Distribution of Linda Across a Network of Workstations

by

Charles N. Schumann, Jr.

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

APPROVED:

James D. Arthur, Chairman

Dennis G. Kafura

Sallie M. Henry

December 1993
Blacksburg, Virginia
LD
5655
V855
1993
S800
C.2
Distribution of Linda Across a Network of Workstations

by

Charles N. Schumann, Jr.
James D. Arthur, Chairman

Computer Science

(ABSTRACT)

The Linda programming language provides an architecturally independent paradigm for writing parallel programs. By designing and implementing Linda on a network of stand alone workstations a scalable multicomputer can be constructed from existing equipment. This thesis presents the design, implementation and testing of a distributable Linda kernel and communications subsystem providing a framework for full distribution of Linda on a network of workstations. Following a presentation of the Linda language, the kernel's design and rationale are presented. The design provides for interprocess communications by implementing a protocol on top of the Unix socket facility. Choosing sockets as the interprocess communications medium has the advantage of wide portability. However, a design critique is presented which addresses several disadvantages of the socket based communications model. Considerable attention is given to quantifying the effectiveness of this design in comparison to a shared memory, non-distributable design from Yale University. A thorough investigation into the source of particular observed phenomena is presented which leads to an improvement in wall time performance of an order of magnitude.
Acknowledgements

There are several people whom I must acknowledge as having contributed to the completion of this thesis. I am obliged to give special thanks and recognition to my committee chairman, Dr. James Arthur. He, against his better judgement, allowed me to join the Linda research group and take on commitments inconsistent with my other roles as husband, father, and full time employee. He has been most understanding and supportive during the long process of research and thesis writing. I have learned a great deal from him as I have been challenged to communicate clearly, encouraged to continue the effort, and forced to defend some of my ideas. I have gained considerably from studying under Dr. Arthur, although I haven't always been willing to admit same.

My most heartfelt appreciation goes out to my wife, Linda, who has, more than any other, supported this effort. She has given me strength during the many late nights that have been necessary to complete the course work and research. Her love for me and our two children has truly sustained me during the difficult times. Without her I would never have "stuck with it" long enough to complete this degree program. It is with love that I dedicate this thesis to her.

I am also indebted to some of my friends and colleagues at Hercules, Inc. Particularly, Chip Batton, Steve Ribble and Denny Carlyle have offered their advice, read drafts and regularly
reminded me to get on with it. Thanks guys.

Finally, I am privileged to be a part of the "peculiar people" who make up the Blacksburg Christian Fellowship. The Fellowship has been a significant part of our family since we have been in Blacksburg. The leadership and friends I have found there have challenged me to seek out and follow Jesus Christ.
# Table of Contents

1.0 Introduction ........................................................................................................ 1

1.1 Background and Motivation ............................................................................. 1

1.2 Problem Statement and Proposed Solution .................................................... 2

1.3 Plan of Thesis .................................................................................................... 3

2.0 Linda Background .............................................................................................. 5

2.1 The Linda Paradigm .......................................................................................... 5

2.1.1 Tuple Space ................................................................................................... 5

2.1.2 The Linda Programmer’s View of Tuple Space ............................................. 6

2.1.3 Static Tuples ................................................................................................ 9

2.1.4 Live Tuples .................................................................................................. 9

2.1.5 Retrieval of Tuples From Tuple Space ...................................................... 12

2.1.6 Template and Tuple Matching .................................................................... 12

2.2 The Linda Compiler ........................................................................................ 14

2.2.1 Corresponding Field Count and Type ....................................................... 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Separate Kernel Process</td>
<td>43</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Portable Communications Subsystem</td>
<td>43</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Working Model for Further Research</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>VPI Kernel Implementation Issues</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Message Based Kernel Services</td>
<td>46</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Temporary Buffer Management</td>
<td>54</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Tuple Space Memory Management</td>
<td>54</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Debugging and Tracing Facilities</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Design Appraisal</td>
<td>57</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Serialized Access to Tuple Space</td>
<td>58</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Redundant Memory Copying Operations</td>
<td>59</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Adverse Affects of Unix Scheduler</td>
<td>60</td>
</tr>
<tr>
<td>4.0</td>
<td>Initial Benchmark Results</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>The <code>Fastloop.cl</code> Benchmark</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>The <code>Cmatrix.cl</code> Benchmark</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>Observations and Interpretation</td>
<td>77</td>
</tr>
<tr>
<td>4.3.1</td>
<td>CPU Utilization</td>
<td>78</td>
</tr>
<tr>
<td>4.3.2</td>
<td>VPI Kernel Distribution</td>
<td>78</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Foreground and Background Processing</td>
<td>79</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Cross Platform Comparison</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>Conclusions and Further Research Options</td>
<td>80</td>
</tr>
<tr>
<td>5.0</td>
<td>Tuple Prefetching</td>
<td>82</td>
</tr>
<tr>
<td>5.1</td>
<td>Prefetch.cl</td>
<td>82</td>
</tr>
</tbody>
</table>

Table of Contents
List of Illustrations

Figure 2.1 Live and Static Tuples Stored in Tuple Space. ......................... 7
Figure 3.1 Yale implementation of Tuple Space in Shared Memory. ............... 33
Figure 3.2 VPI Kernel Design ......................................................... 44
Figure 3.3 Yale tuple/template data structure. ................................... 49
Figure 3.4 VPI Serialized tuple/template structure. .............................. 51
Figure 3.5 Redundant Memory Copy Operations. ................................. 61
Figure 4.1 Yale, VPI(i) and VPI(ii) Wall Time Comparison. .................... 71
Figure 4.2 Yale, VPI(i) and VPI(ii) Application Code Time. ................... 73
Figure 4.3 Yale, VPI(i) and VPI(ii) System Code Times. ........................ 74
Figure 4.4 Yale and VPI Cmatrix.cl Matrix Size versus Wall Time. ........... 76
Figure 5.1 Prefetch.cl initial results. ............................................. 85
Figure 5.2 Priming and Retrieval Phases of Prefetch.cl. ........................ 89
Figure 5.3 Prefetch.cl using rd() Operations ..................................... 92
Figure 5.4 Forced Program Suspension. ........................................... 97
Figure 5.5 Using socket option TCP_NODELAY in the application. ............. 102
Figure 5.6 Using TCP_NODELAY in the application and the kernel. ............ 105
Figure 6.1 Fastloop.cl Comparison on MacIntosh ................................ 111
Figure 6.2 Fastloop.cl Comparison on RS/6000 .................................. 113
Figure 6.3 Bulk.cl Comparison on MacIntosh .................................... 115
| Figure 6.4 | VPI(ii) running Bulk.cl, Block Size versus Wall Time | 117 |
| Figure 6.5 | VPI(ii) running Bulk.cl, Number of Blocks versus Wall Time | 118 |
| Figure 6.6 | Bulk.cl Comparison on RS/6000 | 120 |
| Figure 6.7 | Bulk.cl comparison on RS/6000 | 122 |
| Figure 6.8 | Eval.cl Comparison on RS/6000 | 124 |
| Figure 6.9 | Prime_act.cl on MacIntoshes | 126 |
| Figure 6.10 | Prime_act.cl Comparison on RS/6000 | 128 |
| Figure 6.11 | Segments per Worker versus Wall Time | 130 |
| Figure 6.12 | Yale Cmatrix on the MacIntosh | 132 |
| Figure 6.13 | VPI(i) Cmatrix on the MacIntosh | 133 |
| Figure 6.14 | VPI(ii) Cmatrix on MacIntosh | 134 |
| Figure 6.15 | Yale Cmatrix on RS/6000 | 136 |
| Figure 6.16 | Yale Cmatrix on RS/6000 | 137 |
| Figure 6.17 | VPI Cmatrix.cl on the RS/6000 | 138 |

List of Illustrations
List of Tables

Table 2.1  Linda Operations  .................................................. 8
Table 2.2  Tuples and Related Operations.  ................................. 10
Table 2.3  Field Usage in TS Partitions  .................................... 23
Table 2.4  Comparison of Operational Complexity.  ...................... 26
Table 3.1  Run Time Library Functions Used to Support the Linda Application.  .... 34
"You can tell how far we have to go, when FORTRAN is the language of supercomputers."

- Steven Feiner

1.0 Introduction

1.1 Background and Motivation

This thesis researches parallel computation using the multi-processor environment of networked workstations. Parallelism and networking are normally considered distinct disciplines, and as such have largely progressed independently. For example, networks address the need to share system-wide resources, such as peripheral devices and file systems. However, multi-processor based computers bring to bear the raw computational power of several CPUs to execute a program in parallel. The former is generally used to interconnect a community of users, that is, to focus on the needs of the many. The latter often used is to address a specific application in a timely manner and focuses on a narrow domain of problems.

The programming language Linda is designed to provide simple constructs and techniques that aid the programmer in the formidable task of writing parallel programs. Although originally designed for multi-processor, shared memory architectures, Linda is execution platform orthogonal. However, unlike some other parallel languages, Linda is not restricted to a particular type of hardware such as an array processor or multi-processor based system. Because parallelism in Linda is task (process) based, it is considered course-grained.
Although networked workstations are becoming commonplace, frequently these workstations are used by one user at a time executing an interactive program such as an editor or a programming utility. One recent study states that fully one third of the networked workstations surveyed remain completely idle even during the busiest times of the day [THEM89]. The aggregate computational power of idle networked workstations is completely unused. Linda has the potential to tap these unused resources by applying the Linda paradigm to a multi-processor environment of networked workstations.

1.2 Problem Statement and Proposed Solution

It is the objective of the Linda Research Group, VPI Computer Science Department, to implement the Linda paradigm in the parallel environment of inexpensive networked Unix workstations. That is, to provide parallel, distributed processing of a Linda program executing concurrently on multiple hosts interconnected via a departmental network. Using this approach a scalable parallel computer can be constructed from existing equipment without additional investment in hardware.

To achieve this goal, a copy of the Linda compiler and run time library source code was secured from Yale University. The Yale implementation only supports parallelism on shared memory multi-computers. From this baseline, a two-phase plan for implementing a LAN based distributed Linda environment was organized. The first phase addresses the design of a distributable run time kernel, application-to-kernel communications subsystem and application's run time library. The second phase uses the kernel and the communications subsystem from phase one and adds a topology for application distribution. This thesis represents the design, implementation and testing
of the first phase of the overall project, that is, the design and implementation of a distributable Linda kernel.

1.3 Plan of Thesis

An understanding of the nature of the Linda language is critical to understanding the support structures of the kernel and run time libraries. Chapter 2 presents the background needed to understand the Linda paradigm and the generic kernel services needed by the Linda application as it relates to this thesis.

Central to the distribution of Linda are the kernel and application interdependencies. Chapter 3 presents design and implementation issues of the Yale and VPI kernels, as well as a comparison of the relative merits and demerits of the VPI approach.

A benchmark comparison between the Yale and VPI implementations is given in Chapter 4. Test programs using the Linda paradigm in various ways are executed in the two environments to demonstrate correctness and relative performance of the VPI implementations. Most of the test programs presented in Chapter 4 demonstrate the Yale implementation to be far superior with respect to wall times. Although, CPU metrics for the two implementations are nearly equal.

Isolation and correction of this deficiency (poor run time performance) is the prime investigation of Chapter 5. A specialized test program is used to explore various aspects of the VPI kernel-program communications bottleneck. Resolution of the problem is demonstrated on a single
Linda program which is designed to measure kernel-application communications.

In Chapter 6 the test suite presented in Chapter 4 is expanded and re-evaluated after making the corrections indicated in Chapter 5. The results from these applications demonstrate the VPI implementation to be comparable to the Yale implementation in terms of execution time and superior in terms of memory usage and extensibility.

In addition to summarizing the contributions of this thesis, Chapter 7 lists several aspects of the VPI implementation which could be studied and further refined.
"Any sufficiently advanced technology is indistinguishable from magic."
- Arthur C. Clark

2.0 Linda Background

This chapter presents background material about the Linda model. With this framework the reader
is provided with key information that is needed in Chapters 3 and 4. The objective of this chapter
is to address the following concerns:

• How is the Linda model used by the application programmer?
• How is the application source processed by the Linda compiler?
• What are the run time support mechanisms needed by the Linda application?

2.1 The Linda Paradigm

2.1.1 Tuple Space

The concept of tuple space, TS, is central to the Linda model [CARN87]. All parallel systems must
have some medium of communication and coordination between concurrent threads of execution
and Linda achieves this utilizing TS. Conceptually, TS is a repository for two types of objects: live
and static tuples. A tuple is defined as an ordered set of data fields or elements, similar to its common usage in database systems. Tuple space is, therefore, a collection of live and static tuples of various content, as shown in Figure 2.1.

Access to TS is shared among multiple concurrent processes that make up a single Linda application. The ability to share TS among concurrent tasks provides the programmer with an interprocess communications medium that is conceptually simple yet powerful.

2.1.2 The Linda Programmer's View of Tuple Space

From the programmer's perspective TS is associative, globally shared memory [GELD85a]. TS is associative in that the individual tuple is referenced by its content. Similar to the query-by-example paradigm in database technology, the Linda program accesses a tuple by describing its contents rather than its address. Tuple space is globally shared memory in that it is accessible to all processes in the Linda program.

Tuple space is manipulated by the programmer in one of three ways: process creation, tuple deposit and tuple retrieval. The following six Linda operations are available for TS manipulation: eval, out, in, inp, rd, and rdp. Table 2.1 lists these operations and their semantics.

The Linda model is integrated into a programming language by adding tuple space and the tuple operators to the base language. Although this thesis will limit discussion to Linda-C, the hybrid created when adding Linda operations to "C", many other languages have been "Linda-fied"
Figure 2.1  
Live and Static Tuples Stored in Tuple Space.

Linda Background
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(a);</td>
<td>The tuple <code>&lt;a&gt;</code> is formed then placed in tuple space available for subsequent in() or rd() operations.</td>
</tr>
<tr>
<td>eval(a);</td>
<td>Places the tuple <code>&lt;a&gt;</code> in tuple space to be computed synchronously.</td>
</tr>
<tr>
<td>in(b);</td>
<td>Searches tuple space for a tuple that matches template <code>&lt;b&gt;</code>. When found the tuple is removed from tuple space and associated with the fields in template <code>&lt;b&gt;</code>. The process blocks until a matching tuple is found.</td>
</tr>
<tr>
<td>rd(b);</td>
<td>Similar to in(b) with the exception that the tuple is not removed from tuple space.</td>
</tr>
<tr>
<td>inp(b);</td>
<td>The predicate form of in(b) except if a matching tuple is not available the operation returns 0 and the fields in template <code>&lt;b&gt;</code> are undefined.</td>
</tr>
<tr>
<td>rdp(b);</td>
<td>The predicate form of rd(b).</td>
</tr>
</tbody>
</table>
[GELD90]. Each Linda operation is, by definition, atomic. These operations offer the only means available to the programmer of adding to, viewing or removing tuples from tuple space. Although syntactically these operators are in the form of "C" function calls, they are, in Linda parlance, operations in the language Linda-C. As discussed in detail later, each field is either a value (actual) or a place holder (formal) parameter [CARN87].

2.1.3 Static Tuples

A static tuple is an ordered collection of actual or formal data elements. Each element is either a scalar (of type character, integer, etc) or an aggregate (array, structure, etc) in the base language. The order and type of each element in the tuple is fundamental in the definition of the tuple's structure and to the organization of TS at run time. Tuples having elements which are identical in order and type are said to be in the same tuple set. Table 2.2 gives examples of tuples belonging to several tuple sets and operations which might manipulate tuples in the set. The programmer designs the program so that tuple generating operations [out() and eval()] have corresponding retrieval operations [rd(), rdp(), in(), and inp()] in the same tuple set.

2.1.4 Live Tuples

A live tuple is similar to a static tuple with the exception that at least one of the data elements of the tuple is yet to be computed. The tuple is said to be "alive" in that the process responsible for computing one of the elements has not terminated. The live tuple provides the mechanism for
Table 2.2  Tuples and Related Operations.

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Generative Operations</th>
<th>Retrieval Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&quot;pi&quot;, 3.1415&gt;</td>
<td>out(&quot;pi&quot;, 3.1415);</td>
<td>in(&quot;pi&quot;, ?x);</td>
</tr>
<tr>
<td></td>
<td>eval(&quot;pi&quot;, lookup(&quot;pi&quot;));</td>
<td>rd(&quot;pi&quot;, 3.1415);</td>
</tr>
<tr>
<td>&lt;&quot;row&quot;, 0, {1,1,2,2,1}&gt;</td>
<td>out(&quot;row&quot;, index, row:5);</td>
<td>rd(&quot;row&quot;, 0, data:5);</td>
</tr>
<tr>
<td></td>
<td>in(&quot;row&quot;, ?index, data:5);</td>
<td></td>
</tr>
<tr>
<td>&lt;&quot;NBC&quot;, &quot;Evening News&quot;, &quot;Ted Koppel&quot;&gt;</td>
<td>out(network, show, star);</td>
<td>in(&quot;NBC&quot;, ?listing, star);</td>
</tr>
<tr>
<td>&lt;&quot;PBS&quot;, &quot;Mr. Rogers&quot;, &quot;Trolley&quot;&gt;</td>
<td>eval(&quot;PBS&quot;, &quot;Mr. Rogers&quot;, findstar(&quot;Mr. Rogers&quot;));</td>
<td>in(?channel, ?show, &quot;Trolley&quot;);</td>
</tr>
<tr>
<td>&lt;&quot;PBS&quot;, &quot;Sesame Street!&quot;, &quot;Big Bird&quot;&gt;</td>
<td>out(&quot;PBS&quot;, bestshow, beststar(bestshow));</td>
<td>in(&quot;PBS&quot;, ?favoriteshow, ?favoritecharacter);</td>
</tr>
<tr>
<td>&lt;&quot;Buy low, Sell high&quot;&gt;</td>
<td>eval(fortune());</td>
<td>in(?brokers_advice);</td>
</tr>
<tr>
<td></td>
<td>out(useless_advice());</td>
<td>rd(?an_encouragement);</td>
</tr>
</tbody>
</table>
parallelism by affording the programmer a means of easily defining tasks which can be computed independently of the parent process. Once a live tuple is deposited into TS it is no longer coupled to the depositing process's memory or process state. Consider the following two Linda operations:

- eval("pi", lookup("pi"));
- out("pi", lookup("pi"));

Although the same data tuple (<"pi", 3.1415>) will be deposited in TS, the eval() operation immediately deposits a live tuple, <"pi", lookup("pi")>, and the eval()ing process continues on with its thread of execution. The live tuple has as its second field, lookup("pi"), a function defined in the application. When the function completes the live tuple is replaced with a data tuple <"pi", 3.1415>.

The out() operation computes the value of lookup("pi") before proceeding to deposit the data tuple. The substantive difference is that the out() operation must serially compute the results of lookup("pi") before depositing the tuple in TS. The eval() operation delegates the computation of lookup("pi") to another concurrent process which will be responsible for depositing the static tuple in TS. Since the process performing the eval() operation executes concurrently with the eval()ed function, this operator provides the programmer with a expressive means for creating parallel programs.

Finally, when, where and how live tuples become static tuples is not the concern of the Linda programmer, thus eliminating platform related (hardware and operating system) issues of portability, synchronization and efficiency. This time and space independence of concurrency in the Linda model is one of its unique properties and the key feature exploited by this research.

Linda Background
2.1.5 Retrieval of Tuples From Tuple Space

The contents of TS are available to all processes in the Linda program by use of the retrieval operators rd(), rdp(), in() or inp(). (The "in()" operator will be used in discussion of these operators in general. When a distinction needs to be made it will be clearly indicated by the accompanying text.) The parameters used in these operators define the attributes of the desired tuple. The parameters make up a "template" which specifies which fields must match in value, which can be overlooked, and which must be read into the variable named in the argument list. Tuples are identified with single angle brackets, as in <"pi", 3.1475>, and templates are annotated with double angle brackets, for example <<"pi", ?x>>.

2.1.6 Template and Tuple Matching

Paraphrasing Carriero [CARN87], there are four rules for tuple selection or matching during an in() operation:

1. A tuple and template must have the same number of fields.
2. Corresponding fields of the tuple and template must agree in type.
3. Corresponding data items (actuals) must be equal.
4. There must be no corresponding formals.

Examples of tuples and retrieval operations which do not match will help clarify the application of these rules:

• in("foo") does not match <"foo", "foo"> because of rule 1.
• in(1) does not match <1.0> because of rule 2.
• in("foo") does not match <"bar"> because of rule 3.
• in(?i) will not match <?i> because of rule 4. Although i and j agree in type, the
tuple and template have corresponding formal fields.

However, examples of matching tuple/template pairs are:

• in(?i) matches <13>.
• in("bar") matches <"bar">.
• in(?ratio) matches <1.0> assuming the variable "ratio" is of type float.
• in("foo", "foo") matches <"foo", "foo">.

As indicated by Table 2.1, the operations rd() and in() are blocking, that is, the process performing
the operation pauses until the TS search completes successfully. This provides a mechanism for
synchronization because a blocked process must wait for another process to either out() or
complete an eval() of a matching tuple. The operations inp() and rdp() provide a means for the
programmer to avoid blocking in the case of a match failure and to take alternative action.

If more than one matching tuple is available the operation is non-deterministic with respect to which
tuple is selected. After a match is found the formal fields are appropriately filled, the tuple is
returned to the inp()ing process, and the tuple is deleted from TS in the case of in() and inp(). If a
group of in()s and rd()s are blocked when a process out()s a matching tuple, there are no
guarantees of fairness; zero or more rd()s may be satisfied and only one in() will unblock.

The remainder of this chapter addresses the Linda-C compiler and run time support system
(kernel). Although the Linda compiler and kernel will be discussed separately there exist a high
degree of coupling between them. That is, the compiler makes extensive use of anticipated kernel

Linda Background
run time support and the kernel expects all Linda operations to follow strict conventions enforced by the compiler.

2.2 The Linda Compiler

The Yale Linda-C compiler is a source-to-source compiler: it converts source code from Linda-C into standard C source and supporting data files. The C source is then compiled and linked with a Linda run time library into an executable program by the Unix C compiler. The run time library, in the Yale environment, is the Yale kernel discussed in Chapter 3. As we will see, the process of collating all Linda operations in an application and converting them into an efficient executable is a complex task. The principle concern is how to search TS efficiently. It is obvious that a "brute force" approach to matching templates to tuples would make Linda unsuitable for all but the most unambitious applications. Therefore, in order to optimize tuple matching, the compiler is designed to exploit the following principles:

- All transactions (out() and in() interactions) must obey the matching rules listed in section 2.1.6.
- Many types of transactions do not require an exhaustive TS search.
- Fields can be eliminated from the tuple/template if they are not needed during run time.

By applying these principles to the template/tuple matching problem, run time searching is reduced considerably. The compiler divides the program's use of TS into disjoint partitions, called tuple sets, and Linda semantics guarantee that searching will only occur within such partitions. The compiler's code analysis ensures that no corresponding out()/in() transaction pair exists across
these boundaries. Each tuple set is uniquely structured for efficient searching by one of four matching algorithms found in the Linda kernel: Counting Semaphore (CS), Hash, Queue, or Private (specialized) Hash Table (PHT). By exploiting the disjoint partitions of TS, the compiler, with foreknowledge of the kernel’s various search algorithms, selects the most suitable method for each partition and generates code accordingly. The partitioning of TS into tuple sets is hidden from the programmer.

Without the optimizations made by the Linda compiler it is doubtful that the Linda environment would be much more than an academic curiosity. This section will discuss the implications of the matching rules and how they are exploited to increase performance. Following the compiler discussion, the generic kernel support expected by the compiler will be addressed.

### 2.2.1 Corresponding Field Count and Type

Suppose that during analysis of a Linda-C program the compiler found these Linda operations:

```plaintext
out("sync");
in("foo", ?a);
in(?str, a, ?b);
out(BAR, x);
in("sync");
in("foo", 3);
in("bar");
in(BAR, y);
in("foo", x);
in(?string, 2);
rd("bar");
out("foo", i);
out("foo", 3.0);
out(string, ?x, y);
out("bar");
```

By applying rules 1 (tuple and template must have the same number of fields) and 2 (corresponding tuple and template fields must agree in type) from Section 2.1.6 we see that these operations
require five distinct partitions in TS:

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
<th>Partition 4</th>
<th>Partition 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(&quot;sync&quot;)</td>
<td>in((?str, a, ?b);</td>
<td>out(BAR, x);</td>
<td>in(&quot;foo&quot;, ?a);</td>
<td>out(&quot;foo&quot;, 3.0);</td>
</tr>
<tr>
<td>in(&quot;sync&quot;)</td>
<td>out(string, ?x, y);</td>
<td>in(BAR, y);</td>
<td>out(&quot;foo&quot;, 3);</td>
<td></td>
</tr>
<tr>
<td>rd(&quot;bar&quot;);</td>
<td></td>
<td></td>
<td>in(&quot;foo&quot;, x);</td>
<td></td>
</tr>
<tr>
<td>out(&quot;bar&quot;);</td>
<td></td>
<td></td>
<td>in(?string, 2);</td>
<td></td>
</tr>
</tbody>
</table>

If the analysis of TS usage were to be considered complete at this stage, the resulting partitions would represent self contained tuple sets. The kernel would not (and does not) have to provide searching support across partitions. These five partitions will be discussed individually as illustrative of how the compiler and kernel cooperate in reducing run time matching complexity.

### 2.2.2 Data Items Must Match

Consider Partition 1 from Section 2.2.1:

<table>
<thead>
<tr>
<th>Partition 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(&quot;sync&quot;);</td>
</tr>
<tr>
<td>in(&quot;sync&quot;);</td>
</tr>
<tr>
<td>in(&quot;bar&quot;);</td>
</tr>
<tr>
<td>rd(&quot;bar&quot;);</td>
</tr>
<tr>
<td>out(&quot;bar&quot;);</td>
</tr>
</tbody>
</table>

It is immediately obvious that this "partition" consists of two disjoint parts of TS. The simple application of the field type and count rules places these tuples within the same partition when,
upon further investigation, a more exact division of the operations can be made. Since corresponding data items (actuals) must match, the compiler checks for matching constant fields. If a partition contains only constants in a particular field, the partition can be subdivided into smaller disjoint tuple sets, one set for each unique constant. Thus, Partition 1 is reduced to Partitions 1a and 1b:

<table>
<thead>
<tr>
<th>Partition 1a</th>
<th>Partition 1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(&quot;sync&quot;);</td>
<td>in(&quot;bar&quot;);</td>
</tr>
<tr>
<td>in(&quot;sync&quot;);</td>
<td>rd(&quot;bar&quot;);</td>
</tr>
<tr>
<td></td>
<td>out(&quot;bar&quot;);</td>
</tr>
</tbody>
</table>

2.2.3 Field Removal

The Partitions 1a and 1b contain constant fields and do not require run time matching support since the operations within either partition always match. A more significant conclusion can be drawn: field data transfer to and from TS is not required; tuple storage is likewise unnecessary. In Partitions like 1a and 1b, the Linda compiler removes the field from the Linda operations. Removing fields when ever possible increases run time efficiency. There must, however, be some accounting for the number of out()s that have been made in a partition like this. Each in() operation must, likewise, decrement the number of tuples in its partition, effectively removing the tuple from TS. If an in() is attempted before any out()s have been executed, then the language definition requires that the process block until a matching tuple is out()ed. This is accomplished in the kernel.

Linda Background
by using a "counting semaphore" (CS) algorithm. Any tuple set which does not require data to be transferred at run time is controlled by the CS run time support code.

Field removal is not limited to tuple sets controlled by the CS algorithm. Any field within a tuple set which is compile-time constant is a candidate for removal. For example, the value BAR in Partition 3 above is constant. The template for in(BAR, y) is << BAR, y>>. The tuple for out(BAR, x) is <BAR, x>. All tuples in this partition have as a first field the constant, actual parameter BAR. This field is a candidate for removal since it will not contribute meaningfully at execution time.

2.2.4 Odd Field Polarity

Each field in a tuple or a template is either formal [e.g. in(?x)] or actual [e.g. in(x)]. As stated earlier, for corresponding fields to match either they contain matching data (actual fields) or one field is formal and the other actual. Corresponding formal fields, by definition, fail to match [e.g. in(?x) and out(?x)]. The term field polarity is used to refer to the formal/actual relationship between corresponding fields.

```
Partition 2
in(?str, a, ?b);
out(string, ?x, y);
```

Partition 2, listed above, contains operations in which all corresponding fields have odd polarity, that is, for each field in the generative operations, the corresponding fields in the retrieval operations
must be of opposite polarity. In this partition there are no actual fields in the generative operations which have corresponding actual fields in a retrieval operation. The run time support for tuple sets of this type does not require field matching. But, unlike CS tuple sets, data must be moved from TS to the retrieving process. The kernel provides a queue algorithm for tuple sets exemplified by Partition 2. Initially the queue is empty. Tuples are placed in the queue by out() operations and removed by in()s. If more than one tuple is in the queue, the tuple retrieved by the next in() operation is non-deterministic (implementation dependent) as per the Linda paradigm [GELD85a].

2.2.5 Exhaustive Searching

Partition 3

\begin{verbatim}
out(BAR, x);
in(BAR, x);
\end{verbatim}

The third type of tuple set requires run time matching for at least one field in the tuple. Partition 3 illustrates TS transactions of this nature. Each in() operation requests a specific tuple by value, although no data is required to be transferred. To improve matching efficiency the kernel uses a hash table as the primary storage structure for tuples of this type. As tuples are deposited, the kernel calculates a hash value based on the value of the tuple's hash field. The hash field is selected by the compiler and is fixed for all operations on the partition. The hash value is used as an index into a hash table where tuples are stored in hash bins. Tuples which have equal hash values share the same hash bin, although this does not guarantee the hash fields are equal. Using this mechanism, it can be quickly determined if a matching tuple does not exist because the hash

Linda Background
bin is empty. However, when the bin is not empty, all tuples in the bin must be exhaustively searched.

For tuples which have only a single field to be matched, like those in Partition 3, the hash table algorithm provides a meaningful optimization. If the tuple contains other fields which must match, for example the operations out(x, y) and in(a, b), the compiler selects one of the fields as the hash field. Once a hash value has been calculated for an in() operation, both fields are matched using brute force, that is, each tuple in the hash bin is tested as a match by comparing all actual fields in the template with the actual fields in each tuple until a match is found.

2.2.6 Other Tuples with Actual Fields

Partition 4

\[
\begin{align*}
&\text{in("foo", ?a);} \\
&\text{out("foo", 3);} \\
&\text{in("foo", x);} \\
&\text{in(?string, 2);} \\
&\text{out("foo", i);} \\
\end{align*}
\]

In Partition 4, all operations deal with tuples containing two fields: the first is a NULL terminated string; the second is an integer. The tuples (<"foo", 3> and <"foo", i>) contain actuals in both fields.

The templates contain both actuals and formals in each corresponding field:

\[
\begin{align*}
&\langle\langle\text{foo", ?a}\rangle> \\
&\langle\langle\text{foo", x}\rangle> \\
&\langle\langle\text{?string, 2}\rangle>
\end{align*}
\]

Linda Background
Matching is required for two of the `in()` operations:

- `in("foo", x)` must find a tuple whose second field matches the value of `x`.
- `in(?string, 2)` must find a tuple whose second field matches the value `2`.

The operation "in("foo", ?a)" will accept any tuple whose first field is "foo" and whose second field is an integer. Since all the `out()` operations in Partition 4 meet this requirement at compile time, the operation "in("foo", ?a)" does not require matching support. However, this operation is the only one in the set which requires data to be transferred from TS to the process. The other two `in()`s require run time matching.

The first field could be easily supported by the aforementioned queue algorithm. Interestingly, the second field requires more support than is available in the three algorithms (CS, queue, hash) already discussed. It is similar to the queue, which has an actual/formal field polarity, and to the hash, which has an actual/actual field correspondence. Using a hash table would support all operations except the "in("foo", ?a)". This operation presents a problem for the hash implementation since there is no actual value in the template `<"foo", ?a>>` to use for the hash calculation. The compiler addresses this by assigning tuple sets of this type to a "hybrid" hash algorithm, referred to as a private hash table (PHT). The tuples are stored in hash tables as in the hash algorithm, although the searching of the hash table follows a modification of the standard hash model. When the hash field polarity is actual/actual the kernel limits searching to only a single hash bin. When the field polarity is actual/formal the kernel must exhaustively search the tuple set. Every tuple in the set matches with respect to this field. If no other matching requirements exist, searching stops with the first tuple found, the first non-empty hash bin. If, on the other hand, additional matching is required the "brute force" approach is employed.

Linda Background
To summarize this partition: the first field requires data transfer but no run time matching, the second field needs run time data transfer and exhaustive matching.

In general, when more than one field is involved, the field requiring the more arduous matching defines the algorithm used. This field is known as the key field. All other actual/actual fields are matched by a linear search after the key field is satisfied. The order of strength (and the larger run time cost) in matching algorithms is then:

- counting semaphore (CS)
- queue
- hash
- private hash table

A description of the algorithms is found later in this chapter where the kernel's run time requirements are presented. These ideas are summarized in Table 2.3

### 2.2.7 The Problem With Partition 5

What if during source analysis the compiler found only one operation, "out("foo", 3.0)", with a tuple/template of type <string, float>? This indicates a possible oversight by the programmer and is flagged as a warning. A similar warning will be generated if the operation "eval("foo", (float)gpa())" is found without a template of the form <<?string, ?float>>, <<"foo", ?float>>, <<?string, float>> or <<"foo", float>>.
<table>
<thead>
<tr>
<th>Tuple</th>
<th>Template</th>
<th>Example</th>
<th>Matching Requirements</th>
<th>Kernel Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>Actual</td>
<td><code>out(x)</code>&lt;br&gt;<code>in(x)</code>&lt;br&gt;<code>- - or - -&lt;br&gt;</code>out(&quot;foo&quot;)<code>&lt;br&gt;</code>in(&quot;foo&quot;)`</td>
<td>Values of tuple and template must match but no data is transferred (hash). Special optimization for matching constants (CS).</td>
<td>Hash, CS</td>
</tr>
<tr>
<td>Actual</td>
<td>Formal</td>
<td><code>out(?)</code>&lt;br&gt;<code>in(?)</code></td>
<td>No matching needed, actuals are copied from tuple to template.</td>
<td>Queue</td>
</tr>
<tr>
<td>Actual</td>
<td>Both</td>
<td><code>out(x)</code>&lt;br&gt;<code>in(x)</code>&lt;br&gt;<code>in(1)</code></td>
<td>Templates using actuals must match, no data is transferred. Those using formals require an exhaustive search throughout the tuple set.</td>
<td>Private Hash Table</td>
</tr>
<tr>
<td>Formal</td>
<td>Actual</td>
<td><code>out(?x)</code>&lt;br&gt;<code>in(1)</code></td>
<td>No matching or copying needed since the out's formal field acts as a wildcard.</td>
<td>CS</td>
</tr>
<tr>
<td>Formal</td>
<td>Formal</td>
<td><code>out(?x)</code>&lt;br&gt;<code>in(?x)</code></td>
<td>Operations belong to disjoint partitions.</td>
<td>illegal</td>
</tr>
<tr>
<td>Formal</td>
<td>Both</td>
<td><code>out(?x)</code>&lt;br&gt;<code>in(1)</code>&lt;br&gt;<code>in(?x)</code></td>
<td>In() operations using formals are not part of partition.</td>
<td>illegal</td>
</tr>
<tr>
<td>Both</td>
<td>Actual</td>
<td><code>out(?x)</code>&lt;br&gt;<code>out(x)</code>&lt;br&gt;<code>in(x)</code></td>
<td>Out() operations with formal parameters act as wildcards. Out()s using actuals must match in value to the in's template.</td>
<td>Hash, CS</td>
</tr>
<tr>
<td>Both</td>
<td>Formal</td>
<td><code>out(?x)</code>&lt;br&gt;<code>out(x)</code>&lt;br&gt;<code>in(?x)</code></td>
<td>Out() operations using formal fields are not part of partition.</td>
<td>illegal</td>
</tr>
<tr>
<td>Both</td>
<td>Both</td>
<td><code>out(?x)</code>&lt;br&gt;<code>out(x)</code>&lt;br&gt;<code>in(x)</code>&lt;br&gt;<code>in(?x)</code></td>
<td>Not able to restrict intended TS transactions at compile-time. Run time support must include exhaustive searching.</td>
<td>Private Hash Table</td>
</tr>
</tbody>
</table>
2.3 The Linda Kernel

The Linda compiler generates function calls to a pre-defined run time support system referred to as the kernel. It is the kernel's duty to maintain and manipulate TS for the Linda application. Although the compiler is unconcerned with how the kernel performs its mission, certain services must exist in order to support the code generated by the compiler. The previous section described the methods and strategies used in analyzing the Linda program to produce efficient TS storage and retrieval. This section will provide the framework needed to understand the kernel implementations presented in Chapter 3 by discussing the minimal run time support required by the Linda compiler.

2.3.1 Tuple Storage and Retrieval

The compiler partitions all compatible Linda operations into disjoint tuple sets and assigns one of four matching algorithms to each set. Although all tuple sets can be successfully handled by using brute force matching, a significant reduction in the searching time is obtained by selecting the most appropriate method for each set. How each algorithm stores and retrieves (matches) tuples at run time significantly impacts efficiency. Each algorithm must be capable of the following primitives:

- *Deposit*. When an out() or eval() function completes the kernel will be called upon to store the tuple in TS for later operations.
- *Match*. As discussed above, matching requirements vary considerably between tuple sets. For some sets matching equates to checking for the existence of any tuple, other sets require more advanced searching methods.
• **Copy.** Once a tuple/template match is located, data is typically copied into the formal fields of the template.

• **Remove.** The in() and inp() operations require that the tuple values be copied to the template and the corresponding tuple be removed from TS after a match is found. The tuple must not be available for subsequent matching.

Each algorithm, CS, queue, hash, PHT, is named after the storage structure and search method used by the kernel to execute the above primitives. Table 2.4 compares the complexity of each primitive as required by the compiler's choice of algorithm. A more detailed discussion of searching complexity follows.

### 2.3.1.1 Counting Semaphore

The counting semaphore, CS, algorithm requires a run time tally for each tuple in a CS set. The tally is incremented when a tuple is deposited, and decremented when a tuple is removed. There is no matching requirement beyond checking for tuple existence, a non-zero tally. Interestingly, a "retrieval" operation does not require that data be transferred from TS to the process performing the in(). As discussed earlier, due to the fact that all tuple and template fields in the set are actuals, the compiler, having pre-matched all the operations, has removed the fields from run time consideration. For these reasons this algorithm is by far the most efficient among the four.
<table>
<thead>
<tr>
<th></th>
<th>Deposit</th>
<th>Match</th>
<th>Copy</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>trivial</td>
<td>trivial</td>
<td>N/A</td>
<td>trivial</td>
</tr>
<tr>
<td>Queue</td>
<td>simple</td>
<td>simple</td>
<td>simple</td>
<td>simple</td>
</tr>
<tr>
<td>Hash</td>
<td>simple</td>
<td>moderate</td>
<td>moderate</td>
<td>simple</td>
</tr>
<tr>
<td>Private Hash Table</td>
<td>simple</td>
<td>most exhaustive</td>
<td>moderate</td>
<td>simple</td>
</tr>
</tbody>
</table>
2.3.1.2 Queue

The queue method is nearly as efficient as the CS in primitive operations except that it does require run time data transfer from TS to the incoming process. Instead of a tally, the set storage structure is a queue, which requires slightly more deposit and retrieval overhead. Like the CS, each queue set has been pre-matched by the compiler so that matching is reduced to checking for an empty queue. The name queue implies a FIFO priority of storage and retrieval which is not required by the language specification. Actual implementations (Yale and VPI) use a FILO stack for simplicity. The transfer of the tuple values to the template involves a simple copy operation since all fields are guaranteed to be opposite in actual/formal polarity.

2.3.1.3 Hash

Hash tuple sets require run time matching of field values because pre-matching is not possible during compilation. These sets are organized in hash tables. The compiler designates one field as the hash field, which is used as the input to a hash function generating a hash value. Each hash partition has an associated hash table wherein all the tuples of the set will be stored. The calculation of a hash value directs the kernel to a subset of the tuple set where tuples with equivalent hash values are located. Tuple deposit is fairly straightforward since table address requires only a simple hash value calculation.

Matching, however, is more complicated than with the queue method because fields with equal hash values do not necessarily have matching hash fields. However, searching is still reduced

Linda Background
considerably since conflicting hash values assure a mismatch of the hash field. Once a hash bin has been selected each field must be subjected to the polarity and value matching rules.

After a matching tuple is found a more complicated copying operation is required. Since field polarity has not been pre-matched the kernel must check field polarity to determine if copying is required. Only formal template fields require that data be copied from the tuple to the template. Although the hash algorithm is the most rigorous discussed thus far, in a worst case analysis it is much more efficient than the private hash table.

2.3.1.4 Private Hash Table

Like the hash method, private hash table (PHT) storage structures employ a run time hash table. However, unlike the hash algorithm, there exists at least one operation in the Linda application that contains a formal parameter in the hash field. When processing templates with actual hash fields, the algorithm can follow the same strategy as the hash method. Complications are introduced when the kernel receives a template having as its hash field a formal parameter. Since the formal parameter could match any actual value, all hash bins must be searched exhaustively, in effect reducing searching efficiency to brute force. In this case, the search space is restricted to the partition but no further.
2.3.2 Tuple Space Initialization

Since the kernel is a static entity unaffected by the compilation process, the number and type of tuple sets needed during program execution must be communicated at run time. For example, the number of counting semaphores and queues needed differs from one application to the next. Each hash and PHT tuple set requires its own hash table. The Yale Linda compiler generates an initialization call to the kernel which is invoked before control is passed to the application. This initialization procedure specifies the number of tuple sets required by each algorithm and the maximum amount of memory that the application will require.

2.3.3 Blocking and Predicate Support

The blocking operations in() and rd() place requirements on how the retrieval process is implemented. When the search for a tuple fails the kernel must continually attempt to locate a matching tuple as tuples are added to TS. The net effect is that the kernel must maintain a record of outstanding templates for each tuple set. Every generative operation must check for pending retrieval operations on the set and attempt a match.

The non-blocking operations inp() and rdp() must not block. If the TS search fails, the operation returns 0, success returns non-zero. The kernel must not add these operations to those waiting to be matched.
2.3.4 Eval() Support

There is significant controversy about the semantics of the eval() operation particularly in the area of variable bindings in the eval()ed function [CARN87]. At a minimum, the kernel must be able to create a separate process that will compute the eval()ed field(s) and place the resulting tuple in TS. The tuple set that an eval()ed tuple belongs to is defined by the type, number and polarity of its fields. Eval()ed fields are typed by the return type of the eval()ed function. For example, the first field of the operation "eval(worker())" is defined by the value that is returned by the function "worker()". Polarity for this field is actual since a value will be returned by the eval()ed function. The kernel does not need separate deposit, matching, copying or removal support since these are available as primitives.

The implementation of eval() defines the type of parallelism that the run time system will support. This is an important kernel function and is the topic of considerable research during the second phase of this investigation - LAN-based distributed Linda.
A great many people think they are thinking when they are merely rearranging their prejudices

- William James

3.0 Two Kernel Designs

Implementation of the Linda model of computation has profound impact on run time efficiency, and thus the usefulness of the programming environment. Chapter 2 described the Linda programming environment and the role of the Linda kernel in that environment. This chapter discusses the design of the original (Yale) kernel and the distributed (VPI) kernel. These two kernels provide equivalent functionality, but differ considerably in design and implementation. The Yale environment relies on physically shared memory for tuple space storage. The VPI environment provides tuple space storage as private memory in a kernel process, the end goal being distribution of the environment across a LAN. The chapter concludes with an appraisal of the VPI design and discusses some of the run time costs expected in the VPI design. Chapter 4 presents a detailed comparison of application run time performance in the two environments.

3.1 Yale Kernel Design

In the spirit of the Linda paradigm, the programmer (and the Linda program) is presented with a very high-level view of Tuple Space and concurrency [CARN89a]. The only requirement is that the programmer be familiar with the Linda model, the use of the five Linda operations and the idea of Two Kernel Designs
tuple space. In the Yale Linda environment the compiler and run time support system are designed
to provide a complete abstraction of the Linda model so that no implementation specific knowledge
is needed to make effective use of the environment. The Yale implementation of tuple space and
the Linda operators are built on three primary design features:

- All kernel functions are located in a C function library and are added to the Linda
  program at link time.
- Unix shared memory is used to implement Linda's globally shared memory, Tuple
  Space.
- The Linda eval() operation is implemented as a process fork.

These features, illustrated in Figure 3.1, prevent the Yale environment from supporting distributed
parallelism. The bulk of this research addresses how the VPI kernel overcomes the limitations of
the first two design aspects leading to distribution of the Linda application and tuple space.

3.1.1 The Linkable Kernel

In the Yale system all kernel functions are implemented as routines within the process' code space.
The Linda program, then, is composed of the Linda application, written by the programmer, and a
body of support code added at link time from the kernel library. All management of TS and
transmission of tuple/template data to and from the Linda process is the responsibility of the kernel
code. These kernel routines, listed in Table 3.1, comprise the library that is referenced at link time.
The compiler, with knowledge of the kernel library routines, sets up the data structures used in
process/kernel communications and generates calls to the necessary kernel routines. The library

Two Kernel Designs
Figure 3.1  Yale implementation of Tuple Space in Shared Memory.

Two Kernel Designs
Table 3.1  Run Time Library Functions Used to Support the Linda Application.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_hash()</td>
<td>Process rd() or in() by a Linda process. Each partition has a dedicated retrieval function.</td>
</tr>
<tr>
<td>in_hybrid()</td>
<td></td>
</tr>
<tr>
<td>in_cs()</td>
<td></td>
</tr>
<tr>
<td>in_queue()</td>
<td></td>
</tr>
<tr>
<td>out_hash()</td>
<td>Process out() by a Linda process. Like the retrieval routines, each partition has a dedicated generative routine.</td>
</tr>
<tr>
<td>out_hybrid()</td>
<td></td>
</tr>
<tr>
<td>out_cs()</td>
<td></td>
</tr>
<tr>
<td>out_queue()</td>
<td></td>
</tr>
<tr>
<td>ifork()</td>
<td>Supports the Linda eval() operation.</td>
</tr>
<tr>
<td>exeit()</td>
<td>Supports Linda process termination.</td>
</tr>
<tr>
<td>fork_init()</td>
<td>Registers eval()ed process and initializes run time structures.</td>
</tr>
<tr>
<td>free_tbs()</td>
<td>Releases internal structures after use.</td>
</tr>
<tr>
<td>get_tbs()</td>
<td>Acquires run time tuple storage structure.</td>
</tr>
<tr>
<td>in_hy_exh_search()</td>
<td>Supports private hash table tuple searching.</td>
</tr>
<tr>
<td>match_tuple()</td>
<td>Supports paradigms (hash and hybrid) which require run time matching.</td>
</tr>
<tr>
<td>ptp_copy()</td>
<td>Replicates kernel-to-process tuple structure.</td>
</tr>
<tr>
<td>spinlock()</td>
<td>Supports process wait on a semaphore.</td>
</tr>
<tr>
<td>spinunlock()</td>
<td></td>
</tr>
<tr>
<td>print_times()</td>
<td>Prints all timers and associated information.</td>
</tr>
<tr>
<td>start_timer()</td>
<td>Initializes timer and begins timing.</td>
</tr>
<tr>
<td>timer_split()</td>
<td>Marks current lapse time and posts message.</td>
</tr>
<tr>
<td>main()</td>
<td>Linda program's startup function. This function &quot;real_main()&quot; is the application's entry point.</td>
</tr>
</tbody>
</table>
is designed with the run time flexibility to accommodate any number of disjoint tuple sets in each of the four TS partitions discussed earlier.

Using this approach simplifies the compiler's code generation and maintenance requirements. This simplicity is powerfully demonstrated by the fact that during the course of this research, although the kernel library was completely replaced, no modifications to the compiler were necessary. Any Linda program could be executed in the Yale environment or with the VPI kernel by only changing the run time support library and running the VPI kernel.

### 3.1.2 Shared Memory for Tuple Space

The second major design concept in the Yale kernel is the use of Unix shared memory for tuple storage. This allows simple, rapid and concurrent access to TS by multiple Linda processes. Within each TS partition all processes must pass through the same kernel code and are subject to the same access controls as all other processes. Although the programmer is unaware of it, critical regions and resources are protected by using semaphores and process spin locks. Preserving this high-level perspective of TS is important to the major characteristics of the Linda model:

- associative (content addressable) memory,
- atomic Linda operations, and
- anonymous interprocess communication.

Two Kernel Designs
Although the architectural requirement for shared memory limits portability, it allows exploitation of parallelism by multi-processor systems such as the Intel iPSC/2 and Encore Multimax, as well as others [CARN89a].

Finally, the use of shared memory requires that the Linda application know in advance the amount of memory needed for tuple space. When the program starts, this block of shared memory is requested from the operating system and is fixed throughout program execution. Interestingly, this is not the case with the VPI design because the reliance on shared memory has been removed (c.f. Section 3.2).

### 3.1.3 Process Forks and Eval()ed Functions

The third Yale design concept is to implement the Linda eval() operation as a process fork. Several run time steps are required to achieve this:

- The operating system replicates the process's code and data space and creates a new, concurrent process. The parent process continues executing by returning from the eval() operation.
- The newly created process calls the functions to be eval()ed.
- The return value of the eval()ed functions is placed in the corresponding field in the tuple.
- The tuple is sent to the kernel for placement in TS.
- The forked process terminates.
In the Linda environment this process is understood to be executing as a "live tuple" which, upon completion of its eval()ed functions, becomes a static tuple. While the eval()ed process executes, it may interact with other processes by passing tuples to and from TS or by eval()ing additional processes. The eval()ed process is the basis of parallelism in Linda.

In the Unix environment the fork system call has certain side effects associated with it. How these properties relate to the Linda paradigm is still open to debate since several are in contradiction to the time and space orthogonality of the model. In particular, the forked process inherits many attributes from its parent process which allows non-anonymous and synchronous interprocess communication.

The inherited attributes of interest are:

- the environment,
- attached shared memory segments,
- open files and corresponding file descriptors for each.

Although these methods are available to the programmer, their necessity is minimized by the ease of process generation and communication provided by Linda.

### 3.2 Yale Kernel Implementation Issues

Any design approach has associated implementation details. Certain features of the Yale design require special attention in order to provide the Linda environment with acceptable performance characteristics. This section discusses those implementation issues of the Yale design which relate to this thesis. This information is helpful in understanding the mechanisms that are used in a functional Linda environment. The topics of interest are:

**Two Kernel Designs**
• kernel initialization,
• program termination and cleanup,
• process registration,
• mutual exclusion using semaphores, and
• out() operation mechanics.

**Kernel Initialization**

Initialization of the kernel takes place before the Linda program gains control of the environment. The compiler, based on TS analysis, generates a call to the kernel's initialization routines detailing the resources needed by the Linda program. At execution time, shared memory and semaphores are allocated for each tuple set that the compiler identified. The amount of shared memory is determined by a compiler switch. In the Yale environment the application must be recompiled if shared memory requirements exceed the programmer's original estimate. Process initialization is also necessary for each eval()ed process.

**A Specialized Cleanup Process**

When a Linda program executes, the original thread will typically eval() multiple concurrent processes. The application will solve a given problem utilizing the parallelism provided by these eval()ed processes. According to the Linda paradigm, once a process is eval()ed it is no longer coupled to its parent (time and space independence) [GELD85a]. Consequently, the parent may terminate before any of its siblings. A special purpose process is created (forked) during the startup phase of the Linda application. This process is responsible for maintaining continuity between the initial process and all eval()ed processes; it is also the last process to terminate. After all other

**Two Kernel Designs**
processes in the application have terminated, this cleanup process releases back to the operating system all resources claimed by the Linda environment since startup: semaphores and shared memory.

Eval()ed Process Registration and Termination

In the Yale environment the kernel must perform bookkeeping functions for each process that is eval()ed. The bookkeeping is required upon process creation and termination in order to manage semaphores and shared memory. Each eval() operation invokes a kernel function, lfork, which returns to the parent after forking a child process. Upon returning the parent process continues to execute independently of its newly generated child process. The child process's main purpose is to evaluate the return value for each eval()ed function and place the resulting tuple in TS. In order to perform this, the following steps are performed by the kernel:

- Registration of the child process. This is required by the kernel in order to coordinate concurrent access to shared memory.
- Call all eval()ed functions. The new process computes the value of each eval()ed function and places the result in the corresponding field of the live tuple.
- Send tuple to TS. After calling all eval()ed functions the tuple structure must be completed and deposited in shared memory for access by other processes.
- Unregister and terminate the child process. The kernel reverses the registration process returning Linda resources to the environment. The child process terminates normally with an exit call to the operating system.
Mutual Exclusion with Semaphores

The Linda environment provides concurrent TS access to multiple processes. As discussed in Chapter 2, the compiler partitions all Linda operations and their corresponding tuple types into tuple sets. Each set is inclusive of all tuples that might possibly match and exclusive of all those that are guaranteed not to match. Provided that concurrent processes address separate partitions, this static analysis allows unguarded concurrent access to TS. However, the kernel must coordinate concurrent processes that simultaneously address a single tuple set. Classical semaphores with p and v operations maintain shared memory's integrity. Each tuple set is assigned a semaphore and the kernel code responsible for the set's paradigm (cs, queue, hash, PHT) implements the control strategy. Critical regions are thus assured of mutual exclusion, guaranteeing all Linda operations are atomic.

Mechanics of Coordination During an Out() 

The final implementation issue is related to the dynamic asynchronous nature of parallelism in Linda. At run time, tuple space may not contain matching tuples for in() or rd() operations on a particular tuple set. Consequently, the requesting process is blocked until a matching tuple is deposited by another process executing an out() or an eval() operation. Therefore, every out() must check to see if there are waiting (blocked) in()s or rd()s. Since tuple matching is required in the in() and rd() kernel support code, the Yale implementation alerts the blocked process to the presence of the new tuple rather than include matching logic in the out() handlers. This approach forces the kernel to wake all blocked processes for each out() operation on a particular tuple set. Additionally, this requires that the tuple be deposited in TS before unblocking any waiting processes. Each unblocked process is required to search TS for all newly deposited tuples, test each for a match, and return to the user's

Two Kernel Designs
code if one is found or re-block until the next out(). This complexity is greatly reduced in the VPI design as the natural result of the client/server paradigm.

3.3 VPI Kernel Design

The primary objective of this research is to achieve parallelism by distributing the Linda environment on a multicomputer LAN [SCHC91]. Each participating computer is available to host one or more of the concurrent Linda processes. Using the Yale Linda environment as a starting point the research is divided into two implementation phases. This thesis focuses on the first phase where the objectives are to:

- Remove the kernel's dependency on physically shared memory.
- Remove the run time kernel from the Linda application to allow physical distribution of the Linda program and the kernel. This feature would be exploited heavily in the next phase of the project.
- Provide a portable, multi-platform communications subsystem.
- Provide a working system (or proof of concept) for the second phase of the overall objective. This intermediate system will provide useful information for the design of the second phase of the project.

The second phase of implementation provides for the distribution of concurrent processes. With the ability to have live tuples (eval()ed processes) execute remotely, that is, on a machine other than the machine executing the eval() operation, LAN-based multiprocessor parallelism is achieved. Implementation for the second phase is complete and reports are in preparation.

Two Kernel Designs
This section discusses each of the above mentioned design objectives followed by a presentation of implementation issues related to the VPI kernel's design.

### 3.3.1 Remove Dependency on Shared Memory

Since the Linda environment is conceptually based on a globally shared associative memory (TS), the VPI system must provide some form of common memory. However, being based on workstations distributed across a LAN, the VPI system has no physically shared memory available for this purpose. Physically shared memory is the primary means of Tuple Space implementation in previous Linda systems. The single most important design decision to make during the early design reviews is to use a minor variation on the client/server model. In the VPI environment, the Linda kernel plays the role of server and the processes act as clients. All TS data structures are held in the kernel. Linda operations behave as clients requesting services from the kernel. Based on this approach, the client/server model (and the removal of shared memory) gives rise to several subordinate design features. They are presented graphically in Figure 3.2 and are listed below:

- All tuple matching code is located in the kernel (server). The kernel process is the sole owner of TS. Access is governed by a single-threaded kernel. Therefore, semaphores and spin locks are no longer necessary.

- The Yale kernel is replaced by an I/O interface that communicates with the kernel. This is implemented as a linkable library just as the original kernel.

- BSD sockets are the means of communication between the Linda processes and the kernel.
3.3.2 Separate Kernel Process

The second design objective of the VPI kernel is to define the kernel as a separate process (as opposed to incorporating it into the application). In contrast to the Yale implementation, the VPI environment follows the client/server model of communication. In this model, multiple clients rely on a single server, the kernel, for TS storage and retrieval. The major advantages to this approach are:

- A clear delineation of process and kernel functions. Subsequent research into parallelism relies on this decoupling of kernel and application code space.
- The ability to physically distribute the kernel.
- More efficient process generation. Since the kernel code is not replicated during each eval() operation the fork()ed processes are smaller.

3.3.3 Portable Communications Subsystem

Since the first phase of the overall project provides a distributable kernel, the interprocess communications method chosen must be widely supported. The choice of Unix as the execution environment is predetermined because our target environment consists of networked Unix platforms. Using sockets assures universal support on any of the expected Unix target systems.
Figure 3.2  VPI Kernel Design

Two Kernel Designs
3.3.4 Working Model for Further Research

Several areas of investigation are anticipated before continuing with the second phase of distributing the Linda environment. The VPI kernel is intended to provide a test bed for such research. Some of the issues expected to be pursued are:

- Alternative interprocess communication support. Although the client/server model is being chosen, there is no need to restrict interprocess communications to Unix sockets.

- Separate kernels for each TS partition. By locating TS in a single kernel process concurrent Linda operations on disjoint TS partitions is unnecessarily synchronized (serialized). Distributing TS to separate kernel processes would provide concurrent access to multiple Linda processes.

- Methods for instanciating distributed eval() functions. Process generation using a fork during an eval() operation restricts Linda concurrency to a single node on the network. Another mechanism needs to be investigated which supports distributed parallelism using the eval() operation.

These and other issues are topics for future research and are discussed in more detail in chapter 7.
3.4 VPI Kernel Implementation Issues

Much of the preceding material has been introductory and background information for this section. The first half of this chapter addresses the Yale design and implementation of Linda, followed by the design elements of the VPI approach. The remaining sections of this chapter discuss details of the VPI implementation. Section 3.4 addresses specific issues which require further refinement in the VPI model. Primarily, implementation details which have an impact on run time performance are discussed. The final section of the chapter offers a critical appraisal of the design and implementation.

3.4.1 Message Based Kernel Services

Although not immediately obvious, there is an added level of complexity in the VPI design when the Linda process invokes kernel services. A C function call is all that is needed to communicate with the Yale kernel. There exists a separate function in the library for each kernel service available to the application. The compiler generates direct calls to the appropriate kernel routines. There is no run time overhead spent determining what service is being requested by the application.

Data (tuples or templates) is transmitted to the Yale kernel by using the parameter stack of the Linda process requesting kernel services. When the kernel returns information (tuples or an inp() status) to the process the kernel copies data from shared memory into the process' private memory space.

Two Kernel Designs
Although the VPI design provides equivalent functionality, it does so by placing the kernel in a separate Unix process. A kernel service is no longer a function call within the Linda process' code space. Likewise, no shared parameter stack exists between the application and the kernel. In the VPI design, the process must package the parameters into a message suitable for transmission through a socket. In addition, the particular kernel function service must be specified when the parameters are transmitted to the kernel process.

To solve this problem, the VPI implementation uses an interface protocol between the kernel and the Linda processes. The interface is designed to be portable across communication subsystems so that the use of sockets can be easily replaced by some other communication layer. The following paragraphs give implementation details about data transmission and service invocation between the kernel and the Linda processes.

Run Time Parameter Passing

The Yale kernel is targeted for shared memory platforms and, as such, the designers took advantage of particular architectural features to improve execution efficiency, for example, the use of C pointers in passing data is the rule rather than the exception. Extensive use of structures containing pointers to information which might be needed by the kernel are found throughout the implementation. The challenge here is to convert these run time structures into a serial stream that can be written through a socket. The receiver must be able to reconstruct the original structure from the serial data and pass it on to the intended code module.
Figure 3.3 illustrates the Yale run time structure for a data tuple or a template as it is passed between a Linda process and the kernel. (Internal kernel storage structures were not modified and are not addressed in this thesis.)

When the process sends a tuple or a template, these structures must be interpreted by the run time support code in the VPI library. The Yale structures are converted into two contiguous blocks of data:

- a header block that details how the block to follow is to be interpreted.
- a data block with a copy of the Yale run time structures converted to contiguous memory with all indirection resolved.

Figure 3.4 illustrates the results of serialization of the Yale data structures. This contiguous block, or packet, is suitable for transmission through any block or character driven I/O interface. As discussed below, the conversion process also places a function or service identifier in the header block that uniquely associates the new structure with a specific kernel service.

**Kernel Service Identification**

The Yale compiler converts each Linda operation into a set of data structures, discussed above, and a function call to the run time kernel. A plethora of support functions are supplied by the Yale kernel. Duplicate services are also found in the VPI kernel. By placing the kernel services in a distinct process these services are not call-able in the same way they are in the Yale design. The VPI design employs replacement routines for the Linda process with identical names as those expected by the Yale compiler. The compiler’s output is interpreted by the VPI I/O stub and is converted into a suitable form for serial transmission to the kernel. The kernel function needed for each call is **Two Kernel Designs**
Figure 3.3  Yale tuple/template data structure.
uniquely identified by placing a service request value in the header block previously described. Three types of services are provided. Tables 3.1 and 3.2 list the services necessary to duplicate the functionality of the Yale kernel. Table 3.3 lists the additional services needed to coordinate the many-to-one mapping of the Linda processes and the kernel.

3.4.2 Server's Response To Messages

The VPI kernel is designed as a stand alone Unix process which responds to messages that are passed synchronously through Unix sockets. The kernel is not interrupt driven and, therefore, polls each socket descriptor to determine if a client requires a kernel service. The kernel incorporates four strategies when responding to Linda process requests:

- Handle all new requests for a socket connection before processing incoming messages.
- Process certain messages without sending a response (out(), EXIT, INIT TS, LINK TS).
- Process certain messages with an immediate response (rdp(), inp()).
- Postpone the response to certain messages until tuple data is available (rd(), in()).

Response To Socket Connect Request

The connect call is part of the Unix sockets library and is typically used by a client in the client/server model. A Linda process must establish an interprocess communication channel with the kernel before any kernel services can be used. As part of the eval() operation every Linda process

Two Kernel Designs
Figure 3.4  VPI Serialized tuple/template structure.
establishes a socket connection with the kernel. Since the process blocks until the connection is established, the kernel is designed to respond to all incoming connection requests expeditiously.

Messages Not Requiring a Response

A process can out() a tuple without any synchronization with the kernel or other Linda processes in the application. Since no kernel-to-process communication are required, the process continues its thread of execution unblocked. (This is not the case in the Yale kernel where the out()ing process must obtain access to two critical sections and wake other processes before continuing its own computation.) The kernel receives the out()ed tuple and must perform three steps before the service is completed:

- Check for waiting rd() requests and respond with a copy if the template and tuple match.
- Check for waiting in() requests and send tuple to waiting process if the template and tuple match.
- If the tuple was not consumed by a in() request then deposit the tuple in TS for later access by a rd() or in() request.

The other messages which do not require a kernel response (EXIT, INIT TS, LINK TS) keep the kernel's view of the Linda environment up-to-date by sending the process identifier (PID) to the kernel when a process is created or terminates. The PID also aids in tracing every process-kernel transaction by the debugging support discussed later in this chapter.
Messages Requiring Immediate Response

The predicate forms of the in() and rd() operators (inp() and rdp()) are non-blocking. If a matching tuple is not found in TS, the kernel responds with a false indication. The operations were added to the language to allow the programmer to take alternative action when TS does not contain a match for certain tuples. If the tuple is found, the kernel's response is the same as the non-predicate forms.

Messages With Pending Responses

The tuple retrieval operators in() and rd() may cause the process to block if TS does not contain a matching tuple. In the VPI design the Linda process sends a template of the desired tuple to the kernel and blocks on a socket read waiting for the kernel to respond with a matching tuple. When the kernel has found a match, the matching tuple is sent to the blocked process through the socket interface.

This is in contrast to the Yale design which requires that the requesting process perform its own searching of TS. If a match is not found, the process is put to sleep until a potential match is deposited in TS by another process. The depositing process awakes all processes waiting for a tuple in the set and each process performs a check for a match on the new tuple.
3.4.3 Temporary Buffer Management

Due to the requirement for the sender to convert the Yale structures to the packet format used in the VPI system, and for the receiver to convert the packets back into the Yale structures, several memory operations are performed for each transmission. The size of each packet varies and is not predictable. As a means to reduce the number of dynamic memory allocations made, the VPI design re-uses temporary space when possible. Memory is obtained by calling malloc() only when the previously used buffer is too small for re-use. This approach results in a worst case scenario equal to calling malloc for each temporary buffer needed. However, the typical Linda process will acquire its largest buffer early in the life of the process and continue to re-use this space until process completion. Since this technique is used in both the kernel and the I/O stub, the run time demand for dynamic memory allocation is significantly reduced.

3.4.4 Tuple Space Memory Management

The nature of tuple space is such that the maximum run time requirements of any Linda program is not known. The Yale design requires that the programmer estimate at compile-time the maximum amount of memory that the kernel needs to complete the program. This restricts the executable image from handling larger than anticipated data sets. If a program exceeds the predefined maximum on shared memory, a run time error prevents the program from executing to completion.

This requirement is a direct side effect of the Yale design using shared memory for TS storage. During initialization, the Yale system acquires shared memory from the operating system. The amount of shared memory needed is specified as an argument to the Linda compiler. It could have

Two Kernel Designs
been specified as an argument to the Linda program instead, but the problem would still not be avoided. Since Linda processes, by definition, are de-coupled with respect to time and space, accurately estimating the peak TS requirement is difficult.

The VPI design does not use shared memory and, therefore, does not need to specify at any point in time what its maximum memory requirements will be. The standard malloc() and free() system calls are used to acquire tuple storage and return it to the system after a tuple is retrieved from TS. The VPI implementation does not require that the programmer or the user calculate maximum memory needs. The only limitation placed on the size of TS is the virtual memory limit imposed by the kernel's host.

3.4.5 Debugging and Tracing Facilities

Debugging facilities are incorporated into the kernel and run time library so that the system developer or, later, the programmer can acquire more detailed information about the system's performance. These facilities are controlled by command line arguments to the Linda program and the kernel. The information is written to a text file for post-mortem review. By comparing the kernel's debug file with those from the Linda processes, the internal behavior of the system can be thoroughly investigated.

During design and implementation it was clear that the standard Unix debugging facilities would not easily address the parallel environment of the Linda paradigm. Other members of the research team were interested in studying the dynamic nature of TS during program execution. To address these
concerns the debugging facilities provide three primary functions: low level kernel and process debugging, socket interface debugging, and kernel and TS tracing.

**Low Level Debugging**

Low level debugging is particularly important during the initial phases of system implementation and testing. The low-level debugging features available are:

- **Hexadecimal Dump.** There is a basic need for some type of non-intrusive method of obtaining a hexadecimal dump of a block of memory. This feature is available in several kernel locations. It is also used as part of other debugging features.

- **Kernel Data Structure Walk.** The conversion to and from serial packets and structures using several levels of indirection is particularly taxing on the kernel programmer. This feature provides a window into the kernel's internal storage structure which is not easily obtained by traditional methods.

- **Malloc tracing.** This ability is primarily needed during kernel development to detect memory leaks and boundary violations.

These low-level mechanisms provide the backbone of the two debugging facilities that follow.

**Socket Interface Debugging**

The passing of service requests and the conversion of Linda structures to serial packets between two processes complicates the testing and debugging process. The VPI kernel and run time library have

Two Kernel Designs
the capability to produce a hexadecimal dump of all packets transmitted through or received from a
socket. This ability allows the developer to unambiguously identify packets that are sent with those
that are received. In conjunction with the other debugging and tracing facilities, tuple and template
information can be traced through the entire process of a Linda transaction.

**Kernel and Tuple Space Tracing**

The desire by other members on the research team to study the characteristics of TS during program
execution gives rise to the need for tracing hooks in the kernel and I/O interface. These hooks
provide detailed interprocess message tracing as well as the ability to trace the progress of a tuple
through the kernel. A study of the storage, matching and migration of tuples and templates to and
from a Linda process can be accomplished with the information generated by the tracing support.
Information is generated when a template or tuple is sent out by a Linda process, received by the
kernel, participates in a match, gets deposited in TS, or is transferred to a Linda process.

### 3.5 Design Appraisal

Some of the advantages discussed in previous sections are not without costs. This section
discusses three penalties of the design that can be immediately identified: superfluous
synchronization of tuple space access, redundant memory copy operations, and Unix scheduling
effects on interprocess communications between the kernel and the Linda processes [SCHC91].

Two Kernel Designs
Because of these "design flaws" tuple transfer to and from TS is expected to suffer. The magnitude of these influences on tuple bandwidth cannot be quantified from the design alone.

3.5.1Serialized Access to Tuple Space

Because the VPI design centralizes the kernel routines and tuple space in a single process, all Linda operations are necessarily serialized. Since tuple space is implemented using private memory, concurrent access is forfeited by a single-thread kernel. In the VPI design all processes requesting kernel services send their requests to the kernel via Unix sockets and optionally wait for a response.

On a uni-processor system, requests continue to accumulate until the kernel is scheduled to run by the operating system. The kernel reads the requests one at a time in the order they have been received. Request processing continues until all messages have been serviced or until the scheduler suspends the kernel. Although in a multi-processor environment the kernel's execution is asynchronous with respect to the Linda processes, the kernel, none the less, serializes the servicing of request.

In terms of kernel service time per request, this course grain serialization has no effect vis-a-vis the Yale environment (on a uni-processor) since both environments are serialized at the CPU level. However, there are many situations in which concurrent kernel services do not require mutual exclusion, for example, concurrent access to disjoint tuple sets. In such cases, the VPI design compares unfavorably on multi-processor machines where the kernel code co-exists in the Linda process on several CPUs [CARN87]. Recalling that the compiler partitions TS into mutually exclusive
regions (tuple sets), serialization in multiprocessor environments is only required during concurrent access of a particular tuple set, generally a small subset of tuple space as a whole.

This deficiency is intentional in the VPI kernel given that this research is intended to provide the framework for subsequent distribution of the Linda application, tuple space, and the kernel. Several solutions to the serialization problem are presented in Chapter 5.

3.5.2 Redundant Memory Copying Operations

Redundant memory copy operations are inherent in the use of Unix sockets. The socket write/read sequence requires, at a minimum, two copy operations to deliver data from sender to receiver. First, the sender calls the write() function with a file (socket) descriptor. This transfers (copies) the data to a system send buffer for that socket. Second, the receiving process calls a read() function to transfer the data from the system's receive buffer to the private memory owned by the reading process. In the case of an internetwork socket connection a third copy operation is required. The send buffer is necessarily distinct from the receive buffer since they reside on separate hosts. The data is copied from the send buffer onto the network and then into the target's receive buffer.

In the VPI Linda environment all data being sent requires that it be preceded by a header block (approximately 32 bytes) telling the receiver what is being sent and how to reconstruct it by the receiver. The sender must convert the Yale tuple/template structure to a serial stream before the write() function can be called. This is basically an additional copy operation. Once an entire packet
(header and tuple/template) is received, the Yale structure must be reconstructed, thereby effecting another copy operation.

To summarize, for each Linda operation, the VPI design requires the following copy operations not performed in the Yale design:

- Sender converts the Yale tuple/template structure to contiguous byte stream.
- Socket write() call transfers the data from Linda process’ memory space to a system send buffer.
- Sending host copies the data from the send buffer onto the network.
- Receiving host copies the data from the network to a system receive buffer.
- Receiver process, via the read() call, transfers the data from the system buffer to private memory.
- Receiver converts the byte stream to the Yale tuple/template structure.

This sequence is illustrated in Figure 3.5.

Finally, many of these copy operations have associated memory allocation (malloc) overhead to provide storage space for intermediate buffers. Together, this represents a considerable degree of additional memory manipulation in the VPI environment not present with the Yale design.

3.5.3 Adverse Affects of Unix Scheduler

In the Yale environment Linda operations (kernel functions) are self-contained within the Linda process’s code space. Run time request for kernel services are implemented as function calls to

Two Kernel Designs
Figure 3.5  Redundant Memory Copy Operations.
library routines linked with the Linda application. The overhead associated with each kernel call is
negligible, being equal to the overhead associated with calling other subroutines in the application.

The VPI kernel, on the other hand, provides analogous functionality but in a separate process. The
Yale library function call is replaced with a message based protocol between the Linda application
and the kernel process. For the out() and eval() operations there is no penalty since the application
does not wait for a result from the kernel. Conversely, the in() and rd() are blocking operations and
must receive a reply from the kernel before proceeding. Although the blocking nature of a socket
read() is exploited by the VPI design, locating the kernel in a separate process implies a hidden cost:
scheduling delays.

On a uni-processor, a process executes only when the Unix scheduler permits it. Only one process
is given the CPU at a time. Other competing tasks (tasks able to run) must wait until the current task
either completes or gets blocked, or the scheduler determines that the executing process’ time slice
has expired. The VPI design dilemma is that an application doing an in() operation must wait until
the kernel is allowed to execute before any result can be received. After a Linda process sends a
template to the kernel for matching it gives up the CPU by blocking on a socket read() call. This
permits all competing processes, including the kernel, to have access to the CPU. Once the kernel
has serviced the request and found a match, the results are sent to the requesting process. The
requesting process is no longer blocked but must still wait until the scheduler allows execution and
thereby enabling the socket read() call to complete. The lag time from the kernel’s response until the
in() operation returns is dependent on the Unix scheduling algorithm. For the following discussion,
lag time will be assumed to be equal to a complete round robin of all ready processes with similar
priorities.

Two Kernel Designs
Consider the following Linda-C program segment:

```lindac
for i=0; i<4; i++
    out("foo");
for i=0; i<4; i++
    in("foo");
    computation_work();
```

Let $t_0$ through $t_3$ represent the wall time for four successful in() operations by a Linda process.

The Yale environment wall time for this code segment is shown by the following time line:

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| $t_0$ | $t_1$ | $t_2$ | $t_3$
```

where:

- $t_0$, $t_1$, $t_2$, and $t_3$ are tuple requests, i.e., "in()" operations.
- "———" is the time spent in "computation_work()".

Two Kernel Designs
The VPI design implies a delay in the in0 operation while the Linda process waits for the kernel to respond. This delay is non-deterministic since it is highly dependent on uncontrollable run time variants such as system and network loading. Adding this delay to the Yale time line illustrates the cost each in0 operation must pay for the socket based communication system. The time line with scheduling delays inserted for the VPI environment is shown by:

```
  t_0  t_1  t_2  t_3
```

where:

- t_0, t_1, t_2, and t_3 are tuple requests, i.e., "in0" operations.
- \[ \text{\rule{2cm}{0.1pt}} \] is the time spent in "computation_work0".
- \[ \text{\rule{2cm}{0.1pt}} \] is the time spent waiting for the kernel to return with a matching tuple and for the process to get the CPU.

Perhaps the most notable attribute of this skew in wall time is that the delay is largely dependent upon the number of competing processes to which the scheduler is giving CPU time. As the number of processes in the system increases, the time skew becomes more prominent.
"Wagner’s music is better than it sounds."

- Mark Twain

4.0 Initial Benchmark Results

The VPI kernel is tested for implementation correctness by executing a suite of Linda-C programs consisting of 24 Linda programs obtained primarily from Yale University. Some of the programs are realistic Linda applications. Other programs in the suite are designed to test a particular aspect of the environment, for example, matching algorithm correctness, or proper TS management. Running these programs in both the Yale and VPI environments demonstrates that the VPI and Yale kernels produce equivalent application output.

After demonstrating that the VPI implementation is functionally equivalent to Yale's, the next topic addressed is comparison of run time performance. Two programs from the test suite are presented in this chapter as illustrative of the initial performance differences of the two implementations. These programs are:

- Fastloop.cl, a program focusing on I/O bandwidth between the application and TS, and
- Cmatrix.cl, a more general benchmark involving various kernel searching algorithms and some computation by the application.
*Fastloop.cl* is of interest because it provides a very clear comparison of the relative costs of the two implementations. To place the cost comparison in perspective, *cmatrix.cl* provides a more meaningful comparison of how a real Linda application would behave in the two environments.

Chapter 5 presents the investigative path taken to isolate the primary bottleneck in tuple I/O that is described in this chapter. The discussion of *fastloop.cl* that follows is important background information for Chapter 5.

Chapter 6 presents the final benchmarks after kernel communications bandwidth has been improved. The data presented in Chapter 6 demonstrates the VPI implementation to be comparable to Yale's in most applications with some advantages in others. Of primary importance is that shared memory is no longer used as the physical storage media for TS, allowing further distribution of the VPI environment.

These three chapters represent a general chronology of testing the VPI implementation for correctness and efficiency.

### 4.1 The *Fastloop.cl* Benchmark

A simple Linda application, *fastloop.cl*, is used to test and measure tuple/template transmission between an application and the kernel. The program performs a series of out() operations on a CS tuple, «"BAR"», then retrieves the tuples using in() operations with the template «"BAR"».

Initial Benchmark Results
Key Code Segment from Fastloop.cl

```c
start_timer(); // mark time
for(i = 0; i < total_loops; i++) out("BAR"); // deposit tuples in TS
for(i = 0; i < total_loops; i++) in("BAR"); // retrieve tuples
print_times(); // print lapse time
```

Several important characteristics of fastloop.cl make it ideal for measuring Linda's tuple I/O. Three features are unique to fastloop.cl that the typical Linda-C application does not share. These attributes remove complexities not related to the communications subsystem, making fastloop.cl ideal for measuring kernel/application communications bandwidth.

- The benchmark is a non-parallel Linda program. Unlike most Linda applications, there are no eval()ed functions running concurrently. Neither the Yale nor the VPI environments are penalized by eval()ed processes competing for access to TS. Thus, the serialized access to TS discussed in Chapter 3 is factored out.

- Counting Semaphores are used for tuple storage. Also important to this benchmark is the use of actual parameters in the out() and in() operations. Since the in("BAR") operations are guaranteed to match any <"BAR"> tuple that may exist in TS, the compiler selects the counting semaphore (CS) algorithm for these transactions. Because the application pre-deposits enough <"BAR"> tuples, all in() operations are immediately serviceable by the kernel. Therefore, kernel matching speed is not a factor in this application.

- The majority of the application's time is spent in kernel I/O. Fastloop.cl does not perform meaningful (CPU intensive) calculations which would dilute the influence of communications on the applications time.

Initial Benchmark Results
However, fastloop_cl is ideally suited to exercise the key implementation details where the Yale and VPI kernels differ: passing data between the application and the kernel. Since the benchmark's time critical instructions are the out() and in() operations, two contrasts are made by comparing the run time results.

- Stack versus socket parameter passing. In the Yale environment, the process communicates directly with the kernel (incorporated into the application) using the application's parameter stack. By contrast, the VPI kernel is invoked after receiving a message from the application. As discussed in Chapter 3, numerous memory copy operations are required to transmit each "BAR" tuple and "BAR" template request to the kernel and receive the kernel's response. Additionally, the transmission must go through the operating system's socket interface and possibly over the network if the kernel is executing on a different host than the application.

- Function invocation versus context switch. In the Yale environment this program will not block on the in() operations since a matching tuple exists in TS for each in() performed. Furthermore, the CS kernel service is invoked by directly calling a kernel function that has been incorporated into the application at compile time. This presents an ideal scenario for the Yale environment. Tuple generation and retrieval is allowed to occur at the maximum I/O rate of TS, which is implemented as shared memory. But the application does not share this advantage in the VPI environment. Each in() operation involves writing a block of data out of the socket interface and waiting for the kernel to receive and respond. The application blocks on a socket read until the kernel's response is received. Each retrieval operation is a socket write, transmitting the template, followed by a blocking socket read, waiting for the kernel to return a matching tuple. In particular, the kernel must wait until it is scheduled to execute before it can read the template "BAR" , decrement the

Initial Benchmark Results
counting semaphore, and send the tuple back to the application. Once the application's socket has data available the process is ready to execute, although it must wait until scheduled by the operating system. In general, the application will block total_loops times, each block being at least one round robin of the Unix scheduler.

For the above reasons, fastloop.cl presents a good measure of the penalty that the VPI environment must pay for socket based communications. To better illustrate these ideas, in the following program segment the kernel's functionality is "inlined" into Linda-C code. Because the Yale kernel is included in the application as a linkable library, there is no waiting for the kernel to respond to the in() operations. Also, TS will always contain enough tuples to satisfy each in() so that no blocking is incurred.

```c
Yale Execution Thread

start_timer();
for (i=0; i < total_loops; i++) ++cs;       // increment tuple count
for (i=0; i < total_loops; i++) {
  while (cs == 0) wait;
  --cs;
}
print_times();
```

By way of comparison, the VPI run time library has been inlined in the program segment that follows. Since the kernel's functionality does not exist in the library, the application must send messages to the kernel process via the socket interface. Each message sent is received by the kernel after the application has undergone a context switch and the kernel has been scheduled to execute. The messages from the kernel must also wait until the kernel's time slice is complete and the process
VPI Execution Thread

```
print_times();
for (i = 0; i < total_loops; i++) send_tuple(); // tell kernel to increment cs
for (i = 0; i < total_loops; i++){
  send_template(); // send template for a match
  receive_tuple(); // wait for response
}
print_times();
```

gets CPU time. Additionally, there is some delay inherent in the transmission of data between the kernel and the application. With both the application and the kernel on a single node the delay should be minimal since the data only needs to travel down to the IP layer in order to be routed to the receiver. However, when two nodes are used, the data must be routed down to the physical layer and sent across the network.

By running `fastloop.cl` with various values for `total_loops` the round trip cost for an `out() / in()` transaction can be calculated. As Figure 4.1 shows, the cost for the VPI environment is over 100 times greater than the Yale environment. Also, whether the application is run in the VPI environment with the kernel on the same node [VPI(i)] or on a separate node [VPI(ii)], the cost per transaction is approximately equal.

Although this is an important measure, it does not indicate how much CPU time is being consumed by the application. Since the key objective of Linda is to provide concurrent processing, CPU utilization is an important consideration. As explained below, the poor wall time comparison does not

Initial Benchmark Results
Fastloop.cl Timings
Yale, VPI(i) and VPI(ii) Wall Times

Figure 4.1    Yale, VPI(i) and VPI(ii) Wall Time Comparison.
necessarily indicate that the VPI environment is consuming considerably more CPU resources. The two elements being measured by fastloop.cl (cost of redundant memory copy operations and Unix scheduling delay) can be isolated by looking at CPU utilization. Although present in the wall time data, time spent waiting for the kernel to execute and return a tuple would not be included in CPU time credited to the application. Time spent copying memory between buffers would be included in CPU metrics.

To investigate CPU utilization, the run time libraries of the two environments are instrumented to output more detailed timing information: specifically, the CPU time spent executing application code and the CPU time spent in the operating system on behalf of the application. (These timings are obtained with negligible overhead by using a single system call, times, just prior to program termination.) The data, presented in Figure 4.2 (application code time) and Figure 4.3 (operating system time), indicates that CPU utilization of the two environments is approximately equal. This is consistent with expectations because the real work being done consists primarily of memory moves and looping. The interpretation and implications of this data will be discussed after presenting data from a different benchmark, cmatrix.cl.

4.2 The Cmatrix.cl Benchmark

How does the VPI environment compare when the application performs meaningful work? One of the Linda programs used to investigate this question is cmatrix.cl, a matrix multiply program. The program allows the user to vary the degree of concurrency and the problem size by specifying the number of eval()ed functions and the size of the two matrices, respectively. The effect of

Initial Benchmark Results
Figure 4.2 Yale, VPI(i) and VPI(ii) Application Code Time.

Initial Benchmark Results
Figure 4.3    Yale, VPI(i) and VPI(ii) System Code Times.

Initial Benchmark Results
computation and concurrency on the Yale and VPI environments can be measured through these parameters. The execution time is, therefore, expected to be a function of the matrix size and the number of workers, such that \( t = f(M,W) \), where \( M \) is the matrix size and \( W \) is the number of eval()ed workers.

In the \texttt{cmatrix.cl} program, concurrency is achieved by eval()ing "worker" functions which compute individual row products serially. The main program performs the following steps:

- initialize the matrices so the results can be verified,
- eval() the appropriate number of "worker" functions,
- place the matrices in TS so that the eval()ed functions can read them,
- wait for all the dot products to be computed,
- collect the resulting columns and check the results.

Each eval()ed function computes a column by forming the dot product of matrix \( A \) and a unique row from matrix \( B \). The strategy of the "worker" function is:

- get the index of the next row from matrix \( B \) using \texttt{in()} and save locally,
- increment the index and place it in TS for another worker to \texttt{in()} ,
- get matrix \( A \) and the appropriate row from matrix \( B \) using \texttt{rd()}s,
- compute the dot product and place in TS, and
- loop back to get another row index.

If a worker determines that it has the job of calculating the dot product from the last row index, it will place a flag in TS so that all workers know that all the indices have been assigned. This signals the workers to terminate.

Data gathered from this benchmark shows that for a fixed matrix size very little difference is seen within the Yale environment when the number of workers is varied, that is, \( t = f(C, W) \) is constant for

\textbf{Initial Benchmark Results}
Yale vs. VPI(i) Cmatrix.cl Timings
Matrix Size vs. Wall Time

![Graph showing time vs. matrix size for Yale and VPI(i) Cmatrix.cl](image)

- Yale 6 Workers
- VPI(i) 6 Workers

Figure 4.4 Yale and VPI Cmatrix.cl Matrix Size versus Wall Time.

Initial Benchmark Results
a fixed matrix size (C) and a wide range of workers (W). This is easily rationalized since the amount of real work (total computation) and the amount of I/O (tuple transactions) does not change. The lapse time is expected to likewise remain stable. The same observation is made in the VPI environment: wall time is unaffected by the number of workers for any fixed matrix size.

As shown in Figure 4.4, when the degree of parallelism (number of workers) is held constant and the size of the problem is increased (by increasing the size of the matrices), both environments show polynomial growth in lapse time, that is, \( t = f(M, C) \) is polynomial. This is expected since the matrix multiply problem has a complexity of \( O(n^3) \). Of interest, however, is how the environments compare to each other using \textit{cmatrix.cl}. By applying a regression analysis using the model \( y = ax^b \), the constant \( A \) and the exponent \( B \) favor the Yale environment. It is noteworthy that exponent \( B \) is well within the same order of magnitude: 1.50 for Yale and 1.92 for VPI(). The constant \( A \) likewise favors the Yale environment: 0.0856 for Yale and 0.110 for VPI(), but again within an order of magnitude. This result is encouraging in that the more meaningful benchmark, \textit{cmatrix.cl}, is also more flattering than the less "balanced" \textit{fastloop.cl}.

### 4.3 Observations and Interpretation

The two benchmarks presented in this chapter, \textit{fastloop.cl} and \textit{cmatrix.cl}, provide several insights into the execution characteristics of the two environments and particularly into the performance penalty paid by moving away from shared memory as a basis for tuple space. Presented here are the key observations that motivate the more detailed investigation presented in Chapter 5. Each
observation is considered in light of the expected VPI performance cost presented in Chapter 3: scheduling delays, multiple memory copy instructions and socket transmission delays.

4.3.1 CPU Utilization

As demonstrated in Section 4.1, the two environments exhibit similar CPU requirements. Although the VPI implementation takes considerably more time to complete an application, this time is not spent consuming measurably more CPU resources. Therefore, the lapse time must be spent with the application inactive. The obvious speculation is that this delay is either caused by transmission delays between the application and the kernel or by operating system factors, such as scheduling. Also, this indicates that the cost of redundant memory copy operations in the VPI implementation has no measurable influence on the run time data. It is reassuring that the VPI metrics are favorable in this regard.

4.3.2 VPI Kernel Distribution

In the data presented above for fastloop.c, the VPI environment is tested in two configurations: the kernel and the application sharing a host [VPI(i)], and the kernel on a separate host from the application [VPI(ii)]. No significant difference in wall time exists between these two scenarios. Although this is also true of all the applications in the test suite on which CPU utilization data has been collected, it is unexpected since transmission across a LAN (interhost) should take considerably more time than intrahost communications. To verify that the intrahost socket calls do not needlessly

Initial Benchmark Results 78
send data over the network, a test has been performed in which the network is disconnected from the host. No significant difference in performance was measured.

The above demonstrates that physical I/O is not a factor in the poor performance of the VPI environment. Furthermore, it sheds doubt on the speculation that round robin scheduling delays cause a large skew in tuple transaction time since the kernel and application are able to execute simultaneously.

### 4.3.3 Foreground and Background Processing

During the early phases of data gathering, the VPI kernel was run in background while running the application in foreground. It had been suggested that the background process might not execute at the same priority as a foreground process. Consequently, the test suite has been re-run with the kernel in the foreground with the expectation that kernel response times would improve. The cost per transaction, however, does not improve under these conditions. This result places further doubt on the theory that the majority of the delay is due to scheduling phenomenon.

### 4.3.4 Cross Platform Comparison

For pragmatic reasons, both environments have been ported to a RISC (IBM RS/6000 320h) based system located off campus. After successfully porting, the test suite has been run again to provide a baseline for further research. By way of comparison, the Yale fastloop.cl benchmark performs five

**Initial Benchmark Results**
(5) times faster (wall time) on the IBM than it had on the Amiga UX 3000 (the original port). This is not surprising considering the difference between CPUs in the two systems. However, the VPI benchmark for this program shows no improvement for wall time. *Fastloop.cl* using the VPI kernel exhibits a 200 msec. per tuple transaction on both the RISC and the M68030 based platforms. This is unexpected considering the difference in CPUs. Also of interest, the system times (user code and system code) for the two implementations are also comparable on the RISC platform, that is, the VPI implementation consumes approximately the same amount of CPU resources as the Yale implementation. This result points strongly to a common cause for the poor TS bandwidth demonstrated for the VPI kernel. This casts further doubt on the original speculation that the majority of the performance penalty incurred by the VPI implementation would be traceable to scheduling delays and excessive memory copies.

4.4 Conclusions and Further Research Options

In this chapter the results from initial VPI testing are presented in a somewhat "mater-of-fact" format. Attempting to rationalize these results from the anticipated performance penalties presented in Chapter 3 is not very conclusive. In fact more questions seem to be raised than are answered when the testing presented here is viewed as a whole. In particular, the RISC based testing demonstrates that significantly increasing CPU horsepower has no effect on the VPI implementation.

By way of summary, the table below lists the expected causes of reduced performance from Chapter 3 and degree of support found in the data presented in this chapter.

**Initial Benchmark Results**
<table>
<thead>
<tr>
<th>Section 4.3.1</th>
<th>Scheduling</th>
<th>Socket I/O</th>
<th>Memory Copies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>possible cause</td>
<td>possible cause</td>
<td>unlikely cause</td>
</tr>
<tr>
<td>Section 4.3.2</td>
<td>unlikely cause</td>
<td>unlikely cause</td>
<td></td>
</tr>
<tr>
<td>Section 4.3.3</td>
<td>unlikely cause</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section 4.3.4</td>
<td>unlikely cause</td>
<td></td>
<td>unlikely cause</td>
</tr>
</tbody>
</table>

Clearly, further investigation is warranted. The two most critical questions at this point are: can the source of VPl's poor performance be isolated and can it be eliminated? To address this concern three approaches are considered:

- add time stamping to socket packets to trace their progress through the environment,
- further investigate the effect of the scheduler by varying the kernel and/or application priorities, and
- handcraft a program that communicates with the kernel at a lower level.

The third approach was chosen since the first was considered difficult and time consuming, and the second does not lead to a solution to the problem. Specialized Linda programs derived from the fastloop.cl benchmark are presented in the following chapter. The investigation using the modified fastloop.cl proves effective in both isolating and eliminating the cause of poor performance. A final comparison between the Yale and VPl implementations is presented in Chapter 6.
"What is a magician but a practicing theorist?"

- Obi-Wan Kenobi

5.0 Tuple Prefetching

In this chapter a special purpose program is used to investigate lower level details of the VPI application-kernel communications interface. After encouraging results from an initial experiment, four modifications are made to that program in order to isolate causes of observed phenomena in the tests results. At the conclusion of this chapter a major cause of VPI’s poor performance is identified and the corrective action is described.

5.1 Prefetch.cl

From the discussion of fastloop.cl in the previous chapter, the reader will recall that each in() operation incurs the cost of a context switch in the VPI environment. This is because each in() is effectively a socket write of the template and a blocking socket read waiting for the kernel to receive the template and respond with a matching tuple. The reader will also recall that fastloop.cl is a trivial Linda-C program which guarantees a tuple exist in TS for each in(). The code is repeated here as part of the discussion to follow:
Key Code Segment from Fastloop.cl

```
start_timer(); // mark time
for(i = 0; i < total_loops; i++) out("BAR"); // deposit tuples in TS
for(i = 0; i < total_loops; i++) ln("BAR"); // retrieve tuples
    timer_split("Retrieval done.");
    print_times(); // print lapse time
```

The VPI library functions which map to the Linda operators out() and ln() are inlined in the following code segment to aid in illustrating the derivation of the program used in this chapter:

VPI Execution Thread for Fastloop.cl

```
start_timer();
    for (i = 0; i < total_loops; i++)
        send_tuple(); // tell kernel to increment cs
    for (i = 0; i < total_loops; i++){
        send_template(); // send template for a match
        receive_tuple(); // read response
    }
    timer_split("Retrieval done.");
    print_times();
```

Process to kernel communications packets follow the sequence: send tuple1, send tuple2, send tuple3, \ldots, send tuplen, send template1, read tuple?, send template2, read tuple?, send template3, \ldots, send templaten, read tuple?. Clearly, from this code the send_template() in the second for loop can be moved up into the first for loop without any change in semantics. The code then becomes:

Tuple Prefetching
Prefetch Code

start_timer();
for (i = 0; i < total_loops; i++){
    send_tuple();
    send_template();
} // tell kernel to increment cs
for (i = 0; i < total_loops; i++){
    receive_tuple();
} // send template for match
} // read response
timer_split("Retrieval done.");
print_times();

The sequence of communication packets likewise changes to: send tuple1, send template1, send tuple2, send template2, send tuple3, send template3, . . . , send tupleN, send templateN, read tuple?, read tuple?, . . . This modification carries with it a very significant side effect: the operating system socket interface provides, in effect, a read ahead tuple cache. For this application, which will now be called prefetch.cl, all tuples and templates (matching requests) are sent to the kernel before blocking on a socket read, and quite possibly before yielding to the Unix scheduler. Therefore, prefetch.cl has the potential of only incurring one context switch between the start_timer() and print_times() function calls. During the period of time when prefetch.cl is waiting for the first tuple to be returned, the kernel will have received all the tuples and completed servicing at least one matching request. It is also possible for the kernel to have matched all the templates and sent all the tuples to the application before prefetch.cl is unblocked. By contrast, fastloop.cl requires a context switch for each tuple received from the kernel.

Recall that this program is not truly a Linda program. It is "handcrafted" in order to avoid implementation penalties of the VPI design. This is justifiable since the purpose of this program is to isolate one of the speculated causes of poor performance: delays due to blocking on each in() operation.

Tuple Prefetching
Figure 5.1 Prefetch.cl initial results.

Tuple Prefetching
Figure 5.1 shows the relationship between tuple count (T) and wall time (wt) when this program is run on the RS/6000. It is easy to see that wall time is related to tuple count by the equation: \[ wt = f(T) = [T/70] \times 0.200 \] (ignoring the initial step down). Whereas, the relationship between wall time and tuple count for fastloop.cl is: \[ wt = f(T) = T \times 0.200 \]. Several phenomena are seen in the results from this program:

- Improved TS bandwidth. Although the two programs perform exactly the same amount of kernel I/O, prefetch.cl shows a 70 fold (20 fold for the Amiga) improvement in kernel response time.
- Steps. Wall time is related to tuple count in a non-continuous fashion. No other application shows this type of behavior.
- Step shape. The height is approximately 200 msec. (also true for the data collected on the Amiga). The lengths are quantized in 70 tuple increments (the Amiga data showed a length of 20 tuples). The slope of the horizontal portion of the steps is nearly zero.
- Initial step down. Both the RS/6000 and the Amiga data exhibit a short initial step which is higher than its successor.

By comparing these observations with the data from fastloop.cl, we note that "priming" the kernel with templates has significant influence on wall time data. However, several questions are raised, answers to which, without further refinement of the program, can only be speculated. In particular,

- Noting that both the Amiga and the RS/6000 exhibit the same prefetch step size and the same fastloop.cl average tuple cost, do the height of the steps (200 msec.) and the average tuple turn around for fastloop.cl (200 msec.) have a common origin?
- Can the steps be isolated to the priming or retrieval loop?
- What causes the initial step down?
To pursue these questions three minor modifications to the test program are considered:

- time the priming and retrieval loops separately,
- use a rd() operation instead of an in(),
- place a sleep() function call between the loops.

By timing the loops separately the steps can be isolated to one of the two loops and likewise to either the socket write or read functions. By using a rd() operation the amount of data sent to the kernel is reduced by 50 percent. By placing a sleep between the loops a context switch is forced, allowing the kernel access to the CPU.

The next three sections explore the effect of these three modifications. The final section will present a fourth modification which is not obvious from the data presented thus far: experimenting with Unix socket options to observe their effect on wall time.

5.2 Priming and Retrieval in Prefetch.cl

Measuring the priming and retrieval loops separately has potential use in the other planned testing and, so, is chosen as the first modification on prefetch.cl. The following code segment illustrates the modifications needed to isolate the time measurements to their respective loops:
Code for Separate Priming and Retrieval Timings

start_timer();
for (i = 0; i < total_loops; i++){
    send_tuple();
    send_template();
} // tell kernel to increment cs
// send template for match
timer_split("Priming done.");
for (i = 0; i < total_loops; i++){
    receive_tuple();
} // read response
timer_split("Retrieval done.");
print_times();

The results from this modification point to a clear conclusion, that the stepping phenomenon has its origins in the retrieval phase of the program, as is seen in Figure 5.2. Five other observations can also be made:

- The "horizontal" portion of the steps have a negative slope. This negative slope is approximately equal to the positive slope of the priming curve corresponding to the step. The equal but opposite slopes tend to cancel out and the accumulative effect is seen in the relatively flat step of Figure 5.1.

- Although isolated to the retrieval phase of the program, the magnitude of the steps is unchanged, approximately 200 msec.

- The priming loop begins to show an increase in slope for tuple counts above 280. The average cost of priming for the entire data set is 1.5 msec./tuple. By dividing and analyzing the data in two zones, zone 1 (from 1 to 280 tuples) has an average priming cost of 1.1; zone 2 (from 281 through 400 tuples) has an average priming cost of 2.8.

- The retrieval average is 1.4 msec./tuple. However, using the same approach as above, zone 1 retrieval costs is 1.8 msec./tuple. Although zone 2 is somewhat

Tuple Prefetching
Figure 5.2 Priming and Retrieval Phases of *Prefetch.cl*.

Tuple Prefetching
erratic, linear regression analysis yields a cost of 0.9 msec./tuple.

- Opposite but equal "spikes" appear in the two curves. Between the range from 200 to 400 tuples six positive spikes are easily identifiable in the priming curve. Each spike is accompanied by a negative spike in the retrieval curve of approximately equal magnitude. This are annotated in Figure 5.2 by vertical line segments between the two curves.

- Finally, the initial step down is also isolated to the retrieval phase of the program.

5.3 Prefetch.cl with rd() Operations.

As mentioned above, replacing the in() operator with a rd() reduces the total number of socket writes by 50 percent. This change also affects the kernel's response time in that only one tuple is read from the application and placed in TS. There is no difference in the kernel's memory requirements since the tuple storage method is a counting semaphore. This will, presumably, lead to a reduced wait for the kernel's response since a) the process blocks on the socket read 50% sooner and b) the kernel has almost no work to do before servicing the rd() requests. The code for this test is:
in() Replaced with rd()

start_timer();
send_tuple();
for (i = 0; i < total_loops; i++){
send_template();
// tell kernel to increment cs once
// send template for match
// operation (does not remove tuple)
}
timer_split("Priming done.);
for (i = 0; i < total_loops; i++){
receive_tuple();
// read response
}
timer_split("Retrieval done.");
print_times();

This program is expected to complete the priming loop in measurably less time. However, it is not clear if the retrieval phase of the program would be affected. To the contrary, any change in the retrieval rate is unexpected since the code responsible for reading the tuples from the socket into local memory has not changed.

An additional change is made in order to a) further investigate the apparent change in priming cost between zone 1 and 2 and b) better characterize the affect of tuple count on the retrieval phase: the maximum tuple count was increased from 400 to 2000.

The results from these test are shown in Figure 5.3. From this data several observations (and speculations) are made:

- The magnitude of the steps remains unchanged.
- As noted in section 5.2, spikes in the priming data are accompanied by mirror spikes in the retrieval data.
- Two "zones" are clearly defined by a change in the priming and retrieval slopes. For this program, the transition from zone 1 to 2 occurs at approximately 580 tuple
Figure 5.3  
*Prefetch.cl* using rd() Operations
transactions.

- Priming costs in zone 1 are one 45 percent of zone 1 in modification 1: approximately 0.5 msec./tuple; zone 2 is approximately 1.8 msec./tuple versus 2.8 for modification 1. The average for the entire test is 1.5 msec./tuple, 60 percent of modification 1.

- Retrieval costs begin to level out. Although the stepping phonomania still exist, the average retrieval cost is much less a function of the number of tuples being retrieved. The average retrieval for the entire test is 0.5 msec./tuple. Retrieval cost in zone 1 (not shown in Figure 5.3) is 1.1 versus 1.8 msec./tuple for modification 1. Zone 2 retrieval cost is 0.2 msec./tuple. Although the slope is almost flat growth, there is a large offset.

The following table summarizes the results from modifications 1 and 2:

<table>
<thead>
<tr>
<th></th>
<th>Modification 1</th>
<th>Modification 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. prime:</td>
<td>1.5 msec./tuple</td>
<td>1.5 msec./tuple</td>
</tr>
<tr>
<td>zone 1 prime:</td>
<td>1.1</td>
<td>0.5</td>
</tr>
<tr>
<td>zone 2 prime:</td>
<td>2.8</td>
<td>1.8</td>
</tr>
<tr>
<td>avg. retrieval:</td>
<td>1.4</td>
<td>0.5</td>
</tr>
<tr>
<td>zone 1 retrieval:</td>
<td>1.6</td>
<td>1.1</td>
</tr>
<tr>
<td>zone 2 retrieval:</td>
<td>0.95</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Comparing the priming averages from these two test leads to the false conclusion that no measurable effect is seen when the rd() operation replaces the in(). The two averages are not strictly comparable since the tests are over vastly different ranges: modification 1 can only be compared to the first quarter of the data gathered from modification 2. However, when a comparison is made of the zone Tuple Prefetching
1 priming data from the two programs, slightly more than the expected 50 percent speed up is realized. Zone 2 demonstrates approximately 35 percent faster priming.

The unexpected result from this test is that the retrieval phase of the program experienced an almost three fold speed up in the average: 1.4 versus 0.5. This is, likewise, a skewed interpretation of the data since the two tests cover different domains. Nonetheless, a significant reduction is seen when the data from zone 1 of the two test are compared, nearly 40 percent. Zone 2 comparison is considered insignificant since the value for modification 1 is not reliable due to large variability and small sample size.

Two conclusions can be drawn from the data presented in this section. Firstly, the retrieval phase of the program is not as independent of the priming phases as originally speculated. What mechanism causes the retrieval rate to improve over modification 1? One speculation is that the kernel is performing 50 percent fewer socket reads allowing it to respond in a more timely manner. Another possibility is that some unknown side effect of the socket interface is involved.

Secondly, the time spent writing to the socket is directly proportional to either a) the number of socket calls, b) the amount of data being sent to the receiving process, or c) both.

Yet unanswered is the question, what causes the abrupt change in slopes between zones 1 and 2? Perhaps this represents the threshold where the program's time slice is complete and the Unix scheduler interrupts the I/O process. If this is the case then we would conclude that the granularity of the scheduling algorithm is approximately 250 msec. (The y axis corresponding to the beginning of zone 2). But this is inconsistent with normal operating system characteristics. Typically, time sharing applications are activated more frequently than once every quarter second. This change

Tuple Prefetching
might indicate that the program's socket out buffer or the kernel's in buffer is being filled and causing the process to be suspended until the buffers can be flushed. Support for this speculation can be found in the observation that the change from zone 1 to zone 2 is related to the amount of data being written out of the program's socket. By contrasting the length of zone 1 for the priming phase in these two tests we note that reducing the number of socket writes by a factor of 2 increases the threshold by a factor of 2. However, suspension of the program should only cause a "step" change, an offset, but not a change in the slope of the curve.

5.4 Forcing a Context Switch in Prefetch.c1

A third modification to the test program prefetch.c1 is presented here. By inserting a sleep between the priming and retrieval loops the program is forced to undergo a context switch. This allows the kernel time to read and process all the tuple and template messages sent during the priming phase of the program. If the sleep is long enough and the socket buffers are large enough, the kernel should be able to satisfy all templates by sending tuples to the program for each template received.

The main body of the test program is presented below:
The measurements for the retrieval phase of the program do not include the time spent suspended.

As illustrated in Figure 5.4, the results from this modification are significant and surprising:

- The retrieval slope is considerably reduced. In zone 1 this program shows more than a six fold increase in the retrieval loop's efficiency.

- The steps have been moved from the early retrieval phase to beyond 1100 tuple transactions.

- The initial step down has been removed.

- The priming and retrieval phases behave independently. By contrasting with modification 1, two features of the data indicate this independence: a) the retrieval data has no "mirror spiking" that corresponds to the priming data, and b) the length of zone 1 is much shorter for the priming phase than it is for the retrieval.

The following table summarizes the results from modification 1 and 3. Again, only two differences exist in the test programs: a sleep between the priming and retrieval loops and the range over which
VPI Prefetch Modification 3
RS/6000 7/27/92

avg. prime: 2.6 msec./tuple
avg. retrieval: 0.5 msec./tuple

zone 1 prime: 1.1
zone 1 retrieval: 0.27
zone 2 prime: 2.8
zone 2 retrieval: 1.9

Figure 5.4 Forced Program Suspension.

Tuple Prefetching
the benchmark is run.

<table>
<thead>
<tr>
<th></th>
<th>Modification 1</th>
<th>Modification 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. prime:</td>
<td>1.5 msec./tuple</td>
<td>2.6 msec./tuple</td>
</tr>
<tr>
<td>zone 1 prime:</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>zone 2 prime:</td>
<td>2.8</td>
<td>2.8</td>
</tr>
<tr>
<td>avg. retrieval:</td>
<td>1.4</td>
<td>0.48</td>
</tr>
<tr>
<td>zone 1 retrieval</td>
<td>1.8</td>
<td>0.27</td>
</tr>
<tr>
<td>zone 2 retrieval</td>
<td>0.95</td>
<td>1.9</td>
</tr>
</tbody>
</table>

After examining the results from these three modifications on a pseudo Linda program we have not uncovered any information that can be immediately applied to the VPI implementation. However, two encouraging observations can be made about modification 3 of prefetch.cl:

- The stepping phenomena is not related to the number of socket writes or to the amount of data being sent to the kernel. This program performs twice the number of writes and sends twice the amount of data but does not exhibit the stepping until over 1100 tuples and templates have been transmitted to the kernel.

- Timing has profound effect on tuple availability. When the reading of the tuple has been delayed by a sleep, modification 3 is able to retrieve matched tuples at a rate comparable to the Yale implementation. This can be seen by comparing the aggregate priming and retrieval times from zone 1 of VPI's prefetch.cl modification 3 with Yale's fastloop.cl, 1.37 versus 1.64 msec./tuple, respectively.

All the previous testing assumes that the bottleneck is in the VPI implementation. We now have some evidence that timing of the retrieval phase plays a critical part in efficiency. This raises the

**Tuple Prefetching**
possibility that the problem with the VPI implementation is not how it uses the socket interface but when.

Interestingly, zone 1 retrieval rate for modification 3 is the best among the 3 presented thus far. Speculating the cause of this, two possibilities are offered:

- **Scheduling.** When the program suspends for an extended period of time, in this case for 1 second, the kernel is able to process significantly more templates and respond with matches for each. Upon reactivation, the application finds ample tuples waiting to be read from the socket.

- **Lapse time.** Unix sockets are buffered much like Unix files are. When data is written to a socket or file it remains in a temporary buffer until a block has accumulated. The block is then transferred to the receiver, either disk or target process's socket. This method is much more efficient than sending data in small, indeterminately sized packets. For Unix sockets the buffer has associated with it a time out which insures that a partially filled buffer does not wait indefinitely before being transferred. Presumably, by suspending the program all associated buffers are flushed providing data immediately to the program upon its reactivation.

Similar reasoning could be used to speculate on the mechanism causing the steps to disappear from the retrieval phase for tuples less than 1100. Presuming that the steps and poor performance of VPI's fastloop.cl have a common origin (section 5.1), how can the steps be removed without suspending the program? Obviously, other socket based applications such as rlogin[ROSK90] and ftp[ROSR90] are able to communicate without delays on the order of 200 msec.

Considering these ideas, three possible modifications to prefetch.cl could be tested next:

**Tuple Prefetching**
- further pursue the effect of suspending the program by adjusting the length of the sleep to see if the location of the retrieval phase zone 2 changes, or
- modify prefetch to allow various size tuples to be sent thereby forcing the buffer to be full sooner, or
- experiment with various socket options in order to modify the behavior of the buffer filling and time out features.

The first option, adjusting the sleep time, could further validate the previously stated speculations, but it does not contribute to a solution to the problem. The second option would generate data which is not comparable to the other prefetch modifications. Since the third option would likely be tried irrespective of the outcome of the first option, the third option is attempted first.

### 5.5 Unix Socket Options

During design and implementation of the VPI kernel, primary attention was paid to providing kernel services functionally equal to the Yale implementation. Only those socket details needed to support kernel-process communications were investigated. Consequently, with two exceptions, the various socket options available were allowed to assume their default settings. This section describes the results of reevaluating this premise.

Unix socket options are numerous, manipulating the IP, TCP and socket protocol layers [STEW90]. Four areas of interest exit which might be of immediate usefulness in the VPI implementation: buffer size, low-water mark, socket buffer time-out, and TCP delay. Buffer size and low-water mark effect the amount of data that is buffered by the sender to be transmitted as a single packet. The naive,

**Tuple Prefetching**
and original, assumption is that each socket write effectively causes a packet to be sent to the receiver with out delay. Unix sockets, for reasons of efficiency, buffer the data until a full unit can be transmitted.

Buffer time-out prevents partially filled buffers from waiting indefinitely. Unfortunately, this option is not universally implemented, in particular it is not implemented on the RS/6000.

TCP level delay (TCP_NODELAY) controls the delay option which, when active (the default), provides timed buffering at the TCP level [IBM92, STEW90]. Data written to the socket is delivered to the TCP layer and buffered until a timer expires and is then transmitted to the receiver en mass. This reduces network congestion in much the same way that file buffering reduces disk thrashing.

By setting this option, socket based applications like rlogin and ftp are able to behave in an interactive fashion[ROSK90]. Terminal sessions which require a fixed number of keyboard entries before transmission to the host would, in many cases, be almost unusable.

Testing these and other options could consume considerable amounts of time. Priority is, therefore, assigned to each option based on its potential for contributing to resolving the problem at hand. The literature indicates that TCP_NODELAY is the most likely candidate to influence the timing of socket data dispatch and delivery. Three test configurations are planned using the option, first on the application prefetch.cl, then on the kernel, and finally, on both. For ease of implementation, the order of testing is: application alone, application and kernel together, kernel alone.

The application used for these tests is the original prefetch.cl (the intervening sleep between priming and retrieval phases is removed and an InQ() is the retrieval operation). As Figure 5.5 illustrates,
VPI Prefetch Modification 4a

RS/600C 8/1/92

avg. prime: 2.6 msec./tuple
avg. retrieval: 0.38 msec./tuple

zone 1 retrieval: 2.0
zone 2 retrieval: 0.20
zone 1 prime: 1.3
zone 2 prime: 2.8

Figure 5.5 Using socket option TCP_NODELAY in the application.
applying the socket option (TCP_NODELAY) gives run-time data similar to other tests; that is, both prime and retrieval curves have distinct zones and the retrieval data has a characteristic "saw tooth" shape. Closer examination, however, reveals two significant differences between this modification and those previously presented:

- No initial step down. As with the third modification, the retrieval metrics no longer have an initial negative step.
- Efficient Retrieval. The average retrieval rate is reduced to approximately one fourth of the original prefetched when this socket option is used.

These differences indicate that the efficiencies of modification 3 are attainable without suspending the application and without an associated cost to the application. What will be the result of using TCP_NODELAY in both the application and the kernel is still unknown at this point. Figure 5.6 presents the results from this experiment. Although the metrics do not show any improvement, the shape of the plots are unique in several ways:

- No retrieval steps. Poor VPI performance has long been suspected to be related to the quantization (stepping) found in the retrieval phase. Therefore, the removal of steps in the retrieval data is considered the most significant discovery among the tests run on prefetched.
- Jagged priming data. By comparing the priming curves from this test with all previous tests in this chapter, we note the previous tests have smoother priming plots. Exactly why setting the TCP_NODELAY option would cause this will not be addressed. However, by comparing modification 3 (priming and retrieval phases separated by a process sleep) with modification 4 (using the TCP_NODELAY option) we note that the intervening sleep virtually removes the steps from the retrieval data. As previously stated, modification 3 allows the kernel to process
considerably more tuple deposits [put(s)] and template request than it would have if the application entered the retrieval phase immediately (modification). We can speculate, then, that the steps are related to the kernel's ability to keep up to pace with the application. The data presented in Figure 5.6 supports this theory by illustrating that by using the TCP_NODELAY option in both the application and the kernel a very smooth retrieval curve is observed.

The table below is presented to summarize the results from the five tests performed using various modifications on prefetch.cl:

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Modification 1</th>
<th>Modification 2</th>
<th>Modification 3</th>
<th>Modification 4a</th>
<th>Modification 4b</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. prime:</td>
<td>1.5 msec/tuple</td>
<td>1.5 msec/tuple</td>
<td>2.6 msec/tuple</td>
<td>2.6 msec/tuple</td>
<td>2.6 msec/tuple</td>
</tr>
<tr>
<td>zone 1 prime:</td>
<td>1.1</td>
<td>0.5</td>
<td>1.1</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>zone 2 prime:</td>
<td>2.8</td>
<td>1.8</td>
<td>2.8</td>
<td>2.8</td>
<td>3.1</td>
</tr>
<tr>
<td>avg. retrieval:</td>
<td>1.4</td>
<td>0.5</td>
<td>0.48</td>
<td>0.38</td>
<td>0.83</td>
</tr>
<tr>
<td>zone 1 retrieval:</td>
<td>1.8</td>
<td>1.1</td>
<td>0.27</td>
<td>2.0</td>
<td>2.4</td>
</tr>
<tr>
<td>zone 2 retrieval:</td>
<td>0.95</td>
<td>0.2</td>
<td>1.9</td>
<td>0.20</td>
<td>0.26</td>
</tr>
</tbody>
</table>

The final test planned, setting TCP_NODELAY in the kernel only, is abandoned as being purely academic since more will be gained by testing a real Linda application with this option.
Figure 5.6  Using TCP_NODELAY in the application and the kernel.
5.6 TCP_NODELAY in Fastloop.cl

This chapter focuses primarily on the use of the prefetch.cl program as a test application. However, the data from using the socket options TCP_NODELAY on a conventional Linda program, fastloop.cl, is presented here as the conclusion to these experiments designed to isolate the cause of VPI's poor performance.

Five tests using fastloop.cl are presented in this section. They are:

- fastloop1.cl, with the original VPI library and kernel as a baseline,
- fastloop2.cl, using a_rd() instead of an_in() operation for comparison to prefetch.cl modification 2,
- fastloop3.cl, using the original VPI kernel, an_in() operation, but the TCP_NODELAY option set in the VPI library,
- fastloop4.cl, using the original VPI library, an_in() operation, but the option set in the VPI kernel,
- fastloop5.cl, using an_in() operation but with the option set in both the kernel and the library.

The results from these tests is presented below:

<table>
<thead>
<tr>
<th>Test</th>
<th>Code</th>
<th>TCP_NODELAY</th>
<th>Avg. Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastloop1</td>
<td>for( . . ) out(); for ( . . ) in();</td>
<td>none</td>
<td>200 msec./tuple</td>
</tr>
<tr>
<td>fastloop2</td>
<td>out(); for ( . . ) rd();</td>
<td>none</td>
<td>200 msec./tuple</td>
</tr>
<tr>
<td>fastloop3</td>
<td>for( . . ) out(); for ( . . ) in();</td>
<td>Process only</td>
<td>4.3 msec./tuple</td>
</tr>
<tr>
<td>fastloop4</td>
<td>for( . . ) out(); for ( . . ) in();</td>
<td>Kernel only</td>
<td>201 msec./tuple</td>
</tr>
<tr>
<td>fastloop5</td>
<td>for( . . ) out(); for ( . . ) in();</td>
<td>Kernel and Process</td>
<td>4.3 msec./tuple</td>
</tr>
</tbody>
</table>

Tuple Prefetching
This data represents the major break through of this chapter:

By setting this socket option the VPI environment is able to perform almost 50 times more efficiently.

Also, from this data we conclude that setting the option in the application is critical. Apparently setting the option in the kernel has little effect on the efficiency of the socket interface in the VPI environment. The exact reason for this is not known. One possibility is that the default value for the TCP_NODELAY option is true for a process performing a socket select, such as a server process.

That fastloop1.cl and fastloop2.cl perform nearly identically is not very surprising. The reason for this is found in the fact that both applications are constrained by a blocking socket read. Each application must suspend until the kernel receives the template, matches it and returns a tuple. Without setting the TCP_NODELAY option the Linda program must wait for the socket buffer to time out for each template it sends to the kernel. Therefore, each in() operation incurs a cost approximately equal to the amplitude of the steps identified in prefetch.cl: 200 msec.

By comparing the Yale and VPI environments we make the following observation: incorporating the kernel into the application is almost 3 times more efficient for this purely I/O application, fastloop.cl. The VPI implementation is expected to suffer some cost penalty for separating the kernel out of the application and placing it in a separate process. Surprisingly, stacked based parameter passing is only 3 times more efficient than socket based streams. Again, this is an acceptable cost if a multi-processor Linda environment can be constructed from low end desk top machines connected via a local area network.

Tuple Prefetching
The next chapter compares the two environments by using a larger suite of Linda applications. Also presented is the VPI environment executing in a distributed fashion, with the Linda program and the kernel executing on separate host.
"If you torture the data long enough it will confess."

-Source Unknown

6.0 The Yale and VPI Environments

This chapter presents the final results comparing the Yale and VPI implementations with respect to execution speed. Five Linda applications, designed to quantify specific aspects of the Linda implementations, are presented. Kernel-application communications throughput is addressed by two applications: fastloop.cl and bulk.cl. The cost of eval()ing in the two implementations is explored by eval.cl. Finally, two additional applications, cmatrix.cl and prime_act.cl, are presented as representatives of how the two systems compare with "real world" Linda programs.

In order to observe trends, the test applications are characterized by measuring the wall time required to perform increasing work loads or increasing concurrency. Each application is benchmarked in five different configurations:

- the Yale implementation on an Apple Macintosh,
- the VPI implementation with the kernel and application on a single Apple Macintosh,
- the VPI implementation with the kernel and application on separate Apple MacIntoshes,
- the Yale implementation on an IBM RS/6000,
- the VPI implementation with the kernel and application sharing a single IBM RS/6000.
The Apple MacIntoshes are M68020 (CISC) based micro-computers with 16 Megabytes of memory running A/UX. The IBM RS/6000 is a RISC based workstation also with 16 Megabytes of memory running AIX.

6.1 Linda Throughput: Yale and VPI Communication Costs Compared

6.1.1 Fastloop.c

*Fastloop.c*, the program used in previous chapters to quantify and debug kernel-application communications, gives a simplistic measure of relative communication performance of the two implementations. When this program is executed on the MacIntoshes, as shown in Figure 6.1, the results indicate that the shared memory based environment (Yale) is more efficient than the socket based environment (VPI).

The Yale implementation demonstrates approximately ten times the kernel-application communications throughput as the VPI implementation with both the kernel and the application executing on the same host. The advantages of the Yale environment are attributed primarily to two Yale design features: the speed advantage of shared memory and the non-blocking nature of the Yale kernel services (c.f. Section 3.5.3 and 4.1).
Figure 6.1  Fastloop.cl Comparison on Macintosh

The Yale and VPI Environments
As is generally the case with all the applications presented in this chapter, the VPI(ii) configuration (the VPI kernel on one host and the application on another) has a slight advantage (approximately 9%) over the shared hosts configuration, VPI(i).

When fastloop.cl is run on the RS/6000 the Yale environment is a factor of 15 times faster than the VPI single host environment (Figure 6.2). As on the MacIntoshes, Yale's use of shared memory and non-blocking kernel calls are the likely causes of its relative speed advantage.

Fastloop.cl is unique among the programs discussed in this chapter in that the only type of tuple being used is the counting semaphore (CS). This implies that no actual data is being transferred between the application and the kernel, only an identifier of which semaphore is being manipulated. Finally, this Linda application also contains no eval()ed functions, therefore, it is not a parallel program. Because of these constraints, fastloop.cl provides a simplistic measure of communications overhead with the kernel without the added cost of large tuples or kernel matching.

### 6.1.2 Bulk.cl

In contrast to fastloop.cl, the next benchmark does not change the total amount of data sent to and retrieved from the kernel. Instead, in order to measure the effect of tuple size, bulk.cl varies the size and number of tuples while keeping the total amount of data transmitted to and retrieved from the kernel constant. (The total amount of data transferred is 400k bytes. The block size varies from 4k to 40k while the number of blocks varies from 10,000 to 1,000.)
Figure 6.2  *Fastloop.cl* Comparison on RS/6000

The Yale and VPI Environments
Figure 6.3 illustrates the performance of the Yale, VPI single host and VPI two hosts configurations when running *bulk.cl* in the MacIntosh environment. The most obvious characteristic of this figure is the erratic nature of the Yale implementation when the block size is small and the number of blocks is high. As the number of blocks decreases and the size of each block increases, the Yale implementation outperforms both the VPI single host and the VPI two hosts environments by two to one. Because of limited availability of the MacIntosh systems, the test has not been re-run to determine if other factors influencing the test are the cause of erratic behavior in the Yale environment.

When these results are contrasted with those presented for *fastloop.cl*, we note that there is less disparity between the Yale and VPI environments for *bulk.cl*. The speed advantage of Yale over VPI(i) is a factor of 10 for *fastloop.cl* and a factor of 2 for *bulk.cl*. Above, the *fastloop.cl* benchmark is described as being a "purely I/O" application. Certainly, *bulk.cl* is at least as I/O intensive as *fastloop.cl*; in fact, *bulk.cl* transfers considerably more data per tuple than *fastloop.cl*. Since these "purely I/O" applications do not closely agree it is prudent to make a distinction between the two applications. *Fastloop.cl* is more accurately described as a "purely out0/in0" application. The amount of data transferred between *fastloop.cl* and the kernel is the absolute minimum needed to identify a tuple set. *Bulk.cl*, conversely, adjusts the amount of data per tuple in order to measure the effect of tuple size on transfer rate. Therefore, by comparing the results from *fastloop.cl* with those from *bulk.cl*, we conclude that:
Figure 6.3  Bulk.cl Comparison on MacIntosh

The Yale and VPI Environments 115
the major contributor to VPI's slower tuple I/O is related to the relative inefficiency of the VPI in() and out() operations and not to the actual transfer of data between the application and the kernel.

Additional evidence leading to this conclusion can be seen in Figure 6.4, where the data from the VPI(ii) tests is plotted so that intraconfiguration trends can be identified. Clearly, as the block size increases, the time required [and the number of out()in() operations] decreases in a non-linear fashion.

Assuming the above conclusion is correct, the bulk.cl data should then demonstrate a linear relationship between the number of blocks, effectively the number of out()in() transactions, and wall time. Such a relationship does indeed exist, as illustrated in Figure 6.5. Where Figure 6.4 does not clearly illustrate a relationship between block size and time, Figure 6.5 graphically demonstrates a linear relationship between the number of out()in() operations and wall time. As additional confirmation of this conclusion, we note that the VPI(ii) growth rate for bulk.cl (13.9 msec./block) is surprisingly similar to that recorded for fastloop.cl on the MacIntoshes (15.0 msec./loop).

We then conclude that:

the VPI implementation does not pay a performance penalty as tuple size increases.

Furthermore, lower I/O transfer rates in the VPI implementation can not be attributed to the socket based kernel interface, per se.

The Yale and VPI Environments
Figure 6.4  VPI(ii) running Bulk.cl, Block Size versus Wall Time.

The Yale and VPI Environments
Figure 6.5  VPl(ii) running Bulk.cl, Number of Blocks verses Wall Time
If sockets were a major contributor to I/O costs then it would be reasonable to expect applications benchmarked with the kernel executing on separate hosts would perform more slowly due to physical transmission delays. That this is generally not the case indicates that some other factor or factors play the dominant role in VPI overhead. Therefore, other factors, such as process scheduling and the implied context switch, are considered more influential on I/O throughput than the socket interface.

To summarize the results from running bulk.c in the MacIntosh environment, the following conclusion is made:

the major performance cost incurred by the VPI design is attributable to the implied context switch of the VPI kernel services. Yale kernel services are invoked via a much more efficient method, a procedure call, and do not effect a context switch.

Figure 6.6 presents the same test run on the RS/6000. Clearly, in this test the Yale implementation is not only more stable than that seen on the Macintoshes, it significantly out performs the VPI implementation. The trends in this data are similar to those observed in Figure 6.3, i.e., logarithmic decay particularly noticeable in the VPI data.

When the RS/6000 test results are transformed into number of blocks versus wall time, the conclusions presented above are confirmed: the VPI implementation pays a penalty (a context switch) for placing kernel functions in a separate process, whereas the Yale system employs a non-blocking function call. In addition to illustrating the linear relationship between out()/in()
Figure 6.6  Bulk.cl Comparison on RS/6000.

The Yale and VPI Environments
operations and wall time, Figure 6.7 shows surprising similarity between the slopes presented for fastloop.cl running on the RS/6000 and those presented for bulk.cl. Taken as a whole, these seven graphs (Figures 6.1 through 6.7) illustrate that, for their respective environments, the growth rates (slopes) for fastloop.cl and bulk.cl are nearly identical regardless of tuple size. (However, the data shows bulk.cl has a much larger startup cost (y-intercept) than fastloop.cl.) The two applications differ significantly in that:

- fastloop.cl is a single threaded program and bulk.cl makes use of concurrency by eval()ing a function to consume the blocks that are generated,
- fastloop.cl has very modest memory requirements, whereas bulk.cl incorporates a stack based integer array of 10,000 elements.
- kernel support for bulk.cl requires a large number of malloc calls, fastloop.cl has modest dynamic memory requirements.

6.1.3 Eval.cl

In Linda concurrency is established through use of the eval() operation. Therefore, a measurement of the cost of the eval() in the two implementations is of interest. For this purpose, a benchmark, eval.cl, is presented which measures the amount of time required to eval() a concurrent process. This benchmark does not include the time required for the eval()ed process to complete, only the time required to instantiate it. The source code for this application follows:
Figure 6.7  Bulk.cl comparison on RS/6000.
/** eval.cl: A benchmark program for Linda eval()s. ***/

real_main(int argc, char *argv[]){
    int count = atoi(argv[1]);
    start_timer();
    while(count--) eval(1);
    timer_split("Done.");
    print_times();
}

Figure 6.8 illustrates the results of this benchmark on the RS/6000. (The metrics are not available for the Macintosh configurations since the benchmark was written after the Macintosh testing was completed.) The data shows that the Yale implementation has a 33 percent advantage over VPI. The exact reason for this is not known although the following speculation is given.

Recall that the eval() operation is implemented as a process fork by calling the Unix system function fork(). During a process fork one of the two processes continues to execute while the other waits for the CPU. In the Yale implementation, eval.cl would fork a process whose only function would be to place the tuple <1> into TS (shared memory) and exit. The VPI implementation requires more initialization. The eval()ed process must establish a communications link with the kernel and send the tuple <1> before it exits. The author speculates that the difference between the Yale and VPI eval.cl wall time data is due to the higher VPI initialization cost of the eval()ed process.
Figure 6.8  Eval.cl Comparison on RS/6000.
6.2 Linda Application Performance: Yale and VPI Compared

The two applications presented below represent a more general comparison of the Yale and VPI implementations. These programs are "real world" applications in that the programmer uses the Linda environment to solve a given problem, whereas the programs presented earlier are designed to stress test a particular Linda feature.

6.2.1 Prime_act.cl

*Prime_act.cl*, identifies all the prime numbers below 1,000,000 by defining segments of equal length, each of which is examined by an eval()ed function. For this evaluation the size of each segment is fixed at 2000 integers, resulting in a total of 500 segments. A number of eval()ed functions is generated and the parent process waits for all of them to complete. Each eval()ed function searches a segment for primes by checking each odd number in the segment for primality. The number is prime if it is not divisible by any prime number less than the square root of itself. By varying the number of eval()ed functions the effect of increased parallelism is observed in each configuration. (Since the Yale implementation failed to execute successfully on the MacIntoshes, the Macintosh VPI(i) and VPI(ii) configurations and the RS/6000 Yale and VPI(i) configurations are presented below.)

VIP(i) and VPI(ii) on Macintosh A/UX

Figure 6.9 illustrates the effect that the number of eval()ed functions has on *prime_act.cl* wall time performance. This data presents a unexpected difference between the VPI(i) configuration (where

The Yale and VPI Environments
Figure 6.9  Prime_act.cl on MacIntoshes.

The Yale and VPI Environments
no network communications are involved) and the VPI(ii) (requiring network support). Not only does the distributed configuration [VPI(ii)] significantly outperform the single host setup [VPI(i)], an exponential increase in time is required by the single host environment as the number of eval()ed functions decreases below 10. Yet, when concurrency is greater than 10 eval()s, the wall time required is proportional to the number of eval()ed functions. These two phenomena will be considered separately.

For the single host configuration [VPI(i)], when the number of eval()ed functions increases from 1 to 8 the wall time results improve dramatically as concurrency increases. This is not the case for the distributed configuration [VPI(ii)], which demonstrates a linear growth in wall time as the number of eval()ed functions increases. The amount of computation and kernel I/O is equal between these two configurations. The exponential decay characteristic of the Yale data will be further discussed when the RS/6000 execution data is presented.

When comparing the two configurations for eval()ed functions greater than 10, the growth (slope) is approximately equal, although the VPI(ii) demonstrates a 10 to 12 percent advantage (negative offset). The data is insufficient to determine if the two curves would continue this trend, diverge or converge as the number of eval()ed functions increases beyond 17. (The A/UX operating system limits the number of processes to approximately 20 per user.) The computational complexity of the program is not related to the degree of parallelism employed, that is, the total amount of work to be done by the application is not affected by parallelism.

Figure 6.10 illustrates the performance of the two environments [Yale and VPI(i)] when prime_act.cf is tested on the RS/6000. As seen in Figure 6.9, there exists an exponential decay element to the wall time data when the number of eval()ed functions is small (less than 7). However, unlike the The Yale and VPI Environments
Primeact.cl Yale and VPI(i)
5/93 RS/6000 AIX

for \( y \geq 10 \): \( y = 0.356x + 44.4 \)

Figure 6.10  Prime_act.cl Comparison on RS/6000

The Yale and VPI Environments 128
VPI(i) implementation, Yale implementation continues to follow this trend for eval()ed functions beyond 7. There appears to be an asymptotic limit to the Yale data.

The RS/6000 VPI(i) data, on the other hand, is similar to data collected for VPI(i) on the Macintosh. Initially there is an exponential decay which is followed by a linear (proportional) increase in wall time as the number of eval()ed functions increases. This positive slope is not related to the cost of eval()ing the functions, since VPI(i) eval() costs are determined to be 9.1 msec./eval(), Section 6.8, much smaller than the 356 msec./eval()ed function in this data. This added cost is clearly due to some factor other than VPI's relatively slightly higher eval() cost.

One possibility is the effect that serialization of TS access (c.f. Section 3.5.1) might have on this application. Prime_act.cl is written so that an eval()ed function must in() an upper bound for the segment that it will examine for primes. As parallelism increases so does the number of blocked eval()ed functions waiting to in() the segment's upper bound. This is also true of the Yale implementation. However, the speculation here is that, under the Yale implementation, once TS is available, the blocked process is re-activated sooner than it is under the VPI implementation. Recall that the Yale environment uses Unix semaphores to support mutual exclusion. The re-activation of a blocked VPI process implies that: 1) the kernel has written data to the process' es socket, 2) the kernel has also given up its time slice and 3) other processes scheduled before the unblocked process have also given up their time slice.

Finally, if the RS/6000 data is analyzed by looking at the amount of work each eval()ed function is required to perform an interesting trend is seen. Figure 6.11 presents the same data by plotting the segments per eval()ed function versus the wall time required for the application to complete. The Yale data clearly indicates a linear relationship between the inverse of the number of eval()ed

The Yale and VPI Environments
Yale & VPI Segments/eval()ed Function
5/93 RS/6000 AIX

Figure 6.11 Segments per Worker versus Wall Time.
functions and wall time.

6.2.2 Cmatrix.c

The application, cmatrix.c, performs a matrix multiply in parallel by evaluating "worker" functions which operate on subsets of the larger matrix. The program allows command line specification of the matrix size and the number of evaluated workers. A more detailed description of cmatrix.c is presented in Section 4.2 where it is used as an initial comparison between the Yale and VPI implementations.

The most noticeable performance trend of cmatrix.c executing in the Macintosh environment is the exponential growth in wall time as the size of the matrix increases (Figure 6.12 [Yale], Figure 6.13 [VPI(i)] and Figure 6.14[VPI(ii)]). This is to be expected since the computational complexity of the algorithm grows exponentially with the size of the matrices being multiplied. An interesting trend is also observed in the Yale data: The wall time performance degrades as concurrency increases.

In contrast, the VPI implementation does not demonstrate a performance penalty as the number of evaluated functions increases. When the application and the VPI kernel share a single host the number of evaluated functions has no influence on performance. By comparison, VPI(ii) outperforms Yale for application configurations above 3 evaluate functions on the MacIntosh.

In the A/UX environment, VPI(ii) demonstrates even better performance than VPI(i) with this application. This is encouraging in that the ultimate objective of this research is not only to distribute the kernel but also to distribute the evaluated functions. For this application, significant speed
Cmatrix Yale Matrix Size & Worker Count
i/93 Macintosh A/UX

![Graph showing time in seconds vs. size of matrices with different worker counts](image)

**Figure 6.12** Yale Cmatrix on the Macintosh
Figure 6.13  VPI(i) Cmatrix on the Macintosh

The Yale and VPI Environments
Figure 6.14  VPI(ii) Cmatrix on Macintosh

The Yale and VPI Environments
Improvement should be realized if the eval()ed functions are not serialized on a uni-processor host.

When the application is run on the RS/6000 the Yale environment does not demonstrate the deficiencies mentioned above. Figure 6.15 illustrates the results of varying the matrix size and the worker count. As the number of eval()ed functions (parallelism) is increased the wall time performance improves. This trend is particularly noticeable when the same data is presented as "number of eval()ed workers" versus wall time, as in Figure 6.16. In this presentation we see the same inverse relationship between parallelism and performance that is noted above for prime_act.cf on the RS/6000 (Section 6.2.1): as concurrency increases there is an asymptotic lower limit for wall time.

When the VPI implementation is tested on the RS/6000 no relationship between parallelism and performance is observed. Figure 6.17 (the VPI(i) environment on the RISC processor) is very similar to Figure 6.13 (VPI(i) on the M68020). Although there is no apparent cost associated with eval()ing multiple worker functions, the overall performance compares poorly to Yale implementation on the RS/6000. On average, the VPI implementation requires five times longer to complete a matrix multiply. Assuming that a fully distributed VPI implementation provides linear speedup for this application, the VPI implementation would require five times as many RS/6000 hosts working in parallel to equal the throughput of the Yale implementation.
Figure 6.15   Yale Cmatrix on RS/6000

The Yale and VPI Environments
Yale Cmatrix Number eval()s verses Time
8/93 RS/6000

![Graph showing the relationship between the number of evaluated worker functions and time in seconds for different matrix sizes.]

**Figure 6.16** Yale Cmatrix on RS/6000

The Yale and VPI Environments
Figure 6.17  VPI Cmatrix.cl on the RS/6000
6.3 Conclusions

The test programs in this chapter present a comparison between the VPI and Yale implementations from an applications user's perspective. Both implementations have demonstrated certain advantages and weaknesses depending on the application and the run time environment. In light of the results from `prime_act.cl` and `cmatrix.cl`, it is clear that the Yale implementation outperforms the VPI implementation of the RS/6000. On the other hand, when these applications are executed in the MacIntosh environment, the VPI(i) and VPI(ii) configurations compare favorably to the Yale.

In order to improve the VPI implementation as it is hosted on the RS/6000, further testing and fine tuning is indicated. The same is also true for the Yale implementation executing on the MacIntosh. In the author's opinion, the RISC architecture of the RS/6000 gives the Yale a distinct advantage with its shared memory and callable kernel design features. It is also possible that the M68020-based MacIntosh is not fast enough to expose the added performance cost of the socket-based communications used in the VPI design. Obviously, as the workstation's raw processing speed is increased the VPI(ii) configuration will begin to show some penalty for communication over a network.

Finally, the VPI design provides a basis upon which a fully distributed Linda environment can be built. There is some indication from the data presented in this chapter that a more efficient communications medium might be considered. Other methods of improving the design are considered in chapter 7.
"Any clod can have the facts, but having opinions is an art."

-Charles McCabe, San Francisco Chronicle

7.0 Summary and Future Work

Chapters 2 and 3 of this thesis described the theoretical and design fundamentals of the VPI Linda kernel. Chapters 4, 5 and 6 presented the verification, refinement and relative performance of the kernel. This chapter summarizes the major contributions of this research and outlines areas of possible future work.

7.1 Summary of Contributions

The main contribution made by this research is the design, implementation and validation of a distributable run time kernel for the Linda programming environment. This kernel, based on the client/server model, provides concurrent tuple space access to multiple Linda processes executing on a separate host. Each process communicates with the kernel via a Unix socket, providing reliable and timely peer-to-peer communications. The overall kernel design allows experimentation with alternative communication facilities and, ultimately, utilization of multiple Linda processes executing on multiple hosts. These characteristics are integral to the ultimate objective of implementing a distributable Linda kernel. That the second phase of this research (c.f. Chapter 1 and Section 3.3)
incorporates the VPI kernel with only minor modifications demonstrates that this most important goal has been met.

Another contribution made by this research involves the side-by-side comparison of the Yale kernel and the VPI kernel in several different environments (Amiga, Apple MacIntosh and IBM R/S 6000). This comparison is important for three reasons:

- It identifies areas of relative strength and weakness.
- It identifies computer architectures which might benefit from this implementation as well as those which require more research to improve efficiency.
- It identifies applications which perform well in the Yale or the VPI environment.

One of the original objectives of this research was to provide a "baseline" kernel that could be used for further research into a variety of topics. This work has indeed contributed to ancillary research, particularly in the area of compiler optimization. Some of the research conducted by Mr. Ken Landry [LANK92, LANK93a, LANK93b] is based on the VPI kernel. Further, work by other VPI Linda researchers uses the VPI kernel as a baseline comparison for fully distributed Linda applications [ARTJ91].

In addition to providing a baseline kernel for research into other Linda topics, this investigation provides insight into implementation issues which must also be addressed in a distributed parallel environment. Specifically, the following three issues relating to fully distributing Linda have been addressed in this thesis:

- the impact of the context switch incurred each time a process blocks on a socket read,
- timely socket communications by setting appropriate socket options,
efficient application-kernel communications protocol supporting all Linda operations and data types.

Finally, this Linda kernel implementation includes many low level debugging features which would not normally be considered a contribution to the field of study. However, during the implementation and debugging of the second phase of this work (distribution of the Linda application), the debugging hooks used in the kernel and the interprocess communications subsystem of this work are reported to have been very helpful.

7.2 Future Work

Several topics germane to this effort remain to be researched. These topics address either the VPI environment as it has been developed in this thesis or the fully distributed Linda application environment. These are listed briefly here and described further in the sections below. Topics related directly to this thesis are:

- increasing the efficiency of the VPI environment, particularly on the RS/6000,
- improving the overall robustness and fault tolerance of the environment,
- supporting more than one Linda application concurrently.

Topics of a broader nature include:

- providing support for heterogeneous architectures,
- development and comparison of various methods for selecting eval() host,
- development and comparison of alternative communication subsystems.

Summary and Future Work
7.2.1 Increasing Efficiency

For large data tuples, the amount of data needing to be transferred between the kernel and the Linda application can be considerably reduced by using data compression techniques already in common use [NELM92]. This approach promises not only to improve transmission time but also to improve kernel searching speed. In cases where the data needs to be compared (as in an actual-actual field match, c.f. Section 2.2.6), the comparison could take place on the compressed data, thereby reducing the comparison time proportional to the degree of compression.

Another possible area for improving efficiency of the VPI kernel is to execute the kernel as a real time process [IBM92] or as a daemon [IBM92]. Depending on the strategy used by the scheduler, this can have the effect of reducing the time the application spends blocked on socket reads. The results in this thesis indicate that blocking on socket reads and the forced context switch implied therein is the major cause of delay in the VPI environment. Likewise, by using a mechanism similar to an interrupt handler, the kernel could service application in() operations without other processes intervening. Large wall time cost incurred when a process blocks on a socket read could be reduced considerably.

7.2.2 Improving Robustness and Fault Tolerance

In any useful application environment, some facility must be provided for gracefully handling run time errors such as low memory, process failure, and math errors. Neither the Yale nor the VPI environments handle these types of errors gracefully. Some form of *rewind* ability, backing the

Summary and Future Work
application up to a known state, needs to be investigated. In a fully distributed application environment it should be possible to migrate the kernel or a Linda process from a failing host to a healthy host if conditions indicate. This type of action might seem extreme, but the only alternative is to restart the entire application.

7.2.3 Selection of eval() Hosts

The topic of workstation load balancing has been and continues to be studied by the computer science community. Numerous strategies have been implemented and tested [GRARG93, AGRAA85, STUMM88, WANGY85]. In the context of a Linda environment the ability to select target hosts for eval()ed functions could greatly improve efficiency and throughput. Factors to consider during selection would include a host's CPU loading, network efficiency, and raw computing horsepower. For example, it is inadvisable to add additional load to a host having already high CPU demands. Selection might also take into account the host's ability to read and write to other nodes on the network efficiently. Finally, workstations with specialized hardware, such as a vector processor, might be particularly advantageous in some applications, such as cmatrix.cl, and under utilized in others.
7.2.4 Support for Multiple Linda Applications

The present implementation supports only one Linda application executing concurrently. In order to provide kernel services in a multi-application environment the kernel requires the following two features:

- multiple tuple spaces, one for each application, and
- capacity for many more open file (socket) handles than is the usual maximum for a user process.

This ability could be easily added by modifying the Linda run time startup library and the kernel so as to fork a single copy of the kernel for each application in the system. In a larger multi-workstation environment this capability could be very useful, particularly if the forked kernels are dispatched to execute on separate hosts. Selection of kernel hosts could also employ a strategy similar to that discussed in Section 7.2.3.

7.2.5 Support for Heterogeneous Architectures

To take full advantage of idle CPU cycles and simultaneously increase the computational power of a Linda environment, as many systems as possible should be made available as Linda hosts. This immediately raises the question of how to transfer data to and from hosts which do not share a common binary format for scalar data types. For example, differing floating point types would prevent host A from using a tuple from host B that contained a floating point number. Also, because Linda tuples can be composed of aggregate types such as unions and structures, the problem is more complicated than floating point mismatch. At least one interhost transmission standard as been
proposed [S7ED90], which provides the needed functionality. Incorporating such a protocol into the communications subsystem developed for the VPI kernel would allow the inclusion of many more host workstations in the fully distributed environment.

7.2.6 Development of Alternative Communication Subsystems

Unix sockets, the interprocess communications median chosen for the VPI kernel, has several primary benefits: portability, guaranteed deliverability and high level language support. Other methods of interprocess communication which are available in a networked environment might prove to be more efficient. This is of particular interest when the TS resides on the same node as an eval()ed process, that is, communications are interhost. In such cases it would be advantageous to use a specialized communication medium, such as shared memory, for processes which share a host.
References and Bibliography


References and Bibliography


References and Bibliography 150


References and Bibliography


References and Bibliography


References and Bibliography

153


<table>
<thead>
<tr>
<th>Reference</th>
<th>Citation</th>
</tr>
</thead>
</table>
Vita

Charles N. Schumann, Jr., was born on December 31, 1956, in Richmond, Virginia, where he lived until the age of 12. He graduated from Brandon High School, Brandon, Florida, in 1975 and attended Florida Bible College the following academic year. In the spring of 1977 he entered Virginia Tech as a transfer student. Before graduating from Virginia Tech in 1981, he spent a year abroad studying in Taiwan at National Taiwan Normal University’s Mandarin Institute. Since graduation he has spent 10 years at the Radford arsenal involved in numerous factory and laboratory automation projects. He has also spent two additional years in Taiwan working for an American engineering and OEM company.

He began taking courses on a part-time basis in the Virginia Tech Computer Science Program the 1986-87 Winter quarter. His Master’s thesis, "Distribution of Linda Across a Network of Workstations," was completed the 1993-94 Fall semester.

He is a member of the IEEE Computer Society and his research interest includes real time systems, embedded systems and distributed processing.

Charles N. Schumann, Jr.

Vita