complete this thesis.

I thank the people responsible for awarding me a NSERC scholarship because it provided the incentive I needed to initiate this thesis.

I thank my many friends and relatives for encouraging me and believing in me. They helped me overcome my feelings of inadequacy. I leaned heavily on the little things they have said through the years. I especially want to acknowledge my parents, my brother Patrick, my sister Sheila, and my friends Theo vanKalleveen, Hatem Zaghloul, Valerio Franceschin, Hugo Graumann, Micheal Belenstien, Kevin Jewell, Rick Farmer, Cliff Nelson, Rod Randall, Barry Thate, Roy Colin, Linda Riddle, and Martin Fromme.

I thank the members of the Computer Science Department for providing excellent resources. I appreciate the assistance that John Lewis and Istvan Hernadi provided with the implementation. I also appreciate the assistance of Jules Bloomenthal, Mike Bonham, Alan Dewar, Camille Sinanan, Bev Frangos, Bruce MacDonald, Gerald Vaselenak, Tim Bliek, Brian Scowcroft, David Hankinson and Robert Fridman.

I thank BBN Advanced Computers Inc. and Jade Simulations International Corp. for making the GP1000 Butterfly multiprocessor available. BBN donated a disk drive to allow me to continue the implementation and analysis when the department was not able to provide a replacement drive.

I thank Willowglen Systems Ltd and SRDG for jointly developing the W System. Their implementation provided a base for my research. I appreciate the effort that was required to develop the system. I also appreciate the timely loans of hardware from Willowglen Systems.

Finally, I want to thank the people who participated in the Forum and other WEA courses and seminars with me. That experience empowered me to bring possibility to my work and to go beyond what normally stops me.

To my father, James Robert Milligan,

and

my wife, Myreille Milligan.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Future computer systems are likely to be networks of shared memory multiprocessors and uniprocessors. A problem with designing interprocess communication (IPC) models for these systems is how to efficiently pass data between processes. A new solution based on three data-passing modes is proposed. These modes are incorporated in a new data-passing IPC model for distributed systems.

In distributed systems the coupling between two processors is either tight (the processors share access to physical memory) or loose (the processors do not share access to physical memory). Tightly-coupled processors can use memory to communicate while loosely-coupled processors can only use physical communication channels to communicate. IPC models for distributed systems can hide the coupling of processors allowing multiprocess application programs to be specified independent of the underlying hardware. Two types of IPC models are discussed.

Data-passing IPC models allow processes in separate memory-access *contexts* to communicate. A memory-access context defines a set of memory-cells that a process can access. There are several advantages of using separate contexts. First, processes are protected from memory-access interference which occurs when incorrect results are produced because a process accesses memory at an improper time. Second, data-passing IPC operations can be designed which are semantically transparent with respect to the coupling of processors. Finally, separate contexts support modular structure and failure isolation.

Shared-variable (or memory-sharing) IPC models support processes executing in the same context. An advantage of this approach is that data can be passed

by reference. Another advantage is data need not be explicitly passed from one process to another. A process can voluntarily coordinate its access to shared memory with interprocess synchronisation operations [Dijkstra 68, Hoare 74]. However, memory-sharing IPC models do not provide protection from memory-access interference. Semantically transparent access to memory shared between processes executing on tightly-coupled or loosely-coupled processors has been demonstrated [Li 86].

The motivation for conducting this research is a desire for a data-passing IPC model that is competitive in terms of performance with memory-sharing IPC models when the communicating processes are executing on tightly-coupled processors[1].

The primary problem addressed by this research is how to define data-passing IPC operation semantics for distributed systems so that they can be implemented efficiently. Efficient implementation includes minimising data copying.

1.1 The Goals: Enriched Efficient Data-Passing

The goals of this research are the following.

- Demonstrate that data-passing between processes executing in separate contexts can be implemented efficiently when the processes are executing on tightly-coupled processors.
- Develop an IPC model for distributed systems that provides simple data-passing operations which support semantics enriched with memory access information.

Design considerations for the IPC model are as follows.

- It must also be possible to implement the data-passing operations efficiently for processes executing on loosely-coupled processors.

[1]This includes the trivial case where the communicating processes are executing on the same processor (via time multiplexing).

- The IPC model must support run-time enforced separate contexts so that memory-access interference can be controlled.

1.2 The Approach: Data-Passing Modes

Simple data-passing modes are proposed that allow the implementation to avoid unnecessary data copying by taking advantage of memory management unit (MMU) hardware. The modes are duplicate, move and dynamic-share. They specify the access that the passing process has to the memory containing the passed data.

Data are passed from a source context to a destination context. If separate copies of the data are required in the source and destination contexts, then the data are passed using the duplicate mode. This is analogous to photocopying a sheet of paper and giving it to another person. If the same copy of the data is required in the source and destination contexts then the data are passed using the dynamic-share mode (memory sharing is established at the time the data are passed). This is analogous to one person allowing another person to simultaneously use the same sheet of paper. If the data are no longer required in the source context then the data are passed using the move mode. There are two possibilities: the memory-cells containing the moved data are or are not shared. If they are not shared then move is analogous to sending a sheet of paper by mail. If they are shared then move is analogous to one person (the source) being replaced by another person (the destination) in a group of people who are simultaneously using the same sheet of paper.

Data-passing operations can be enriched with these data-passing modes. Existing data-passing IPC models support data-passing operations that pass data *by value*. A copy of the data becomes accessible to the process executing in the destination context. The memory containing the original data either remains accessible to the process executing in the source context (the duplicate mode) or it is no longer ac-

cessible to that process (the move mode). Existing data-passing operations can be enriched with the dynamic-share data-passing mode which allows data to be passed *by binding*. The memory containing the original data becomes accessible to the process executing in the destination context. This memory either remains accessible to the process executing in the source context (the dynamic-share mode) or it is no longer accessible to that process (the move mode).

Dynamic memory sharing between separate contexts is not novel but the dynamic-share data-passing mode is. This mode establishes memory sharing. This memory sharing differs from the memory sharing provided by shared-variable IPC models because the processes are executing in separate contexts. It is possible (see Section 3.2.3.1) to configure separate contexts so that references can be passed between processes that share memory (the shared data can be passed *by reference*). Once data has been passed with the dynamic-share mode then explicit data-passing is no longer required because the processes share access to the memory containing the data.

The dynamic-share mode is useful for allowing two or more (possibly unrelated) processes to concurrently access specific memory. For example, the dynamic-share mode can be used to pass processes access to the data structures that are used to update a display screen (rather than passing data with the duplicate or move mode to a process that updates the screen on their behalf). Each process is trusted to update its part of the screen without interfering with other parts of the screen. The screen can be updated concurrently without the overhead of data-passing IPC operations because memory sharing has been established. However, a process can modify part of the screen that is not allocated to that process. Applications programmers have the choice to prevent memory-access interference by not using the dynamic-share data-passing mode or to avoid explicit data-passing operations (involving the duplicate or

move modes) by using the dynamic-share mode once to establish memory sharing[2].

Efficient implementation of the data-passing modes requires hardware with the following features.

- Process addresses can be dynamically bound to physical memory.
- Read and write operations on memory via specific addresses can be intercepted and restarted by the processor.

The first feature allows processes executing in separate contexts to share memory. It also allows data to be passed between processes executing on tightly coupled processors without copying the data. The second feature supports the enforcement of access semantics to memory shared between processes executing on loosely coupled processors. It also allows copying of data (passed with the duplicate mode) to be deferred until an attempt is made to modify either of the duplicates. If the duplicates are never modified or one of the duplicates is deleted before an attempt is made to modify the other duplicate then copying is avoided.

1.2.1 Regions: An IPC Model that Incorporates the Modes

The data-passing modes can be supported by data-passing models used for IPC such as asynchronous message-passing, synchronous message-passing and remote procedure call [Accetta 86, Birrell 84, Cheriton 88, Seitz 88].

A new IPC model called Regions is presented in this thesis. Regions is based on synchronous message passing. The design objectives of Regions are to show that the data-passing modes can be supported with simple IPC operations and can also be implemented efficiently. Regions supports the separate-context semantics of

[2]Establishing memory sharing with the dynamic-share mode as opposed to passing data with the duplicate or move modes should reduce elapsed time overhead when the processes are executing on tightly-coupled processors. However it will increase elapsed time overhead when the processes are executing on loosely-coupled processors and they are contending for the data stored in the shared memory.

data-passing IPC models and it supports the shared-memory semantics of memory-sharing IPC models because it supports the dynamic-share data-passing mode. These semantics can be implemented efficiently on networks of multiprocessors.

Shared-memory semantics imply that more than one process can access common (shared) memory with read and write operations. It is possible to support the semantics of read and write operations to shared memory with data-passing IPC operations [Cheriton 86]. However, an implementation of Regions can support read and write operations efficiently with memory referencing processor instructions. Support for read and write operations on shared memory by processes executing on loosely coupled processors has been previously demonstrated [Spector 82, Li 86].

By default, memory accessible via one context is not accessible via another context. However, the dynamic-share data-passing mode can cause memory to be accessible via separate contexts. Processes cannot share contexts.

The Regions synchronous message-passing IPC model is defined in terms of messages and DRegions. A *message* is a 64 byte collection of data. A *DRegion* is a variable sized collection of memory. A client process sends a request message to a server process and waits for a response message. The server receives the request message, performs the request and returns a response message. The server can pass DRegions to and from the client before returning the response message.

Messages are passed by copying the data from memory accessible via one context to memory accessible via another context. DRegions can be passed using the duplicate, move or dynamic-share data-passing mode. The choice of data-passing mode depends on the expected access to the memory containing the passed data.

The implementation of the DRegion passing operations takes advantage of shared physical memory when it is available and it takes advantage of MMU hardware.

1.3 Related Work

The use of data-passing modes is a novel approach to minimise data copying and thereby make IPC efficient. However the concepts and implementation techniques are based on ideas used in several existing systems that take advantage of MMU hardware. Alternatively, some existing IPC models avoid data-passing by allowing processes to share access to memory.

1.3.1 Message-Passing Distributed Operating Systems

Related systems that provide data-passing IPC models are now discussed.

1.3.1.1 V System

> "The V distributed system is an operating system designed for a cluster of computer workstations connected by a high-performance network." [Cheriton 88]

The V System is a well known distributed system that supports synchronous message-passing.

Separate-context semantics are supported. Memory accessible via one context is not accessible via any other context. Data copying is minimised by allowing multiple processes (called a team of processes) to share a context. However, memory sharing is static.

Data is passed between processes executing in separate contexts by copying it from the memory accessible via one context to the memory accessible via another context. There is a copy of the passed data in the source and destination context even when both copies are not needed and the processes are executing on tightly coupled processors.

The V System IPC was not designed to take advantage of MMU hardware. The emphasis appears to be on efficient data-passing between processes executing on

loosely coupled processors [Cheriton 83].

1.3.1.2 Mach

> "Mach is a multiprocessor operating system kernel ..." [Accetta 86]
> "Mach is designed to support computing environments consisting of networks of uniprocessors and multiprocessors." [Baron 88]

Mach supports data-passing and dynamic memory sharing between separate contexts. In addition, the implementation takes advantage of MMU hardware. The subsequent description is intended to emphasise the complexity of the IPC model.

A Mach task is an abstraction that includes a memory-access context. Multiple processes (called threads) can execute in a task. By default, memory accessible via one task is not accessible via another task. However, memory can be accessible via separate tasks that (1) are created from a common ancestor task using the inheritance feature or (2) are using a external memory management server to share memory.

Several types of IPC are supported including asynchronous message-passing, synchronous message-passing, and dynamic memory sharing. IPC is defined in terms of memory objects, messages and ports. A memory object is a collection of memory. A message is a variable sized collection of data consisting of a fixed sized header and optional references to memory objects. A port is an object consisting of a queue of messages. Send and receive operations can be invoked on a port.

Dynamic sharing of memory objects between tasks is supported with the external memory management feature [Young 87]. An interface consisting of 11 operations is defined that allows a process to manage access to and consistency of memory objects. It is possible for processes executing on loosely coupled processors to use this feature to share memory.

Although the model is based on only five abstractions, there are many operations and some of the operations have several options and exceptions.

Data are passed by copying or duplicating. Message headers are copied from the memory accessible via one task to the memory accessible via another task. Memory objects are duplicated in the destination task.

The implementation of multiple threads per task, shared memory objects through inheritance, and shared memory objects through external memory management processes takes advantage of shared physical memory and MMU hardware. The implementation of memory object duplication and process creation by forking is based on the deferred data copying technique called copy-on-write. This implementation technique also takes advantage of MMU hardware.

1.3.1.3 UNIX System V

> "The UNIX System V IPC package consists of three mechanisms. Messages ..., shared memory allows processes to share parts of their virtual address space, ..." [Bach 86]

UNIX System V provides a simple method of dynamic memory sharing between separate contexts.

A process obtains a *handle* to a shared memory region through a form of IPC such as message-passing, pipes, or inheritance. The process can use the handle to make the shared memory region accessible in the process's context. The implementation takes advantage of MMU hardware. The current implementation is restricted to a single machine.

1.3.1.4 Dash

> "The DASH ... system's major design goals are centered in three areas 1) IPC performance, 2) global architecture, and 3) local architecture." [Anderson 88]

In DASH data copying is minimised by taking advantage of MMU hardware and one of the design goals is efficient data-passing between processes executing on

tightly-coupled processors.

Separate-context semantics are supported. Data copying is minimised by allowing multiple processes to execute in a context called a virtual address space (VAS). The memory accessible via one VAS is not accessible via another VAS.

Asynchronous and synchronous message-passing IPC are supported. IPC is defined in terms of an IPC region, IPC pages, messages and message-passing objects (MPOs). There is an IPC region in every VAS that is intended for efficient data-passing. An IPC page is a sub-region (virtual page) of the IPC region. A message consists of a header and a collection of references to IPC pages. An MPO is an object consisting of a queue of messages. Send and receive operations can be invoked on MPOs.

The IPC operation semantics are complicated with options and restrictions that are intended to improve the performance of data-passing between VASs where the communicating processes are executing on tightly-coupled processors.

Data are passed by copying or moving. Message headers are copied from the memory accessible via one VAS to the memory accessible via another VAS. IPC pages are moved from the source VAS to the destination VAS.

The implementation of multiple processes per VAS takes advantage of shared physical memory. It is not possible for processes executing in separate VASs to share access to memory. The implementation of passing IPC pages by moving takes advantage of MMU hardware.

1.3.1.5 Reactive Kernel

> "The Reactive Kernel (RK), a new node operating system for medium-grain multicomputers, ..." [Seitz 90]

The RK implementation on the Ametek 2010 [Seitz 88] takes advantage of MMU hardware to minimise data copying.

Separate-context semantics are supported. Memory accessible via one context is not accessible via any other context. Processes do not share contexts.

IPC is based on asynchronous message-passing. A message is a dynamically allocated collection of data. If a process sends a message then that message is deallocated from that process's context and the process continues executing. When a process receives a message then the message is allocated in that process's context. There are two receive operations: one causes the process to wait until a message arrives while the other returns immediately allowing the process to continue if there are no messages. Sending a message is like deallocating a message and receiving a message is like allocating a previously initialised message.

Messages are passed by moving the message from the source context to the destination context. If the source process needs a copy of the message it must copy the message before sending it.

The implementation does not take advantage shared physical memory. However, messages are allocated on page boundaries so that they can be allocated in the destination context without copying. On the Ametek 2010, it is faster to send a message from one node to another over the communications channel than it is to have the processor copy the message from one location to another in memory.

1.3.2 Shared-Variable Distributed Operating Systems

Related systems that provide memory-sharing IPC models are now discussed.

1.3.2.1 Cedar

> "Cedar is a large project concerned with developing a programming environment that is powerful and convenient for the building of experimental programs and systems." [Birrell 84]

Cedar data-passing operations can be enriched with the data-passing modes and Cedar provides an alternative method of memory access control.

Separate-context semantics are supported where memory accessible via one context is not accessible via any other context. All processes executing on a machine normally share a context (although theoretically, multiple versions of portable Cedar [Atkinson 89] should be able to run on top of UNIX on a single machine). Memory access control between separate processes executing in the same context is supported by run time enforced data typing. All processes are written in a strongly typed language (that is also called Cedar).

A remote procedure call (RPC) feature is provided that allows a process executing in one context to execute a procedure in another context. RPC is not a method of IPC but it can be used to allow processes executing in separate contexts to communicate.

Data (RPC arguments and return values) are passed by copying the data from memory accessible via one context to memory accessible via another context.

The RPC implementation does not take advantage of physically shared memory or MMU hardware. Birrell and Nelson discarded the possibility of emulating shared memory between loosely-coupled processors when they implemented RPC for Cedar because they were not willing to undertake the research.

1.3.2.2 IVY and Similar Systems

> "IVY is a shared virtual memory system developed for experimental purposes." [Li 86]

IVY is the result of research into emulating shared memory between loosely-coupled processors.

A single context is supported across a network of workstations. Processes executing on the same or separate workstations share access to common memory.

Data-passing IPC is not required because the all processes statically share memory. Eventcounts [Reed 79] are supported for interprocess synchronisation. A method of memory access control between separate processes is not provided.

The implementation takes advantage of MMU hardware to ensure that a process reads the data most recently written to a *virtual* memory-cell.

IVY demonstrated that shared memory semantics can be supported efficiently across a communication channel. Impressive improvements in the performance of parallel algorithms were demonstrated by increasing the number of workstations that processes execute on. However part of the improvement was due to a reduction in disk activity involving swapping pages of memory.

The performance of parallel algorithms that do contend for shared data can be improved by dynamically changing the implementation technique used to share data. This was demonstrated on systems where the access time to separate memories is not uniform[3] [Bolosky 91].

> "Munin is a system that allows programs written for shared memory multiprocessors to be executed efficiently on distributed memory machines." [Bennett 90]

The Munin system attempts to further improve performance by statically assigning a shared type to each shared variable. The sharing type is a hint that Munin uses when deciding how to share the variable across a communication channel.

1.4 Contributions

The elapsed time of a data-passing IPC operation is a measure of implementation efficiency. Low elapsed times are achieved for data-passing between processes executing in separate contexts on tightly-coupled processors. The elapsed times for the

[3]Such systems are commonly called NUMA (non-uniform memory access) architectures.

Regions IPC operations that take advantage of MMU hardware are lower than times reported for similar systems. The synchronous message-passing times are comparable with the best times reported for similar data-passing operations [Bershad 89] and they are better than the times reported for other message-passing systems.

A simple way of providing dynamic memory sharing and data-passing between separate contexts with a single operation is presented. This approach is simpler than hybrid approaches provided by other systems where separate operations are required for dynamic memory sharing and data-passing.

Data-passing modes are proposed that enrich the semantics of data-passing operations by allowing the intended use of the passed data to be specified. The data-passing operations can be implemented efficiently because the enriched semantics specify when separate copies of the data are required.

1.5 Overview

The data-passing modes and related concepts are explained in Chapter 2.

In Chapter 3 the Regions IPC model is explained. IPC operations that support the dynamic-share, move and duplicate data-passing modes are explained.

In Chapter 4 an implementation of the Regions IPC primitives is described using an object-based paradigm. Implementations on Sun 3 workstations and BBN Butterfly multiprocessors are discussed.

The elapsed time performance of the Regions IPC primitives on a Sun 3/75 and GP1000 Butterfly is presented and analysed in Chapter 5.

Conclusions and further work are presented in Chapter 6.

Chapter 2

Data-Passing Modes

A *data passing mode* is a way of passing data from a source container to a destination container.

Because the modes are not restricted to data-passing alone a more generalised definition is presented. This definition is used to define three data-passing modes: dynamic-share, move, and duplicate. The copy data passing mode is also defined and it is distinguished from the duplicate data passing mode. Practical issues of specifying the data to be passed and passing groups of data are discussed because they are relevant to IPC performance.

2.1 Dynamic-Share, Move and Duplicate Passing Mode Definitions

The following notation is used to define the dynamic-share, move and duplicate passing modes. A and B are containers. A contains the entity E.

Dynamic-share causes E to be in B. E is shared in A and B.

Move causes E to be in B and not be in A. E is moved from A to B.

Duplicate causes a newly created entity E′ to be in B where the only difference between E and E′ is that they are separate entities. E is duplicated in B.

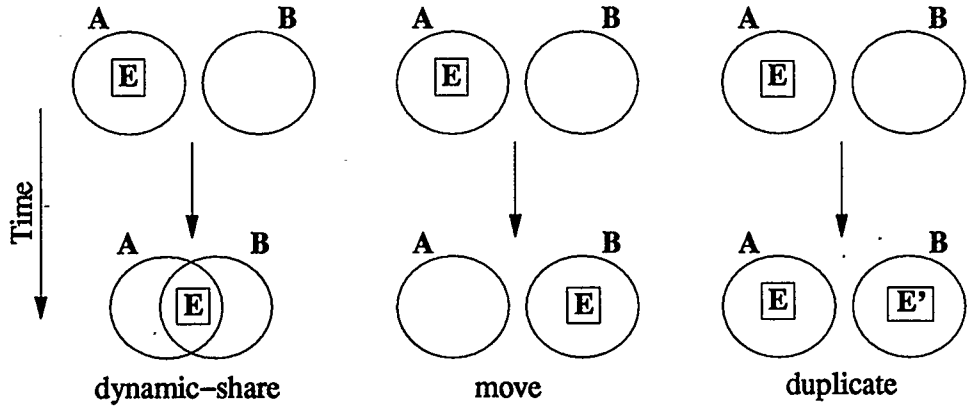

Figure 2.1: Passing modes: entity not shared.

An example of each passing mode is shown in Figure 2.1 where the entity is in only one container before passing. Figure 2.2 shows passing mode examples where the entity is shared in containers A and C before passing it from A to B.

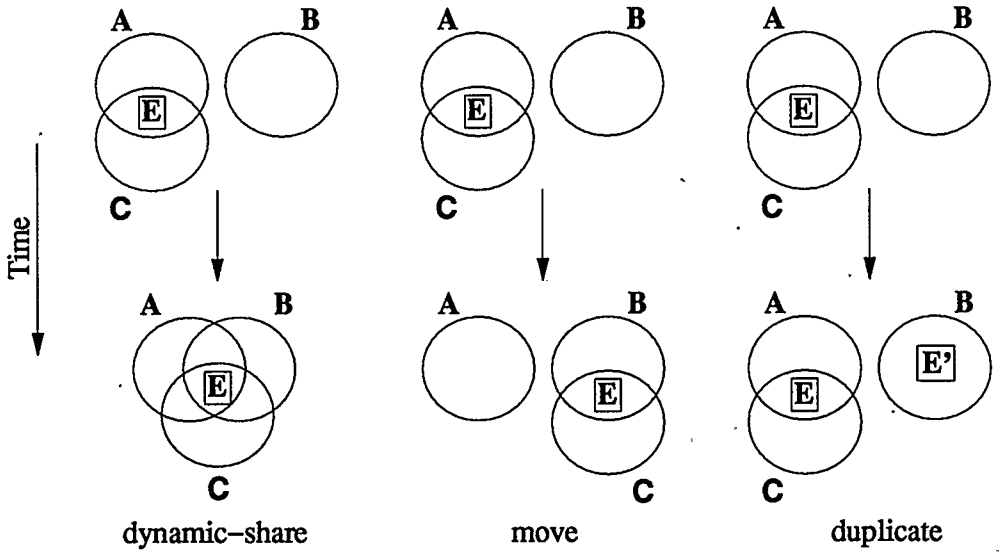

Figure 2.2: Passing modes: entity shared.

2.2 Passing Modes Involving Datum-Containers

These passing modes can be used to pass data and containers of data. This is explained by defining objects that are containers for data and objects that are containers for containers of data.

2.2.1 Object Notion

To simplify the explanation the notion of an object is used. An object has an *interface* that specifies the operations that can be invoked on that object. The arguments and return values of each operation are also specified in the interface. Every interface provides operations called create and delete. The create operation allocates and initialises an instance of an object. The delete operation deallocates an instance of an object.

2.2.2 Datum-Container Definition

A *datum-container* is a container for a datum[1]. It has the following interface.

```
INTERFACE datum-container {
   write(d)        //Store datum d in the datum-container.//
   read() returns d  //Return the datum d that is stored in the datum-container.//
}
```

2.2.3 The Datum-Container Passing Modes

A *datum-container-set* is a container for datum-containers. It has the following interface.

```
INTERFACE datum-container-set {
   insert(D)  //Insert datum-container D in the datum-container-set.//
   remove(D)  //Remove datum-container D from the datum-container-set.//
}
```

[1] *Data* is the plural of datum — Webster's New Collegiate Dictionary.

The following notation is used to define datum-container passing modes. DS_s and DS_d are datum-container-sets. D is a datum-container that is in DS_s.

Dynamic-share causes D to be in DS_d ($DS_d \rightarrow$**insert**(D)). D is in DS_s and DS_d.

Move causes D to be in DS_d and not be in DS_s ($DS_s \rightarrow$**remove**(D) and $DS_d \rightarrow$**insert**(D)). D is moved from DS_s to DS_d.

Duplicate causes a newly created datum-container D_0 to be in $DS_d$and the datum in D to be copied to D_0 (datum-container::**create**() *returns* D_0, $DS_d \rightarrow$**insert**(D_0), $D_0 \rightarrow$**write**(D$\rightarrow$**read**())). D is duplicated in DS_d.

The datum in a datum-container is passed along with the datum-container; therefore, a datum-container passing mode is also a datum-passing mode.

2.2.4 The Copy Datum-Passing Mode

The *Copy* datum-passing mode is a way of passing a datum from a source datum-container D_s to a destination datum-container D_d. The contents of D_s are written to D_d ($D_d \rightarrow$**write**($D_s \rightarrow$**read**())). This mode causes the same datum to be in two separate datum-containers.

The copy datum passing mode and duplicate datum-container passing modes are similar. They are distinguished because of the way they can be implemented rather than the differences in their semantics.

2.3 Specification of Datum-Containers With Addresses

A datum-container must be specified when an operation is invoked on it. The technique of specifying a datum-container with an address via a context can be implemented efficiently on conventional hardware.

An *address* is a positive integer that is used to specify the location of a datum-container.

A *binding* is an association between an address and a datum-container.

The notation (A,D) is used to represent a binding that associates address A with datum-container D.

2.3.1 Context Definition

A *context* is a container for bindings. It has the interface:

```
INTERFACE context {
   bind(A,D)  //Insert binding (A,D) in the context where (A,D) is the only//
              //binding in the context that associates A with a datum-container.//
   unbind(A)  //Remove every binding that associates address A with a//
              //datum-container from the context.//
}
```

Bindings can be added or removed from a context but they cannot be changed.

There cannot be two bindings in a context where the same address is in both bindings. This property eliminates ambiguity about which datum-container is being specified.

If the binding (A,D) is in context C then address A is *bound* in C and datum-container D is *bound* in C. If there is no binding in context C that associates A with a datum-container (or D with an address) then A is *unbound* in C (and D is *unbound* in C).

2.4 Passing Modes Involving Separate Contexts

A datum-container can be passed from a source context to a destination context because a context implicitly defines a datum-container-set. A context explicitly contains bindings; however, it implicitly defines a datum-container-set which contains the datum-containers that are bound in the context.

Contexts add a level of indirection to the data-container passing modes.

Datum-container passing modes between separate contexts are defined using the following notation. C_s and C_d are contexts. (A_s,D) is a binding in C_s. A_d is unbound in C_d.

Dynamic-share causes A_d to be bound to D in C_d ($C_d \rightarrow$ **bind**(A_d,D)). D is bound in C_s and C_d.

Move causes A_s to be unbound in C_s and A_d to be bound to D in C_d ($C_s \rightarrow$ **unbind**(A_s) and $C_s \rightarrow$ **bind**(A_d,D)). D is moved from C_s to C_d by removing a binding from C_s and adding a binding to C_d.

Duplicate causes A_d to be bound to a newly created datum-container D_0 in C_d and the datum in D to be copied to D_0 (datum-container::**create**() *returns* D_0, $C_d \rightarrow$ **bind**(A_d,D_0), $D_0 \rightarrow$ **write**(D$\rightarrow$**read**())). D is duplicated in C_d.

2.5 Passing Groups of Datum-Containers

The datum-container passing concept is expanded to groups of datum-containers. One motivation for grouping datum-containers is that groups of datum-containers can be passed efficiently by modifying the MMU translation tables that implement contexts.

An *address-region* is a continuous range of addresses.

A *data-region* is an ordered set of datum-containers.

A *region-binding* is a set of bindings from the addresses of an address-region to the datum-containers of a data-region.

Passing a data-region DR is similar to passing a single datum-container. Before passing DR there is a region-binding between an address-region AR_s and DR in a source context C_s and there is an unbound address-region AR_d in a destination

context. After passing DR, AR_d is bound to DR (or a newly created data-region DR_0 if the mode is duplicate) in C_d. If the mode is move then AR_s is unbound in C_s.

The copy datum-passing mode can also be expanded to data-regions. The contents of a data-region DR_s are copied to another data-region DR_d. DR_s is specified by the starting address of an address-region AR_s that is bound to DR_s in a context C_s. DR_d is specified by the starting address of an address-region AR_d that is bound to DR_d in a context C_d.

2.6 Interprocess Data-Passing

The motivation for the data-passing modes is to help support simple efficient data-passing between processes executing in separate contexts.

2.6.1 Process Definition

A *process* is a logical sequence of actions that can execute concurrently with other processes.

Invocation of an operation on an object is an action.

The *lifetime* of a process is the duration from the time the first action starts until the time the last action completes.

2.6.2 Relationship Between Processes and Contexts

A process[2] uses only one context during its lifetime. It uses the context to specify datum-containers with addresses.

[2] *Thread* or *thread of execution* can be defined as a sequence of actions that is not restricted to a single context.

The interface of a context is extended with **write** and **read** so that a process can access a datum-container via a context.

```
INTERFACE context {
   bind(A,D)
   unbind(A)
   write(A,d)  //Store datum d in the datum-container bound to A in the context.//
   read(A) returns d  //Return the datum d that is stored in the//
                      //datum-container bound to A in the context.//
}
```

A process can access (**write** and **read**) only the datum-containers that are bound in its context. The datum-container passing modes change the datum-containers a process can access.

2.6.3 Datum-Containers Shared by Processes

Passing a datum-container D from context C_s to context C_d using the dynamic-share mode causes the processes that use C_s and C_d to share access to D. A datum written to D by a sharing process will be returned to any sharing process that invokes C→**read**(A) (where address A is bound to D in context C). The sharing processes can concurrently invoke operations on the shared datum-container.

If data in shared datum-containers have properties that must be violated while updating the data then a process can read data that are not consistent with the intended properties. Therefore, a method of ordering **write** and **read** operations on shared datum-containers containing such data is required.

Ordering of operations is required when shared datum-containers contain data which can be in a state that is not consistent with the intended properties of the data. Synchronisation operations can be used to order accesses to shared datum-containers.

More than one process can use a context. A context is shared while it is used concurrently by more than one process. The processes that share a context share access to the datum-containers bound in the shared context.

2.7 Summary

Concepts related to data passing are defined.

The dynamic-share, move, duplicate and copy datum-passing modes are discussed. The first three modes pass a datum by passing the datum-container that contains the datum.

The dynamic-share mode causes datum-containers to be shared between separate contexts.

There are IPC models that provide data-passing operations which support the duplicate [Accetta 86] or move [Seitz 88] modes and non data-passing operations which establish dynamic memory sharing [Accetta 86, Bach 86]. In this chapter the duplicate, move and dynamic-share modes are presented as variations of datum-container passing modes.

Chapter 3

Regions IPC Model

The Regions IPC model is motivated by the need for simple efficient data-passing. The purpose of the Regions IPC model is to demonstrate how IPC operations can provide the dynamic-share, move and duplicate data-passing modes. The data-passing operations can be implemented efficiently by taking advantage of shared physically memory. The operations can also be implemented efficiently across communications channels in a distributed system.

Regions is based on the synchronous message-passing IPC model. Regions is a descendant of Thoth [Cheriton 79] via Port [Vasudevan 87] and the W System [Vasudevan 88]. Other data-passing models used for IPC such as asynchronous message-passing or remote procedure call could have been used to demonstrate operations that support the dynamic-share, move and duplicate data-passing modes.

The syntax and semantics of the Regions IPC primitive operations are explained following a description of some basic notions.

3.1 Basic Notions

3.1.1 Process Identification

A process is identified by a unique *process identifier (PID)* that distinguishes the process from all other processes in the system.

3.1.2 Private Context

A context is private if the context is used by only one process. In Regions every process has a private context.

To recapitulate, a context is a set of bindings. A binding is an association between an address and a datum-container.

Two types of bindings are now defined.

3.1.2.1 Permanent Bindings to Private Datum-Containers

A *permanent binding* has the following properties.

1. It is inserted into a context when the context is created and it is never removed.
2. It binds a datum-container D to an address where D is not bound to any other address in any other context. D is a private datum-container.

3.1.2.2 Dynamic Bindings to Sharable DRegions

In contrast a dynamic binding has the following properties.

1. It can be inserted into and removed from a context.
2. It binds a group of datum-containers DR to a range of addresses where DR can also be bound to other ranges of addresses. DR is a sharable group of datum-containers.

The dynamic binding definition is based on DRegions and ARegions.

A *DRegion* is a fixed sized ordered set of datum-containers where a datum-container cannot be part of more than one DRegion. The size of a DRegion is fixed when it is created and the maximum size is set by the system.

An *ARegion* is a continuous range of addresses where an address cannot be part of more than one ARegion within a context.

A *dynamic binding*

- is a set of bindings in a context that associate the addresses of an ARegion with the datum-containers of a DRegion, and
- is inserted or removed from a context by primitive operations.

A DRegion exists while there is a dynamic binding to it.

A process can invoke **write** and **read** operations on datum-containers bound in its context. This includes the datum-containers in shared DRegions. Datum-containers have *strictly consistent* semantics [Nitzberg 91]; **read** returns the most recently written datum. The moment in time that a datum is written to a datum-container occurs during the interval between when **write** is invoked and when it completes. If an operation is invoked on a datum-container D before another operation invoked on D completes, the results are the same as if one of the operations completed before the other was invoked. However, the order of the operations is undefined[1].

A process's context is created with all of its permanent bindings and without any dynamic bindings (without any shared datum-containers). The operation that creates a DRegion and the operations that pass a DRegion cause dynamic bindings to be in a context.

3.2 IPC-Related Primitive Operations

The synchronous message passing primitives are:
send(receiver, request-message, response-message) *returns* **success,**
receive(receive-message) *returns* **sender,** and
reply(sender, reply-message).

The DRegion passing primitives are:

[1]The duration between invoking and completing an operation on a datum-container that is shared across a communication channel can vary by several orders of magnitude.

pass-region-to(sender, src-ARegion, mode) *returns* **dst-ARegion** and
pass-region-from(sender, src-ARegion, mode) *returns* **dst-ARegion.**

Other primitives that involve dynamic bindings to DRegions are:
create-region(size) *returns* **ARegion,**
delete-region(ARegion),
rebind-region(src-ARegion, dst-ARegion) *returns* **success,**
size-of-region(ARegion) *returns* **size,** and
state-of-region(ARegion) *returns* **state.**

3.2.1 Creating and Deleting Dynamic Bindings

Because a process's context is created without any dynamic bindings, an operation is required to create a binding to a DRegion. A process can also delete a binding to a DRegion.

3.2.1.1 The create-region Primitive

create-region binds an ARegion to a newly created DRegion of a specified size in the caller's context.

> **create-region(size)** *returns* **ARegion**
> where
> **size** is the number of datum-containers(bytes) in the created DRegion, and
> **ARegion** is the ARegion bound to the created DRegion or an error value.

create-region will fail if **size** is invalid (**size**<1 or MAXIMUM<**size**), or there is insufficient memory available to create a DRegion, or there are no unbound ARegions available in the caller's context to bind to the DRegion. If **create-region** fails an error value is returned.

3.2.1.2 The delete-region Primitive

delete-region causes a specified ARegion to be unbound in the caller's context.

delete-region(ARegion)
where
ARegion becomes unbound in the caller's context.

delete-region cannot fail; if a bound ARegion is specified then it is unbound.

3.2.2 Passing Messages and DRegions

Processes use IPC primitives to synchronise their execution relative to each other and to pass messages and DRegions.

A *message* is a short fixed sized data-region (64 bytes). Messages are intended for passing control information.

Messages are passed using the COPY data passing mode. The data is written to datum-containers that the destination process already has access to.

3.2.2.1 The send, receive and reply Primitives

Processes communicate using **send-receive-reply** (SRR) transactions. Two examples of SRR transactions are shown in Figure 3.1. The process that invokes **send** is called the sender. The process that invokes **receive** is called the receiver. The receiver can invoke **receive** before a request message arrives from **send** (example (a)) or after request messages arrives (example (b)).

send initiates delivery of a request message and causes the sender to wait until the response message arrives or a failure occurs. **send** returns the response message or an error value.

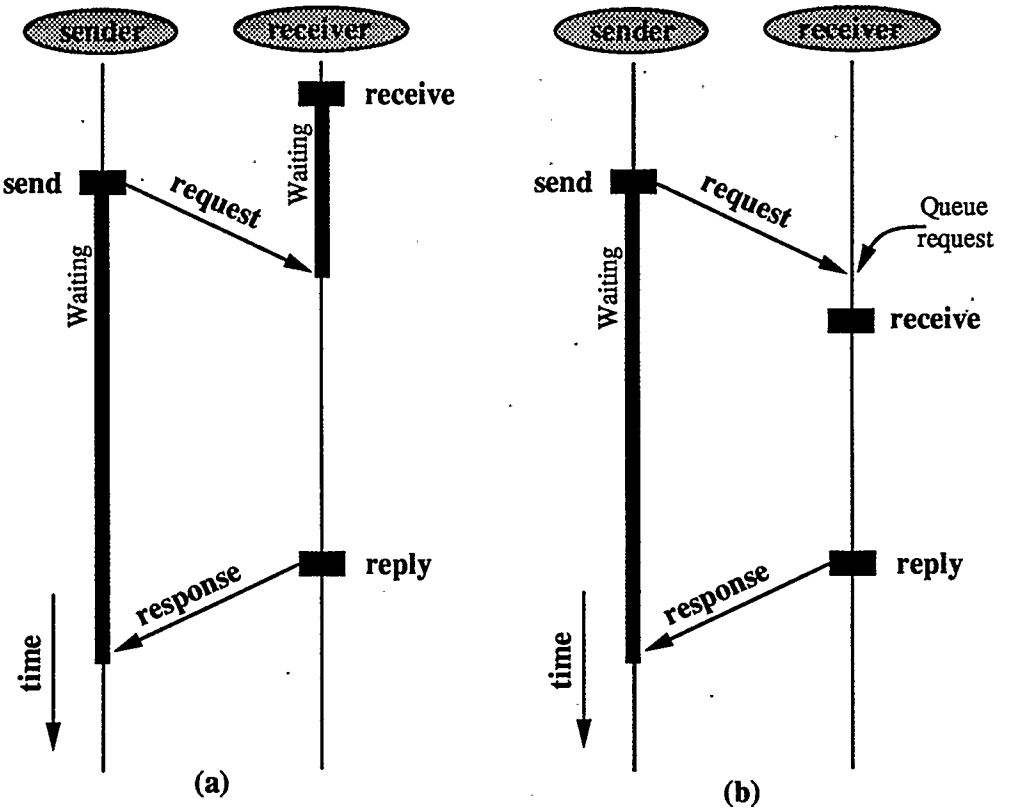

Figure 3.1: Two examples of SRR transactions.

receive returns the request message and the PID of the process that sent the message. If request messages arrive before **receive** is invoked then they are queued in the order they arrive. If no request messages have arrived when **receive** is invoked the receiver waits until a request message arrives.

reply initiates delivery of the response message.

The message passing primitives are:

send(receiver, request-message, response-message) *returns* **success**

receive(receive-message) *returns* **sender**

reply(sender, reply-message)

where

receiver is the PID of the process being sent to,

sender is the PID of the process that sent the received message,

request-message is the message that **send** passes,

receive-message is the message that **receive** returns,

reply-message is the message that **reply** passes,

response-message is the message that **send** returns, and

success is the constant SUCCESSFUL or an error value.

The contents of the sender's **request-message** are copied into the receiver's **receive-message**. The contents of the receiver's **reply-message** are copied into the sender's **response-message**.

Multiple **sends** to the same receiver are queued and returned in first-come-first-served (FIFO) order. A process can invoke multiple **receives** before invoking **replys** in response to those **receives**. There is no restriction on the order that a process invokes **replys** in response to **receives**.

send will fail if one of the messages is not bound to datum-containers or communication with **receiver** is not possible[2]. If **send** fails because **request-message** or **response-message** is not bound to datum-containers then the caller terminates. If **send** fails because communication is not possible then an error value is returned. The failure semantics of **send** are discussed further in Section 3.2.4.

receive will fail if **receive-message** is not bound to datum-containers. If **receive** fails then the caller terminates.

reply will fail if **reply-message** is not bound to datum-containers. If **reply** fails then the caller terminates.

3.2.2.2 The pass-region-to and pass-region-from Primitives

A process can pass DRegions to and from another process that has invoked **send** and is awaiting a response.

pass-region-to passes a DRegion to a sender using a specified data passing mode. The DRegion is specified by an ARegion that is bound to it in the caller's context. The ARegion that is bound to the passed DRegion in the sender's context is returned.

pass-region-from is similar to **pass-region-to** except that it passes a DRegion from a sender. The DRegion is specified by an ARegion that is bound to it in the sender's context. The ARegion that is bound to the passed DRegion in the caller's context is returned.

[2]Inability to communicate can be due to the specified process not existing, or a machine failure, or a communication channel failure.

The primitives for passing DRegions are:

pass-region-to(sender, src-ARegion, mode) *returns* **dst-ARegion**

pass-region-from(sender, src-ARegion, mode) *returns* **dst-ARegion**

where

sender is the PID of a process awaiting a response to **send**,

mode is the data passing mode used to pass the DRegion,

src-ARegion is the source ARegion bound to a DRegion, and

dst-ARegion is the destination ARegion that is bound to the passed DRegion.

pass-region-to will fail if (1) an invalid argument is specified, (2) there are insufficient resources to pass the DRegion or (3) communication with **sender** is not possible. An invalid argument is a **sender** that is not awaiting a response, a **src-ARegion** that is not bound to a DRegion in the caller's context, or an invalid **mode**. There are insufficient resources when an unbound ARegion is not available in **sender**'s context to be bound to the passed DRegion or when memory is not available to create a DRegion (if one must be created). If **pass-region-to** fails then an error value is returned.

pass-region-from will fail for similar reasons.

The failure semantics of **pass-region-to** and **pass-region-from** are discussed further in Section 3.2.4.

If **sender** is specified as ME then the caller's context is the source and destination context. This is useful for creating a duplicate DRegion within a context.

3.2.2.3 Data-Passing Modes for DRegions

A process P_s passes a DRegion DR to a process P_d using one of the data passing modes: DYNAMIC-SHARE, MOVE, or DUPLICATE. DR is bound to AR_s in P_s's context

C_s. AR_d is an unbound ARegion in P_d's context C_d that can be bound to DR.

If DR is passed using the DYNAMIC-SHARE mode then AR_d becomes bound to DR in C_d. P_s and P_d share access to DR.

If DR is passed using the MOVE mode then AR_s becomes unbound in C_s and AR_d becomes bound to DR in C_d. P_d has access to DR via AR_d and P_s does not have access to DR via AR_s.

If DR is passed using the DUPLICATE mode then AR_d is bound to a newly created DRegion DR_0 in C_d where DR and DR_0 are separate DRegions. The contents of DR are copied to DR_0. P_d has access to DR_0 via AR_d and no other process has access to DR_0. P_s retains access to DR via AR_s.

3.2.3 Other Useful DRegion Related Primitives

3.2.3.1 The rebind-region Primitive

If a DRegion contains complex data types where pointers are implemented as absolute addresses then that DRegion must be bound to a specific ARegion in every context. **create-region**, **pass-region-to** and **pass-region-from** select the ARegion to bind to a DRegion. Therefore, an operation for changing the ARegion that is bound to a DRegion is provided.

rebind-region binds a specified destination ARegion[3] to the DRegion bound to a specified source ARegion and unbinds the source ARegion.

[3]The ARegion that a DRegion must be bound to can be stored within the DRegion. When a DRegion is created with **create-region** the returned ARegion AR can be written into the DRegion. Then AR can be read from the DRegion and used as the destination argument to **rebind**.

rebind-region(src-ARegion, dst-ARegion) *returns* **success**

where

src-ARegion is the source ARegion bound to a DRegion in the caller's context,

dst-ARegion is an unbound destination ARegion in the caller's context, and

success is the constant SUCCESSFUL or an error value.

rebind-region will fail and return an error value if **src-ARegion** is not bound or **dst-ARegion** is bound in the caller's context.

3.2.3.2 The size-of-region Primitive

A process can determine the size of a DRegion that is bound in its context.

size-of-region returns the size of a DRegion bound to a specified ARegion in the caller's context.

size-of-region(ARegion) *returns* **size**

where

ARegion is bound to a DRegion in the caller's context, and

size is the number of datum-containers (bytes) in the DRegion.

size-of-region will fail and return an error value if **ARegion** is not bound to a DRegion in the caller's context.

3.2.3.3 The state-of-region Primitive

state-of-region returns one of the four constants INVALID, UNBOUND, PRIVATE, or SHARED depending on the specified address. If the specified address is not the starting address of a valid ARegion then INVALID is returned. If the address does specify an ARegion but it is unbound in the caller's context then UNBOUND is returned. If

the ARegion is bound to a DRegion in the caller's context and it is the only ARegion bound to that DRegion then PRIVATE is returned. If there are other ARegions bound to that DRegion then SHARED is returned.

state-of-region(ARegion) *returns* **state**

where

ARegion is any address (but is usually an ARegion), and

state is the state of the ARegion: INVALID, UNBOUND, PRIVATE, or SHARED.

state-of-region cannot fail.

If **state-of-region** returns SHARED then there was (and still might be) more than one binding to the DRegion. A communication failure can prevent the system from detecting that the number of bindings to a DRegion has been increased from one or reduced to one.

3.2.3.4 ARegion State Changes

The ARegion state changes caused by **create-region** and **delete-region** are presented in Table 3.1.

Primitive	State Change
create-region	UNBOUND → PRIVATE
delete-region	PRIVATEorSHARED → UNBOUND

Table 3.1: ARegion state changes caused by **create-region** and **delete-region.**

The ARegion state changes caused by **pass-region-to** or **pass-region-from** are presented in Table 3.2.

Invoking **delete-region** on a ARegion that is bound to a shared DRegion can cause the state of another ARegion bound to that DRegion to change. These asynchronous state changes is presented are Table 3.3.

Data Passing Mode	Source State Change	Destination State Change
DYNAMIC-SHARE	PRIVATE → SHARED	UNBOUND → SHARED
	SHARED → SHARED	UNBOUND → SHARED
MOVE	PRIVATE → UNBOUND	UNBOUND → PRIVATE
	SHARED → UNBOUND	UNBOUND → SHARED
DUPLICATE	PRIVATE → PRIVATE	UNBOUND → PRIVATE
	SHARED → SHARED	UNBOUND → PRIVATE

Table 3.2: ARegion state changes caused by passing DRegions.

Primitive	State of Other ARegions Bound to the DRegion	Note
delete-region	SHARED → SHARED	(1)
	SHARED → PRIVATE	(2)

Table 3.3: ARegion state changes caused asynchronously by **delete-region.**

Note (1): If **delete-region** is invoked on an ARegion that is bound to a DRegion DR where there are two or more other bindings to DR then that ARegion's state changes from SHARED to UNBOUND and the state of the other ARegions bound to DR do not change.

Note (2): If **delete-region** is invoked on an ARegion bound to a DRegion that has only one other ARegion bound to it then the state of the other ARegion asynchronously changes from SHARED to PRIVATE.

When a process terminates, all of the bindings are deleted. Therefore, process termination can cause ARegion state changes asynchronously.

3.2.4 Failure Semantics of the Regions Primitives

The failure semantics of the primitive operations defined in the previous section are now discussed in greater detail.

The operations are intended to be implemented on distributed systems where communication failures can prevent an operation from determining whether or not a

result occurred. The ability to recover from communication failures is an important characteristic of an IPC model for distributed systems[4]. Justification is presented for operations that initiate a request without waiting for confirmation and operations that cause a process to terminate rather than returning an error value.

3.2.4.1 Exactly-Once and At-Most-Once Semantics

The Regions primitives have one of the following two types of semantics.

An operation with *exactly-once* semantics has the following properties. If the operation completes successfully then all the results were produced exactly once. If the operation does not complete successfully then an error value can be returned indicating which results were produced exactly once and which results were not produced.

An operation with *at-most-once* semantics has the following properties. If the operation completes successfully then all the results were produced exactly once. If the operation does not complete successfully then an error value can be returned indicating which results were produced exactly once, which results were not produced and which results were either produced once or not produced.

Operations that use a communication channel have at-most-once semantics if a communication failure[5] can prevent delivery of a confirmation indicating whether a requested result was produced or not.

The primitives **send**, **pass-region-to** and **pass-region-from** have at-most-once semantics. All other primitives discussed in this chapter have exactly-once semantics.

It is not possible to determine whether or not a **send**'s request message was delivered unless a response arrives. The possibilities are that the request was not

[4]The importance of fault recovery depends in part on the probability of failure and in part on the consequences of failure.

[5]Control packets can be passed periodically by the implementation to distinguish an inability to communicate from a delayed response.

received, the request was received but the receiver did not reply, or the receiver replied but the response did not arrive.

It is not possible to determine whether or not a **pass-region-to** request caused an ARegion to be bound to the passed DRegion in the sender's context unless a response arrives. The passed DRegion is either bound **or not** bound in the sender's context.

It is not possible to determine whether of not a **pass-region-from** request, where **mode** is MOVE, caused or did not cause the source ARegion to be unbound from the passed DRegion unless a response arrives. If **mode** is DUPLICATE or DYNAMIC-SHARE then **pass-region-from** has exactly-once semantics because the caller can determine whether or not the passed DRegion is bound in the caller's context.

An advantage of primitives that use the DYNAMIC-SHARE, MOVE and DUPLICATE passing modes over primitives that use COPY is that the destination process receives all of the data or none of the data. If data is copied it must be buffered until all the data has arrived and then copied to the final destination to avoid the possibility of partially overwriting the destination datum-containers.

3.2.4.2 Unreliable Delivery of Requests

There are several primitives that initiate delivery of a request and do not wait for confirmation that the request was completed successfully. The end-to-end argument [Saltzer 84] is used to justify these semantics.

reply initiates delivery of a response message but does not wait for a confirmation. A confirmation indicates that the response was delivered but does not indicate whether or not the response was acted on. Providing reliable delivery can only improve efficiency. End-to-end confirmation is required to ensure that the message was acted on. Waiting for a confirmation would increase the elapsed time of **reply**.

If MOVE is specified as the data passing mode to a **pass-region** primitive then the

source ARegion is unbound in the source context without waiting for confirmation that the DRegion was successfully moved. Retaining a binding to the DRegion is only important for fault recovery. A binding to a DRegion with the original data must be retained until confirmation is received that the data is no longer needed. This is true regardless of the data passing mode[6].

If a DRegion is shared across disjoint memories on separate machines then the number and source of bindings to the DRegion must be maintained. **delete-region** reduces the number of bindings to a DRegion. The **pass-region** primitives increase the number of bindings if **mode** is DYNAMIC-SHARE and they change the source of bindings if **mode** is MOVE. DRegion state (number and source of bindings) is updated by initiating delivery of requests. The primitives do not wait for a confirmation; therefore, if communication fails then bindings to a DRegion can be removed, added or moved without updating the DRegion state.

3.2.4.3 Failures Causing Process Termination

If a **write** or **read** operation is invoked on an address that is not bound to a datum-container then the process that invoked the operation is terminated. Returning an error value is not useful because there is no reason for a process to attempt to specify an address that might not be bound to a datum-container. In addition, it is inefficient and complex for a process to check for an error after each datum-container access. This is also the justification for terminating a process that invokes a **send**, **receive** or **reply** with an specified message that is not bound to datum-containers.

If a **write** or **read** operation is invoked on an address that is bound to a DRegion which is shared across disjoint memories then a communication failure can cause the operation to fail. The process is terminated if **write** or **read** fails because of a

[6]If a DRegion is not successfully passed then MOVE unbinding the DRegion is analogous to a process modifying data that was passed with the DUPLICATE or COPY data passing mode.

communication failure.

The model would be improved if exception handling allowed a process to recover from a **write** or **read** failure caused by a communication failure.

read has exactly-once semantics. Implementations that provide either exactly-once or at-most-once semantics for **write** are discussed in Section 4.8 .

3.3 Discussion

3.3.1 Efficiency and Equivalence of the Data Passing Modes

Under certain circumstances the results produced by one data passing mode can be produced by another data passing mode in combination with other operations. The equivalences between data passing modes are explained. The three modes are supported because unnecessary data copying can be avoided when the mode appropriate to the intended use of the passed data is used.

The DUPLICATE and MOVE modes can be used with DRegions that are not shared without introducing sharing. A server process that efficiently provides client processes with synchronised access to a DRegion with DUPLICATE and MOVE without introducing sharing is explained in Section 3.3.3.2.

If the state of **src-ARegion** is PRIVATE then

pass-region-to(sender, src-ARegion, DUPLICATE**)** *returns* **dst-ARegion**

is equivalent to

size-of-region(src-ARegion) *returns* **size**

create-region(size) *returns* **AR**

(*Copy contents of* **src-ARegion** *to AR.*)

pass-region-to(sender, AR, MOVE**)** *returns* **dst-ARegion.**

However, using DUPLICATE can allow the implementation to avoid unnecessary

data copying.

If the caller and sender are executing on machines that share physical memory then DUPLICATE can be deferred until a **write** operation is invoked on the original or deferred duplicate (see Chapter 4). Therefore, if a **write** is not invoked before the binding from the original or deferred duplicate is removed then unnecessary copying is avoided.

If the caller and sender are executing on machines that do not share physical memory then the implementation of DUPLICATE or MOVE will copy the data to the destination machine. Therefore, DUPLICATE avoids the extra copy before the move.

Inversely, if the state of **src-ARegion** is PRIVATE then

pass-region-to(sender, src-ARegion, MOVE**)** *returns* **dst-ARegion**

is equivalent to

pass-region-to(sender, src-ARegion, DUPLICATE**)** *returns* **dst-ARegion**

delete-region(src-ARegion).

However, MOVE is preferable to DUPLICATE even when duplicating is deferred because of the overhead of the delete and the deferring[7].

The DYNAMIC-SHARE mode introduces sharing.

pass-region-to(sender, src-ARegion, MOVE**)** *returns* **dst-ARegion**

is equivalent to

pass-region-to(sender, src-ARegion, DYNAMIC-SHARE**)** *returns* **dst-ARegion**

delete-region(src-ARegion).

However, MOVE is preferable to DYNAMIC-SHARE especially when the DRegion is passed across a communication channel. If the passed DRegion is not shared then DYNAMIC-SHARE requires that sharing be established and delete causes the sharing

[7]If the Regions model supported passing a DRegion with a message using **send** or **reply** then the destination process could **write** to the DRegion before the source process deleted its binding causing the data to be unnecessarily copied.

to be eliminated whereas MOVE only requires that the data be copied. If the passed DRegion is shared then DYNAMIC-SHARE increases the sharing and delete decreases the sharing whereas MOVE just changes the source of the binding.

3.3.2 Explicit Data-Passing

IPC models that do not allow data sharing force cooperating processes to explicitly pass data. This complicates the specification of parallel algorithms. In addition, the time required to pass the data can be a significant part of the time required to execute the parallel algorithm.

Initially, processes must explicitly pass data in the Regions model. However, once sharing of data is established with DYNAMIC-SHARE the processes only need to ensure that they do not cause incorrect results by interfering with each other. This can simplify the specification of parallel algorithms.

3.3.3 Synchronisation

A method of ordering operations on shared datum-containers is required because the DYNAMIC-SHARE mode establishes sharing between processes. Two methods of ordering are explained. Both methods provide synchronisation between processes accessing shared DRegions. The processes can be executing anywhere in a distributed system.

If data in a shared DRegion is intended to have properties that are violated when the data are updated then a process can read data that is not consistent with the intended properties. Therefore, an efficient method of ordering **write** and **read** operations on shared DRegions containing such data is required. Ordering is also required when that DRegion is duplicated or copied.

Sketches of algorithms for two server processes are presented that illustrate how

processes can order their operations to ensure that inconsistent data is not read.

3.3.3.1 Semaphore Server Example

An algorithm that provides a semaphore [Dijkstra 68] service is presented. The algorithm does not use operations involving DRegions; it assumes that client processes use the service to synchronise access to a DRegion that was passed with the DYNAMIC-SHARE mode.

Clients invoke the P and V operations on semaphores as follows.

P(SEMAPHORE-INDEX)
    Insert operation P and SEMAPHORE-INDEX into **message**
    send(semaphore-server, message, response-message)

V(SEMAPHORE-INDEX)
    Insert operation V and SEMAPHORE-INDEX into **message**
    send(semaphore-server, message, response-message)

Semaphore Server — Algorithm 3.1

```
Initialise array of semaphores S to AVAILABLE
Initialise array of queues for waiting processes W to EMPTY
while(TRUE)
    receive( message) returns client
    Extract the operation op and the index i from message
    if ( op is P) then
        if ( semaphore S[i] is BUSY ) then
            Put client in queue W[i]
        else
            reply(client, <message indicating P successful>)
            Set semaphore S[i] to BUSY
    else if ( op is V ) then
        reply(client, <message indicating V successful>)
        if ( semaphore S[i] is BUSY ) then
            if ( queue W[i] is EMPTY ) then
                Set semaphore S[i] to AVAILABLE
```

else

Remove next **waiting-client** from queue **W[i]**
reply(waiting-client, <message indicating P successful>)

Processes executing on separate machines can use this server to synchronise.

A server can provide multiple semaphores so there does not have to be one server per semaphore; however, the server might serialise P and V operations to independent semaphores.

P and V operations on semaphores can be implemented with hardware test-and-set instructions in memory-sharing IPC models. Therefore, this operation would be more efficient in a Shared Variable model especially when processes do not have to wait for a busy semaphore. However, this difference in performance is only significant when frequent synchronisation is required.

3.3.3.2 Synchronous Shared Memory Server Example

The next algorithm enforces synchronised access to the contents of a private DRegion with the MOVE and DUPLICATE data passing modes (DYNAMIC-SHARE is not used). The algorithm solves the readers and writers problem and it has some interesting features.

The reader and writer client processes can be executing on the same machine or different machines in a distributed system. If the client is executing on another machine then the contents of the DRegion must be copied. If the client is executing on the same machine then the implementation can pass the DRegion by modifying the MMU translation tables of the relevant contexts.

A reader can modify its copy of the DRegion with no adverse consequences because the reader's copy is passed with the DUPLICATE data passing mode. If the reader is executing on another machine then it already has a separate copy of the DRegion. If the reader is executing on the same machine then DUPLICATE is deferred

and a **write** operation causes the DRegion to be duplicated. This is an advantage over systems that provide readers with read-only access to the data for two reasons. First, a reader does not have to explicitly copy the data if it needs to modify a private copy of the data. Second, writers do not have to wait for readers.

A writer does not have to wait if readers have acquired access to the DRegion. The original copy of the DRegion is immediately passed to the writer that requests it with the MOVE data passing mode. If the writer is executing on the same machine as the server and a reader then the first **write** operation by the writer causes the data to be copied. Otherwise the data does not have to be copied. The algorithm can be modified to cause a writer to wait for readers if waiting is preferable to copying.

Requests by readers and writers that occur while a writer has access to the data are queued in the order they arrive. When the writer releases the shared data then the server grants reader requests until the queue is empty or a writer request is encountered. It grants one writer request (if there was one) and waits for the writer to release its access.

When a writer releases its copy of the data the server uses MOVE to obtain the DRegion. Therefore, a writer can release a different DRegion then it acquired.

Readers do not have to release their access but if they do (by deleting their binding to the DRegion) then unnecessary copying will be avoided.

Clients can invoke the following six operations on the server.

create-shared-DRegion(n)
Insert operation CREATE and size **n** in **message.**
send(sync-memory-server, message, response)
Extract index **i** of created DRegion from **response.**

destroy-shared-DRegion(i)
Insert operation DESTROY and index **i** in **message.**
send(sync-memory-server, message, response)

acquire-write-DRegion(i)

Insert operation ACQUIRE-WRITE and index **i** in **message.**
send(sync-memory-server, message, response)
Extract ARegion **destination** from **response.**

release-write-DRegion(i, source)

Insert operation RELEASE-WRITE, index **i** and ARegion **source** in **message.**
send(sync-memory-server, message, response)

acquire-read-DRegion(i)

Insert operation ACQUIRE-READ and index **i** in **message.**
send(sync-memory-server, message, response)
Extract ARegion **destination** from **response.**

release-read-DRegion(i, source)

delete-region(source)

Synchronous Memory Server — Algorithm 3.2

There is an array **S** for shared DRegions where an entry **i** of **S** consists of
S[i]:state - Shared state: UNDEFINED, NOT-READ-WRITE, or READ-WRITE.
S[i]:queue - Processes waiting for access to the shared DRegion.
S[i]:ARegion - ARegion bound to shared DRegion in the server's context.
S[i]:writer - Process with read-write access to the shared DRegion.
Initialise **S[i]:state** to UNDEFINED and **S[i]:queue** to EMPTY for all **i**
while(TRUE)
receive(message) *returns* **client**
Extract the operation **op** from **message**
if (**op** is CREATE) **then create(client, message)**
else if (**op** is DESTROY) **then destroy(client, message)**
else if (**op** is ACQUIRE-WRITE) **then acquire-write(client, message)**
else if (**op** is RELEASE-WRITE) **then release-write(client, message)**
else if (**op** is ACQUIRE-READ) **then acquire-read(client, message)**

The following routine creates a sharable DRegion.

create(client, message)
Extract the size **n** from **message**
Select index **i** where **S[i]:state** is UNDEFINED and **S[i]:queue** is EMPTY
create-region(n) *returns* **S[i]:ARegion**
Set **S[i]:state** to NOT-READ-WRITE
Insert **i** in **message**
reply(client, message)

The following routine destroys a sharable DRegion.

destroy(client, message)
Extract index **i** from **message**
if (**S[i]:state** is UNDEFINED) **then**
reply(client, <message indicating error>)
else
if (**S[i]:state** is NOT-READ-WRITE) **then**
Set **S[i]:queue** to EMPTY
delete-region(S[i]:ARegion)
else
Insert (**client**,DESTROY) in **S[i]:queue**
Set **S[i]:state** to UNDEFINED
reply(client,<message indicating destroy successful>)

The following routine handles write requests.

acquire-write(client, message)
Extract index **i** from **message**
if (**S[i]:state** is UNDEFINED) **then**
reply(client,<message indicating error>)
else if (**S[i]:state** is READ-WRITE) **then**
Insert (**client**,READ-WRITE) in **S[i]:queue**
else
pass-region-to-writer(client, S[i])

The following routine handles a release by a writer.

```
release-write(client, message)
    Extract index i and source address source from message
    if ( S[i]:state is not READ-WRITE ) or (S[i]:writer is not client ) then
        reply(client,<message indicating error>)
    else
        pass-region-from(client, source, MOVE) returns S[i]:ARegion
        state-of-region(S[i]:ARegion) returns state
        if ( state is SHARED ) then eliminate-sharing(S[i])
        reply(client,<message indicating released>)
        Set readers? to TRUE
        while ( readers? and S[i]:queue is not EMPTY )
            Remove next (waiting-client, request) from S[i]:queue
            if ( request is READ-ONLY ) then
                pass-region-to-reader(waiting-client, S[i])
            else
                Set readers? to FALSE
                if ( request is READ-WRITE ) then
                    pass-region-to-writer(waiting-client, S[i])
                else <request is DESTROY>
                    Set S[i]:queue to EMPTY
                    delete-region(S[i]:ARegion)
        if ( S[i]:queue is EMPTY ) then
            Set S[i]:state to NOT-READ-WRITE
```

The following routine handles read requests.

```
acquire-read(client, message)
    Extract index i from message
    if ( S[i]:state is UNDEFINED ) then
        reply(client,<message indicating error>)
    else if ( S[i]:state is READ-WRITE ) then
        Insert (client,READ-ONLY) in S[i]:queue
    else
        pass-region-to-reader(client, S[i])
```

The following three routines are used by the preceding routines.

The first routine passes a shared DRegion to a writer.

pass-region-to-writer(client, S)
    pass-region-to(client, S:ARegion, MOVE**)** *returns* **destination**
    Insert **destination** in **message**
    reply(client, message)
    Set **S:state** to READ-WRITE
    Set **S:writer** to **client**

The second routine passes a shared DRegion to a reader.

pass-region-to-reader(client, S)
    pass-region-to(client, S:ARegion, DUPLICATE**)** *returns* **destination**
    Insert **destination** in **message**
    reply(client, message)

If a writer releases a DRegion that is shared then the third routine eliminates the sharing by duplicating the DRegion.

eliminate-sharing(S)
    pass-region-to(ME**, S:ARegion,** DUPLICATE**)** *returns* **destination**
    delete-region(S:ARegion)
    Set **S:ARegion** to **destination**

3.3.3.3 Region's Servers and Monitors

Server processes in the Regions model are more similar to monitors [Hoare 74] than servers in other models like the V System [Cheriton 88]. A client can use the MOVE or DYNAMIC-SHARE modes to efficiently share state information with a server.

3.3.4 Asynchronous Binding to DRegion Example

Other IPC models [Accetta 86, Bach 86] provide dynamic sharing with asynchronous operations. A process must obtain an identifier for a shared memory region. Then it can create a binding to that shared memory without interacting with another process. The shared memory continues to exist even if there are no bindings to it.

An algorithm is sketched for a server that provides similar semantics to the UNIX System V shared memory feature. However, this server also provides the service to processes executing on any machine in a distributed system.

A client can invoke the following operations on the server.

register(key, size)
Insert operation REGISTER, **key** and **size** in **message**
send(DRegion-binding-server, message, response)
Extract index **i** from **response**

destroy(i)
Insert operation DESTROY and index **i** in **message**
send(DRegion-binding-server, message, response)

bind(i, ARegion)
Insert operation BIND, index **i** and address **ARegion** in **message**
send(DRegion-binding-server, message, response)

unbind(ARegion)
delete-region(ARegion)

The UNIX System V shared memory operations are similar to and correlate with the preceding four operations as follows:

register is similar **shmget(key, size, flags)**,

destroy is similar **shmctl(i, cmd, &buf)**,

bind is similar **shmat(i, address, flags)**, and

unbind is similar **shmdt(address)**.

DRegion Binding Server — Algorithm 3.3

There is an array **S** of shared DRegions where an entry **i** of **S** consists of
S[i]:state - Shared DRegion state: UNDEFINED or DEFINED.
S[i]:key - the key that processes use when interacting with the server.
S[i]:registered - List of processes registered to the shared DRegion.

S[i]:size - Size of the shared DRegion.
S[i]:ARegion - ARegion bound to the DRegion in the server's context.
S[i]:must-create - TRUE if the DRegion has not been created.

```
Initialise S[i]:state to UNDEFINED for all i
while(TRUE)
    receive(message) returns client
    Extract operation op from message
    if ( op is REGISTER ) then register(client, message)
    else if ( op is DESTROY ) then destroy(client, message)
    else if ( op is BIND ) then bind(client, message)
```

The current version of the Regions model cannot provide read-only access to a DRegion that can be modified by another process[8].

The following routine registers a process so it can bind to a shared DRegion.

```
register(client, message)
    Extract key from message
    Find index i where S[i]:key is key
    if ( key is not found in S ) then
        Select index i where S[i]:state is UNDEFINED
        Set S[i]:state to DEFINED, S[i]:key to key, S[i]:create to TRUE
        Extract size from message and set S[i]:size to size
    Register client by adding it to S[i]:registered
    reply(client, <message containing i>)
```

The following routine destroys a shared DRegion so that no other process can attach to it; however, processes with bindings to the DRegion can continue to access it.

```
destroy(client, message)
```

[8] A DYNAMIC-SHARE-NONWRITABLE data passing mode was included in the original design of the model; however, this feature complicated the model. An alternative approach based on capabilities could provide this feature and also solve other access permission problems with the current version of the model (see Chapter 6).

```
Extract i from message
if ( client not in list S[i]:registered ) then
    reply(client, <message indicating error>)
else
    delete-region(S[i]:ARegion)
    Set S[i]:state to UNDEFINED
    reply(client, <message indicating success>)
```

The following routine binds an ARegion in the client's context to a shared DRegion.

```
bind(client, message)
    Extract i and ARegion from message
    if ( client not in S[i]:registered ) then
        reply(client, <message indicating error>)
    else
        if ( S[i]:must-create ) then
            create-region(S[i]:size) returns S[i]:ARegion
        pass-region-to(client, ARegion, DYNAMIC-SHARE)
                                        returns S[i]:ARegion
        reply(client, <message indicating success>)
```

3.4 Summary

A simple IPC model is explained where data can be passed with the dynamic-share, move and duplicate data-passing modes. Processes do not share memory initially and sharing is not required for communication. However, the same operations that pass data can also dynamically establish memory sharing by changing an argument value. The model provides a simple abstraction that hides machine boundaries and it can be implemented efficiently on distributed systems.

The Regions IPC model is an improvement over existing data-passing IPC models[9]. Explicit data-passing can be avoided once sharing is established. Passing complex data types is simple and can be implemented efficiently. Unnecessary data copying can be avoided by the implementation.

The Regions IPC model is an improvement over memory-sharing IPC models. Processes executing in separate contexts can share memory. Processes can establish sharing of the memory that needs to be shared. Processes do not have to be relocated so that they can exist within the same context. The context implementation does not have to be distributed across machines in a distributed system.

The Regions IPC model is an improvement over IPC models that provide data-passing and dynamic memory sharing. Data passing and dynamic memory sharing are integrated in simple IPC operations. Dynamic memory sharing is provided across machines in a distributed system.

[9]Explicit data-passing, unnecessary data copying and difficulty passing complex data types have been cited as problems with data-passing IPC models [Li 86]

Chapter 4

Implementing Regions IPC

The motivation for an implementation is to demonstrate that the Regions IPC model can be implemented efficiently. A general object-based specification is presented. Implementation details for Sun 3 workstations [SUN3 86] and BBN Butterfly multiprocessors [BBN 88] are discussed.

4.1 Hardware Assumptions

It is assumed that a processor accesses (reads and writes) physical memory via MMU hardware. MMU hardware is used to implement the bindings that allow processes to access datum-containers. A binding is an association between an address and a datum-container. A datum-container is implemented with one or more memory-cells. The memory hardware permanently binds a physical address to a memory-cell. MMU hardware dynamically binds a region of processor addresses (a processor page) to a region of memory-cells (a physical page).

An MMU with the following features is assumed.

- Memory is organised into fixed sized pages.
- Translation tables *map* processor pages to physical pages.
- There is no limit to the number of processor pages that can be mapped to a single physical page.
- A processor page is either mapped to a physical page or not mapped. If the processor attempts to access memory with an address from a processor page that is not mapped then an exception[1] occurs.

[1]An exception is caused by the processor but it is like a hardware interrupt. An exception

- A mapping (from a processor page to physical page) provides either READ-WRITE or READ-ONLY access. If the processor attempts to write to memory with an address in a processor page that is mapped with READ-ONLY access then an exception occurs.
- A mapping provides either NON-PRIVILEGED or PRIVILEGED access. If the processor is not in PRIVILEGED mode and it attempts to access an address in a processor page that is mapped with PRIVILEGED access then an exception occurs.

In addition, on multiple processor systems, it is assumed that each processor has its own translation table. Separate processes cannot execute concurrently on separate processors that share a translation table because processes do not share contexts.

4.2 Kernel Notion

A *kernel* is a program that implements the process abstraction and primitive operations of a model.

Kernels executing on separate machines cooperate to implement the abstractions across machine boundaries.

4.3 Deferred Copying

Unnecessary data copying can be avoided under specific circumstances by using the deferred copying implementation technique called copy-on-write[2]. If a DRegion is passed using DUPLICATE and shared physical memory is available then copying data

handler routine is asynchronously invoked when an exception occurs.

[2]The Accent [Fitzgerald 86], Mach [Accetta 86] and Chorus [Abrossimov 89] implementations use the copy-on-write technique.

from the original DRegion to the new duplicate DRegion can be deferred until any attempt is made to modify one of the duplicates. If the contents of these duplicate DRegions are not modified before one of them is deleted then unnecessary data copying is avoided.

The copy-on-write technique can be implemented with DRegions as follows[3]. Before a DUPLICATE data passing operation the source ARegion's processor page is mapped to the DRegion's physical page P. During the DUPLICATE operation the kernel maps a destination ARegion processor page to P with READ-ONLY access and marks the binding as COPY-ON-WRITE. If the source ARegion's page (or any other page) is mapped with READ-WRITE access then its access is also changed to READ-ONLY and the binding is marked COPY-ON-WRITE. The processes with bindings to these duplicate DRegions can read the contents of the page. If one of those processes attempts to write to the page then an exception occurs causing the kernel to invoke the exception handler routine. The exception handler detects that copying is deferred, copies the data to a free page, maps this page with READ-WRITE access in the exception causing process's context and restarts the write instruction.

It is also possible to defer the allocation of the physical page and data structures required for the new duplicate DRegion. However, care must be taken to ensure that the failure semantics of the IPC operation are supported. The IPC operations return an error value if there are insufficient resources to duplicate a DRegion. If the allocation of resources is deferred then an error value cannot be return. Therefore, the process must wait until sufficient resources become available[4].

[3]This explanation assumes that ARegions and DRegions consist of a single page (see Section 4.4)

[4]The current implementation incorrectly defers the allocation of resources and terminates the process if there are insufficient resources when a copy-on-write exception occurs.

4.4 Design Considerations

The following design considerations and decisions are motivated by the goal of demonstrating efficient implementation of the data passing IPC operations when shared physical memory is available.

The maximum size of a DRegion is restricted to the page size. If a DRegion's size is less than a full page then memory is wasted; however, if the contents of the DRegion must be copied then time is saved by copying only the relevant data[5].

Limiting DRegions to a single page also simplifies ARegion allocation. Each process has a fixed number of one page ARegions. If an ARegion is unbound then it can be allocated.

DRegions are implemented without information about the ARegions that are bound to the DRegion because maintaining a list of the bindings would be space and time inefficient. The amount of space required for each DRegion to keep a list of the ARegions bound to it is not known in advance. Therefore, memory would have to be dynamically allocated.

Data copying is not deferred if the DRegion being duplicated is shared to avoid implementation complexity and save time. Deferred copying of shared DRegions requires data structures that provide indirection[6] to ensure the bindings to a shared DRegion are properly updated when a write attempt causes the deferred copy to be performed. This indirection adds complexity and takes time. In addition, a likely reason for sharing a DRegion is to provide access to data that is expected to be modified. If copying is deferred but a write operation causes the exception handler

[5]MMUs, like the MC68851[Motorola 89b], have a feature to restrict access to a specific range of bytes in a page. The size of DRegions can be enforced with this feature. If the MMU does not have this feature then processes can access the entire page regardless of the DRegion size; however, the kernel only ensures that size bytes are shared, moved or duplicated.

[6]Mach [Rashid 88] provides indirection with shadow objects and shared objects to implement deferred copying of shared memory objects.

to perform the copy then a significant amount of time is wasted (see Chapter 5).

A process's context is separated into permanent and dynamic bindings for two reasons. First, existing programming language compilers are not designed to deal with pages of the execution stack being moved or shared; therefore permanent bindings are required for the stack. Second, the purpose of the implementation is to evaluate the performance of the model; therefore, effort was not put into integrating heap allocation with ARegion allocation.

4.5 Object-Based Specification

A general object-based specification is presented for the following Regions primitives: **create-region**, **delete-region**, **rebind-region**, **size-of-region**, **state-of-region**, **pass-region-to**, and **pass-region-from**. Partial algorithms for **send**, **receive** and **reply** are provided in Chapter 5.

The purpose of this specification is to provide a machine independent description of how the model can be implemented.

A notation similar to C++ [Stroustrup 86] is used for the specification. An object has an interface that specifies the (public) operations that can be invoked on the object. An object can also have private variables and private operations that are used to implement the public operations. Every interface provides operations called create and delete. The create operation allocates and initialises an instance of an object. The delete operation deallocates an instance of an object.

The following syntactic conventions are used. Names of variable and object instances are lower case. Names of variable types and object interfaces are capitalised. Constants are upper case. Prose describing implementation steps are in italics and parentheses. Comments are delimited by double slashes (//).

The object-based specification is presented as follows. The variable types are

defined. Then the Regions primitives are presented as public operations of an object type called Kernel. A specification of each public operation follows. These specifications are based on variables, operations and objects that are private to the Kernel interface. A specification of each private operation is presented. Then the private objects are specified.

4.5.1 Variable Types

The following variable types are used in the Kernel specification.

Public Variable Types
- Boolean (TRUE or FALSE)
- Integer //Standard integer.//
- Address //A nonnegative integer.//
- ARegion //Starting address of a processor page.//
- State (INVALID, UNBOUND, PRIVATE, or SHARED) //ARegion state.//
- Mode (DUPLICATE, MOVE, or DYNAMIC-SHARE) //Data passing mode.//
- PID //Process identifier.//

Private Variable Types
- Paddress //Starting address of a physical page of memory.//
- Access (READ-ONLY, or READ-WRITE) //MMU page mapping attribute.//

4.5.2 The Kernel Object Interface

The Regions primitives are public operations of the Kerenl object interface.

INTERFACE *Kernel* {
- **create**() *returns* Kernel::k //Initialise the kernel.//
- **delete**() //Clean up before terminating the kernel.//
- **create-region**(Integer size) *returns* (ARegion) ar
- **delete-region**(ARegion ar)
- **rebind-region**(ARegion ar_s, ARegion ar_d) *returns* (Boolean) success
- **size-of-region**(ARegion ar) *returns* (Integer) size
- **state-of-region**(Address addr) *returns* (State) s
- **pass-region-to**(PID sender, ARegion ar_s, Mode m) *returns* (ARegion) ar_d
- **pass-region-from**(PID sender, ARegion ar_s, Mode m) *returns* (ARegion) ar_d

...//The remaining public Kernel operations are not relevant.//
Private Variables
PID current-pid //The PID of the currently executing process.//
Private Object Instances
Context::current-context //The context currently installed in the MMU.//
Kernel::this-kernel //This Kernel object (the local kernel).//
}

4.5.3 Regions Primitive Specification

The Kernel public operations are specified in terms of variables, operations and objects that are private to a Kernel object.

A Kernel object is created to start the system.

Kernel::create() {
(*Allocate an object with interface Kernel.*) *returns* k
(*Create a stack of free physical pages.*)
(*Create a stack of free DRegion objects.*)
...//Nonrelevant initialisation actions.//
this-kernel ←k
return k
}

When the system is shutdown the Kernel object is deleted.

Kernel::delete() {
(*Initiate delivery of delete requests for DRegion shared with other kernels.*)
...//Nonrelevant shutdown actions.//
(*Deallocate self.*)
}

Kernel::**create-region**(Integer size) {
current-context→get-unbound-binding() *returns* Binding::b
create(size) *returns* DRegion::dr
b→bind(dr, FALSE) //Not a deferred copy binding.//
b→get-ARegion() *returns* (ARegion) ar
return ar
}

```
Kernel::delete-region(ARegion ar) {
    current-context→get-binding(ar) returns Binding::b
    b→unbind()
}

Kernel::rebind-region(ARegion ar_s, ARegion ar_d) {
    current-context→get-binding(ar_s) returns Binding::b_s
    current-context→extract-binding(ar_d) returns Binding::b_d
    do-rebind(b_s, b_d)
    return TRUE
}

Kernel::size-of-region(ARegion ar) {
    current-context→get-binding(ar) returns Binding::b
    b→get-DRegion() returns DRegion::dr
    dr→get-size() returns (Integer) size
    return size
}

Kernel::state-of-region(Address addr) {
    if ( addr is an ARegion ) then
        current-context→get-binding(addr) returns Binding::b
        b→get-bound() returns (Boolean) bound
        if ( bound ) then
            b→get-DRegion() returns DRegion::dr
            dr→get-shared() returns (Boolean) shared
            if ( shared ) return SHARED
            return PRIVATE
        return UNBOUND
    return INVALID
}

Kernel::pass-region-to(PID sender, ARegion ar_s, Mode m) {
    get-kernel(sender) returns Kernel::k_d
    if ( k_d is this-kernel ) then
        get-context(sender) returns Context::c_d
        local-pass(current-context, c_d, ar_s, m) returns (ARegion) ar_d
```

```
    else
        current-context→get-binding(ar_s) returns Binding::b_s
        if ( m is DYNAMIC-SHARE ) then
            do-deferred-copy(b_s) returns DRegion::dr
            dr→inc-copiers() //Avoid deallocating dr (see Section 4.7).//
            deliver(k_d, "to-bind", sender, dr)
        if ( m is MOVE ) then
            b_s→get-DRegion() returns DRegion::dr
            dr→get-shared() returns (Boolean) shared
            dr→inc-copiers() //Avoid deallocating dr (see Section 4.7).//
            if ( shared ) then deliver(k_d, "to-bind", sender, dr)
            else deliver(k_d, "to-copy", sender, dr)
            b_s→unbind()
        if ( m is DUPLICATE ) then
            b_s→get-DRegion() returns DRegion::dr
            dr→get-shared() returns (Boolean) shared
            dr→inc-copiers() //Avoid deallocating dr (see Section 4.7).//
            deliver(k_d, "to-copy", sender, dr)
        block(current-pid) returns (ARegion) ar_d
    return ar_d
}

Kernel::pass-region-from(PID sender, ARegion ar_s, Mode m) {
    get-kernel(sender) returns Kernel::k_d
    if ( k_d is this-kernel ) then
        get-context(sender) returns Context::c_d
        local-pass(c_d, current-context, ar_s, m) returns (ARegion) ar_d
    else
        deliver(k_d, "from-req", sender, ar_s, m)
        current-context→get-unbound-binding() returns Binding::b_d
        b_d→get-ARegion() returns (ARegion) ar_d
        block(current-pid) returns DRegion::dr
        dr→get-shared() returns (Boolean) shared
        if ( m is MOVE) then
            if ( shared ) then (Set m to DYNAMIC-SHARE.)
            else (Set m to DUPLICATE.)
        if ( m is DYNAMIC-SHARE ) then
```

```
            b_d→bind(dr, FALSE)
        if ( m is DUPLICATE ) then
            dr→get-size() returns (Integer) size
            create(size) returns DRegion::dr_0
            copy(dr, dr_0)
            dr→dec-copiers() //dr can now be deallocated (see Section 4.7).//
            b_d→bind(dr_0, FALSE)
    return ar_d
}
```

4.5.4 Private Kernel Operations

The following private operations are used to implement the Kernel public operations.

```
Private Operations
    get-process(Context::c) returns (PID) p
    get-context(PID p) returns Context::c
    get-kernel(PID p) returns Kernel::k

    block(PID p) //Suspend a process.//
    schedule(PID p) //Unsuspend a process.//

    do-rebind(Binding::b_s, Binding::b_d)

    local-pass(Context::c_s, Context::c_d, ARegion ar_s, Mode m) returns (ARegion) ar_d
    do-deferred-copy(DRegion::dr, Binding::b)
    read-only-exception-handler(PID p, Address addr)
    copy(DRegion::dr_s, DRegion::dr_d)

        //Deliver a request to another Kernel. The other Kernel invokes//
        // name(arg_1, arg_2, ...). The sending Kernel (this-kernel) and//
        // calling process (current-pid) variables are passed implicitly.//
    deliver(Kernel::k, "name", arg_1, arg_2, ...)

        //The following routines can be remotely invoked with deliver().//
    to-bind(PID p, DRegion::dr)
    to-copy(PID p, DRegion::dr)
```

to-done(PID p, ARegion ar_d)
from-req(PID p, ARegion ar_s, Mode m)
from-done(PID p, DRegion::dr)

The following are sketches of algorithms that implement the private operations.

```
Kernel::get-process(Context::c) {
    (Find process executing in c) returns (PID) p
    return p
}

Kernel::get-context(PID p) {
    (Find context that p is using.) returns Context::c
    return c
}

Kernel::get-kernel(PID p) {
    (Find kernel where p is executing.) returns Kernel::k
    return k
}

Kernel::block(PID p) {
    suspend(p) //Remove from ready queue.//
    if ( p is current-pid ) then
        (Switch to next ready process.)
}

Kernel::schedule(PID p) {
    unsuspend(p) //Put into ready queue.//
}

Kernel::do-rebind(Binding::b_s, Binding::b_d) {
    b_s→get-DRegion() returns DRegion::dr
    b_s→get-deferred() returns (Boolean) deferred
    b_d→bind(dr, deferred)
    b_s→unbind()
}
```

```
Kernel::local-copy(Context::c_s, Context::c_d, ARegion ar_s, Mode m) {
    c_s→get-binding(ar_s) returns Binding::b_s
    c_d→get-unbound-binding() returns Binding::b_d
    if ( m is DYNAMIC-SHARE ) then
        do-deferred-copy(b_s) returns DRegion::dr
        b_d→bind(dr, FALSE)
    if ( m is MOVE ) then
        do-rebind(b_s, b_d)
    if ( m is DUPLICATE ) then
        b_s→get-DRegion() returns DRegion::dr
        dr→get-shared() returns (Boolean) shared
        if ( shared ) then //Shared DRegions copied immediately.//
            dr→get-size() returns (Integer) size
            create(size) returns DRegion::dr_0
            copy(dr, dr_0)
            b_d→bind(dr_0, FALSE)
        else //Copying of non-shared DRegions deferred.//
            b_s→defer()
            b_d→bind(dr, TRUE)
    b_d→get-ARegion() returns (ARegion) ar_d
    return ar_d
}

Kernel::do-deferred-copy(Binding::b) {
    b→get-DRegion() returns DRegion::dr
    b→get-deferred() returns (Boolean) deferred
    if ( deferred ) then
        dr→get-copiers() returns (Integer) copiers
        if ( copiers is 1 ) then
            b→undefer()
        else
            dr→get-size() returns (Integer) size
            create(size) returns DRegion::dr_0
            copy(dr, dr_0)
            B→bind(dr_0, FALSE)
            return dr_0
    return dr
```

```
}

Kernel::read-only-exception-handler(PID p, Address addr) {
    if ( addr is an ARegion ) then
        (Find ARegion that addr is in) returns (ARegion) ar
        current-context→get-binding(addr) returns Binding::b
        do-deferred-copy(b)
        (Restart instruction that caused exception.)
    else
        (Delete the current process and reclaim its resources.)
}

Kernel::deliver(Kernel::k, "name", arg1, arg2, ...) {
    (Initiate delivery to k of request "name")
    (k is interrupted and invokes name(arg1, arg2, ...))
}
```

The following five routines are invoked as a consequence of deliver(). The first three are associated with **pass-region-to** and the last two are associated with **pass-region-from.**

```
Kernel::to-bind(PID pd, DRegion::dr) {
    get-context(pd) returns Context::cd
    cd→get-unbound-binding() returns Binding::b
    b→bind(dr, FALSE)
    dr→dec-copiers() //dr can now be deallocated (see Section 4.7).//
    b→get-ARegion() returns (ARegion) ard
    deliver(ks, "to-done", ps, ard)
     //Respond to the kernel ks that requested "to-bind" on behalf of process ps.//
}

Kernel::to-copy(PID pd, DRegion::dr) {
    get-context(pd) returns Context::cd
    cd→get-unbound-binding() returns Binding::b
    dr→get-size() returns (Integer) size
    create(size) returns DRegion::dr0
```

```
    copy(DR, dr0)
    dr→dec-copiers() //dr can now be deallocated (see Section 4.7).//
    b→bind(dr0, FALSE)
    b→get-ARegion() returns (ARegion) ard
    deliver(ks, "to-done", ps, ard)
     //Respond to the kernel ks that requested "to-copy" on behalf of process ps.//
}

Kernel::to-done(PID ps, (ARegion) ard) {
    (Pass ard to ps via a kernel data structure.)
    schedule(ps)
}

Kernel::from-req(PID pd, ARegion ars, Mode m) {
    get-context(pd) returns Context::cs
    cs→get-binding() returns Binding::bs
    if ( M is DYNAMIC-SHARE ) then
        do-deferred-copy(bs) returns DRegion::dr
    else
        bs→get-DRegion() returns DRegion::dr
    dr→inc-copiers() //Avoid deallocating dr (see Section 4.7).//
    deliver(ks, "from-done", ps, dr)
     //Respond to the kernel ks that requested "from-req" on behalf of process ps//
}

Kernel::from-done(PID ps, DRegion::dr) {
    (Pass dr to ps via a kernel data structure.)
    schedule(ps)
}
```

4.5.5 Private Kernel Objects

The interface and implementation of private objects are defined for a context, binding, DRegion and MMU translation table page entry.

4.5.5.1 Context Object

A context object is created or deleted whenever a process is created or deleted, respectively.

```
INTERFACE Context {
    create() returns Context::c
    delete()
    get-binding(ARegion ar) returns Binding::b
    extract-binding(ARegion ar) returns Binding::b
    get-unbound-binding() returns Binding::b
    put-unbound-binding(Binding::b)
Private Variables
    Array-of-Bindings db //System defined number of dynamic bindings.//
    Stack-of-Bindings unbound-db //Stack of unbound dynamic bindings.//
}
```

The following are sketches of algorithms that implement the Context operations.

```
Context::create() {
    (Allocate an object with interface Context.) returns Context::c
    (Allocate db and unbound-db)
    for i from 1 to n
        create(c) returns Binding::b
        c→db[i] ←b
        push(b, c→unbound-db)
    (Allocate permanent bindings.)
    return c
}

Context::delete() {
    for i from 1 to n  //Delete all bound ARegions.//
        self→db[i]→get-bound() returns (Boolean) bound
        if ( bound ) then
            self→db[i]→unbind()
    (Deallocate permanent bindings.)
}
```

```
Context::get-binding(ARegion ar) { //Assume ar is not in unbound-db.//
    (Convert ar to i.)
    return self→db[i]
}

Context::extract-binding(ARegion ar) { //Assume ar is in unbound-db.//
    (Convert ar to i.)
    remove(self→db[i], self→unbound-db)
    return self→db[i]
}
```

The remove() operation can be implemented without traversing the stack unbound-db if the elements of db are used to implement the stack as a doubly linked list.

```
Context::get-unbound-binding() {
    pop(self→unbound-db) returns Binding::b
    return b
}

Context::put-unbound-binding(Binding::b) {
    push(b, self→unbound-db)
}
```

4.5.5.2 Binding Object

Dynamic bindings are implemented as Binding objects with the following interface.

```
INTERFACE Binding {
    create(Context::c, ARegion ar) returns Binding::b
    delete()
    get-ARegion() returns (ARegion) ar
    get-bound() returns (Boolean) bound
    get-deferred() returns (Boolean) deferred
    get-DRegion() returns DRegion::dr
    bind(DRegion::dr, Boolean deferred)
    unbind()
    defer()
    undefer()
```

```
Private Variables
    ARegion ar
    Boolean bound
    Boolean deferred
Private Object Instances
    DRegion::dr
    PageEntry::pe
    Context::context
}
```

The following are sketches of algorithms that implement the Binding operations.

```
Binding::create(Context::c, ARegion ar) {
    (Allocate an object with interface Binding.) returns Binding::b
    b→bound ←FALSE
    b→context ←c
    b→ARegion ←ar
    return b
}

Binding::delete() {
    (Deallocate self)
}

Binding::bind(DRegion::dr, Boolean deferred) {
    self→bound ←TRUE
    self→dr ←dr
    self→deferred ←deferred
    create(self→ar, self→context) returns PageEntry::pe
    self→pe ←pe
    dr→get-paddr() returns (Paddress) paddr
    if ( deferred ) then
        dr→inc-copiers()
        pe→bind(paddr, READ-ONLY)
    else
        dr→inc-writers()
        pe→bind(paddr, READ-WRITE)
}
```

```
Binding::unbind() {
    self→bound ←FALSE
    self→context→put-unbound-binding(self)
    if ( self→deferred ) then
        self→dr→dec-copiers()
    else
        self→dr→dec-writers()
    self→pe→unbind()
}

Binding::defer() {
    if ( not self→deferred ) then
        self→deferred ←TRUE
        self→pe→read-only()
        self→dr→inc-copiers()
        self→dr→dec-writers()
}

Binding::undefer() {
    if ( self→deferred ) then
        self→deferred ←FALSE
        self→pe→read-write()
        self→dr→inc-writers()
        self→dr→dec-copiers()
}
```

The routines defer() and undefer() increment then decrement the DRegion counters so that another kernel can read the counter values without requiring mutual exclusion (see section 4.7).

4.5.5.3 DRegion Object

DRegions are implemented as DRegion objects with the following interface.

```
INTERFACE DRegion {
    create(Integer size) returns DRegion::dr
    delete()
```

```
    get-size() returns (Integer) size
    get-paddr() returns (Paddress) paddr
    get-copiers() returns (Integer) copiers
    get-shared() returns (Boolean) shared
    inc-copiers()
    inc-writers()
    dec-copiers()
    dec-writers()
Private Variables
    Paddress paddr
    Integer size
    Integer writers
    Integer copiers
}
```

DRegions can be shared between Kernels therefore the implementation of shared DRegions might be distributed (see Section 4.8).

The following are sketches of algorithms that implement the DRegion operations.

```
DRegion::create(Integer size) {
    (Allocate an object with interface DRegion.) returns DRegion::dr
    (Allocate page.) returns (Paddress) paddr
    dr→paddr ←paddr
    dr→size ←size
    dr→writers ←0
    dr→copiers ←0
    return dr
}

DRegion::delete() {
    (Deallocate self)
}

DRegion::get-shared() {
    return (Boolean) (self→writers > 1)
}
```

```
DRegion::inc-copiers() {
    (self→copiers)++ //Must detect and prevent overflow.//
}

DRegion::inc-writers() {
    (self→writers)++ //Must detect and prevent overflow.//
}

DRegion::dec-copiers() {
    (self→copiers)-
    if ( (self→copiers + self→writers) < 1 ) then
        delete() //DRegion is deallocated if it is not bound.//
}

DRegion::dec-writers() {
    (self→writers)-
    if ( (self→copiers + self→writers) < 1 ) then
        delete() //DRegion is deallocated if it is not bound.//
}
```

4.5.5.4 PageEntry Object

The translation table entries that bind a processor page to a physical page are implemented as PageEntry objects with the following interface.

```
INTERFACE PageEntry {
    create(ARegion ar, Context::c) returns PageEntry::pe
    delete()
    bind(Paddress paddr, Access a)
    unbind()
    read-only()
    read-write()
Private Variables
    Paddress paddr
    Access access
    Boolean valid
    ARegion ar
Private Object Instances
```

```
    Context::context
}
```

The PageEntry operations can be implemented as follows.

```
PageEntry::create(ARegion ar, Context::c) {
    if ( Translation table for c does not have an entry for ar. ) then
        (Allocate an entry for ar.) returns PageEntry::pe
        pe→ar ←ar
        pe→valid ←FALSE
        pe→context ←c
        if ( c is installed in MMU ) then (Load entry into MMU.)
    else
        (Find the entry for ar.) returns PageEntry::pe
    return pe
}

PageEntry::delete() {
    if ( Translation table does not need an entry for self→ar ) then
        (Deallocate entry for self→ar.)
        if ( self→context is installed in MMU ) then (Invalidate entry in MMU.)
}

PageEntry::bind(Paddress paddr, Access a) {
    self→paddr ←paddr
    self→access ←a
    self→valid ←TRUE
    if ( self→context is installed in MMU ) then (Load entry into MMU.)
}

PageEntry::unbind() {
    self→valid ←FALSE
    delete() //Check to see if the entry should be deallocated.//
}

PageEntry::read-only() {
    self→access ←READ-ONLY
}
```

```
PageEntry::read-write() {
    self→access ←READ-WRITE
}
```

4.6 Sun 3 Uniprocessor Implementation

The Regions IPC model is implemented on Sun 3 workstations[7].

The kernel implementation is similar to the object-based specification in the previous section. The kernel is written in C [Kernighan 78]. The Kernel private objects are global variables. The private operations and private object operations are implemented in line (not as procedures).

PageEntry objects are implemented directly with the Sun 3 MMU translation tables.

The Sun 3 MMU groups pages into segments where a segment contains 16 pages. The translation tables consist of a table of segment entries and groups of 16 page entries called pmegs[8]. A segment entry can point at any one of 255 pmegs.

The page-entry objects are implemented directly with pmegs. If a page-entry is required for a logical page starting at address Addr then the segment containing Addr must be mapped to a pmeg. If Addr's segment is not mapped, a pmeg is popped off a stack of free pmegs, the 16 page entries are marked invalid and Addr's segment is mapped to it. Addr's page-entry is allocated by marking it valid. A page-entry is deallocated by marking it invalid. If a segment is mapped to a pmeg with no valid page-entries then the pmeg is pushed onto a stack of free pmegs and the segment is unmapped.

[7]The Sun 3 implementation has not yet been extended across the Ethernet. Therefore, DRegions can only be passed between processes executing on the same workstation.

[8]The Sun 3 MMU also provides 8 contexts where each context has a segment table and 255 pmegs; however this feature was not used.

Pmegs are a limited resource on Sun 3 workstations. Therefore the performance of this implementation will be degraded when all pmegs are being used.

If a non-shared DRegion is passed with the DUPLICATE mode between processes executing on a workstation then copying is deferred.

4.7 BBN GP1000 NUMA Multiprocessor Implementation

The Regions IPC model is also implemented on BBN GP1000 multiprocessors.

Each node of the GP1000 has a processor, an MMU and a local memory. The MMU can map processor pages to the node's local physical memory or to the local physical memory of other nodes (remote memory). However, the ratio between the time to access (read or write) remote memory as opposed to local memory is between 8 and 11 in the absence of contention[9]. Therefore, the a separate kernel is executed on each node because an efficient implementation must avoid remote memory accesses.

The kernel implementation for each node of the GP1000 is similar to the kernel implementation for a Sun 3 workstation. The GP1000 MMU is configured to behave similar to the Sun 3 MMU but the variable types and instructions sequences differ. The GP1000 implementation also supports passing DRegions between separate nodes.

A kernel causes another kernel to invoke an operation by delivering a request specify the operation and parameters. Each node has a multi-producer single-consumer queue that any processor can access. The requests are inserted into the destination kernel's queue by first ensuring that no other kernel is delivering a request. Then the source kernel interrupts the destination kernel. The destination kernel interrupt handler routine removes the requests from the queue and invokes the requested

[9]Contention occurs when multiple processors contend for the same hardware to access remote memory.

operation.

Operations that involve more than one node are implemented so that the kernels on each node can execute in parallel. Delivery of a request is initiated before all the arguments are specified. Each process has a data structure associated with it that is used to send remote requests to other nodes[10]. Therefore, every kernel has access to the remote request buffers of every node. When a remote request is delivered the type of request is inserted in the buffer B, B is marked BUSY, a pointer to B is inserted in the destination node's queue, and the destination processor is interrupted. The source kernel inserts the remaining parameters in B then marks B COMPLETE. The destination kernel interrupt handler removes the request from its queue, invokes the requested routine, performs as many operations as it can and then waits until B is marked COMPLETE.

If a DRegion is passed to a process executing on the same node then the kernel executes the same operations as the Sun 3 implementation. Therefore, copying is deferred if the passing mode is DUPLICATE and the DRegion is not shared.

If a DRegion is passed to a process executing on another node then the DRegion is copied if possible. If the mode is DUPLICATE then the DRegion is copied. If the mode is MOVE and the DRegion is not shared then the DRegion is copied. Otherwise the DRegion is shared and the destination ARegion is bound to the DRegion.

The implementation must delay the deallocation of a page that is being copied until copying is completed because a deallocated page can be reallocated and modified[11]. If the source process is asynchronously terminated then the process's resources are reclaimed. The deferred copy feature is used to avoid this problem. The DRegion copiers counter is incremented before page copying starts. When the copy is complete

[10]Process descriptors are used in the current implementation. The alternative is to dynamically allocate request buffers.

[11]The stack of free pages is currently implemented by storing a pointer in the page. Therefore, a page is modified when it is deallocated.

a request to decrement copiers is delivered. This also allows the kernel to unbind the source when the mode is MOVE without waiting for a confirmation.

If a DRegion is shared between processes executing on different nodes then the DRegion object is only implemented on the node where the DRegion was created. If a kernel invokes an operation on a remote DRegion object then there are two possibilities: the operation does or does not modify the DRegion object's variables. If the operation only reads the DRegion object's variables then the operation can be implemented by remotely reading[12] the contents of the variables[13]. If the operation modifies the DRegion variables then a request is delivered to the remote kernel to perform the operation.

4.8 Discussion

A DRegion can be shared between processes executing on processors that do not share physical memory. Two techniques for implementing this sharing are *remote access* and *distributed sharing.*

The current GP1000 implementation uses the remote access technique. The data exists on the node where the DRegion was created. If the DRegion is shared with a process P executing on a separate node then an ARegion in P's context is mapped to the remote physical page. Read and write operations by P on the shared DRegion are remote memory accesses. The remote access sharing technique has also been demonstrated across a communications network [Spector 82]. **write** operations have at-most-once semantics because a communication failure can prevent confirmation that the write was completed from being delivered.

[12]Remote access to DRegion variables requires providing every kernel with access to every nodes DRegion variables.

[13]The DRegion operations defer() and undefer() increment then decrement the DRegion counters to ensure that incorrect transient values cannot be read.

Distributed sharing is an alternative technique involving *migrating* and *replicating* the data. A DRegion is shared between processes executing on the same machine and separate machines. Initially the data is in a physical page that is local to a machine where one of the sharing processes is executing. The physical page can be mapped to a processor page in the context of any sharing process executing on that machine. Those processes can read or write the contents of the page. The processor pages of sharing processes executing on other machines are unmapped and the bindings are marked NON-RESIDENT.

If a process attempts to read data from a processor page that is not mapped but where the binding is marked NON-RESIDENT then an exception occurs. The kernel X on that machine delivers a *replicate* request to the kernel Y on the machine where the data exists. Y changes its mappings to the page containing the data to READ-ONLY access and delivers a copy of the data to X. X maps the page containing the copied data with READ-ONLY access and restarts the read instruction. Subsequent attempts to read a page marked NON-RESIDENT result in additional READ-ONLY mappings to copies of the data. The data is cached on the machines that require read access.

If a process attempts to write data to a page that is mapped READ-ONLY or a binding that is marked NON-RESIDENT then an exception occurs. The kernel X on that machine delivers a *migrate* request to the other kernels where the page is mapped. Those kernels unmap the page that contains their copy of the data and mark their bindings NON-RESIDENT. If X does not have a copy of the data then it also requests a copy from on of the other kernels. X maps the page containing the data with READ-WRITE access and restarts the write instruction.

The distributed sharing technique has been demonstrated across a communications network [Li 86]. **write** operations have exactly-once semantics because the datum is not written until the page of data arrives.

The remote access technique is less efficient than the distributed sharing technique if processes on one node must perform many remote accesses while no other processes are performing accesses. However, remote access is more efficient if processes on different nodes alternately access the data. Methods of dynamically selecting the remote access or distributed share techniques have been demonstrated and analysed for NUMA systems [Bolosky 91].

4.9 Summary

Issues related to a correct efficient implementation of the Regions IPC model are discussed. Several design choices were made with emphasis on simplicity and efficiency.

The implementation takes advantage of MMU hardware and assumes the MMU hardware has specific features.

Data copying is deferred when a non-shared DRegion is passed with the DUPLICATE mode between processes executing on processors that share physical memory. Data copying is not deferred when the source DRegion is shared.

A system independent object-based specification is presented. Then the details of implementations on Sun 3 workstations and BBN GP1000 Butterfly multiprocessors are discussed.

Two techniques of implementing shared DRegions across communications channels are discussed.

Chapter 5

Performance of the Regions IPC Primitives

The elapsed time performance of the Regions IPC primitives implemented on a Sun 3 workstation and a BBN Butterfly multiprocessor are analysed. The elapsed time of data-passing primitives using the dynamic-share, move, and duplicate data-passing modes are compared with the elapsed time of a primitive that uses the copy data-passing mode. Total elapsed times and component elapsed times are presented. The component times identify where the time is being spent.

5.1 Hardware

Implementations of the Regions IPC model on a Sun Microsystems Sun 3/75 workstation and a 12 node BBN Butterfly GP1000 multiprocessor were used to obtain the elapsed time measurements.

The Sun 3/75 processor is a MC68020 [Motorola 89a] running at 16.7 MHz. The memory cycle time is 270 ns. The MIPS rating is 1.5 .

The processor of each GP1000 node is also a MC68020 running at 16.7 MHz. Timing measurements of identical instruction sequences confirm that the Sun 3/75 and GP1000 execute at the same rate when memory is not referenced and show that the GP1000 is about 1.15 times slower when local[1] memory is referenced.

The Sun 3/75 and GP1000 do not provide a data cache. The MC68020 provides

[1]Each node of the GP1000 has a processor and memory. A processor can directly access the memory of other nodes via a communications network called the Butterfly switch. Timing measurements of identical memory reference instruction sequences on the GP1000 show that instructions that reference memory are about 8 to 11 times slower when the memory is on a remote node as opposed to a local node.

an instruction cache.

The Sun 3 MMU is described in Section 4.6. The MMU uses private memory to store the translation table[2].

Each node of the GP1000 uses a MC68851 MMU [Motorola 89b]. The translation table is stored in the node's main memory. The MMU caches page translations in an *on-chip* address translation cache (ATC). One or more cache entries might have to be invalidated when a translation table entry is modified. The MC68851 is configured to behave similar to the Sun 3 MMU (128K byte segments and 8K byte pages) therefore, an ATC cache miss requires at least two memory accesses. Cache entries of a context are invalidated when a context switch occurs[3]. An efficient way of invalidating context cache entries is to invalidate the entire ATC. A feature that locks a page translation in the ATC is used to avoid invalidating kernel page translations with every context switch.

5.2 Software

The Regions IPC model has been implemented by modifying the W System [Vasudevan 88] distributed operating system. The kernel program and process programs are written primarily in the C programming language [Kernighan 78] and compiled with the Sun Microsystems SunOS 4.1 C compiler. Assembly language is used for:

- interrupt control,
- switches between PRIVILEGED and NON-PRIVILEGED mode (invoking a primitive; responding to an interrupt or exception),
- process context switches (including switching the 68020 state),
- accessing the Sun 3 MMU translation tables, and

[2]A translation table is also called a translation look-a-side buffer (TLB).

[3]The 68851 feature that distinguishes between page translations from separate contexts was not used because of implementation effort and anticipated insignificant time reductions.

- copying more than about 64 bytes of data (the C compiler does not use the processor's instruction cache to minimise copying time).

5.3 Measurement Techniques

The total elapsed time of a primitive is obtained by measuring the elapsed time to execute the primitive a large number of times (10000) and dividing that elapsed time by the number of iterations. Real time clocks in the Sun 3/75 (10 ms resolution) and GP1000 (62.5 μs resolution) were used for the measurements.

The elapsed times of individual components of a primitive are obtained by inserting instructions into the kernel to toggle an external signal[4].

An HP5402A digital oscilloscope was used to measure the average, minimum and maximum width of the resulting pulses. Pulse generation overhead time T_{ov} was measured by executing instructions to turn on and turn off the signal without intervening code. T_{ov} was subtracted from the average pulse width measurements.

T_{ov} had a variance of about 1 μs depending on the position of the instructions in memory and possibly other factors. Therefore, these measurements are approximate[5] indications of the time spent on each component of a primitive.

The purpose of measuring component times is to determine where the time is being spent. Effort was not placed on error analysis or improvement of accuracy and precision because approximate measurements are sufficient for identifying which components take the most time. As a consequence the sum of the component times can differ from the corresponding total times.

[4]The Sun 3/75 serial port RTS line and a control line attached to a GP1000 LED were used.

[5]This technique was used to measure individual non-memory referencing instructions on the GP1000, Sun 3/75, Sun 3/50 and Sun 3/60. Those measurements correlated with the 68020 specifications [Motorola 89a]. However, the pulse widths were on the order of milliseconds; therefore, the variance of T_{ov} was not significant.

Effort spent collecting the component measurements was reduced by instrumenting the entire kernel with signal control instructions. Those instructions could be included in or excluded from the compiled kernel. If they are included a user interface is used to successively select the component to measure.

5.4 Data Copying Primitives

copy-data-to and **copy-data-from** [Cheriton 88, Vasudevan 87] are IPC primitives that use the copy data passing mode. These primitives are not part of the Regions IPC model but they are part of the W System. The performance of **copy-data-to** and **copy-data-from** can be compared directly with the performance of **pass-region-to** and **pass-region-from.**

copy-data is used to refer to both **copy-data-to** and **copy-data-from.**

pass-region is used to refer to both **pass-region-to** and **pass-region-from.**

5.5 Total Elapsed Times

The total elapsed times of the Regions IPC primitives executing on a Sun 3/75[6] and GP1000 are presented in Tables 5.1 and 5.2.

If communicating processes are executing on the same node (or workstation) then communication is *local*, otherwise communication is *remote.*

If communication is local the kernel passes a DRegion by modifying the MMU translation table entries and only copies the DRegion contents when necessary. If communication is remote and the DRegion is not shared then the kernel passes a

[6]The current implementation does not support **pass-region** between Sun 3/75 workstations but it does support **send-receive-reply** and **copy-data**. Unoptimised total elapsed times for IPC between processes executing on separate 3/75s connected by a 10 Mbit Ethernet are: 1.6 ms for an SRR transaction, 1.3 ms for a 1 byte **copy-data**, 2.7 ms for a 1474 byte **copy-data-to** and 3.0 ms for a 1474 byte **copy-data-from**. An Ethernet packet has room for 1474 bytes of process data.

SRR Transaction Total Elapsed Time (μs)			
	Sun 3/75	GP1000	
		local	remote
send-receive-reply	288	398	851

Table 5.1: **send-receive-reply** total elapsed time.

Data Passing Total Elapsed Time (μs)						
	Sun 3/75		GP1000			
			local		remote	
	(1 byte)	(8 KB)	(1 byte)	(8 KB)	(1 byte)	(8 KB)
pass-region-from						
DYNAMIC-SHARE	171	171	271	271	719	719
MOVE	210	210	358	358	.	.
non-shared	.	.	.	.	928	3084
shared	.	.	.	.	(not measured)	
DUPLICATE	.	.	.	.	695	2860
deferred	171	171	270	270	.	.
non-deferred	210	1505	302	1797	.	.
deferred-write	317	1616	568	2060	.	.
copy-data						
COPY	95	1392	128	1627	503	2693

Table 5.2: **pass-region-from** and **copy-data** total elapsed time.

DRegion by creating a duplicate and copying its contents. Therefore, if the MOVE mode is used to pass a nonshared DRegion between GP1000 nodes then a duplicate is created on the destination node and the source is deallocated. Otherwise, the data is passed by modifying MMU translation table entries. The elapsed time to pass a shared DRegion with the MOVE data passing mode between GP1000 nodes was not measured (this time is expected to be approximately the same as the time for passing a DRegion with the DYNAMIC-SHARE mode between GP1000 nodes).

The DUPLICATE data passing mode was implemented with and without deferred copying for local communication so that the times could be compared. Therefore, there are three times for passing DRegions with the DUPLICATE mode.

- *non-deferred*: The DRegion is duplicated before the primitive returns.
- *deferred*: Duplication of the DRegion is deferred and a **write** is not invoked on the DRegion before the binding is deleted therefore it is never duplicated.
- *deferred-write*: Duplication of the DRegion is deferred and a **write** causes the exception handler to duplicate the DRegion.

Every measurement involved two processes: a sender and a receiver. For the **send-receive-reply** measurements the receiver continuously invoked **receive** and **reply** in an infinite loop while the sender invoked **send** to the receiver a fixed number of times. For the other measurements the sender invoked **send** to the receiver passing an address in the message and the receiver invoked the primitive a fixed number of times.

copy-data can copy data continuously between the same source and destination.

pass-region always allocates an ARegion and there are a limited number of ARegions so **pass-region** cannot be invoked continuously without intervening **delete-region** operations.

If the data passing mode is MOVE then the receiver passes the DRegion to and

from the sender during each iteration. The times for MOVE presented in Table 5.2 are half of the total measured time. The current implementation of **pass-region-to** and **pass-region-from** invoke a common procedure for local communication and perform the same components in a different order for remote communication. Therefore, the times of **pass-region-to** and **pass-region-from** are expected to be similar.

If the mode is DUPLICATE or DYNAMIC-SHARE then the receiver passes the DRegion from the sender and deletes the new binding during each iteration[7]. The times shown in the table are the total time minus the time to delete the new binding (see Section 5.6.2 for the delete times).

pass-region-from of a nonshared DRegion between processes executing on separate nodes of the GP1000 is over 200 μs faster if the DUPLICATE mode is used rather than the MOVE mode. This is because the implementation of DUPLICATE allocates a page so that the destination node can copy the data directly into it. Whereas the implementation of MOVE does not allocate a page because a page is not required if the source DRegion is shared.

5.5.1 Benefits and Costs of Avoiding Unnecessary Copying

The elapsed times of the IPC primitives with respect to the number of bytes passed is presented in Figure 5.1 for local communication on a Sun 3/75, Figure 5.2 for local communication on a GP1000 node, and Figure 5.3 for remote communication between nodes of a GP1000.

These three figures demonstrate that for the current implementation the data passing modes that are implemented as MMU translation table manipulations are more efficient than **copy-data** if more than about 1024 bytes are passed. However, **copy-data** is more efficient if less than about 256 bytes are passed.

[7]Elapsed times of **pass-region-to** are not presented but they are expected to be within a few μs for local communication and a few 10's of μs (because of parallelism) for remote communication.

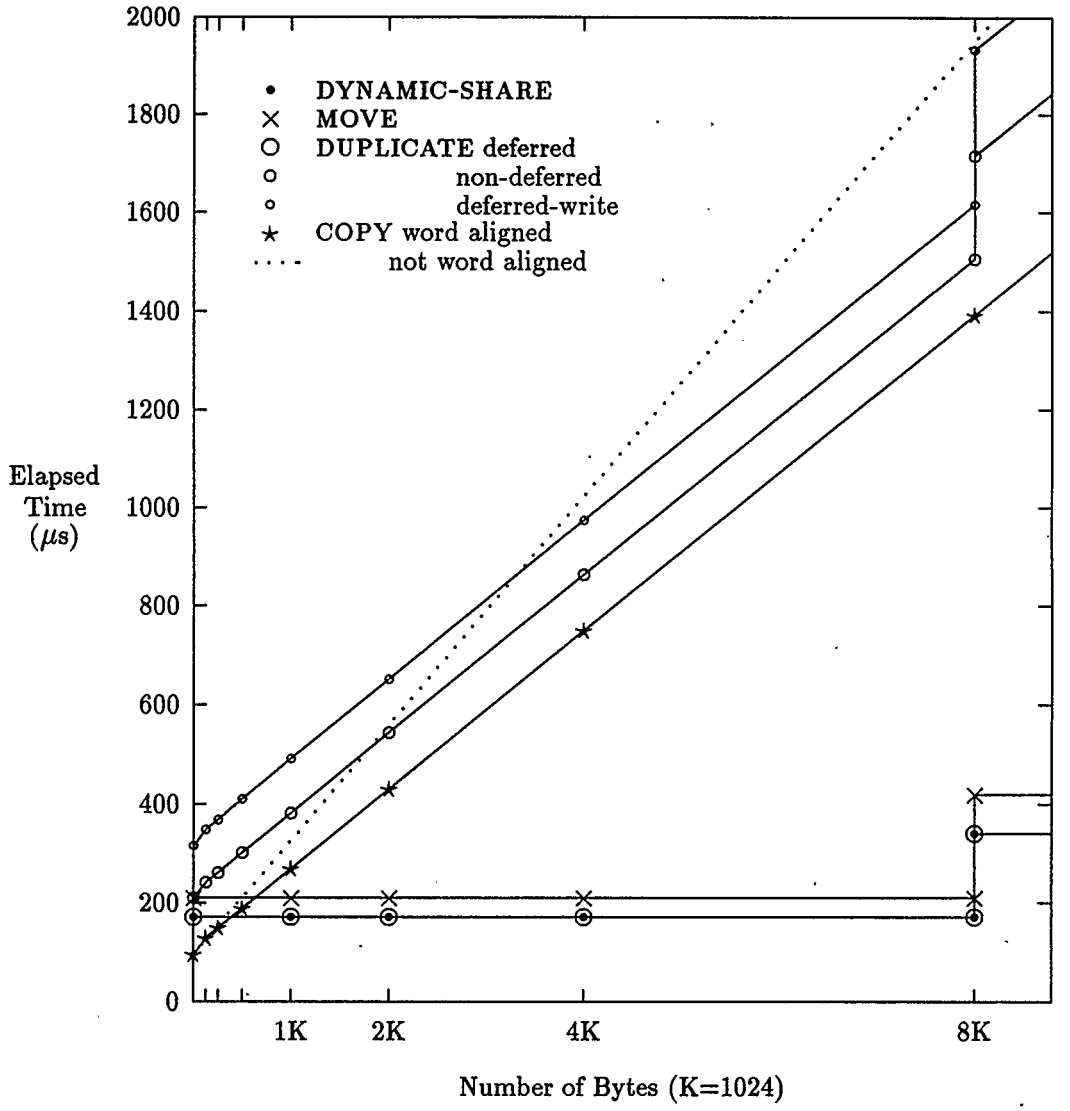

Figure 5.1: Local Sun 3/75 elapsed time versus number of bytes.

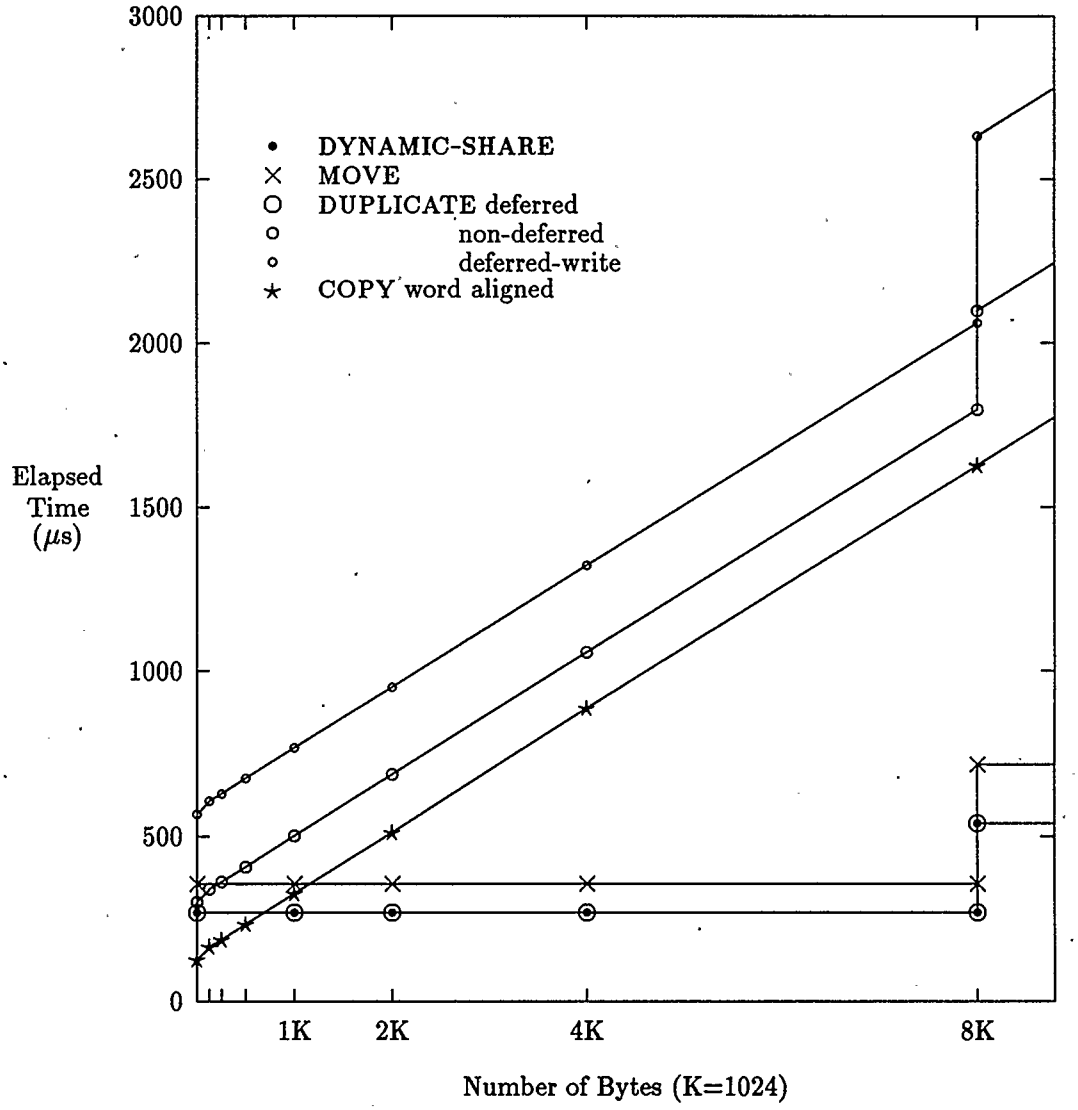

Figure 5.2: Local GP1000 elapsed time versus number of bytes.

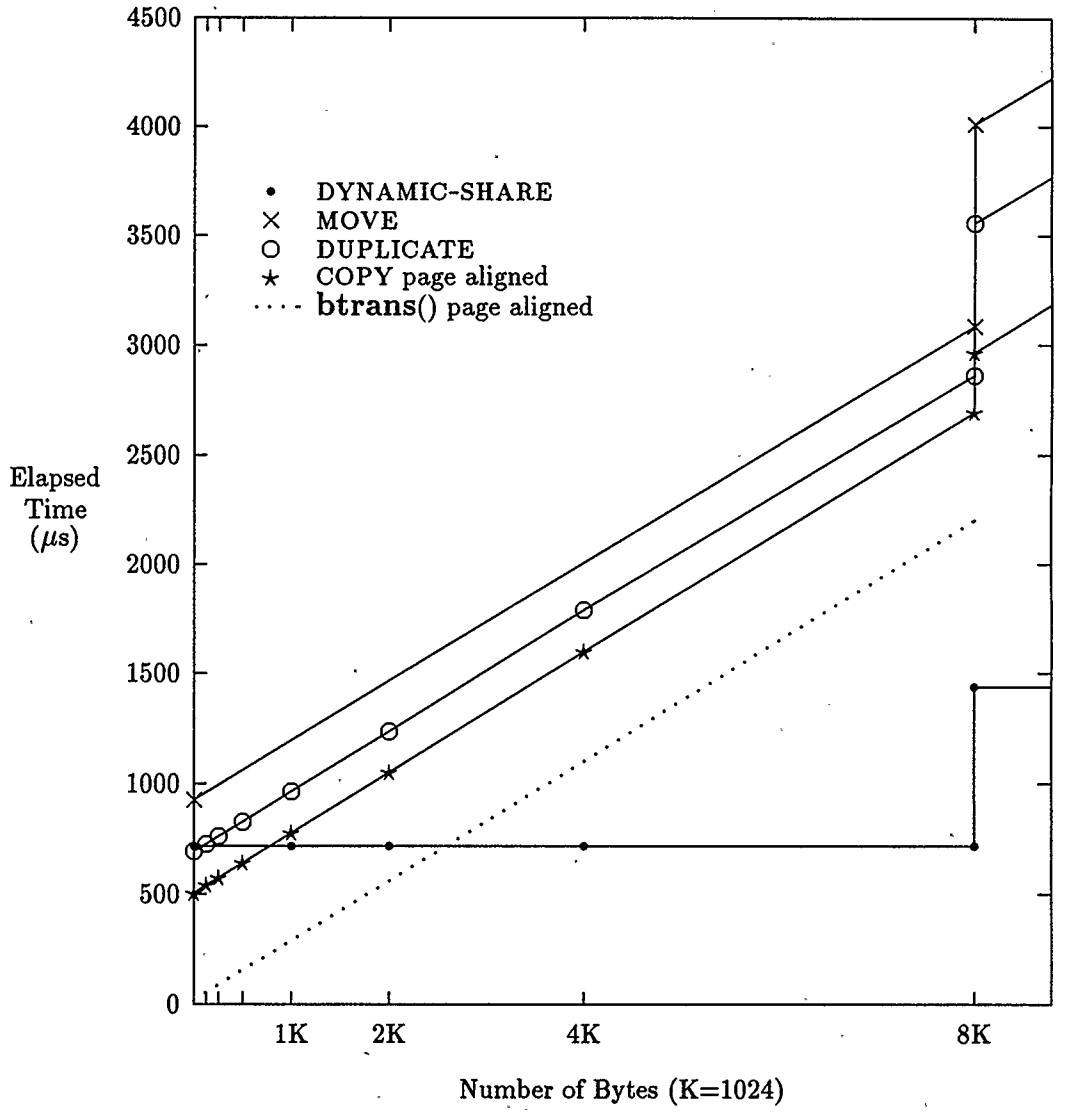

Figure 5.3: Remote GP1000 elapsed time versus number of bytes.

The DYNAMIC-SHARE mode is efficient if the processes are executing on processors that share physical memory because the data only has to be passed once. The performance of sharing data between processes executing on separate nodes of the GP1000 or separate Sun 3 workstations has not be measured.

pass-region with the MOVE mode is only more efficient than **copy-data** if communication is local and there is more than 1024 bytes of data.

pass-region with the DUPLICATE mode is more efficient than **copy-data** if communication is local, duplication is deferred and a **write** is not invoked on either duplicate of the DRegion. It is also more efficient if the source and destination of **copy-data** are not word aligned and there are more than about 4096 bytes of data because of the 68020 memory access characteristics (see Figure 5.1).

5.5.2 Factors that Increase the Elapsed Time

These total elapsed times are the lowest possible times for the current implementation. There are factors that increase the total elapsed time.

When an ARegion is allocated a page entry is required. Page entries are implemented in pmegs as described in Section 4.6. The measurements do not include the allocation or deallocation of pmegs. For the current implementation, pmeg allocation increases the total elapsed time by 64 μs and 54 μs on the Sun 3/75 and GP1000, respectively. Pmeg deallocation increases the total elapsed time by 63 μs and 44 μs on the Sun 3/75 and GP1000, respectively.

No other processes were executing during the measurements. Therefore, when a process is rescheduled, it is the highest priority process. If there are higher priority processes in the ready queue when a process is rescheduled then the processing time will increase by about 10 μs + 2 μs/process on a Sun 3/75.

The receiver does not receive messages from any other senders during the SRR

transaction measurements. If other messages are received and not replied to then the total elapsed time of **send-receive-reply** increases by about 3 μs/process.

The processes had minimum sized contexts: 1 segment (128K byte) for process-code, 1 segment for process-static-data and 4 segments for process-dynamic-data. Context switch elapsed time increases by about 2 μs/segment on a Sun 3/75.

The data copied by **copy-data** do not cross segment boundaries and they are word and page aligned. If the data cross a segment boundary then the elapsed time increases by about 2 μs for each additional segment. If the source and destination data-regions are not word aligned then the total elapsed time increases by about 0.226μs/byte on a Sun 3/75 (as shown in Figure 5.1). If the source and destination are not page aligned then the remote GP1000 elapsed time increases because the most efficient internode data transfer method (**btrans**()) requires physical addresses.

The total elapsed time of remote communication on the GP1000 also increase if there is contention on the GP1000 Butterfly switch or contention for the queue of incoming kernel requests.

5.6 Component Times

The total elapsed times of the primitives are now analysed. The differences between the data passing mode elapsed times are identified.

5.6.1 Passing DRegions

The component times for **copy-data** and **pass-region-from** between processes executing on the same node (or workstation) are presented in Table 5.3 and Table 5.4 for the Sun 3/75 and GP1000, respectively. Component times were not measured for communication between remote nodes of GP1000 because of time constraints.

Local Sun 3/75 Data Passing Component Times (μs)

Component Category	copy-data	pass-region-from			
		DUPLICATE			MOVE
		non -deferred	deferred	deferred -write	
System Call	52	52	52	52	52
Confirm Arguments	21	28	28	28	28
Interrupt Control	3	3	3	3	3
Map Other Context	12	12	12	12	12
Allocate ARegion	.	26	26	26	26
Remap Page	.	.	12	12	12
Update DRegion	.	.	7	7	.
Incur Exception	.	.	.	55	.
Deferred Duplicate?	.	.	.	24	.
Allocate DRegion	.	20	.	17	.
Allocate Page	.	27	.	27	.
Copy Data (1 byte)	8	8	.	8	.
Map New Page	.	.	.	11	.
Deallocate ARegion	.	.	.	.	18
PMEG Empty?	.	.	.	.	20
Overhead	.	31	31	31	35
Component Total	96	207	171	313	206
Measured Total	95	210	171	317	210

Table 5.3: Local Sun 3/75 **pass-region-from** and **copy-data** component times.

The component times for the DYNAMIC-SHARE mode are not presented because

Local GP1000 Data Passing Component Times (μs)					
Component Category	**copy-data**	**pass-region-from**			
		DUPLICATE			MOVE
		non -deferred	deferred	deferred -write	
System Call	62	56	56	56	62
Confirm Arguments	21	32	32	32	32
Interrupt Control	3	4	4	4	4
Map Other Context	22	20	20	20	20
Allocate ARegion	.	34	34	34	34
Remap Page	.	.	69	69	96
Update DRegion	.	.	7	7	.
Incur Exception	.	.	.	66	.
Deferred Duplicate?	.	.	.	35	.
Allocate DRegion	.	20	.	19	.
Allocate Page	.	60	.	60	.
Copy Data (1 byte)	16	16	.	16	.
Map New Page	.	.	.	91	.
Deallocate ARegion	.	.	.	.	18
PMEG Empty?	.	.	.	.	35
Overhead	.	48	48	48	53
Component Total	124	290	270	557	354
Measured Total	128	301	270	568	358

Table 5.4: Local GP1000 **pass-region-from** and **copy-data** component times.

they are very similar to the component times for the DUPLICATE mode when duplication is deferred and **write** is not invoked on the DRegion.

The component times are grouped into the following categories.

System Call: Time to execute system calls.

Confirm Arguments: Time to confirm that argument values are valid.

Interrupt Control: Time to disable and enable interrupts.

Map Other Context: Time to map page entries that are modified or that are required for directly copying the data.

Allocate ARegion: Time to allocate an ARegion.

Remap Page: Time to modify page entries.

Update DRegion: Time to update DRegion counters.

Incur Exception: Time to detect a **write** to a page with READ-ONLY access, invoke the exception handler, and restart the **write**.

Deferred Duplicate?: Time to confirm that the exception is due to a deferred duplicate.

Allocate DRegion: Time to allocate and initialise a DRegion (not including the time to allocate a physical page of memory).

Allocate Page: Time to allocate a physical page of memory.

Copy Data (1 byte): Time to copy 1 byte of data.

Map New Page: Time to map the duplicated page into the context.

Deallocate ARegion: Time to unbind the source ARegion (MOVE).

PMEG Empty?: Time to check if the pmeg is empty and should be deallocated.

Overhead: Time to call the procedure that passes DRegions and to cache values in processor registers.

Component Total: Total of the component times.

Measured Total: Total elapsed times from Table 5.2.

The system call time for **copy-data** and **pass-region-from** with the MOVE mode is greater than the other system call times on the GP1000 because of a measurement artifact. The technique used to measure the component times involved passing a value from the kernel level to the process level indicating whether or not the next primitive was to be measured. This value was passed in the data for these two primitives and the GP1000 incurred an ATC miss when the value was accessed at the process level.

The elapsed times of **copy-data** and **pass-region** can be improved significantly. The system call time for **copy-data** and **pass-region** can be reduced by about 25 μs if the technique described in Section 5.7 is used. The allocation and deallocation of ARegions and DRegions can be improved. The time required to check if a pmeg is empty and to allocate or deallocate a pmeg can be reduced to a few μs. The GP1000 implementation for allocating and mapping pages is very inefficient and should be implemented in assembly language. The overhead time required to invoke the procedure that passes DRegions and cache values in processor registers can be reduced. These improvements were not implemented because of time constraints.

5.6.2 Creating and Deleting Regions

The component times for **create-region** and **delete-region** are presented in Table 5.5.

The following component time categories have not been explained previously.

Update Region Variables: Time to update the variables used to implement ARegions and DRegions.

ARegion Allocation: Time for **create-region** or **delete-region** to allocate or deallocate an ARegion, respectively.

Page-entry Allocation: Time for **create-region** or **delete-region** to allocate

Create and Delete Component Times (μs)				
Components	**create-region**		**delete-region**	
	Sun 3/75	GP1000	Sun 3/75	GP1000
System Call	48	50	45	47
Confirm Arguments	7	7	6	6
Interrupt Control	4	3	4	4
Update Region Variables	.	.	7	8
ARegion Allocation	25	30	20	21
Page-entry Allocation	6(70)	7(61)	19(83)	35(78)
Invalidate ATC Entry	.	.	.	11
DRegion Allocation	13	16	16	17
Page Allocation	26	55	24	45
Circumstances	.	.	5	7
Miscellaneous	1	1	5	7
Component Total	130	169	151	208
Total If DRegion Shared			111	146

Table 5.5: **create-region** and **delete-region** component times.

or deallocate a page entry, respectively. The time to allocate or deallocate a pmeg is included in parentheses.

Invalidate ATC Entry: Time to invalidate an entry of the GP1000 68851 MMU translation cache.

DRegion Allocation: Time for **create-region** or **delete-region** to allocate or deallocate a DRegion, respectively. This does not include the time to allocate or deallocate a physical page of memory.

Page Allocation: Time for **create-region** or **delete-region** to allocate or deallocate a physical page of memory, respectively.

Circumstances: Time to determine the circumstances of the operation (are there other bindings to the DRegion?).

Miscellaneous: Time to cache values in processor registers.

Total If DRegion Shared: Time to delete a binding to a shared DRegion as

opposed to a non-shared DRegion. This time does not include the time to deallocate a DRegion (and physical page).

The measured total elapsed time for **create-region** followed immediately by **delete-region** are 280 μs and 384 μs on the Sun 3/75 and GP1000, respectively (the sum of the component times from Table 5.5 are 281 μs and 377 μs).

5.6.3 SRR Transactions

Because an SRR transaction is the only way[8] processes can synchronise their execution its performance is important for applications where IPC is used frequently.

The component times for an SRR transaction grouped by function are presented in Table 5.6. The Sun 3/75 and local GP1000 measurements provide information about elapsed time and processor utilisation. The remote GP1000 measurements provide information about processor utilisation but not about elapsed time because parallelism is not taken into account. This explains the 55 μs difference between the total of the component times and the measured total elapsed time from Table 5.1.

The following component time categories have not been explained previously.

Context Switches: Time to perform context switches.

Data Transfer: Time to copy messages including mapping process segments into kernel-data area.

Priority Scheduling: Time to block and schedule processes.

Circumstances: Time to determine the circumstances of the operations (Is receiver local or remote? Is receiver blocked waiting for messages? etc.).

House Keeping: Time to record information needed to handle potential process termination and communication failures.

Miscellaneous: Time to cache values in registers and set return values.

[8]Processes could perform a busy wait on a shared variable value; however, this technique is inefficient and an atomic test-and-set instruction is not provided for distributed shared memory.

SRR Transaction Functional Component Times (μs)							
	Category	Sun 3/75		GP1000			
				local		remote	
A	Context Switches	76	26.4%	112	28.6%	228	25.1%
B	System Calls	75	26.0%	104	26.6%	104	11.5%
C	Data Transfer	58	20.1%	83	21.2%	110	12.1%
D	Confirm Arguments	26	9.0%	28	7.2%	115	12.7%
E	Priority Scheduling	15	5.2%	18	4.6%	13	1.4%
F	Interrupt Control	10	3.5%	10	2.6%	11	1.2%
G	Circumstances	10	3.5%	10	2.6%	12	1.3%
H	House Keeping	10	3.5%	10	2.6%	35	3.9%
I	Miscellaneous	6	2.1%	8	2.0%	40	4.4%
J	DRegion Checks	2	0.7%	8	2.0%	19	2.1%
K	Packet Delivery	.		.		208	23.0%
L	Page Alignment	.		.		11	1.2%
	Component Total	288	100%	391	100%	906	99.9%
	Measured Total	288		398		851	

Table 5.6: send-receive-reply component times grouped by function.

DRegion Checks: Time to check whether a DRegion is being passed with the message (a feature to pass a DRegion with a message was included in the implementation but not in the Regions model).

Packet Delivery: Time to deliver a packet to a kernel executing on another node (parallelism is ignored).

Page Alignment: Time to check if messages cross page boundaries.

The remote GP1000 Context Switch time includes the time to return from the internode interrupt.

The remote GP1000 Confirm Arguments time includes MMU operations to convert process addresses to physical addresses.

The components of local and remote SRR transactions are presented in the order they are executed in Tables 5.7 and 5.8. The elapsed time and category from

Table 5.6 is specified for each component. Components from the Miscellaneous, DRegion Checks, and Page Alignment categories are not included to simplify the presentation. The concurrency of the remote SRR transaction is illustrated.

Local communication is faster than remote communication on the GP1000 because:

- data must be copied across the Butterfly switch,
- remote processors must be interrupted,
- the atomic test-and-set operations used to synchronise access to kernel request queues are slow, and
- process addresses must be converted to physical addresses.

Local SRR Transaction Sequential Component Times (μs)				
Component Category and Time			**Sender**	**Receiver**
	3/75	GP1000		
B	28	36	**send** system call	(blocked waiting for
D	12	12	confirm arguments	**send**)
F	2	1	disable interrupts	
H	6	7	save arguments	
G	3	3	if (receiver local)	
G	3	3	if (receiver waiting)	
H	2	2	attach to receiver	
C	29	41	transfer message	
E	1	1	block sender	
E	3	4	if (receiver top pri)	
A	38	56	switch to receiver	
F	2	2		enable interrupts
B	27	37		**reply** system call
D	3	3		confirm arguments
F	2	2		disable interrupts
G	3	3		if (sender local)
D	7	9		find & detach sender
C	29	42		transfer message
E	7	9		schedule sender
F	1	2		enable interrupts
B	20	31		**receive** system call
D	4	4		confirm argument
F	2	1		disable interrupts
G	1	1		if (no messages)
H	2	1		save argument
E	1	1		block receiver
E	3	3		if (sender top pri)
A	38	56		switch to sender
F	1	2	enable interrupts	
B	28	36	**send** system call	

Table 5.7: Local **send-receive-reply** component times in execution order.

Remote GP1000 SRR Transaction Sequential Component Times (μs)					
Component Category and Time		**Sender**	Component Category and Time		**Receiver**
B	36	**send** system call		.	(blocked waiting for
D	39	confirm arguments		.	**send**)
F	1	disable interrupts		.	(another process
H	8	save arguments		.	executes)
G	3	if (receiver remote)		.	
K	2	initialise packet		.	
K	73	deliver packet		.	receive packet
K	39	complete packet	D	20	confirm arguments
E	1	block sender	G	3	if (receiver waiting)
	.	(another process	H	8	attach to receiver
	.	executes)	K	5	wait for packet
	.		C	46	transfer message
	.		E	5	schedule receiver
	.		A	114	switch to receiver
	.		F	2	enable interrupts
	.		B	37	**reply** system call
	.		D	39	confirm arguments
	.		F	2	disable interrupts
	.		G	4	if (sender remote)
	.		D	7	find & detach sender
	.		K	13	reinitialise packet
	.	receive packet	K	71	deliver packet
D	7	confirm arguments	C	64	transfer message
H	17	detach receiver	K	3	complete packet
K	2	wait for packet	F	2	enable interrupts
E	6	schedule sender	B	31	**receive** system call
A	114	switch to sender	D	3	confirm argument
F	2	enable interrupts	F	2	disable interrupts
	.		G	2	if (no messages)
	.		H	2	save argument
	.		E	1	block receiver
B	36	**send** system call		.	

Table 5.8: Remote GP1000 **send-receive-reply** component times in execution order.

5.7 Lessons Learned

Several lessons were learned while achieving an efficient implementation.

Keep it simple.

The performance of an earlier version of the implementation was significantly improved by rewriting the implementation to use simple special purpose procedures and macros rather than more complex general purpose procedures and macros.

Actually measure elapsed time of components.

Attempts to improve performance based on hypothetical explanations can actually degrade performance.

For example, it was assumed that copying 64 byte messages was expensive. Therefore, the kernel design allows messages to be copied directly from one context to another by mapping a context into the kernel-data area. However, measurements showed that mapping an entire context was significantly more expensive than copying 64 bytes. Fortunately, only a single segment is required under most circumstances. Careful coding (using the macro and in-line features described below) resulted in elapsed times less than the alternative method of copying the data twice. However, if messages cross segment boundaries (or if messages were shorter) then copying twice might be more efficient.

In addition, an *optimised* data copying routine was used to copy messages. However, the C compiler generates faster code for copying 64 bytes.

Real time measurements also revealed that significant time was spent performing system calls. The arguments were copied from the process stack to the kernel stack to conform with C argument passing semantics. This time was significantly reduced[9] for the **send**, **receive**, and **reply** primitives[10] by passing a pointer to the process

[9]This technique can reduce the elapsed time of a system call by about 10–30 μs.

[10]The technique was not applied to the other primitives due to time constraints.

stack in a register.

The elapsed time of a procedure call and argument passing on the MC68020 is significant[11]. Procedures are useful for modularising code, reducing coding effort and reducing object code size; however, implementing very simple operations significantly increases the total elapsed time.

Use the features of the compiler to achieve efficiency.

The C language allows requests to bind variables to registers but does not specify how the compiler should do this. Inspection of assembler instructions produced by the C compiler (-S option) revealed that the order of declaration is used for register allocation. Reducing the number of variables and carefully allocating registers to variables resulted in significant performance improvements.

The C compiler provides macro and in-line expansion features to avoid procedure call overhead. Macro calls were used to make the code more readable without incurring procedure call overhead. Assembly instructions are required for interrupt control and Sun 3 MMU operations. The in-line feature was used to invoke those operations without incurring procedure call overhead[12].

5.8 Comparisons With Related Systems

It may not be appropriate to compare the elapsed time performance of primitives from different models because the objectives of the design and implementation of the models are not necessarily the same. In addition, the models are often implemented on different hardware with different characteristics.

However, some related work is mentioned to emphasise the efficiency of this implementation.

[11]Newer processors, like the SPARC, reduce this time with the register window feature.

[12]The asm() feature for specifying assembly code directly in the C source code disabled compiler optimisation.

Bershad, *et. al.* demonstrated efficient remote procedure call between contexts on the same machine (LRPC) [Bershad 89]. They measured elapsed times for *null*-RPC between contexts on a DEC SRC Firefly multiprocessor [Thacker 88]. The elapsed time for a single processor to perform a null-RPC was 157 μs. The elapsed time to perform a null-RPC to another processor waiting for the call was 125 μs. They compared these times with the elapsed times of operations from other well known systems that claim to provide efficient implementations. Their times are significantly lower than times for similar systems[13].

An SRR transaction is similar to a null-RPC but there is an additional system call and two messages are copied. The elapsed time of a local SRR transaction on a Sun 3/75 for the current implementation is 288 μs. This time is reduced to $288 - (20 + 58) = 210\ \mu s$[14] if the additional system call and data transfers are removed. The Firefly's C-Vax processor is faster than the Sun 3/75's 16.7 MHz 68020; therefore, this implementation of **send-receive-reply** provides elapsed times that are very close to the lowest times reported in the literature.

Abrossimov, *et. al.* analysed the elapsed time of deferred copying for large memory objects (DRegions) in the Chorus distributed system [Abrossimov 89]. The elapsed time for passing a one page memory object with deferred copying between processes executing on a Sun 3/60[15] was 400 μs and 2700 μs for Chorus and Mach, respectively. If a write operation is invoked on the memory object then the total time is 2100 μs and 4820 μs for Chorus and Mach, respectively. These times are for passing large memory objects. Abrossimov, *et. al.* mentioned that they will be improving the efficiency of deferred copying for shorter (maximum 8 pages?) mes-

[13]The implementors of the other systems probably did not concentrate on the elapsed time of local operations.

[14]These costs can be reduced in special cases by combining **reply** and **receive** into a single primitive and by passing data in processor registers [Cheriton 84].

[15]A Sun 3/75 is about 1.2 to 1.3 times slower than a Sun 3/60 because of the 60's higher clock frequency (20 MHz) and lower memory cycle time (250 ns).

sages. This work should be directly comparable with the current implementation of the Regions IPC model.

It is interesting to note that the Chorus implementors emphasised efficiency; however, they failed to eliminate a significant cost. Their implementation takes 1400 μs to copy an 8K byte page. However, measurements on a Sun 3/60 using a copy routine that takes advantage of the 68020 instruction cache indicate that an 8K byte page can be copied in 985 μs. Therefore their times could be reduced by 20–30%.

Tzou and Anderson took advantage of MMU hardware to reduce the elapsed time of passing data between separate contexts in the DASH distributed system [Tzou 88]. The elapsed time for passing an 8K byte page on a Sun 3/50[16] was 1194 μs.

Experiments were not performed to demonstrate the usefulness of the DYNAMIC-SHARE mode or the deferred copying implementation technique because of time constraints. Deferred copying has been evaluated for UNIX fork operations[Smith 88].

5.9 Summary

An analysis of the elapsed time performance of implementations of the Regions IPC primitives on a Sun 3/75 workstation and a Butterfly multiprocessor has been presented. The elapsed time measurements demonstrate that the Regions primitives save significant time by avoiding unnecessary copying if more than 1024 bytes of data are passed. If data must be copied then the Regions primitives are less efficient than a primitive that copies the data by a fixed overhead. This overhead can be significantly reduced by improvements to the implementation.

The elapsed time of a local SRR Transaction is comparable with the lowest times

[16]A Sun 3/50 is about 1.3 to 1.4 times slower than a Sun 3/75 because of the 50's lower clock frequency (15 MHz) and slower memory cycle time (320 ns).

reported in the literature. The elapsed times of the primitives that take advantage of the MMU are the best among the times reported for similar systems.

Chapter 6

Conclusions

We conclude that the semantics and efficiency of data-passing IPC operations can be significantly improved by supporting the data-passing modes. The Regions IPC model is thus an improvement over other IPC models used in contemporary distributed systems.

Data-passing semantics are enriched by the duplicate, move and dynamic-share modes. The mode expresses the intended access to the memory containing the passed data. This enables efficient implementation of data-passing by avoiding unnecessary data copying.

Regions supports separate-context and shared-memory semantics. Memory-access interference is controlled with run-time enforced contexts. Memory can be shared between contexts when the benefits of sharing outweigh the risk of interference. Interference is restricted to processes that have access to the shared memory. Programmers not only have control over which processes share which memory but also the duration that each process shares the memory.

Regions demonstrates how the data-passing modes can be supported in a synchronous message-passing IPC model. Other models such as asynchronous message-passing and remote procedure call can also be enriched with these data-passing modes.

Efficient implementation of the data-passing modes makes a data-passing IPC model more competitive with memory-sharing IPC models. This is because the enriched model does not prevent processes from sharing memory. Therefore it is possible to take advantage of shared physical memory.

Regions IPC is an improvement over the IPC of other distributed operating systems because it is simple, it supports enriched data-passing semantics, and it can be implemented efficiently on networks of shared memory multiprocessors. The simple semantics and good performance of the Regions IPC operations provide a reference to compare other IPC operations with. The possibility of implementing IPC operations (with different or more complex semantics) as library routines based on Regions IPC operations can be considered.

Use of the Regions IPC operations requires an understanding of the client/server concept. A server process is an object that provides an interface to client processes. A server can provide operations which support IPC semantics that are not provided directly by the Regions IPC operations. This is demonstrated in Chapter 3 with sketches of three algorithms. The synchronous shared memory server (see Section 3.3.3.2) demonstrates how memory sharing and synchronisation operations can be combined.

The implementation of Regions IPC demonstrates low elapsed times for data-passing operations that avoid unnecessary data copying. It also demonstrates that message-passing between processes executing in separate contexts on the same machine can be significantly more efficient then previously reported[1]. The implementation of the Reactive Kernel on the Ametek 2010 [Seitz 88] implies that Regions can be implemented efficiently on systems that integrate communication channel hardware with memory management hardware.

The performance analysis identifies the real costs of an implementation of the model. Speculation about which factors contribute to the elapsed time can lead to misconceptions. A complete empirical performance analysis is not difficult to do and eliminates potential misconceptions about elapsed time performance.

[1]Regions message-passing performance is comparable with lightweight remote procedure call [Bershad 89].

6.1 Further Work

The Regions IPC model with the data-passing modes can be an efficient basis for supporting a variety of parallel programming approaches. The semantics required for simple specification and efficient execution of parallel algorithms using each programming approach must be determined.

Several other models support a shared context for a set of processes. It would be useful to resolve whether or not Regions would benefit from supporting shared contexts. This would involve evaluating (1) the complexity of specifying parallel algorithms and (2) the performance of those algorithms on shared memory multiprocessors.

Hardware support for efficient data-passing across communication channels can be proposed based on the duplicate, move and dynamic-share data-passing mode semantics. Integration of MMU hardware, communication channel hardware and DMA hardware could significantly reduce the elapsed time of passing data from one context to another across a communication channel. MMU hardware that supports efficient context switch, translation table manipulations, and deferred copying could significantly reduce the elapsed time of passing data from one context to another when shared physical memory is available.

The Regions model design and implementation can be improved is several ways. Capability based protection [Dennis 66] can be introduced to restrict which DRegions a server process can pass to or from a client's context. The performance of the model on uniform memory access multiprocessors and across communication channels could be demonstrated. ARegion allocation could be integrated with data heap allocation. Multiple page DRegion could be implemented.

References

[Abrossimov 89] Abrossimov, V., Rozier, M., Shapiro, M.,
Generic Virtual Memory Management for Operating System Kernels,
Proceedings of the Twelfth Symposium on Operating Systems Principles,
ACM, pp 123–136, December 1989.

[Accetta 86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R.,
Tevanian, A., Young, M.,
Mach: A New Kernel Foundation for UNIX Development,
Proceedings of the USENIX 1986 Summer Technical Conference, pp 93–112,
July 1986.

[Anderson 88] Anderson, D.P., Ferrari, D.,
The Dash Project: An Overview,
Tech. Report UCB/CSD 88/405, Computer Sc. Div., Univ. of California at
Berkeley, pp 1–21, February 1988.

[Atkinson 89] Atkinson, R., Demers, A., Hauser, C., Jacobi, C., Kessler, P.,
Weiser, M.,
Experiences Creating a Portable Cedar,
Xerox PARC Report CSL-89-8, pp 1–12, June 1989

[Bach 86] Bach, M.,
The Design of the UNIX Operating System,
Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Baron 88] Baron, R.V., Black, D., Bolosky, W., Chew, J., Draves, R.P.,
Golub, D.B., Rashid, R.F., Tevanian, A. Jr., Young, M.W.,
MACH Kernel Interface Manual,
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA,
September 1988.

[BBN 88] *Inside the GP1000*,
Part No. A370007G10, BBN Advanced Computers Inc., 10 Fawcett St., Cambridge, MA, 02238, October 1988.

[Bennett 90] Bennett, J., Carter, J.B., Zwaenepoel, W.,
Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,
Rice COMP TR90-108, Rice University, pp 1–9, March 1990.
(also in Proceedings of the Second Annual SIGPLAN Symposium on the Principles and Practice of Parallel Programming(PPOPP), SIGPLAN NOTICES, 25(3), March 1990)

[Bershad 89] Bershad, B.N., Anderson, T.E., Lazowska, E.D., Levy, H.M.,
Lightweight Remote Procedure Call,
Proceedings of the Twelfth Symposium on Operating Systems Principles, ACM, pp 102–113, December 1989.

[Birrell 84] Birrell, A.D., Nelson, B.J.,
Implementing Remote Procedure Calls,
ACM Transactions on Computer Systems, 2(1), pp 39–59, February 1984.

[Bolosky 91] Bolosky, W.J., Scott, M.L., Fitzgerald, R.P., Fowler, R.J., Cox, A.L.,
NUMA Policies and Their Relations to Memory Architecture,
ACM Operating Systems Review, Special Issue, 25, pp 212–221, April 1991.

[Cheriton 79] Cheriton, D.R., Malcolm M.A., Melen, L.S., Sager, G.R.,
Thoth, a Portable Real-Time Operating System,
Communications of the ACM, 22(2), pp. 105–115, February 1979.

[Cheriton 83] Cheriton, D.R., Zwaenepoel, W.,
The Distributed V Kernel and its Performance for Diskless Workstations,
Proceedings of the Ninth Symposium on Operating Systems Principles, ACM, pp 129–140, October 1983.

[Cheriton 84] Cheriton, D.R.,
An Experiment Using Registers for Fast Message-Based Interprocess Communication,
ACM Operating System Review, 18(4), pp 12–19, October 1984.

[Cheriton 86] Cheriton, D.R.,
Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design,
Proceedings of the 6th International Conference on Distributed Computer Systems, IEEE Computer Society, pp 190–197, May 1986.
(also ACM Operating Systems Review, 19(4), pp 26–33, October 1985.)

[Cheriton 88] Cheriton, D.R.,
The V Distributed System,
Communications of the ACM, 31(3), pp 314–333, March 1988.

[Dennis 66] Dennis, J.B., Van Horn, E.C.,
Programming Semantics for Multiprogrammed Computations,
Communications of the ACM, 9(3), pp 143–155, March 1966.

[Dijkstra 68] Dijkstra, E.W.,
The Structure of the 'THE' Multiprogramming System,
Communications of the ACM, 11(5), pp 341–346, May 1968.

[Fitzgerald 86] Fitzgerald, R., Rashid, R.F.,
The Integration of Virtual Memory Management and Interprocess Communication in Accent,
ACM Transactions on Computer Systems, 4(2), pp 147–177, May 1986.

[Hoare 74] Hoare, C.A.R.,
Monitors: An Operating System Structuring Concept,
Communications of the ACM 17(10), pp 549–557, October 1974.

[Kernighan 78] Kernighan, B.W., Ritchie, D.M.,
The C Programming Language,
Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Li 86] Li, K.,
Shared Virtual Memory on Loosely Coupled Multiprocessors,
PhD Thesis, Yale, 1986.
(also Yale technical report Yale/DCS/RR-492, September 1986).

[Motorola 89a] *MC68020: 32-Bit Microprocessor User's Manual Third Edition*,
Motorola Inc., Prentice Hall, Englewood Cliffs, N.J., 1989.

[Motorola 89b] *MC68851: Paged Memory Management Unit User's Manual Second Edition*,
Motorola Inc., Prentice Hall, Englewood Cliffs, N.J., 1989.

[Nitzberg 91] Nitzberg, B., Lo, V.,
Distributed Shared Memory: A Survey of Issues and Algorithms,
Computer, 24(8), pp 52–60, August 1991.

[Rashid 88] Rashid, R., Tevanian, A. Jr., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W.J., Chew, J.,
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures,
IEEE Transactions on Computers, 37(8), pp 896–908, August 1988.

[Reed 79] Reed, D.P.,
Synchronization With Eventcounts and Sequencers,
Communications of the ACM, 22(2), pp 115–123, February 1979.

[Saltzer 84] Saltzer, J.H., Reed, D.P., Clark, D.D.,
End-to-end Arguments in System Design,
ACM Transactions on Computer Systems, 2(4), pp 277–288, November 1984

[Schroeder 89] Schroeder, M., Burrows, M.,
Performance of Firefly RPC,
Proceedings of the Twelfth Symposium on Operating Systems Principles, ACM, pp 83–90, December 1989.
(also ACM Operating Systems Review, Special Issue, 23(5), pp 83–90, December 1989.)

[Seitz 88] Seitz, C.L., Athas, W.C., Flaig, C.M., Martin, A.J., Seizovic, J., Steele, C.S., Su, W.-K.,
The Architecture and Programming of the Ametek Series 2010 Multicomputer,
Proc. Third Conf. on Hypercube Concurrent Computers and Applications, 1, pp 33–36, 1988.

[Seitz 90] Seitz, C.L., *Multicomputers*, pg 131–200 in Hoare, C.A.R.,
Developments in Concurrency and Communication,
Addison-Wesley, Reading, Mass, 1990.

[Smith 88] Smith, J.M., Maguire, G.Q. Jr.,
Effects of Copy-On-Write Memory Management on the Response Time of UNIX Fork Operations,
Computing Systems (The Journal of the USENIX Association), 1(3), pp 255–278, Summer 1988.

[Spector 82] Spector, A.Z.,
Performing Remote Operations Efficiently on a Local Computer Network,
Communications of the ACM 25(4), pp 246–260, April 1982.

[Stroustrup 86] Stroustrup, B.,
The C++ Programming Language,
Reading, Mass., Addison-Wesley, 1986.

[SUN3 86] *Sun 3 Architecture: A Sun Technical Report*,
Sun Microsystems Inc, 2550 Garcia Ave, Mountain View, CA, 94043, Revised August 1986.

[Thacker 88] Thacker, C.P., Stewart, L.C., Satterthwaite, Jr., E.H.,
Firefly: A Multiprocessor Workstation,
IEEE Transactions on Computers, 37(8), pp 909-920, August 1988.

[Tzou 88] Tzou, S.-Y., Anderson, D.P.,
A Performance Evaluation of the DASH Message-Passing System,
Tech. Report UCB/CSD 88/452, Computer Sc. Div., Univ. of California at Berkeley, pp 1–16, October 1988.

[Vasudevan 87] Vasudevan, R.,
Network Transparency in Multiprocess Structuring,
PhD Thesis, Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1987.

[Vasudevan 88] Vasudevan, R.,
A High-Performance Distributed Software Base,
Multi'88, Conference on Distributed Simulation, San Diego, February 88.

[Young 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R.,
The Duality of Memory and Communications in the Implementation of a Multiprocessor Operating System,
ACM Operating Systems Review, 21(5), pp 63–76, 1987.
(also Proceedings of the Eleventh Symposium on Operating System Principles, ACM, November 1987.)