Technical Report 1245

Pi: A Parallel Architecture Interface for Multi-Model Execution

Donald Scott Wills

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

Pi: A Parallel Architecture Interface for Multi-Model Execution

by

Donald Scott Wills

Massachusetts Institute of Technology

July 1990

© **Donald Scott Wills 1990**

Revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on 14 May 1990 in partial fulfillment of the requirements for the degree of Doctor of Science.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-88-K-0738, N00014-87-K-0825, and N00014-85-K-0124, and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

Abstract

Existing parallel architectures are constructed monolithically, with no well defined boundaries separating model and machine issues. This makes it difficult to evaluate the effect of a single component of an architecture, or compare it with the corresponding components of other architectures. Machine hardware is specialized to support a single programming model, even though similar but more general mechanisms could support a variety of models. Often details of an implementation become visible in the programming environment, restricting future implementation improvements because of compatibility. The lack of an interface between model and machine issues also complicates the translation of a machine improvement into a performance improvement for model applications.

This thesis defines Pi, a parallel architecture interface that separates model and machine issues, allowing them to be addressed independently. This provides greater flexibility for both the model and machine builder. Pi addresses a set of common parallel model requirements including low latency communication, fast task switching, low cost synchronization, efficient storage management, the ability to exploit locality, and efficient support for sequential code. Since Pi provides generic parallel operations, it can efficiently support many parallel programming models including hybrids of existing models. Pi also forms a basis of comparison for architectural components.

Pi is evaluated in two ways. First, several mechanisms required by existing parallel models are constructed on the interface. These examples are executed and evaluated using a Pi simulation environment. Then a machine substrate that supports the interface is specified. It is designed to efficiently support the generic parallel operations provided in Pi. The feasibility of gate array implementation is considered. The effectiveness of this machine substrate at supporting Pi is evaluated. The role of a machine compiler below the interface is also discussed.

This thesis demonstrates a parallel architecture interface. Pi efficiently supports several model mechanisms. A machine substrate which effectively implements Pi is presented.

Thesis Supervisor: William J. Dally

Title: Associate Professor, EECS

Acknowledgments

Many people contributed, directly and indirectly, to this research. I'd especially like to thank the following people:

Linda Wills, my wife and best friend, for comments and contribution on this thesis, and for making my life wonderful for the past five years. Her encouragement makes everything I do possible, including this thesis.

Bill Dally, my thesis advisor, for considerable help and guidance with my thesis and my doctoral program. Bill has been a continuous source of energy and enthusiasm over the past four years. He has worked hard to make the Concurrent VLSI Architectures Group a great environment for parallel architecture research.

Tom Knight, for participating on my committee, and contributing to the thesis. Tom has been my mentor since I arrived at MIT. Bill and Tom have both taught me a great deal about research and teaching. In the future, I hope I can provide my students with the same knowledge and inspiration that they have provided to me.

Anant Agarwal, for participating on my committee, and contributing to this thesis.

I have learned more from my fellow graduate students than from any other source. I've also had the privilege of having these people as friends over the past years.

Andrew Chien, my officemate and thesis buddy, for sharing his ideas and insight on many subjects. Andrew has greatly contributed to the ideas in this thesis.

Peter Nuth, my officemate, with whom I have shared many far reaching conversations on many topics. Peter's sense of humor has made the CVA group a lot of fun.

Mike Noakes, for many valuable comments and contributions on the thesis. Mike has also made this thesis possible by keeping the group machinery going over the past year.

Stuart Fiske, my long time colleague, for contributions to PiMac, and for contributing to the positive group spirit.

John Keen, for teaching me about I/O automata and other interesting topics, and for being the CVA straight man.

Rich Lethin, for contributing to the group CAD environment, and sharing lots of interesting stories about life on the "outside" (at Multiflow).

Julia Bernard, my friend, for lots of moral support and encouragement.

Waldemar Horwat, for contributing the the CVA software environment.

To the CVA UROPers, Stephen Bertram, Linda Chao, Ye Gu, Lucien Van Elsen, Michael G. Larivee, Ellen Spertus, Brian Totty, and Deborah Wallach, for adding to the diversity of the group.

John Geist, Bibb Cain, and the rest of the ATD gang at Harris, for technical and financial support during my doctoral program.

My sister, Kitty, for cheering me up and being my friend.

Mom and Dad, for constant encouragement and support for everything I do.

Contents

1 Introduction 9

2 The Pi Interface 15

2.1 Storage 15

2.1.1 Read-Write Segments 16

2.1.2 Associative Segments 17

2.1.3 Storage Reclamation 19

2.1.4 Nodals 19

2.2 Synchronization 20

2.2.1 Data Synchronization 20

2.2.2 Barrier Synchronization 20

2.2.3 Producer-Consumer Synchronization 21

2.2.4 Attributes 21

2.3 Communication 23

2.3.1 The Network 24

2.4 Task Management 25

2.4.1 Task Storage 25

2.4.2 Task Dispatch 25

2.4.3 Task Atomicity 26

2.4.4 Task Prioritization 26

2.4.5 Calling Tasks 27

2.4.6 Variable Argument Passing 27

2.5 Locality 28

2.6 Sequential Operations 29

2.7 Summary 29

3 Building Models on Pi **31**

3.1 PiSim: A Pi Simulator 32

3.1.1 Handlers 32

3.1.2 Nodals and Constants 33

3.1.3 Special Operations 33

3.1.4 PiSim Implementation 35

3.2 Shared Memory with Caches 35

3.2.1 A Shared Address Space 36

3.2.2 The Components 37

3.2.3 The Protocol 38

3.3 Set Synchronization 43

3.3.1 The Components 44

3.4 Object Name Translation 46

3.4.1 The Components 48

3.5 Non-Resident Handlers 49

3.5.1 The Components 49

3.6 N-Body Simulation 50

3.6.1 The Problem 52

3.6.2 The Components 53

3.7 Relaxation 55

3.7.1 The Components 56

3.8 Summary 58

4 An Evaluation of Pi **60**

4.1 Evaluation Metrics 60

4.2 Shared Memory with Caches 61

4.2.1 The Experiment 63

4.3 Set Synchronization 64

4.3.1 The Experiment 65

4.4 Object Name Translation 66

4.4.1 The Experiment 68

4.5 Non-Resident Handlers 69

4.5.1 The Experiment 70

4.6 N-Body Simulation 70
4.6.1 The Experiment 71
4.7 Relaxation 72
4.7.1 The Experiment 72
4.8 Summary 73
4.8.1 Instructions 73
4.8.2 Operations 75
4.8.3 Segments 76
4.8.4 Tasks 78

5 A Pi Substrate **81**
5.1 Design Goals 82
5.2 PiMac: A Pi Machine Architecture 82
5.2.1 Microarchitecture 83
5.2.2 Segments 86
5.2.3 Word Format 87
5.2.4 Register Architecture 89
5.2.5 Instruction Set Architecture 93
5.2.6 Network 99
5.2.7 Handlers 99
5.2.8 Tasks and Messages 101
5.2.9 Microsequences 102
5.2.10 Traps 106
5.3 A Gate Array Implementation 106
5.3.1 The Message Driven Processor 107
5.3.2 PiMac Module Designs 108
5.3.3 PiMac Timing 111
5.3.4 A PiMac Gate Array 112
5.4 Summary 113

6 Pi on PiMac **114**

6.1 Storage 114

6.1.1 Read-Write Storage 114

6.1.2 Associative Segments 116

6.1.3 Storage Reclamation 117

6.1.4 Nodals 117

6.2 Synchronization 117

6.2.1 Data Synchronization 118

6.2.2 Barrier Synchronization 118

6.2.3 Producer-Consumer Synchronization 118

6.2.4 Attributes 118

6.3 Communication 119

6.3.1 The Network 119

6.4 Task Management 119

6.4.1 Task Storage 119

6.4.2 Task Dispatch 120

6.4.3 Task Atomicity 120

6.4.4 Task Prioritization 120

6.4.5 Calling Tasks 120

6.4.6 Variable Argument Passing 121

6.5 Locality 121

6.6 Sequential Operations 121

6.7 Summary 121

7 Conclusion **124**

A Example Source Code **133**

A.1 Common Handlers 133

A.2 Shared Memory With Caches 134

A.3 Shared Memory With Address Braiding 143

A.4 Shared Memory Tests 145

A.5 Set Synchronization 149

A.6 Object Name Translation 152

A.7 Object Name Translation Tests 157
A.8 Non-Resident Handlers 160
A.9 N-Body Simulation 165
A.10 Relaxation 169

B Example Execution Logs **173**
B.1 Shared Memory With Caches 173
B.2 Shared Memory With Address Braiding 176
B.3 Set Synchronization 179
B.4 Object Name Translation 182
B.5 Non-Resident Handlers 185
B.6 N-Body Simulation (10 Bodies) 188
B.7 N-Body Simulation (100 Bodies) 190
B.8 Relaxation 193

Chapter 1

Introduction

The von Neumann model of sequential computers was a milestone in computer evolution because it defined an interface between programming models and machine implementations. Both models and machines have advanced considerably since then, but the interface, the way each side *sees* the other, has remained relatively unchanged. This separation of concerns is responsible in part for a key capability of today's computers: the ability to support a variety of programming languages and applications using the latest machine design techniques and technology.

Wanted: A Parallel Interface

A similar interface can provide the same beneficial effects for parallel computers. This type of interface is sorely needed. Currently, parallel architectures are constructed monolithically, as shown in Figure 1.1. An architecture is a composition of model and machine features with no well-defined boundaries separating different aspects of the design. This makes it difficult to evaluate the effect of a single component, or compare it with the corresponding components of other architectures.

Sometimes details of an implementation become visible in the programming environment, restricting future implementation improvements because of compatibility. The lack of an interface between model and machine issues also complicates the translation of a machine improvement (e.g., an improved communication network) into a performance improvement for model applications.

Despite the benefit of a parallel interface, it has not been possible to specify one. Adequate experience with model and machine issues in parallel architectures is required. It is first necessary to identify what parallel architectural features and mechanisms are needed, and obtainable.

Parallel Architecture Requirements

Recently, a consensus on requirements has begun to surface in parallel architecture research. The following capabilities have been identified:

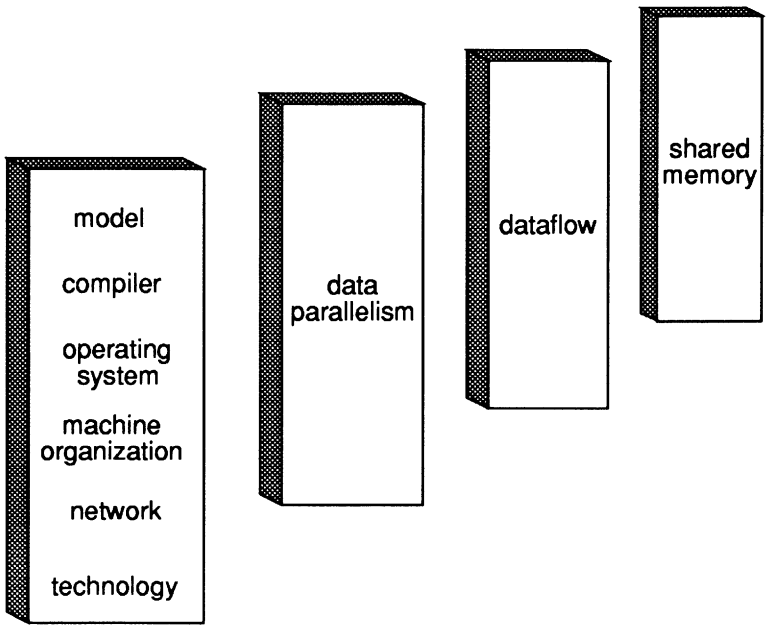


Figure 1.1: The Problem: Architectural Monoliths

- low latency communication
- fast task switching
- low cost synchronization
- efficient storage management
- the ability to exploit all types of locality
- the ability to efficiently support sequential code segments in a parallel environment

These requirements are common to most existing parallel models. With them, it is now possible to specify an interface.

Introducing Pi

This thesis introduces Pi, a parallel architecture interface. Pi is based on the generic parallel model requirements described above, so a machine substrate that supports Pi can implement multiple parallel models. Pi provides an abstraction that separates model and machine issues [1], allowing them to be addressed independently. It forms a basis of comparison for architectural features and it provides greater flexibility for both the model and machine builder.

[1] Model issues are related to the problem solving style. Machine issues are related to the physical realization of a problem solving system

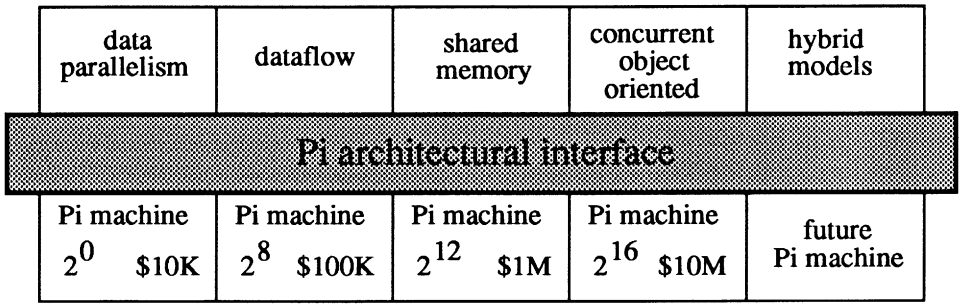


Figure 1.2: Pi Overview

Figure 1.2 presents an overview of Pi. Each box above the interface represents the mechanisms required for a model, constructed with Pi operations. For example, the data parallelism model includes a mechanism for supporting set synchronization (a synchronization tree, for example). In addition to existing parallel models, Pi also supports hybrids of mechanisms from different models. Below the interface, each box represents a different Pi machine substrate. All of these machines are designed specifically to support Pi (i.e., they are not specialized for a particular model). The estimated number of nodes and machine cost is listed for each substrate. As new machine building techniques and technologies are incorporated into future substrates, the performance improvements can be realized by all models above the interface.

Pi does not eliminate the interrelationship of models and machines. The costs of Pi operations on a particular machine affect how model mechanisms are best constructed. They also have an influence on which model provides the most effective solution to a problem. The interface attempts to provide the functionality of the underlying machine in a way that does not unnecessarily bias a solution towards a particular model. An advance in machine design may lower the cost of an interface operation, requiring changes in a model's implementation or the solution to a problem. But since only cost, not functionality, has changed, a major reimplementation is not expected.

A system incorporating an architectural interface is shown in Figure 1.3. A problem solving style is supported by a machine independent compiler. This compiler produces a "program" of basic parallel operations from Pi. The interface is supported below by a combination of a machine dependent compiler and machine hardware.

Pi serves as an image of what lies on the other side. It must accurately represent the requirements and constraints, while abstracting away unnecessary details. It must be expressive enough to allow efficient interaction across the interface, but not overly constrain the design freedom of either side. These interface issues are embodied in the following design criteria, used in the development of Pi:

- *Interface Abstractness:* Does the abstractness of the interface provide a reasonable balance of flexibility between the model and machine implementation?
- *Presentation Accuracy:* Are the functionality and requirements of each side of the interface accurately presented?

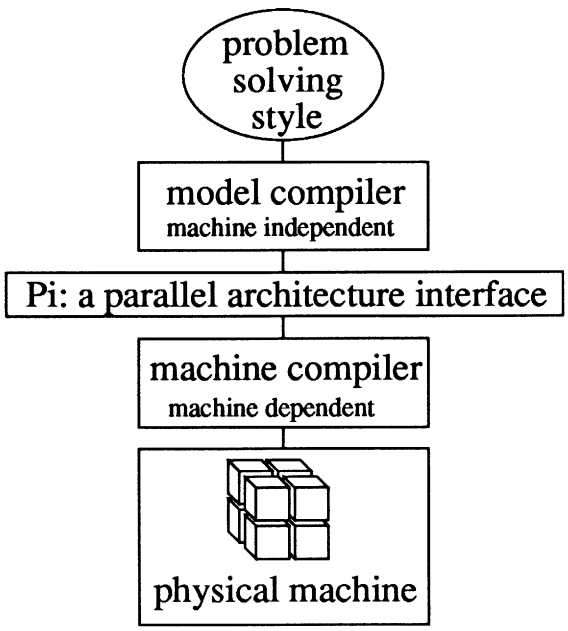


Figure 1.3: Pi: An Architectural Interface Between Model and Machine

- *Functionality Match:* Does the functionality provided in the interface match (a) the requirements of model mechanisms, and (b) the capabilities of machine implementations?
- *Implementation Bias:* Do interface operations minimize bias towards particular implementations of models and machines?

These criteria, together with the parallel model requirements, formed the basis of the design of Pi.

Thesis Contributions

- **Pi Interface:** A parallel architecture interface based on generic parallel model requirements is defined.
- **Model Mechanisms in Pi:** Several model mechanisms (a shared memory protocol, set synchronization, object name translation, and non-resident handlers) and two parallel applications (n-body and relaxation) are implemented and evaluated on a parallel architectural interface.
- **A Pi Substrate:** A machine architecture that efficiently supports generic parallel operations is defined. The feasibility of one implementation is demonstrated.

Related Work

Other research has studied parallel architecture interfaces at different levels of abstractness. Related work falls into three classes: *language interfaces*, *machine interfaces*, and *theoretical models*.

The first class of related work assumes a "language-centered" view of the problem (e.g., Paralation LISP [49], dataflow languages [1], AFL-1 [10], and Nial [41]). They provide a convenient programming environment, often supporting multiple execution model mechanisms (e.g., data parallelism and MIMD fork/joins). However, this approach often poses a large "semantic gap" between the interface and the machine implementation. As a result, an efficient implementation of the language interface is often difficult. In addition, they often provide no separation of model and machine issues.

The second class of related work assumes a "machine-centered" view of the problem (e.g., HEP [34], MPP [7], Cosmic Cube [51], the NON-VON [52], the Transputer [31], the Connection Machine [56], WARP [4], the NCUBE 2 [43], RP3 [46], J-Machine [18], the BBN Butterfly [8], and the BBN Monarch [48]). In this class, novel machine organizations and new technological advances are combined in a machine design. The machines often sport impressive raw performance. Unfortunately, programming model considerations are often neglected, resulting in a large semantic gap between the machine and the programming environment. This often restricts the use of these high performance machines to only specialized classes of applications and programming models.

The last class of related work includes theoretical models of parallel computation (e.g., Communicating Sequential Processors [26], PRAM [50], Actors [3], Type Architectures [55], Bulk-Synchronous Parallel Computers [58]). These models are valuable for studying the theoretical capabilities and limits of parallel computation. However, they usually pursue mathematical clarity, rather than being sullied with the practical details of solving real-world problems with real-world machines. They typically provide poor architectural interfaces. In the worst case, they produce two semantic gaps (programming language $\leftrightarrow$ model, *and* model $\leftrightarrow$ implemented machine). However, because of their clarity, theoretical models often provide inspiration for more practical interfaces. The relationship of these models to the desired interface is analogous to the relationship of the Turing machine model to von Neumann's interface.

Thesis Outline

This thesis begins with the definition of Pi, presented in Chapter 2. A discussion of interface design choices is presented.

An architectural interface cannot be examined in isolation. This thesis also studies how model mechanisms are built on Pi. In Chapter 3, several mechanisms from parallel models are constructed including: shared memory (with caches), set synchronization, object name translation, non-resident handler support, n-body simulation, and relaxation simulation.

In Chapter 4, these examples are evaluated using the Pi source code, and execution statistics from a Pi simulation environment.

Since Pi is to be supported by a specially designed hardware substrate, an example of a Pi machine architecture is presented in Chapter 5. This substrate design incorporates the generic parallel model requirements described earlier, as well as the results from the evaluation of the examples. This chapter also includes a discussion of gate array implementation.

Chapter 6 examines how the requirements of Pi are supported on this machine architecture. It also provides some cost estimates for Pi operations supported on the described hardware architecture.

Finally, Chapter 7 summarizes the conclusions of the thesis and describes future directions for the work.

Chapter 2

The Pi Interface

This chapter presents Pi, a parallel architecture interface. Pi is defined in the form of an abstract machine, although a specific machine implementation is not implied. Structurally, the Pi abstract machine has a familiar form: a collection of nodes connected by a communication network. However its behavior embodies a broad class of machine mechanisms.

The Pi abstract machine is fine-grain (i.e., thousands of nodes, small task sizes). It can also support the requirements of coarse-grain models and machines (i.e., tens of nodes, large task sizes). Mixing the grain size of models and machines produces varying results. It is difficult to efficiently execute small tasks on a machine designed for large tasks, since the task overhead is unacceptable high. However, large tasks are efficiently executed on a small task machine with its low task overhead. Unfortunately, larger task models have less potential for parallelism.

The Pi abstract machine is message driven, although other choices are possible (e.g., communication by shared memory, task invocation by remote procedure calls). Message passing was selected because it provides a convenient and efficient casting of most machine mechanisms.

This chapter examines several aspects of the abstract machine, including storage, synchronization, communication, task management, locality, and sequential operation support.

2.1 Storage

In Pi, storage is represented as logical collections of data or *segments*. A segment contains a number of related pieces of information. It is uniquely named on the node where the storage resides. A segment name is used to reference the stored data. Each location in a segment, referenced by an offset, contains a *value* which can be a number, symbol or boolean. Pi does not specify how these data types are supported. However, a few special values (e.g., UNBOUND) are referred to in this definition. Segments are distributed across all nodes of the abstract machine.

Each segment is named by a *segment ID*. Pi does not specify how segment IDs are supported in an implementation. An machine implementation can use the physical address of the segment base for this purpose. If this technique is employed, storage compaction is problematic if segment IDs are copied outside the node. An alternate implementation separates segment naming and storage allocation by providing a translation between names and absolute pointers. This technique does not require the restriction mentioned above for storage compaction. However a translation mechanism (albeit a simple one) is necessary.

Although a storage hierarchy is not part of the abstract machine, it is not precluded from implementations of Pi. Normally, storage access costs reflect the benefit of reference locality.

There are many alternatives to representing storage as segments. On a abstract node, memory could be addressed as a single linear array. However, this erodes the model machine abstraction and restricts the implementation of memory in a Pi substrate. Memory could be separated from the processing component of an abstract node (i.e., separate memory and processing nodes). This approach restricts the exploitation of spatial locality above the interface. It also couples communication and storage access operations, preventing explicit control of communication above the interface.

Two type of segments are supported in Pi: *read-write* and *associative*

2.1.1 Read-Write Segments

Read-Write segments are finite blocks of linearly addressed storage locations. They are accessed through familiar read and write operations:

> **(read segment offset)**
>
> This operation returns the value in the specified segment at the specified offset.

> **(write segment offset new-value)**
>
> This operation writes the new value into the specified segment and offset.

Read-Write segments are created using the create-read-write-segment operation:

> **(create-read-write-segment size)**
>
> This operation allocates a read-write segment of the requested size (in words), and returns a segment name.

Segment names are only created through segment allocation operations. Once generated, they are immutable. Address arithmetic is only possible on segment offsets,

and offsets must fall within the segment boundaries. Once a read-write segment is created, its size cannot be changed.

Read-Write segments support the basic storage naming mechanism in Pi: the tuple <**Node, Segment, Offset**> or **NSO**. NSOs are referred to throughout this document as handles to storage locations. More general naming mechanisms (which support segment migration for example) can be supported by Pi, but are not explicitly part of the interface.

2.1.2 Associative Segments

Associative segments are a second type of storage supported in Pi. They are provided in addition to read-write segments because they constitute a different access model of storage. Associative segments could be supported on top of the interface using read-write segments. But then the purpose of the storage would be obscured or lost, preventing the machine builder from providing the best implementation of associative segments on the available hardware. Pi attempts to capture the *intent* of the model builder without specifying all the details.

Associative segments provide access to segment data via the following operations:

(insert segment key new-value)

This operation inserts the value into the segment with the specified key. If the new entry cannot be inserted into the segment (because of size bounds), the key of the existing entry (which caused the collision) is returned. Otherwise the key of the inserted entry is returned. If the new value is specified as UNBOUND, the appropriate key is returned, but the segment is unmodified.

(match segment key)

This operation returns the value associated with the key in the segment. The symbol UNBOUND is returned if the key is not found in the segment.

(remove segment key)

This operation removes the key and associated value from the segment. The old value is returned if the entry existed in the segment. Otherwise UNBOUND is returned.

(clear segment)

This operation clears all entries from an associative segment.

Associative segments have two important parameters that are specified when the segment is created: Size and Safety. The size parameter indicates whether the segment size is fixed or not. It is either BOUNDED (fixed at some value), or UNBOUNDED[1] (able to grow to some relatively large size, with respect to the system storage). The safety parameter indicates whether segment entries can be lost, usually as a result of another insertion. An associative segment is either SAFE (entries cannot be lost), or UNSAFE (entries can be discarded without notice).

These parameters are orthogonal. Here are the behaviors of the four combinations:

BOUNDED-SAFE

This segment type guarantees that the segment size is not increased and that existing entries are not overwritten. This is accomplished only by refusing new insertions. When an insertion collision occurs, the insertion operation returns the key of the existing entry. This key can be used to safely remove the entry before retrying the insertion. This segment type requires the implementation to use a repeatable replacement policy.

UNBOUNDED-SAFE

This segment type guarantees safe acceptance of all insertions, by increasing the size of the segment. An insert operation is normally not rejected, nor is an existing entry overwritten.

BOUNDED-UNSAFE

This segment type accepts all insertions without increasing the segment size. It may unsafely overwrite an existing entry. This behavior is similar to that of hardware caches.

UNBOUNDED-UNSAFE

This segment type behavior is underspecified, since an insertion collision can be resolved either by increasing the segment size, or by overwriting an existing entry. This affords the machine dependent compiler and runtime system flexibility to better optimize behavior.

Pi places no restrictions on how entries are mapped into or replaced in an associative segment beyond those described above. As a consequence, a segment of size N does not consider size and safety issues at the (N + 1)th insertion. A collision can occur as early as the second insertion.

Also, UNBOUNDED segments only guarantee safe acceptance of insertions within the bounds of specific machine limitations. If this limit is exceeded, an exception results. This error is similar in nature to a divide by zero exception; it is not normally expected.

An associative segment size is specified even for UNBOUNDED types, to give the machine specific compiler an estimate of the expected size.

[1] An UNBOUND size parameter is different than an UNBOUND value. The latter means the value is unknown or uninitialized.

Associative segments are created using the Create-Associative-Segment operation.

(create-associative-segment size type)

This operation creates an associative segment. The size parameter specifies the desired initial size (number of key-value bindings). The type parameter specifies the boundedness and safety of the segment. The segment type is one of the following: BOUNDED-SAFE, UNBOUNDED-SAFE, BOUNDED-UNSAFE, and UNBOUNDED-UNSAFE.

2.1.3 Storage Reclamation

Explicit storage deallocation is supported in Pi via the Destroy-Segment operation:

(destroy-segment segment)

This operation frees the storage in the specified segment. Any segment type can be freed using this operation.

Pi neither requires nor precludes automatic storage reclamation (garbage collection). This decision is left to the machine compiler. However certain applications expressed in some programming models perform poorly, or not at all without automatic storage reclamation.

It is often necessary to determine the size of a read-write or associative segment. The Segment-Size operation supports this need.

(segment-size segment)

This operation returns the size of a read-write or associative segment.

2.1.4 Nodals

A segment can be accessed only if the segment name is known. However there are many applications where it is expedient to have globally "named" segments and values. This is supported via node variables or *nodals*. They are analogous to global variables in sequential computing, except a copy of each variable exists on each node. Nodals are globally named but locally consistent (i.e., a nodal's value is not consistent *across* nodes). When a nodal is defined, an independent copy of that named storage location is created on every node. They are maintained in a special nodal segment on each node.

(nodals)

This operation returns the name of the segment containing the nodals for the current node. When a nodal is defined, it is assigned a location in the nodal segment. Nodals can then be accessed via read-write segment accessor operations, using this operation to access the node's nodal segment.

2.2 Synchronization

Synchronization in parallel system is composed of two components: a piece of named synchronization state, and an inter-node communication mechanism to allow access of that state. In Pi, communication is provided by the message passing operation which is presented later in Section 2.3.

The named state, which indicates the status of synchronization events (both data and control), is maintained in specially annotated storage locations (locations are annotated using the attribute operation described in Section 2.2.4). In this way, synchronization event naming is supported via the existing storage naming facility (NSO).

As an alternative to this approach, the naming, storage, and communication requirements of synchronization could be combined in a single synchronization operation. However, synchronization communication patterns vary. Sometimes hundreds of tasks are involved, sometimes only two. Synchronizing tasks may be located on different nodes, or they may be on one node. It is difficult capture these different communication patterns in a general operation. By separating the communication component, it can be handled using the same techniques as general communications. By maintaining synchronization state in read-write storage locations, synchronization naming and access is simplified.

Several forms of synchronization are explicitly supported in Pi. Three forms are considered in detail: data, barrier, and producer-consumer synchronization.

2.2.1 Data Synchronization

Data synchronization is supported by storage locations annotated as *d-syncs*. This is similar to other data synchronization mechanisms such as futures in Multilisp [33], context futures in CST [19], I-Structures in Id [6], and full/empty bits in HEP [53]. When a data synchronization is created, it is marked as not yet containing valid data. Any attempt to read it results in the suspension of the active task. A d-sync maintains a ordered set of suspended tasks, waiting for the location to be written. When a d-sync is written, the read lock is cleared, permitting read accesses of the location. Suspended tasks waiting to read the location are requeued for execution in the order they suspended. All locations in newly created read-write segments are annotated as d-syncs by default.

2.2.2 Barrier Synchronization

A barrier synchronization or *b-sync* is a control synchronization mechanism, similar to counting semaphores described by Dijkstra [21]. A b-sync maintains a count which is adjusted (incremented or decremented) and tested. When a b-sync is tested, the active task suspends if the barrier is positive. A b-sync maintains a ordered set of

tasks which have suspended on the barrier. When a barrier synchronization count is reduced to a non-positive value, suspended tasks waiting on the barrier are requeued. If the barrier count is increased to a positive number, tasks will once again suspend if they test the count.

Two operations are provided to support barrier synchronization: test-count and adjust-count.

(test-count segment offset)

This operation tests the count of a barrier synchronization. If it is positive, the active task is suspended and added to an ordered set of tasks waiting on the barrier.

(adjust-count segment offset count-change)

This operation adjusts the barrier count by count-change. **count-change** can be positive or negative, allowing the required count to be increased or decreased. If the count becomes zero or negative, suspended tasks waiting on the barrier are requeued in the order they suspended.

2.2.3 Producer-Consumer Synchronization

A producer-consumer synchronization or *s-sync* is a combined data and control synchronization. An s-sync guarantees that each each value written to the synchronization location is read exactly once. It is useful for the support of several model mechanisms including lock step execution of a set of tasks, often required in data parallelism.

An s-sync initially behaves like a d-sync, prohibiting reads until the location has been written. After it has been written, write attempts are suspended until the s-sync has been read. Then the cycle repeats. This behavior guarantees that a value is not overwritten before it is read, or read twice.

2.2.4 Attributes

All synchronizations are supported in read-write storage locations. Storage locations are designated as synchronizations by *attributes*. Each word of a read-write segment is annotated with an attribute which governs the permissible access patterns for that location. Read-Write locations are initially attributed as d-syncs. This attribute can be changed using the attribute operation to one of the seven attribute types summarized in Figure 2.1.

(attribute segment offset attribute-type)

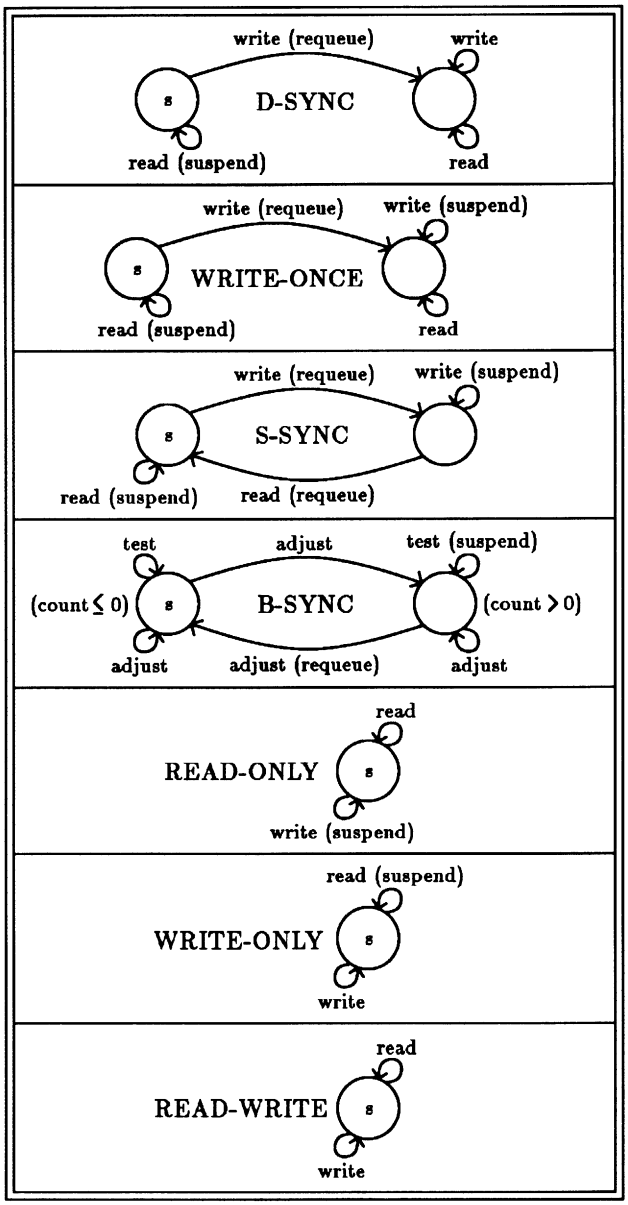


Figure 2.1: Synchronization Attribute Summary

This operation attributes the specified segment location. The following attribute types are supported: D-SYNC, WRITE-ONCE, S-SYNC, B-SYNC, READ-ONLY, WRITE-ONLY, and READ-WRITE. When a location is attributed, all suspended tasks on the location are requeued in the order they suspended.

It is often desirable to check whether a local memory access operation is going to result in a suspension without actually executing it. The probe operation is provided for this purpose.

(probe segment offset access-operation)

This operation is a predicate, indicating whether the specified access operation will result in a suspension of the active task. Access operations include: read, write, and test-count.

Global time is valuable for supporting certain model mechanisms including temporal synchronization. Global time is provided by the time operation.

(time)

This operation returns the global time of the system.

2.3 Communication

All communication requirements of Pi are satisfied by a single mechanism: message passing. A messages passing cycle is composed of the following steps:

1. A message is injected into a network by a task on the source node. The message destination, length, priority, and type are sent (in that order), followed by a variable number of arguments.
2. The communication network routes the messages to the correct destination. The destination field of the message is removed during delivery.
3. The destination node receives the first field of the message, the message length. It then allocates storage (a read-write segment)in local node memory for the messages.
4. The destination then accepts the remainder of the message from from the network and stores it into the allocated segment.
5. The message is then scheduled as a task using the task ID (the ID of the segment containing the message) in a task/message queue. The specific queue is specified by the priority field. Task scheduling and invocation is discussed further in Section 2.4.

There are many alternatives to this approach for supporting communication. For example, communication and task invocation can be separated into two operations. The local task invocation operation is similar to Pi's **call** operation, described in Section 2.4.5. For the communication-only operation, naming of the communicated data becomes problematic. A complex scheme is required to allow a task on the destination node to connect with the appropriate received data. All of the approaches (e.g., scanning a message arrival queue, supporting a shared address space, etc.) require a substantial mechanism overhead. By combining communication and remote task invocation, the invoked task knows the name of the communicated data since it is included in the task segment. If communication-only messages are required, a simple task which stores the data in the appropriate location can be employed.

Another alternative is to support more complicated communication operations like remote reads and writes. However, this complexity is not always required, resulting in wasted communication resources. Also, this mechanism can be easily synthesized using message passing. Most often, a simple but general operation is preferable to a more complicated one.

The simplest operation does not always provide acceptable performance. For example, another communication alternative is an operation that transmits one word of data to an adjacent neighbor. Since each "hop" in the communication network requires node intervention, this creates unacceptable communication latency, and unnecessarily burdens the nodes in the routing path. Additionally, the arbitrary source and destination message passing operation in Pi provides a more abstract view of the communication network, allowing greater implementation freedom for the machine builder.

A single operation supports message sending in Pi:

> **(send destination length priority type arg-1 arg-2 arg-3 . . . arg-n)**
>
> This operation constructs a message containing the send arguments and enqueues it for delivery to the named destination node. During transmission, **destination** is removed from the message by the network. At the destination node, storage (a segment) is allocated for the incoming message, using **length**. The message fields **type** and **priority** specify how, and in what order the message is handled by the receiving node. Message handling is described in a later section.

When a message arrives at a node, it is enqueued as a segment in the node's storage. In the Pi abstract machine, this happens independently of the node's current activities (i.e., a task executing on the node is not suspended when a new message is received).

2.3.1 The Network

The communication network is capable of delivering messages between a sender and receiver node. The details of the delivery (routing, in-transit buffering, etc.) are not visible in the interface. However, the network exhibits the following properties:

- Message delivery is guaranteed.
- Message latency is indeterminate.
- Message delivery order between two nodes is not preserved.
- The network can exert backpressure[2] on message sending nodes.

2.4 Task Management

Communication and task management are closely related in Pi. Message sending is one of two mechanisms for creating tasks (the other is task *calling*). All node activity stems directly from message reception. This section considers several aspects of task management in the Pi abstract machine.

2.4.1 Task Storage

Normally, all task storage is allocated at message arrival time. When a message is received, the task storage requirement is computed as the sum of the message length field and the task storage overhead size (i.e., space for processor state, etc.). A segment of this size is allocated, and the message is stored there.

In some models, the required task size cannot be determined statically. In these dynamic cases, an additional segment is allocated at runtime to provide the required storage.

Task naming, like synchronizations, is supported by segment names (NSO). Therefore, the allocation of task storage includes the assignment of a unique name to the task: the segment name. This name is obtained via the self operation.

> (self)
>
> This operation returns the segment name, which is used as part of a NSO address.

2.4.2 Task Dispatch

Both mechanisms for task invocation, **send** and **call**, use a type field to specify the appropriate handler. Depending on the machine implementation, this field can contain one of two value types: a physical address of a code segment or the segment ID of the code segment (if this is implemented differently). This distinction (address or ID) can be made at machine compile time or run time (using type tagging). In either case, the code segment must be resident on the node.

[2]This means that the network can halt nodes that are injecting messages.

To support handlers which are not guaranteed to be resident (i.e., *non-resident handlers*), a separate argument in the message or called task specifies a name which can be resolved to a segment ID using a user defined translation mechanism. After this translation is accomplished (and the non-resident handler is copied locally), **call** is invoked to begin handler execution. An example of non-resident handler support is described in Chapter 3.

2.4.3 Task Atomicity

Task execution is non-preemptive. A task executes until termination, or until a synchronization point forces a suspension. Non-busy wait techniques (explicit wake-up) are employed to resume suspended tasks.

A task can voluntarily suspend itself, perhaps allowing higher priority tasks (defined in the next section) to execute.

> **(suspend)**
>
> This operation suspends execution of the current task. It is immediately requeued for execution.

A task is terminated when the associated segment is destroyed.

2.4.4 Task Prioritization

Task execution ordering is supported via task queues. FIFO ordering in these queues is required, since non-FIFO ordering can result in livelock and deadlock in certain circumstances, and excessive task buffering. This guarantees that tasks which suspend on a synchronization are requeued in the order they suspended. If non-FIFO buffering is employed (for example FILO), two tasks can enter a livelock condition arbitrating for a synchronization[3].

Since the Pi abstract machine is fine-grained, each node normally has many tasks waiting to execute. In this circumstance, simple arrival ordering of tasks is inadequate, since the waiting time for critical tasks becomes long. Therefore, multiple task queues are used to allow more critical tasks to begin ahead of other earlier tasks.

When a message is received, the priority field determines where it should be enqueued. Priorities can be positive or negative integers, where zero is "normal" by convention. These queues determine an execution ordering where no task in queue **N** is started until all queues greater than **N** are empty. However, once a task begins, it executes to suspension or completion regardless of the priority of any received messages. There are no task interrupts.

[3]This condition was actually produced in a demonstration example used in Chapter 3. An early version of a simulator was erroneously coded with stack task queues. A livelock condition resulted.

Multiple task queues are recommended but not required of machine implementations. Machine designers are free to support a single queue, using resources for other features. Because of uncertainty in message delivery time, models should employ multiple queues for performance improvements rather than for assuring functional correctness.

2.4.5 Calling Tasks

Call is the second Pi mechanism for creating tasks. It is syntactically similar to a sequential procedure call. The new task is executed on the same node. Also, the caller task is suspended in a special way, so that no other node tasks can intervene when the new task is called or after it completes. Calling a task behaves differently than sending a message to the current node. With a message send, other messages can execute before or after the new task, disrupting the caller task's atomicity.

Call guarantees atomic execution of a code sequence, while allowing code sharing. Without it, code sequences would need to be macro-expanded at each call. With calls, the machine compiler can choose to macro-expand calls independently, while supporting Pi's semantics.

> (call length type arg-1 arg-2 arg-3 . . . arg-n)
>
> This operation invokes a handler on the current node. The length field specifies the required storage size in the task segment. Type specifies the handler. The task is executed immediately, with no opportunity for waiting tasks to run. When the called task completes, the caller task is immediately reinvoked.

Since called tasks are executed atomically within the caller, a called task can directly return a result to the caller task, rather than the more complicated alternative of sending a reply message. If a return operation is executed at any point during the called task the result value is returned from the Call operation. The returned result need not be read by a caller task. Additionally, non-called tasks (resulting from message passing) can execute the return operation with no effect.

> (return return-value)
>
> This operation stores return-value away, to be returned later when the called handler completes. If a return operation is executed more than once, the last value is returned.

2.4.6 Variable Argument Passing

Pi supports handlers which accept a variable number of arguments. However, the send and call operations pass a fixed number of parameters. A variant of these operations exists for variable argument passing.

(send-segment length type segment)

This operation is similar to send, except the arguments are taken from the specified segment.

(call-segment length type segment)

This operation is similar to call, except the arguments are taken from the specified segment.

2.5 Locality

Many types of locality can be exploited in a parallel problem solving system. The only concern here is identifying what facilities are required in this architectural interface to support locality. For example, as stated earlier, Pi neither requires nor precludes exploitation of storage reference locality. It is handled below the interface and is not part of Pi.

Spatial locality in node selection requires support in Pi since the communication topology is abstracted below the interface. Yet information about spatial locality is often available above the interface.

Several alternatives for specifying spatial locality were considered. However, no scheme allowed general spatial specification without placing constraints on the underlying substrate topology.

Pi provides three operations to support node spatial locality:

(node-id)

This operation returns the node ID of the current node. This supports the specification of NSO handles, and locality designation for tasks which should be executed on the current node.

(another-node-id where)

This operation returns a node ID of a node in the system. The locality of the node is determined by the where parameter which is one of the following: NEAR, FAR, or ANY. NEAR indicates that requested ID should be near to the current node. FAR indicates that a distant node is required. ANY indicates that any node in the system (including the current node) is appropriate. ANY assumes a random node ID selection. A Pi implementation may use this information to balance storage or compute usage, but this is not required.

(distance node-x node-y)

This operation computes the distance between node-x and node-y. The units of the measurement are unspecified, but they are proportional to communication latency between the nodes. This operation is most valuable for making relative comparisons of locality (e.g., what node is closest).

2.6 Sequential Operations

Beyond the operations already described, Pi provides a base of sequential operations. Even the most ambitious parallel models are beginning to recognize this need [5, 23, 44]. The following operations are included:

(plus x y)	(minus x y)	(times x y)	(divide x y)
(not x)	(or x y)	(and x y)	(xor x y)
(compare x y)	(rotate x y)	(arithmetic-shift x y)	(logical-shift x y)
(branch-zero test-variable label)		(branch-not-zero test-variable label)	
(branch-plus test-variable label)		(branch-not-plus test-variable label)	
(branch-minus test-variable label)		(branch-not-minus test-variable label)	

Most of these operations have obvious meanings. The compare operation compares two values (say X and Y), and returns 1 if $X < Y$, 0 if $X = Y$, and -1 if $X > Y$. Using compare and the branch operations, all branch types can be formed.

Sequential operations pose a dilemma in Pi. The design of Pi has avoided including things in the specification which are irrelevant to its purpose as a parallel architectural interface. Therefore, a complete set of sequential operations is not specified in Pi. The definition of such a set is outside the scope of this thesis.

However it is impractical to construct a complete sequential operation set using the operations specified above. For example, if hardware which computes square root is present in hardware, one would not want to have a software square root routine built from the operations above.

Therefore, the set of sequential operations included in Pi is intentionally left open. Future work in the design of a sequential architectural interface will complete this aspect of Pi.

2.7 Summary

This chapter has defined Pi, a parallel architecture interface. Pi is based on a set of generic parallel model requirements. Pi provides:

- operations to support both linearly indexed and associatively addressed storage
- several forms of data and control synchronizations
- a communication operation
- support for task management
- support for topology-independent specification of locality

operation class	Pi operations
storage (read-write)	read, write
storage (associative)	insert, match, remove, clear
storage management	create-read-write-segment, create-associative-segment, destroy-segment, segment-size
storage (node variables)	nodals
synchronization	attribute, probe, time, test-count, adjust-count
communication	send, send-segment
task management	self, suspend, call, call-segment, return
locality	node-id, another-node-id, distance
sequential operations	plus, minus, times, divide, not, or, and, xor, compare, rotate, arithmetic-shift, logical-shift, branch-zero, branch-not-zero, branch-plus, branch-not-plus, branch-minus, branch-not-minus

Table 2.1: Pi Operation Summary

- sequential operations

Table 2.1 summarizes the Pi operations.

Pi has been guided by evaluation criteria described in Chapter 1. Using these criteria, it is hoped that the interface is appropriately abstract, accurate, well-matched, and unbiased about model and machine implementation. Yet this can only be determined by studying the interface in use. The remainder of this thesis examines the effectiveness of this interface in supporting model mechanisms, and in being supported by a machine substrate.

Chapter 3

Building Models on Pi

A programming model is a collection of problem solving tools. The tools are abstract, and they are usually not provided directly in Pi. Instead, this abstract functionality is embodied in *model mechanisms* constructed from interface operations. A set of model mechanisms must be constructed for each model supported in Pi.

In this chapter, Pi's ability to support several model mechanisms is examined. A broad range of examples is presented to demonstrate various aspects of mechanism construction including task management, naming, synchronization, storage management, communication, locality, and sequential sequence support. This chapter examines four specific mechanism examples and two simple applications, including:

- shared memory (with caches)
- set synchronization
- object name translation
- non-resident handler support
- n-body simulation
- relaxation simulation

Since Pi is an architectural interface, not a programming language, Pi programs are very low level, with many details exposed. Programming in Pi is analogous to programming in a machine-independent parallel assembly language. Task management, naming, synchronization, storage management, communication, and locality issues are all explicitly defined in the programs.

High level language programming environments are necessary to develop application programs on a Pi machine. They provide a convenient specification environment with model mechanisms presented cleanly and abstractly. High level languages and

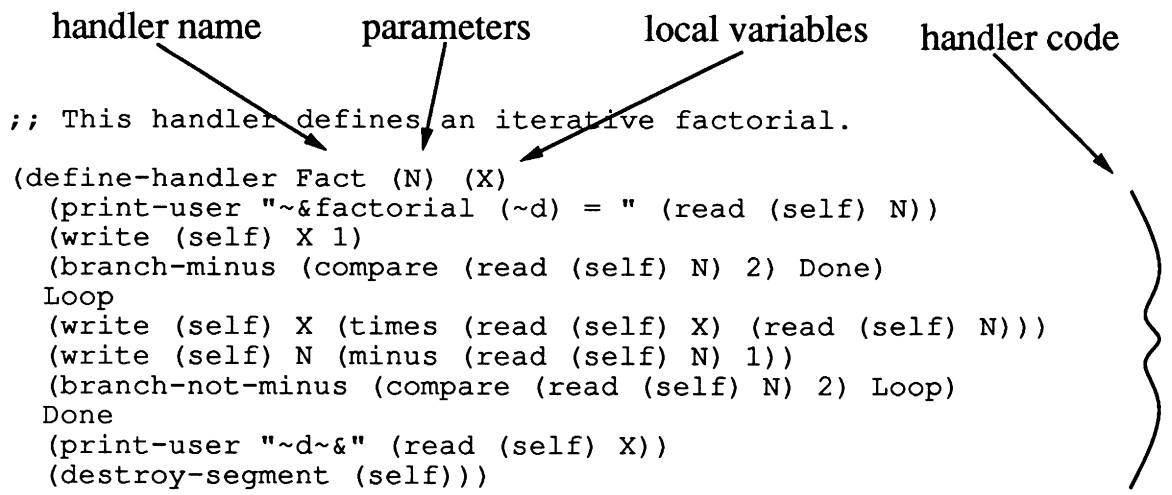


Figure 3.1: Factorial Handler

compilers are absent from this thesis because (a) their specification and construction is an open research issue in parallel computing, and (b) the goal of the thesis is to show that the model mechanisms which underlie high level languages can be efficiently supported by an architectural interface.

The model examples have been implemented, tested, and metered using a Pi simulator described in the next section. This chapter describes their implementation. In Chapter 4, the results of these executions are presented and evaluated. The code for all examples is provided in Appendix A.

3.1 PiSim: A Pi Simulator

It is difficult to construct mechanisms in Pi without an execution environment. For this reason, a Pi simulator, PiSim, was created. PiSim supports the operations defined in Chapter 2. It provides a way to define, debug, and meter Pi programs. PiSim includes a syntax for expressing Pi programs. This section presents a short description of the PiSim environment.

3.1.1 Handlers

Pi programs are collections of message handlers. Handlers in Pi are analogous to procedures in a sequential program. A handler consists of a sequence of operations which are sequentially executed on a node in response to a message. A handler also specifies the state (e.g., parameters and locals) used during the execution.

PiSim assumes that the code for each handler is present on all nodes (support for non-resident handlers is presented in an example later in this chapter). A simple handler for computing factorial is shown in Figure 3.1.

In a compiler generated Pi program, parameters and local variables are not mnemonically named as in this example. A handler specification only includes the number of parameters and required locals. These variables are named in PiSim to improve program readability. When a variable name is used in a handler, it is translated by PiSim to the appropriate offset into a segment. When a handler accepts a variable number of parameters, its parameters are referred to by offsets directly rather than names.

Sending a message is similar to calling a procedure. Handlers are invoked by **send** and **call** operations in other handlers. When a message is received on a node, the message type is used to select the appropriate handler to service the message. That handler is then invoked on the message. Figure 3.2 illustrates a factorial handler invocation.

PiSim also provides an operation to inject a start message into the system:

> (inject 'handler-name arg-1 arg-2 arg-3 . . . arg-n)
>
> This PiSim operation sends the message to a random node at time zero. Since this operation computes the message length, it cannot be used to invoke handlers which accept a variable number of arguments. When a message is injected into the system, all simulated state is first initialized. This prevents previous state from affecting the execution of a program, thus improving repeatability for debugging.

PiSim is implemented using a single priority message/task queue on each node. As explained in Chapter 2, this is a valid implementation of the interface. Multiple message priorities have not been used in the examples.

3.1.2 Nodals and Constants

An example of a nodal and constant definition is shown in Figure 3.3. Constants are often used in the examples to define a data structure. The constants define slot offsets in a segment.

Variables in PiSim are automatically typed. Automatic typing is not required in Pi, although PiSim provides no way of declaring a variable's type.

3.1.3 Special Operations

The Pi examples presented in this chapter contain a few additional operations which are not included in the Pi definition presented in Chapter 2. These operations are not formally part of Pi, but they are useful for presentation clarity.

> (print-user format arg-1 arg-2 arg-3 . . . arg-n)
>
> This operation prints the arguments using the LISP format string.

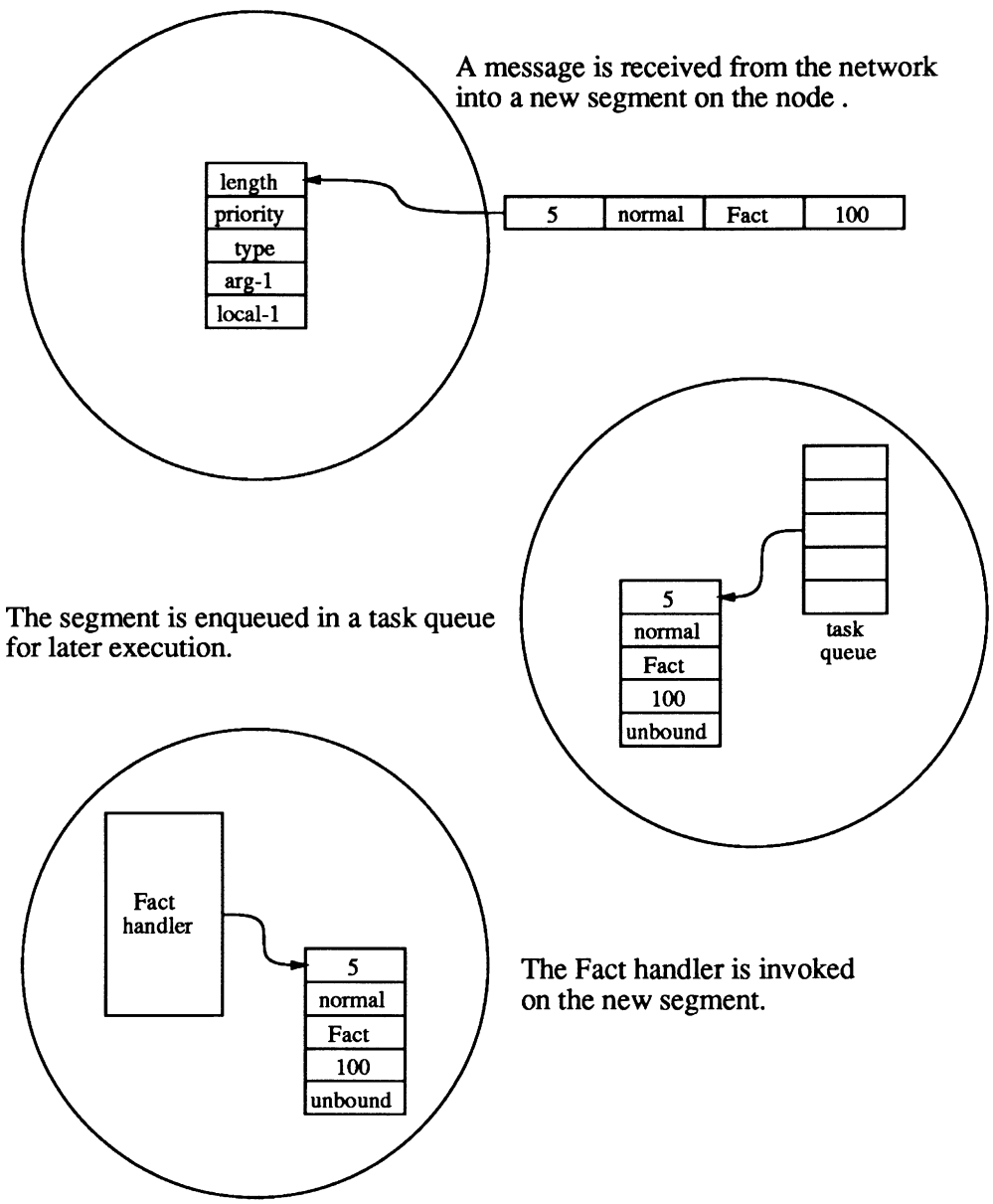


Figure 3.2: Handler Invocation

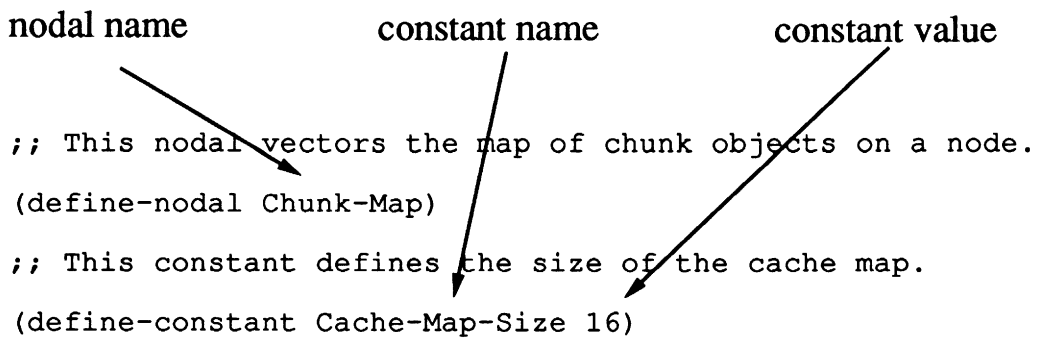


Figure 3.3: Nodal and Constant Definition

(set-debug-level level)

This operation sets the debug level, which controls the amount of debugging information provided by PiSim.

These sequential operations are used in the examples. Since PiSim supports them with the Common LISP equivalent, a detailed definition of their operation can found in [32].

(random x) (exponent x y) (mod x y) (ceiling x y)

3.1.4 PiSim Implementation

This PiSim implementation also includes several program evaluation and metering functions including operation profiling, task and storage statistics, and concurrency information. These facilities are used to evaluate these mechanism examples in Chapter 4.

PiSim is implemented in Common Lisp using Sun Common Lisp window and graphics extensions. It is expressed in roughly 2000 lines of code. The examples in this chapter execute on a Sun 4/60 in a range of minutes to several hours.

3.2 Shared Memory with Caches

This section describes an approach for supporting a shared memory model with caches, using Pi. This scheme has not been evaluated using application trace data. Its purpose is to exemplify the requirements of medium-grain to fine-grain shared memory systems[1] (e.g., [2, 9, 11, 12, 22, 35, 36]). This example demonstrates efficient shared memory mechanisms constructed with Pi.

[1]This task is complicated by the lack of message order preservation in Pi.

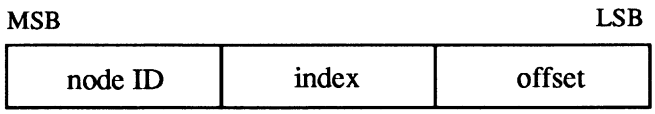
3.2.1 A Shared Address Space

The shared memory model requires the support of a logically contiguous address space built out of distributed, discontinuous memory. The basic unit of the model is a *line*, a small contiguous piece of memory that is managed as an atomic element of storage. Each line is mapped into a unique place in the shared address name space. The number of words contained in a line is normally a power of two.

Since shared memory it too large to fit on a single node, it is divided into pieces which are distributed across all the nodes in the system. Each piece is called a *chunk*[2]. In this implementation, storage for the entire shared memory is allocated when the system is initialized, rather than being allocated on an as-required basis. Chunk storage is composed of a set of line sized segments. A *chunk map* is constructed on each node to maintain the locations of line segments allocated there. Chunks are composed as collections of small objects (lines) rather than a single large array to simplify accesses using the segment paradigm in Pi.

Accessing a shared memory address requires locating the correct line on the correct node. There are a variety of ways to map node and line identification into the shared address. In this demonstration example, a simple scheme has been chosen.

A shared memory address is composed of the following fields:



Starting from the least significant bit, the first field is the line offset. This identifies a word in the line. This is followed by the index field which specifies a line in the chunk map of a node. Finally, the node ID field indicates the location of the shared address storage.

The node ID field is actually the node number of an abstract machine node; there is no indirection. This precludes the relocation of a specific shared memory location. Supporting shared memory relocation requires a distributed name translation mechanism. However it is not clear that this capability justifies the added complexity. Caches themselves provide a form of relocation. The functionality may be appropriate to support *virtual* shared memory addresses.

Address *braiding* is often used in shared memory systems. This is the technique of dividing and intermixing the node and index fields to scatter sequential address sequences (e.g., large objects) across several nodes. It is not employed in this example, although it has been implemented in a simpler, non-caching scheme. The requirements and overhead of supporting braiding are evaluated later. Address braiding is omitted in this scheme for clarity.

Caches take advantage of locality to avoid remote accesses on each shared memory reference. When a word of shared memory is accessed, the entire line in which it

[2]Other names (e.g., page, segment, block) were considered and dismissed because they implied unintended connotations about the implementation.

resides is obtained and stored in the cache to exploit spatial locality. Words are retained in the cache for future accesses to exploit temporal locality.

Like chunk storage, caches are constructed using a set of lines. However the *cache map* (analogous to the chunk map) is realized using associative storage. In order to maintain cache consistency, cached copies of a shared memory line are chained together forming a list. The permanent (chunk) line maintains a pointer to the head of this list, so that the copies can be invalidated when necessary.

The details of lines and maps are presented in the next section.

3.2.2 The Components

Each node in the system contains a local cache and a chunk of the shared memory. The cache and chunk are constructed using three segment types: lines, chunk maps, and cache maps.

A line is the basic storage element in this shared memory implementation. It is realized as a read-write segment containing twelve words. The segment is composed of the following fields:

Store: This field contains the eight words of data maintained in this line.

Link: For chunk lines, this field indicates the node containing the first cache copy of this line. For cache lines, this field holds the node containing the next copy of this line. If no copies exist, or if this is the last copy in the chain, this slot contains the value END.

Status: This field maintains the status of the line. A chunk line status can be one of the following: UNLOCKED, LOCKED, READ-ONLY, or READ-WRITE. Cache line status can be one of the following: INVALID, READ-ONLY, or READ-WRITE.

Request: This field is used by the cache lines to serialize cache line requests. When a cache line is not found, a memory access tries to acquire this field of the cache line. If it is locked (via the READ-ONLY attribute), the request suspends until it is released. Once this lock is obtained, it is held until the access is complete, guaranteeing that other accesses will not interfere.

Barrier: This field is used for synchronization between the cache and chunk line during an access operation. It is implemented as a b-sync.

Two map segments exist on each node to translate shared memory addresses to line segments. The chunk map is a read-write segment which maps an index for a node's chunk to a line containing the data. The cache map is an associative lookup table keyed on shared memory addresses (minus the line offset). It translates to a cache line. If the line status is READ-ONLY or READ-WRITE on a read, or READ-WRITE

on a write, the cache hits. If the line status is INVALID for a read or INVALID or READ-ONLY for a write, the cache misses.

The cache map must be *safe* (i.e., entries are not overwritten by other insertions). Cache maps are also bound in size to control the amount of local memory used by the cache. Because of these two facts, a read or write operation may result in the replacement of the current cache contents.

Cache lines are locked via the request field of a cache line. Chunk lines are locked using the status field.

3.2.3 The Protocol

The shared memory scheme implemented in this example is defined by two interacting protocols: the chunk line protocol and the cache line protocol. These protocols are presented in this section.

The chunk protocol is summarized in Figure 3.4. It defines the behavior of a line of chunk memory. There are three normal states: Unlocked, Read-Only, and Read-Write, plus several locked intermediate states. The normal states have the following meaning:

Unlocked When a chunk line is in this state, there are no cache copies on any other node. If an access request is made to a line in this state, the data can be furnished immediately.

Read-Only This state indicates that one or more read-only cache copies exist in other nodes. A read access can be added to the chain of linked cached copies. However a write access requires that all read-only cache copies be invalidated first.

Read-Write This state indicates that a single read-write cache copy is contained by another node in the system. Any access to this line must be preceded by updating the chunk with the state of the exclusive cache copy.

The locked states in the protocol are intermediate states when an access is being completed. When in these states, all additional accesses are blocked until the current access completes. This is why read and write messages are excluded in these states. The locked states have the following meaning:

Locked Read In this state, a read access is in progress. This state is entered when the line data has been replied to the node that requested the line. It stays in this state until an acknowledgment is received from the requesting node.

Locked Write This state is similar to Locked Read, except an exclusive read-write copy of the cache has been replied to the requesting node.

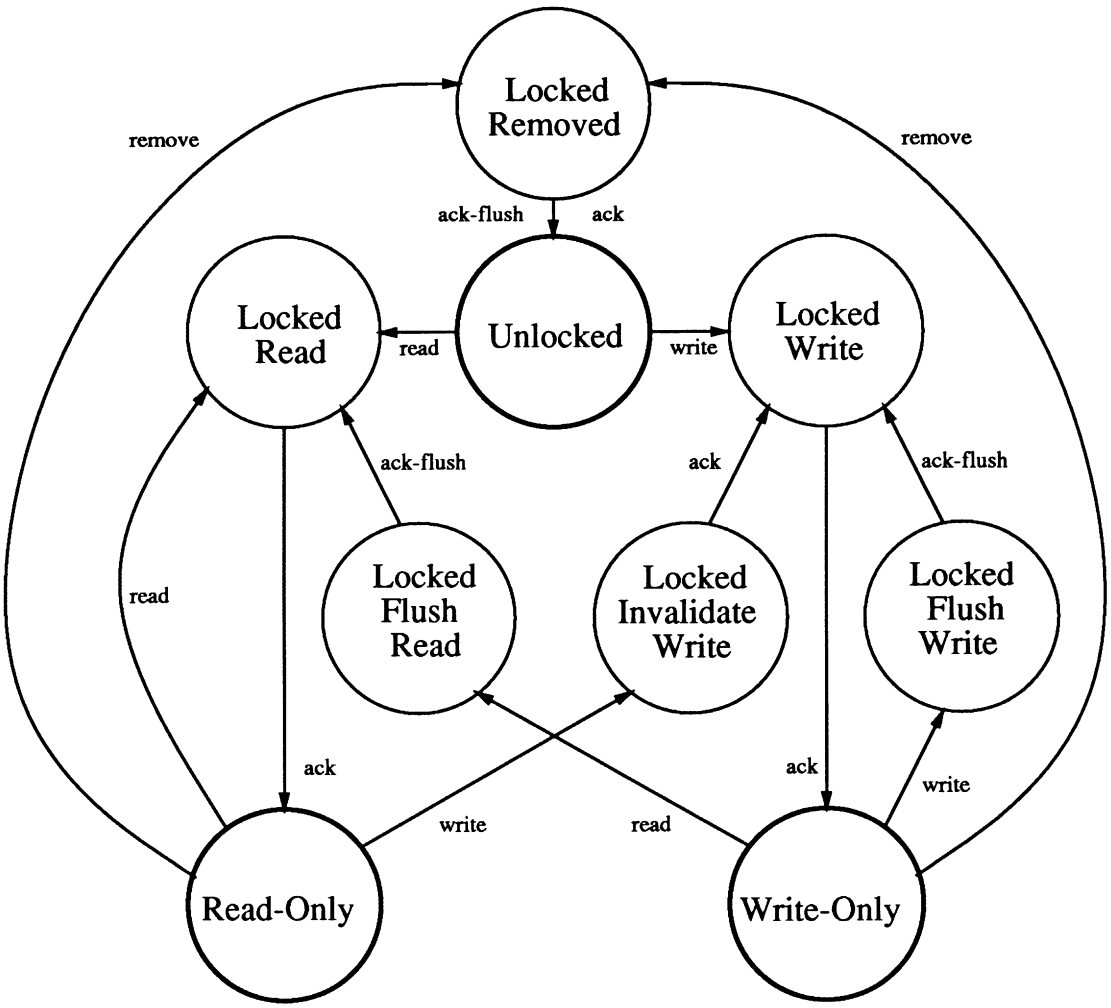


Figure 3.4: Chunk Entry Protocol

Locked Invalidate Write When a write access is received and the current state is Read-Only, all read-only copies must first be invalidated. This is accomplished by sending an invalidate message to the first node in the read-only cache chain. This message is forwarded along the chain until all copies are invalidated. At this point, an acknowledgment message is returned to chunk line, changing its state to Locked Write.

Locked Flush Read When a read access is received and the current state is Read-Write, the exclusive cache copy must first be flushed (and invalidated[3]) before the read-only copy can be replied to the read requester. After the flush message is sent, the chunk line remains in this state until a flush acknowledgment message (with the updated line data) is received. The chunk then moves to the Locked Read state.

Locked Flush Write This state is similar to Locked Flush Read, except a write access is requested. After a flush acknowledgment is received, the chunk line enters the Locked Write state.

The other half of this shared memory scheme is specified in the cache protocol summarized in Figure 3.5. It defines the behavior of a line of cache memory. In the protocol, accesses are separated into two classes: those that match the address contained in the line, and those that don't. Reads and writes that match are printed in italics. Reads and writes that access a different address are printed in boldface. This protocol includes three normal states: Invalid, Read-Only, and Read-Write, plus several locked intermediate states. The normal states have the following meaning:

Invalid When a cache line is in this state, the cache entry is empty. The empty line continues to be included in the cache map. However, it can be acquired immediately during an access.

Read-Only This state indicates the cached copy of the line can be read but not written. A cache line in this state results in read hits and write misses. The cache line can also receive an invalidate message, which causes it to invalidate the line and forward the request to the next node in the cache, or, if it is the last line in the chain, it sends an acknowledgment to the chunk line.

Read-Write In this state, all accesses can be served locally. A cache line in this state can receive a flush message from the chunk line. Here, it sends a flush acknowledgment, which contains the line update, to the chunk line. The cache line then enters the invalid state.

The locked states of the cache line protocol differ from those of the chunk line protocol in that some accesses are permitted (not blocked). The unlocked states have the following meaning:

[3]Alternately, the Read-Write cache copy could be converted to read-only and placed on the read-only chain.

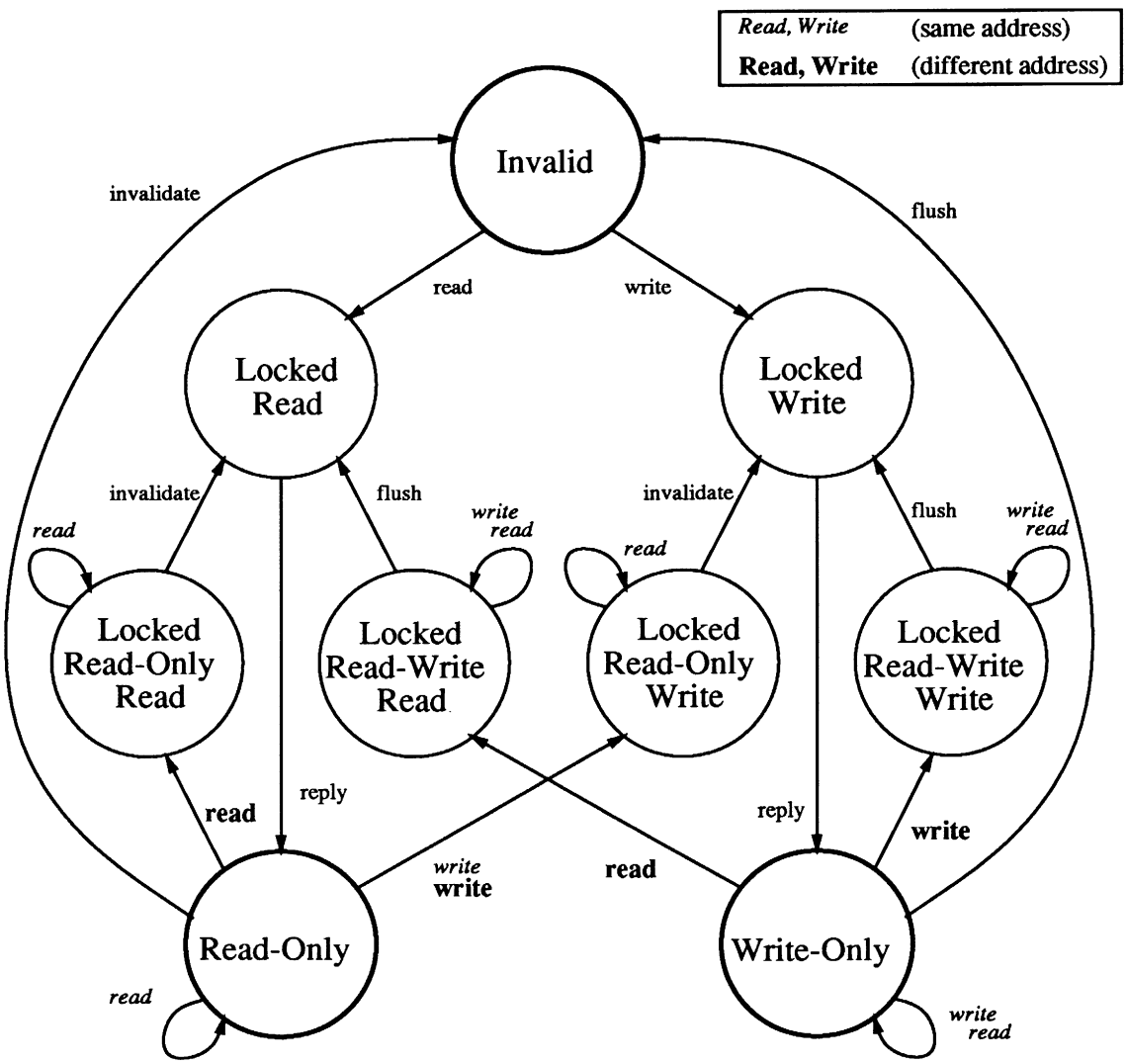


Figure 3.5: Cache Entry Protocol

Locked Read In this state, a read message has been sent to the chunk line, but the replied data has not been received. This state implies that the required cache line has already been acquired. In this state, all accesses are blocked until the request data is received.

Locked Write This state is similar to Locked Read, except a write message has been sent to the chunk line.

Locked Read-Only Read This state indicates that cache line contains a read-only copy, but different read accesses require the same cache line. A remove message has been sent to the chunk line of the current copy. While in this state, read accesses of the current cached line can complete. All other accesses block. This state is left when an invalidate message is received from the current chunk line.

Locked Read-Only Write This state is similar to the Locked Read-Only Read state except a write access is pending.

Locked Read-Write Read This state results from a read request that requires a cache line currently containing an exclusive read-write copy. As in the Locked Read-Only states, a remove message has been sent to the chunk line owning the current cache line contents. In this state, read and write accesses to the current cache line can complete. All other accesses are blocked.

Locked Read-Write Write This state is similar to Locked Read-Write Read except a write access is pending.

In both the chunk and cache protocols, messages other than blocked accesses (read or write) not drawn in the diagram are undefined and result in an error. A complete cross-product of messages and states is presented in the shared memory code in Appendix A.

This shared memory scheme does not require node to node message order preservation on the network, since message ordering is not guaranteed in Pi. Because of this, both chunk and cache lines are locked, guaranteeing exclusive access. This creates an additional requirement for separate instruction and data caches, since a deadlock situation could result if an instruction and a value it requests require the same cache line.

A virtual shared address space can be constructed using the same mechanisms employed in this example. A *page* field is added to the address, and the described shared memory scheme serves as the "physical" memory. When the page misses in the cache, and it is not present in physical memory (as determined by the page table), the page is fetched from disk. Disks appear in Pi as specially designated abstract nodes with larger local memories.

3.3 Set Synchronization

This section demonstrates set synchronization, a form of intertask control synchronization used in several parallel models of computation, especially data parallelism. It is defined as follows:

Consider a set of tasks $T_1 ... T_m$. Each task T_i contains two synchronization events: an enabling event E_i and a dependent event D_i. A set synchronization guarantees that no task passes its dependent event until all tasks have reached their enabling event. This constraint requires that $\forall i, j, D_i > E_j$.

Intuitively, a set synchronization keeps a set of related tasks "in step" within the quantization of the enabling and dependent events. After all tasks have reached E, they can execute independently until the D for the next iteration is reached.

Set synchronization is often used for data parallel iterative algorithms. In these algorithms, the set synchronization guarantees that all data sets are executing the same iteration of a procedure.

The perceived importance of set synchronization varies widely across the parallel computing community. It has been used effectively to implement many applications, especially ones employing data parallelism. These applications are normally executed on machines which provide a fast global synchronization mechanism (e.g., SIMD machines such as the Connection Machine [56]).

Some argue that the inclusion of a hardware global synchronization mechanism is overly restrictive, particularly when synchronization is provided at each instruction (SIMD). Since most algorithms require less frequent synchronization, program execution is unnecessarily constrained. The technical ramifications of hardware supported global synchronization are often serious because of the required communication. Also multiple independent set synchronizations are not possible on these machines. Others argue that global synchronization is subsumed by data synchronization (e.g., dataflow). However, some programming models employ global synchronization to eliminate the need for data dependency analysis.

A global synchronization mechanism is not explicitly included in the Pi abstract machine. Because of the difficulties and liabilities of a hardware implementation (e.g., long cycle times for global signals to propagate, restrictions of machine scalability), global synchronization hardware support is not expected in machine designs. Instead, set synchronization (including global synchronization) is supported on top of the Pi abstract machine. Synchronization time is not constant as in SIMD machines. It is a function of (a) the log of the number of elements in the set, (b) the distance between elements in the set, and (c) message traffic (network traffic and message queue lengths). While this overhead may be greater than that of SIMD machines, it is paid only at required synchronization points. The set synchronization overhead is partially masked if tasks perform work that overlaps the synchronization after E but before D. Pi also provides simultaneous multiple set synchronization not supported in SIMD machines.

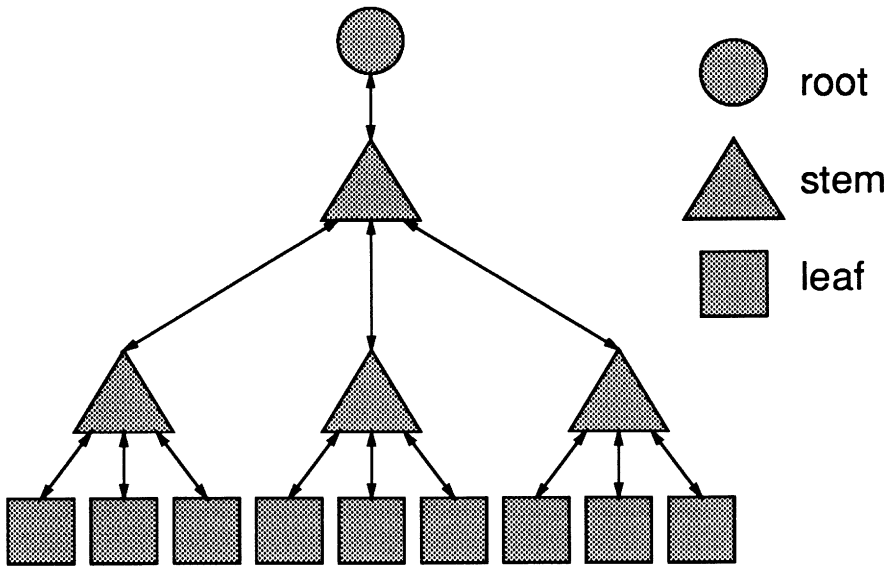


Figure 3.6: Set Synchronization Overview

3.3.1 The Components

This example demonstrates set synchronization in Pi. A set synchronization is used to guarantee lock step execution of iterations of a procedure on several nodes. Implementing a real application using set synchronization would unnecessarily complicate the example and obscure the mechanism being illustrated. Instead, a simple counting loop serves as the "work" of each iteration. However, a real task would perform correctly in place of the loop.

Synchronization is accomplished via a synchronization tree in which the iteration tasks are leaves. The structural elements, or *stems*, distribute start signals from the root and combine finish signals from the leaves. Figure 3.6 illustrates this assembly.

The leaf behavior is summarized in Figure 3.7. It begins by waiting for a start signal from its parent (a stem). When the start signal arrives, the leaf begins execution of an iteration of a procedure. In this example, this task is a simple counting loop. However a more complicated procedure could be undertaken which communicates with other leaves, performs computation on a piece of a distributed data set, or executes run-time selected sub-procedures. When a task reaches a defined point in the computation, it sends a finished signal to its parent. This point is normally at the end of the iteration, although a task may perform additional computation before waiting for the next start signal.

Stem objects form the structure of the synchronization tree. Their behavior is illustrated in Figure 3.8. Stems distribute and combine signals between the leaves and the root. A stem begins waiting for a start signal from its parent. When it arrives, the stem forwards the signal to its children. An s-sync is used to guarantee that each start signal is relayed. Then the stem adjusts a b-sync to receive the expected number of finish signals coming from its children. The stem then waits for all finished signals

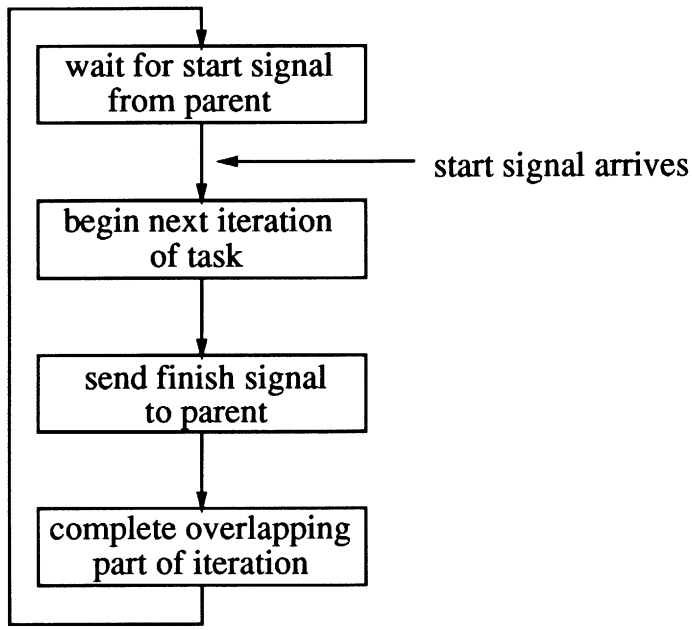


Figure 3.7: Leaf Behavior

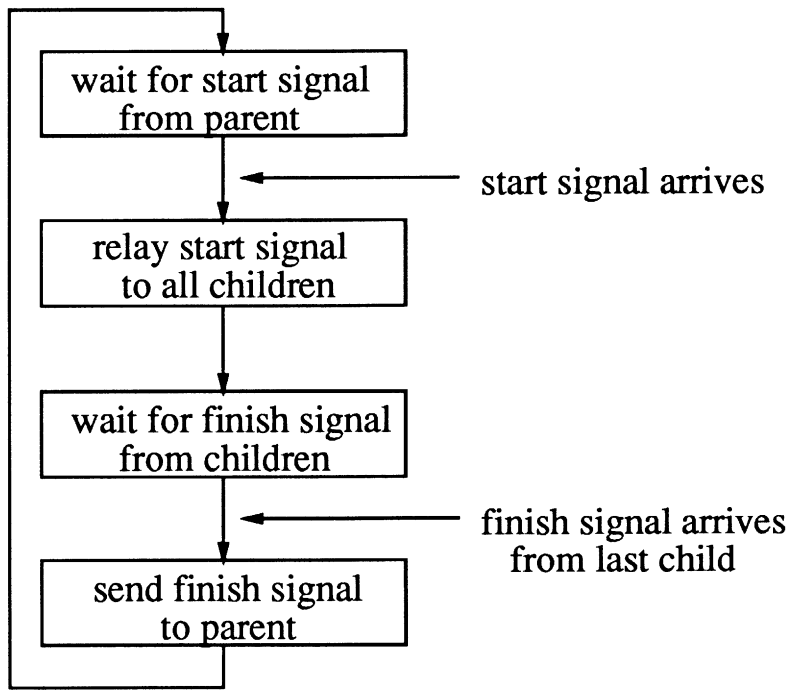


Figure 3.8: Stem Behavior

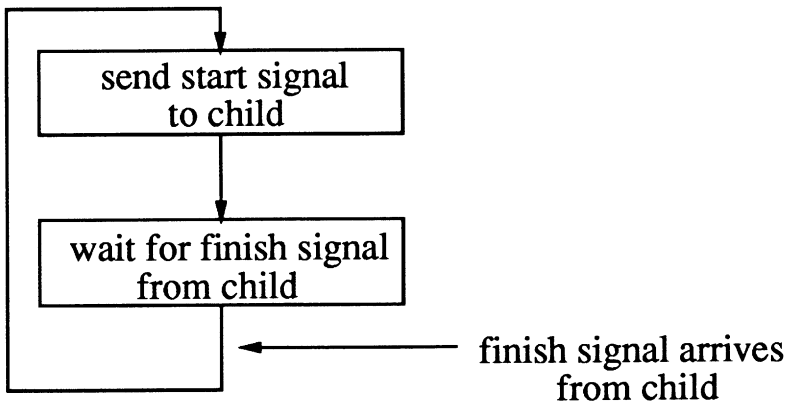


Figure 3.9: Root Behavior

to be received. After the last signal is received, a finished signal is forwarded to the parent.

The execution of the tree is orchestrated by the root. It behavior is shown in Figure 3.9. It starts the execution of an iteration by sending a start signal to the top stem. It then waits for a finished signal indicating that all leaves have completed the iteration. It then sends a start signal for the next iteration.

In this example, data is not transmitted in the start and finish signals. However, this feature can support applications requiring broadcasted data for each iteration, or data combination and collection to the root. For example, the finish signal may include information used by the root to determine the data broadcast in the next start signal.

This example employs a general tree building algorithm to create an optimal tree of a specified set size. The stem fanout can be adjusted to achieve the desired balance between communication delay (tree depth) and message serialization at the stems (branching factor). When the root determines that the final iteration has been completed, it sends a special end signal. When this signal is received, each element of the tree deallocates all storage and ends (i.e., the algorithm is self cleaning).

3.4 Object Name Translation

This example demonstrates object[4] name translation. This mechanism translates an object name to the location of the object (e.g., a node number). Several name translation mechanisms are employed in other examples in this chapter. For example, in the n-body example (described later in this chapter), a translation table for all objects is maintained on a single node. In the n-body example, the communication patterns are static, so all translations are performed during the initialization phase.

[4]An object can be a segment or task (or both).

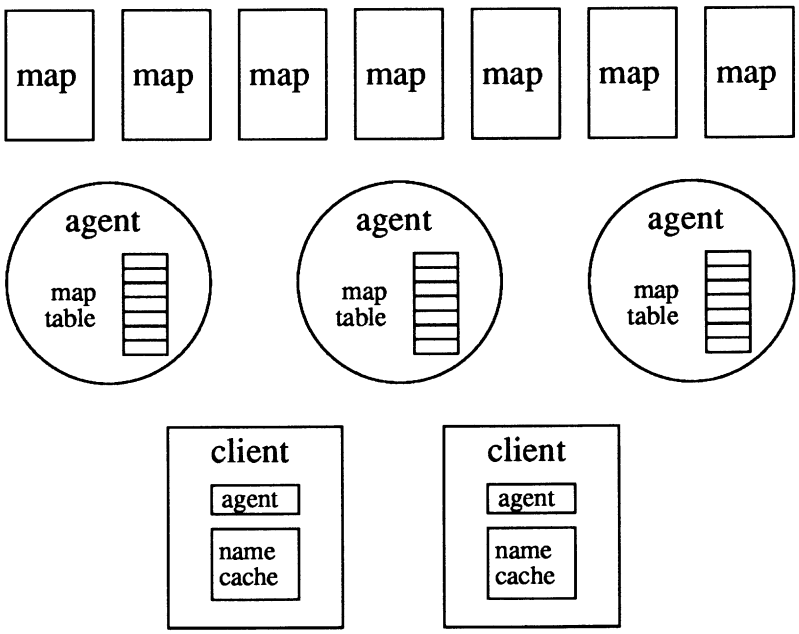


Figure 3.10: Translation Overview

Shared memory demonstrates two other types of translation. A line in the shared address space is algorithmically translated to the node number where it resides. When received at the correct node, the address is translated to a segment ID via a table. The shared address space is itself a translation mechanism of addresses to values. It supports consistent caching so translations can change over time. However the translation data itself is fixed. Also, name allocation is restricted by a rigid connection between logical names and physical addresses.

In the n-body and shared memory examples, objects are fixed. Therefore no provision is made for a translation to change. Once a translation is known, it assumed to be correct forever. The translation mechanism provided by shared address space supports an important additional capability. Objects are allowed to migrate. Object migration requires that the translation system maintains consistent information about the locations of objects even as the objects move from place to place. However, guaranteeing this consistency has a high synchronization and communication cost. Since an incorrect object translation can be detected easily, a less expensive technique can be employed.

In this example, a logical segment namespace is supported which (a) allows segment migration, and (b) the segment name is not rigidly connected to a physical address. The first feature provides more flexible storage, task, and communication management. The second feature supports more general name allocation techniques, and more flexible name resolution.

3.4.1 The Components

The three components in this system, Maps, Agents, and Clients, are shown in Figure 3.10. Maps store the translations in the system. Each map is a small associatively addressed table of name/value pairs. Maps can grow, and have some mechanism for dividing if they become too large. However this division is an internal process. A map continues to be seen as a single (named) object by the other system components. Maps are distributed randomly throughout the system.

Agents are the intermediaries between maps and clients. An agent is the "front end" to the distributed collection of maps. Each agent maintains a table of the current map locations. When a translation transaction is received, the agent performs a hash operation on the name to select the appropriate map to receive the request. The agent then forwards the transaction to the map.

Agents are connected in a ring[5] for communicating special transactions which involve all agents. These include map location updates (used both in system initialization and when maps migrate) and client requests for the closest agent. Using the agent ring, new agents can be created when one agent becomes overloaded. The agent simply creates a new agent "downstream" in the ring and passes a copy of the map location tables to it. It can then redirect clients to the new agent.

Clients are the end users of the translation system. Each client maintains the location of a nearby agent. All translation transactions are directed to that agent. A client also maintains a local translation cache which eliminates message requests for recently accessed translations.

Maps and agents are created at system initialization time. Clients are created as required by an application. A client is created on the same node as the application task needing access to the translation system. More than one client may be created on a node to improve cache performance.

When a new client is created, it is given a prototype agent. This can be any agent in the system. As part of the client's initialization procedure, it requests the nearest agent from the prototype agent. This way, a nearby agent is acquired, reducing network traffic and transaction latency.

In the example system, map division, map migration, and agent spawning are not implemented. However, their implementation is straightforward.

This translation system is designed to maintain object name $\rightarrow$ object location translations. Since some models allow object migration, translation bindings can change over time. As a result, translations are not guaranteed.

There are two circumstances which result in incorrect translation. Since client caches are not updated when a translation is changed, they can contain stale data which is incorrect. Or, a transaction can get a stale translation from the map (a new location arrives after a location request).

[5]If the number of agents is large, a lower diameter connection structure can more efficiently provide this function (e.g., a tree).

Because of the possibility of incorrect translations being supplied, the system that uses the information (e.g., message delivery) must be capable of detecting errors. One error response is to return the message to the sender. The sender then requests the translation again, this time without checking the client cache. The returned value updates the client cache, overwriting a possibly stale entry.

In this implementation, the translation system supplies the node where the object (a segment) resides. The message is then sent to that node along with the object name. That node performs a second translation to obtain the segment number. This is the mechanism which detects incorrect translations. If an object is moved within the node memory, but remains on the node, the distributed translation address does not change. Updates only result from internode migration.

3.5 Non-Resident Handlers

PiSim assumes that the code for all handlers resides on each node. For many models of computation (e.g., concurrent object oriented), this is impractical. There are two classes of handlers supported in Pi: *resident* and *non-resident.* Resident handlers are small in number, and "wired down" on every node (i.e., they are always present). Non-resident handlers are installed on a node as they are required. After they have been used, they can be removed.

This example demonstrates support for non-resident handlers. In this scheme, a single, reference copy of each handler (a *reference handler*) resides on one node in the system. When a handler is required to handle a message on a node, a copy of the reference handler is obtained first. Handlers are cached on the nodes to reduce handler requests and improve handler dispatch time. Since handlers are immutable, a cache consistency scheme is not required.

3.5.1 The Components

An overview of the example is shown in Figure 3.11. It is composed on five pieces: the reference handler map, the current handler map, the translation client, the requested handler set, and the dispatch handler.

Reference Handler Map One reference map resides on each node. It includes all reference handlers that are maintained on that node. It is associatively addressed using the non-resident handler's name.

Current Handler Map Each node also contains a current handler map. It includes all cached handlers on that node. The size of this map can be bound to prevent cached handler storage from growing too large. It also is associatively addressed using the handler's name.

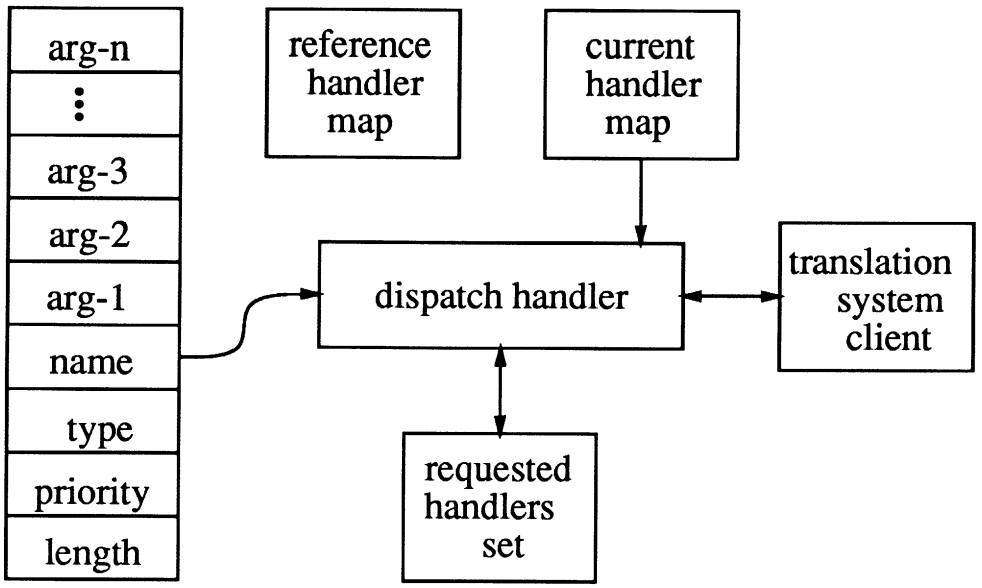


Figure 3.11: Non-Resident Handler Overview

Translation Client This non-resident handler example uses the translation mechanisms described in the last example to determine the location of a required reference handler. The translation system translates a non-resident handler name to the node ID on which it resides.

Requested Handler Set Each node contains a requested handler set which maintains information about which handlers have already been requested. It also provides access to the synchronization object which requeues dispatches when the handler arrives.

Dispatch Handler This is a resident (wired down) handler which provides access to the non-resident handler mechanism.

The non-resident handler dispatch procedure is shown in Figure 3.12. This example demonstrates variable argument passing in Pi. It also shows how synchronizations can be combined with associative sets (synchronization objects are stored in the requested handler set for non-busy waiting).

The next two sections consider collections of several model mechanisms used to solve two parallel scientific applications: n-body and relaxation.

3.6 N-Body Simulation

This example simulates the mutual interaction of N bodies in the absence of external forces. Algorithmically, this is a straightforward implementation which does not

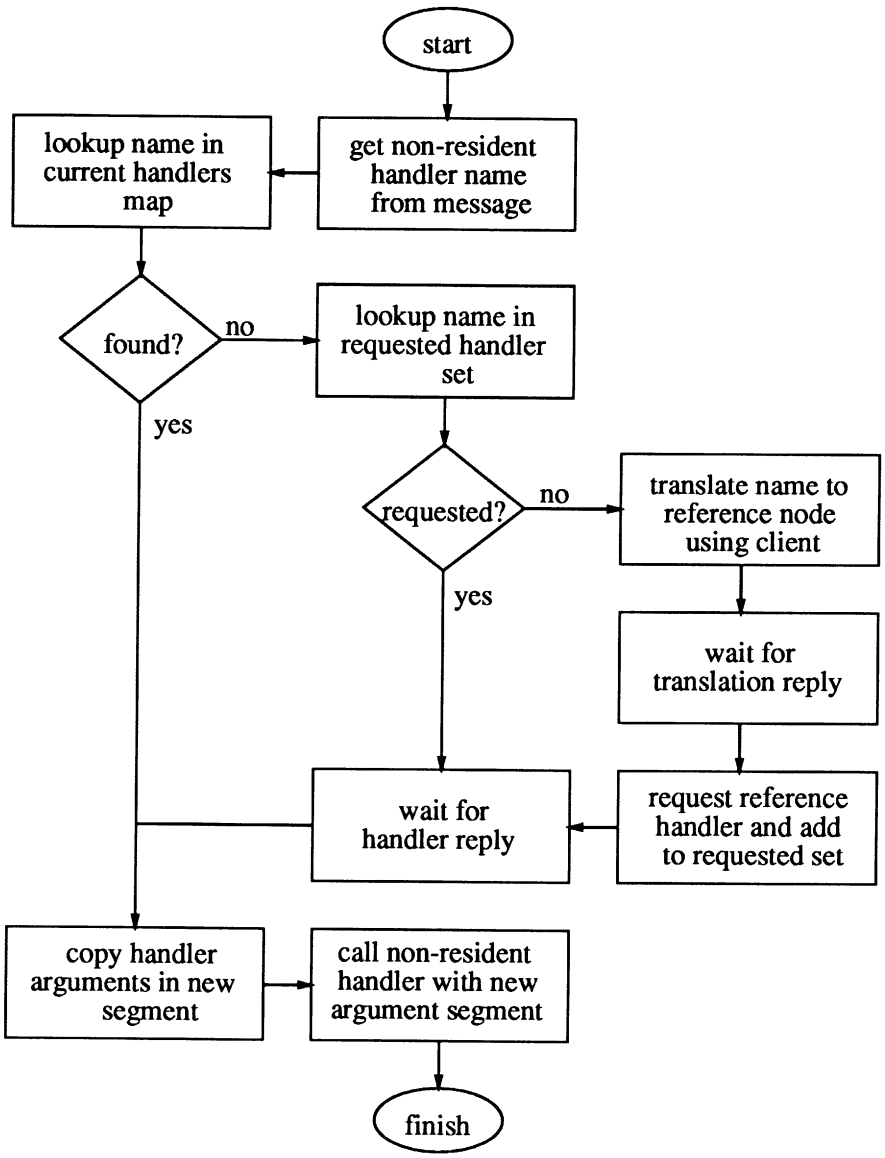


Figure 3.12: Dispatch Procedure

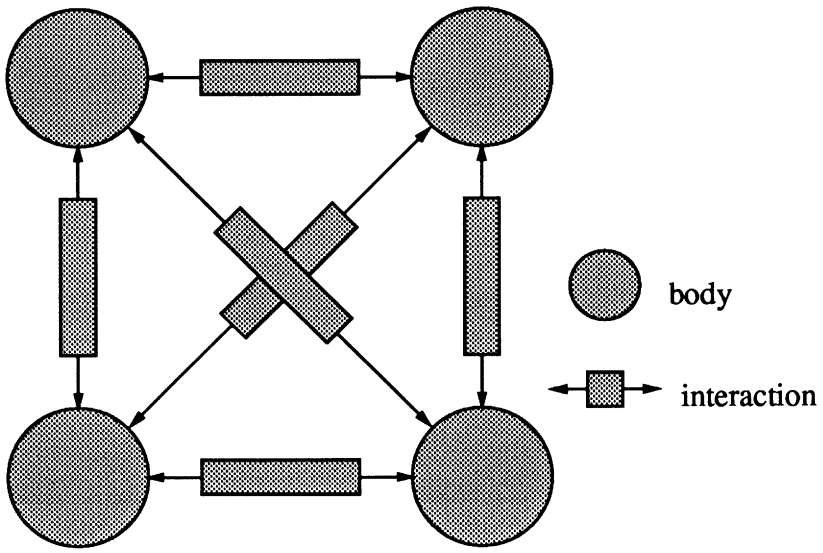


Figure 3.13: N-Body Overview

incorporate recent approaches to the problem (e.g., Greengard [24], Zhao [59]). However, the application demonstrates several fundamental model mechanisms required by static and dynamic dataflow, concurrent object oriented, and CSP models. The program simulates the two dimensional problem, but is easily generalized to three dimensions.

Normally, applications would not be writing directly in Pi. This example, and the relaxation example which follows, demonstrate how application requirements are met with Pi. A language compiler would ease the programming and debugging task.

3.6.1 The Problem

The n-body example is illustrated in Figure 3.13. Consider N bodies which are randomly positioned in a plane. Each pair of bodies is acted on by a mutual force between them, which is proportional to some characteristic of the bodies (e.g., the body's mass) and the distance between them (e.g., an inverse square relationship). In this example, gravitational attraction is modeled. The force between two bodies is expressed by the equation:

$$\overline{F} = \frac{G \cdot M_1 \cdot M_2 \cdot \overline{(X_2 - X_1)}}{|(X_2 - X_1)|^3}$$

If there are N bodies in the system, there are $(N * (N-1)/2)$ interactions. For this example, the mass of each object is:

$$ObjectMass = BaseMass + (ID \cdot DeltaMass)$$

where ID is the object number, assigned $0 \ldots (N-1)$, and base and delta masses are parameters of the system.

3.6.2 The Components

The simulation is composed of two basic types of objects: *bodies* and *interactions*. Body objects maintain the ID, position, velocity, and net acceleration of the body, and pointers to the $N-1$ appropriate interactions. Interactions maintain the distances and masses of the two bodies participating in the interaction, and compute the net force and resulting acceleration components. Interaction objects also maintain pointers to the two appropriate body objects.

The first interesting model mechanism demonstrated by this example occurs in the system initialization handler (the code is provided in Appendix A). Here, the handler must interconnect newly created body and interaction objects which are randomly placed on nodes in the system. Although a clever node encoding scheme could reduce network traffic and simplify interconnect, such a scheme was already demonstrated in the shared memory example. An alternative technique is used.

The startup handler creates two segments to hold the newly created node object locations (node and segment ID). As the node creation messages are sent, the destination node ID is stored in the node segment. But the object segment ID must be returned when the object is instantiated on the remote node. After the startup handler sends out the body initialization messages, the are followed by the interaction initialization messages. These messages include the locations of the bodies. However, since the segments containing body locations are initialized as d-syncs, no explicit synchronization is required between the two processes. This demonstrates simple data synchronization, which is required in nearly every model.

When the body and interaction initialization messages are received on the selected node, object storage is allocated and initialized with the formals as the message is removed from the network (i.e., the message storage *is* the task storage). Storage requirements which are not known at compile time (e.g., interaction location storage maintained by bodies) is allocated explicitly.

When body and interaction objects are created, d-syncs support rendezvous synchronization. Interaction objects send their locations directly to the appropriate body objects. When these locations are written on the body objects, linkage is complete.

The body and interaction initialization handlers include the main simulation loop. Therefore no additional task invocations (other than the communication messages) are required.

The behavior of a body is described in Figure 3.14. It begins by sending its position to each interaction in which its involved. It then waits for all acceleration components from the interactions. It calculates a new velocity and position. Then, if the final iteration count has not been reached, it begins another iteration.

An interaction's task is shown in Figure 3.15. It calculates the distance and force between two objects. Then it distributes acceleration components to the two bodies. Acceleration rather than force components are computed since the interaction requires the body masses for computing the interaction force. Body objects do not maintain their mass. An interaction also tests the iteration count to detect termination.

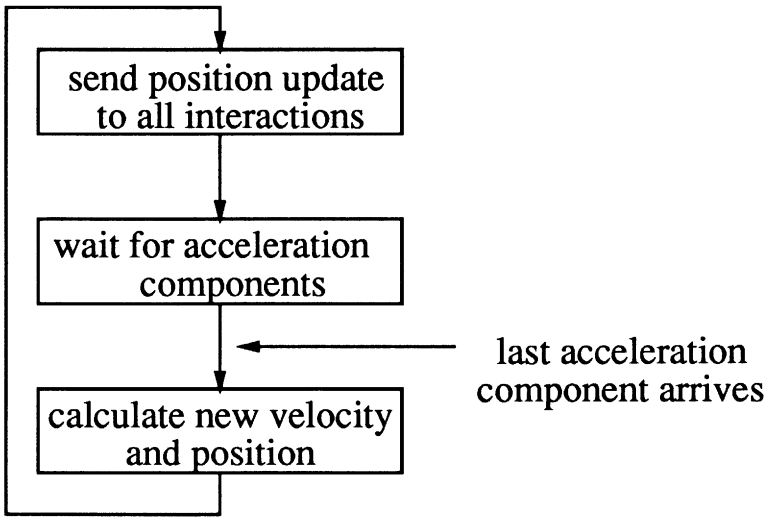


Figure 3.14: Body Behavior

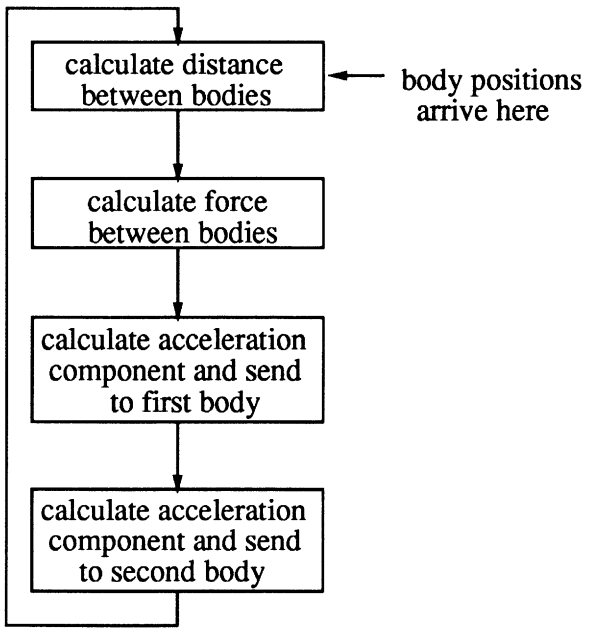


Figure 3.15: Interaction Behavior

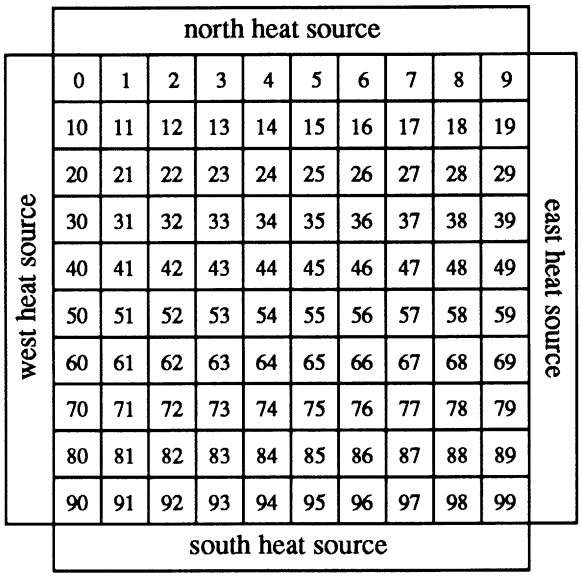


Figure 3.16: Relaxation Overview

Two synchronizations are required for these interaction loops. The body routine requires a barrier synchronization (b-sync) to determine when all acceleration components are received. Before the position updates are sent, the b-sync is set to the number of interactions. When all updates are sent, the b-sync count is tested. This typically results in suspension of the body loop. When an acceleration component is received, the barrier is reduced by one. The last component requeues the body loop to continue.

The interaction loop employs an s-sync to guarantee that each position update is used exactly once. If the data has not been received, the interaction loop suspends until it is written. If the position data has not been read by the loop, the update handler which writes the value suspends until it has been read. Access is synchronized on a per datum bases, avoiding unnecessary synchronizations between objects.

S-syncs can be used in static dataflow to eliminate acknowledgments without placing restrictions on dataflow graphs. Only flow control messages are required to avoid buffer overflow.

When both loops end, all storage is reclaimed.

3.7 Relaxation

This example uses a relaxation algorithm to computes the thermal equilibrium of a rectangular plate in contact with four heat sources at different temperatures. The plate is divided into an array of *elements*. Each element performs several iterations where it computes its new temperature as the average of its four neighbors.

This is a simple relaxation algorithm. A more sophisticated multi-grid technique (e.g., Mol [42]) is more suitable for real applications.

3.7.1 The Components

The number of elements created in the simulation is specified by constants in the program. Figure 3.16 illustrates the layout for the 10 by 10 plates simulated in this example. Each element is identified by its *index* (the number in the element box).

A single task constructs all the elements by sending a start message to different nodes. As in the shared memory example, node numbers are used as a naming scheme, and as a way to spread out the computation. An element resides on node N where N is its index modulo the number of nodes used in the example. An element computes the address of its neighbors using the array sizes. It also determines whether it borders on the plate boundaries.

Unlike the n-body example, an element never gets a pointer to its neighboring element's segments. Here, the creating task would be swamped by neighbor segment requests. Instead, an element map is created on each node. It performs translations between index name and elements residing on that node. So an element reaches its neighbors by computing the neighbor's node and sending a message there. When the message arrives, the destination's name is translated using the node's element map to the correct segment.

Once all elements are initialized, they begin the first iteration of the relaxation. This procedure is shown in Figure 3.17. Each element executes a specified number of iterations, then returns its final temperature to the creating task for display.

This scheme demonstrates the utility of s-syncs in regulating producer/consumer synchronization. Each element includes an s-sync for each neighbor. The neighbors produce temperature updates which write these values. The element consumes the values when calculating its new temperature.

Because of this algorithm's design, an s-sync never has more than one value presented to be written before a read is executed. This is because of the data dependency cycle between neighboring cells. In producer-consumer situation where the producer's output rate is independent from the consumer's input rate, it is possible to have multiple tasks waiting to write an s-sync with a produced value. Since the network does not preserve message order, some form of message sequence numbers is required to prevent produced values from being written out of order. When the producer generates a value, it also assigns it a sequence number. Before these produced values are written to an s-sync, the value sequence number is first tested to see if it is the next produced value.

This scheme presents a problem: how does an element know when its neighbors have been initialized? This example employs a spin-lock technique as follows. When a temperature update arrives on a node, the element index is referenced in the element map. If it is not found, the task suspends, then tries again. This allows other tasks to

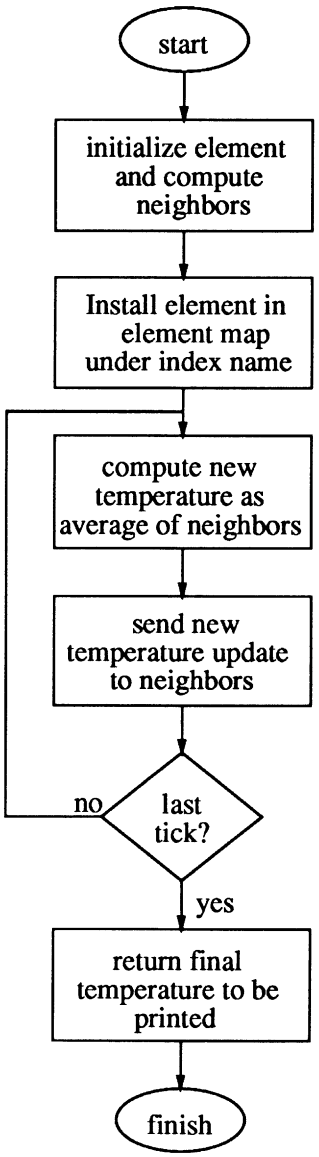


Figure 3.17: Element Procedure

execute on the node before the next query. When the element creation task installs the requested element in the map, the update task accesses it and completes.

The use of a synchronization object in the associative set, used for non-resident handlers, could also be used here instead spin-locking. Since the synchronization object requires no busy waiting, it is computationally more efficient. However, when a short wait is expected, the simplicity of the spin-locking is sometimes appropriate.

This scheme does not include a mechanism to determine when all element temperature changes are within a specified epsilon. There is a straightforward implementation using the combining tree from the set synchronization example. This tree does *not* serve to synchronize the iterations. Instead, it periodically (e.g., every J iterations) combines temperature changes from all elements to determine if all are within the specified epsilon. If so, the tree distributes a stop iterating message to all elements. The combining tree operates independently from the element iterations.

3.8 Summary

This chapter demonstrates how mechanisms from different computational models are represented in Pi. One of the interface evaluation metrics described in Chapter 1 is functionality match. The model mechanisms are directly implemented using Pi operations (i.e., there is a good functional match between Pi operations and the model requirements). Figure 3.18 shows the parallel issues are best demonstrated by the examples. It also shows model or models which are most similar.

In the next chapter, the effectiveness of these examples is considered. Their cost (in terms of basic parallel operations) is examined and their execution behavior is characterized.

experiment	parallel issues addressed			model(s)
Shared Memory	storage management	naming	synchronization	shared memory MIMD
			communication	
Set Synchronization		naming	synchronization	data parallelism
			locality	
	task management		communication	
Translation	storage management	naming		concurrent object oriented
			locality	
			communication	
Non-Resident Handlers	storage management	naming	synchronization	concurrent object oriented
	task management		communication	
N-Body	storage management	naming	synchronization	dataflow
				concurrent object oriented
	task management	sequential sequences	communication	
Relaxation	storage management	naming	synchronization	dataflow
			locality	data parallelism
	task management	sequential sequences	communication	

Figure 3.18: Model Mechanism Map

Chapter 4

An Evaluation of Pi

This chapter evaluates the model mechanisms and applications described in Chapter 3. Features of Pi that support the examples are discussed and some key code fragments are considered. Results collected during the execution experiments are also presented and evaluated. First, the evaluation metrics are defined.

4.1 Evaluation Metrics

One way to evaluate a mechanism implementation is by examining the Pi code. Pi is not an abstract programming language. In a Pi program, parallel operations are explicit. By inspecting the code that implements a mechanism, one can estimate its cost in terms of parallel operations (e.g., communications, synchronizations, etc.).

In this chapter, *operations* denote Pi operations such as **read**, **self**, and **attribute**. Usually, many operations are combined to form a single *logical instruction*. For example, the Pi code fragment:

```
(write (self) C (plus (read (self) A) (read (self) B)))
```

contains seven operations (one **write**, one **plus**, two **reads**, and three **selfs**). Yet this is considered a single instruction since adding two values is considered a single logical step. Logical instructions are a more meaningful metric when considering costs on a Pi substrate. They are used as the main unit of evaluation rather than Pi operations.

Other characteristics about the examples are gathered by executing them. PiSim is instrumented to collect several statistics:

Total Tasks Executed This is the total number of tasks executed during a trial. It gives a measure of the size of the experiment. Tasks are created by the **send**, **send-segment**, **call**, and **call-segment** operations.

Total Instructions Executed This is the total number of logical instructions executed during a trial. This is another measure of the experiment size.

Average Instructions/Task This is the average total number of logical instructions executed by each task (i.e., the total number executed independent of task suspensions). This metric measures task grain size.

Average Instruction Run Length This is the average number of logical instructions executed by a task between suspensions. For example, if a new task executes six instructions, suspends, then resumes and executes four more instructions before completing, its average instruction run length is five (ten instructions / two running periods). This metric measures execution grain size.

Average Operations/Instruction This is the average ratio of Pi operations to logical instructions. It measures the average complexity of the logical operations.

Total Read-Write Segments Created This is the total number of read-write segments created during a trial. This parameter includes both task segments resulting from calls and sends, and explicitly created data segments. This metric measures storage required by a trial.

Average Read-Write Segment Length This is the average size of a read-write segment allocated during the experiment. It includes both data-only and task segments. This metric measures the storage grain size.

PiSim also collects instruction type, operation type, segment size, segment age, and task run and wait time histograms. A detailed log of each experiment is provided in Appendix B.

4.2 Shared Memory with Caches

The definition and implementation of the shared memory protocol in Pi is similar to those proposed for specialized shared memory machines [2, 9, 11, 12, 22, 35, 36]. The sequences for supporting the protocol employ the same communication and synchronization operations.

When evaluating Pi's implementation, special issues that must be addressed include (a) critical sequences in the protocol where machine specialization can improve the performance (e.g., for cache read hits), and (b) protocol differences resulting from characteristics of machine hardware (e.g., message order preservation). This section discusses these issues.

If the cache hit rate is high, the overhead for accessing a cache hit is important for memory access performance. Figure 4.1 gives the code for a shared memory read. On a cache hit, the following logical instructions are executed: two **and**, one **match**, two **branch**, and one **read**. So, a total of six logical instructions are executed when a cached value is read.

Above the interface, this cost can be reduced to two logical instructions (one **match** and one **branch**) by adding each shared memory word to the cache separately. Larger

```
;; This handler executes a "read" code fragment.
(define-handler Read-Address (Address) (Base Offset Line)
  (write (self) Base (and (read (self) Address) Base-Mask))
  (write (self) Offset (and (read (self) Address) Offset-Mask))
  (write (self) Line (match (read (nodals) Cache-Map) (read (self) Base)))
  (branch-zero (compare (read (self) Line) UNBOUND) Get-Line)
  (branch-zero (compare (read (read (self) Line) Status-Offset) INVALID)
               Get-Line)
  (return (read (read (self) Line) (read (self) Offset)))
  (destroy-segment (self))
  Get-Line
  (write (self) Line (call 8 Get-Cache-Line (read (self) Base) READ))
  Read-Line
  (return (read (read (self) Line) (read (self) Offset)))
  (attribute (read (self) Line) Request-Offset READ-WRITE)
  (destroy-segment (self)))
```

Figure 4.1: Read Access Code Fragment

lines (e.g., eight words) can still be requested over the network to provide spatial locality.

Below the interface, the cost of supporting read hits can be reduced if *segment caches* are provided in the hardware substrate. Segment caches are fast, associatively addressed read-write segment buffers that provide very low latency accesses. The machine dependent compiler can transform the Pi code sequence into segment cache references that provide comparable access times to that of specialized shared memory machines. In addition, segment caches also improve the performance of other model mechanisms.

A Pi implementation of shared memory must deal with two issues not normally addressed in other implementations. First, since Pi does not preserve message order between nodes, special handling is required to guarantee consistency. Chunk lines must be locked for a longer period, and an extra acknowledgment must be sent from the cache to the chunk.

The second additional issue is related to chunk request wait time. Since Pi does not allow tasks to be interrupted when a message is received, a shared memory access request (e.g., a chunk request) must wait until the task executing on the node suspends or completes before it can be serviced. Prioritized task queues allow memory requests to execute ahead of application tasks. However, if the run length of the application task is long, the access request wait time can become unacceptably long. To prevent this, the machine compiler must insert voluntary suspend operations in long execution sequences to reduce the run length. If the suspension/ resumption overhead is low, this does not significantly affect execution performance.

A non-caching protocol that incorporates address braiding is also included in Appendix A. Braiding requires three logical instructions to decode (unbraid) the address,

total tasks executed	33,004
total instructions executed	329,359
average instructions/task	10.0
average instruction run length	8.1
average operations per instruction	4.6
total read-write segments created	33,770
average read-write segment length	6.9

Table 4.1: Shared Memory Statistics

but these operations are only executed on cache misses.

4.2.1 The Experiment

To demonstrate the shared memory example, several tests were conducted on a 4096 word shared memory[1]. The tests execute sequences of local and remote read accesses that exercise the protocol. For example, one test writes a location locally, then reads it back remotely to verify that its read-write cache copy was flushed properly. Another test writes and tests each memory location in the shared address space to evaluate the protocol in a high request traffic situation. The shared memory tests do not provide "typical" access patterns since they are primarily testing "miss" behavior.

Table 4.1 presents statistics from the experiment. This is the largest model mechanism example, in terms of number of instructions. Because some shared memory tests are very long, this example has the second largest instruction per task ratio. Yet in spite of this, the experiment has a run length of 8.1. Except for one atypical example, all experiments in this chapter have a run length less than ten. This demonstrates Pi ability to represent fine-grain tasks.

This example also allocated a large number of segments. Most are task segments created by **send** and **call** (33,004 of the 33,770 segments). Of 766 segments created explicitly, 637 were 12 word line objects (512 for chunk storage, 125 for cache). These segments represent only two percent of the segments created. However, since these segments are long-lived, whereas task segments are typically ephemeral, they constitute a significant component of the storage requirement for this example.

This example presents a shared memory scheme constructed with Pi. This implementation is non-optimal, but it demonstrates the necessary mechanisms. Future research will examine particular tradeoffs in the protocol specification using trace data from concurrent applications.

[1]This size was selected to allow experimentation on a sequential Pi simulator (PiSim). The overhead costs are the same for any shared memory size provided the addresses can be contained in a single word.

A shared memory location is more costly, in terms of storage and access time, than a normal segment location. This reflects the extra cost of maintaining global consistency. However, an efficient shared memory protocol implemented on an appropriate Pi substrate can compete with the performance of specialized shared memory architectures.

4.3 Set Synchronization

In Pi, a set synchronization delay depends on (a) the log of the number of elements in the set, (b) the distance between elements in the set, and (c) message traffic (network traffic and message queue lengths). A machine that is specialized to support set synchronizations can reduce the cost of these factors in several ways.

One approach is to distribute global synchronization information as a wired-or signal. The delay is still dependent on the separation of the set elements (here, the size of the machine), but the constant factor is smaller, since dedicated wires and control logic is used. But these resources are idle, unless a synchronization is taking place. Also, this approach can only support one global synchronization.

A more general approach is to dedicate special communication channels for synchronization information and add hardware to reducing the overhead of distributing communication information. This supports multiple set synchronizations with low overhead. But like the previous approach, the resources used to make set synchronization fast are taken away from other functions (especially data communication).

Pi uses the general communication mechanism **send**, to support set synchronizations. Special hardware is provided on each node to provide fast set synchronization tasks. This hardware also improves other features of Pi, making it generally more valuable.

The cost of performing a set synchronization in Pi can be computed from the code in Appendix A. If F is the fanout and N is the number of leaves, the cost for a single synchronization (not including tree initialization, which is amortized) can be determined.

The root does a send, adjusts a barrier, and waits for the FINISH signal to arrive. When it does, the barrier is adjusted, and the signal segment is destroyed. The total cost is one **send**, two **adjust-count**, one **test-count**, and one **destroy-segment**.

After a START signal is received (one s-sync **write** and one **destroy-segment**), the stem forwards F copies[2] of the START signal to its children (one initialization, one s-sync **read**, F **send**, **plus**, and **branch** instructions). It then waits for N FINISH signals before sending a FINISH signal to its parent (two **branch**, $F+1$ **adjust-count**, one **test-count**, F **destroy-segment**, and one **send** instructions).

Each leaf receives a START signal (one s-sync **write** and one **destroy-segment**), executes the task, sends a FINISH signal and waits for the next signal (one **send**, one **branch**, and one

[2]Depending on N and F, a few stems have less than F children. Therefore this is the worst case.

instruction	root	stem	leaf	F=5, N=100
test-count	1	1		26
adjust-count	2	$F+1$		152
s-sync read		1	1	125
s-sync write		1	1	125
send	1	$F+1$	1	251
initialize		1		25
plus		F		125
branch		$F+2$	1	275
destroy-segment	1	$F+1$	1	251

Table 4.2: Set Synchronization Costs

```
  Main-Loop
;; This is the section of iteration work that cannot be overlapped.
  (write (self) Index (minus (read (self) Index) 1))
  (branch-not-zero (read (self) Index) Main-Loop)
;; The finish signal is sent to the parent stem.
  (send (read (self) Up-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Up-Segment) DOWN-BARRIER-OFFSET)
;; This is the section of the iteration work that can be overlapped.
  (write (self) Index WORK-SIZE)
;; The leaf waits to begin the next iteration.
  (write (self) Temp (read (self) Up-Sync)) ;; wait for start signal
  (branch-not-zero (compare (read (self) Temp) END) Main-Loop)
```

Figure 4.2: Leaf Task Fragment

s-sync read).

The total iteration cost for each object is summarized in Table 4.2. For a given F and N, the cost of an iteration can be computed by multiplying the number of each type of object by the cost of that object and summing the result. For example, if $F = 5$ and $N = 100$, the synchronization tree contains one root, 25 stems, and 100 leaves. For this tree, the overhead cost of one synchronization is listed in Table 4.2.

Strict synchronization is not always necessary. By overlapping non-synchronized code during the signaling phase, the cost of synchronization can be reduced. For example, if a leaf task can do part of the iteration work (that can be overlapped) *after* sending the FINISH signal to the stem, the time of that work is subtracted from the synchronization cost (see Figure 4.2).

4.3.1 The Experiment

total tasks executed	25,579
total instructions executed	1,165,353
average instructions/task	45.6
average instruction run length	28.4
average operations per instruction	4.1
total read-write segments created	25,632
average read-write segment length	6.0

Table 4.3: Set Synchronization Statistics

The set simulation experiment was simulated with 100 tasks executing 100 iterations of simulated work task (here, the task is counting to 50). Table 4.3 summarizes the statistics.

This example has an abnormally high number of instructions, average instructions per task, and average instruction run length because of the simulated iteration task: a counting loop that executes ~100 logical instructions without suspension. In a real application, the task would probably suspend several times during an iteration because of communication with other leaves. If an application task could run unsuspended for an extended period, the compiler would insert voluntary suspends in the iteration sequence.

In this experiment, a scalable synchronization tree is implemented in Pi. The example demonstrates the use of synchronization trees. Future work will investigate compiler techniques to reduce set synchronization costs.

4.4 Object Name Translation

This example is a straightforward implementation of a distributed name resolution system. It uses associative segments in two capacities: as map storage and client caches. Yet these usages have significantly different requirements. Maps require safe storage of translations, and can grow to a relatively large size. Client caches need not be safe, since the translation can always be retrieved from the appropriate map. However, since clients reside on nodes with the application that uses the translation system, they must have fixed-sized storage. These particular requirements are indicated by the boundedness and safety parameters of the associative segment.

This example exploits locality using the Pi operation **distance**. When a new client is created, a prototype agent is initially supplied. Then the Closest-Agent handler, shown in the code fragment in Figure 4.3, is sent by the client to the prototype agent. This handler measures the distance between the new client and each agent, returning the closest agent.

This technique results from a compromise made in Pi's design. If the underling substrate topology was visible in Pi, this technique would not be necessary. Agents

```
;; This handler finds the closest agent to a client.  Each agent
;; compares its distance to the current distance.  If it is closer, it
;; updates the agent node and segment with it own value and sets the
;; new distance.  When the last agent makes the comparison, the
;; closest agent is returned to the client.
(define-handler Closest-Agent (Count This-Agent Client-Node Client-Segment
                               Distance Agent-Node Agent-Segment) ()
  (branch-zero (compare (read (self) Distance) -1) Skip-A)
  (branch-not-minus (compare (distance (read (self) Client-Node) (node-id))
                             (read (self) Distance)) Skip-B)
  Skip-A
  (write (self) Distance (distance (read (self) Client-Node) (node-id)))
  (write (self) Agent-Node (node-id))
  (write (self) Agent-Segment (read (self) This-Agent))
  Skip-B
  (write (self) Count (plus (read (self) Count) 1))
  (branch-zero (compare (read (self) Count) NUMBER-OF-AGENTS) Skip-C)
  (send (read (read (self) This-Agent) AGENT-NEXT-NODE)
        10 NORMAL Closest-Agent (read (self) Count)
        (read (read (self) This-Agent) AGENT-NEXT-SEGMENT)
        (read (self) Client-Node) (read (self) Client-Segment)
        (read (self) Distance) (read (self) Agent-Node)
        (read (self) Agent-Segment))
  (branch-zero 0 Skip-D)
  Skip-C
  (send (read (self) Client-Node) 6 NORMAL Reply-Value
        (read (self) Agent-Node) (read (self) Client-Segment)
        CLIENT-AGENT-NODE)
  (send (read (self) Client-Node) 6 NORMAL Reply-Value
        (read (self) Agent-Segment) (read (self) Client-Segment)
        CLIENT-AGENT-SEGMENT)
  Skip-D
  (destroy-segment (self)))
```

Figure 4.3: Closest Agent Fragment

total tasks executed	2,649
total instructions executed	14,049
average instructions/task	5.3
average instruction run length	4.5
average operations per instruction	6.2
total read-write segments created	2,786
average read-write segment length	8.1

Table 4.4: Translation Statistics

could be distributed uniformly across the machine at known locations, so a new agent could directly access the nearest one. But this would commit the implementation to a specific topology. Using the more general technique in this example, the closest agent is obtained (with some computation cost) regardless of the topology.

The cost of translation is computed from the Match, Agent-Match, and Match-In-Map handlers in Appendix A. If a match hits in the client translation cache, the cost is one **match** and one **branch** operation. If the translation misses in the cache, the translation must be fetched from the appropriate map via an agent. This requires two **match**, one **branch**, three **send**, one **insert**, three **destroy-segment**, one d-sync **attribute**, one d-sync **read**, and one **mod** instructions.

4.4.1 The Experiment

This experiment was executed with ten maps, 25 agents, and 25 clients. The test program injects 100 translations into the system via different clients. Then it reads the translations backs and tests the results. It then rereads the results to test the client cache.

Table 4.4 summarizes the statistics. This is the smallest example, in terms of instructions. It also sets the lower bound for task size and run length. A Pi substrate must have low enough task management overhead to support an execution sequence of 4.4 logical instructions.

This translation system demonstrates one straightforward example of a distributed service. It is efficiently implemented (typically with two logical instructions, 13 instructions worst case) using short tasks and local communication. This translation system is used by the next mechanism example.

```
;; When a non-resident handler is invoked, this handler is called as
;; the normal handler type.  It then uses the first argument as the
;; non-resident handler type.  It tests if the handler is present on
;; the node.  If not, it obtains it.  Then it invokes it with the
;; remaining arguments.  A special argument segment must be created.
;; Since the locals for a non-resident handler are included in this
;; space, care must be taken to only copy parameters into it.
(define-handler Dispatch (Name :VARIABLE) (Handler Message-Length Arguments
                                           Index-A Index-B)
  (write (self) Handler (match (read (nodals) Current-Handlers)
                               (read (self) Name)))
  (branch-not-zero (compare (read (self) Handler) UNBOUND) Handler-Known)
  (call 6 Lookup-Handler (read (self) Name))
  (write (self) Handler (match (read (nodals) Current-Handlers)
                               (read (self) Name)))
  Handler-Known
  (write (self) Message-Length (minus (read (self) LENGTH) 6))
  (write (self) Arguments
         (create-read-write-segment (minus (read (self) LENGTH) 10)))
  (write (self) Index-A 9)
  (write (self) Index-B 0)
  Loop
  (branch-not-zero (probe (self) (read (self) Index-A) READ) Skip)
  (write (read (self) Arguments) (read (self) Index-B)
         (read (self) (read (self) Index-A)))
  Skip
  (write (self) Index-A (plus (read (self) Index-A) 1))
  (write (self) Index-B (plus (read (self) Index-B) 1))
  (branch-minus (compare (read (self) Index-A) (read (self) LENGTH)) Loop)
  (call-segment (read (self) Message-Length) (read (self) Handler)
                (read (self) Arguments))
  (destroy-segment (read (self) Arguments))
  (destroy-segment (self)))
```

Figure 4.4: Dispatch Fragment

4.5 Non-Resident Handlers

Non-Resident handlers employ several features of Pi including variable argument messages and non-resident handler calling. A slightly modified version[3] of the dispatch handler is shown in Figure 4.4.

This is one example where the segment passed to the call-segment operation can be eliminated by the machine dependent compiler. A selected block of the active segment is copied to a new segment to specify that a variable number of arguments are to be passed. The compiler can figure out that the segment is created only for use by the

[3]PiSim does not support direct dispatches on segments (since handlers are not stored as segments). Therefore, in the implemented example, the code segment contains the name of the handler being executed. See Appendix A for the actual code.

total tasks executed	3,021
total instructions executed	23,915
average instructions/task	7.9
average instruction run length	7.1
average operations per instruction	5.7
total read-write segments created	3,683
average read-write segment length	8.4

Table 4.5: Non-Resident Handler Statistics

call-segment operation. Therefore, it can call the message directly from the active segment, avoiding the unnecessary work.

The cost of dispatching a non-resident handler (which is present in the node's handler cache) can be computed from the code of the dispatch handler. Besides argument copying (which may be avoided as explained above), the cost is one **match**, one **branch**, one **call-segment**, and one **destroy-segment** instructions.

4.5.1 The Experiment

This experiment was executed using a test non-resident handler that prints the active segment (the printing was disabled for the log output in Appendix B). After the handler is created in the reference map and added to the translation system, it is executed twice on each node in the system. The first trial tests the handler fetching mechanism. The second trial tests the handler caching mechanism.

Table 4.5 summarizes the statistics from the experiment. Only 128 test handlers are dispatched. Most of the statistics in this table reflect the translation system initialization. In the trial, a **times** instruction was used in place of the print statement to shorten the (rather dull) log. The instruction profile in Appendix B shows that 128 **times** were executed. The task type profiles shows that handler lookup system (e.g., match-agent) was only executed 64 times. Therefore, the local handler caches worked properly.

This example demonstrates the construction of system code in Pi. It efficiently supports (with only four additional logical instructions) a dispatch mechanism that differs from the resident handler dispatch mechanism provided by Pi.

4.6 N-Body Simulation

In this example, and in the relaxation example in the next section, an entire application is coded in Pi. They employ several mechanisms from different parallel models

```
Loop-A
(write (self) AX 0)
(write (self) AY 0)
(adjust-count (self) Barrier (read (self) Count))
(write (self) Index 0)
Loop-B
(send (read (read (self) I-Nodes) (read (self) Index))
      7 NORMAL Update-Position
      (read (self) X) (read (self) Y) (read (self) ID)
      (read (read (self) I-Segments) (read (self) Index)))
(write (self) Index (plus (read (self) Index) 1))
(branch-not-zero (compare (read (self) Index) (read (self) Count)) Loop-B)
(test-count (self) Barrier)
(write (self) X (plus (read (self) X) (read (self) VX)))
(write (self) Y (plus (read (self) Y) (read (self) VY)))
(write (self) VX (plus (read (self) VX) (read (self) AX)))
(write (self) VY (plus (read (self) VY) (read (self) AY)))
(write (self) Tick (plus (read (self) Tick) 1))
(branch-not-zero (compare (read (self) Tick) (read (self) Last-Tick))
                 Loop-A)
```

Figure 4.5: Body Inner Loop Fragment

(specifically dataflow and concurrent object oriented). This n-body example employs most of the parallel programming features of Pi. By examining the code that implements the application, one can directly observe the cost of the implementation, in terms of basic mechanisms.

For example, Figure 4.5 shows the inner loop of the body object handler. If there are N bodies, each body executes $N+1$ **adjust-count**, one **test-count**, $N-1$ **send**, $N+5$ **plus**, three initialization, $N+1$ branches, and $N-1$ **destroy-segment** instructions.

4.6.1 The Experiment

total tasks executed	213,051
total instructions executed	2,243,745
average instructions/task	10.5
average instruction run length	7.9
average operations per instruction	6.3
total read-write segments created	213,253
average read-write segment length	7.1

Table 4.6: N-Body Statistics (100 Bodies, 10 Iterations)

total tasks executed	180,156
total instructions executed	2,028,903
average instructions/task	11.3
average instruction run length	8.4
average operations per instruction	6.3
total read-write segments created	180,178
average read-write segment length	6.5

Table 4.7: N-Body Statistics (10 Bodies, 1000 Iterations)

Two n-body experiments were executed. The first included 100 bodies simulated for ten time steps (Table 4.6). The second experiment included 10 bodies simulated for 1000 iterations (Table 4.7). The operations that print the body positions were disabled for the log in Appendix B.

Note the short tasks and run lengths. The task lengths are short because of the high percentage of communication-only tasks (93% of the 100 body trial, 99.9% of the 10 body trial). The run length (and task length) of the communication-only tasks is four logical instructions. The run length of body and interaction tasks is longer, but still kept short by synchronization suspends.

This example shows the low overhead of supporting an application. In the 10 body example, 24.4% of the logical instruction types were **times**, used in the interaction object handler to compute the interaction force.

4.7 Relaxation

This application is implemented using only four handlers types.

Using the Pi code for this example, the "overhead" of supporting the algorithm can be computed. The cost of the inner loop of the relaxation (for a non-boundary element) is four **send**, four **match**, four **destroy-segment**, nine **branch**, one initialize, four s-sync **write**, four s-sync **read**, four **plus**, and one **divide** instructions.

This cost could be further reduced by handling boundary cases specially and using a more sophisticated initialization scheme. With these improvements, the inner loop would contain ten logical instructions, five of which are the arithmetic computation.

4.7.1 The Experiment

In this experiment, a 100 x 100 element plate was simulated for 100 time steps. Table 4.8 summarizes the statistics. These results are similar to those of the n-body trials. These larger examples tend to be dominated by communication messages.

total tasks executed	36,201
total instructions executed	305,873
average instructions/task	8.4
average instruction run length	5.2
average operations per instruction	5.3
total read-write segments created	36,266
average read-write segment length	7.0

Table 4.8: Relaxation Statistics

Relax	1	< 0.1%
Reply-Value	100	0.3%
Start-Element	100	0.3%
Update-Temp	36,000	99.4%
total	36,201	100.0%

Table 4.9: Relaxation Task Type Profile

Table 4.9 shows the task type profile for the experiment. Note that over 99% of the messages are the communication/synchronization message type Update-Temp. This means that the overhead for these short, ephemeral tasks play a key role in the performance of this application.

4.8 Summary

This section presents composite statistics for all experiments described in this thesis.

4.8.1 Instructions

Table 4.10 lists the composite dynamic logical instruction profile for all the experiments. Logical instructions that occurred less than .1 percent are excluded. Note that over eighty percent of the executed logical instructions are basic sequential operations. This shows the importance of efficient sequential support for this small set of examples. When designing a parallel architecture, sequential performance must not be overlooked.

Communication (7.8%), storage management (8.4%), synchronization (5.1%), and associative segment (.9%) instructions have a significant presence in the profile. If these functions are not efficiently supported in the machine architecture, overall performance could be jeopardized. Parallel machines based on conventional micropro-

instruction type	number	percentage
branch-not-zero	1,129,039	17.8%
times	1,066,004	16.8%
plus	916,048	14.4%
minus	819,268	12.9%
move	552,919	8.7%
destroy-segment	534,011	8.4%
send	492,619	7.8%
adjust-count	237,683	3.7%
exponent	94,500	1.5%
call	89,222	1.4%
branch-zero	76,567	1.2%
initialize	65,745	1.0%
and	55,334	0.9%
match	50,326	0.8%
test-count	48,120	0.8%
attribute	39,378	0.6%
a-shift	29,144	0.5%
return	14,933	0.2%
branch-minus	11,160	0.2%
divide	10,000	0.2%
or	8,730	0.1%
insert	5,670	0.1%
total	6,355,758	100.0%

Table 4.10: Dynamic Instruction Profile Summary

operation type	number	percentage
self	14,267,394	39.4%
read	11,967,178	33.1%
write	3,540,240	9.8%
branch-not-zero	1,129,039	3.1%
times	1,066,415	2.9%
plus	902,473	2.5%
minus	701,767	1.9%
compare	667,264	1.8%
destroy-segment	534,011	1.5%
send	488,793	1.4%
adjust-count	237,683	0.7%
exponent	94,500	0.3%
call	89,224	0.2%
branch-zero	76,564	0.2%
nodals	72,608	0.2%
and	55,334	0.2%
match	50,538	0.1%
test-count	48,120	0.1%
attribute	39,378	0.1%
mod	35,449	0.1%
a-shift	29,144	0.1%
node-id	25,372	0.1%
total	36,181,081	100.0%

Table 4.11: Dynamic Operation Profile Summary

cessors are vulnerable unless the deficiencies can be overcome with external circuitry. A Pi substrate must efficiently support these operations.

4.8.2 Operations

Table 4.11 lists the composite operation profile for all the experiments. Operations that occurred less than .1 percent are excluded. Since references to the active segment are explicit in Pi programs, the leading operation is **self**. It is used with nearly every logical instruction. The machine dependent compiler will reduce references to the active segment by using registers for temporaries. Yet the expected frequency is high enough to justify special handling of the active task segment in a Pi substrate.

In Pi programs, all **read** and **write** operations reference a segment. While the number of segment accesses would also be reduced by register usage by the compiler, the number is still large enough to require efficient support for offset addressing and bounds checking.

size	number	percentage	total storage	percentage
1	65	< .1%	65	< .1%
2	64	< .1%	128	< .1%
3	156	< .1%	468	< .1%
4	16,567	3.1%	66,268	1.8%
5	2,426	0.5%	12,130	0.3%
6	228,666	42.7%	1,371,996	37.1%
7	248,829	46.5%	1,741,803	47.1%
8	16,136	3.0%	129,088	3.5%
9	11,406	2.1%	102,654	2.8%
10	2,570	0.5%	25,700	0.7%
11	2	< .1%	22	< .1%
12	639	0.1%	7,668	0.2%
13	778	0.1%	10,114	0.3%
14	154	< .1%	2,156	0.1%
16	1,624	0.3%	25,984	0.7%
19	110	< .1%	2,090	0.1%
21	100	< .1%	2,100	0.1%
23	4,995	0.9%	114,885	3.1%
25	22	< .1%	550	< .1%
99	200	< .1%	19,800	0.5%
100	3	< .1%	300	< .1%
1024	64	< .1%	65,536	1.8%
total	535,576	100.0%	3,701,505	100.0%

Table 4.12: Read-Write Segment Size Profile

The **exponent** operation has a significant presence in both the instruction and operation profiles. As explained in Chapter 3, this is not really a Pi operation. It would be supported by a sequence of other Pi operations. Since this sequence would contain mostly arithmetic instructions, these operations are underrepresented in the profiles.

4.8.3 Segments

Table 4.12 lists the composite read-write segment size profile for all the experiments. The average segment size is 6.9. Note that 95.8% of the segments would fit in an eight word block. 99.0% would fit in a sixteen word block.

Another interesting parameter is segment lifetime. This is the length of time from when a segment is created to when a segment is destroyed. Segment lifetimes are measured in Pi operation executions. Table 4.13 shows the collected data. Note that profile bins where used to collect the data. Each entry indicates the number of segments that had a lifetime falling within the range of that bin.

low	high	number	percent
9	16	46,519	8.7
17	32	162,046	30.3
33	64	34,585	6.5
65	128	59,312	11.1
129	256	45,513	8.5
257	512	33,103	6.2
513	1,024	15,813	3.0
1,025	2,048	40,428	7.6
2,049	4,096	50,181	9.4
4,097	8,192	34,112	6.4
8,193	16,384	6,496	1.2
16,385	32,768	28	0.0
32,769	65,536	176	0.0
65,537	131,072	132	0.0
131,073	262,144	60	0.0
262,145	524,288	5,051	0.9
524,289	1,048,576	454	0.1
1,048,577	2,097,152	2	0.0
total		534,011	100.0

Table 4.13: Segment Age Profile

task status	number	percentage
NEW	488,865	63.9%
WAITING	231,992	30.3%
CALL	44,740	5.8%
total	765,597	100.0%

Table 4.14: Task Status Profile

The average segment lifetime is 5490 operations. The median segment lifetime is 96 operations. This discrepancy is cause by a large number of short-lived segments and a small number of very long-lived segments. The short-lived segments are primarily communication-only task segments. For example, in the relaxation example, Update-Temp represented 99.4% of the tasks (and task segments) executed in the trial. Yet this task typically executed only four logical instructions (20 Pi operations).

From these experiments, the typical segment can be characterized as small (~7 words) and ephemeral (~100 Pi operations). A Pi substrate must provide extremely fast allocation and deallocation for small segments. Task overhead is closely dependent on segment management efficiency.

4.8.4 Tasks

Table 4.14 lists the composite task status profile for the experiments. NEW is the number of new tasks invoked in all examples. WAITING is the number of task suspensions and resumptions. CALL is the number of called tasks. On the average, a task suspends 0.5 times during its execution. This average results from a few tasks that suspend several times and many tasks that never suspend.

During the experiments, the total amount of time each task spent running and waiting was recorded. The running time includes the sum of all time when the task is actively executing on a node. The waiting time includes time spent in the message queue waiting to execute, as well as waiting time between suspensions. Table 4.15 presents the task run time profile. Note that 93.5% of the tasks executed under 32 Pi operations. 99.0% executed under 128 operations. The average for all experiments is 74.3 operations.

The task wait time profile is shown in Figure 4.16. The average task wait time is 4980 Pi operations. The median task wait time is 48 operations. Note that 38.8% of the tasks executed with zero wait time. These tasks (likely communication-only) arrived, executed immediately, and completed without suspending.

Task wait time is dependent on the ratio of the average number of tasks executing on each node. In these experiments, 64 nodes were simulated. Some experiments, like relaxation, had a small number of tasks per node (typically 7.8 tasks/nodes), while others, like 100-body, had a higher number (~150 tasks/node). As a result, the wait time because of long message queues is much higher for the 100-body experiment. To

low	high	number	percent
9	16	62,070	11.6
17	32	437,094	81.9
33	64	23,290	4.4
65	128	5,590	1.0
129	256	150	0.0
513	1,024	3	0.0
1,025	2,048	5,016	0.9
2,049	4,096	5	0.0
4,097	8,192	24	0.0
8,193	16,384	105	0.0
16,385	32,768	100	0.0
32,769	65,536	100	0.0
65,537	131,072	39	0.0
131,073	262,144	7	0.0
262,145	524,288	12	0.0
total		533,605	100.0

Table 4.15: Task Run Time Profile

observe the this effect, the 100-body experiment was executed with 4096 simulated nodes. The median task wait time dropped from 1536 Pi operations (on 64 nodes) to one Pi operation (on 4096 nodes).

Table 4.17 shows the composite statistics for all the experiments described in this thesis. The instructions/task value (11.9) indicates that the task size of these mechanism and applications is small. The run length of 8.8 logical instructions suggests that the lack of a task preemption facility in Pi does not have a major effect on task waiting time.

In these examples, it is possible to determine the exact costs of mechanisms in terms of parallel operations. Related implementations in other architectures do not explicitly provide this measurement because the mechanisms are inseparable from the machine architecture. The ability to make mechanism comparisons in terms of basic parallel operations is a major feature of Pi.

While these examples are not a typical cross section of Pi programs, they provide rough data for the specification of a Pi substrate. The characteristics of these examples are used to drive the design of a Pi substrate in the next chapter.

low	high	number	percent
0	0	206,830	38.8
1	1	531	0.1
2	2	2,856	0.5
3	4	1,297	0.2
5	8	6,788	1.3
9	16	7,167	1.3
17	32	11,414	2.1
33	64	36,609	6.9
65	128	40,446	7.6
129	256	43,064	8.1
257	512	24,686	4.6
513	1,024	15,880	3.0
1,025	2,048	40,453	7.6
2,049	4,096	49,707	9.3
4,097	8,192	33,823	6.3
8,193	16,384	6,428	1.2
16,385	32,768	128	0.0
32,769	65,536	75	0.0
65,537	131,072	132	0.0
131,073	262,144	157	0.0
262,145	524,288	5,006	0.9
524,289	1,048,576	128	0.0
total		533,605	100.0

Table 4.16: Task Wait Time Profile

total tasks executed	720,857
total instructions executed	6,355,758
average instructions/task	11.9
average instruction run length	8.8
average operations per instruction	5.7
total read-write segments created	535,576
average read-write segment length	6.9

Table 4.17: Composite Example Statistics

Chapter 5

A Pi Substrate

Chapter 3 dealt with issues *above* the interface, namely the construction of mechanisms for different computational models. This chapter considers issues *below* the interface, the machine substrate that supports Pi.

Pi is supported by hardware in combination with a machine dependent compiler. In this chapter, the machine architecture is examined. The machine dependent compiler is considered in Chapter 4. The design of a Pi substrate is examined only as far as necessary to demonstrate that it can be built. It is left for future research (and funding!) to complete the engineering and construction of the substrate.

This chapter includes the definition of a machine architecture, PiMac, plus a discussion of an implementation. To avoid confusion, here are the definitions of "machines" described in this thesis.

Pi Abstract Machine This is the architecture of the Pi interface. A machine definition is used as a medium for defining the interface. However, a particular approach to supporting the interface is not specified.

PiMac Machine Architecture This a specific machine architecture design to support Pi. Its definition includes details, such as microarchitecture and instruction set architecture. A specific hardware implementation is not specified.

PiMac 1: a Gate Array PiMac Implementation This is a design for an implementation of the PiMac machine architecture. It uses a specific technology and incorporates a certain set of implementation goals and restrictions.

This chapter begins the design goals of a Pi substrate. It then describes the PiMac machine architecture. Finally, a gate array implementation[1] is considered.

[1]For a more detailed discussion of the differences between machine architecture and machine implementation, see [47].

word size	48 bits
maximum storage per node	65,536 words
minimum segment size	8 words
maximum segment size	4096 words
task limit (per node)	255 tasks
segment limit (per node)	1,023 segments

Table 5.1: PiMac Machine Details

5.1 Design Goals

Chapter 1 introduces a list of parallel model requirements which are the basis of the architectural interface. The support of these requirements underlies the design goals stated here. The detailed goals driving the design of a Pi machine include:

- low latency communication
- fast task switching (suspensions, queuing, and resumptions)
- multiple task/message queues
- support for attributes (synchronization)
- efficient storage management
- support for read/write and associative segments
- fast sequential execution

The design of PiMac aspires to one additional goal that is not required by Pi. PiMac is fine-grained; both in terms of physical nodes size, and the task size which it can efficiently support. A coarse-grain Pi machine is also feasible. In fact, a coarse-grain system can tolerate higher task overhead, since fewer tasks are executed overall. Unfortunately, coarse-grain systems have less potential for parallelism as well.

5.2 PiMac: A Pi Machine Architecture

PiMac is a parallel machine designed to meet the requirements of the Pi architectural interface. Table 5.1 summarizes some details of the machine. These parameters are selected based on (a) recent results from fine-grain message-passing research [27, 28] and (b) from the examples in Chapter 3. This machine architecture incorporates many ideas from the Message Driven Processor [14, 15]. The design of other parallel machines has influenced PiMac [53, 7, 51, 31, 56, 4, 43, 8, 45, 13, 30].

The memory limit is small compared with microprocessors with comparable sequential performance. The Amdahl/Case rule [25] suggests one megabyte of memory per MIPS of processor performance. By this rule, a PiMac node with 65 thousand words (256K bytes) is balanced with .25 MIPS processor implementation. Yet a significantly faster implementation is expected (~25 MIPS).

The Amdahl/Case rule is used to guarantee high processor utilization. It evolved when processor hardware was expensive relative to memory. A low processor utilization ($< 10\%$), was considered excessively wasteful. Now, processor logic consumes a much smaller fraction of the available resources (silicon area). Under this circumstance, excessive concern about processor utilization is inappropriate.

The Amdahl/Case rule also suggests one megabit of I/O bandwidth per MIP of processor performance. For a 25 MIPS PiMac, this rule would expect 25 megabits per second of I/O bandwidth. The router used in PiMac provides an estimated 540 megabits per second. This reflects the increased requirement for communication in a fine-grain multiprocessor.

PiMac's memory and I/O parameters reflect the fine-grain nature of the machine. Results by Horwat [28] suggest that these parameters are adequate for fine-grain tasks. This choice of memory size is also supported by the examples in Chapter 3, where the average segment size is ~7 words. However, the issue of small memory nodes is an open research question.

5.2.1 Microarchitecture

The PiMac microarchitecture is illustrated in Figure 5.1. It is composed of six modules.

Router

PiMac nodes are connected in a three dimensional mesh network. The network transports messages as chains of 18 bit *flits*[2] [17, 54] which are directed using a non-adaptive routing algorithm. The network employs wormhole routing to reduce latency. The router module is one node's contribution to the composite system network. Unless a message is being sent or received by a node, the message router operates independently from the rest of the node. The specifications of the router module are taken directly from the MDP [18, 54].

Network Controller

The network controller transfers messages from the network to local memory. It also inserts messages from the datapath into the network. When a message arrives at

[2] A flit is the smallest piece of data that is arbitrated in the network.

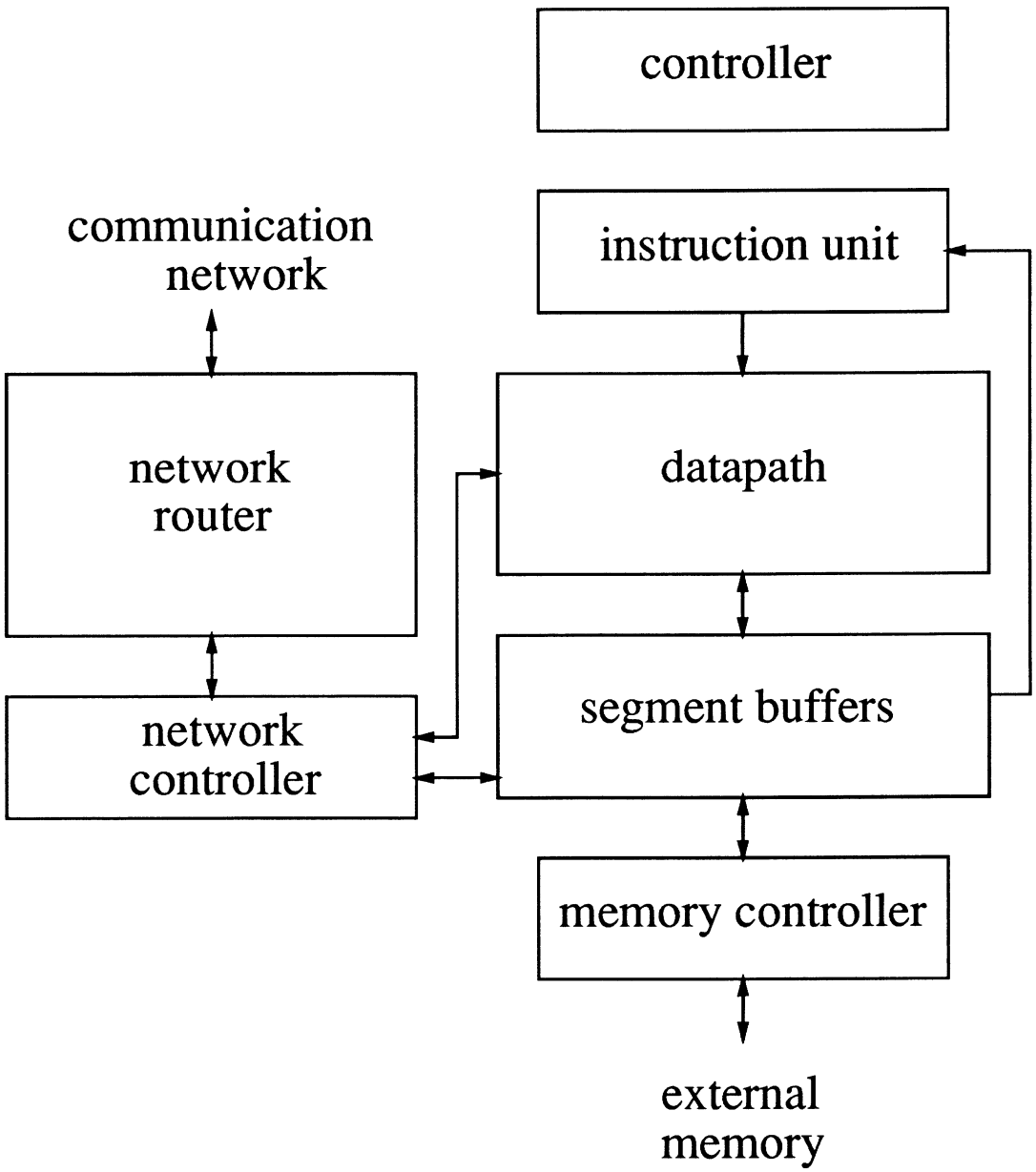


Figure 5.1: PiMac Microarchitecture

a node, the network controller allocates an appropriate sized storage block in local memory. The controller then receives and stores the message in memory. Finally, it adds the message to a task queue for later execution.

Datapath

The datapath module is composed of two separate datapaths. The first is the traditional integer datapath containing a few data registers and a general purpose ALU. This datapath is designed to provide fast context switching, and efficient support for several Pi operations.

The second datapath supports the specialized requirements of addresses. This datapath performs operations to support segment addressing, including bounds checking and offset accessing.

Segment Buffers

These buffers are positioned between the datapath and the memory. They act as a cache for recently accessed memory segments. They also provide hardware support for Pi synchronizations. One segment buffer is also used for buffering incoming messages received from the network controller.

Memory Controller

The memory controller supports memory transfers between the buffers and internal and/or external memory. In some PiMac implementations, memory caches are also provided in this module.

Instruction Unit

This module fetches and decodes instructions for execution. It executes in parallel with the datapath to provide one instruction per clock for most non-branching instructions. A special segment buffer caches several instructions from memory.

Controller

This module coordinates most of the activity in the microarchitecture. An exception is the network controller and router which operate autonomously. The network controller bridges the two "machines".

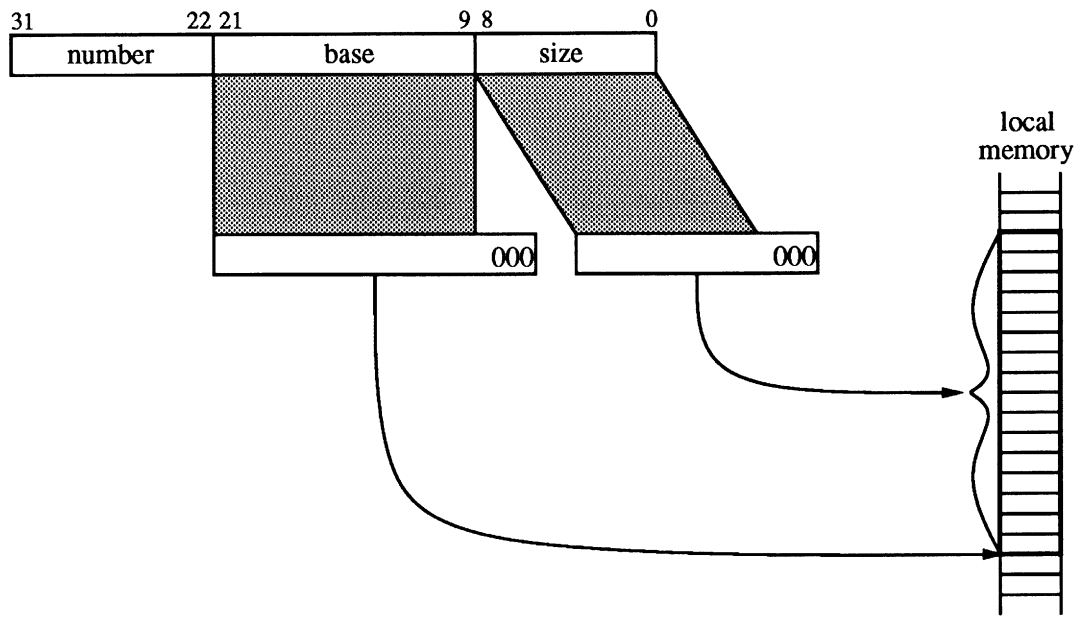


Figure 5.2: A Segment Descriptor

5.2.2 Segments

The node memory space is addressed in sixteen bits. PiMac allows linear addressing of memory. However, the expected mode of node memory access is via segments. A segment can reside anywhere in the address space, but must be aligned on an eight word boundary. An individual segment can be up to 4096 words long. Segment sizes are integer multiples of eight words. Therefore a segment base pointer is 16 - 3 = 13 bits long and a segment size is 12 - 3 = 9 bits. A segment pointer includes both the base and size specifiers, so segment accesses are bounds checked. This is illustrated in Figure 5.2.

When a segment size field is zero, bounds checking is disabled. A zeroed segment pointer is used for non-segment (linear) addressing of memory. The offset into this segment is an absolute memory address.

To minimize segment allocation time, a list of free segments is maintained by a free segment list which contains a linked list of fixed sized[3] free segments.

Usually, the processor or network controller can quickly access an adequate sized segment from this list. If the length field is larger than the fixed size of segments on the free list, or if the free list is empty, a trap is generated, causing the node processor to execute a special handler to allocate the required segment or segments.

Each segment is assigned a ten bit segment ID which provides a logical name for it on the node. This name is provided externally to identify the segment. A *segment map* maintains the base and size specifiers for each segment.

[3]The optimal size will be determined in future parallel operating system experiments. Based on the examples in Chapter 3, an expected size is 24.

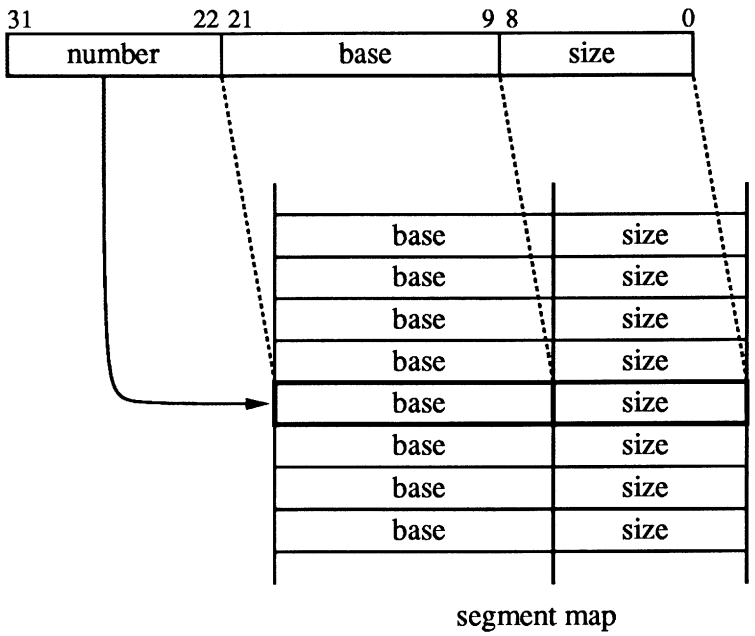


Figure 5.3: Indexing into the Segment Map

Segment ID allocation is usually combined with storage allocation. When a segment's storage is freed, the segment ID is reclaimed with it. The segment storage and ID are placed on the free segment list together. When a segment is allocated from the free list, it is assigned the same segment ID. Since most segments are short-lived, this is the usual technique for segment ID management.

Two cases require more complicated segment ID allocation technique. If the free segment list is empty, or if the fixed segment size of the free list is not large enough to accommodate the required storage. In these cases, the segment map is scanned for unused segment IDs.

A complete segment descriptor includes the segment ID, base pointer, and size descriptor. If segments are relocated (e.g., during heap compaction), the segment bases of all segment descriptors are invalidated. When the segment descriptor is referenced again, the segment ID is used to access the segment map to obtain the segment base and size specifiers. This process is illustrated in Figure 5.3.

5.2.3 Word Format

A named word in PiMac is composed of five fields:

Type (3)	Access Pattern (3)	Access Locks (2)	Suspended Tasks (8)	Value (32)

The Type, Access Pattern, Access Locks, and Suspended Tasks fields are collectively called the *tag* of the value.

type	description	format
0	integer	two's compliment value
1	symbol	value (0 = NIL)
2	boolean	one bit boolean
3	segment descriptor	(segment number, base, size)
4	instruction	instruction value
5	user defined type A	
6	user defined type B	
7	user defined type C	

Table 5.2: Word Types

code	name	description
0	D-Sync	d-sync access pattern
1	S-Sync	s-sync access pattern
2	B-Sync	b-sync access pattern
3	Write Once	write once access pattern
4	Read Only	read only access pattern
5	Write Only	write only access pattern

Table 5.3: Access Pattern Types

Type

This field determines the type of data stored in the value field. The types are defined in Table 5.2. Arithmetic operations on user defined types result in user trap handlers. User defined types are used to support special types such as floating point numbers and complex numbers.

Access Pattern

This field defines the access pattern for this word of memory. Used with the access lock bits, it determines what accesses are permissible, and how access locks bits should be changed by an access. The access patterns are defined in Table 5.3.

Access Locks

Each named word include two access locks: read and write. On each access to the word, the corresponding access bit is tested. If the access is locked, the operation is canceled and a microsequence is initiated which suspends the current task.

If an access is not locked, the access completes, and the lock bits are modified in accordance with access pattern defined in the access pattern field. For some access patterns, a particular access results in the requeuing of tasks contained in the suspended task list.

Suspended Tasks

This field points to the head of a list of tasks suspended on this word. If an access of this word causes the current task to be suspended, it is pushed on the head of this list. If an access of this word causes requeuing of the suspended tasks, each task is added to the appropriate task queue. Tasks in the suspended task list are linked together by *task links*. A task's link is maintained by the suspended tasks field of first word of the task segment.

Value

The remaining field of a word is the value field. It contains a 32 bit value.

5.2.4 Register Architecture

The PiMac register architecture is illustrated in Figure 5.4. This section describes each component.

Active Segment

At any given moment, a PiMac node is either executing a task or waiting for a message (idle). If a task is executing, the active segment is defined in the active segment register. This register includes a pointer to the physical location of the segment in local memory. This is used in several addressing modes of the instruction set architecture. It also maintains the *task number*, a logical name of the task used in both task management and synchronization support in PiMac. The value of the active segment register is set automatically when a task begins execution.

Code Segment

Each task segment includes a reference to a segment which contains the code for the handler being executed. This handler is also set when a task begins execution.

Instruction Pointer

This register indexes the current instruction in the code segment. It is analogous to a program counter in a microprocessor.

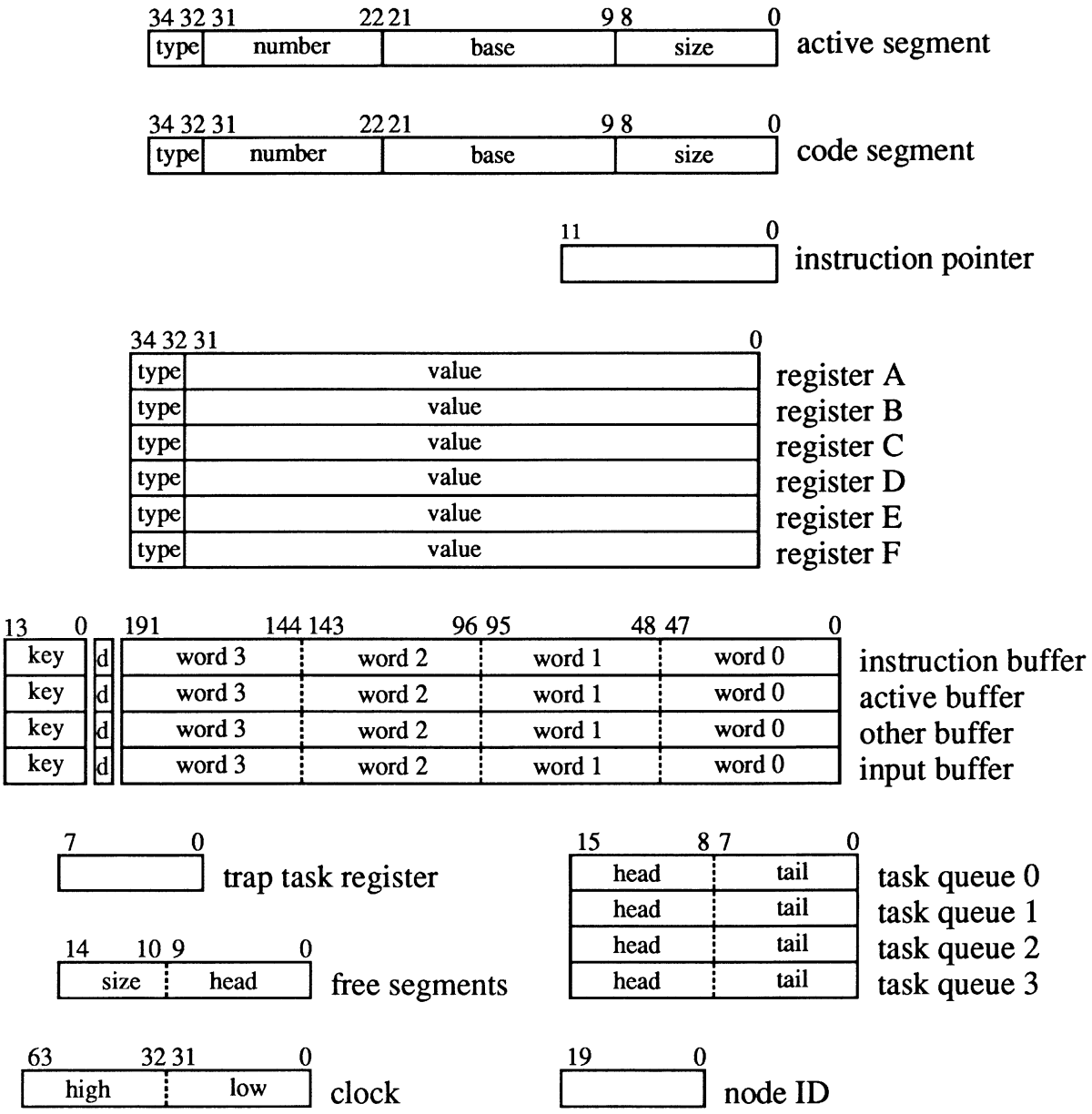


Figure 5.4: PiMac Register Architecture

General Registers

Six general purpose registers are provided in the register architecture. They are used to supplement storage in the active segment and reduce swapping of the segment buffers.

Segment Buffers

Four segment buffers provide dedicated caches for node memory. Each buffer contains a quadword[4] of a segment. Each buffer maintains a key referencing its physical address in memory. Each buffer also includes a dirty bit which indicates whether the buffer must be written back to memory when it is replaced.

One buffer is dedicated for the code segment of the active task. Another always includes a quadword of the active segment. The third buffer can include a quadword from any segment addressed by a segment-accessing instruction. The last buffer is used to buffer incoming messages to the node. Outgoing messages are sent directly, without buffering.

Task/Message Queues

These registers maintain the head and tail of four task queues (one for each priority). When a message is received from the network, it is enqueued at the tail of the appropriate task/message queue by the network controller. When a task is requeued after a suspension, it is inserted at the tail of a priority queue. The particular queue is determined by the priority field of the task.

These registers are shared by the network controller and the processor. Accesses to them, and node memory, are arbitrated by hardware to guarantee mutual exclusion.

Trap Task Register

This register maintains the ID of the task which was executing when the trap occurred. Since traps are disabled during trap handling, this register is never inadvertently overwritten.

Free Segments

This register maintains the head of the free segment list. It also contains the fixed segment size contained in the list. The fixed size is a multiple of eight words, so the five bits of the size field are the most significant of a eight bit size field.

[4]Quadwords are aligned on four word boundaries.

state	size	saved on switch?	resumed?
active segment pointer	one word	no	no[5]
code segment pointer	one word	no	no[5]
instruction pointer	12 bits	yes	yes
general registers	six words	yes	yes
instruction buffer	quadword	no	yes
active segment buffer	quadword	sometimes	sometimes
other segment buffer	quadword	sometimes	sometimes
message input buffer	quadword	not affected	not affected
task queues	two words	not affected	not affected
free segments pointers	15 bits	not affected	not affected
clock	two words	not affected	not affected
node ID	20 bits	not affected	not affected
trap task pointer	eight bits	not affected	not affected
total	30 words	8 + 8 words	12 + 8 words

Table 5.4: Register Architecture Suspend/Resume State

Clock

When a PiMac machine is initialized, a 64 bit free running clock on each node is zeroed. This clock can be accessed to provide synchronized timestamps on different nodes. It is also used to support the Pi operation **time**. At 50 MHz, this clock wraps around every eleven thousand years.

Node ID

This register is set by software during initialization. It supports the **node-id** operation in Pi.

Segment Map

Each segment on a node is assigned a unique name or *segment ID*. The node maintains a *segment map* which maps these numbers to the corresponding segment in memory. This table is located in local memory at address 0000–03FF. It can be accessed in a microsequence in one memory cycle.

[5]These values must be restored from the segment map.

5.2.5 Instruction Set Architecture

The instruction set for PiMac contains the typical collection of sequential instructions, plus several additional instructions for efficient Pi support. The instruction set architecture is composed of several *instruction types*, each supporting one or more *addressing modes.*

Addressing Modes

PiMac supports register to register addressing modes. In addition, it provides special modes for segment addressing. The eight addressing modes in the instruction set architecture are:

mode 0 — register ← register *op* (segment + variable offset)

This addressing mode accesses one operand from a register. The second operand is accessed from a segment. The segment is specified by a register containing a segment pointer. The offset is specified by a register containing an integer. Both registers are specified by the instruction.

mode 1 — register ← register *op* (segment + constant offset)

This addressing mode accesses one operand from a register. The second operand is accessed from a segment. The segment is specified by a register containing a segment pointer. The offset is specified by a 14 bit constant stored in the instruction.

mode 2 — register ← register *op* register

In this addressing mode, both operands and the destination are specified by registers.

mode 3 — register ← register *op* constant

This addressing mode accesses one operand from a register. The second operand is a 16 bit sign extended constant stored in the instruction. The result is stored in a register.

mode 4 — (segment + variable offset) ← register

This addressing mode is for storing a value from a register to a segment. As in mode 0, The segment slot is specified by two registers, a segment pointer and an integer offset.

mode 5 — (segment + constant offset) ← register

This addressing mode is for storing a value from a register to a segment. As in mode 3, The segment slot is specified by a register containing a segment pointer and a 14 bit offset constant.

mode 6 — register ← special register

code	name	description
0	Rz	constant (zero)
1	Rs	active segment pointer
2	Ra	register A
3	Rb	register B
4	Rc	register C
5	Rd	register D
6	Re	register E
7	Rf	register F

Table 5.5: Instruction Register Codes

This addressing mode is for loading a value from a special register.

mode 7 **special register ← register**

This addressing mode is for storing a special register.

As mentioned earlier, a zero segment size descriptor disables segment bounds checking. This provides a mechanism for non-segment memory access using a segment pointer with a zero base and size descriptor. In mode 0 and 4, the offset register can contain up to a 16 bit offset. This allows the entire local memory to be treated as a contiguously addressable block. This feature is important for system routines like heap compaction.

Table 5.5 lists the codes for register fields of an instruction. Besides the six general registers, it includes two special references. Code zero accesses a pseudo-register R0 which always contains a zero. It can be written (unsuccessfully) to other values without an error. It is useful for supporting operations like **move (add Ri, Rj, R0)**, and **nop (and R0, R0, R0)**. It is also useful as the base register for non-segment (absolute) addressing. Since the segment number, base, and size fields are all zero, it can support linear addressing in address modes 0 and 4. This special register does not cause type traps. The second special reference, Rs references the active segment. This pseudo-register is set automatically, and cannot be modified. It is provided for low overhead access to the active segment.

Table 5.6 lists the codes for the special registers accessed in modes six and seven. When accessing the four priority queues and the free segment list, the tail pointer is positioned at word bits 6-15. The head pointer is positioned at word bits 22-31.

Figure 5.5 presents the instruction format of each addressing mode. The address mode decoding is determined by the mode bits. The first mode bit (bit 29) indicates whether a constant or register-l is being accessed from the instruction. The second mode bit (bit 30) determines whether the operation accesses a segment (as specified in registers k, l, or a constant). The last address mode bit (bit 31) (in combination with bit 30) indicates whether an ALU operation is involved, and determines the direction of a segment access. It also determines whether general or special registers are being accessed.

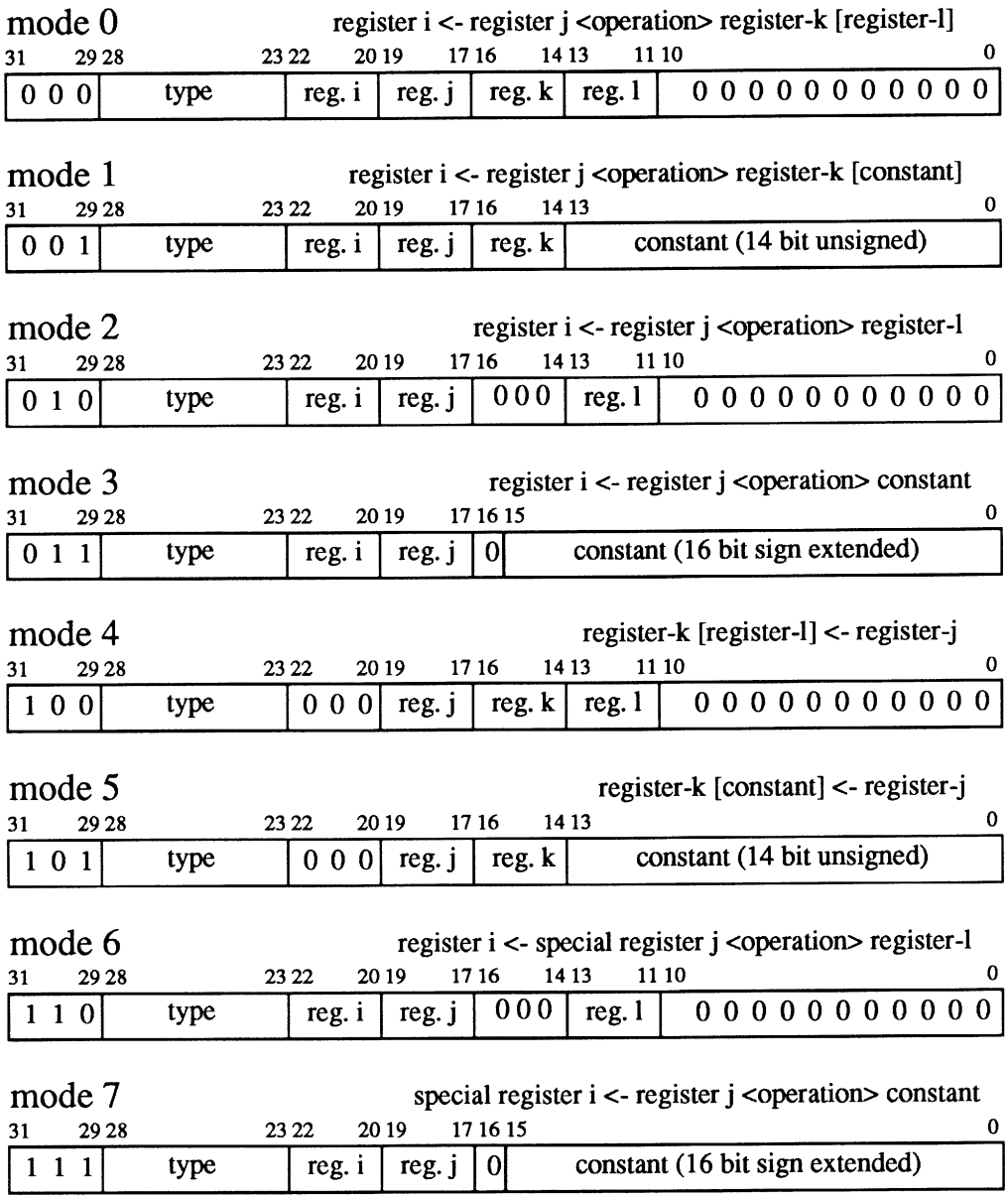


Figure 5.5: PiMac Instruction Formats

code	description
0	clock (low 32 bits)
1	clock (high 32 bits)
2	node ID
3	task queue 0
4	task queue 1
5	task queue 2
6	task queue 3
7	free segments

Table 5.6: Special Register Codes

class	instruction types
arithmetic	Add, Subtract, Multiply
logical	And, Or, Xor
shifting	Arithmetic-Shift, Logical-Shift, Rotate
comparison	Compare
memory store	Store
tag support	Read-Tag, Write-Tag
sync support	Read-Count
hash support	Hash, Match
branching	Branch-Plus, Branch-Plus-Zero, Branch-Zero, Branch-Not-Zero, Branch-Minus, Branch-Minus-Zero
task support	Resume, Send, Call, Suspend, End

Table 5.7: Instruction Types

These instruction formats could be packed more tightly, increasing the code density. Perhaps two instructions could be packed into a single word. However, this would complicate instruction decoding, and cloud the clarity of the machine architecture. PiMac is intentionally simple.

Instruction Types

The PiMac instruction set architecture includes 27 instruction types. A type defines the operation performed by an instruction. The instruction types are summarized in Table 5.7.

Add $C = A + B$

Subtract $C = A - B$

Multiply $C = A * B$

These instruction types perform two's complement arithmetic operations on integers, producing an integer result. These operations can produce overflow/underflow traps.

And $C = A \wedge B$

Or $C = A \vee B$

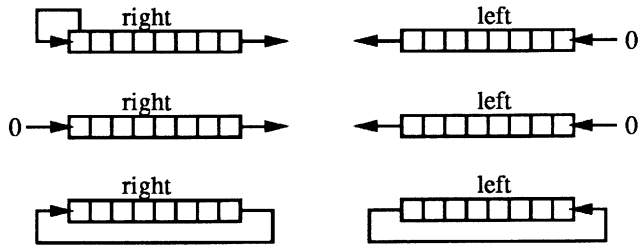
Xor $C = A \oplus B$

These instruction types perform the logical operations on boolean and integer values. The logical operation **Not** is achieved as **(Value Xor -1)**.

Arithmetic-Shift

Logical-Shift

Rotate



These instruction types shift or rotate an integer by a specified number of binary digits. Arithmetic shifts preserve the sign of the integer. Logical shifts insert zeros. Rotate wraps around the shifted out bit. B can be positive or negative. Arithmetic shifts can cause overflow/underflow traps.

Compare $C = \text{compare}(A, B)$

This instruction type produces an integer comparison of two values. The values can be of any type, but they must be the same. If the compared values are integers or boolean, the comparison result is 1 if $X > Y$, 0 if $X = Y$ and -1 if $X < Y$. For all other types, the comparison performs an equivalency test, zero if $X = Y$, non-zero if $X \neq Y$. The result is always an integer.

Store $C \leftarrow A$

This instruction type stores a value in a segment. The value can be any type.

Read-Tag $C = \text{tag}(A)$

Write-Tag $\text{tag}(C) = A$

These instruction types access the 16 bit tag of a segment value or the three bit tag of a register. When a tag is read, it is copied to the least significant 16 bits of the register in the format of a segment tag. The format is the same as the tag format specified in section 5.2.3. Tags are written using the same format. Register tags are copied to the same position in the lower 16 bits. All other tag fields are zeroed. Copied tags are typed as integers.

Read-Count $C = A$

This instruction reads the value of a word. It bypasses the read lock mechanism so it never causes a task suspension. It is used to support the adjust-count operation in Pi.

Hash — C = hash (key, size)

This instruction type computes a hash index using the specified key and size. The function is computed using specialized hardware. No specific hash function is guaranteed. The value, key, and result are all integers. This hash function is identical on all nodes.

Match — C = match (key, segment quadword)

This instruction type performs four simultaneous comparisons to quadword aligned values in a segment. If a value matches, the segment offset of the value is returned. If no match is found, -1 is returned. The key is an integer; the segment quadword is specified as a segment pointer, offset pair. The resulting offset is an integer.

Branch-Plus — if (A $>$ 0), IP $\leftarrow$ IP + branch offset

Branch-Plus-Zero — if (A $\geq$ 0), IP $\leftarrow$ IP + branch offset

Branch-Zero — if (A = 0), IP $\leftarrow$ IP + branch offset

Branch-Not-Zero — if (A $\neq$ 0), IP $\leftarrow$ IP + branch offset

Branch-Minus — if (A $<$ 0), IP $\leftarrow$ IP + branch offset

Branch-Minus-Zero — if (A $\leq$ 0), IP $\leftarrow$ IP + branch offset

These instruction types perform a conditional, relative branch if the specified test is true. The test variable and the offset are integers.

Resume — resume (task A)

This instruction type forces a suspension of the active task, and a resumption of the task segment specified by the instruction. The segment specifier must be a segment pointer.

Send — network controller $\leftarrow$ A

This instruction type gives one word to the network controller to be sent out on the network. A task executing the send instruction waits until the network controller acknowledges that the word has been sent. For simplicity, a two word send instruction is not included here.

Call — call (task A)

This instruction calls a task specified by a segment ID. First the current task is suspended and inserted on the head of the highest priority task queue. Then the specified task ID is invoked, as if it had just been dequeued.

Suspend — **suspend (active task)**

This instruction type forces a suspension of the active task. The dequeue microsequence is then executed.

End — **end (active task)**

This instruction type ends the current task execution thread. The dequeue microsequence is executed.

Table 5.8 illustrates the addressing mode/instruction type combinations supported in PiMac.

5.2.6 Network

PiMac nodes are connected via a three dimensional mesh network. Each node contains a piece of the network (a router), plus a network controller. The router and interconnect topology is incorporated directly from the J-machine [16, 20, 18, 54]. The network controller has several distinguishing characteristics.

Like the controller in the J-Machine, the PiMac network controller sits on the edges of the network mesh and enqueues/dequeues messages for a specific node. However, the PiMac controller has no output buffering. If a message being sent into the network blocks, the task executing on the node stalls until the network clears. A message sent to and from the same node bypasses the network controller, so a trivial deadlock situation is avoided.

The network controller buffers an incoming message in the input buffer. Once the buffer is filled, the incoming message is stalled in the network while the network controller arbitrates for access to node memory to write the buffer. When the network controller receives the first word of the message, the length, it blocks the message until it can obtain a segment equal to or exceeding the storage requirement. It then absorbs the message from the network and writes it to the segment four words at a time. A segment is allocated from the free segment list.

After the network controller has stored the incoming message into a segment, it inserts it onto the end of the appropriate priority message/task queue. The queues are maintained by the task queue registers in the register architecture.

5.2.7 Handlers

Code for PiMac is stored in read-write segments. A code segment implements a specific message *handler*. Handlers are identified by their segment ID. There are two types of handlers: *resident* and *non-resident*. Resident handlers are created (permanently) on every node at system initialization time. The machine dependent compiler allocates an identical segment ID for each resident handler on all nodes. Non-resident handlers are not present on a node unless they are specifically copied there. They are not guaranteed to have identical segment IDs on all nodes. However, both handler types are invoked identically.

types	0	1	2	3	4	5	6	7
Add	•	•	•	•			•	•
Subtract	•	•	•	•			•	•
Multiply	•	•	•	•			•	•
And	•	•	•	•			•	•
Or	•	•	•	•			•	•
Xor	•	•	•	•			•	•
Arithmetic-Shift	•	•	•	•			•	•
Logical-Shift	•	•	•	•			•	•
Rotate	•	•	•	•			•	•
Compare	•	•	•	•			•	•
Store					•	•		
Read-Tag	•	•	•	•				
Write-Tag			•	•	•	•		
Read-Count	•	•	•	•			•	•
Hash	•	•	•	•			•	•
Match	•	•	•	•			•	•
Branch-Plus			•	•				
Branch-Plus-Zero			•	•				
Branch-Zero			•	•				
Branch-Not-Zero			•	•				
Branch-Minus			•	•				
Branch-Minus-Zero			•	•				
Resume	•	•	•	•			•	•
Send	•	•	•	•			•	•
Call	•	•	•	•			•	•
Suspend			•					
End			•					

Table 5.8: Instruction/Addressing Mode Map

0	code segment
1	instruction pointer
2	register A
3	register B
4	register C
5	register D
6	register E
7	register F
8	message length
9	message priority
10	message type
11	message locals

Table 5.9: Task Segment Format

5.2.8 Tasks and Messages

A task is created when a message is received on a node. The message contents are stored on the node in a *task segment.* A task segment is a read-write segment which obeys special conventions for storage in the bottom twelve words. Table 5.9 shows the format of a task segment.

The first four words of the message are stored in the task segment beginning at offset eight. The network controller reads the locals (number of locals) field to determine the offset for the first argument. Argument zero is stored at offset *locals* + 12. The network controller also stores the message type field in the segment ID field of the code segment (offset 0). The type field contains the segment ID of the appropriate message handler. The instruction pointer is also zeroed to begin the execution with the first word of the code segment.

Only 255 segments on each node can be the active segment for a task. Since segment names are used as task names, this limits the number of tasks that can exist on a node simultaneously. This design limit is based on the expectation that most tasks are very short lived. In the examples in Chapter 3, over 99 percent of executed tasks are for communication only (data delivery). They execute few instructions and are non-blocking. This ephemeral nature means the probability of deadlock on a node, caused by many suspended task and no free task IDs, is unlikely. If a node does run out of segment IDs, the condition is handled by a catastrophe routine that handles errors such as out of memory, by zero, etc. Typically, this handler halts activity on the node and issues error messages to the host.

Each task segment includes a *link* which is used to create chains for task queues and lists. The task ID zero is reserved as an end-of-chain marker.

5.2.9 Microsequences

There are two task management microsequences in PiMac: Suspend and Dequeue. They are supported as microsequences rather than trap handlers because of their critical impact on task overhead. When a microsequence is invoked, normal program execution is suspended and a sequence of predefined microoperations is performed. Microsequences only occur at the completion or abortion of a PiMac instruction. Microsequences cannot cause traps or invoke other microsequences.

Suspend

A task can suspend for three reasons: (a) the task accesses a locked segment location (i.e., a synchronization suspend), (b) the task executes the **suspend** instruction, or (c) a trap event occurs. In all cases, the following microsequence occurs:

1. Registers C, D, E, and F are written as a quadword to the active segment beginning at offset four.
2. The code segment, instruction pointer, register A, and register B are written as a quadword to the active segment beginning at offset zero.
3. If the active buffer dirty bit is set, it is written back to its location in the active segment.
4. If the other buffer dirty bit is set, it is written back to the appropriate segment.

For synchronization suspends, several additional operations are performed concurrently with these steps. During step 1, the task ID of the current task is exchanged with the suspended task field of the accessed location. The instruction pointer is also decremented during this step. During step 2, the previous suspended task field is stored as the link of the task segment.

For suspend operations, two additional steps occur:

5. The tail of the current priority queue is used to index a task in the segment map.
6. The first quadword of the indexed task in loaded into the other buffer and its task link is set to the recently suspended task ID. The tail of the current priority queue is set to the suspended task ID as well.

These steps enqueue the task at the end of the appropriate suspended task queue.

For a trap event, the suspended task ID is stored in the trap task ID register.

Consider an example of this microsequence. Figure 5.6 illustrates a case where a task (ID 5) executes a blocked access on the second word (d2) of segment (ID 7). The

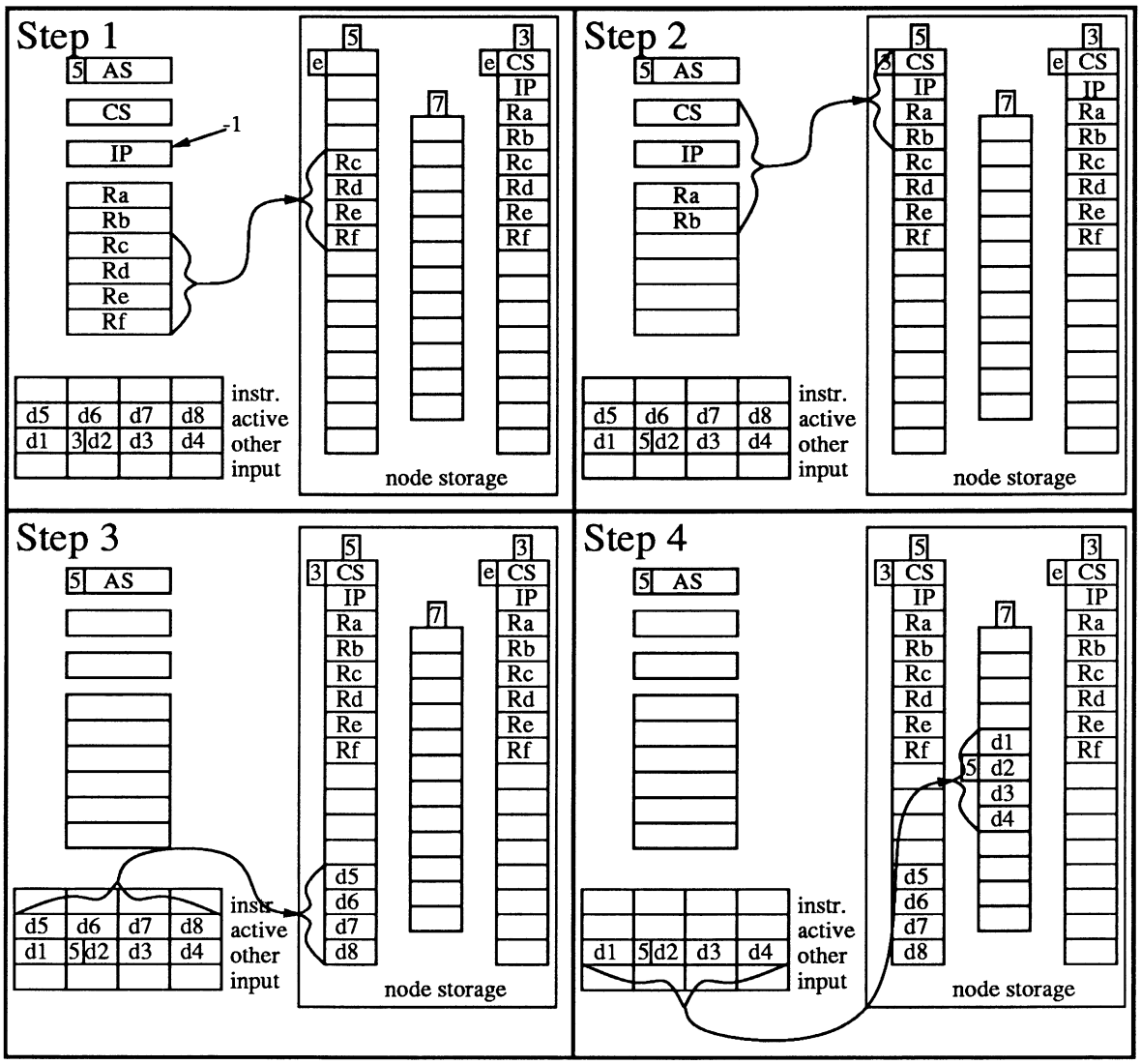


Figure 5.6: Suspend Microsequence Example

slot d2 already has one suspended task (ID 3) in its suspended task field. In step 1, registers C through F are stored in the appropriate place in the active task segment. The suspended task field of the accessed value is exchanged with the current task ID. Here, task ID 3 is exchanged with task ID 5. Also in step 1, the instruction pointer is decremented.

In step 2, register A and B, the code segment, and the instruction pointer are written to the active segment. During this write, the link of the task segment (ID 5) is set to the previous head of the suspended task chain (ID 3) of word d2.

In step 3, the active buffer is written back to the appropriate place in the active segment if its dirty bit is set.

Step 4 writes the other buffer back to segment 7. This also updates the new suspended task chain head for value d2. After the microsequence completes, the state of task 5 has been preserved in the task segment, and the task has been pushed onto the suspended task chain of the value d2 of segment 7.

Dequeue

This microsequence occurs whenever a task completes or is suspended. It also executes at the completion of a trap handler. It is responsible for selecting, dequeuing, and starting or restarting a task. The following steps occur:

1. The highest priority, non-empty queue is selected. The head pointer is used to index a task in the segment map.
2. The first quadword of the index task is loaded into register A and B, the code segment, and the instruction pointer. The queue head pointer is set to the value task link. The active segment task link is also set to empty.
3. The second quadword is loaded into registers C, D, E, and F.
4. If the code segment base and bound fields are empty, the segment is loaded from the segment map.

Execution is then turned over to the task at the current instruction pointer. For a trap handler completion, step 1 obtains the task ID from the trap task register rather than a task queue.

An example dequeue sequence is shown in Figure 5.7.

In step 1, the next task to be executed is identified from the task queue. The highest priority queue, zero, is empty (indicated by *e*). The next highest priority queue contains three tasks: five, three, and seven. The first task, five, is selected as the current active task. It's base and size are loaded from the segment map. In step 2, the code segment, instruction pointer, and register A are restored. The task queue head is set to the next task in line (ID 3). Also, the task link of task 5 is cleared. In step 3, the remaining processor state is loaded. Finally, the code segment base and size are restored from the segment map (assuming they were empty) in step 4.

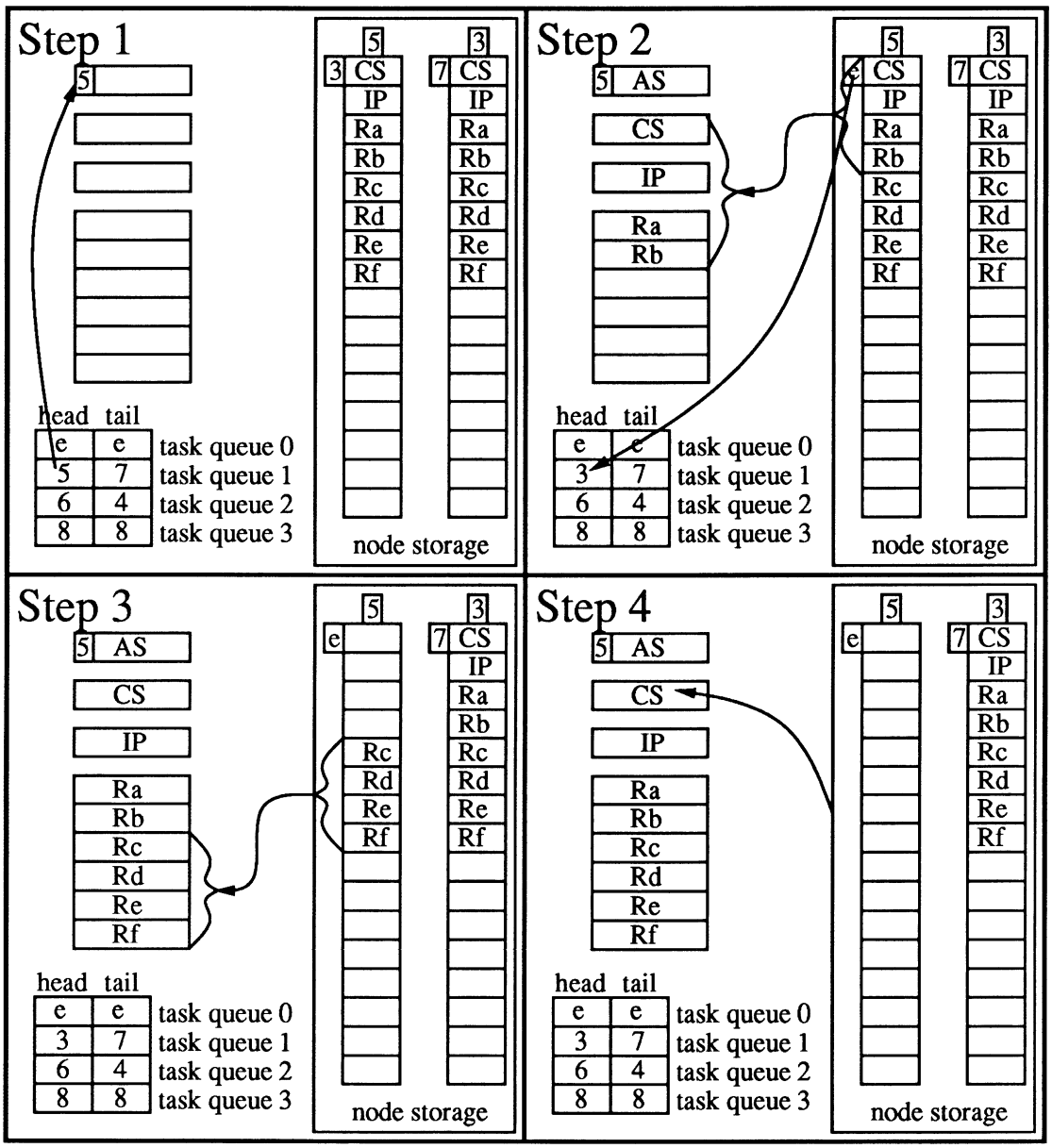


Figure 5.7: Dequeue Microsequence Example

5.2.10 Traps

The PiMac architecture includes a trap mechanism to handle exceptions. When a trap is generated, the suspend sequence is generated and the suspended task ID is stored in the trap task register. At this point, traps are disabled, preventing unrecoverable double trap errors. Then the trap number is used to reference the trap handler segment ID (which is loaded at system initialization time with the resident handlers). This handler is invoked using a trap segment that is reserved for use as the active segment for traps. When the trap handler completes, the dequeue microsequence is executed using the trap task register as the source task. Trap handlers contain system code, but are identical in format to other handlers.

Most trap handlers perform the standard responses to exceptions (e.g., ALU overflow, queue empty, etc.). These handlers are outside the scope of this thesis. The following trap handlers perform more specific functions in the machine architecture.

Enqueue

The enqueue trap can occur (a) if a task performs an access on a segment location which enables the suspended tasks to be requeued for execution, (b) a task executes the suspend instruction, or (c) a trap occurs. For (b) and (c), the enqueue trap is executed immediately following the suspend trap.

This trap takes a suspended task chain, and inserts it, in reverse order, on the appropriate priority task queue. The chain order must be reversed to preserve FIFO ordering of suspended tasks. The trap walks down the suspended task chain, reversing the chain links. Then it walks back up the chain inserting each entry on the tail of a task queue. The task queue is determined by accessing the message priority field from the task.

Segment Allocation

This trap occurs when the free segment list becomes empty, or when an abnormal sized segment is required by the network controller. The trap handler allocates one or more segments and adds them to the segment free list. This trap handler scans the segment map for free segment IDs.

This trap does not violate the atomicity requirement of Pi since the trap handler does not affect any state related to the executing task. It is an implementation simplification.

5.3 A Gate Array Implementation

In this section, a gate array implementation of the PiMac machine architecture is considered. In order to evaluate the feasibility of an implementation, estimated gate

module	device count	gate count
diagnostic interface	3,696	616
instruction prefetcher	3,168	528
controller	8,373	1,396
address datapath	75,244	12,541
integer datapath	39,368	6,561
external memory interface	8,990	1,498
internal memory interface	13,373	2,229
network interface (in)	4,450	742
network interface (out)	18,214	3,036
router (three dimensions)	30,540	5,090
total	205,416	34,236

Table 5.10: MDP Device and Gate Count Estimates (24 July 1990)

counts and timing of a gate array PiMac are examined. This section draws heavily on the design of the Message Driven Processor [14, 15]. The specification of several modules of the microarchitecture is taken directly from corresponding modules in the MDP. Because of the close relationship of PiMac to the MDP, an overview of the MDP is provided.

5.3.1 The Message Driven Processor

The MDP is being currently being designed in the Concurrent VLSI Architecture Group at MIT. Work began on the MDP in 1986. Nearly all of the design has been completed by graduate students. The design team has averaged around six to eight members over the design period.

The architectural specification was completed in the Summer of 1988. A register level specification was completed in Spring of 1989. The hardware architecture is implemented using an Intel standard cell library. The implementation will be fabricated in a one micron CMOS process on a 10 x 15 mm die. The expected clock rate is 15 MHz.

Table 5.10 summarizes device and gate estimates for the MDP[6]. It is difficult to compare the library cells used for the MDP with the gate array gates used for PiMac. The gate counts provided in Table 5.10 are computed using an average device per gate ratio of six. The device per cell averages are higher (between eight and twelve), but many of these larger cells require multiple gates to implement.

The implementation of PiMac described here uses a less aggressive gate array technology. The PiMac machine architecture is in many ways simpler than the MDP.

[6]This table includes logic only; MDP also includes 4096 words of on-chip memory (~890,000 devices)

However, the MDP still provides the most accurate information for estimating details about this PiMac implementation.

5.3.2 PiMac Module Designs

This section examines each module of PiMac's microarchitecture with respect to the corresponding MDP module or modules. Gate estimates used in this section are derived from LSI Logic macrocells [37, 40].

Router

The PiMac router design is taken directly from the MDP. It is comprised of three independent dimensions. Each dimension of the router buffers a flit of the message passing through it, and compares the count of the destination field to the node's address (in the appropriate dimension). If the dimension matches the node's address, the message is passed to the next dimension for routing, or to the node if the final dimension is routed. Otherwise, the message is routed towards the destination in the current dimension.

The router module requires roughly 5000 gates. It can transmit a flit in one clock, with a half flit transmitted on each clock edge.

Network Controller

PiMac's network controller differs from MDP's network interface in several ways. First, PiMac has no outgoing message buffer. The send operation takes a word from the datapath and inserts it directly into the network router via the network controller. The next instruction cannot begin until the sent word has started entering the network router. Because of this difference, the gate estimate for MDP's outgoing network interface is higher than the PiMac counterpart.

Second, PiMac network controller performs a more complicated procedure for receiving a message from the router, since it must determine the required storage size, allocate the storage from the free segment list, or request a larger segment from the processor. While no additional buffering is required, the control mechanism is much more complicated. This gate complexity is comparable to that of the MDP's outgoing network interface.

PiMac's network controller requires roughly 5000 gates. For an outgoing message, it has a delay comparable to the datapath. For incoming messages, the network interface requires one clock to compare the length field to the fixed message length, one clock to allocate the segment from the free segment list, or request a larger segment from the processor, and one clock to store the first word into memory. After the first word is stored, additional words are stored every two clocks (the reception rate from the router). The message is stored into memory via the input buffer.

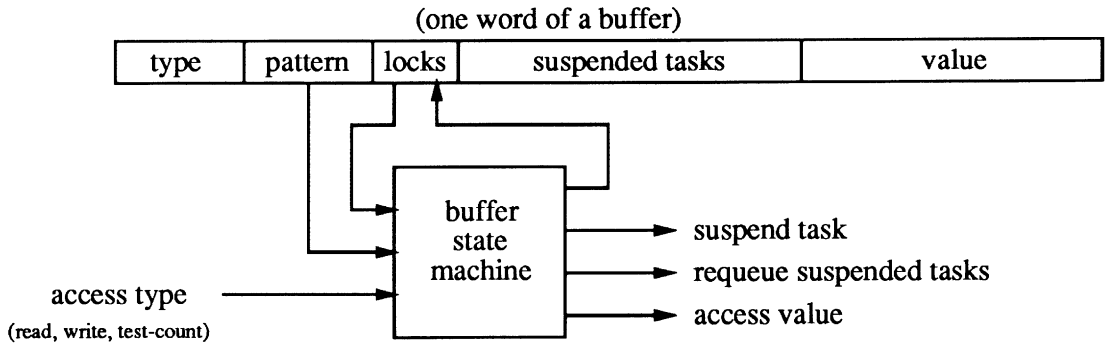


Figure 5.8: Buffer State Machine

Datapath

PiMac's datapath is similar to the MDP's. It provides fewer registers (463 register bits for PiMac[7] versus 1660 register bits for MDP). It includes logic from MDP's address datapath, integer datapath, and internal memory interface.

The approximately 6500 gates required by the MDP integer datapath is comparable to the corresponding datapath in PiMac. PiMac has fewer data registers (six for PiMac versus twelve for MDP). However, PiMac's registers have an extra port to allow them to be accessed as a quadword with memory. The datapaths contain similar logic in the ALU.

The address datapath requires a more detailed comparison. The MDP's address datapath module was designed to support a wide range of architectural features besides address arithmetic. Many of these other features are not provided in PiMac. For example, PiMac does not support circular queues, ID registers, status bits, dedicated address registers, multiple execution priority levels, multiple fault handling, or special translation addressing. Most of the features unique to PiMac are supported in the segment buffer module.

Using a breakdown of the MDP's address datapath, an estimated 3500 gates are required to support address and instruction pointer arithmetic. This brings the total gate count estimate for PiMac's datapath to 10000.

Segment Buffers

The four segment buffers provide memory caching for the datapath, instruction unit, and network controller (for incoming messages). The active segment and other segment buffers also include simple state machines on each word to support access behaviors for attributes. One of these state machines is shown in Figure 5.8.

The active segment and other segment buffers also include comparators to support the match instruction. These comparators are shown in Figure 5.9.

[7]PiMac contains an additional 828 register bits in the segment buffers but they are not counted here.

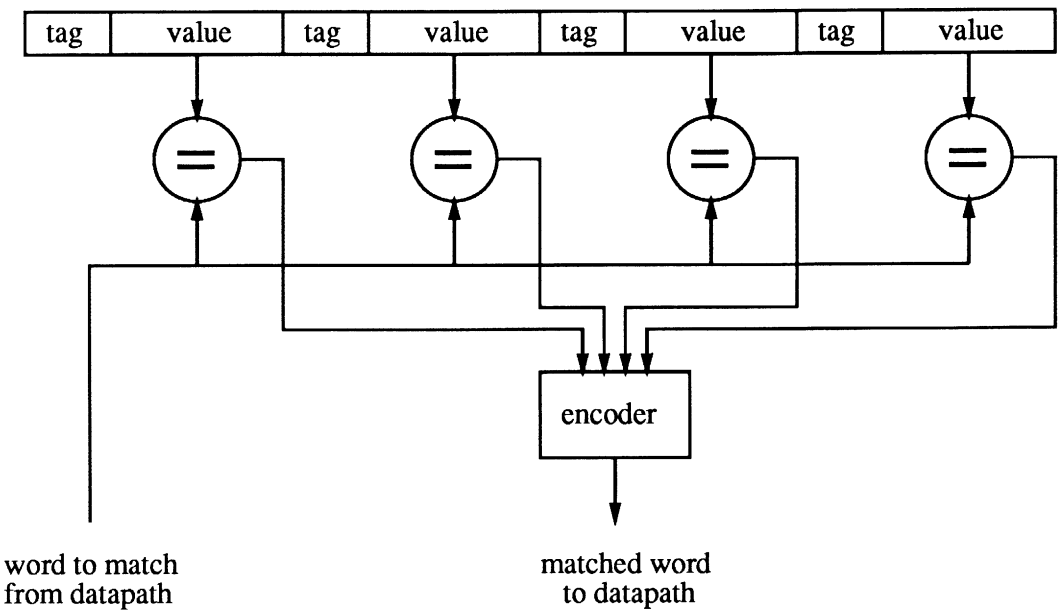


Figure 5.9: Buffer Comparators

buffer type	total bits	gates/bit	total gates
active and other	384	7	2,688
input	192	6	1,152
instruction	192	4	768
key and dirty bits	60	4	240
total	828		4,848

Table 5.11: PiMac Buffer Gates (not including comparators and selection logic)

There are four buffers in this module. Each buffer consists of four 46 bit words, a dirty bit, and a 14 bit tag. Two of the four buffers are used in conjunction with the datapath (active and other). Each bit of these buffers is gated on four busses (three for datapath, one for memory interface). A gate array implementation of a buffer bit requires seven gates. The input buffer contains an internal shift register. Both the input and instruction registers have a single input and output port. Input buffer bits require six gates, and an instruction buffer bits require four gates. The total estimate for buffer storage is summarized in Table 5.11.

An additional 1000 gates are required for the comparators and 500 gates for selection logic. Therefore, the total gate estimate for the segment buffers module is 6500 gates.

module	gate count
network router	5,000
network controller	5,000
datapath	10,000
segment buffers	6,500
memory controller	1,500
instruction unit	500
controller	1,500
total gates	30,000

Table 5.12: PiMac Module Summary

Memory Controller

PiMac's memory controller module is taken directly from MDP's external memory interface. It should therefore require roughly 1500 gates to implement. The external memory interface accesses a quadword of external memory as 16 twelve bit chunks. The entire access takes approximately 720 ns. An on-chip memory cache is considered in Section 5.3.4.

Instruction Unit

The instruction unit fetches and decodes instructions in parallel with instruction execution in the datapath. Instructions are fetched, four at a time, into the instruction buffer. Simple logic predecodes the address mode and instruction type and presents it to the controller. Because of the simpler (but less compact) instruction format of PiMac, a slightly smaller instruction unit is expected. The gate estimate is 500 gates.

Controller

PiMac's controller is comparable to the MDP's controller. Both architectures have a similar number of features that contribute to the controller's complexity. For example, PiMac has simpler fault handling than the MDP, but it includes two "hardwired" microsequences. The estimate for PiMac's controller is 1500 gates.

PiMac module gate count estimates are summarized in Table 5.12.

5.3.3 PiMac Timing

Like the MDP, it is expected that PiMac is designed with a maximum of eight gate delays per clock phase (16 total). Most PiMac instruction types can execute in a single clock cycle. However, several conditions affect the instruction rate.

Segment Buffer Accesses

When an instruction accesses a word in a segment, the segment address is compared with the tags of the active and other segment buffers. If there is a match, the correct word is selected and made available to the datapath in the next cycle. This means that instructions that use addressing modes 0, 1, 4, and 5 execute in a minimum of two clocks.

If the accessed word is not in the segment buffer, it must be loaded from memory. This process involves freeing either the active or other segment buffer. If the appropriate buffer's dirty bit is set, the buffer must first be written back to memory. Then the accessed word can be loaded. The extra cycles[8] required for these memory accesses are added to the instruction time.

Instruction Buffer Accesses

The instruction buffer holds four instructions. When the instruction unit fetches an instruction found in the buffer, it is accessed in a single cycle. When an instruction is not present in the buffer, the appropriate quadword of the code must be fetched from memory. This also adds extra cycles to an instruction time.

Microsequence Invocations

Microsequences can be invoked either directly (via the **resume**, **call**, **suspend**, or **end** instructions), or indirectly by a segment access. In either case, the microsequence execution time is added to the instruction time.

5.3.4 A PiMac Gate Array

The proposed PiMac gate array contains approximately 30,000 gates and executes at a cycle time of 16 gate delays. This could be realized using several commercial gate arrays. For example, an LSI Logic LCA100K gate array (.7 micron HCMOS) could accommodate this design with a clock rate of up to 50 MHz (20 ns) [38].

A more interesting alternative is the embedded array gate arrays which combine gate array logic with a high density static memory array on a single die. For example, the LEA100K series would provide an estimated 1000 words of memory on-chip with the 30,000 gate PiMac design [39]. This would reduce the memory access delay for a quadword (192 bits) from 16 external memory cycles (720 ns), to a single quadword on-chip memory access (40 ns).

A higher performance implementation of the PiMac machine architecture could be constructed. This implementation has selected because it is (a) simple to design, and (b) it is not aggressive with respect to hardware techniques and technology. It is the

[8]The number of extra cycles depends of the memory access speed.

kind of implementation that can be undertaken in a university research group. If a commercial version of a PiMac-like machine architecture is ever produced, it will certainly incorporate more advanced techniques and technology producing a much higher performance machine. This gate array implementation defines a baseline Pi substrate.

5.4 Summary

This chapter presents PiMac, a Pi machine architecture. Its design is based on the requirements of Pi, and the composite model mechanism characteristics from Chapter 4. PiMac has also been strongly influenced by the design of the Message Driven Processor at MIT. The significant features of PiMac include:

fast task suspends and dequeues (four memory cycles each)

fast task allocation/deallocation via the free segments list

multiple task queues for priority messages

autonomous network router

low overhead for message sending and receiving

segment buffers for low memory access latency

segment addressing modes with bounds checking

low cost synchronization with per word granularity

support for associative segments

This chapter also described a proposed implementation of PiMac on a single gate array with 1000 words of on-chip memory and a clock rate in excess of 25 MHz.

While PiMac is still only a proposed system[9], it ratifies the goal of this chapter: to demonstrates the feasibility of a Pi substrate. The effectiveness of this substrate in supporting Pi is the subject of the next chapter.

[9]The actual realization of PiMac will be pursued in future research.

Chapter 6

Pi on PiMac

This chapter examines the support of the Pi interface on PiMac. The match of the Pi abstract machine to the PiMac machine architecture is considered. Some functions of the machine dependent compiler and the runtime operating system are also discussed. To avoid confusion, the term *instruction* is used to denote PiMac machine instructions; the term *operation* is used to denote Pi abstract machine operations. The compiler and runtime systems described in this chapter were influenced by [27, 28, 29, 57].

6.1 Storage

In both the Pi abstract machine and the PiMac machine architecture, storage is presented as part of a complete node. This provides locality between the processing and storage elements of the system. Read-Write and associative segments are supported using the same local memory in the PiMac machine architecture.

6.1.1 Read-Write Storage

PiMac's instruction set architecture is designed to provide efficient support for read-write segment accesses. Segment offset addressing with bounds checking is an addressing mode for most instruction types. Segment buffers cache named segment locations, reducing the latency of these accesses. Register addressing modes are also provided to reduce the load on the memory system and simplify the instruction formats.

A Pi read operation is supported by addressing modes 0 and 1. One read and a dyadic operation (e.g., add) is supported by a single PiMac instruction. If the other operand for the dyadic operation is not present in a register, it is placed in one, using a pseudo-load instruction (e.g., (or operand 0)).

The machine cycle cost of performing a dyadic Pi operation depends on where the operands reside. Table 6.1 summarizes the cost for each case. Since some operations are not commutative, operand order can affect the cost. This table assumes that (a)

operand A	operand B	cost	mode
register	register	1 cycle	mode 2
register	constant	1 cycle	mode 3
constant	register	2 cycles	mode 2[1]
register	segment	2 cycles	mode 0
segment	register	3 cycles	mode 2[2]
constant	segment	3 cycles	mode 0[1]
segment	constant	3 cycles	mode 3[2]
segment	segment	4 cycles	mode 0[2]

Table 6.1: Dyadic Operation Costs

segment locations are present in the segment buffers, and (b) the operation destination is a register. If the operands are in segment locations that are not present, the cost of loading them must be added to the total. If the result must be written to a segment, an additional cycle is required. Additional time is also required if the access of an operand results in an enqueue trap (the time is dependent on the length of the suspended tasks list).

A Pi **write** operation is supported directly by the **store** instruction (addressing modes 4 or 5). This instruction requires two machine cycles. Sometimes the machine dependent compiler can eliminate segment reads and writes by retaining results in registers. This is possible if (a) the segment location is not attributed, and (b) other tasks do not read the value in the segment. By retaining local and intermediate values in registers, segment lengths can also be reduced.

Read-Write segment allocation is supported by the runtime portion of the operating system. Since storage and segment ID allocation are a critical part of task overhead, PiMac supports special mechanisms and techniques for efficient read-write segment allocation.

Most read-write segments used in the examples in Chapter 3 are small and ephemeral. Therefore the runtime system uses a fixed segment size for nearly all segment requests. In Chapter 4, it was shown that 95.8% of segments in the example are less than eight words long. 99.0% are less than 16 words long. Since PiMac requires an additional eight words for processor state, a fixed size of 24 words would satisfy most segment needs.

Free segments of this fixed size are maintained by the free segment list on the PiMac register architecture. This supports very fast segment allocation for the most common type of requested segment (segments less than 24 words). The network controller can allocate a segment from this list in two machine cycles. A Pi **create-read-write-segment** operation allocates a fixed sized segment with the following sequence:

[1]The constant operand is first preloaded into a register.
[2]The segment operand is first preloaded into a register.

1. Compare the requested segment size to the fixed size.
2. If it is too long, branch to a more complicated allocation routine.
3. Read the head of the free segments list.
4. Read the segment link (the suspended tasks field of word zero).
5. Write the new head of the free segment list.

This sequence takes six machine cycles. Read-Write segments that are larger than the fixed size are handled by a slower, heap storage management mechanism. The network controller generates a segment allocation trap if a large segment is required.

Since segment IDs are retained in the free segment list, ID allocation is provided automatically, except for the large segment case, where the segment map is scanned for an unused ID.

The machine dependent compiler predefines certain segment descriptors (including the segment IDs) for resident handler code segments, the nodal segment, trap handler code and local variable segments. It also defines nodal variable offsets. These descriptors and offsets are compiled in-line, avoiding more expensive runtime lookups. This allows these objects to be accessed quickly (in two cycles if the appropriate value is in the segment buffer).

6.1.2 Associative Segments

Associative segments are required much less frequently than read-write segments. When created, they have a much longer lifetime. Because of this, less emphasis is placed on fast allocation. A Pi Create-Associative-Segment is compiled into a sequence of instructions that allocates a read-write segment of the correct size, and initializes it as an associative segment. For an eight word segment, this takes approximately 45 machine cycles.

An associative segment of size N contains $2N+4$ slots. The first N slots contain keys; the second N slots contain values. The extra four slots are used for segment type information. When a key is looked up in the segment (using the match operation), the hash operation is computed on the key and segment entry size. The computed hash value is used to index a quadword in the associative segment. Then a segment buffer is used as a four-way set-associative cache to attempt to match the entry. If a match is found, the value is accessed from the value slots in the segment using the sum of the hash value and the match result as an offset. This procedure takes five machine cycles.

The size and safety parameters play a key role in supporting the insert operation. For SAFE segments, the insert operation must first test to see if the slot in the segment is currently holding another entry. For a collision, the boundedness of a segment determines the behavior. For UNBOUNDED segments, the insert operation

must resize the segment by allocating another read-write segment and rehashing the entries. For BOUNDED segments, the insertion is rejected. For UNSAFE segments, collisions can result in the overwriting of an entry, or a resizing of the segment, depending on the segment boundedness.

The **remove** operation employs the **hash** and **match** instruction to locate and remove the specified entry (six cycles). The **clear** operation reinitializes the associative segment. For an eight element associative segment, this takes approximately 40 cycles.

It is unlikely that any Pi substrate will support all associative segment types in hardware. The most straightforward type is BOUNDED-UNSAFE, since this is supported by a conventional set associative memory. Other associative segment types can be supported either by software, or by some combination of software and hardware. In PiMac, the **hash** and **match** instructions provide a balance of efficiency and resource cost for all associative segment types.

6.1.3 Storage Reclamation

Storage can be explicitly deallocated using the **destroy-segment** operation. If the segment is a fixed-sized, read-write segment, it is placed on the free segment list. Otherwise it is marked as free for later reclamation at the next heap compaction. Before placing a segment on the free list, each word must be reinitialized to an empty d-sync. If eight words are used, **destroy-segment** costs approximately 20 machine cycles.

PiMac supports heap compaction via the segment map. Since only segment IDs are passed out of a node, segments can be moved freely within a node as long as the segment map is updated, and internal segment descriptors are adjusted. After a compaction, the base and size fields of each segment descriptor (identified by the type field) are zeroed. On the next access, the updated fields are restored from the segment map.

6.1.4 Nodals

Nodals are stored in a special segment on each node. A nodal is accessed by referencing the nodal segment descriptor using the nodal offset. Both the nodal segment descriptor and the nodal offset are predefined by the machine dependent compiler.

6.2 Synchronization

The synchronizations defined in Pi are supported as attributed locations in read-write segments. These attributes are supported directly in PiMac via the access pattern, access locks, and suspended tasks fields of a segment location's tag.

6.2.1 Data Synchronization

When a location is attributed as D-SYNC, the tag of the location is accessed using the **read-tag** instruction. The read lock bit is set, the d-sync access pattern is stored, and the suspended tasks list is cleared. The location tag is updated using the **write-tag** instruction. The buffer access state machine suspends (via the suspend microsequence) any task attempting to read the location until it is written. When written, the buffer access state machine initiates a requeue microsequence of the suspended task list after the operation has completed.

6.2.2 Barrier Synchronization

Barrier synchronizations are initialized by setting the access pattern to b-sync, clearing the access locks, and zeroing the value fields. The **adjust-count** operation uses the **read-count** instruction to access the value field (thus avoiding a possible suspension). The count is adjusted and tested. If greater than zero, the read lock bit is set. Otherwise, the read-lock bit is cleared.[3] The new count is stored in the value field. The **test-count** operation is supported by reading the value field of the barrier synchronization.

6.2.3 Producer-Consumer Synchronization

Producer-consumer synchronizations are initialized in the same way as data synchronizations, except the access pattern is set to s-sync. The buffer access state machine controls the behavior of the synchronization as defined in Pi.

6.2.4 Attributes

In PiMac, attributes are supported as fields of every word in local memory. This adds a 13 bit overhead to every word, yet it provides fast and selective synchronizations. An alternate approach is to provide a common suspended task list for an entire segment. This reduces the attribute overhead by eight bits per location, and allows longer task IDs. However, this approach has less selectivity, since all suspended tasks are requeued together, independent of which synchronization becomes ready. Also, since the suspended task list is not located with the synchronization, it must be loaded whenever a suspension or requeue occurs. Since this usually involves several loads and stores of a buffer, the overhead of a synchronization is increased.

The **probe** operation is supported by the **read-tag** instruction. The **time** operation is supported by the node clock registers.

[3]The read lock bit is only written if its value changes.

6.3 Communication

The principal communication operation, **send**, is supported directly by the **send** instruction. This instruction injects one word (32 bits value, three bits type) into the network. It is important that the processor supply data as fast as the network can accept it to prevent message "bubbles". Since most messages are short, the compiler can preload message data in registers and buffers. Some message data is also stored as constants in the code sequence (e.g., the end-of-message word). To send an eight word message, eight **send** instructions are executed. If half of the message is constants or registers, and half is in segments, the eight word message takes twelve machine cycles to transmit.

6.3.1 The Network

The PiMac network conforms to the properties of the Pi interface. In this PiMac implementation, message order between two nodes is preserved. This will change as adaptive routing techniques are incorporated in the future.

The PiMac network exerts immediate backpressure on a node by blocking the **send** instruction, halting the processor. The active task is not suspended. The processor waits until the network accepts the word being sent.

In a superior, but more complicated approach, the message sending task is suspended, and communication-only messages (that are guaranteed not to block or send additional messages) are executed. This frees local storage space without creating additional tasks, but there is a greater risk of creating a bubble in the message being sent.

Both techniques use the network for concurrency control. The network acts as a *concurrency moderator*, slowing down message sends (and task creations) as the network becomes more heavily loaded. More sophisticated network moderation techniques employ time averaging, and network router communication messages to control system concurrency more effectively. These techniques will be addressed in future research.

6.4 Task Management

The PiMac machine architecture is designed to support low overhead tasks. Efficient message sends and fast segment allocation contribute to this goal. The section considers other architectural features that reduce task overhead.

6.4.1 Task Storage

Task storage is obtained using the segment allocation mechanisms described in Section 6.1.1. The operation **self** is supported by the Rs register. It always contains the active task segment descriptor.

6.4.2 Task Dispatch

Every message contains a *type field* that specifies the segment ID of the resident handler to be executed. When the message is dispatched, the type field is used to invoke the correct handler. The machine dependent compiler defines the segments IDs for all resident handlers. The operating system uses these IDs when it distributes the resident handlers to each node. This technique supports task dispatch via a single reference to the segment map (two machine cycles).

6.4.3 Task Atomicity

The definition of Pi requires that an executing task cannot be preempted by the arrival of another task. PiMac conforms with the intent of this requirement (i.e., task execution is atomic except for synchronization or explicit suspensions). The PiMac machine architecture does allow suspensions of the active task (via traps) to support several runtime system services (e.g., segment allocation, heap compaction, etc.). But these trap handlers do not interfere with the atomic execution of the active task.

The suspend operation is supported directly by the suspend instruction in six machine cycles. When a task executes a destroy-segment operation on (self), the active segment is deallocated, and the end instruction relinquishes control of the processor.

6.4.4 Task Prioritization

Each task includes a priority word that is used in scheduling. Pi provides an arbitrary number of priority levels, but does not guarantee strict correctness of message scheduling. PiMac supports four priority queues with registers in the architecture. This provides low overhead queuing and dequeuing of tasks. The machine dependent compiler can usually map an arbitrary number of priority levels in a Pi program into the four levels provided by PiMac. If runtime computed priorities are employed, only the least significant two bits are used. Alternatively. the compiler can remove runtime determined priorities, and use a different scheme for priority selection.

6.4.5 Calling Tasks

The call operation is supported by the call instruction. Atomicity is preserved by inserting the calling task onto the head of the highest priority queue.

The return operation is supported by a special nodal variable that contains the return value. The use of a single return value does not prohibit nested called handlers, but the machine dependent compiler must move the return value store to the end of the called task.

6.4.6 Variable Argument Passing

Variable argument passing is done in Pi via the **send-segment** and **call-segment** operations. These operations are supported using the **send** and **call** instructions. The machine dependent compiler can often eliminate the additional segment if (a) it is generated within the Pi task that invokes the **send-segment** or **call-segment** operation, and (b) adequate storage is available in the active segment.

6.5 Locality

Locality is primarily supported by the machine dependent compiler and the runtime system. The proximity information provided in the **another-node-id** operation can be used either at compile time, runtime, or both to decide where a task should be created.

The **node-id** operation is supported in PiMac by the node ID register. It is not generally possible to reschedule a task specified to run on (**node-id**) on another node, since it may reference another segment that resides on the same node.

If the topology of a specific Pi substrate is known, Pi abstract node numbers can be used to best exploit locality. If a Pi program that explicitly uses these abstract node numbers is transported to another substrate (with a different topology), it will execute correctly, although not with the same locality-dependent performance.

This is true because all possible abstract node numbers are mapped to a physical node in the Pi substrate. Pi programs that are likely to be executed on several Pi substrates (with different topologies and number of physical nodes), may have better overall performance if generic locality specifications are used. This allows the compiler and runtime system more flexibility in scheduling tasks.

6.6 Sequential Operations

All the sequential operations defined in Pi, except **divide**, are supported by a single PiMac instruction. The **divide** operation is supported by a short sequence of arithmetic instructions (approximately 25 machine cycles).

6.7 Summary

Table 6.2 summarizes the costs of Pi operations on PiMac. The *number* field indicates the dynamic frequency of the operation in the examples in Chapter 3. The *ops* and $\sim$ fields indicate the number of PiMac instructions and machine cycles necessary to support the operation.

operation	number	ops	~	operation	number	ops	~
adjust-count	237,634	2	4	insert	5,403	5	8
and	55,268	1	2	l-shift	0	1	2
another-node-id	6,386	2	2	match	50,572	3	5
attribute	39,344	1	2	minus	701,767	1	2
a-shift	29,078	1	2	mod	35,449	20	25
branch-minus	11,160	1	2	move	565,131	1	2
branch-not-minus	2,273	1	2	not	0	1	2
branch-not-plus	0	1	2	or	8,730	1	2
branch-not-zero	1,129,059	1	2	plus	902,473	1	2
branch-plus	1	1	2	probe	740	2	3
branch-zero	76,640	1	2	remove	1,625	3	6
call	89,180	1	6	return	10,564	1	2
call-segment	256	1	6	send	488,760	8	12
clear	0	30	40	send-segment	64	8	12
compare	667,282	1	2	suspend	225	1	6
create-associative	429	35	45	test	150	1	2
create-read-write	1,541	5	6	test-count	48,054	1	2
destroy-segment	533,956	11	20	time	0	1	1
distance	2,428	8	8	times	1,066,416	1	2
divide	10,000	20	25	xor	0	1	2
initialize	65,810	1	2	average	6,843,848	2.5	4.4

Table 6.2: Pi Operation Cost Summary

The Pi operations **self**, **read**, and **write** are excluded from this table since their cost is included in other Pi operations. The pseudo-operations initialize, move, and test, are provided for cases where reads and writes occur in the absence of other operations.

Dyadic operations assume the case where the A operand is in a register and the B operand is in a segment (two cycles). This is the anticipated average.

The average presented in the table is weighted by the frequency of occurrence in the examples (dynamic average). The static average for all Pi operations is 4.6 PiMac instructions and 6.9 PiMac machine cycles.

Note that many Pi operations are supported directly by a PiMac instruction[4]. It was originally intended that each logical instruction (from Chapter 4) would be represented as a single PiMac machine instruction. However, some logical instructions, such as **create-associative-segment** and **send**, perturb the ratio to approximately 2.5 PiMac instructions per logical instruction.

The figures in this table are estimates. The exact costs will be determined by a machine dependent compiler running a typical mixture of applications.

Overall, PiMac provides reasonable support for the interface. With a 40 ns machine cycle time, PiMac executes approximately six million Pi operations per second (5.7 MPOPS).

The greatest unresolved issue is the hit rate of the segment buffers. In the gate array implementation of PiMac, going off-chip requires 16 memory cycles (720 ns). If the hit rate is not high, much of the performance gains of PiMac will be lost. It is likely that the 1000 words of on-chip (low latency) memory will be employed as a segment cache. Future research will provide the answers.

[4]This is not a general requirement of Pi substrates. The emphasis is on efficient support of parallel operations, not "one instruction per Pi operation".

Chapter 7

Conclusion

This thesis addresses the problem of monolithic parallel architectures. It defines Pi, a parallel architecture interface, to separate model and machine issues. The interface is based on several generic parallel model requirements: low latency communication, fast task switching, low cost synchronization, efficient storage management, the ability to exploit all types of locality, and the ability to efficiently support sequential code segments in a parallel environment. This foundation is the central contribution of this interface design.

The interface provides many features for model mechanism construction including:

- Linear-addressed segments (e.g., **read**, **write**)
- Associatively addressed segments (e.g., **insert**, **match**)
- Common synchronization primitives (data, barrier, producer-consumer)
- Communication, both for data transport and remote task invocation
- Task management, including prioritized scheduling, local or remote task execution, voluntary suspension, and variable argument passing
- A common namespace for storage, synchronizations, and tasks
- Topology-independent specification of locality (e.g., **another-node-id**, **distance**)
- Efficient sequential code sequences

The interface is used to construct several key model mechanisms including: shared memory (with caches), set synchronization, object name translation, and non-resident handler support. In addition, two applications which incorporate multiple model mechanisms are built: n-body and relaxation. The examples are executed using PiSim, a Pi simulation and metering environment. Their behavior and performance, in terms of basic parallel operations, is determined. Using composite statistics from all of the executions, a first-order profile of operation frequency is derived.

The information from these examples is used in the specification of PiMac, a Pi machine architecture. This design is also driven by the generic parallel model requirements identified earlier. The specific design goals for PiMac include: low latency communication, fast task switching (suspensions, queuing, and resumptions), multiple task/message queues, support for attributes (synchronization), efficient storage management, support for read/write and associative segments, and fast sequential execution.

PiMac is fine-grained, supporting a maximum of 65 thousand words of memory and 255 tasks per node. It provides fast microsequences for task suspension and dequeuing (~4 memory cycles for each). Segment buffers play a significant role in the support of several Pi features including read-write segments, associative segments, and attributes. PiMac also contains an autonomous network router which provides low latency communication with other nodes.

The design of PiMac has been significantly influenced by the Message Driven Processor. A gate-array implementation of PiMac is considered based on data from the MDP's design. The PiMac core is expected to require ~30,000 gates. Existing technology can easily accommodate this design, plus a modest-sized segment cache (1000 word) to further reduce memory latency.

Pi performance on PiMac is evaluated. Most Pi operations have direct and efficient realizations on PiMac. An unweighted average Pi operation requires 4.6 PiMac instructions and 6.9 PiMac machine cycles. Using the operation frequencies from the model mechanism examples, the weighted average instruction requires 2.5 PiMac instructions and 4.4 PiMac machine cycles. Based on this weighted average, a PiMac implementation with a 40 ns machine clock executes 5.7 million Pi operations per second.

Why Pi?

This thesis introduces Pi, a parallel architecture interface. Pi provides several benefits for parallel architectures including:

- A well-defined boundary separating model and machine issues
- A platform for combining multiple model mechanisms to address the requirements of specific applications
- An evaluation medium for architecture components with explicit parallel operation costs
- An efficient vehicle for delivering new machine techniques and technology to existing programming environments

A key feature of Pi is that it is designed specifically to be an architecture interface. It has no additional purposes that could compromise its effectiveness as an interface. This is not true of language-centered interfaces, machine-centered interfaces,

and theoretical models, which have other goals. Would a language designer remove a model-biasing feature if it made programming easier? Would a machine designer neglect a model-biasing mechanism if it provided a significant performance improvement? Using Pi, both the language and machine designer can pursue a single goal. The interface defines a context for their separate problems.

By focusing on basic parallel requirements, Pi provides a model and machine independent platform with which to build parallel systems. This lack of bias improves Pi's generality and increases its chances of compatibility with future models. No one knows what *the* programming model of the future will be, but it will probably share the same basic parallel requirements of existing models: communication, synchronization, locality, linear and associative storage, and sequential sequences.

In the introduction, four interface evaluation criteria were defined: interface abstractness, presentation accuracy, functional match, and implementation bias. In the course of this thesis, these criteria have guided the design of Pi. As model mechanisms were developed, Pi operations that restricted, or complicated implementation were removed and new ones were added. In the design of PiMac, the interface was examined from the other side and adjustments were made.

It is difficult to make a quantitative assessment of these criteria. However, through the evaluation and refinement in this thesis, Pi has achieved a successful balance between models and machines.

Future Directions

Pi is a starting point. There is a large agenda of future research including the following.

- In order to get a complete and accurate representation of parallel programs, programming language compilers must be written or modified to produce Pi programs. Using the output of these compilers, a larger and more varied set of examples can be collected. This also requires the construction of a complete set of model mechanisms to support each programming language. Executing these programs will provide statistics on model mechanism usage as well as Pi operations.

- A machine dependent compiler and operating system must be constructed to provide a more accurate measure of Pi operation costs. This is part of a larger study into the cost of supporting generic parallel operations. This research must also address several open research areas like resource management in a fine-grained execution environment. New modes of compiler and operating system interaction are required (e.g., post-execution recompilation using collected statistics).

- A PiMac prototype must be constructed to accomplish two goals: (a) to support execution of larger and more realistic programming examples, and (b) to provide

a performance incentive for people outside computer architecture who have problems they'd like to solve faster and big problems they can't solve at all. The latter goal is necessary if parallel computers are ever to become more than academic curiosity.

- New programming environments that take advantage of the multi-model capability of Pi must be developed. These environments allow problem solving paradigms to be selected based on the requirements of application, rather than machine or language availability.

Certainly parallel architecture interfaces will continue to evolve as this work progresses. This thesis advocates the use of an interface based on generic parallel model requirements. Pi is a first step towards that goal.

Bibliography

[1] William B. Ackerman. Data Flow Languages. *IEEE Computer*, 15(2):15–25, February 1982.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *The 15th Annual International Symposium on Computer Architecture.* IEEE Computer Society and ACM, June 1988.

[3] Gul A. Agha. *A Model of Concurrent Computation in Distributed Systems.* Artificial Intelligence Series. MIT Press, 1986.

[4] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica S. Lam, Onat Menzilcioglu, Ken Sarocky, and Jon A. Webb. Warp Architecture and Implementation. In *The 13th Annual International Symposium on Computer Architecture.* IEEE Computer Society and ACM, June 1986.

[5] Arvind and Rishiyur S. Nikhil. A Dataflow Approach to General-purpose Parallel Computing. Technical Report Computation Structures Group Memo 302, MIT Laboratory for Computer Science, July 1989.

[6] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data Structures for Parallel Computing. Technical Report 269, Computational Structures Group, MIT Laboratory for Computer Science, February 1987.

[7] Kenneth E. Batcher. The Massively Parallel Processor System Overview. In Jerry L. Potter, editor, *The Massively Parallel Processor*, pages 142–149. MIT Press, 1985.

[8] BBN, Advanced Computers, Inc., Cambridge, MA. *Butterfly Product Overview*, 1987.

[9] Roberto Bisiani and Alessandro Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, 37(8), August 1988.

[10] Guy E. Blelloch. AFL-1: A Programming Language for Massively Concurrent Computers. Technical Report 918, MIT Artificial Intelligence Laboratory, November 1986.

[11] Gregory T. Byrd and Bruce A. Delagi. Support for Fine-Grained Message Passing in Shared Memory Multiprocessors. Technical Report KSL-89-15, Knowledge Systems Laboratory, Stanford University, March 1989.

[12] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[13] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, 37(8), August 1988.

[14] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architectural of the Message-Driven Processor. In *The 14th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM, June 1987.

[15] William Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Peter Nuth, Jerry Larivee, and Brian Totty. Message-Driven Processor Architecture, Version 11. Technical Report AI Memo 1069, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, August 1988.

[16] William J. Dally. Wire Efficient VLSI Multiprocessor Communication Networks. In Paul Losleben, editor, *The Proceedings of the Stanford Conference on Advanced Research in VLSI*, pages 391–415, March 1987.

[17] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, C-39(6):775–785, June 1990.

[18] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keene, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A Fine Grain Concurrent Computer. In *IFIP Congress '89*, August 1989.

[19] William J. Dally and Andrew A. Chien. Object-Oriented Concurrent Programming in CST. In *The Third Conference on: Hypercube Concurrent Computers and Applications, Volume I - Architecture, Software, Computer Systems and General Issues*. ACM, January 1988.

[20] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnected Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[21] E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[22] K. Gharachorloo, D. Lenoski, J. Laudon, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.

In *The 17th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM, May 1990.

[23] V.G. Grafe, G.S. Davidson, J.E. Hoch, and V.P. Holmes. The Epsilon Dataflow Processor. In *The 16th Annual International Symposium on Computer Architecture*, pages 36–45. IEEE Computer Society and ACM, May 1989.

[24] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulation. Technical Report Research Report YALEU/DCS/RR-495, Yale University, April 1986.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[26] C.A.R. Hoare. Communicating Sequential Processors. *Communications of the ACM*, 21(8):666–677, August 1978.

[27] Waldemar Horwat. A Concurrent Smalltalk Compiler for the Message-Driven Processor. Technical Report AI Memo 1080, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1988.

[28] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Master's thesis, MIT Artificial Intelligence Laboratory, May 1989.

[29] Waldemar Horwat, Brian Totty, and William J. Dally. COSMOS: An Operating System for a Fine-Grain Concurrent Computer. *to appear*, May 1990.

[30] Robert Alan Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report 418, MIT Laboratory for Computer Science, July 1988.

[31] INMOS. *Transputer Reference Manual*, January 1987.

[32] Guy L. Steele Jr. *COMMON LISP: The Language.* Digital Press, 1984.

[33] R. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[34] J. S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputer and Its Application.* MIT Press, 1985.

[35] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, and M. Lam. Design of the Stanford DASH Multiprocessor. Technical Report CSL-89-403, Stanford University, Computer Systems Laboratory, December 1989.

[36] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *The 17th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM, May 1990.

[37] LSI Logic Corporation, Milpitas, CA. *1.5-Micron Compacted Array Technology Databook*, July 1987.

[38] LSI Logic Corporation, Milpitas, CA. *LCA100K Compacted Array Plus Series 0.7-Micron HCMOS*, 1989.

[39] LSI Logic Corporation, Milpitas, CA. *LEA100K Embedded Array Series*, 1989.

[40] LSI Logic Corporation, Milpitas, CA. *Shortform Catalog*, November 1989.

[41] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. Nial: A Candidate Language for Fifth Generation Computer Systems. In *Proceedings ACM'84 Annual Conference: The Fifth Generation Challenge*, pages 157–166. ACM, October 1984.

[42] W.J.A. Mol. On the choice of suitable operators and parameters in multigrid methods. Technical Report NW 107/81, Department of Numerical Mathematics, stichting mathematisch centrum, June 1981.

[43] NCUBE, Beaverton, Oregon. *NCUBE 2 6400 Series Supercomputer: Technical Overview*, 1989.

[44] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *The 16th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM, May 1989.

[45] Gregory Michael Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report 432, MIT Laboratory for Computer Science, December 1988.

[46] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings on the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE Computer Society and ACM, August 1985.

[47] Richard S. Piepho and William S. Wu. A Comparison of RISC Archiectures. *IEEE Micro*, 9(4):51–62, August 1989.

[48] Randall D. Rettberg, William R. Crowther, Philip P. Carvey, and Raymond S. Tomlinson. The Monarch Parallel Processor Hardware Design. *IEEE Computer*, 23(4):18–30, April 1990.

[49] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, 1988.

[50] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

[51] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[52] David Elliot Shaw. Architecture and Applications of a Heterogeneous, Massively Parallel Machine. In Joseph F. Traub, Barbara J. Grosz, and Butler W. Lampson Nils J. Nilsson, editors, *Annual Review of Computer Science*, volume 1, pages 139–151. Annual Reviews Inc., 1986.

[53] Burton Smith. The Architecture of HEP. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Application*, chapter 1, pages 41–55. MIT Press, 1985.

[54] Paul Y. Song. Design of a Network for Concurrent Message Passing Systems. Master's thesis, MIT Artificial Intelligence Laboratory, May 1988.

[55] Lawrence Synder. Type Architectures, Shared Memory, and the Corollary of Modest Potential. In Joseph F. Traub, Barbara J. Grosz, and Butler W. Lampson Nils J. Nilsson, editors, *Annual Review of Computer Science*, volume 1, pages 289–317. Annual Reviews Inc., 1986.

[56] Thinking Machines Corporation, Cambridge, MA. *Connection Machine Model CM-2 Technical Summary*, April 1987.

[57] Brian Totty. An OS Kernel for the Jellybean Machine: Version 1.0. Technical report, MIT Concurrent VLSI Architecture Group, August 1987.

[58] Leslie G. Valiant. Bulk-Synchronous Parallel Computers. Technical Report TR-08-89, Aiken Computation Laboratory, Harvard University, 1989.

[59] Feng Zhao. An O(N) Algorithm for Three-dimensional N-body Simulations. Technical Report 995, MIT Artificial Intelligence Laboratory, October 1987.

Appendix A

Example Source Code

This appendix contains the complete code for the examples described in chapter 3.

A.1 Common Handlers

```
;;                      C o m m o n
;;
;;        Scott Wills                                  14 September 1989
;;
;;   These are common handlers for the Pi examples.
(in-package 'user)
;; These constants define offsets to the Length, Priority and
;; Task-Type fields in a task segment.
(define-constant LENGTH 0)
(define-constant PRIORITY 1)
(define-constant TASK-TYPE 2)
(define-constant LOCALS-OFFSET 3)
;; This handler stores a value in a read-write segment at a specified
;; offset.
(define-handler Reply-Value (Value Segment Offset) ()
  (write (read (self) Segment) (read (self) Offset) (read (self) Value))
  (destroy-segment (self)))
;; This handler adjusts a barrier by a count.
(define-handler Adjust-Barrier (Count-Change Segment Offset) ()
  (Adjust-Count (read (self) Segment) (read (self) Offset)
                (read (self) Count-Change))
  (destroy-segment (self)))
;; This handler adds a value to a location and reduces a b-sync.  Both
;; the add location and b-sync must be in the same read-write segment.
(define-handler Add-And-Reduce (Add-Value Segment B-Sync-Offset
                                Add-Offset) ()
  (write (read (self) Segment)
         (read (self) Add-Offset)
         (plus (read (read (self) Segment) (read (self) Add-Offset))
```

```
            (read (self) Add-Value)))
    (Adjust-Count (read (self) Segment) (read (self) B-Sync-Offset) -1)
    (destroy-segment (self)))
```

A.2 Shared Memory With Caches

```
;;                      S h a r e d   M e m o r y
;;
;;        Scott Wills                              9 October 1989
;;
;;   This is shared memory model example with caches.
;;
;;   There are two protocols in this example.  One is for a line of a
;;   chunk.  The other is for a line of a cache.  Both protocols are
;;   defined below.
;;
;; Chunk Protocol
;; ----------------
;;
;; Chunk States: UNLOCKED, READ-ONLY, READ-WRITE, LOCKED-REMOVE,
;; LOCKED-READ, LOCKED-WRITE, LOCKED-FLUSH-READ, LOCKED-FLUSH-WRITE,
;; LOCKED-INVALIDATE-WRITE, ERROR
;;
;; Chunk Handlers: Read, Write, Remove Ack, Ack-Flush
;;
;; UNLOCKED
;;    Read: send Reply; goto LOCKED-READ.
;;    Write: send Reply; goto LOCKED-WRITE.
;;    Remove: goto ERROR.
;;    Ack: goto ERROR.
;;    Ack-Flush: goto ERROR.
;;
;; READ-ONLY
;;    Read: send Reply; add to chain; goto LOCKED-READ.
;;    Write: send Invalidate; goto LOCKED-INVALIDATE-WRITE.
;;    Remove: send Invalidate; goto LOCKED-REMOVE.
;;    Ack: goto ERROR.
;;    Ack-Flush: goto ERROR.
;;
;; READ-WRITE
;;    Read: send Flush; goto LOCKED-FLUSH-READ.
;;    Write: send Flush; goto LOCKED-FLUSH-WRITE.
;;    Remove: send Flush; goto LOCKED-REMOVE.
;;    Ack: goto ERROR.
;;    Ack-Flush: goto ERROR.
;;
;; LOCKED-REMOVE
;;    Read: suspend task; goto LOCKED-REMOVE.
```

```
;;    Write: suspend task; goto LOCKED-REMOVE.
;;    Remove: suspend task; goto LOCKED-REMOVE.
;;    Ack: requeue suspended tasks; goto UNLOCKED.
;;    Ack-Flush: update line; requeue suspended tasks; goto UNLOCKED.
;;
;; LOCKED-READ
;;    Read: suspend task; goto LOCKED-READ.
;;    Write: suspend task; goto LOCKED-READ.
;;    Remove: suspend task; goto LOCKED-READ.
;;    Ack: requeue suspended tasks; goto READ-ONLY.
;;    Ack-Flush: goto ERROR.
;;
;; LOCKED-WRITE
;;    Read: suspend task; goto LOCKED-WRITE.
;;    Write: suspend task; goto LOCKED-WRITE.
;;    Remove: suspend task; goto LOCKED-WRITE.
;;    Ack: requeue suspended tasks; goto READ-WRITE.
;;    Ack-Flush: goto ERROR.
;;
;; LOCKED-FLUSH-READ
;;    Read: suspend task; goto LOCKED-FLUSH-READ.
;;    Write: suspend task; goto LOCKED-FLUSH-READ.
;;    Remove: suspend task; goto LOCKED-FLUSH-READ.
;;    Ack: goto ERROR.
;;    Ack-Flush:
;;
;; LOCKED-FLUSH-WRITE
;;    Read: suspend task; goto LOCKED-FLUSH-WRITE.
;;    Write: suspend task; goto LOCKED-FLUSH-WRITE.
;;    Remove: suspend task; goto LOCKED-FLUSH-WRITE.
;;    Ack: goto ERROR.
;;    Ack-Flush: update line; goto LOCKED-WRITE.
;;
;; LOCKED-INVALIDATE-WRITE
;;    Read: suspend task; goto LOCKED-INVALIDATE-WRITE.
;;    Write: suspend task; goto LOCKED-INVALIDATE-WRITE.
;;    Remove: suspend task; goto LOCKED-INVALIDATE-WRITE.
;;    Ack: goto LOCKED-WRITE.
;;    Ack-Flush: goto ERROR.
;;
;;  Cache Protocol
;; ----------------
;;
;; Cache States: INVALID, READ-ONLY, READ-WRITE, LOCKED-READ,
;; LOCKED-WRITE, LOCKED-READ-ONLY-READ, LOCKED-READ-ONLY-WRITE,
;; LOCKED-READ-WRITE-READ, LOCKED-READ-WRITE-WRITE, ERROR
;;
;; Cache Handlers: Read-Same, Write-Same, Read-Different,
;; Write-Different, Reply, Invalidate, Flush
;;
;; INVALID
;;    Read-Same: send Read; goto LOCKED-READ.
;;    Write-Same: send Write; goto LOCKED-WRITE.
;;    Read-Different: send Read; goto LOCKED-READ.
```

```
;;    Write-Different: send Write; goto LOCKED-WRITE.
;;    Reply: goto ERROR.
;;    Invalidate: goto ERROR.
;;    Flush: goto ERROR.
;;
;; READ-ONLY
;;    Read-Same: read value; goto READ-ONLY.
;;    Write-Same: send remove; goto LOCKED-READ-ONLY-WRITE.
;;    Read-Different: send remove; goto LOCKED-READ-ONLY-READ.
;;    Write-Different: send remove; goto LOCKED-READ-ONLY-WRITE.
;;    Reply: goto ERROR.
;;    Invalidate: send Invalidate or Ack; goto INVALID.
;;    Flush: goto ERROR.
;;
;; READ-WRITE
;;    Read-Same: read value; goto READ-WRITE.
;;    Write-Same: write value; goto READ-WRITE.
;;    Read-Different: send remove; goto LOCKED-READ-WRITE-READ.
;;    Write-Different: send remove; goto LOCKED-READ-WRITE-WRITE.
;;    Reply: goto ERROR.
;;    Invalidate: goto ERROR.
;;    Flush: send Ack-Flush; goto INVALID.
;;
;; LOCKED-READ
;;    Read-Same: suspend task; goto LOCKED-READ.
;;    Write-Same: suspend task; goto LOCKED-READ.
;;    Read-Different: suspend task; goto LOCKED-READ.
;;    Write-Different: suspend task; goto LOCKED-READ.
;;    Reply: send Ack; update line; requeue suspended tasks; goto READ-ONLY.
;;    Invalidate: goto ERROR.
;;    Flush: goto ERROR.
;;
;; LOCKED-WRITE
;;    Read-Same: suspend task; goto LOCKED-WRITE.
;;    Write-Same: suspend task; goto LOCKED-WRITE.
;;    Read-Different: suspend task; goto LOCKED-WRITE.
;;    Write-Different: suspend task; goto LOCKED-WRITE.
;;    Reply: send Ack; update cache; requeue suspended tasks;
;;           goto READ-WRITE.
;;    Invalidate: goto ERROR.
;;    Flush: goto ERROR.
;;
;; LOCKED-READ-ONLY-READ
;;    Read-Same: read value; goto LOCKED-READ-ONLY-READ.
;;    Write-Same: suspend task; goto LOCKED-READ-ONLY-READ.
;;    Read-Different: suspend task; goto LOCKED-READ-ONLY-READ.
;;    Write-Different: suspend task; goto LOCKED-READ-ONLY-READ.
;;    Reply: goto ERROR.
;;    Invalidate: send Ack; change name; goto LOCKED-READ.
;;    Flush: goto ERROR.
;;
;; LOCKED-READ-ONLY-WRITE
;;    Read-Same: read value; goto LOCKED-READ-ONLY-WRITE.
;;    Write-Same: suspend task; goto LOCKED-READ-ONLY-WRITE.
```

```
;;    Read-Different: suspend task; goto LOCKED-READ-ONLY-WRITE.
;;    Write-Different: suspend task; goto LOCKED-READ-ONLY-WRITE.
;;    Reply: goto ERROR.
;;    Invalidate: send Ack; change name; goto LOCKED-WRITE.
;;    Flush: goto ERROR.
;;
;; LOCKED-READ-WRITE-READ
;;    Read-Same: read value; goto LOCKED-READ-WRITE-READ.
;;    Write-Same: write value; goto LOCKED-READ-WRITE-READ.
;;    Read-Different: suspend task; goto LOCKED-READ-WRITE-READ.
;;    Write-Different: suspend task; goto LOCKED-READ-WRITE-READ.
;;    Reply: goto ERROR.
;;    Invalidate: goto ERROR.
;;    Flush: send Ack-Flush; change name; goto LOCKED-READ.
;;
;; LOCKED-READ-WRITE-WRITE
;;    Read-Same: read value; goto LOCKED-READ-WRITE-WRITE.
;;    Write-Same: write value; goto LOCKED-READ-WRITE-WRITE.
;;    Read-Different: suspend task; goto LOCKED-READ-WRITE-WRITE.
;;    Write-Different: suspend task; goto LOCKED-READ-WRITE-WRITE.
;;    Reply: goto ERROR.
;;    Invalidate: goto ERROR.
;;    Flush: send Ack-Flush; change name; goto LOCKED-WRITE.
(in-package 'user)
;; This constant defines the number of nodes.
(define-constant Number-Of-Nodes 64)
;; This constant defines the number of lines in a node chunk.
(define-constant Chunk-Lines 8)
;; This constant defines the base mask (minus the line offset).
(define-constant Base-Mask 4088) ;; (4096 - 8)
;; This constant defines the offset mask (minus the line base).
(define-constant Offset-Mask 7)
;; This constant defines the shift to move the node to the bottom of
;; the word.
(define-constant Node-Shift -6)
;; This constant defines the node mask.
(define-constant Node-Mask 63)
;; This constant defines the shift to move the index to the bottom of the
;; word.
(define-constant Index-Shift -3)
;; This constant defines the index mask.
(define-constant Index-Mask 7)
;; This constant defines the size of the cache map.
(define-constant Cache-Map-Size 16)
;; These constants specify offsets into a line.
(define-constant Store-Offset 0)
(define-constant Status-Offset 8)
(define-constant Link-Offset 9)
(define-constant Request-Offset 10)
(define-constant Barrier-Offset 11)
(define-constant Line-Size 12)
;; This nodal vectors the map of chunk objects on a node.
(define-nodal Chunk-Map)
;; This nodal vectors the cache map.
```

```
(define-nodal Cache-Map)
;; This handler initializes shared memory in the entire system.
(define-handler Shared-Memory-Setup () (Barrier Index)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier Number-Of-Nodes)
  (write (self) Index Number-Of-Nodes)
  Loop
  (write (self) Index (minus (read (self) Index) 1))
  (send (read (self) Index) 9 NORMAL Node-Setup (node-id) (self) Barrier)
  (branch-not-zero (read (self) Index) Loop)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;; This handler sets up a node to support a shared memory model.  It
;; allocates and initializes the shared memory chunk and cache
;; storage.
(define-handler Node-Setup (Ack-Node Ack-Segment Ack-Offset)
                           (Index-A Index-B Line)
  (write (nodals) Chunk-Map (create-read-write-segment Chunk-Lines))
  (write (self) Index-A Chunk-Lines)
  Loop-1
  (write (self) Index-A (minus (read (self) Index-A) 1))
  (write (self) Line (create-read-write-segment Line-Size))
  (write (self) Index-B (plus Store-Offset Line-Size))
  Loop-2
  (write (self) Index-B (minus (read (self) Index-B) 1))
  (write (read (self) Line) (read (self) Index-B) 0)
  (branch-not-zero (compare (read (self) Index-B) Store-Offset) Loop-2)
  (write (read (self) Line) Status-Offset UNLOCKED)
  (write (read (self) Line) Link-Offset END)
  (attribute (read (self) Line) Barrier-Offset B-SYNC)
  (write (read (nodals) Chunk-Map) (read (self) Index-A)
         (read (self) Line))
  (branch-not-zero (read (self) Index-A) Loop-1)
  (write (nodals) Cache-Map
         (create-associative-segment Cache-Map-Size BOUND-SAFE))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;; This handler executes a "read" code fragment.
(define-handler Read-Address (Address) (Base Offset Line)
  (write (self) Base (and (read (self) Address) Base-Mask))
  (write (self) Offset (and (read (self) Address) Offset-Mask))
  (write (self) Line (match (read (nodals) Cache-Map) (read (self) Base)))
  (branch-zero (compare (read (self) Line) UNBOUND) Get-Line)
  (branch-zero (compare (read (read (self) Line) Status-Offset) INVALID)
                  Get-Line)
  (return (read (read (self) Line) (read (self) Offset)))
  (destroy-segment (self))
  Get-Line
  (write (self) Line (call 8 Get-Cache-Line (read (self) Base) READ))
  Read-Line
  (return (read (read (self) Line) (read (self) Offset)))
  (attribute (read (self) Line) Request-Offset READ-WRITE)
  (destroy-segment (self)))
```

```
;; This handler executes a "write" code fragment.
(define-handler Write-Address (Address Value) (Base Offset Line)
  (write (self) Base (and (read (self) Address) Base-Mask))
  (write (self) Offset (and (read (self) Address) Offset-Mask))
  (write (self) Line (match (read (nodals) Cache-Map) (read (self) Base)))
  (branch-zero (compare (read (self) Line) UNBOUND) Get-Line)
  (branch-not-zero (compare (read (read (self) Line) Status-Offset)
                   READ-WRITE) Get-Line)
  (write (read (self) Line) (read (self) Offset) (read (self) Value))
  (destroy-segment (self))
  Get-Line
  (write (self) Line (call 8 Get-Cache-Line (read (self) Base) WRITE))
  (write (read (self) Line) (read (self) Offset) (read (self) Value))
  (attribute (read (self) Line) Request-Offset READ-WRITE)
  (destroy-segment (self)))
;; This handler loads a line into the cache in the specified mode.
;; It may be necessary to remove an existing cache line first.  It
;; may also be necessary to create an empty cache line if no cache
;; exits.
(define-handler Get-Cache-Line (Base Mode) (Current-Base Node Line)
  (write (self) Current-Base
         (insert (read (nodals) Cache-Map) (read (self) Base) UNBOUND))
  (write (self) Line
         (match (read (nodals) Cache-Map) (read (self) Current-Base)))
  (branch-not-zero (compare (read (self) Line) UNBOUND) Remove-Line)
  (write (self) Line (create-read-write-segment Line-Size))
  (write (read (self) Line) Status-Offset INVALID)
  (insert (read (nodals) Cache-Map) (read (self) Base) (read (self) Line))
  Remove-Line
  (write (read (self) Line) Request-Offset (read (self) Base))
  (attribute (read (self) Line) Request-Offset READ-ONLY)
  (write (self) Current-Base
         (insert (read (nodals) Cache-Map) (read (self) Base) UNBOUND))
  (branch-zero (compare (read (read (self) Line) Status-Offset) INVALID)
               Get-Line)
  (attribute (read (self) Line) Barrier-Offset B-SYNC)
  (adjust-count (read (self) Line) Barrier-Offset 1)
  (write (self) Node (a-shift (read (self) Current-Base) Node-Shift))
  (write (self) Node (and (read (self) Node) Node-Mask))
  (send (read (self) Node) 6 NORMAL Remove (read (self) Current-Base))
  (test-count (read (self) Line) Barrier-Offset)
  Get-Line
  (remove (read (nodals) Cache-Map) (read (self) Current-Base))
  (insert (read (nodals) Cache-Map) (read (self) Base) (read (self) Line))
  (attribute (read (self) Line) Barrier-Offset B-SYNC)
  (adjust-count (read (self) Line) Barrier-Offset 1)
  (write (self) Node (a-shift (read (self) Base) Node-Shift))
  (write (self) Node (and (read (self) Node) Node-Mask))
  (send (read (self) Node) 7 NORMAL (read (self) Mode)
        (read (self) Base) (node-id))
  (test-count (read (self) Line) Barrier-Offset)
  (return (read (self) Line))
  (destroy-segment (self)))
;;============================================================================
```

```
;;                        Chunk Handlers
;; This handler reads a line from chunk memory.  First the chunk line
;; status is tested, and the lock is acquired.  If the status is
;; READ-WRITE, the exclusive copy is flushed.  Then a READ-ONLY copy
;; is sent to the requesting node, and it is pushed on the READ-ONLY
;; chain.  When the acknowledgment is received, the chunk line is
;; unlocked.
(define-handler Read (Base Reply-Node) (Index Old-Status)
  (write (self) Index (a-shift (read (self) Base) Index-Shift))
  (write (self) Index (and (read (self) Index) Index-Mask))
  (write (self) Index (read (read (nodals) Chunk-Map) (read (self) Index)))
  (write (self) Old-Status (read (read (self) Index) Status-Offset))
  (attribute (read (self) Index) Status-Offset D-SYNC)
  (branch-not-zero (compare (read (self) Old-Status) READ-WRITE) Skip-Flush)
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (read (self) Index) Link-Offset) 6 NORMAL Flush
        (read (self) Base))
  (test-count (read (self) Index) Barrier-Offset)
  Skip-Flush
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (self) Reply-Node) 16 NORMAL Reply
        (read (read (self) Index) 0) (read (read (self) Index) 1)
        (read (read (self) Index) 2) (read (read (self) Index) 3)
        (read (read (self) Index) 4) (read (read (self) Index) 5)
        (read (read (self) Index) 6) (read (read (self) Index) 7)
        READ-ONLY (read (read (self) Index) Link-Offset) (read (self) Base))
  (write (read (self) Index) Link-Offset (read (self) Reply-Node))
  (test-count (read (self) Index) Barrier-Offset)
  (write (read (self) Index) Status-Offset READ-ONLY)
  (destroy-segment (self)))
;; This handler replies an exclusive READ-WRITE copy of a line.  First
;; the chunk line status is tests, and the lock is acquired.  If the
;; status is READ-WRITE, the exclusive copy is flushed.  If the
;; status is READ-ONLY, the READ-ONLY chain is invalidated.  Then an
;; exclusive READ-WRITE copy is sent to the requesting node, and the
;; link field is set.  When the acknowledgment is received, the
;; chunk line is unlocked.
(define-handler Write (Base Reply-Node) (Index Old-Status)
  (write (self) Index (a-shift (read (self) Base) Index-Shift))
  (write (self) Index (and (read (self) Index) Index-Mask))
  (write (self) Index (read (read (nodals) Chunk-Map) (read (self) Index)))
  (write (self) Old-Status (read (read (self) Index) Status-Offset))
  (attribute (read (self) Index) Status-Offset D-SYNC)
  (branch-not-zero (compare (read (self) Old-Status) READ-WRITE) Skip-Flush)
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (read (self) Index) Link-Offset) 6 NORMAL Flush
        (read (self) Base))
  (test-count (read (self) Index) Barrier-Offset)
  Skip-Flush
  (branch-not-zero (compare (read (self) Old-Status) READ-ONLY)
                   Skip-Invalidate)
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (read (self) Index) Link-Offset) 6 NORMAL Invalidate
        (read (self) Base))
```

```
  (test-count (read (self) Index) Barrier-Offset)
  Skip-Invalidate
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (self) Reply-Node) 16 NORMAL Reply
        (read (read (self) Index) 0) (read (read (self) Index) 1)
        (read (read (self) Index) 2) (read (read (self) Index) 3)
        (read (read (self) Index) 4) (read (read (self) Index) 5)
        (read (read (self) Index) 6) (read (read (self) Index) 7)
        READ-WRITE END (read (self) Base))
  (write (read (self) Index) Link-Offset (read (self) Reply-Node))
  (test-count (read (self) Index) Barrier-Offset)
  (write (read (self) Index) Status-Offset READ-WRITE)
  (destroy-segment (self)))
;; This handler removes a cache copy from a node.  If the line status
;; is UNLOCKED, no action is taken.  If the status is READ-ONLY, all
;; read-only copies are invalidated, and the link is set to END.  If the
;; status is READ-WRITE, the exclusive copy is flushed, and the link is set
;; to END.
(define-handler Remove (Base) (Index Old-Status)
  (write (self) Index (a-shift (read (self) Base) Index-Shift))
  (write (self) Index (and (read (self) Index) Index-Mask))
  (write (self) Index (read (read (nodals) Chunk-Map) (read (self) Index)))
  (write (self) Old-Status (read (read (self) Index) Status-Offset))
  (attribute (read (self) Index) Status-Offset D-SYNC)
  (branch-not-zero (compare (read (self) Old-Status) READ-WRITE) Skip-Flush)
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (read (self) Index) Link-Offset) 6 NORMAL Flush
        (read (self) Base))
  (test-count (read (self) Index) Barrier-Offset)
  Skip-Flush
  (branch-not-zero (compare (read (self) Old-Status) READ-ONLY)
                   Skip-Invalidate)
  (adjust-count (read (self) Index) Barrier-Offset 1)
  (send (read (read (self) Index) Link-Offset) 6 NORMAL Invalidate
        (read (self) Base))
  (test-count (read (self) Index) Barrier-Offset)
  Skip-Invalidate
  (write (read (self) Index) Link-Offset END)
  (write (read (self) Index) Status-Offset UNLOCKED)
  (destroy-segment (self)))
;; This handler acknowledges a Reply or Invalidate, and adjusts the
;; chunk barrier.
(define-handler Ack (Base) (Index)
  (write (self) Index (a-shift (read (self) Base) Index-Shift))
  (write (self) Index (and (read (self) Index) Index-Mask))
  (write (self) Index (read (read (nodals) Chunk-Map) (read (self) Index)))
  (adjust-count (read (self) Index) Barrier-Offset -1)
  (destroy-segment (self)))
;; This handler updates the chunk data from a flush acknowledgment.
;; It then adjusts the chunk barrier.
(define-handler Ack-Flush (Word-0 Word-1 Word-2 Word-3 Word-4 Word-5 Word-6
                           Word-7 Base) (Index)
  (write (self) Index (a-shift (read (self) Base) Index-Shift))
  (write (self) Index (and (read (self) Index) Index-Mask))
```

```
  (write (self) Index (read (read (nodals) Chunk-Map) (read (self) Index)))
  (write (read (self) Index) 0 (read (self) Word-0))
  (write (read (self) Index) 1 (read (self) Word-1))
  (write (read (self) Index) 2 (read (self) Word-2))
  (write (read (self) Index) 3 (read (self) Word-3))
  (write (read (self) Index) 4 (read (self) Word-4))
  (write (read (self) Index) 5 (read (self) Word-5))
  (write (read (self) Index) 6 (read (self) Word-6))
  (write (read (self) Index) 7 (read (self) Word-7))
  (write (read (self) Index) Link-Offset END)
  (adjust-count (read (self) Index) Barrier-Offset -1)
  (destroy-segment (self)))
;;=============================================================================
;;                        Cache Handlers
;; This handler stores a line (ten words) into a cache-line.  It then
;; adjusts the cache line barrier.
(define-handler Reply (Word-0 Word-1 Word-2 Word-3 Word-4 Word-5 Word-6
                       Word-7 Status Link Base) (Node Line)
  (write (self) Node (a-shift (read (self) Base) Node-Shift))
  (write (self) Node (and (read (self) Node) Node-Mask))
  (send (read (self) Node) 5 NORMAL Ack (read (self) Base))
  (write (self) Line (match (read (nodals) Cache-Map) (read (self) Base)))
  (write (read (self) Line) 0 (read (self) Word-0))
  (write (read (self) Line) 1 (read (self) Word-1))
  (write (read (self) Line) 2 (read (self) Word-2))
  (write (read (self) Line) 3 (read (self) Word-3))
  (write (read (self) Line) 4 (read (self) Word-4))
  (write (read (self) Line) 5 (read (self) Word-5))
  (write (read (self) Line) 6 (read (self) Word-6))
  (write (read (self) Line) 7 (read (self) Word-7))
  (write (read (self) Line) Status-Offset (read (self) Status))
  (write (read (self) Line) Link-Offset (read (self) Link))
  (adjust-count (read (self) Line) Barrier-Offset -1)
  (destroy-segment (self)))
;; This handler flushes an exclusive read-write cache copy and
;; invalidates the entry.  It then adjusts the cache line barrier.
(define-handler Flush (Base) (Node Line)
  (write (self) Node (a-shift (read (self) Base) Node-Shift))
  (write (self) Node (and (read (self) Node) Node-Mask))
  (write (self) Line (match (read (nodals) Cache-Map) (read (self) Base)))
  (write (read (self) Line) Status-Offset INVALID)
  (send (read (self) Node) 13 NORMAL Ack-Flush
        (read (read (self) Line) 0) (read (read (self) Line) 1)
        (read (read (self) Line) 2) (read (read (self) Line) 3)
        (read (read (self) Line) 4) (read (read (self) Line) 5)
        (read (read (self) Line) 6) (read (read (self) Line) 7)
        (read (self) Base))
  (adjust-count (read (self) Line) Barrier-Offset -1)
  (destroy-segment (self)))
;; This handler invalidates a read-only cache copy entry.  If the
;; cache link is END, and Ack is returned to the chunk.  Otherwise,
;; the invalidate is forwarded to the next cache copy in the list.
;; The cache line barrier is also adjusted.
(define-handler Invalidate (Base) (Node Line)
```

```
  (write (self) Line (match (read (nodals) Cache-Map) (read (self) Base)))
  (write (read (self) Line) Status-Offset INVALID)
  (adjust-count (read (self) Line) Barrier-Offset -1)
  (branch-zero (compare (read (read (self) Line) Link-Offset) END) Ack-Chunk)
  (send (read (read (self) Line) Link-Offset) 6 NORMAL Invalidate
        (read (self) Base))
  (destroy-segment (self))
  Ack-Chunk
  (write (self) Node (a-shift (read (self) Base) Node-Shift))
  (write (self) Node (and (read (self) Node) Node-Mask))
  (send (read (self) Node) 5 NORMAL Ack (read (self) Base))
  (destroy-segment (self)))
```

A.3 Shared Memory With Address Braiding

```
;;                      S h a r e d   M e m o r y
;;
;;         Scott Wills                                  14 September 1989
;;
;;   This is shared memory mechanism (without caches) writen in Pi.
;;   It demonstrates address braiding.
(in-package 'user)
;; This constant defines the number of nodes.
(define-constant Number-Of-Nodes 64)
;; This is the node variable that holds the pointer to the shared memory
;; space.
(define-nodal SM-Space)
;; This handler sets up a node to support a shared memory model (2,3).
;; It preallocates all of the required storage in a read-write
;; segment.  The segment is vectored by the nodal: SM-Space.
;; This handler initializes shared memory in the entire system.
(define-handler Shared-Memory-Setup () (Barrier Index)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier Number-Of-Nodes)
  (write (self) Index Number-Of-Nodes)
  Loop
  (write (self) Index (minus (read (self) Index) 1))
  (send (read (self) Index) 7 NORMAL Node-Setup (node-id) (self) Barrier)
  (branch-not-zero (read (self) Index) Loop)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;; This handler preallocates 1024 words of storage (64 16 word blocks) in a
;; segment vectored by the node variable SM-Space.
(define-handler Node-Setup (Ack-Node Ack-Segment Ack-Offset) (Segment)
  (write (self) Segment (create-read-write-segment 1024))
  (write (nodals) SM-Space (read (self) Segment))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
```

```
  (destroy-segment (self)))
;; This handler reads a value and replies the content.
(define-handler Read (Offset Reply-Node Reply-Segment Reply-Offset) ()
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value
        (read (read (nodals) SM-Space) (read (self) Offset))
        (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
;; This handler writes a value.  An acknowledgment is returned.
(define-handler Write (Offset Value Ack-Node Ack-Segment Ack-Offset) ()
  (write (read (nodals) SM-Space) (read (self) Offset) (read (self) Value))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;;         Shared Memory Two
;;
;;  This version of shared memory is the most basic.  It uses absolute
;; addressing of preallocated shared memory space.
;; This handler support a simple shared memory model.  No caches yet!
;; An address is broken into two parts.
;;
;; |--- node (6) ---|--- offset (10) ---|
;;
;; The address space is composed of up to 64 chunks, each containing
;; 1024 words.
;; This handler executes a "read" code fragment.
(define-handler Read-Address-2 (Address) (Offset Node Value)
  (write (self) Offset (and (read (self) Address) 1023))
  (write (self) Node (a-shift (read (self) Address) -10))
  (write (self) Node (and (read (self) Node) 63))
  (attribute (self) Value WRITE-ONCE)
  (send (read (self) Node) 7 NORMAL Read (read (self) Offset)
        (node-id) (self) Value)
  (return (read (self) Value))
  (destroy-segment (self)))
;; This handler executes a "write" code fragment.
(define-handler Write-Address-2 (Address Value) (Barrier Offset Node)
  (write (self) Offset (and (read (self) Address) 1023))
  (write (self) Node (a-shift (read (self) Address) -10))
  (write (self) Node (and (read (self) Node) 63))
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 1)
  (send (read (self) Node) 8 NORMAL Write (read (self) Offset)
        (read (self) Value) (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;;         Shared Memory Three
;;
;; This version of shared memory is similar to the one above, but it
;; uses a braided shared memory address.
;;
;; This handler support a simple shared memory model.  No caches yet!
;; An address is broken into three parts:
;;
;; |--- block (6) ---|--- node (6) ---|--- offset (4) ---|
```

```
;;
;; The address space is composed of up to 64 blocks.  Each block is spread
;; across all 64 nodes.  Each piece of a block is 16 words long.
;; This handler executes a "read" code fragment.
(define-handler Read-Address (Address) (Offset Node Value)
  (write (self) Value (and (read (self) Address) 15))
  (write (self) Offset (a-shift (read (self) Address) -6))
  (write (self) Offset (and (read (self) Offset) 1008))
  (write (self) Offset (or (read (self) Offset) (read (self) Value)))
  (write (self) Node (a-shift (read (self) Address) -4))
  (write (self) Node (and (read (self) Node) 63))
  (attribute (self) Value WRITE-ONCE)
  (send (read (self) Node) 7 NORMAL Read (read (self) Offset)
        (node-id) (self) Value)
  (return (read (self) Value))
  (destroy-segment (self)))
;; This handler executes a "write" code fragment.
(define-handler Write-Address (Address Value) (Barrier Offset Node)
  (write (self) Node (and (read (self) Address) 15))
  (write (self) Offset (a-shift (read (self) Address) -6))
  (write (self) Offset (and (read (self) Offset) 1008))
  (write (self) Offset (or (read (self) Offset) (read (self) Node)))
  (write (self) Node (a-shift (read (self) Address) -4))
  (write (self) Node (and (read (self) Node) 63))
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 1)
  (send (read (self) Node) 8 NORMAL Write (read (self) Offset)
        (read (self) Value) (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (destroy-segment (self)))
```

A.4 Shared Memory Tests

```
;;                 S h a r e d   M e m o r y   T e s t s
;;
;;         Scott Wills                                9 October 1989
;;
;;   This file contains tests for the shared memory mechanisms.
(in-package 'user)
;; This constant is a multiple of the cache stride
(define-constant Stride 16)
;; This handler executes several tests on shared memory.
(define-handler Test () ()
  (call 5 Shared-Memory-Setup)
  (print-user "~&shared memory initialized~&")
  (send (another-node-id ANY) 6 NORMAL Test-1 1100)
  (send (another-node-id ANY) 6 NORMAL Test-2 1200)
  (send (another-node-id ANY) 5 NORMAL Test-3 1300)
```

```
  (send (another-node-id ANY) 8 NORMAL Test-4 1400)
  (send (another-node-id ANY) 6 NORMAL Test-5 1500)
  (call 8 Test-6 0 256)
  (send (another-node-id ANY) 8 NORMAL Test-7 0 4096)
  (destroy-segment (self)))
;; This handler performs a read on a remote node.  The value is returned.
(define-handler Remote-Read (Address Reply-Node Reply-Segment Reply-Offset)
                            (Value)
  (write (self) Value (call 7 Read-Address (read (self) Address)))
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value (read (self) Value)
        (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
;; This handler performs a write on a remote node.  The barrier is reduced.
(define-handler Remote-Write (Address Value Ack-Node Ack-Segment Ack-Offset)
                             ()
  (call 8 Write-Address (read (self) Address) (read (self) Value))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;; This handler performs a test on a node.  The barrier is reduced.
(define-handler Remote-Test (Address Value Test
                             Ack-Node Ack-Segment Ack-Offset) ()
  (call 7 Test-Location (read (self) Address) (read (self) Value)
        (read (self) Test))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;; This handler tests a memory location for a specified value.
(define-handler Test-Location (Address Value Test) (Test-Value)
  (write (self) Test-Value (call 7 Read-Address (read (self) Address)))
  (branch-zero (compare (read (self) Test-Value) (read (self) Value))
               Skip-Error)
  (print-user "~&ERROR: Test ~d: ~d <> ~d~&" (read (self) Test)
              (read (self) Test-Value) (read (self) Value))
  Skip-Error
  (destroy-segment (self)))
;; This handler prints counts on hundred boundaries, and new lines of
;; thousand boundaries.
(define-handler Print-Count (Count) ()
  (branch-not-zero (mod (read (self) Count) 1000) Skip-A)
  (print-user "~&" (read (self) Count))
  Skip-A
  (branch-not-zero (mod (read (self) Count) 100) Skip-B)
  (print-user "~4d " (read (self) Count))
  Skip-B
  (destroy-segment (self)))
;; This test remotely writes a location, then reads it locally, then writes
;; it locally.
(define-handler Test-1 (Address) (Barrier Result)
  (print-user "~&Test 1: read a remote write, invalidate READ-ONLY cache~&")
  (attribute (self) Barrier B-SYNC)
  (Adjust-Count (self) Barrier 1)
  (send (another-node-id FAR) 8 NORMAL Remote-Write
        (read (self) Address) 11111 (node-id) (self) Barrier)
```

```
  (test-count (self) Barrier)
  (call 7 Test-Location (read (self) Address) 11111 1)
  (call 8 Write-Address (read (self) Address) 111110)
  (call 7 Test-Location (read (self) Address) 111110 1)
  (destroy-segment (self)))
;; This tests writes a location, reads a location that results in a cache
;; replacement, then reads the first location back.
(define-handler Test-2 (Address) (Result Barrier)
  (print-user "~&Test 2: same cache flush~&")
  (call 8 Write-Address (read (self) Address) 22222)
  (write (self) Address (plus (read (self) Address) Stride))
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 1)
  (send (another-node-id ANY) 8 NORMAL Remote-Write
        (read (self) Address) 20202 (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (call 7 Read-Address (read (self) Address))
  (write (self) Address (minus (read (self) Address) Stride))
  (call 7 Test-Location (read (self) Address) 22222 2)
  (destroy-segment (self)))
;; This test locally writes a location, then reads it remotely on
;; several nodes.
(define-handler Test-3 (Address) (Barrier)
  (print-user "~&Test 3: multiple remote reads of a local write~&")
  (call 8 Write-Address (read (self) Address) 33333)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 3)
  (send (another-node-id ANY) 9 NORMAL Remote-Test (read (self) Address)
        33333 3 (node-id) (self) Barrier)
  (send (another-node-id ANY) 9 NORMAL Remote-Test (read (self) Address)
        33333 3 (node-id) (self) Barrier)
  (send (another-node-id ANY) 9 NORMAL Remote-Test (read (self) Address)
        33333 3 (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;; This test locally writes a location, then reads it remotely on
;; several nodes.  Then it writes it again and reads it back to make sure
;; the read chain was invalidated.
(define-handler Test-4 (Address) (Node-1 Node-2 Node-3 Barrier)
  (print-user "~&Test 4: read chain invalidation~&")
  (write (self) Node-1 (another-node-id FAR))
  (write (self) Node-2 (another-node-id FAR))
  (write (self) Node-3 (another-node-id FAR))
  (call 8 Write-Address (read (self) Address) 40404)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 3)
  (send (read (self) Node-1) 9 NORMAL Remote-Test (read (self) Address)
        40404 4 (node-id) (self) Barrier)
  (send (read (self) Node-2) 9 NORMAL Remote-Test (read (self) Address)
        40404 4 (node-id) (self) Barrier)
  (send (read (self) Node-3) 9 NORMAL Remote-Test (read (self) Address)
        40404 4 (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (call 8 Write-Address (read (self) Address) 44444)
```

```
  (adjust-count (self) Barrier 3)
  (send (read (self) Node-1) 9 NORMAL Remote-Test (read (self) Address)
        44444 4 (node-id) (self) Barrier)
  (send (read (self) Node-2) 9 NORMAL Remote-Test (read (self) Address)
        44444 4 (node-id) (self) Barrier)
  (send (read (self) Node-3) 9 NORMAL Remote-Test (read (self) Address)
        44444 4 (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;; This test locally writes three cache conflicting locations, then
;; tests there values.
(define-handler Test-5 (Base) (Address Barrier)
  (print-user "~&Test 5: multiple writes and reads on node~&")
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 3)
  (write (self) Address (read (self) Base))
  (send (node-id) 8 NORMAL Remote-Write (read (self) Address) 55551
        (node-id) (self) Barrier)
  (write (self) Address (plus (read (self) Address) Stride))
  (send (node-id) 8 NORMAL Remote-Write (read (self) Address) 55552
        (node-id) (self) Barrier)
  (write (self) Address (plus (read (self) Address) Stride))
  (send (node-id) 8 NORMAL Remote-Write (read (self) Address) 55553
        (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (write (self) Address (read (self) Base))
  (adjust-count (self) Barrier 3)
  (send (node-id) 9 NORMAL Remote-Test (read (self) Address)
        55551 5 (node-id) (self) Barrier)
  (write (self) Address (plus (read (self) Address) Stride))
  (send (node-id) 9 NORMAL Remote-Test (read (self) Address)
        55552 5 (node-id) (self) Barrier)
  (write (self) Address (plus (read (self) Address) Stride))
  (send (node-id) 9 NORMAL Remote-Test (read (self) Address)
        55553 5 (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;; This location remotely writes a block of memory locations starting at
;; Address, then remotely tests them.
(define-handler Test-6 (Address Size) (Limit Index Barrier)
  (print-user "~&Test 6: remote sequential block write and read~&")
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier (read (self) Size))
  (write (self) Limit (plus (read (self) Address) (read (self) Size)))
  (write (self) Index (read (self) Address))
  (print-user "~&writing ")
  Loop-6A
  (send (another-node-id ANY) 8 NORMAL Remote-Write (read (self) Index)
        (read (self) Index) (node-id) (self) Barrier)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Limit)) Loop-6A)
  (test-count (self) Barrier)
  (adjust-count (self) Barrier (read (self) Size))
  (write (self) Index (read (self) Address))
```

```
  (print-user "~&reading ")
  Loop-6B
  (send (another-node-id ANY) 9 NORMAL Remote-Test (read (self) Index)
        (read (self) Index) 6 (node-id) (self) Barrier)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Limit)) Loop-6B)
  (test-count (self) Barrier)
  (destroy-segment (self)))
;; This location writes a block of memory locations starting at Address,
;; then reads them back.
(define-handler Test-7 (Address Size) (Limit Index Value)
  (print-user "~&Test 7: sequential block write and read~&")
  (write (self) Limit (plus (read (self) Address) (read (self) Size)))
  (write (self) Index (read (self) Address))
  (print-user "~&writing ")
  Loop-7A
  (call 4 Print-Count (read (self) Index))
  (write (self) Value (times (read (self) Index) 10))
  (call 8 Write-Address (read (self) Index) (read (self) Value))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Limit)) Loop-7A)
  (write (self) Index (read (self) Address))
  (print-user "~&reading ")
  Loop-7B
  (call 4 Print-Count (read (self) Index))
  (write (self) Value (times (read (self) Index) 10))
  (call 7 Test-Location (read (self) Index) (read (self) Value) 7)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Limit)) Loop-7B)
  (destroy-segment (self)))
```

A.5 Set Synchronization

```
;;                Set  Synchronization
;;
;;        Scott Wills                              23 January 1990
;;
;;   This is a set synchronization mechanisms demonstration.
(in-package 'user)
(define-constant WORK-SIZE 50)
(define-constant END -1)
(define-constant FANOUT 5)
;; These constant define the offset of the Down-Barrier in a stem or
;; root.
(define-constant DOWN-BARRIER-OFFSET 4)
;; These constant define the offset of the Down-Segments segment in a
;; stem or root.
(define-constant DOWN-SEGMENTS-OFFSET 6)
```

```
;; This constant defines the offset of the Up-Sync for stems and
;; leaves.
(define-constant UP-SYNC-OFFSET 3)
;; This handler tests set synchronization by constructing a
;; synchronization test of a specified size, and performing a
;; specified number of iterations using it.
(define-handler Set-Sync-Test (Number Iterations)
                    (Index Down-Barrier Down-Node Down-Segments)
  (write (self) Down-Node (another-node-id NEAR))
  (send (read (self) Down-Node) 14 NORMAL Stem
        (read (self) Number) (node-id) (self) 0)
  (attribute (self) Down-Barrier B-SYNC)
  (adjust-count (self) Down-Barrier 1)
  (write (self) Down-Segments (create-read-write-segment 1))
  (test-count (self) Down-Barrier)  ;; wait for child to ack
  (write (self) Index 0)
  Main-Loop
  (send (read (self) Down-Node) 6 NORMAL Reply-Value (read (self) Index)
        (read (read (self) Down-Segments) 0) UP-SYNC-OFFSET)
  (adjust-count (self) Down-Barrier 1)
  (test-count (self) Down-Barrier)  ;; wait for child finish signal
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Iterations))
                   Main-Loop)
  (send (read (self) Down-Node) 6 NORMAL Reply-Value END
        (read (read (self) Down-Segments) 0) UP-SYNC-OFFSET)
  (destroy-segment (read (self) Down-Segments))
  (destroy-segment (self)))
;; This recursive handler creates the necessary tree structure and
;; leaf cells for the specified number of cells and fanout.  After
;; the necessary children are created, the handler waits for each to
;; return its segment name via the ack message.  When all children
;; have acked, the handler acks its parent and enters the main loop.
;; This loop consists of the following procedure:  First the stem
;; waits for the start signal from its parent.  Then it forwards the
;; start signal to each of its children.  Then it tests the signal to
;; see if it is the END signal (end of simulation).  If so, it
;; destroys itself.  Otherwise it waits for all children to send
;; finish signals.  Then it sends a finish signal to its parent and
;; returns to the beginning of the main loop.
(define-handler Stem (Number Up-Node Up-Segment Up-Index)
                     (Up-Sync Down-Barrier Down-Nodes Down-Segments
                     Index Count Temp)
;;(print-user "~&creating stem ~a parent ~a~&"
;;            (self) (read (self) Up-Segment))
  (write (self) Down-Nodes (create-read-write-segment FANOUT))
  (write (self) Index 0)
  Loop-A
  (write (self) Count (minus FANOUT (read (self) Index)))
  (write (self) Temp (ceiling (read (self) Number) (read (self) Count)))
  (branch-zero (compare (read (self) Temp) 1) Skip-A)
  (branch-not-minus (compare (read (self) Temp) FANOUT) Skip-A)
  (write (self) Temp FANOUT)
  Skip-A
```

```
  (write (read (self) Down-Nodes) (read (self) Index) (another-node-id NEAR))
  (branch-zero (compare (read (self) Temp) 1) Skip-B)
  (send (read (read (self) Down-Nodes) (read (self) Index)) 14 NORMAL Stem
        (read (self) Temp) (node-id) (self) (read (self) Index))
  (branch-zero 0 Skip-C)
  Skip-B
  (send (read (read (self) Down-Nodes) (read (self) Index)) 9 NORMAL Leaf
        (node-id) (self) (read (self) Index))
  Skip-C
  (write (self) Index (plus (read (self) Index) 1))
  (write (self) Number (minus (read (self) Number) (read (self) Temp)))
  (branch-not-zero (read (self) Number) Loop-A)
  (write (self) Count (read (self) Index))
  (attribute (self) Down-Barrier B-SYNC)
  (adjust-count (self) Down-Barrier (read (self) Count))
  (write (self) Down-Segments
         (create-read-write-segment (read (self) Count)))
  (test-count (self) Down-Barrier)  ;; wait for children segment acks
  (attribute (self) Up-Sync S-SYNC)
  (send (read (self) Up-Node) 6 NORMAL Ack
        (self) (read (self) Up-Segment) (read (self) Up-Index))
  Main-Loop
  (write (self) Index 0)
  (write (self) Temp (read (self) Up-Sync)) ;; wait for start signal
;;(print-user "~&~a starting~&" (self))
  Loop-B
  (send (read (read (self) Down-Nodes) (read (self) Index))
        6 NORMAL Reply-Value (read (self) Temp)
        (read (read (self) Down-Segments) (read (self) Index))
        UP-SYNC-OFFSET)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Count)) Loop-B)
  (branch-zero (compare (read (self) Temp) END) Simulation-End)
  (adjust-count (self) Down-Barrier (read (self) Count))
  (test-count (self) Down-Barrier)  ;; wait for children finish signals
;;(print-user "~&~a stopping~&" (self))
  (send (read (self) Up-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Up-Segment) DOWN-BARRIER-OFFSET)
  (branch-zero 0 Main-Loop)
  Simulation-End
;;(print-user "~&~a ending~&" (self))
  (destroy-segment (read (self) Down-Nodes))
  (destroy-segment (read (self) Down-Segments))
  (destroy-segment (self)))
;; This handler creates a leaf (work) node.  It first acks its
;; segment name to it parent.  Then it wait for a start signal.  If
;; the signal is an END signal, it destroys itself.  Otherwise it
;; performs one iteration of the work and sends a finish signal.
;; Then it awaits another start signal.
(define-handler Leaf (Up-Node Up-Segment Up-Index) (Up-Sync Index Temp)
;;(print-user "~&creating leaf ~a parent ~a~&" (self)
;;            (read (self) Up-Segment))
  (attribute (self) Up-Sync S-SYNC)
  (send (read (self) Up-Node) 6 NORMAL Ack
```

```
        (self) (read (self) Up-Segment) (read (self) Up-Index))
  (branch-zero 0 Wait-Point)
  Main-Loop
  (write (self) Index (minus (read (self) Index) 1))
  (branch-not-zero (read (self) Index) Main-Loop)
;;(print-user "~&~a working~&" (self))
  (send (read (self) Up-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Up-Segment) DOWN-BARRIER-OFFSET)
  Wait-Point
  (write (self) Index WORK-SIZE)
  (write (self) Temp (read (self) Up-Sync)) ;; wait for start signal
  (branch-not-zero (compare (read (self) Temp) END) Main-Loop)
;;(print-user "~&~a ending~&" (self))
  (destroy-segment (self)))
;; This handler acknowledges the creation of a child (leaf or stem)
;; and writes the segment at the appropriate place in the
;; Down-Segments segment.  It then reduces the Down-Barrier by one.
(define-handler Ack (Segment Up-Segment Up-Index) ()
  (write (read (read (self) Up-Segment) DOWN-SEGMENTS-OFFSET)
         (read (self) Up-Index) (read (self) Segment))
  (adjust-count (read (self) Up-Segment) DOWN-BARRIER-OFFSET -1)
  (destroy-segment (self)))
```

A.6 Object Name Translation

```
;;                      T r a n s l a t i o n
;;
;;         Scott Wills                              26 January 1990
;;
;;   This is an object name translation mechanism writen in Pi.
(in-package 'user)
;; This constant is the number of maps.
(define-constant NUMBER-OF-MAPS 10)
;; This constant defines the initial map size
(define-constant INITIAL-MAP-SIZE 25)
;; This constant is the number of agents.
(define-constant NUMBER-OF-AGENTS 25)
;; These constants defines offsets into the Agent segment.
(define-constant AGENT-MAP-NODES 0)
(define-constant AGENT-MAP-SEGMENTS 1)
(define-constant AGENT-NEXT-NODE 2)
(define-constant AGENT-NEXT-SEGMENT 3)
(define-constant AGENT-SIZE 4)
;; These constants defines offsets into the client segments.
(define-constant CLIENT-AGENT-NODE 0)
(define-constant CLIENT-AGENT-SEGMENT 1)
(define-constant CLIENT-CACHE 2)
(define-constant CLIENT-SIZE 3)
```

```
;; This constant defines the initial client translation cache size.
(define-constant INITIAL-CLIENT-CACHE-SIZE 10)
;; This handler initializes the translation system (except for the
;; clients which must be created separately where needed.  It creates
;; the maps and agents.  A prototype agent location is returned (node
;; and segment).
(define-handler Initialize-Translations (Reply-Node Reply-Segment
                                         Reply-Agent-Node
                                         Reply-Agent-Segment)
                                        (Agent-Node Agent-Segment)
  (write (self) Agent-Node (another-node-id FAR))
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value
        (read (self) Agent-Node) (read (self) Reply-Segment)
        (read (self) Reply-Agent-Node))
  (send (read (self) Agent-Node) 10 NORMAL Create-Agent 0 0 0
        (node-id) (self) Agent-Segment)
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value
        (read (self) Agent-Segment)
        (read (self) Reply-Segment) (read (self) Reply-Agent-Segment))
  (send (another-node-id FAR) 6 NORMAL Create-Map
        0 (read (self) Agent-Node) (read (self) Agent-Segment))
  (destroy-segment (self)))
;; This handler creates a new map.  The segment name is replied to the
;; head of the agent ring.  If the index is less than NUMBER-OF-MAPS,
;; an additional map is created.
(define-handler Create-Map (Index Reply-Node Reply-Segment) ()
  (send (read (self) Reply-Node) 8 NORMAL Map-Location-Update
        0 (node-id)
        (create-associative-segment INITIAL-MAP-SIZE UNBOUND-SAFE)
        (read (self) Index) (read (self) Reply-Segment))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-zero (compare (read (self) Index) NUMBER-OF-MAPS) Skip-A)
  (send (another-node-id FAR) 6 NORMAL Create-Map (read (self) Index)
        (read (self) Reply-Node) (read (self) Reply-Segment))
  Skip-A
  (destroy-segment (self)))
;; This handler creates the agent ring.  This procedure is a bit
;; complicated.  When the ring is create, it is created in the reverse
;; direction of normal message travel.  Each new agent receives the
;; node and segment of the agent in front of it.  It also forwards the
;; return field including the node and segment ID of the start of the
;; ring (to close the ring at the last agent).  An index of zero
;; indicates the starting agent.  He returns his segment to the root
;; and passes his node and segment ID as the return field (for the
;; ring closing).  For the starting agent case, the input parameters
;; are used abnormally.  Return-Node contains the root node number.
;; Return-Segment contains the root segment number.  Return-Offset
;; contains the segment offset of returned value.
(define-handler Create-Agent (Index Next-Node Next-Segment
                              Return-Node Return-Segment Return-Offset)
                             (Agent)
  (write (self) Agent (create-read-write-segment AGENT-SIZE))
  (write (read (self) Agent) AGENT-MAP-NODES
         (create-read-write-segment NUMBER-OF-MAPS))
```

```
  (write (read (self) Agent) AGENT-MAP-SEGMENTS
         (create-read-write-segment NUMBER-OF-MAPS))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) 1) Skip-A)
  (send (read (self) Return-Node) 6 NORMAL Reply-Value
        (read (self) Agent) (read (self) Return-Segment)
        (read (self) Return-Offset))
  (write (self) Return-Node (node-id))
  (write (self) Return-Segment (read (self) Agent))
  (branch-zero 0 Skip-B)
  Skip-A
  (write (read (self) Agent) AGENT-NEXT-NODE (read (self) Next-Node))
  (write (read (self) Agent) AGENT-NEXT-SEGMENT (read (self) Next-Segment))
  Skip-B
  (branch-not-zero (compare (read (self) Index) NUMBER-OF-AGENTS) Skip-C)
  (send (read (self) Return-Node) 6 NORMAL Reply-Value
        (node-id) (read (self) Return-Segment) AGENT-NEXT-NODE)
  (send (read (self) Return-Node) 6 NORMAL Reply-Value
        (read (self) Agent) (read (self) Return-Segment) AGENT-NEXT-SEGMENT)
  (branch-zero 0 Skip-D)
  Skip-C
  (send (another-node-id FAR) 10 NORMAL Create-Agent
        (read (self) Index) (node-id) (read (self) Agent)
        (read (self) Return-Node) (read (self) Return-Segment) 0)
  Skip-D
  (destroy-segment (self)))
;; This handler creates a new client.  The segment name is replied to the
;; root table.  It then requests the nearest agent, and initializes the
;; local translation cache.
(define-handler Create-Client (Agent-Node Agent-Segment
                               Reply-Node Reply-Segment Reply-Index) (Client)
  (write (self) Client (create-read-write-segment CLIENT-SIZE))
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value (read (self) Client)
        (read (self) Reply-Segment) (read (self) Reply-Index))
  (send (read (self) Agent-Node) 10 NORMAL Closest-Agent
        0 (read (self) Agent-Segment) (node-id) (read (self) Client) -1 0 0)
  (attribute (read (self) Client) CLIENT-AGENT-NODE D-SYNC)
  (attribute (read (self) Client) CLIENT-AGENT-SEGMENT D-SYNC)
  (write (read (self) Client) CLIENT-CACHE
         (create-associative-segment INITIAL-CLIENT-CACHE-SIZE BOUND-UNSAFE))
  (destroy-segment (self)))
;; This handler finds the closest agent to a client.  Each agent
;; compares its distance to the current distance.  If it is closer, it
;; updates the agent node and segment with it own value and sets the
;; new distance.  When the last agent makes the comparison, the
;; closest agent is returned to the client.
(define-handler Closest-Agent (Count This-Agent Client-Node Client-Segment
                               Distance Agent-Node Agent-Segment) ()
  (branch-zero (compare (read (self) Distance) -1) Skip-A)
  (branch-not-minus (compare (distance (read (self) Client-Node) (node-id))
                             (read (self) Distance)) Skip-B)
  Skip-A
  (write (self) Distance (distance (read (self) Client-Node) (node-id)))
  (write (self) Agent-Node (node-id))
```

```
  (write (self) Agent-Segment (read (self) This-Agent))
  Skip-B
  (write (self) Count (plus (read (self) Count) 1))
  (branch-zero (compare (read (self) Count) NUMBER-OF-AGENTS) Skip-C)
  (send (read (read (self) This-Agent) AGENT-NEXT-NODE)
        10 NORMAL Closest-Agent (read (self) Count)
        (read (read (self) This-Agent) AGENT-NEXT-SEGMENT)
        (read (self) Client-Node) (read (self) Client-Segment)
        (read (self) Distance) (read (self) Agent-Node)
        (read (self) Agent-Segment))
  (branch-zero 0 Skip-D)
  Skip-C
  (send (read (self) Client-Node) 6 NORMAL Reply-Value
        (read (self) Agent-Node) (read (self) Client-Segment)
        CLIENT-AGENT-NODE)
  (send (read (self) Client-Node) 6 NORMAL Reply-Value
        (read (self) Agent-Segment) (read (self) Client-Segment)
        CLIENT-AGENT-SEGMENT)
  Skip-D
  (destroy-segment (self)))
;; This handler updates a map location maintained by each agent.  The
;; map number is indicated by Map-Index.  If the count is not equal to
;; NUMBER-OF-AGENTS, the message is also forwarded to the next agent
;; (with the count incremented).
(define-handler Map-Location-Update (Count Map-Node Map-Segment Map-Index
                                     Agent-Segment) ()
  (write (read (read (self) Agent-Segment) AGENT-MAP-NODES)
         (read (self) Map-Index) (read (self) Map-Node))
  (write (read (read (self) Agent-Segment) AGENT-MAP-SEGMENTS)
         (read (self) Map-Index) (read (self) Map-Segment))
  (write (self) Count (plus (read (self) Count) 1))
  (branch-zero (compare (read (self) Count) NUMBER-OF-AGENTS) Skip-A)
  (send (read (read (self) Agent-Segment) AGENT-NEXT-NODE)
        8 NORMAL Map-Location-Update
        (read (self) Count) (read (self) Map-Node)
        (read (self) Map-Segment) (read (self) Map-Index)
        (read (read (self) Agent-Segment) AGENT-NEXT-SEGMENT))
  Skip-A
  (destroy-segment (self)))
;; This handler inserts a translation on the specified client.  It
;; must be invoked on the same node as the client.  The handler waits
;; for an acknowledgment before ending.  If this handler is called,
;; the insertion is synchronous.  If is sent (on the same node), the
;; insertion is asynchronous.
(define-handler Insert (Name Value Client-Segment) (Barrier)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 1)
  (send (read (read (self) Client-Segment) CLIENT-AGENT-NODE)
        10 NORMAL Agent-Insert
        (read (self) Name) (read (self) Value)
        (read (read (self) Client-Segment) CLIENT-AGENT-SEGMENT)
        (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (destroy-segment (self)))
```

```
;; This called handler matches a translation on the specified client.
;; It must be invoked on the same node as the client.  The handler
;; first checks for a match in its local agent cache.  Failing there,
;; it sends a match request to the agent.    The value is returned to
;; the caller.
(define-handler Match (Name Client-Segment) (Value)
  (write (self) Value (match (read (read (self) Client-Segment) CLIENT-CACHE)
                             (read (self) Name)))
  (branch-not-zero (compare (read (self) Value) UNBOUND) Skip-A)
  (attribute (self) Value D-SYNC)
  (send (read (read (self) Client-Segment) CLIENT-AGENT-NODE)
        9 NORMAL Agent-Match
        (read (self) Name)
        (read (read (self) Client-Segment) CLIENT-AGENT-SEGMENT)
        (node-id) (self) Value)
  (insert (read (read (self) Client-Segment) CLIENT-CACHE)
          (read (self) Name) (read (self) Value))
  Skip-A
  (return (read (self) Value))
  (destroy-segment (self)))
;; This called handler matches a translation when an earlier
;; translation produces an incorrect result.  It first invalidates the
;; translation in the client cache (if one exists).  Then it requests
;; the translation using the Match handler.
(define-handler Rematch (Name Client-Segment) ()
  (remove (read (read (self) Client-Segment) CLIENT-CACHE)
          (read (self) Name))
  (return (call 6 Match (read (self) Name) (read (self) Client-Segment)))
  (destroy-segment (self)))
;; This handler inserts a translation into the distributed translation
;; map.  An ack is returned.
(define-handler Agent-Insert (Name Value This-Agent
                              Ack-Node Ack-Segment Ack-Offset) (Index)
  (write (self) Index (mod (read (self) Name) NUMBER-OF-MAPS))
  (send (read (read (read (self) This-Agent) AGENT-MAP-NODES)
              (read (self) Index))
        9 NORMAL Insert-In-Map
        (read (read (read (self) This-Agent) AGENT-MAP-SEGMENTS)
              (read (self) Index))
        (read (self) Name) (read (self) Value) (read (self) Ack-Node)
        (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;; This handler matches a translation into the distributed translation
;; map.  The resulting value is returned.
(define-handler Agent-Match (Name This-Agent
                             Reply-Node Reply-Segment Reply-Offset) (Index)
  (write (self) Index (mod (read (self) Name) NUMBER-OF-MAPS))
  (send (read (read (read (self) This-Agent) AGENT-MAP-NODES)
              (read (self) Index))
        8 NORMAL Match-In-Map
        (read (read (read (self) This-Agent) AGENT-MAP-SEGMENTS)
              (read (self) Index))
        (read (self) Name) (read (self) Reply-Node)
        (read (self) Reply-Segment) (read (self) Reply-Offset))
```

```
  (destroy-segment (self)))
;; This handler inserts a translation to the specified map.  An
;; acknowledgment is returned.
(define-handler Insert-In-Map (This-Map Name Value
                               Ack-Node Ack-Segment Ack-Offset) ()
  (insert (read (self) This-Map) (read (self) Name) (read (self) Value))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;; This handler matches a translation in the specified map.  The value
;; is returned.
(define-handler Match-In-Map (This-Map Name
                              Reply-Node Reply-Segment Reply-Offset) ()
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value
        (match (read (self) This-Map) (read (self) Name))
        (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
```

A.7 Object Name Translation Tests

```
;;                Translation  Tests
;;
;;         Scott Wills                              1 February 1990
;;
;;   This file contains tests for the translation mechanism.
(in-package 'user)
;; This constant is the number of clients.
(define-constant NUMBER-OF-CLIENTS 25)
;; This handler begins by initializing the translation network.  It
;; then creates the clients.  It then executes several tests on shared
;; memory.  All tests requiring client locations must be executed on
;; the same node as this handler.
(define-handler Test () (Agent-Node Agent-Segment
                         Client-Nodes Client-Segments Index)
  (print-user "~&Translation Tests~&")
  (send (another-node-id ANY) 9 NORMAL Initialize-Translations
        (node-id) (self) Agent-Node Agent-Segment)
  (write (self) Client-Nodes
         (create-read-write-segment NUMBER-OF-CLIENTS))
  (write (self) Client-Segments
         (create-read-write-segment NUMBER-OF-CLIENTS))
  (write (self) Index 0)
  Loop-A
  (write (read (self) Client-Nodes) (read (self) Index)
         (another-node-id ANY))
  (send (read (read (self) Client-Nodes) (read (self) Index))
        9 NORMAL Create-Client
        (read (self) Agent-Node) (read (self) Agent-Segment)
```

```
        (node-id) (read (self) Client-Segments) (read (self) Index))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) NUMBER-OF-CLIENTS) Loop-A)
  (call 9 Test-1 100 (read (self) Client-Nodes)
        (read (self) Client-Segments))
  (call 9 Test-2 5 10 (read (self) Client-Nodes)
        (read (self) Client-Segments))
  (destroy-segment (self)))
;; This test inserts a specified number of translations, waits for all
;; to be acknowledged, then reads them all back and tests them.  It
;; then rereads all translations (on the same clients) to test the
;; client caches (check the number of inserts and matches in the
;; profile).
(define-handler Test-1 (Number Client-Nodes Client-Segments)
                       (Index Client Barrier)
  (print-user "~&Test 1: sequential insert and match and cache test~&")
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier (read (self) Number))
  (write (self) Index 0)
  Loop-A
  (write (self) Client (mod (read (self) Index) NUMBER-OF-CLIENTS))
  (send (read (read (self) Client-Nodes) (read (self) Client))
        9 NORMAL Client-Insert
        (read (self) Index) (times (read (self) Index) 10)
        (read (read (self) Client-Segments) (read (self) Client))
        (node-id) (self) Barrier)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Number)) Loop-A)
  (test-count (self) Barrier)
  (adjust-count (self) Barrier (read (self) Number))
  (write (self) Index 0)
  (send (node-id) 8 NORMAL Compare-All-Clients (read (self) Number)
        (read (self) Client-Nodes) (read (self) Client-Segments))
  (send (node-id) 8 NORMAL Compare-All-Clients (read (self) Number)
        (read (self) Client-Nodes) (read (self) Client-Segments))
  (destroy-segment (self)))
;; This tests creates a client cache inconsistency, then corrects it.
;; The following steps are performed:
;;
;; (1) the translation (1000, 12345) is inserted on Client-B
;; (2) the translation for 1000 is compared on Client-A (caching it)
;; (3) the translation (1000, 54321) is inserted on Client-B (changing it)
;; (4) the translation on Client-A is tested for the old value (12345)
;; (5) the translation on Client-A is retested for the new value (54321)
(define-handler Test-2 (Client-A Client-B Client-Nodes Client-Segments)
                       (Value Barrier)
  (print-user "~&Test 2: client cache consistency test~&")
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 1)
  (send (read (read (self) Client-Nodes) (read (self) Client-B))
        9 NORMAL Client-Insert 1000 12345
        (read (read (self) Client-Segments) (read (self) Client-B))
        (node-id) (self) Barrier)
  (test-count (self) Barrier)
```

```
  (send (read (read (self) Client-Nodes) (read (self) Client-A))
        9 NORMAL Client-Match 1000
        (read (read (self) Client-Segments) (read (self) Client-A))
        (node-id) (self) Value)
  (branch-zero (compare (read (self) Value) 12345) Skip-A)
  (print-user "~&ERROR: ~a translation value ~a does not match expected ~a~&"
              1000 (read (self) Value) 12345)
  Skip-A
  (attribute (self) Value D-SYNC)
  (adjust-count (self) Barrier 1)
  (send (read (read (self) Client-Nodes) (read (self) Client-B))
        9 NORMAL Client-Insert 1000 54321
        (read (read (self) Client-Segments) (read (self) Client-B))
        (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (send (read (read (self) Client-Nodes) (read (self) Client-A))
        9 NORMAL Client-Match 1000
        (read (read (self) Client-Segments) (read (self) Client-A))
        (node-id) (self) Value)
  (branch-zero (compare (read (self) Value) 12345) Skip-B)
  (print-user "~&ERROR: ~a translation value ~a does not match expected ~a~&"
              1000 (read (self) Value) 12345)
  Skip-B
  (attribute (self) Value D-SYNC)
  (send (read (read (self) Client-Nodes) (read (self) Client-A))
        9 NORMAL Client-Rematch 1000
        (read (read (self) Client-Segments) (read (self) Client-A))
        (node-id) (self) Value)
  (branch-zero (compare (read (self) Value) 54321) Skip-C)
  (print-user "~&ERROR: ~a translation value ~a does not match expected ~a~&"
              1000 (read (self) Value) 54321)
  Skip-C
  (destroy-segment (self)))
;; This handler tests a specified number of translations on the
;; clients.  This handler must be executed on the same node as the
;; client node and segment arrays.
(define-handler Compare-All-Clients (Number Client-Nodes Client-Segments)
                                    (Index Client)
  (write (self) Index 0)
  Loop-A
  (write (self) Client (mod (read (self) Index) NUMBER-OF-CLIENTS))
  (send (read (read (self) Client-Nodes) (read (self) Client))
        7 NORMAL Client-Compare
        (read (self) Index) (times (read (self) Index) 10)
        (read (read (self) Client-Segments) (read (self) Client)))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Number)) Loop-A)
  (destroy-segment (self)))
;; This handler inserts a translation on the specified client, and
;; waits for the insertion to complete.  This handler must be executed
;; on the same node as the client.  An acknowledgment is returned
;; when the insertion completes.
(define-handler Client-Insert (Name Value Client-Segment
                               Ack-Node Ack-Segment Ack-Offset) ()
```

```
  (call 7 Insert (read (self) Name) (read (self) Value)
        (read (self) Client-Segment))
  (send (read (self) Ack-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Ack-Segment) (read (self) Ack-Offset))
  (destroy-segment (self)))
;; This handler matches a translation.  It requests the match from the
;; client.  If the translated value does not match the specified
;; value, an error is printed.  This test must be run on the same node
;; as the client.
(define-handler Client-Compare (Name Compare-Value Client-Segment) (Value)
  (write (self) Value
         (call 6 Match (read (self) Name) (read (self) Client-Segment)))
  (branch-zero (compare (read (self) Value) (read (self) Compare-Value))
               Skip)
  (print-user "~&ERROR: ~a translation value ~a does not match expected ~a~&"
              (read (self) Name) (read (self) Value)
              (read (self) Compare-Value))
  Skip
  (destroy-segment (self)))
;; This handler matches a translation.  It returns the matched value
;; to the caller.  This handler must must be executed on the same node
;; as the client.
(define-handler Client-Match (Name Client-Segment
                              Reply-Node Reply-Segment Reply-Offset) (Value)
  (write (self) Value
         (call 6 Match (read (self) Name) (read (self) Client-Segment)))
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value (read (self) Value)
        (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
;; This handler is similar to Client-Match, except Rematch is invoked.
;; This handler must must be executed on the same node as the client.
(define-handler Client-Rematch (Name Client-Segment Reply-Node
                                Reply-Segment Reply-Offset) (Value)
  (write (self) Value
         (call 5 Rematch (read (self) Name) (read (self) Client-Segment)))
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value (read (self) Value)
        (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
```

A.8 Non-Resident Handlers

```
;;              N o n - R e s i d e n t   H a n d l e r
;;
;;         Scott Wills                                1 February 1990
;;
;;   This example demonstrates non-resident handler support in Pi.
(in-package 'user)
;; These constants define offsets into a handler segment.
```

```
(define-constant HANDLER-LENGTH 0)
(define-constant HANDLER-NAME 1)
;; This constant defines the number of nodes.
(define-constant NUMBER-OF-NODES 64)
;; This constant defines the reference and current handler map size.
;; The reference handler map can grow larger; the current handler map
;; size is fixed.
(define-constant HANDLER-MAP-SIZE 5)
;; This nodal maintains the node's client segment.
(define-nodal Node-Client)
;; This nodal maintains the node's reference non-resident handler map.
(define-nodal Reference-Handlers)
;; This nodal maintains the node's current non-resident handler map.
(define-nodal Current-Handlers)
;; This nodal maintains the set of outstanding handler requests
(define-nodal Requested-Handlers)
;; This is test handler for the non-resident handler mechanism.  It
;; starts by initializing the NR handler system.  Then it injects a
;; test handler into the distributed reference handler map and inserts
;; then into the handler name translation system.  Then it invokes it
;; on each node twice.
(define-handler Test () (Barrier Index)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier 1)
  (send (another-node-id ANY) 10 NORMAL Initialize-NR-Handlers
        (node-id) (self) Barrier)
  (test-count (self) Barrier)
  (adjust-count (self) Barrier 1)
  (send (another-node-id ANY) 12 NORMAL Add-Handler
        25 (node-id) (self) Barrier Test-NR-Handler)
  (test-count (self) Barrier)
  (write (self) Index 0)
  Loop-A
  (send (read (self) Index) 14 NORMAL Dispatch 25 HAVE A NICE DAY)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-minus (compare (read (self) Index) NUMBER-OF-NODES) Loop-A)
  (write (self) Index 0)
  Loop-B
  (send (read (self) Index) 14 NORMAL Dispatch 25 HAVE A NICE DAY)
  (write (self) Index (plus (read (self) Index) 1))
  (branch-minus (compare (read (self) Index) NUMBER-OF-NODES) Loop-B)
  (destroy-segment (self)))
;; This handler is the test case for a non-resident handler.  It
;; prints the contents of the active segment.
(define-handler Test-NR-Handler (:VARIABLE) (Index)
;;  (print-user "~&nr handler dispatched on node ~d: [ " (node-id))
;;  (write (self) Index 0)
;;  Loop
;;  (print-user "~a " (read (self) (read (self) Index)))
;;  (write (self) Index (plus (read (self) Index) 1))
;;  (branch-minus (compare (read (self) Index) (read (self) LENGTH)) Loop)
;;  (print-user "]~&")
  (write (self) Index (times (read (self) LENGTH) (read (self) LENGTH)))
  (destroy-segment (self)))
```

```
;; This this is the main handler for non-resident handler support.
;; When a non-resident handler is invoked, this handler is called as
;; the normal handler type.  It then uses the first argument as the
;; non-resident handler type.  It tests if the handler is present on
;; the node.  If not, it obtains it.  Then it invokes it with the
;; remaining arguments.  A special argument segment must be created.
;; Since the locals for non-resident handler are included in this
;; space, care must be taken to only copy parameters into it. Note:
;; since PiSim presently does not support direct handler segment
;; calls, the first argument of the non-resident handler is a symbol
;; of a normal handler.
(define-handler Dispatch (Name :VARIABLE) (Handler Message-Length Arguments
                                           Index-A Index-B)
  (write (self) Handler (match (read (nodals) Current-Handlers)
                               (read (self) Name)))
  (branch-not-zero (compare (read (self) Handler) UNBOUND) Handler-Known)
  (call 6 Lookup-Handler (read (self) Name))
  (write (self) Handler (match (read (nodals) Current-Handlers)
                               (read (self) Name)))
  Handler-Known
  (write (self) Message-Length (minus (read (self) LENGTH) 6))
  (write (self) Arguments
         (create-read-write-segment (minus (read (self) LENGTH) 10)))
  (write (self) Index-A 9)
  (write (self) Index-B 0)
  Loop
  (branch-not-zero (probe (self) (read (self) Index-A) READ) Skip)
  (write (read (self) Arguments) (read (self) Index-B)
         (read (self) (read (self) Index-A)))
  Skip
  (write (self) Index-A (plus (read (self) Index-A) 1))
  (write (self) Index-B (plus (read (self) Index-B) 1))
  (branch-minus (compare (read (self) Index-A) (read (self) LENGTH)) Loop)
  (call-segment (read (self) Message-Length) (read (read (self) Handler) 2)
                (read (self) Arguments))
;;  (call-segment (read (self) Message-Length) (read (self) Handler)
;;                (read (self) Arguments))
  (destroy-segment (read (self) Arguments))
  (destroy-segment (self)))
;; This handler lookups a non-resident handler.  It is called when a
;; handler is not present in the current handler map.  It first looks
;; to see if the handler has been requested by checking the requested
;; handler set.  If the handler has not been requested, it creates a
;; d-sync for future requests and inserts it into the requested
;; handler set.  It then looks up the reference handler location, and
;; requests the handler.  Then it waits on the d-sync it created.  If
;; the handler had already been requested, the task waits on the
;; d-sync in the requested handler set.  This d-sync is written by the
;; handler which replies the requested handler.
(define-handler Lookup-Handler (Name) (Handler Location)
  (write (self) Handler (match (read (nodals) Requested-Handlers)
                               (read (self) Name)))
  (branch-not-zero (compare (read (self) Handler) UNBOUND) Handler-Requested)
  (write (self) Handler (create-read-write-segment 1))
```

```
  (insert (read (nodals) Requested-Handlers)
          (read (self) Name) (read (self) Handler))
;;  (print-user "~&fetching handler ~a on node ~d~&"
;;              (read (self) Name) (node-id))
  (write (self) Location
         (call 6 Match (read (self) Name) (read (nodals) Node-Client)))
  (send (read (self) Location) 7 NORMAL Fetch-Handler
        (read (self) Name) (node-id))
  Handler-Requested
  (read (read (self) Handler) 0)
  (destroy-segment (self)))
;; This handler fetches a handler from a reference node.  That segment
;; is then sent to the caller.
(define-handler Fetch-Handler (Name Reply-Node) (Handler Size)
  (write (self) Handler (match (read (nodals) Reference-Handlers)
                               (read (self) Name)))
  (write (self) Size (plus (read (read (self) Handler) HANDLER-LENGTH) 6))
  (send-segment (read (self) Reply-Node) (read (self) Size) NORMAL
                Reply-Handler (read (self) Handler))
  (destroy-segment (self)))
;; This handler installs a handler into the current handler map on the
;; local node.  Handler requiring the new non-resident handler are
;; reactivated when this handler writes the entry in the requested
;; handler set.
(define-handler Reply-Handler (Handler-Length Handler-Name :VARIABLE)
                              (Handler Index-A Index-B)
  (write (self) Handler
         (create-read-write-segment (read (self) Handler-Length)))
  (write (self) Index-A Handler-Length)
  (write (self) Index-B 0)
  Loop
  (write (read (self) Handler) (read (self) Index-B)
         (read (self) (read (self) Index-A)))
  (write (self) Index-A (plus (read (self) Index-A) 1))
  (write (self) Index-B (plus (read (self) Index-B) 1))
  (branch-minus (compare (read (self) Index-A) (read (self) LENGTH)) Loop)
  (insert (read (nodals) Current-Handlers)
          (read (self) Handler-Name) (read (self) Handler))
  (write (self) Index-A (match (read (nodals) Requested-Handlers)
                               (read (self) Handler-Name)))
  (write (read (self) Index-A) 0 HANDLER-PRESENT)
  (remove (read (nodals) Requested-Handlers) (read (self) Handler-Name))
  (destroy-segment (self)))
;; This handler adds a segment into the reference handler map on this
;; node.  It also adds the handler name to the translation system via
;; the node client.  The length of the handler segment is computed
;; by subtracting the number of locals and other parameters (plus 3)
;; from the message length field.  Two is added to the handler segment
;; for the length and name slots.
(define-handler Add-Handler (Name Reply-Node Reply-Segment Reply-Offset
                             :VARIABLE) (Size Index-A Index-B  Handler)
  (call 7 Insert (read (self) Name) (node-id) (read (nodals) Node-Client))
  (write (self) Size (minus (read (self) LENGTH) 9))
  (write (self) Handler (create-read-write-segment (read (self) Size)))
```

```
  (write (read (self) Handler) HANDLER-LENGTH (read (self) Size))
  (write (read (self) Handler) HANDLER-NAME (read (self) Name))
  (write (self) Index-A :VARIABLE)
  (write (self) Index-B 2)
  Loop
  (write (read (self) Handler) (read (self) Index-B)
         (read (self) (read (self) Index-A)))
  (write (self) Index-A (plus (read (self) Index-A) 1))
  (write (self) Index-B (plus (read (self) Index-B) 1))
  (branch-plus (compare (read (self) Index-A) (read (self) LENGTH)) Loop)
  (insert (read (nodals) Reference-Handlers)
          (read (self) Name) (read (self) Handler))
  (send (read (self) Reply-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
;; This handler initializes the non-resident handler mechanism.  It
;; initializes the translation system and creates an client on every
;; node.  It also creates reference and current non-resident handler
;; maps on each node plus the requested handler set.  The reference
;; map maintains the sole reference copy of handlers stored on that
;; node.  The current handler map maintains handlers that have been
;; temporarily cached for use on that node.  The requested handler set
;; includes all handlers that have been requested on the node.
(define-handler Initialize-NR-Handlers (Reply-Node Reply-Segment
                                        Reply-Offset) (Agent-Node
                                        Agent-Segment Index Barrier)
  (send (another-node-id ANY) 9 NORMAL Initialize-Translations
        (node-id) (self) Agent-Node Agent-Segment)
  (attribute (self) Barrier B-SYNC)
  (adjust-count (self) Barrier NUMBER-OF-NODES)
  (write (self) Index NUMBER-OF-NODES)
  Loop
  (write (self) Index (minus (read (self) Index) 1))
  (send (read (self) Index) 9 NORMAL Create-Node-Maps
        (read (self) Agent-Node) (read (self) Agent-Segment)
        (node-id) (self) Barrier)
  (branch-not-zero (read (self) Index) Loop)
  (test-count (self) Barrier)
  (send (read (self) Reply-Node) 6 NORMAL Adjust-Barrier
        -1 (read (self) Reply-Segment) (read (self) Reply-Offset))
  (destroy-segment (self)))
;; This handler creates a translation client on this node and stored a
;; pointer to it in a nodal.  Then it creates the reference and
;; current handler maps and the requested handler set and stores them
;; in the corresponding nodals.
(define-handler Create-Node-Maps (Agent-Node Agent-Segment Reply-Node
                                  Reply-Segment Reply-Offset)
                                 (Client-Segment)
  (send (node-id) 9 NORMAL Create-Client
        (read (self) Agent-Node) (read (self) Agent-Segment)
        (node-id) (self) Client-Segment)
  (write (nodals) Node-Client (read (self) Client-Segment))
  (write (nodals) Reference-Handlers
         (create-associative-segment HANDLER-MAP-SIZE UNBOUND-SAFE))
```

```
(write (nodals) Current-Handlers
       (create-associative-segment HANDLER-MAP-SIZE UNBOUND-SAFE))
(write (nodals) Requested-Handlers
       (create-associative-segment HANDLER-MAP-SIZE UNBOUND-SAFE))
(send (read (self) Reply-Node) 6 NORMAL Adjust-Barrier
      -1 (read (self) Reply-Segment) (read (self) Reply-Offset))
(destroy-segment (self)))
```

A.9 N-Body Simulation

```
;;                              N - B o d y
;;
;;         Scott Wills                                  27 October 1989
;;
;;   This is an N-Body simulator writen in Pi.
(in-package 'user)
;; The mass of each body if a function of its ID:
;;
;;        ((body ID * Delta-Mass) + Base-Mass)
(define-constant Base-Mass 1000.0)
(define-constant Delta-Mass 100.0)
;; Bodies are randomly position within the box described by points
;; (0,0) and (Maximum-Position, Maximum-Position).
(define-constant Maximum-Position 1000.0)
;; These offset define the location of the slots in Body objects.
(define-constant I-Nodes-Offset 3)
(define-constant I-Segments-Offset 4)
(define-constant AX-Offset 9)
(define-constant AY-Offset 10)
(define-constant Barrier-Offset 11)
;; These offset define the location of the slots in Interaction objects.
(define-constant ID-A-Offset 16)
(define-constant X-A-Offset 3)
(define-constant Y-A-Offset 5)
(define-constant X-B-Offset 4)
(define-constant Y-B-Offset 6)
;; This is the gravitational constant.
(define-constant G 1.0)
;; This handler creates the necessary bodies and links for the N body
;; simulation.  First the bodies are created (with the nodes and
;; segments being stored in arraies).  Then each interaction between
;; the bodies is generated.  The simulation is self-starting.
(define-handler Initialize-System (Number-Bodies Last-Tick) (Body-Nodes
                                   Body-Segments N-1 Index-A Index-B Node)
  (branch-not-minus (minus (read (self) Number-Bodies) 2) Skip)
  (write (self) Number-Bodies 2)
  Skip
;;  (print-user "~&(Data-Size ~d ~d)~&" (read (self) Number-Bodies)
```

```
;;              (read (self) Last-Tick))
  (write (self) Body-Nodes
         (create-read-write-segment (read (self) Number-Bodies)))
  (write (self) Body-Segments
         (create-read-write-segment (read (self) Number-Bodies)))
  (write (self) N-1 (minus (read (self) Number-Bodies) 1))
  (write (self) Index-A 0)
  Loop-A
  (write (self) Node (another-node-id ANY))
  (write (read (self) Body-Nodes) (read (self) Index-A) (read (self) Node))
  (send (read (self) Node) 19 NORMAL Start-Body (read (self) Index-A)
        (read (self) N-1) (read (self) Last-Tick)
        (node-id) (read (self) Body-Segments))
  (write (self) Index-A (plus (read (self) Index-A) 1))
  (branch-not-zero (compare (read (self) Number-Bodies)
                            (read (self) Index-A)) Loop-A)
  (write (self) Index-A 0)
  (write (self) Index-B 1)
  Loop-B
  (send (another-node-id ANY) 23 NORMAL Start-Interaction
        (read (self) Index-A) (read (self) Index-B)
        (read (read (self) Body-Nodes) (read (self) Index-A))
        (read (read (self) Body-Nodes) (read (self) Index-B))
        (read (read (self) Body-Segments) (read (self) Index-A))
        (read (read (self) Body-Segments) (read (self) Index-B))
        (read (self) Last-Tick))
  (write (self) Index-B (plus (read (self) Index-B) 1))
  (branch-not-zero (compare (read (self) Number-Bodies)
                            (read (self) Index-B)) Loop-B)
  (write (self) Index-A (plus (read (self) Index-A) 1))
  (write (self) Index-B (plus (read (self) Index-A) 1))
  (branch-not-zero (compare (read (self) N-1) (read (self) Index-A)) Loop-B)
  (destroy-segment (read (self) Body-Nodes))
  (destroy-segment (read (self) Body-Segments))
  (destroy-segment (self)))
;; This handler initializes and starts a body.  The bodies storage
;; is created when this handler is invoked, so all the parameters are
;; initialized automatically.  The interaction node and segment arrays
;; are created first. After that, the body segment pointer can be
;; replied to the system initialization program. The X and Y position is
;; randomly selected (within specified limits).  The body velocity and
;; acceleration are initially zero.  The tick count is set to zero.  At
;; this point, the handler enters the update/adjust loop.  First it
;; sends it location to all interactions, then it wait for the
;; acceleration updates to be received (via a B-SYNC).  Then it adjusts
;; its velocity and position, increments the tick, and repeats (unless
;; the last tick has been reached).
(define-handler Start-Body (ID Count Last-Tick Reply-Node Reply-Segment)
                           (I-Nodes I-Segments X Y VX VY AX AY
                            Barrier Tick Index)
  (write (self) I-Nodes (create-read-write-segment (read (self) Count)))
  (write (self) I-Segments (create-read-write-segment (read (self) Count)))
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value (self)
        (read (self) Reply-Segment) (read (self) ID))
```

```
  (write (self) X (random Maximum-Position))
  (write (self) Y (random Maximum-Position))
  (write (self) VX 0)
  (write (self) VY 0)
  (attribute (self) Barrier B-SYNC)
  (write (self) Tick 0)
;;  (print-user "~&(Body-Position ~d ~d ~5,1f ~5,1f)~&" (read (self) Tick)
;;              (read (self) ID) (read (self) X) (read (self) Y))
  Loop-A
  (write (self) AX 0)
  (write (self) AY 0)
  (adjust-count (self) Barrier (read (self) Count))
  (write (self) Index 0)
  Loop-B
  (send (read (read (self) I-Nodes) (read (self) Index))
        7 NORMAL Update-Position
        (read (self) X) (read (self) Y) (read (self) ID)
        (read (read (self) I-Segments) (read (self) Index)))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (compare (read (self) Index) (read (self) Count)) Loop-B)
  (test-count (self) Barrier)
  (write (self) X (plus (read (self) X) (read (self) VX)))
  (write (self) Y (plus (read (self) Y) (read (self) VY)))
  (write (self) VX (plus (read (self) VX) (read (self) AX)))
  (write (self) VY (plus (read (self) VY) (read (self) AY)))
  (write (self) Tick (plus (read (self) Tick) 1))
;;  (print-user "~&(Body-Position ~d ~d ~5,1f ~5,1f)~&" (read (self) Tick)
;;              (read (self) ID) (read (self) X) (read (self) Y))
  (branch-not-zero (read (self) ID) Skip)
  (branch-not-zero (mod (read (self) Tick) 5) Skip)
  (print-user "~4d " (read (self) Tick))
  (branch-not-zero (mod (read (self) Tick) 50) Skip)
  (print-user "~&")
  Skip
  (branch-not-zero (compare (read (self) Tick) (read (self) Last-Tick))
                   Loop-A)
  (destroy-segment (read (self) I-Nodes))
  (destroy-segment (read (self) I-Segments))
  (destroy-segment (self)))
;; This handler creates and starts an interaction. It begins by
;; attributing the position slots of A and B bodies.  It then calculates
;; the mass of the bodies.  The handler then sends a message to the two
;; bodies setting this interaction's location.  For the B body, the A
;; body's ID is used as the interaction identification.  For the A body,
;; one minus B's ID is used (since no body has an interaction with
;; itself).  This also starts the simulation on the body objects.  and
;; initializes the tick.  At this point, the interaction begins the
;; simulation loop by calculating the delta distances.  This will lock
;; on the S-SYNCs until the values are updated by bodies.  The force
;; between the two bodies is then calculated, and the acceleration
;; component on each body is determined.  These components are sent to
;; the bodies and the process begins again (until the last tick is
;; reached).  The following formulas is used.
;;
```

```
;; AX-A = K * M-B * (X-A - X-B) / ((X-B - X-A)^2 + (Y-B - Y-A)^2)^1.5
;; AY-A = K * M-B * (Y-A - Y-B) / ((X-B - X-A)^2 + (Y-B - Y-A)^2)^1.5
;; AX-B = K * M-A * (X-B - X-A) / ((X-B - X-A)^2 + (Y-B - Y-A)^2)^1.5
;; AY-B = K * M-A * (Y-B - Y-A) / ((X-B - X-A)^2 + (Y-B - Y-A)^2)^1.5
(define-handler Start-Interaction (ID-A ID-B Node-A Node-B Segment-A
                                   Segment-B Last-Tick) (X-A X-B Y-A Y-B
                                   Mass-A Mass-B Tick DX DY FX FY AX AY)
  (attribute (self) X-A S-SYNC)
  (attribute (self) Y-A S-SYNC)
  (attribute (self) X-B S-SYNC)
  (attribute (self) Y-B S-SYNC)
  (write (self) Mass-A (times (read (self) ID-A) Delta-Mass))
  (write (self) Mass-A (plus (read (self) Mass-A) Base-Mass))
  (write (self) Mass-B (times (read (self) ID-B) Delta-Mass))
  (write (self) Mass-B (plus (read (self) Mass-B) Base-Mass))
  (write (self) ID-B (minus (read (self) ID-B) 1))
  (send (read (self) Node-A) 9 NORMAL Write-Interaction-Location
        (read (self) ID-B) (node-id) (self) (read (self) Segment-A))
  (send (read (self) Node-B) 9 NORMAL Write-Interaction-Location
        (read (self) ID-A) (node-id) (self) (read (self) Segment-B))
  (write (self) Tick 0)
  Loop-A
  (write (self) DX (minus (read (self) X-B) (read (self) X-A)))
  (write (self) DY (minus (read (self) Y-B) (read (self) Y-A)))
  (write (self) FX (times (read (self) DX) (read (self) DX)))
  (write (self) FY (times (read (self) DY) (read (self) DY)))
  (write (self) FX (plus (read (self) FX) (read (self) FY)))
  (write (self) FX (exponent (read (self) FX) -1.5))
  (write (self) FX (times (read (self) FX) G))
  (write (self) FY (times (read (self) FX) (read (self) DY)))
  (write (self) FX (times (read (self) FX) (read (self) DX)))
  (write (self) AX (times (read (self) FX) (read (self) Mass-B)))
  (write (self) AY (times (read (self) FY) (read (self) Mass-B)))
  (send (read (self) Node-A) 6 NORMAL Add-Acceleration
        (read (self) AX) (read (self) AY) (read (self) Segment-A))
  (write (self) AX (times (read (self) FX) (read (self) Mass-A)))
  (write (self) AY (times (read (self) FY) (read (self) Mass-A)))
  (write (self) AX (times (read (self) AX) -1))
  (write (self) AY (times (read (self) AY) -1))
  (send (read (self) Node-B) 6 NORMAL Add-Acceleration
        (read (self) AX) (read (self) AY) (read (self) Segment-B))
  (write (self) Tick (plus (read (self) Tick) 1))
  (branch-not-zero (compare (read (self) Tick) (read (self) Last-Tick))
                   Loop-A)
  (destroy-segment (self)))
;; This handler writes the interaction location in the correct position
;; in the body's I-Node and I-Segment arrays.
(define-handler Write-Interaction-Location (I I-Node I-Segment Body-Segment)
                                           (I-Nodes I-Segments)
  (write (self) I-Nodes (read (read (self) Body-Segment) I-Nodes-Offset))
  (write (self) I-Segments
         (read (read (self) Body-Segment) I-Segments-Offset))
  (write (read (self) I-Nodes) (read (self) I) (read (self) I-Node))
  (write (read (self) I-Segments) (read (self) I) (read (self) I-Segment))
```

```
  (destroy-segment (self)))
;; This handler updates the bodies position on an interaction.  The body
;; ID is used to determine whether the update is body A or B.
(define-handler Update-Position (X Y ID I-Segment) ()
  (branch-not-zero (compare (read (read (self) I-Segment) ID-A-Offset)
                            (read (self) ID)) Update-Body-B)
  (write (read (self) I-Segment) X-A-Offset (read (self) X))
  (write (read (self) I-Segment) Y-A-Offset (read (self) Y))
  (destroy-segment (self))
  Update-Body-B
  (write (read (self) I-Segment) X-B-Offset (read (self) X))
  (write (read (self) I-Segment) Y-B-Offset (read (self) Y))
  (destroy-segment (self)))
;; This handler adds the acceleration component of one interaction to
;; the bodies composite acceleration values.  The bodies barrier is
;; also decremented.
(define-handler Add-Acceleration (AX AY Body-Segment) ()
  (write (read (self) Body-Segment) AX-Offset
         (plus (read (read (self) Body-Segment) AX-Offset) (read (self) AX)))
  (write (read (self) Body-Segment) AY-Offset
         (plus (read (read (self) Body-Segment) AY-Offset) (read (self) AY)))
  (adjust-count (read (self) Body-Segment) Barrier-Offset -1)
  (destroy-segment (self)))
```

A.10 Relaxation

```
;;                      R e l a x a t i o n
;;
;;         Scott Wills                                23 February 1990
;;
;;   This is a 2d relaxation simulator writen in Pi.
(in-package 'user)
;; This constant defines the array dimensions
(define-constant X-Size 10)
(define-constant Y-Size 10)
;; This constant defines the number of nodes in the system.
(define-constant Number-Of-Nodes 64)
;; This constant defines the initial size of the element map
(define-constant ELEMENT-MAP-SIZE 2)
;; These constants define neighbor temperature offsets in an element.
(define-constant NORTH-OFFSET 7)
(define-constant EAST-OFFSET 8)
(define-constant WEST-OFFSET 9)
(define-constant SOUTH-OFFSET 10)
;; These constants define the boundary temperatures.
(define-constant NORTH-BORDER-TEMP 100.0)
(define-constant EAST-BORDER-TEMP 1000.0)
(define-constant WEST-BORDER-TEMP 1000.0)
```

```
(define-constant SOUTH-BORDER-TEMP 100.0)
;; This nodal contains the element map; an associative set of elements
;; accessed by their index.
(define-nodal Element-Map)
;; This handler creates the relaxation elements in the system.  Each
;; element is assigned a index equal its position in the array.  A
;; elements index = (Y*Y-Size + X-Size).  A element is created on node
;; (mod index Number-Of-Nodes).  Each element performs a fixed number of
;; iterations (defined by Last-Tick), then returns its final temperature to
;; an array created by this task.  This task then prints out the final
;; results.
(define-handler Relax (Last-Tick) (Number-Elements Index Node Results)
  (write (self) Number-Elements (times X-Size Y-Size))
  (write (self) Results
         (create-read-write-segment (read (self) Number-Elements)))
  (write (self) Index 0)
  Start-Loop
  (write (self) Node (mod (read (self) Index) Number-Of-Nodes))
  (send (read (self) Node) 21 NORMAL Start-Element (read (self) Index)
        (read (self) Last-Tick) (node-id) (read (self) Results))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-minus (compare (read (self) Index) (read (self) Number-Elements))
                Start-Loop)
  (write (self) Index (times X-Size Y-Size))
  (write (self) Index (minus (read (self) Index) X-Size))
  (print-user "~&")
  Print-Loop
  (print-user "~5,1f " (read (read (self) Results) (read (self) Index)))
  (write (self) Index (plus (read (self) Index) 1))
  (branch-not-zero (mod (read (self) Index) X-Size) Print-Loop)
  (print-user "~&")
  (write (self) Index (minus (read (self) Index) (times 2 X-Size)))
  (branch-not-minus (read (self) Index) Print-Loop)
  (destroy-segment (read (self) Results))
  (destroy-segment (self)))
;; This handler starts by initializing the element object and
;; installing it in the element map.  It also determines whether it is
;; a boundary element and makes the appropriate adjustment.  S-Syncs
;; are the basis of this example.  Four s-syncs are created for the
;; north, south, east, and west neighbors.  The element task then
;; enters a loop of sending its value to its neighbors, then averaging
;; the its neighbor's values to compute it's new value.
(define-handler Start-Element (Index Last-Tick Reply-Node Reply-Segment)
                              (N-Index E-Index W-Index S-Index
                               N-Temp E-Temp W-Temp S-Temp
                               N-Node E-Node W-Node S-Node
                               Temp Tick)
  (write (self) N-Index (plus (read (self) Index) X-Size))
  (write (self) E-Index (plus (read (self) Index) 1))
  (write (self) W-Index (minus (read (self) Index) 1))
  (write (self) S-Index (minus (read (self) Index) X-Size))
  (attribute (self) N-Temp S-SYNC)
  (attribute (self) E-Temp S-SYNC)
  (attribute (self) W-Temp S-SYNC)
```

```
(attribute (self) S-Temp S-SYNC)
(write (self) N-Node (mod (read (self) N-Index) NUMBER-OF-NODES))
(write (self) E-Node (mod (read (self) E-Index) NUMBER-OF-NODES))
(write (self) W-Node (mod (read (self) W-Index) NUMBER-OF-NODES))
(write (self) S-Node (mod (read (self) S-Index) NUMBER-OF-NODES))
(write (self) Temp 0.0)
(write (self) Tick 0)
(branch-minus (compare (read (self) Index)
                       (times X-Size (minus Y-Size 1))) Skip-A)
(write (self) N-Index BORDER)
(attribute (self) N-Temp D-SYNC)
(write (self) N-Temp NORTH-BORDER-TEMP)
Skip-A
(branch-not-zero (compare (mod (read (self) Index) X-Size)
                          (minus X-Size 1)) Skip-B)
(write (self) E-Index BORDER)
(attribute (self) E-Temp D-SYNC)
(write (self) E-Temp EAST-BORDER-TEMP)
Skip-B
(branch-not-minus (compare (read (self) Index) X-Size) Skip-C)
(write (self) S-Index BORDER)
(attribute (self) S-Temp D-SYNC)
(write (self) S-Temp SOUTH-BORDER-TEMP)
Skip-C
(branch-not-zero (mod (read (self) Index) X-Size) Skip-D)
(write (self) W-Index BORDER)
(attribute (self) W-Temp D-SYNC)
(write (self) W-Temp WEST-BORDER-TEMP)
Skip-D
(branch-zero (probe (nodals) Element-Map READ) Map-Exists)
(write (nodals) Element-Map
       (create-associative-segment ELEMENT-MAP-SIZE UNBOUND-SAFE))
Map-Exists
(insert (read (nodals) Element-Map) (read (self) Index) (self))
Relax-Loop
(branch-zero (compare (read (self) N-Index) BORDER) Skip-E)
(send (read (self) N-Node) 7 NORMAL Update-Temp
      (read (self) N-Index) SOUTH-OFFSET (read (self) Temp))
Skip-E
(branch-zero (compare (read (self) E-Index) BORDER) Skip-F)
(send (read (self) E-Node) 7 NORMAL Update-Temp
      (read (self) E-Index) WEST-OFFSET (read (self) Temp))
Skip-F
(branch-zero (compare (read (self) W-Index) BORDER) Skip-G)
(send (read (self) W-Node) 7 NORMAL Update-Temp
      (read (self) W-Index) EAST-OFFSET (read (self) Temp))
Skip-G
(branch-zero (compare (read (self) S-Index) BORDER) Skip-H)
(send (read (self) S-Node) 7 NORMAL Update-Temp
      (read (self) S-Index) NORTH-OFFSET (read (self) Temp))
Skip-H
(write (self) Temp (plus (read (self) N-Temp) (read (self) E-Temp)))
(write (self) Temp (plus (read (self) Temp) (read (self) W-Temp)))
(write (self) Temp (plus (read (self) Temp) (read (self) S-Temp)))
```

```
  (write (self) Temp (divide (read (self) Temp) 4))
  (write (self) Tick (plus (read (self) Tick) 1))
  (branch-minus (compare (read (self) Tick) (read (self) Last-Tick))
                Relax-Loop)
  (send (read (self) Reply-Node) 6 NORMAL Reply-Value
        (read (self) Temp) (read (self) Reply-Segment) (read (self) Index))
  (destroy-segment (self)))
;; This handler deposites a temperature in the appropriate slot of an
;; element.
(define-handler Update-Temp (Index Offset Temp) (Element)
  Wait-Loop
  (write (self) Element (match (read (nodals) Element-Map)
                               (read (self) Index)))
  (branch-not-zero (compare (read (self) Element) UNBOUND) Write-Temp)
  (suspend)
  (branch-zero 0 Wait-Loop)
  Write-Temp
  (write (read (self) Element) (read (self) Offset) (read (self) Temp))
  (destroy-segment (self)))
```

Appendix B

Example Execution Logs

This appendix contains the execution logs for the examples described in chapter 3.

B.1 Shared Memory With Caches

```
------------------------------------------------------------
12 June 1990 0:26:30pm sm-test PiSim version 0.1
------------------------------------------------------------
task type profile
ACK.....................................2,181 (  6.6%)
ACK-FLUSH...............................778 (  2.4%)
ADJUST-BARRIER..........................593 (  1.8%)
FLUSH...................................778 (  2.4%)
GET-CACHE-LINE.....................1,560 (  4.7%)
INVALIDATE..............................780 (  2.4%)
NODE-SETUP..............................64 (  0.2%)
PRINT-COUNT.....................8,192 ( 24.8%)
READ....................................782 (  2.4%)
READ-ADDRESS.....................4,368 ( 13.2%)
REMOTE-TEST............................268 (  0.8%)
REMOTE-WRITE...........................261 (  0.8%)
REMOVE.............................1,323 (  4.0%)
REPLY..............................1,560 (  4.7%)
SHARED-MEMORY-SETUP.....................1 (  0.0%)
TEST....................................1 (  0.0%)
TEST-1..................................1 (  0.0%)
TEST-2..................................1 (  0.0%)
TEST-3..................................1 (  0.0%)
TEST-4..................................1 (  0.0%)
TEST-5..................................1 (  0.0%)
TEST-6..................................1 (  0.0%)
TEST-7..................................1 (  0.0%)
TEST-LOCATION.....................4,367 ( 13.2%)
WRITE...................................778 (  2.4%)
```

```
WRITE-ADDRESS.....................4,362 ( 13.2%)
total.............................33,004 (100.0%)
-------------------------------------------------------
task status profile
CALL..............................22,851 ( 35.9%)
NEW...............................10,153 ( 16.0%)
WAITING...........................30,604 ( 48.1%)
total.............................63,608 (100.0%)
-------------------------------------------------------
task run time profile
9-16..............................8,785 ( 26.6%)
17-32.............................7,237 ( 21.9%)
33-64............................11,391 ( 34.5%)
65-128............................5,505 ( 16.7%)
129-256.....................19 (  0.1%)
513-1,024.....................1 (  0.0%)
1,025-2,048.....................64 (  0.2%)
8,193-16,384.....................1 (  0.0%)
262,145-524,288.....................1 (  0.0%)
total.............................33,004 (100.0%)
-------------------------------------------------------
task wait time profile
0................................20,826 ( 63.1%)
1.....................................9 (  0.0%)
2.....................................3 (  0.0%)
3-4...................................9 (  0.0%)
5-8.....................................34 (  0.1%)
9-16....................................39 (  0.1%)
17-32...................................40 (  0.1%)
33-64.............................4,502 ( 13.6%)
65-128.....................917 (  2.8%)
129-256.....................1,305 (  4.0%)
257-512.....................2,762 (  8.4%)
513-1,024.....................403 (  1.2%)
1,025-2,048.....................336 (  1.0%)
2,049-4,096.....................568 (  1.7%)
4,097-8,192.....................726 (  2.2%)
8,193-16,384.....................521 (  1.6%)
16,385-32,768.....................2 (  0.0%)
32,769-65,536.....................1 (  0.0%)
524,289-1,048,576.....................1 (  0.0%)
total.............................33,004 (100.0%)
-------------------------------------------------------
instruction type profile
A-SHIFT...........................11,684 (  3.5%)
ADJUST-COUNT......................12,522 (  3.8%)
AND...............................29,144 (  8.8%)
ANOTHER-NODE-ID.....................3 (  0.0%)
ATTRIBUTE.....................9,405 (  2.9%)
BRANCH-NOT-ZERO..................41,944 ( 12.7%)
BRANCH-ZERO.....................19,036 (  5.8%)
CALL..............................45,702 ( 13.9%)
CREATE-ASSOCIATIVE-SEGMENT.....64 (  0.0%)
CREATE-READ-WRITE-SEGMENT.....702 (  0.2%)
```

```
DESTROY-SEGMENT.....33,004 ( 10.0%)
INITIALIZE.....13,901 (  4.2%)
INSERT.....4,806 (  1.5%)
MATCH.....13,408 (  4.1%)
MINUS.....6,721 (  2.0%)
MOVE.....40,450 ( 12.3%)
PLUS.....9,223 (  2.8%)
PRINT-USER.....104 (  0.0%)
REMOVE.....1,560 (  0.5%)
RETURN.....5,928 (  1.8%)
SEND.....10,152 (  3.1%)
TEST-COUNT.....11,704 (  3.6%)
TIMES.....8,192 (  2.5%)
total.....329,359 (100.0%)
-----------------------------------------------------
instructions per messages              32.4
-----------------------------------------------------
instructions per task                  10.0
-----------------------------------------------------
instructions run length                 8.1
-----------------------------------------------------
operation type profile
A-SHIFT.....11,684 (  0.8%)
ADJUST-COUNT.....12,522 (  0.8%)
AND.....29,144 (  1.9%)
ANOTHER-NODE-ID.....526 (  0.0%)
ATTRIBUTE.....9,405 (  0.6%)
BRANCH-NOT-ZERO.....41,944 (  2.7%)
BRANCH-ZERO.....19,036 (  1.2%)
CALL.....45,702 (  3.0%)
COMPARE.....44,020 (  2.9%)
CREATE-ASSOCIATIVE-SEGMENT.....64 (  0.0%)
CREATE-READ-WRITE-SEGMENT.....702 (  0.0%)
DESTROY-SEGMENT.....33,004 (  2.2%)
INSERT.....4,806 (  0.3%)
MATCH.....13,408 (  0.9%)
MINUS.....6,721 (  0.4%)
MOD.....16,384 (  1.1%)
NODALS.....26,256 (  1.7%)
NODE-ID.....2,159 (  0.1%)
PLUS.....9,223 (  0.6%)
PRINT-USER.....104 (  0.0%)
READ.....476,974 ( 31.3%)
REMOVE.....1,560 (  0.1%)
RETURN.....5,928 (  0.4%)
SELF.....543,595 ( 35.6%)
SEND.....10,152 (  0.7%)
TEST-COUNT.....11,704 (  0.8%)
TIMES.....8,192 (  0.5%)
WRITE.....141,098 (  9.2%)
total.....1,526,017 (100.0%)
-----------------------------------------------------
operations per instruction              4.6
-----------------------------------------------------
```

```
segment size profile
3.....................................1 (  0.0%)
4.....................................8,192 ( 24.3%)
5.....................................2,183 (  6.5%)
6.....................................3,477 ( 10.3%)
7.....................................10,295 ( 30.5%)
8.....................................6,250 ( 18.5%)
9.....................................332 (  1.0%)
12....................................638 (  1.9%)
13....................................778 (  2.3%)
16....................................1,624 (  4.8%)
total.................................33,770 (100.0%)
------------------------------------------------------------
average segment size                        6.9
------------------------------------------------------------
segment age profile
9-16..................................8,509 ( 25.8%)
17-32.................................1,747 (  5.3%)
33-64.................................12,192 ( 36.9%)
65-128................................2,742 (  8.3%)
129-256...............................1,524 (  4.6%)
257-512...............................3,604 ( 10.9%)
513-1,024.............................444 (  1.3%)
1,025-2,048...........................398 (  1.2%)
2,049-4,096...........................576 (  1.7%)
4,097-8,192...........................732 (  2.2%)
8,193-16,384..........................532 (  1.6%)
16,385-32,768.........................2 (  0.0%)
32,769-65,536.........................1 (  0.0%)
1,048,577-2,097,152...................1 (  0.0%)
total.................................33,004 (100.0%)
------------------------------------------------------------
last tick at 1,132,375
------------------------------------------------------------
simulation time: 19 minutes 5 seconds
------------------------------------------------------------
```

B.2 Shared Memory With Address Braiding

```
------------------------------------------------------------
12 June 1990 11:54:21am sm-nc-test PiSim version 0.1
------------------------------------------------------------
task type profile
ADJUST-BARRIER........................4,955 ( 12.4%)
NODE-SETUP............................64 (  0.2%)
PRINT-COUNT...........................8,192 ( 20.5%)
READ..................................4,368 ( 10.9%)
READ-ADDRESS..........................4,368 ( 10.9%)
```

```
REMOTE-TEST.....................268 (  0.7%)
REMOTE-WRITE....................261 (  0.7%)
REPLY-VALUE....................4,368 ( 10.9%)
SHARED-MEMORY-SETUP.............1 (  0.0%)
TEST............................1 (  0.0%)
TEST-1..........................1 (  0.0%)
TEST-2..........................1 (  0.0%)
TEST-3..........................1 (  0.0%)
TEST-4..........................1 (  0.0%)
TEST-5..........................1 (  0.0%)
TEST-6..........................1 (  0.0%)
TEST-7..........................1 (  0.0%)
TEST-LOCATION...................4,367 ( 10.9%)
WRITE..........................4,362 ( 10.9%)
WRITE-ADDRESS..................4,362 ( 10.9%)
total.........................39,944 (100.0%)
-------------------------------------------------------
task status profile
CALL...........................21,291 ( 30.4%)
NEW............................18,653 ( 26.7%)
WAITING........................30,031 ( 42.9%)
total.........................69,975 (100.0%)
-------------------------------------------------------
task run time profile
9-16...........................26,309 ( 65.9%)
17-32..........................4,897 ( 12.3%)
33-64..........................8,733 ( 21.9%)
65-128.........................2 (  0.0%)
513-1,024......................1 (  0.0%)
8,193-16,384...................1 (  0.0%)
262,145-524,288................1 (  0.0%)
total.........................39,944 (100.0%)
-------------------------------------------------------
task wait time profile
0..............................25,470 ( 63.8%)
1..............................23 (  0.1%)
2..............................5 (  0.0%)
3-4............................17 (  0.0%)
5-8............................25 (  0.1%)
9-16...........................45 (  0.1%)
17-32..........................121 (  0.3%)
33-64..........................8,620 ( 21.6%)
65-128.........................4,863 ( 12.2%)
129-256........................153 (  0.4%)
257-512........................5 (  0.0%)
513-1,024......................71 (  0.2%)
1,025-2,048....................7 (  0.0%)
2,049-4,096....................386 (  1.0%)
4,097-8,192....................131 (  0.3%)
16,385-32,768..................1 (  0.0%)
524,289-1,048,576..............1 (  0.0%)
total.........................39,944 (100.0%)
-------------------------------------------------------
instruction type profile
```

```
A-SHIFT.....17,460 (  7.1%)
ADJUST-COUNT.....9,327 (  3.8%)
AND.....26,190 ( 10.7%)
ANOTHER-NODE-ID.....3 (  0.0%)
ATTRIBUTE.....8,737 (  3.6%)
BRANCH-NOT-ZERO.....25,152 ( 10.3%)
BRANCH-ZERO.....4,367 (  1.8%)
CALL.....42,582 ( 17.4%)
CREATE-READ-WRITE-SEGMENT.....64 (  0.0%)
DESTROY-SEGMENT.....39,944 ( 16.3%)
INITIALIZE.....1 (  0.0%)
MINUS.....65 (  0.0%)
MOVE.....8,800 (  3.6%)
OR.....8,730 (  3.6%)
PLUS.....8,711 (  3.6%)
PRINT-USER.....104 (  0.0%)
RETURN.....8,736 (  3.6%)
SEND.....18,652 (  7.6%)
TEST-COUNT.....8,744 (  3.6%)
TIMES.....8,192 (  3.3%)
total.....244,561 (100.0%)

------------------------------------------------------------
instructions per messages              13.1

------------------------------------------------------------
instructions per task                   6.1

------------------------------------------------------------
instructions run length                 5.0

------------------------------------------------------------
operation type profile
A-SHIFT.....17,460 (  1.6%)
ADJUST-COUNT.....9,327 (  0.9%)
AND.....26,190 (  2.4%)
ANOTHER-NODE-ID.....526 (  0.0%)
ATTRIBUTE.....8,737 (  0.8%)
BRANCH-NOT-ZERO.....25,152 (  2.3%)
BRANCH-ZERO.....4,367 (  0.4%)
CALL.....42,582 (  3.9%)
COMPARE.....13,071 (  1.2%)
CREATE-READ-WRITE-SEGMENT.....64 (  0.0%)
DESTROY-SEGMENT.....39,944 (  3.7%)
MINUS.....65 (  0.0%)
MOD.....16,384 (  1.5%)
NODALS.....8,794 (  0.8%)
NODE-ID.....9,329 (  0.9%)
OR.....8,730 (  0.8%)
PLUS.....8,711 (  0.8%)
PRINT-USER.....104 (  0.0%)
READ.....295,187 ( 27.1%)
RETURN.....4,368 (  0.4%)
SELF.....431,365 ( 39.6%)
SEND.....18,652 (  1.7%)
TEST-COUNT.....8,744 (  0.8%)
TIMES.....8,192 (  0.8%)
WRITE.....82,583 (  7.6%)
```

```
total.....................1,088,628 (100.0%)
-----------------------------------------------------------
operations per instruction              4.5
-----------------------------------------------------------
segment size profile
3.....................................1 (  0.0%)
4.....................................8,192 ( 20.5%)
5.....................................2 (  0.0%)
6.....................................9,326 ( 23.3%)
7.....................................13,167 ( 32.9%)
8.....................................8,988 ( 22.5%)
9.....................................268 (  0.7%)
1024..................................64 (  0.2%)
total.....................40,008 (100.0%)
-----------------------------------------------------------
average segment size                    8.0
-----------------------------------------------------------
segment age profile
9-16..................................25,490 ( 63.8%)
17-32.................................106 (  0.3%)
33-64.................................106 (  0.3%)
65-128................................13,243 ( 33.2%)
129-256...............................395 (  1.0%)
257-512...............................7 (  0.0%)
513-1,024.............................70 (  0.2%)
1,025-2,048...........................8 (  0.0%)
2,049-4,096...........................386 (  1.0%)
4,097-8,192...........................130 (  0.3%)
8,193-16,384..........................1 (  0.0%)
16,385-32,768.........................1 (  0.0%)
1,048,577-2,097,152...................1 (  0.0%)
total.....................39,944 (100.0%)
-----------------------------------------------------------
last tick at 1,167,698
-----------------------------------------------------------
simulation time: 17 minutes 50 seconds
-----------------------------------------------------------
```

B.3 Set Synchronization

```
-----------------------------------------------------------
12 June 1990 11:24:28am set-sync-100-100 PiSim version 0.1
-----------------------------------------------------------
task type profile
ACK...................................126 (  0.5%)
ADJUST-BARRIER.....................12,600 ( 49.3%)
LEAF..................................100 (  0.4%)
REPLY-VALUE........................12,726 ( 49.8%)
```

```
SET-SYNC-TEST.....................1 (  0.0%)
STEM.............................26 (  0.1%)
total........................25,579 (100.0%)

------------------------------------------------------------
task status profile
NEW..........................25,579 ( 62.3%)
WAITING......................15,453 ( 37.7%)
total........................41,032 (100.0%)

------------------------------------------------------------
task run time profile
9-16.........................25,452 ( 99.5%)
2,049-4,096.....................1 (  0.0%)
8,193-16,384...................26 (  0.1%)
32,769-65,536..................100 (  0.4%)
total........................25,579 (100.0%)

------------------------------------------------------------
task wait time profile
0............................11,207 ( 43.8%)
1...............................100 (  0.4%)
2...............................202 (  0.8%)
3-4.............................1 (  0.0%)
5-8.............................402 (  1.6%)
9-16............................412 (  1.6%)
17-32...........................106 (  0.4%)
33-64.........................1,119 (  4.4%)
65-128........................3,543 ( 13.9%)
129-256.......................1,335 (  5.2%)
257-512.......................4,219 ( 16.5%)
513-1,024.....................2,606 ( 10.2%)
1,025-2,048...................200 (  0.8%)
131,073-262,144...............100 (  0.4%)
262,145-524,288...............27 (  0.1%)
total........................25,579 (100.0%)

------------------------------------------------------------
instruction type profile
ADJUST-COUNT..................15,453 (  1.3%)
ANOTHER-NODE-ID...............126 (  0.0%)
ATTRIBUTE.....................153 (  0.0%)
BRANCH-NOT-MINUS..............25 (  0.0%)
BRANCH-NOT-ZERO...........522,950 ( 44.9%)
BRANCH-ZERO...................5,601 (  0.5%)
CEILING.......................125 (  0.0%)
CREATE-READ-WRITE-SEGMENT.....53 (  0.0%)
DESTROY-SEGMENT...............25,632 (  2.2%)
INITIALIZE....................12,773 (  1.1%)
MINUS.........................500,250 ( 42.9%)
MOVE..........................38,330 (  3.3%)
PLUS..........................12,850 (  1.1%)
SEND..........................25,578 (  2.2%)
TEST-COUNT....................5,454 (  0.5%)
total....................1,165,353 (100.0%)

------------------------------------------------------------
instructions per messages              45.6

------------------------------------------------------------
```

```
instructions per task                    45.6
-------------------------------------------------------
instructions run length                  28.4
-------------------------------------------------------
operation type profile
ADJUST-COUNT.....................15,453 (  0.3%)
ANOTHER-NODE-ID..................126 (  0.0%)
ATTRIBUTE........................153 (  0.0%)
BRANCH-NOT-MINUS.....................25 (  0.0%)
BRANCH-NOT-ZERO..................522,950 ( 10.9%)
BRANCH-ZERO.....................5,601 (  0.1%)
CEILING.........................125 (  0.0%)
COMPARE.........................25,726 (  0.5%)
CREATE-READ-WRITE-SEGMENT.............53 (  0.0%)
DESTROY-SEGMENT.................25,632 (  0.5%)
MINUS...........................500,250 ( 10.4%)
NODE-ID.........................126 (  0.0%)
PLUS............................12,850 (  0.3%)
READ............................1,271,975 ( 26.5%)
SELF............................1,832,068 ( 38.2%)
SEND............................25,578 (  0.5%)
TEST-COUNT.....................5,454 (  0.1%)
WRITE...........................551,781 ( 11.5%)
total...........................4,795,926 (100.0%)
-------------------------------------------------------
operations per instruction               4.1
-------------------------------------------------------
segment size profile
1.....................................1 (  0.0%)
4.....................................5 (  0.0%)
5....................................47 (  0.2%)
6................................25,452 ( 99.3%)
9...................................101 (  0.4%)
14...................................26 (  0.1%)
total............................25,632 (100.0%)
-------------------------------------------------------
average segment size                     6.0
-------------------------------------------------------
segment age profile
9-16............................11,611 ( 45.3%)
17-32...............................818 (  3.2%)
33-64...............................710 (  2.8%)
65-128..............................3,645 ( 14.2%)
129-256.............................1,543 (  6.0%)
257-512.............................4,217 ( 16.5%)
513-1,024...........................2,708 ( 10.6%)
1,025-2,048.........................200 (  0.8%)
262,145-524,288.....................180 (  0.7%)
total............................25,632 (100.0%)
-------------------------------------------------------
last tick at 301,010
-------------------------------------------------------
simulation time: 22 minutes 2 seconds
-------------------------------------------------------
```

B.4 Object Name Translation

```
------------------------------------------------------------
12 June 1990 11:20:55am X-LATE-TEST PiSim version 0.1
------------------------------------------------------------
task type profile
ADJUST-BARRIER.....................204 (  7.7%)
AGENT-INSERT.......................102 (  3.9%)
AGENT-MATCH........................202 (  7.6%)
CLIENT-COMPARE.....................200 (  7.6%)
CLIENT-INSERT......................102 (  3.9%)
CLIENT-MATCH........................2 (  0.1%)
CLIENT-REMATCH......................1 (  0.0%)
CLOSEST-AGENT.....................625 ( 23.6%)
COMPARE-ALL-CLIENTS.................2 (  0.1%)
CREATE-AGENT.......................25 (  0.9%)
CREATE-CLIENT......................25 (  0.9%)
CREATE-MAP.........................10 (  0.4%)
INITIALIZE-TRANSLATIONS.............1 (  0.0%)
INSERT............................102 (  3.9%)
INSERT-IN-MAP.....................102 (  3.9%)
MAP-LOCATION-UPDATE...............250 (  9.4%)
MATCH.............................203 (  7.7%)
MATCH-IN-MAP......................202 (  7.6%)
REMATCH............................1 (  0.0%)
REPLY-VALUE.......................285 ( 10.8%)
TEST-1.............................1 (  0.0%)
TEST-2.............................1 (  0.0%)
X-LATE-TEST........................1 (  0.0%)
total.............................2,649 (100.0%)
------------------------------------------------------------
task status profile
CALL..............................308 (  9.0%)
NEW...............................2,341 ( 68.7%)
WAITING...........................757 ( 22.2%)
total.............................3,406 (100.0%)
------------------------------------------------------------
task run time profile
9-16..............................793 ( 29.9%)
17-32.............................758 ( 28.6%)
33-64............................1,093 ( 41.3%)
65-128.............................1 (  0.0%)
513-1,024..........................1 (  0.0%)
2,049-4,096........................3 (  0.1%)
total............................2,649 (100.0%)
------------------------------------------------------------
task wait time profile
0................................1,393 ( 52.6%)
```

```
1.....9 (  0.3%)
2.....63 (  2.4%)
3-4.....15 (  0.6%)
5-8.....15 (  0.6%)
9-16.....59 (  2.2%)
17-32.....79 (  3.0%)
33-64.....94 (  3.5%)
65-128.....399 ( 15.1%)
129-256.....85 (  3.2%)
257-512.....120 (  4.5%)
513-1,024.....56 (  2.1%)
1,025-2,048.....4 (  0.2%)
2,049-4,096.....14 (  0.5%)
4,097-8,192.....219 (  8.3%)
8,193-16,384.....24 (  0.9%)
16,385-32,768.....1 (  0.0%)
total.....2,649 (100.0%)
```

```
instruction type profile
ADJUST-COUNT.....310 (  2.2%)
ANOTHER-NODE-ID.....26 (  0.2%)
ATTRIBUTE.....358 (  2.5%)
BRANCH-NOT-MINUS.....600 (  4.3%)
BRANCH-NOT-ZERO.....578 (  4.1%)
BRANCH-ZERO.....2,318 ( 16.5%)
CALL.....614 (  4.4%)
CREATE-ASSOCIATIVE-SEGMENT.....25 (  0.2%)
CREATE-READ-WRITE-SEGMENT.....102 (  0.7%)
DESTROY-SEGMENT.....2,649 ( 18.9%)
DISTANCE.....61 (  0.4%)
INITIALIZE.....5 (  0.0%)
INSERT.....506 (  3.6%)
MATCH.....203 (  1.4%)
MOD.....604 (  4.3%)
MOVE.....895 (  6.4%)
NODE-ID.....62 (  0.4%)
PLUS.....1,235 (  8.8%)
PRINT-USER.....3 (  0.0%)
REMOVE.....1 (  0.0%)
RETURN.....205 (  1.5%)
SEND.....2,479 ( 17.6%)
TEST-COUNT.....210 (  1.5%)
total.....14,049 (100.0%)
```

```
instructions per messages                6.0
```

```
instructions per task                    5.3
```

```
instructions run length                  4.5
```

```
operation type profile
ADJUST-COUNT.....310 (  0.4%)
ANOTHER-NODE-ID.....61 (  0.1%)
ATTRIBUTE.....358 (  0.4%)
```

```
BRANCH-NOT-MINUS.....600 (  0.7%)
BRANCH-NOT-ZERO.....578 (  0.7%)
BRANCH-ZERO.....2,315 (  2.7%)
CALL.....616 (  0.7%)
COMPARE.....2,891 (  3.3%)
CREATE-ASSOCIATIVE-SEGMENT.....35 (  0.0%)
CREATE-READ-WRITE-SEGMENT.....102 (  0.1%)
DESTROY-SEGMENT.....2,649 (  3.1%)
DISTANCE.....661 (  0.8%)
INSERT.....304 (  0.4%)
MATCH.....405 (  0.5%)
MOD.....604 (  0.7%)
NODE-ID.....1,221 (  1.4%)
PLUS.....1,235 (  1.4%)
PRINT-USER.....3 (  0.0%)
READ.....32,193 ( 37.1%)
REMOVE.....1 (  0.0%)
RETURN.....204 (  0.2%)
SELF.....33,182 ( 38.2%)
SEND.....2,340 (  2.7%)
TEST-COUNT.....210 (  0.2%)
TIMES.....301 (  0.3%)
WRITE.....3,421 (  3.9%)
total.....86,800 (100.0%)

------------------------------------------------------------
operations per instruction              6.2
------------------------------------------------------------
segment size profile
3.....25 (  0.9%)
4.....25 (  0.9%)
5.....1 (  0.0%)
6.....702 ( 25.2%)
7.....302 ( 10.8%)
8.....455 ( 16.3%)
9.....437 ( 15.7%)
10.....827 ( 29.7%)
25.....12 (  0.4%)
total.....2,786 (100.0%)

------------------------------------------------------------
average segment size                    8.1
------------------------------------------------------------
segment age profile
9-16.....604 ( 22.8%)
17-32.....261 (  9.9%)
33-64.....786 ( 29.7%)
65-128.....258 (  9.7%)
129-256.....293 ( 11.1%)
257-512.....113 (  4.3%)
513-1,024.....71 (  2.7%)
1,025-2,048.....3 (  0.1%)
2,049-4,096.....15 (  0.6%)
4,097-8,192.....218 (  8.2%)
8,193-16,384.....26 (  1.0%)
16,385-32,768.....1 (  0.0%)
```

```
total.............................2,649 (100.0%)
-------------------------------------------------------
last tick at 18,440
-------------------------------------------------------
simulation time: 44 seconds
-------------------------------------------------------
```

B.5 Non-Resident Handlers

```
-------------------------------------------------------
12 June 1990 11:22:41am NR-HANDLER-TEST PiSim version 0.1
-------------------------------------------------------
task type profile
ADD-HANDLER.............................1 (  0.0%)
ADJUST-BARRIER.........................67 (  2.2%)
AGENT-INSERT............................1 (  0.0%)
AGENT-MATCH............................64 (  2.1%)
CLOSEST-AGENT.....................1,600 ( 53.0%)
CREATE-AGENT...........................25 (  0.8%)
CREATE-CLIENT..........................64 (  2.1%)
CREATE-MAP.............................10 (  0.3%)
CREATE-NODE-MAPS.......................64 (  2.1%)
DISPATCH..............................128 (  4.2%)
FETCH-HANDLER..........................64 (  2.1%)
INITIALIZE-NR-HANDLERS..................1 (  0.0%)
INITIALIZE-TRANSLATIONS.................1 (  0.0%)
INSERT..................................1 (  0.0%)
INSERT-IN-MAP...........................1 (  0.0%)
LOOKUP-HANDLER.........................97 (  3.2%)
MAP-LOCATION-UPDATE....................250 (  8.3%)
MATCH..................................64 (  2.1%)
MATCH-IN-MAP...........................64 (  2.1%)
NR-HANDLER-TEST.........................1 (  0.0%)
REPLY-HANDLER..........................64 (  2.1%)
REPLY-VALUE...........................261 (  8.6%)
TEST-NR-HANDLER.......................128 (  4.2%)
total.............................3,021 (100.0%)
-------------------------------------------------------
task status profile
CALL...................................290 (  7.9%)
NEW................................2,731 ( 74.9%)
WAITING................................627 ( 17.2%)
total..............................3,648 (100.0%)
-------------------------------------------------------
task run time profile
9-16...................................521 ( 17.2%)
17-32..................................247 (  8.2%)
33-64..............................2,058 ( 68.1%)
```

```
65-128.....65 (  2.2%)
129-256.....128 (  4.2%)
1,025-2,048.....2 (  0.1%)
total.....3,021 (100.0%)

------------------------------------------------------------
task wait time profile
0.....1,720 ( 56.9%)
2.....120 (  4.0%)
3-4.....11 (  0.4%)
5-8.....15 (  0.5%)
9-16.....141 (  4.7%)
17-32.....69 (  2.3%)
33-64.....142 (  4.7%)
65-128.....112 (  3.7%)
129-256.....115 (  3.8%)
257-512.....76 (  2.5%)
513-1,024.....158 (  5.2%)
1,025-2,048.....112 (  3.7%)
2,049-4,096.....219 (  7.2%)
4,097-8,192.....11 (  0.4%)
total.....3,021 (100.0%)

------------------------------------------------------------
instruction type profile
ADJUST-COUNT.....71 (  0.3%)
ANOTHER-NODE-ID.....1 (  0.0%)
ATTRIBUTE.....195 (  0.8%)
BRANCH-MINUS.....960 (  4.0%)
BRANCH-NOT-MINUS.....1,536 (  6.4%)
BRANCH-NOT-ZERO.....1,043 (  4.4%)
BRANCH-PLUS.....1 (  0.0%)
BRANCH-ZERO.....4,998 ( 20.9%)
CALL.....324 (  1.4%)
CALL-SEGMENT.....256 (  1.1%)
CREATE-ASSOCIATIVE-SEGMENT.....256 (  1.1%)
CREATE-READ-WRITE-SEGMENT.....396 (  1.7%)
DESTROY-SEGMENT.....3,149 ( 13.2%)
DISTANCE.....187 (  0.8%)
INITIALIZE.....453 (  1.9%)
INSERT.....258 (  1.1%)
MATCH.....514 (  2.1%)
MINUS.....193 (  0.8%)
MOD.....65 (  0.3%)
MOVE.....1,832 (  7.7%)
NODE-ID.....188 (  0.8%)
PLUS.....3,743 ( 15.7%)
READ.....194 (  0.8%)
REMOVE.....64 (  0.3%)
RETURN.....64 (  0.3%)
SEND.....2,774 ( 11.6%)
SEND-SEGMENT.....64 (  0.3%)
TEST-COUNT.....8 (  0.0%)
TIMES.....128 (  0.5%)
total.....23,915 (100.0%)

------------------------------------------------------------
```

```
instructions per messages                8.8
------------------------------------------------------------
instructions per task                    7.9
------------------------------------------------------------
instructions run length                  7.1
------------------------------------------------------------
operation type profile
ADJUST-COUNT.............................71 (  0.1%)
ANOTHER-NODE-ID.........................38 (  0.0%)
ATTRIBUTE...............................195 (  0.1%)
BRANCH-MINUS............................960 (  0.7%)
BRANCH-NOT-MINUS.....................1,536 (  1.1%)
BRANCH-NOT-ZERO......................1,043 (  0.8%)
BRANCH-PLUS.............................1 (  0.0%)
BRANCH-ZERO..........................4,998 (  3.7%)
CALL....................................324 (  0.2%)
CALL-SEGMENT............................256 (  0.2%)
COMPARE.............................6,296 (  4.6%)
CREATE-ASSOCIATIVE-SEGMENT..............266 (  0.2%)
CREATE-READ-WRITE-SEGMENT...............396 (  0.3%)
DESTROY-SEGMENT.....................3,149 (  2.3%)
DISTANCE............................1,723 (  1.3%)
INSERT..................................194 (  0.1%)
MATCH...................................578 (  0.4%)
MINUS...................................321 (  0.2%)
MOD.....................................65 (  0.0%)
NODALS..............................1,093 (  0.8%)
NODE-ID.............................2,337 (  1.7%)
PLUS................................3,743 (  2.7%)
PROBE...................................640 (  0.5%)
READ...............................45,674 ( 33.5%)
REMOVE..................................64 (  0.0%)
RETURN..................................64 (  0.0%)
SELF...............................49,655 ( 36.4%)
SEND................................2,666 (  2.0%)
SEND-SEGMENT............................64 (  0.0%)
TEST-COUNT..............................8 (  0.0%)
TIMES..................................128 (  0.1%)
WRITE...............................7,956 (  5.8%)
total..............................136,502 (100.0%)
------------------------------------------------------------
operations per instruction               5.7
------------------------------------------------------------
segment size profile
1.......................................64 (  1.7%)
3......................................129 (  3.5%)
4......................................153 (  4.2%)
5......................................193 (  5.2%)
6......................................499 ( 13.5%)
7.......................................65 (  1.8%)
8......................................442 ( 12.0%)
9......................................258 (  7.0%)
10.................................1,741 ( 47.3%)
12.....................................1 (  0.0%)
```

```
14.....................................128 (  3.5%)
25.....................................10 (  0.3%)
total..............................3,683 (100.0%)
------------------------------------------------------------
average segment size                    8.4
------------------------------------------------------------
segment age profile
9-16...................................301 (  9.6%)
17-32..................................204 (  6.5%)
33-64.............................1,455 ( 46.2%)
65-128.................................301 (  9.6%)
129-256................................285 (  9.1%)
257-512.................................75 (  2.4%)
513-1,024..............................180 (  5.7%)
1,025-2,048............................113 (  3.6%)
2,049-4,096............................216 (  6.9%)
4,097-8,192.............................18 (  0.6%)
8,193-16,384............................1 (  0.0%)
total..............................3,149 (100.0%)
------------------------------------------------------------
last tick at 12,444
------------------------------------------------------------
simulation time: 50 seconds
------------------------------------------------------------
```

B.6 N-Body Simulation (10 Bodies)

```
------------------------------------------------------------
12 June 1990 1:12:04pm n-body-10-1000 PiSim version 0.1
------------------------------------------------------------
task type profile
ADD-ACCELERATION.....................90,000 ( 50.0%)
INITIALIZE-SYSTEM.......................1 (  0.0%)
REPLY-VALUE.............................10 (  0.0%)
START-BODY..............................10 (  0.0%)
START-INTERACTION.......................45 (  0.0%)
UPDATE-POSITION.....................90,000 ( 50.0%)
WRITE-INTERACTION-LOCATION..............90 (  0.0%)
total..............................180,156 (100.0%)
------------------------------------------------------------
task status profile
NEW................................180,156 ( 74.7%)
WAITING............................61,086 ( 25.3%)
total..............................241,242 (100.0%)
------------------------------------------------------------
task run time profile
9-16...................................10 (  0.0%)
17-32..............................180,090 (100.0%)
```

```
2,049-4,096.....1 (  0.0%)
65,537-131,072.....39 (  0.0%)
131,073-262,144.....6 (  0.0%)
262,145-524,288.....10 (  0.0%)
total.....180,156 (100.0%)
------------------------------------------------------------
task wait time profile
0.....101,644 ( 56.4%)
1.....4 (  0.0%)
2.....1,994 (  1.1%)
3-4.....385 (  0.2%)
5-8.....4,017 (  2.2%)
9-16.....1,822 (  1.0%)
17-32.....5,218 (  2.9%)
33-64.....14,020 (  7.8%)
65-128.....17,019 (  9.4%)
129-256.....26,986 ( 15.0%)
257-512.....6,992 (  3.9%)
524,289-1,048,576.....55 (  0.0%)
total.....180,156 (100.0%)
------------------------------------------------------------
instruction type profile
ADJUST-COUNT.....100,000 (  4.9%)
ANOTHER-NODE-ID.....10 (  0.0%)
ATTRIBUTE.....190 (  0.0%)
BRANCH-NOT-MINUS.....1 (  0.0%)
BRANCH-NOT-ZERO.....246,264 ( 12.1%)
CREATE-READ-WRITE-SEGMENT.....22 (  0.0%)
DESTROY-SEGMENT.....180,178 (  8.9%)
EXPONENT.....45,000 (  2.2%)
INITIALIZE.....30,078 (  1.5%)
MINUS.....141,063 (  7.0%)
MOVE.....180,380 (  8.9%)
PLUS.....410,163 ( 20.2%)
PRINT-USER.....220 (  0.0%)
RANDOM.....20 (  0.0%)
SEND.....180,224 (  8.9%)
TEST-COUNT.....20,000 (  1.0%)
TIMES.....495,090 ( 24.4%)
total.....2,028,903 (100.0%)
------------------------------------------------------------
instructions per messages              11.3
------------------------------------------------------------
instructions per task                  11.3
------------------------------------------------------------
instructions run length                 8.4
------------------------------------------------------------
operation type profile
ADJUST-COUNT.....100,000 (  0.8%)
ANOTHER-NODE-ID.....56 (  0.0%)
ATTRIBUTE.....190 (  0.0%)
BRANCH-NOT-MINUS.....1 (  0.0%)
BRANCH-NOT-ZERO.....246,264 (  1.9%)
COMPARE.....235,064 (  1.8%)
```

```
CREATE-READ-WRITE-SEGMENT.....22 (  0.0%)
DESTROY-SEGMENT.....180,178 (  1.4%)
EXPONENT.....45,000 (  0.4%)
MINUS.....90,047 (  0.7%)
MOD.....1,200 (  0.0%)
NODE-ID.....100 (  0.0%)
PLUS.....410,163 (  3.2%)
PRINT-USER.....220 (  0.0%)
RANDOM.....20 (  0.0%)
READ.....4,376,106 ( 34.4%)
SELF.....5,077,747 ( 40.0%)
SEND.....180,155 (  1.4%)
TEST-COUNT.....20,000 (  0.2%)
TIMES.....495,090 (  3.9%)
WRITE.....1,250,809 (  9.8%)
total.....12,708,432 (100.0%)
------------------------------------------------------------
operations per instruction              6.3
------------------------------------------------------------
segment size profile
6.....90,010 ( 50.0%)
7.....90,000 ( 50.0%)
9.....110 (  0.1%)
10.....2 (  0.0%)
11.....1 (  0.0%)
19.....10 (  0.0%)
23.....45 (  0.0%)
total.....180,178 (100.0%)
------------------------------------------------------------
average segment size                    6.5
------------------------------------------------------------
segment age profile
17-32.....108,053 ( 60.0%)
33-64.....8,045 (  4.5%)
65-128.....25,020 ( 13.9%)
129-256.....25,997 ( 14.4%)
257-512.....12,985 (  7.2%)
2,049-4,096.....3 (  0.0%)
524,289-1,048,576.....75 (  0.0%)
total.....180,178 (100.0%)
------------------------------------------------------------
last tick at 851,237
------------------------------------------------------------
simulation time: 1 hour 10 minutes 57 seconds
------------------------------------------------------------
```

B.7 N-Body Simulation (100 Bodies)

```
------------------------------------------------------------
12 June 1990 3:16:48pm n-body-100-10 PiSim version 0.1
------------------------------------------------------------
task type profile
ADD-ACCELERATION.....................99,000 ( 46.5%)
INITIALIZE-SYSTEM.....................1 (  0.0%)
REPLY-VALUE..........................100 (  0.0%)
START-BODY...........................100 (  0.0%)
START-INTERACTION....................4,950 (  2.3%)
UPDATE-POSITION.....................99,000 ( 46.5%)
WRITE-INTERACTION-LOCATION..........9,900 (  4.6%)
total..............................213,051 (100.0%)
------------------------------------------------------------
task status profile
NEW................................213,051 ( 74.9%)
WAITING............................71,324 ( 25.1%)
total..............................284,375 (100.0%)
------------------------------------------------------------
task run time profile
9-16.................................100 (  0.0%)
17-32..............................207,900 ( 97.6%)
1,025-2,048.........................4,950 (  2.3%)
16,385-32,768.......................100 (  0.0%)
131,073-262,144.....................1 (  0.0%)
total..............................213,051 (100.0%)
------------------------------------------------------------
task wait time profile
0..................................36,647 ( 17.2%)
1..................................196 (  0.1%)
2..................................174 (  0.1%)
3-4................................342 (  0.2%)
5-8................................956 (  0.4%)
9-16..............................2,255 (  1.1%)
17-32.............................2,659 (  1.2%)
33-64.............................3,931 (  1.8%)
65-128............................7,697 (  3.6%)
129-256...........................6,706 (  3.1%)
257-512...........................6,717 (  3.2%)
513-1,024........................12,529 (  5.9%)
1,025-2,048......................39,767 ( 18.7%)
2,049-4,096......................48,520 ( 22.8%)
4,097-8,192......................32,736 ( 15.4%)
8,193-16,384.....................5,883 (  2.8%)
16,385-32,768.....................24 (  0.0%)
32,769-65,536.....................73 (  0.0%)
65,537-131,072...................132 (  0.1%)
131,073-262,144...................57 (  0.0%)
262,145-524,288..................4,979 (  2.3%)
524,289-1,048,576.....................71 (  0.0%)
total..............................213,051 (100.0%)
------------------------------------------------------------
instruction type profile
ADJUST-COUNT.....................100,000 (  4.5%)
ANOTHER-NODE-ID...................100 (  0.0%)
```

```
ATTRIBUTE.....................19,900 (  0.9%)
BRANCH-NOT-MINUS.....................1 (  0.0%)
BRANCH-NOT-ZERO..................254,661 ( 11.3%)
CREATE-READ-WRITE-SEGMENT.............202 (  0.0%)
DESTROY-SEGMENT..................213,253 (  9.5%)
EXPONENT.........................49,500 (  2.2%)
INITIALIZE.......................8,253 (  0.4%)
MINUS...........................170,765 (  7.6%)
MOVE............................237,800 ( 10.6%)
PLUS............................416,148 ( 18.5%)
PRINT-USER.........................2 (  0.0%)
RANDOM...........................200 (  0.0%)
SEND............................216,560 (  9.7%)
TEST-COUNT.......................2,000 (  0.1%)
TIMES...........................554,400 ( 24.7%)
total..........................2,243,745 (100.0%)
-------------------------------------------------------
instructions per messages              10.5
-------------------------------------------------------
instructions per task                  10.5
-------------------------------------------------------
instructions run length                 7.9
-------------------------------------------------------
operation type profile
ADJUST-COUNT....................100,000 (  0.7%)
ANOTHER-NODE-ID...................5,052 (  0.0%)
ATTRIBUTE.....................19,900 (  0.1%)
BRANCH-NOT-MINUS.....................1 (  0.0%)
BRANCH-NOT-ZERO..................254,661 (  1.8%)
COMPARE..........................253,649 (  1.8%)
CREATE-READ-WRITE-SEGMENT.............202 (  0.0%)
DESTROY-SEGMENT..................213,253 (  1.5%)
EXPONENT.........................49,500 (  0.3%)
MINUS...........................103,952 (  0.7%)
MOD..................................12 (  0.0%)
NODE-ID.........................10,000 (  0.1%)
PLUS............................416,148 (  2.9%)
PRINT-USER.........................2 (  0.0%)
RANDOM...........................200 (  0.0%)
READ...........................4,956,243 ( 34.9%)
SELF...........................5,685,447 ( 40.0%)
SEND............................213,050 (  1.5%)
TEST-COUNT.......................2,000 (  0.0%)
TIMES...........................554,400 (  3.9%)
WRITE..........................1,370,554 (  9.6%)
total.........................14,208,226 (100.0%)
-------------------------------------------------------
operations per instruction              6.3
-------------------------------------------------------
segment size profile
6................................99,100 ( 46.5%)
7................................99,000 ( 46.4%)
9.................................9,900 (  4.6%)
11.....................................1 (  0.0%)
```

```
19.....100 (  0.0%)
23.....4,950 (  2.3%)
99.....200 (  0.1%)
100.....2 (  0.0%)
total.....213,253 (100.0%)
------------------------------------------------------------
average segment size                    7.1
------------------------------------------------------------
segment age profile
17-32.....39,430 ( 18.5%)
33-64.....5,300 (  2.5%)
65-128.....7,582 (  3.6%)
129-256.....8,461 (  4.0%)
257-512.....7,057 (  3.3%)
513-1,024.....12,275 (  5.8%)
1,025-2,048.....39,674 ( 18.6%)
2,049-4,096.....48,985 ( 23.0%)
4,097-8,192.....33,014 ( 15.5%)
8,193-16,384.....5,936 (  2.8%)
16,385-32,768.....24 (  0.0%)
32,769-65,536.....73 (  0.0%)
65,537-131,072.....132 (  0.1%)
131,073-262,144.....60 (  0.0%)
262,145-524,288.....4,871 (  2.3%)
524,289-1,048,576.....379 (  0.2%)
total.....213,253 (100.0%)
------------------------------------------------------------
last tick at 559,691
------------------------------------------------------------
simulation time: 2 hours 27 minutes 34 seconds
------------------------------------------------------------
```

B.8 Relaxation

```
------------------------------------------------------------
12 June 1990 0:56:52pm relax-100 PiSim version 0.1
------------------------------------------------------------
task type profile
RELAX.....1 (  0.0%)
REPLY-VALUE.....100 (  0.3%)
START-ELEMENT.....100 (  0.3%)
UPDATE-TEMP.....36,000 ( 99.4%)
total.....36,201 (100.0%)
------------------------------------------------------------
task status profile
NEW.....36,201 ( 62.1%)
WAITING.....22,110 ( 37.9%)
total.....58,311 (100.0%)
```

```
------------------------------------------------------------
task run time profile
9-16.....................................100 (  0.3%)
17-32...............................35,965 ( 99.3%)
33-64....................................15 (  0.0%)
65-128...................................17 (  0.0%)
129-256...................................3 (  0.0%)
4,097-8,192..............................24 (  0.1%)
8,193-16,384.............................77 (  0.2%)
total...............................36,201 (100.0%)
------------------------------------------------------------
task wait time profile
0.....................................7,923 ( 21.9%)
1.....................................190 (  0.5%)
2.....................................295 (  0.8%)
3-4...................................517 (  1.4%)
5-8...................................1,324 (  3.7%)
9-16..................................2,394 (  6.6%)
17-32.................................3,122 (  8.6%)
33-64.................................4,181 ( 11.5%)
65-128................................5,896 ( 16.3%)
129-256...............................6,379 ( 17.6%)
257-512...............................3,795 ( 10.5%)
513-1,024.................................57 (  0.2%)
1,025-2,048...............................27 (  0.1%)
16,385-32,768.............................100 (  0.3%)
32,769-65,536.............................1 (  0.0%)
total...............................36,201 (100.0%)
------------------------------------------------------------
instruction type profile
ATTRIBUTE.................................440 (  0.1%)
BRANCH-MINUS.........................10,200 (  3.3%)
BRANCH-NOT-MINUS.....................110 (  0.0%)
BRANCH-NOT-ZERO.....................36,447 ( 11.9%)
BRANCH-ZERO.........................40,247 ( 13.2%)
CREATE-ASSOCIATIVE-SEGMENT..............64 (  0.0%)
CREATE-READ-WRITE-SEGMENT...............1 (  0.0%)
DESTROY-SEGMENT.....................36,202 ( 11.8%)
DIVIDE..............................10,000 (  3.3%)
INITIALIZE..............................281 (  0.1%)
INSERT..................................100 (  0.0%)
MATCH...............................36,201 ( 11.8%)
MINUS...................................211 (  0.1%)
MOD.....................................500 (  0.2%)
MOVE................................44,432 ( 14.5%)
PLUS................................53,975 ( 17.6%)
PRINT-USER..............................113 (  0.0%)
SEND................................36,200 ( 11.8%)
SUSPEND.................................147 (  0.0%)
TIMES.....................................2 (  0.0%)
total..............................305,873 (100.0%)
------------------------------------------------------------
instructions per messages                8.4
------------------------------------------------------------
```

```
instructions per task                          8.4
-------------------------------------------------------------
instructions run length                        5.2
-------------------------------------------------------------
operation type profile
ATTRIBUTE.....................................440 (  0.0%)
BRANCH-MINUS.............................10,200 (  0.6%)
BRANCH-NOT-MINUS.........................110 (  0.0%)
BRANCH-NOT-ZERO..........................36,447 (  2.2%)
BRANCH-ZERO..............................40,247 (  2.5%)
COMPARE..................................86,547 (  5.3%)
CREATE-ASSOCIATIVE-SEGMENT.....64 (  0.0%)
CREATE-READ-WRITE-SEGMENT.....1 (  0.0%)
DESTROY-SEGMENT..........................36,202 (  2.2%)
DIVIDE...................................10,000 (  0.6%)
INSERT.....................................100 (  0.0%)
MATCH....................................36,147 (  2.2%)
MINUS......................................411 (  0.0%)
MOD........................................800 (  0.0%)
NODALS...................................36,465 (  2.2%)
NODE-ID....................................100 (  0.0%)
PLUS.....................................40,400 (  2.5%)
PRINT-USER.................................111 (  0.0%)
PROBE......................................100 (  0.0%)
READ....................................512,826 ( 31.5%)
SELF....................................614,335 ( 37.7%)
SEND.....................................36,200 (  2.2%)
SUSPEND....................................147 (  0.0%)
TIMES......................................112 (  0.0%)
WRITE...................................132,038 (  8.1%)
total..................................1,630,550 (100.0%)
-------------------------------------------------------------
operations per instruction                     5.3
-------------------------------------------------------------
segment size profile
2..........................................64 (  0.2%)
6.........................................100 (  0.3%)
7......................................36,000 ( 99.3%)
8..........................................1 (  0.0%)
21........................................100 (  0.3%)
100........................................1 (  0.0%)
total..................................36,266 (100.0%)
-------------------------------------------------------------
average segment size                           7.0
-------------------------------------------------------------
segment age profile
9-16.......................................4 (  0.0%)
17-32...................................11,427 ( 31.6%)
33-64.....................................5,991 ( 16.5%)
65-128....................................6,521 ( 18.0%)
129-256...................................7,015 ( 19.4%)
257-512...................................5,045 ( 13.9%)
513-1,024.....................................65 (  0.2%)
1,025-2,048...................................32 (  0.1%)
```

```
32,769-65,536.....102 (  0.3%)
total.....38,202 (100.0%)
-------------------------------------------------------
last tick at 38,200
-------------------------------------------------------
simulation time: 10 minutes 35 seconds
-------------------------------------------------------
```

CS-TR Scanning Project
Document Control Form

Date : 6/22/95

Report # AI-TR-1245

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 196(204-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1) UNNUMBERED TITLE PAGE	
(2-196) PAGES NUMBERED 2-196	
(197-201) SCANCONTROL, COVER, SPINE, DOD(2)	
(202-204) TRGT'S (3)	

Scanning Agent Signoff:

Date Received: 6/22/95 Date Scanned: 6/26/95 Date Returned: 6/29/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1245	2. GOVT ACCESSION NO. AD-A228345	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Pi: A Parallel Architecture Interface for Multi-Model Execution		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Donald Scott Wills		8. CONTRACT OR GRANT NUMBER(s) N00014-88-K0738, N00014-87-K-0825 N00014-85-K0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE July 1990 13. NUMBER OF PAGES 196
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

interface
parallel architecture
multi-model execution

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Existing parallel architectures are constructed monolithically, with no well defined boundaries separating model and machine issues. This makes it difficult to evaluate the effect of a single component of an architecture, or compare it with the corresponding components of other architectures. Machine hardware is specialized to support a single programming model, even though similar but more general mechanisms could

(con't on back)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(block 20 con't)

support a variety of models. Often details of an implementation become visible in the programming environment, restricting future implementation improvements because of compatibility. The lack of an interface between model and machine issues also complicates the translation of a machine improvement into a performance improvement for model applications.

This thesis defines Pi, a parallel architecture interface that separates model and machine issues, allowing them to be addressed independently. This provides greater flexibility for both the model and machine builder. Pi addresses a set of common parallel model requirements including low latency communication, fast task switching, low cost synchronization, efficient storage management, the ability to exploit locality, and efficient support for sequential code. Since Pi provides generic parallel operations, it can efficiently support many parallel programming models including hybrids of existing models. Pi also forms a basis of comparison for architectural components.

Pi is evaluated in two ways. First, several mechanisms required by existing parallel models are constructed on the interface. These examples are executed and evaluated using a Pi simulation environment. Then a machine substrate that supports the interface is specified. It is designed to efficiently support the generic parallel operations provided in Pi. The feasibility of gate array implementation is considered. The effectiveness of this machine substrate at supporting Pi is evaluated. The role of a machine compiler below the interface is also discussed.

This thesis demonstrates a parallel architecture interface. Pi efficiently supports several model mechanisms. A machine substrate which effectively implements Pi is presented.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 6/26/1995

M.I.T. Libraries
Document Services