Instructional Footprinting: A Basis for Exploiting Concurrency Through Instructional Decomposition and Code Motion

by

Kenneth D. Landry

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

Doctorate of Philosophy

in

Computer Science

Approved:

James D. Arthur, Chairman

Calvin Ribbens

Marc Abrams

Dennis Kafura

Adrienne Bloss

December, 1993
Blacksburg, Virginia

Instructional Footprinting: A Basis for Exploiting Concurrency Through Instructional Decomposition and Code Motion

by

Kenneth D. Landry

Committee Chairman: James D. Arthur
Computer Science

(Abstract)

In many languages, the programmer is provided the capability of communicating through the use of function calls with other, separate, independent processes. This capability can be as simple as a service request made to the operating system or as advanced as Tuple Space operations specific to a Linda programming system. The problem with such calls, however, is that they block while waiting for data or information to be returned. This synchronous nature and lack of concurrency can be avoided by initiating a non-blocking request for data earlier in the code and retrieving the returned data later when it is needed. To facilitate a better understanding of how this type of concurrency can be exploited, we introduce an instructional footprint model and application framework that formally describes instructional decomposition and code motion activities. To demonstrate the effectiveness of such an approach, we apply instructional footprinting to programs using the Linda coordination language. *Linda Primitive Transposition* (LPT) and *Instruction Piggybacking* are discussed as techniques to increase the size of instructional footprints, and thereby improve the performance of Linda programs. We also present the concept of *Lexical Proximity* to demonstrate how the overlapping of footprints contributes to the speedup of Linda programs.

Dedicated to my happy thought, my son Joshua.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my major advisor, Dr. James D. Arthur, for his dedication, wordsmithing, and patience over the past 5½ years. Without his persistence, patience, and letters to GPRAC, I probably would still be working on my bibliography.

My thanks also go to my committee members, Adrienne Bloss, Dennis Kafura, Marc Abrams, and Cal Ribbens. Their insight and probing questions have significantly improved the quality of this research effort.

I also thank my co-workers at Management Systems Laboratories for enduring me over the past 2 years and for not reminding me how many times I said it will be just a few more months.

Finally, I would like to thank my wife, Jennifer, and my son, Joshua, for sacrificing much over the past several years for the sake of my degree. I thank you and I love you.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER 1 - Introduction

1.1 Problem Statement

Many languages offer the capability to communicate with a separate, independent process to perform a service or a computation. The research presented in this report focuses on calls to independent functions. An independent function is defined to be one that is callable from the current process and produces no side-effects except through its parameters. I/O calls to an operating system and remote procedure calls are examples of calls to independent functions. This research is interested in independent function calls that are synchronous in nature, causing the calling process to block while waiting for data to be returned. The independent function performs the computation for the requested service. The inherent characteristic we are attempting to minimize is the lost computing time the calling process spends blocked waiting for return data. The proposed solution is to provide the capability to parallelize computations of the independent function with normal computations of the calling process. In other words, the goal is to transform synchronous service calls into asynchronous ones.

For example, suppose a program makes a request to the operating system to retrieve a record from a file. The program initiates the request by passing to the independent function (in this case the operating system) the necessary information to retrieve the record from the file. At this point, the program blocks awaiting the returned record from the operating system's file i/o service routine. Three activities occur while the program is blocked - 1) the requested information is transferred to the operating system, 2) the service is performed, and 3) the record is returned to the program. Once the record is returned, the program requesting the record can continue processing.

One way to exploit concurrency in this scenario is to recognize the presence of the independent function calls in a program and automatically initiate the data request for the

call earlier in the code and get the returned data later in the code at the time it is needed. This transformation of synchronous function calls to asynchronous ones can be achieved automatically without the programmer being aware that it is happening. We define the span of code from the point where the initiation of the data request is made for an independent function call to the point where the return data is received to be the *footprint* of the function call. This research is concerned with determining the footprints of independent function calls (instructional footprinting) and, for one application domain, the appropriate mechanisms for initiating the data request and receiving the return data for the function call. The result of optimizing programs with instructional footprinting is execution speedup. This speedup, obtained by converting synchronous function calls to asynchronous ones, is directly affected by the *quantity* and *quality* of a footprint. The quantity of a footprint refers to the number of instructions of which it is comprised. The quality of a footprint refers to the speedup that can be attributed to each of the instructions in the footprint. In order to achieve speedup, both the quantity and the quality of footprints need to be examined with the goal of maximizing each.

As with any research effort, there are issues that need to be addressed and questions that need to be answered. In particular, the question of program equivalence must be examined because program transformations are being made in order to convert synchronous function calls into asynchronous ones. This issue has been addressed in similar research efforts through control and data dependency analysis [BANER76 and WOLFE90]. Likewise, for independent function calls, some form of dependency analysis must be formulated so that the corresponding instructional footprint can be determined. Because of our chosen application domain, we are assuming that the only possible side-effects of an independent function call are through reference parameters and the return value.

In performing data dependence analysis, the question of how subscripted variables, structured variables and pointer variables are handled needs to be addressed. In addition to these *intraprocedural* issues, side-effects and aliasing introduced by procedure calls (*interprocedural* analysis) must be considered. Finally, the question of how to handle GOTOs must be addressed. Each of these issues is examined in detail in this report.

In addition to addressing issues that deal with the process of footprint determination, we need to address issues such as maximizing the quantity and quality of instructional footprints. Techniques aiding the footprinting process are developed in an effort to help researchers and programmers maximize program speedup through optimizing footprint quantity and quality.

1.2 Motivation

This research effort is motivated by the need to improve performance in Linda[1] programs. Linda is a *coordination* language [CARRI89b, GELER92 and ZENIT90] that provides primitives to create processes, as well as to coordinate their communication. Because Linda is a coordination language, Linda primitives can be introduced into many base computational languages. Linda has been embedded in a wide variety of languages -- C++, Fortran, various Lisps, PostScript, Joyce, Modula-2, and soon Ada [BORRM88, CARRI90, CARRI92, GELER90, JELLI90, and LEICH89]. In addition, Linda has been

---

[1] Linda is the product of a research project conducted by Gelernter and Carriero at Yale back in the mid 80's in an effort to design a coordination language for parallel programming that is conceptually simple and both architecture and language independent.

implemented on a wide variety of architecture platforms -- workstations such as Sun, DEC, Apple Mac II and Commodore AMIGA 3000UX, as well as a network of DEC VAX machines [ARTHU91]. Linda has also been ported to parallel machines such as the Sequent, S/Net and the iPSC/2 Hypercube [BJORN89a, BJORN89b, BJORN92, CARRI86a, CARRI86b, CARRI87, CARRI92, LEICH89, and LUCCO86] including a Linda machine currently being built [KRISH87 and KRISH88]. Many "real world" applications have been written using Linda; some of these are described in [ASHCR89 and CARRI88].

The Linda approach supports process creation and intercommunication through a shared data/process repository called Tuple Space (TS) [CARRI87, CARRI89a, CHIBA92, GELER85a, GELER85b, KAMBH91, and PATTE93]. Linda provides operations to generate data tuples (**OUT**), to read data tuples (**RD**), and to remove them from TS (**IN**). TS contains not only data tuples but also process tuples (created with the **eval** operation), which are often called "live tuples." A process tuple is instantiated and is eventually replaced by a data tuple when the instantiated process finishes executing. TS can also be used to share data structures among processes and synchronize the order of actions that processes perform. Refer to Appendix C for more information on Linda primitives.

Several implementations of Linda use a separate process (i.e. an independent function) in controlling TS [CARRI87 and SCHUM91]. In particular, network versions of Linda are often implemented with independent processes controlling TS [CARRI87, SCHUM91 and WHITE88]. In the case when TS is managed by a separate process and when an **IN** is performed to retrieve a tuple from TS, the process initiating the **IN** must block until a tuple is returned. This waiting time includes the time it takes to find the tuple requested,

as well as the time it takes to transfer information to and from the TS managing process. Moreover, the requested tuple may not be present in TS, in which case the TS manager will process other pending requests, while occasionally checking for a matching tuple for the blocked Linda process. Meanwhile, the calling process is blocked the entire time the TS manager is looking for a matching tuple to arrive. This "wall" time may be accentuated when Linda programs are placed on a LAN platform because of the communications overhead of transferring tuple structures and data across a network.

It is apparent that TS is potentially a serious performance bottleneck. One solution for improving the performance of a Linda system is to parallelize the normal computation of Linda programs with the requested services of TS. This involves providing two additional language primitives to the programmer -- one to *initiate* a data request for an **IN** operation and one to *receive* the tuple data being returned. In general, this would involve extensive modifications to the Linda compiler and to the underlying run-time kernel. Such modifications would compromise the conceptual simplicity of the Linda language in order to provide certain capabilities that may not be wanted or needed by all Linda programmers. This is called the *second system effect* which is warned against by Brooks [BROOK75]. An alternative approach, and one that is transparent to the programmer, is to do the following:

1) Provide primitives for the initiation and retrieval routines for **IN** operations,
2) Automatically determine the optimally safe positions for the initiation and retrieval of an **IN**, and then
3) Place the initiation and retrieval routines at these positions.

To summarize, the focal point of this research is to aid in the transformation of synchronous calls to independent functions into asynchronous calls. Assuming the

availability of mechanisms for initiating an independent function call and for later retrieving the return data, this research investigation has two goals:

1) Develop a generalized instructional footprint model and application framework for the determination of footprints, and

2) Maximize, in the Linda domain, the resulting speedup of Linda programs by optimizing the quantity and quality of footprints.

In determining a footprint for an independent function call, the instructional footprint model concentrates on ascertaining two pieces of information:

1) The earliest point in the code at which a data request for the call can be safely initiated, and

2) The latest point at which the call's return data can be safely retrieved.

Chapter 2 provides background research relevant to this research effort. Chapter 3 presents the instructional footprint model which describes how instructions, in general, can be safely moved around in a program. Chapter 4 presents a technique that increases footprint quantity and quality by allowing initiations and receives of **IN**s and **RD**s to be safely moved past other Linda operations. Chapters 5 and 6 address issues surrounding footprint quantity and quality, respectively. These chapters describe techniques that both the Linda researcher and programmer can use to increase footprint quantity and quality. Finally, Chapter 7 offers some concluding remarks and identifies future areas of research in instructional footprinting.

CHAPTER 2 - Background

2.1 Related Work

This research effort involves the use of techniques similar to those applied in other related fields. The following sections describe consanguineous research in the areas of parallel programming, futures, lazy evaluation, remote procedure calls and code transformations such as vectorization and loop parallelization.

2.2 Tuple Pre-Fetch

Researchers at Yale have proposed [CARRI90] an optimization technique similar to the one described in this paper. Their optimization, called *tuple pre-fetch*, focuses on breaking the performance bottleneck created by a centralized TS managing process. However, at Yale the emphasis is placed on control flow and not on data flow. Their proposed research is closely related to work associated with loop parallelization. In [CARRI90], Carriero describes tuple pre-fetch:

> *When compile-time flow analysis can be established that, once some branch point is passed, a given in or rd downstream must be executed, we can initiate the in or rd early, thus minimizing the interval during which the in or rd is blocked.*

and gives the following as an example.

```
while (1) {
      in(task descriptor);
      if (the task descriptor is a "poison pill") break;
      do the task...
}
```

In this simplified example, once the `if` statement has been passed, the next task descriptor can be initiated. The one issue that is not addressed in this example is that of data dependency. Suppose the **`in`** operation retrieving the task descriptor also returned a data value, say `x`, that is used to compete the task. In order to pre-fetch the next task descriptor (and the next value for `x`), steps would have to be taken to insure that `x` does

not get overwritten with the next value for x until the current task is complete. The primary difference between tuple pre-fetch and the research on instructional footprinting described in this report is in the type of control flow analysis performed. In addition, this proposed research employs data flow analysis as well as control flow analysis.

2.3 Futures

Futures, which were first developed and implemented by Halstead [HALST85] for a Lisp variant called MultiLisp and intended for use on a multi-processor machine, provide an explicit means of parallel processing. Through the use of futures, MultiLisp can spawn concurrent computations and provide for the synchronization of these processes. For example,

```
(setq Y (future   X))
```

when evaluated, will spawn a new process to evaluate X and will immediately assign to Y a `future` (a place holder if you will) that represents the future value return by the evaluation of X. When the value of Y is referenced, the future is checked to insure that the computation of X is complete. If it is not, then the reference is delayed until the evaluation of X is complete.

Futures provide a means of parallelizing activities, but unlike instructional footprinting it only addresses half of the problem. The concurrency provided by futures begins at the function invocation whereas our approach provides for the early initiation of concurrency. In addition to Lisp, futures have been used in other languages such as C [CALLA90] and C++ [CHATT89]. Listov [LISTO88] proposes the use of a new data type called a *promise* that is designed to support asynchronous calls. Promises are similar in functionality to futures but are based upon an asynchronous communication mechanism,

the *call-stream*. Listov's work, as it relates to remote procedure calls (RPCs), is discussed further in Section 2.5.

2.4 Lazy Evaluation

Lazy evaluation [BLOSS88 and HUDAK89], which has been associated mostly with modern functional languages such as Haskell [HUDAK92] and LML [AUGUS92], is similar in some respects to futures and our research as well. With lazy evaluation, computations (such as function parameters) are not evaluated until they are needed. At first glance, computations **appear** to be finished immediately after they are started (as with futures and our research). However, no parallel computation is used with lazy evaluation to improve performance. Rather, speed is gained when, because of control flow reasons, the computations that are *lazily evaluated* (i.e. the evaluation is postponed until needed) do not need to be performed.

For example,

```
Foo  1  2  (5 * X / Y * (Bar 2))
```

is a Haskell invocation of the function `Foo`. With lazy evaluation, the third parameter would **not** be evaluated upon the invocation of `Foo`, but rather it will be postponed until the associated formal parameter is referenced inside `Foo`. At this point, `5*X/Y*(Bar 2)` is evaluated and the resulting value is bound to the formal parameter. Speedup is realized if, in the execution of `Foo`, the initial value of the associated formal parameter is never needed and therefore never evaluated.

2.5 Remote Procedure Calls

A remote procedure call (RPC) is one example of an independent function. In most cases, RPCs are synchronous, requiring the process initiating the call to block while the remote procedure executes [BIRRE84 and WEIHL89]. Listov addresses the need for asynchronous calls in [LISTO86] and proposes the use of a new data type called a *promise* to be used in conjunction with an asynchronous communication mechanism called a *call-stream* [LISTO88]. Call-streams are used to make an RPC asynchronous while a promise, as Listov describes it, can be considered a "claim ticket" for an RPC that must later be used to "claim" the returned result. However, unlike instructional footprinting, it is up to the programmer to decide when to initiate the call-stream and when it is needed to claim the return value of the called function. In our research, safe positions for initiating and retrieving return data for an independent function call are automatically determined without any guidance from the programmer.

2.6 Program Transformations

Parallelizing and vector compilers make use of code transformations to automatically exploit inherent parallelism in serially written programs. This automatic detection of parallelism and/or vector operations is beneficial because of the abundant amount of code already written for non-parallel/vector machines. Therefore, the vast computing resources of vector and parallel machines can be tapped from existing code with little or no modification.

Although program transformations to exploit parallelism have also been applied to Lisp and Prolog [BANSA89, FRADE91, LARUS88b and RAMKU89], Fortran is the primary language of choice for this type of research [POLYC90 and ZIMA90]. The two major areas of applied research for program transformations have been array vectorization

[NOBAY89, POLYC90, TSUDA90, WOLFE90 and ZIMA90] and loop parallelization [DOWLI90, EBCIO90, IWANO90, SALTZ89 and SCHWI91]. Automatic vectorization deals with identifying array operations primarily within loops and converting them, if possible, to vector operations. Loop parallelization deals with analyzing loops to identify which iterations of a loop are independent of each other. Independent iterations of loops are placed on different processors and executed in parallel. Vectorization and loop parallelization both rely on extensive control and data flow analysis in order to create vector/parallel programs from serial ones.

The determination of instructional footprints relies on the same types of control and data dependency analysis. Much work has been done to determine data dependencies between instructions [BANER76, BANER79, BURKE90, LI90 and WOLFE90] including in-depth analysis of reference patterns involving subscript and structure variables [BALAS89, BURKE86, CHASE90 and LARUS88a]. Program dependency graphs (PDGs) [BAXTE89, FERRA87, and HORWI88] are often used to represent a program and its control/data dependencies[2]. For automatic vectorization and loop parallelization, PDGs help describe the essential dependency characteristics of a program so that analysis (often involving the use of recurrence relations or dependency equations and tests) can be performed to exploit potential vectorization and/or parallelism.

Because the effect of procedure calls on data dependencies was not considered in earlier research efforts, subsequent research has been directed towards interprocedural side-

---

[2] Horwitz in [HORWI88] shows the adequacy of PDGs for representing programs by proving that "if the PDGs of two programs are isomorphic then the two programs are strongly equivalent."

effects and procedure-induced aliasing [CALLA87], all of which is important in doing complete instructional footprint analysis. In addition, other research has been performed on understanding the effect that pointers [CHASE90, HORWI89 and LANDI90] and the passing of function parameters [NEIRY87] has on data flow analysis. Because the use of GOTOs is also crucial we cite Ramshaw [RAMSH88], who states that it is possible to transform a program with GOTOs into a functionally equivalent one without GOTOs. Therefore, depending on the type of analysis needed to determine an instruction's footprint, it is often possible to ignore the effects of GOTOs.

CHAPTER 3 - Instructional Footprint Model

3.1 Introduction

The Instructional Footprint Model (**IFM**) [LANDR92b and LANDR93b], proposed in this dissertation, is a tool for analyzing the interaction of program instructions. Similar to Bernstein's conditions that are used to determine the interdependence (or independence) of processes [MAEKA87], the **IFM** can be used to ascertain the *footprint* of an instruction, how far back or forward in a program it can be moved. The footprint is primarily defined by the existence of certain dataflow dependencies. These dependencies create restrictions on instruction movement. The model can be used to analyze the *mobility* of an instruction relative to other instructions or as a compiler optimization technique to improve the performance of a program.

The model is not designed to footprint all instructions of a program at once. However, it can be applied to any individual instruction within a program to determine its footprint. Once an individual instruction has been identified, the model can be applied to determine the *heel* and *toe* of the footprint. The heel is the earliest position and the toe is the latest position in the code where the instruction can be *safely* executed. The question is, *what is safely executed.* Figure 3-1 exemplifies where the heel and toe of a given instruction can be *safely* placed for program execution.

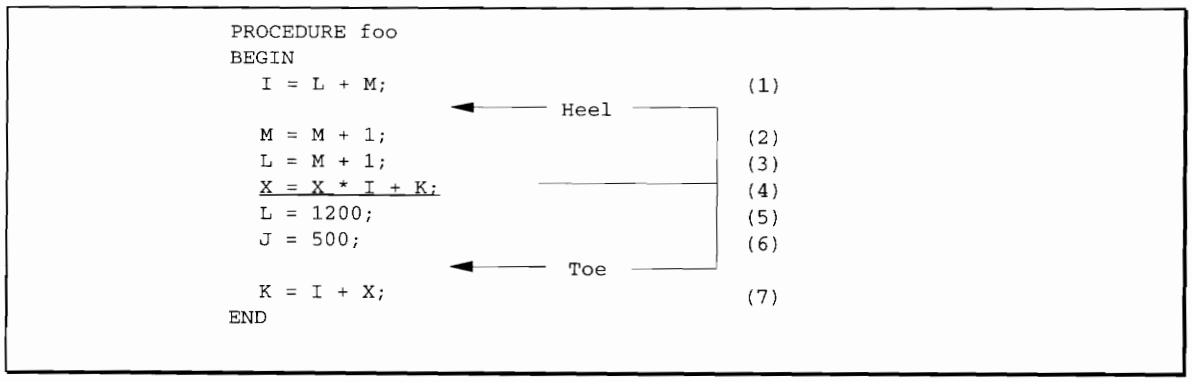


Figure 3-1. Example of an instruction footprint.

In determining the footprint of instruction 4, the earliest position that the heel can be safely placed is between instructions 1 and 2. The reason the heel cannot be placed before instruction 1 is due to the fact that variable **I** is being written to in instruction 1 and referenced in instruction 4. This creates a conflict between the two instructions, therefore instruction 4 cannot be *safely* moved past instruction 1. Similarly, the toe is positioned between instructions 6 and 7 because there is a conflict with the reading and writing of **X** between instructions 4 and 7.

The conflict of variables is one criterion for determining the safe placement of the heel and toe of instructional footprints. Another determining factor for safe placement is the identification of procedural boundaries as seen in Figure 3-1. The placement of instruction 4 obviously cannot be moved past the confines of its procedure because the semantics of the procedure would be changed. This nesting restriction applies not only to procedures but to conditional and looping constructs as well. For example, the heel and toe for an instruction within a WHILE loop cannot be placed outside of the loop because this would change the semantics of the WHILE block. These nesting restrictions help define what are called the *hard boundaries*, which are simply the outermost limits for the placement of the heel and toe. The actual positions of the heel and toe (called the *soft boundaries*) are located within the limits of the hard boundaries. This model is geared toward procedural languages; therefore procedural boundaries will always act as hard boundaries. This means that given an instruction to footprint, we only have to address the code between the `BEGIN` and `END` of the defining procedure.

For a given instruction, the first step in the footprint determination process is to define the hard boundaries for that instruction. Once the hard boundaries are in place (above and below the instruction), the process of locating the heel and toe can begin. This process

simulates the movement of the instruction to be footprinted backward in the code to find the heel and then forward to find the toe. The movement of an instruction is complicated when conditional and looping constructs are encountered.

```
PROCEDURE foo
BEGIN
      IF (I < 100)
            I = L + M;                    (1)
            M = M + 1;                    (2)
            L = M + 1;                    (3)
      ENDIF;
      X = X * I + K;                      (4)
      L = 1200;                           (5)
      J = 500;                            (6)
      K = I + X;                          (7)
END
```

Figure 3-2. Example of a complicating Conditional.

For example, in Figure 3-2 the heel of instruction 4's footprint cannot be placed between instructions 1 and 2 (where the conflict occurs) because it is within a conditional. This would change the semantics of the code. Because a conflict has been found within the conditional, the heel cannot be moved inside the `IF`. It would be helpful, and less complicating, to know if a conflict exists before entering into the `IF`. This can be accomplished through the use of *aggregate* instructions which represent a group of instructions. For example, in Figure 3-2 the entire `IF` statement can be combined into a single aggregate instruction. This means that one check, instead of many individual ones, can be made to determine if a conflict exists between the `IF` and instruction 4. Because a conflict does exist, the `IF` can be *deaggregated* into its component instructions for analysis. The information about the existing conflict can be used (before entering the `IF`) to make a decision about the placement of the heel. The `IF` is one of four aggregate

instructions addressed in this model. The others are the `WHILE`, `REPEAT`, and `BLOCK`. The specific details for handling conflicts for each of the aggregate instructions are discussed in Section 3.7.

Consider the code segments between the hard boundaries and the instruction to be footprinted. The process of determining an instruction's footprint can be simplified by first taking these code segments and aggregating them each into a single instruction. In the process of determining the heel and toe positions, deaggregation only takes place when the instruction being footprinted cannot move past the aggregate instruction due to conflicting data flow dependencies. Once the instruction is deaggregated, the process of finding the footprint's heel and toe continues. The specific details concerning the termination of the footprint determination process are described in Section 3.9.

The remainder of this chapter describes the details of the model starting with the canonical language to which the model is applied. This is followed by the description and definition of a program and an instruction. The aggregation and deaggregation functions are then detailed followed by a discussion of the hard and soft boundaries. The next two sections describe the details of the aggregation setup process and how the footprint is determined for an instruction. The last section addresses how function calls, `GOTO`s and pointers are handled both by compile-time analysis and with run-time extensions.

3.2 The CL Language

Because the **IFM** can be used to analyze the interaction of program instructions, a language is used as the basis for the model. Instead of using a specific programming language, we define a generic language for use in the **IFM**. This *canonical* language

(**CL**) contains the standard procedural programming language elements: variables, structured types, pointers, functions/procedures, statements , GOTOs, conditionals, and looping. The five statements of **CL** are the assignment, the procedure call, the IF, the WHILE and the REPEAT. The specific details of the language will not be discussed because the model, for the most part, only needs general knowledge about a program and not specific language details. Although the syntax of the blocking language constructs (i.e. IF, WHILE and REPEAT) is significant, the syntax and detailed semantics of individual **CL** instructions (i.e. expressions, assignments and procedure/function calls) are not important to the development of the model. However, knowing which variables a **CL** instruction reads and writes to is crucial.

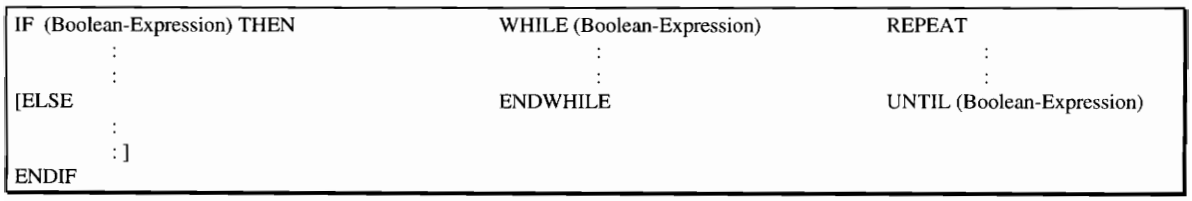


Figure 3-3. Constructs of CL.

There are three major constructs in **CL**: the IF, the WHILE, and the REPEAT. The REPEAT is a part of **CL** for simplicity and is not an essential construct. Figure 3-3 illustrates the syntax of each construct. Only the basic language constructs are present in the canonical language, making it necessary to convert from the target language being used to the canonical language in order to use the model. A conversion from **CL** back to the target language is also necessary in order to implement the results of the **IFM** analysis.

C	CL

IF (Boolean-Expression) {
  <Then-Code>
[} ELSE {
  <Else-Code>]
}
	IF (Boolean-Expression) THEN
	  <Then-Code>
	[ELSE
	  <Else-Code>]
	ENDIF

SWITCH (Expression) {
 	
  CASE <Item1> : {<Statements1>}
  :
  :
  :
  CASE <ItemN> : {<StatementsN>}
}
 	
If BREAK appears in <Statements-I> then
 	
  CASE <ItemI> : {<Statements-I>}
	IF (Expression == <Item1>)
	THEN
	  <Statements1>
	ENDIF
	  :
	  :
	IF (Expression == <ItemN>)
	THEN
	  <StatementsN>
	ENDIF
	
	IF (Expression == <ItemI>)
	THEN
	  <StatementsI>
	ELSE
	  IF (Expression == <ItemI+1>
	  THEN
	  <StatementsI+1>
	  ENDIF
	  :
	  :
	ENDIF

WHILE (Boolean-Expression){
  <Statements>
}
	WHILE (Boolean-Expression)
	  <Statements>
	ENDWHILE

DO { <Statements> } UNTIL (Boolean-Expression)	REPEAT <Statements> UNTIL (Boolean-Expression)

FOR (Stmt1; Stmt2; Stmt3) {
  <Body-Statements>
}
	Stmt1;
	WHILE (Stmt2)
	  <Body-Statements>
	  Stmt3;
	ENDWHILE;

Figure 3-4. Construct transformations from C to CL.

Figure 3-4 illustrates the conversion to and from the canonical language using the programming language **C**. The constructs of interest in **C** are the `IF`, `SWITCH`, `WHILE`, `DO..WHILE` and the `FOR`. A transformation of **C** constructs to the **CL** language is necessary for the model to be applied. A transformation back to **C** from the canonical language is also necessary to identify where the heel and toe of the instruction footprint are actually located. The **IFM** does not require the use of an actual programming language (such as **C**), but rather one can reason about the safety of code motion using CL and have no concern for the actual implementation language.

3.3 A CL Program

A **CL** program is a sequence of instructions where each instruction has two important attributes - computation and control. The questions are 1) *what is considered an instruction* and 2) *how are the concepts of computation and control captured in the **IFM**.* The answer to the first question is analogous to what constitutes a line of code in a program. For instance, one person may tally lines of code by counting semicolons while another may count the number of statements. In either case, what is considered a line of code is determined by the person performing the analysis. Similarly, the pieces of a program important to this model are considered instructions. Figure 3-5 shows examples of instructions (represented as nodes) as they relate to the `BLOCK`, `IF`, `WHILE` and `REPEAT` statements in **CL**.

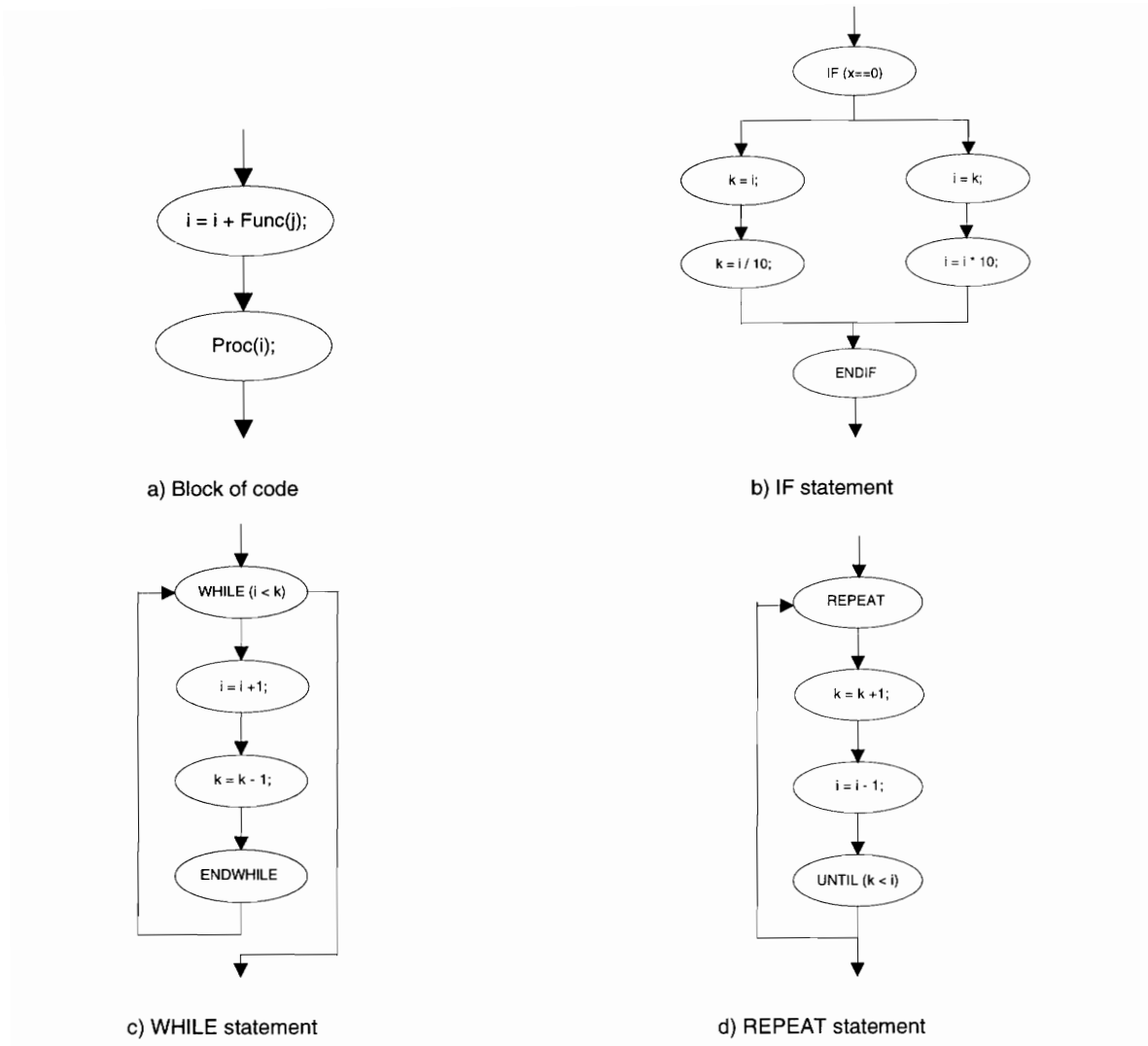


Figure 3-5. Sample instruction graphs of code segments.

As seen in Figure 3-5a, assignment and procedure calls are considered instructions in **CL**. The conditional parts of the `IF`, `WHILE` and `REPEAT` are also regarded as instructions. Although the `ENDIF`, `ENDWHILE` and the `REPEAT` do not perform any computation, they do provide flow of control for their constructs. They are viewed as distinct instructions in **CL** because they provide flow of control and are useful in the formulation of other parts of the model such as the aggregate and deaggregate functions. Notice that

the THEN and ELSE keywords have not been represented in the IF graph of Figure 3-5b. The reason is that the distinction between the THEN and the ELSE parts is not important to the development or use of the **IFM**. Therefore, they are **not** considered instructions.

The answer to the question *How are the concepts of computation and control captured in this model?* is reflected in the definition of a **program**. A program, in formal terms, is a pair (**I**, **C**) in which **I** and **C** are unordered sets representing computation and control respectively. Recall that when an instruction is being footprinted, we are only concerned with the program segment between the BEGIN and END of the defining procedure. This means that for each instruction to be footprinted, **I** and **C** can be defined to focus only on the program segment between the procedure's BEGIN and END. For a program (segment) **P**, the following defines **I** and **C**:

- **I** - An unordered set containing unique designators for instructions in **P**.
- **C** - An unordered set of control flow pairs, (**i j**), such that statement **j** can succeed statement **i** in execution. (These control flow pairs describe the entire flow of control for **P**.)

To clarify the roles that **I** and **C** play in the definition of a program, consider the following example.

```
PROCEDURE bar
BEGIN
      I = 2;                          (1)
      WHILE (I > 0)                   (2)
        I = (I-1) + (I-2);            (3)
      ENDWHILE;                       (4)
      K = I;                          (5)
END
```

Figure 3-6. Example procedure for the construction of I and C.

The set **I** would be { 1 2 3 4 5 } representing actual statements and the set **C** would be { (1 2) (2 3) (2 5) (3 4) (4 2) } representing control flow possibilities between statements. The control flow pairs of **C** capture the flow of control for the procedure. The pairs (2 3) and (2 5) represent the flow entering and exiting the loop. The pair (4 2) represents the looping back to test the conditional of the WHILE.

For a given program **P**, the set **I** is constructed by including each component of program **P** that is considered an instruction. Recall that all assignment statements and procedure calls are considered instructions, as are the conditional parts of the `IF`, `WHILE` and `REPEAT` and the `ENDIF`, `ENDWHILE` and `REPEAT`. The only components of a program between the `BEGIN` and `END` not considered instructions are the `THEN` and `ELSE` keywords.

In general, the set **C** is constructed by taking each instruction (assignments and procedure calls), say **i**, and adding to **C** the control flow pair (**i j**) where **j** is the designator for the instruction immediately following **i** in lexographical order. This construction changes when `IF`s, `WHILE`s and `REPEAT`s are involved. The previous example illustrates how the control flow pairs are constructed for a `WHILE`. Essentially, the instruction representing the `WHILE` condition produces two control flow pairs - one to stand for the flow from the `WHILE` condition to the first instruction in the body and the other pair to represent the flow to the instruction immediately following the `ENDWHILE`. In addition, a control flow pair is added to represent the flow from the `ENDWHILE` to the `WHILE` part. The construction of control flow pairs is similar for the `REPEAT`. One pair is added for the flow from the `REPEAT` instruction to the first instruction in the body. Two more are added for the flow from the `UNTIL` instruction to the `REPEAT` (for looping) and for the flow from the `UNTIL` to the following instruction (exiting the `REPEAT` loop). For the `IF`

statement, two control flow pairs are added for the flow from the `IF` condition instruction to the first instruction of the `THEN` and `ELSE` parts. Two more pairs are added from the last instruction of the `THEN` and `ELSE` parts to the `ENDIF` instruction. Finally, one pair is added from the `ENDIF` to the following instruction.

Two accessory functions, τ and ϕ, facilitate the process of determining footprints. Given an instruction designator **i** and the set **C** , these functions return a set of instruction designators. For τ (the *TO* function), the set returned indicates the instructions that can immediately follow **i** in execution. The *FROM* function, ϕ, returns a set of designators for instructions that can immediately precede **i** in execution. The following are the formal definitions of τ and ϕ.

$$\tau(\mathbf{i}, \mathbf{C}) = \{ \mathbf{j} \mid (\mathbf{i}\ \mathbf{j}) \text{ is an element of } \mathbf{C} \}$$

$$\phi(\mathbf{i}, \mathbf{C}) = \{ \mathbf{j} \mid (\mathbf{j}\ \mathbf{i}) \text{ is an element of } \mathbf{C} \}$$

Consider the example procedure in Figure 3-6. In applying τ and ϕ to instructions 2 and 5 respectively, $\tau(2,\mathbf{C})$ returns {3, 5} and $\phi(5,\mathbf{C})$ returns {2}.

3.4 A CL Instruction

Recall that in Section 3.3, the term *instruction* refers to parts of a program that play an *important* role in characterizing the **IFM** (see Figure 3-5 for examples). In the same spirit, the attributes associated with instructions must be related to the model. In particular, we must address the flow of data (the input and output of the instruction) and the concept of aggregation as they pertain to individual instructions and the **IFM**. An instruction in the **IFM** can be described using three attributes - the *read* set, the *write* set and the *component instruction* set. With respect to the flow of data, an instruction can be considered a black box performing a computation using some input (the read set) and

producing some output (the write set). While the specifics of the mapping are not necessarily important, the resulting read and write sets are. The concept of aggregate instructions is incorporated into the model through the use of the component instruction set. In other words, the component instruction set defines the segment of code an aggregate instruction represents. More intuitively, an aggregate instruction represents a collection of instructions (any of which can also be an aggregate instruction) whose unique designators are elements of the component instruction set. These three sets are defined as follows:

π_i - The *component instruction set* is an ordered set of instruction designators representing a segment of code.

In the **IFM**, we find four aggregate instructions. The following describes the format of π for each of the four aggregate instructions.

`BLOCK` - For the `BLOCK` aggregate instruction, π_{BLOCK} contains a set of individual instruction designators that are being considered a single instruction.

`IF` - For the `IF` instruction, π_{IF} contains four instruction designators. The first instruction designator represents the `IF` condition. This is followed by designators two for single instructions (which can be aggregate instructions themselves) representing the `THEN` and `ELSE` parts. The last instruction designator is for the `ENDIF`.

`WHILE` - For the `WHILE` instruction, π_{WHILE} contains three instruction designators. The first designator represents the `WHILE` condition. This is followed by a designator for a single (possibly

aggregate) instruction representing the WHILE body. The last designator is for the ENDWHILE.

REPEAT - For the REPEAT instruction, π_{REPEAT} contains three instruction designators. The first designator represents the REPEAT. This is followed by a designator for a single (possibly aggregate) instruction representing the REPEAT body. The last designator is for the UNTIL condition.

Notice that for the IF instruction, π_{IF} requires that the THEN and ELSE blocks of code be single instructions. Therefore, if the THEN and ELSE are not single instructions, they must first be aggregated into block instructions before the IF instruction can be aggregated. This is also true of looping bodies for the WHILE and the REPEAT. These restrictions allow the aggregation and deaggregation functions to be defined in an uncomplicated fashion. For non-aggregate instructions, π_i is defined to be the single element set { **d** }, where **d** is the unique designator for instruction **i**.

3.5 Aggregation Function

Before the footprint of an instruction can be determined, the segments of code between the instruction to be footprinted and the hard boundaries need to be aggregated into a single instruction. The underlying motivation for aggregation, when searching for the soft boundaries, is to avoid repeated complex and costly analysis of IFs, WHILEs and REPEATs. This can be achieved by aggregating sets of instructions and their associated attributes into aggregate instructions and then checking the aggregated attributes as a single entity for data flow dependency conflicts. Effectively, program segments can be collapsed into aggregate instructions.

The aggregation function, `aggr()`, takes as input a set of instruction designators **S**, as well as the sets **I** and **C**, which represent the program containing the instructions represented in **S**. The contents of **I** and **C** are modified to reflect the collapse of the set **S** into a single aggregate instruction.

```
PROCEDURE foobar                                    PROCEDURE foobar
BEGIN                                               BEGIN
--- Hard Boundary ---                               --- Hard Boundary ---
  I = K;                          (1)                 J = K;                (1)
  IF (I=0)                        (2)                 IfInst;               (2.5)
  THEN                                                --- Hard Boundary ---
     I = 1;                       (3)                 J = I;                (6)
  ELSE                                                L = K * J;            (7)
     K = I;                       (4)                 --- Hard Boundary ---
  ENDIF                           (5)                 END
  J = I;                          (6)
  L = K * J;                      (7)
--- Hard Boundary ---
END
```

In the above example, **I** is defined to be { 1 2 3 4 5 6 7 } and **C** is { (1 2) (2 3) (2 4) (3 5) (4 5) (5 6) (6 7) }. The aggregation of the IF statement involves collapsing instructions 2 through 5 into a single instruction. The first step is to create the aggregate instruction, call it 2.5, and define each of its attributes - the read, write and component instruction sets -- as if it were a single instruction in its own right.

IF Instruction (2.5):

$$\rho_{2.5} = \rho_2 \cup \rho_3 \cup \rho_4$$
$$\omega_{2.5} = \omega_2 \cup \omega_3 \cup \omega_4$$
$$\pi_{2.5} = \{ 2\ 3\ 4\ 5 \}$$

Now, the sets **I** and **C** need to be modified to reflect the creation of the aggregate instruction and the fact that it is replacing instructions 2 through 5. The designator for instruction 2.5 must be added to **I**, while the set of instruction designators in $\pi_{2.5}$ needs to be removed. For **C**, the set of control flow pairs { (2 3) (2 4) (3 5) (4 5) }

representing the structure of the `IF` need to be removed, and the pairs { (1 2) (5 6) } must be replaced with { (1 2.5) (2.5 6) }.

The following definitions formally characterize the `aggr()` function. The definition is followed by restrictions on the use of `aggr()` for each of the four types of aggregate instructions.

DEFINITION:

`aggr(` **S, I, C** `)` In defining and describing the `aggr()` function, **S** is the set of designators representing the sequence of instructions to be aggregated. **F** refers to the first instruction, and **L** refers to the last instruction in the set **S**. **AI** refers to the BLOCK aggregate instruction.

ρ_{AI} = union of all ρ_i where **i** is any instruction designator in the set **S**

ω_{AI} = union of all ω_i where **i** is any instruction designator in the set **S**

π_{AI} = **S**

I = **I** + { AI } - π_{AI}
For all pairs (**x** F) in **C** such that **x** belongs to **I**
C = **C** + { (**x** AI) }
case 1: aggregation is for a `WHILE`
For all pairs (F **x**) in **C** such that **x** belongs to **I**
C = **C** + { (AI **x**) }
case 2: aggregation is **not** for a `WHILE`
For all pairs (L **x**) in **C** such that **x** belongs to **I**
C = **C** + { (AI **x**) }
C = **C** - { all pairs of the form (**x i**) or (**i x**) in **C**
such that **x** belongs to **I** and **i** belongs to **S**}

RESTRICTIONS:

`BLOCK` The input set **S** is a set of two or more instructions designators. In addition, S must represent a sequence of either aggregate, assignment or procedure call instructions.

`IF`	The input set **S** is a set of four instruction designators. The first is the instruction designator for the `IF` condition. This is followed by designators for two single (possibly aggregate) instructions representing the `THEN` and `ELSE` parts. The last instruction designator in **S** represents the `ENDIF`.
`WHILE`	The input set **S** is a set of three instruction designators. The first is the instruction designator for the `WHILE` condition. This is followed by a designator for a single (possibly aggregate) instruction representing the `WHILE` body. The last instruction designator in **S** represents the `ENDWHILE`.
`REPEAT`	The input set **S** is a set of three instruction designators. The first is the instruction designator for the `REPEAT`. This is followed by a designator for a single (possibly aggregate) instruction representing the `REPEAT` body. The last instruction designator in **S** represents the `UNTIL` condition.

3.6 Deaggregation Function

In the process of determining an instruction's footprint, aggregate instructions that have data flow conflicts with the instruction being footprinted may need to be deaggregated in order to process the individual instructions in more detail. Deaggregation involves the modification of **I** and **C** (representing the program containing the aggregate instruction) to reflect the replacement of the aggregate instruction with its component instructions. The following figure shows procedure `bar` with the `WHILE` instruction aggregated.

```
BEGIN                                   C = { (1  2.4)  (2.4  5) }
--- Hard Boundary ---
      I = 2;                    (1)     ρ2.4 = ρ2 + ρ3  = { I }
      WhileInst;                (2.4)   ω2.4 = ω2 + ω3 = { I }
      K = I;                    (5)     π2.4 = { 2 3 4 }
--- Hard Boundary ---
END
```

Figure 3-7. Example of the aggregation of the WHILE from Figure 3-6.

Because the `WhileInst` conflicts with the instruction to be footprinted (variable **I**), the `WhileInst` needs to be deaggregated. Applying `deaggr( WhileInst,` **I, C**) results in instruction 2.4 being replaced with instructions 2, 3 and 4. In addition, the control flow pairs involving 2.4 need to be replaced with those for the `WHILE` structure. This results in **I**={ 1 2 3 4 5 } and **C**={ (1 2) (2 3) (2 5) (3 4) (4 2) }.

Given an aggregate instruction **i** in a program represented by **I** and **C**, deaggr() can be formally defined as follows:

`deaggr(` **i, I, C**)

$\mathbf{C_i}$ refers to the set of control flow pairs for the component instructions represented in π_i of aggregate instruction **i**. **F** refers to the first instruction in π_i and **L** refers to the last instruction in π_i

$\mathbf{I} = \mathbf{I} - \{ i \} + \pi_i$
for all pairs (**i x**) where **x** is an element of **I**
$\mathbf{C} = \mathbf{C} - (\mathbf{i}\ \mathbf{x}) + \mathbf{C_i}$
also,

- for all pairs (**x i**) where **x** is an element of **I**
  $\mathbf{C} = \mathbf{C} + \{ (\mathbf{x}\ F) \}$
- case 1: **i** is a `WHILE`
  for all pairs (**x i**) where **x** is an element of **I**
  $\mathbf{C} = \mathbf{C} + \{ (F\ \mathbf{x}) \}$

· case 2: **i** is **not** a `WHILE`
for all pairs (**x** **i**) where **x** is an element of **I**
C = **C** + { (L **x**) }

3.7 Hard and Soft Boundaries

In searching for the footprint of an instruction, the placement of the heel and toe is limited by certain boundaries. Most notably are the boundaries imposed by functions. The footprint of an instruction must remain within the `BEGIN-END` boundaries of the defining function. These are called hard boundaries and are imposed by the language constructs that represent block-level abstractions. For the previously defined canonical language, no footprint can cross a functional, conditional or looping boundary. That is, like the `BEGIN-END` boundaries of a function definition, if the instruction being moved resides within a loop or a conditional statement, then its heel and toe must remain within that construct. Other hard boundary limitations may be applied depending on the target language being used.

The soft boundary positions, i.e. those that define the heel and toe of an instruction's footprint, are determined primarily by variable contentions between instructions. Moving an instruction backward (or forward) in a program reduces to the problem of successively swapping that instruction with its predecessor (or successor). In order to safely swap two instructions, **i** and **j**, the read/write sets of one instruction cannot conflict with the write set of the other. The following theorem describes the restrictions for swapping two instructions.

Theorem 1. Two instructions **i** and **j** can be positionally swapped if for every variable **x** that is an element of $\rho_i \cup \omega_i$, **x** is not an element of ω_j. Similarly, for every variable **x** that is an element of $\rho_j \cup \omega_j$, **x** is not an element of ω_i.

In addition to satisfying Theorem 1, other restrictions must be observed when determining a soft boundary involving IFs, WHILEs or REPEATs. These constructs alter the normal sequential flow of control for a program and need to be addressed in the determination of the soft boundaries.

Suppose, for example, that the instruction being footprinted conflicts (by use of Theorem 1) with an instruction in an IF. The conflict may be with the THEN part, the ELSE part or both. Because either path might be executed, the difficulty is not knowing (at compile-time) which part is to be executed at run-time. A solution is to assume that both the THEN and the ELSE parts are to be executed and then place the soft boundary accordingly. In other words, propagate the instruction being footprinted through the code for both the THEN and ELSE parts. This results in a split position for the soft boundary. For example, Figure 3-8 shows a procedure called foobar in which the footprint of instruction 6 is being determined. Because instruction 6 conflicts with instruction 3, the soft boundary for the heel would be placed within the IF. In particular, the heel for instruction 6 can be placed after instruction 3 in the THEN part and before instruction 4 in the ELSE part.

```
Procedure foobar
BEGIN
---- Hard Boundary ----
      I = K;                                    (1)
      IF (I=0)                                  (2)
      THEN
            I = 1;                              (3)
            --- Soft Boundary (Heel) ---
      ELSE
            --- Soft Boundary (Heel) ---
            K = I;                              (4)
      ENDIF                                     (5)
      J = I;                                    (6)
      L = K * J;                                (7)
---- Hard Boundary ----
END
```

Figure 3-8. Example of a soft boundary placed inside a conditional.

Suppose now that an instruction to be footprinted conflicts (by use of Theorem 1) with an instruction within a WHILE or REPEAT (see Figure 3-7). Instruction 5 conflicts with the WHILE instruction and therefore cannot be moved past it. Instruction 5 cannot be placed within the WHILE statement because the body of the loop is not guaranteed to execute exactly once, which is an implicitly required condition for instruction 5. The soft boundary for the heel, therefore, must be placed (or remain) after the WHILE instruction.

In summary, a soft boundary (heel or toe) for an instruction cannot be placed within a conflicting WHILE or REPEAT loop. In addition, if an instruction being footprinted conflicts with an IF statement, then the corresponding soft boundary must be placed within both the THEN and the ELSE parts.

Altogether, conditions related to soft and hard boundaries define the entire set of movement restrictions for an instruction. The soft boundary will never be outside the hard boundary and will always define either the heel or the toe of an instruction (depending on the direction the instruction is being moved). Figure 3-9 shows the relative positions of the hard boundaries, the soft boundaries, and the instruction being footprinted (the box indicates the instruction footprint). Effectively, hard and soft boundaries exist on both sides of the instruction being footprinted, and thereby restrict the size of its footprint.

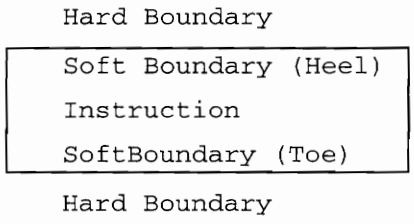


Figure 3-9. Relative position of an instruction to its boundaries.

3.8 Aggregation Setup Process

Given an instruction to be footprinted, there exists two segments of code between the hard boundaries and the instruction to be footprinted. The soft boundaries will be placed within these two segments. Before the heel and toe of an instruction's footprint can be determined, nonetheless, it is expeditious to convert all `IF`s, `WHILE`s and `REPEAT`s into aggregate instructions. In addition to the compound statements mentioned above, blocks of code can and should be aggregated into single instructions. As stated previously, the reason for this aggregation is to simplify the footprint determination process. Dealing with compound instructions as a single unit is conceptually simpler than working with instructions on an individual basis. The goal of the aggregation setup is to take each of the program segments between the hard boundaries and the instruction to be footprinted and perform successive aggregations until the segments of code are single aggregate instructions as depicted below.

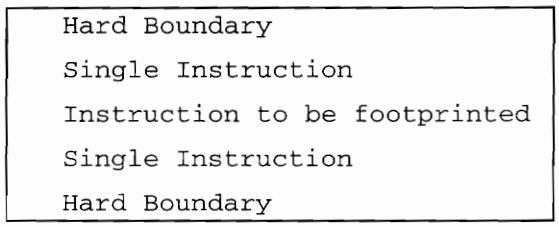


Figure 3-10. Aggregation of code segments into single instructions.

The *Aggregation Setup Process* is the prepatory step to determining the heel and toe (the soft boundaries) of an instruction's footprint. Due to the restrictions on the aggregation function (i.e. the bodies of code in the `IF`, `WHILE` and `REPEAT` must be single, possibly aggregate, instructions) the order in which the sequence of instructions are aggregated is

critical. Given the set of instructions shown in Figure 3-11, the aggregation order of the setup process proceeds as shown.

```
PROCEDURE ordering
BEGIN
----- Hard Boundary -----
      X = Z;-----------------------------------------+
      IF ( X < 100 )-------------------------------+  |
      THEN                                         |  |
            X = (X-1) + (X-2);---------------+     |  |
            Z = Z + 1;                       |     |  |
            WHILE (Y < X - Z)----------+     |     |  |
                  A = A * B;-----+     |     |     |  |
                  C = C * A;     (1)   (2)   (3)   (5)  (6)
                  Z = Z + 1;-----+     |     |     |  |
            ENDWHILE-----------------+     |     |  |
            Y = Y - 1;-----------------------+     |  |
      ELSE                                         |  |
            Y = (Y-1) + (Y+2);---------------+     |  |
            X = Y;                           (4)   |  |
            Z = 0;---------------------------+     |  |
      ENDIF-----------------------------------+     |  |
      C = Z;-----------------------------------------+
      Value = X * E;
      Next = X + 1;----------------------------------+
      Prev = Prev - 1;                              (1)
      NewOne = Value + 12;-------------------------+
----- Hard Boundary -----
END
```

Figure 3-11. Example of the aggregation setup process order.

The aggregation setup process is applied twice in the above example, first to the segment of code between the upper hard boundary and the instruction to be footprinted, and then to the segment of code between the footprint instruction and the lower hard boundary. Notice that the aggregation of instructions starts from the inside out. This is due to the restrictions of the aggregation function. For example, the body of the `WHILE` loop needs to be aggregated into a single instruction before the `WHILE` can be aggregated. The same

applies to the body of the REPEAT as well as to the THEN and ELSE parts of the IF. Figure 3-12 outlines an algorithm for the aggregation setup process in which the function SELECT() is used. SELECT() takes four parameters as input - the starting instruction, the ending instruction, **I** and **C**, and returns as its value a set of instructions that are to be aggregated during the "current" iteration.

```
aggregationsetup ( S, E, I, C )
      REPEAT
            NextSet = select( S, E, I, C );
            aggr( NextSet, I, C );
      UNTIL (S = First Inst. in NextSet) AND (E = Last Inst. in NextSet)
```

Figure 3-12. Algorithm for the aggregation setup process.

Like the select() function, the above algorithm takes as input a starting and ending instruction, **I** and **C**. The algorithm repeatedly finds the next segment of code to be aggregated (via the select() function) and then calls aggr() to perform the aggregation. This process ends when the last segment of code aggregated is the segment of code between **S** and **E**.

3.9 Footprint Determination

The previous sections have laid a foundation for the actual determination of an instruction's footprint. Recall that the aggregation setup process takes the segments of code between the hard boundaries and the instruction to be footprinted and aggregates them each into single instructions. Following this process, Theorem 1 and the rules regarding soft boundaries within IFs, WHILEs and REPEATs (described in Section 3.7) are applied and the footprint of an instruction is determined. Figure 3-13 outlines the footprinting algorithm that determines soft boundaries of an instruction. The input

parameter **i** is the instruction being footprinted and **n** is the nearest-neighbor instruction of **i**.

```
footprint( i, n, I, C ) :

BEGIN
      IF CanSwap( i, n ) THEN        // Using Theorem 1
            Swap( i, n, I, C )
            footprint( i, New-Neighbor, I, C )
      ELSE
            IF p_n = { n }
                  STOP               // Found soft boundary
            ELSE
                  Case 1: n is a BLOCK aggr. inst.
                        deaggr( n, I, C )
                        footprint( i, New-Neighbor, I, C )


                  Case 2: n is a REPEAT or WHILE aggr. inst.
                        STOP         // Found soft boundary


                  Case 3: n is an IF aggr. inst.
                        deaggr( n, I, C )
                        // process both the THEN and ELSE parts but
                        // need to be able to stop at the end of
                        // the IF.
                        footprint( i, Then-New-Neighbor, I, C )
                        footprint( i, Else-New-Neighbor, I, C )
            ENDIF
      ENDIF
END
```

Figure 3-13. Algorithm for determining the footprint of an instruction.

In the footprint determination algorithm above, the two instructions **i** and **n** are checked using Theorem 1 to see if they can be swapped. If the swap is performed, then the `footprint()` function is called again with **i** and its new neighbor. Otherwise, the instruction type of **n** (the neighbor) is checked. If **n** is not an aggregate instruction, then the soft boundary is found. If **n** is a `REPEAT` or `WHILE` aggregate instruction, however, the processing stops - the soft boundary is found. Otherwise, deaggregation occurs for **n**

and `footprint()` is called again. In the case where **n** is an aggregate `IF` instruction, `footprint()` is called twice - first for the `THEN` part and then for the `ELSE` part.

The footprinting algorithm in Figure 3-13 can be used to footprint a single instruction. The time and space complexity in the worst case scenario is shown below.

Time Complexity = O(n)

Space Complexity = $O(n^2)$

The variable **n** refers to the number of lines of code in the function containing the instruction being footprinted. A proof of correctness for the footprinting algorithm in Figure 3-13 can be found in Appendix A.

3.10 Function Calls, Gotos, and Pointers

In the model described thus far, function calls have not been considered, the control flow effect of `GOTO`s has been ignored and the ability to reference other variables through the destructive and nondestructive dereferencing of pointers has been disregarded. These three programming capabilities can be incorporated into the model through compile-time analysis and/or the use of run-time extensions. Using compile-time analysis, an inspection of the code is all that is needed to determine soft boundaries.

3.10.1 Function Calls

The difficulty in the **IFM** (with respect to function calls) is in determining what side-effects and potential aliasing are caused by a function call. This poses a problem when applying the model; one needs to determine if an instruction being footprinted conflicts with another instruction that contains a function call. At compile-time, one of two approaches can be taken:

1) Perform some form of interprocedural side-effect and alias analysis, or

2) Assume the worst case scenario, the function call always conflicts with instructions to be footprinted.

If the first approach is taken, assumptions may need to be made regarding the effects that functions have on side-effects and aliasing in order to simplify the analysis. In addition, run-time extensions could be added to facilitate the detection of interprocedural side-effects and aliases to dynamically aid the process of determining the footprint of an instruction.

3.10.2 Gotos

With the exceptions of the `IF`, `REPEAT` and `WHILE`, the instructional footprint model so far has assumed a sequential flow of control. `GOTO`s have been purposely excluded. Nonetheless, the placement of `GOTO`s can cause an instruction that has been moved to its soft boundary to be skipped when it should be executed or executed when it should not be. For example, if we apply our current footprinting model to the code in Figure 3-14, the statement `Z=X` would be placed at the position immediately following `X=0` and `X=1` in the `IF`. This would be a mistake because if `X=Y` is true, then the `GOTO` would be executed, which implies that `Z=X` should not be executed. With current footprinting rules, `Z=X` would be executed independent of the value returned by the boolean expression, `X=Y`.

```
PROCEDURE GOTOexample
BEGIN
--- Hard Boundary ---
        IF   X=Y
             X=0;
             --- Soft boundary (Heel) ---
             GOTO  Label
        ELSE
             X=1;
             --- Soft boundary (Heel) ---
        ENDIF
        Z = X;  (Instruction to be footprinted)
Label:  Next = X + 1;
        A = Next;
        -- Soft Boundary (Toe) ---
--- Hard Boundary ---
END
```

Figure 3-14. Example of the hazard of using GOTOs.

One compile-time solution is to group code into "regions." These regions (4 in total) are determined by the position of the instruction being footprinted, its heel and its toe. This allows us to systematically determine the effects of a GOTO originating and ending in one of the four regions. If it is found that a particular GOTO has a "harmful" effect (due to the regions it encompasses), the soft boundaries can be adjusted to change the region that the GOTO originates in and the region to where it branches.

3.10.3 Pointers

Consider the read/write sets of instructions for a moment. Currently the model has defined them as sets of variables, but to be more accurate they should be defined as sets of *objects*. The reason for the shift in semantics is because if the use of pointers is considered, then there is no longer a one-to-one correspondence between variables and the objects to which they refer. In other words, several variables can have access to the same memory location (an object). So far the model has disregarded the use of pointers,

assuming all references are made to non-pointer variables. Consider what would happen if an instruction that is to be moved happens to dereference a pointer. In the following example, it is not known whether the integer pointer `ip` points to the integer `i`, so the analysis cannot determine whether the two instructions can be safely swapped.

```
PROCEDURE pointers
BEGIN
--- Hard Boundary ---
      i = 0;       (1)
      k = *ip;     (2)    (Instruction to be footprinted)
--- Hard Boundary ---
END;
```

The only action that can be taken at compile time is to assume the worst case, i.e. there is a conflict between the write set of instruction 1 and the read set of instruction 2. One way to facilitate this process in the model is to place the type `int` in the write set of instruction 2. Therefore, checking to see if `int` is an element of a read/write set would return true if `int` was the type of any element in the set.

The compile-time analysis for pointers is minimal because the values of pointers are not known at compile-time. In order to see what, if any, run-time extensions can be provided, consider Figure 3-15 which shows the soft boundaries (the heel and the toe) relative to the original instruction position.

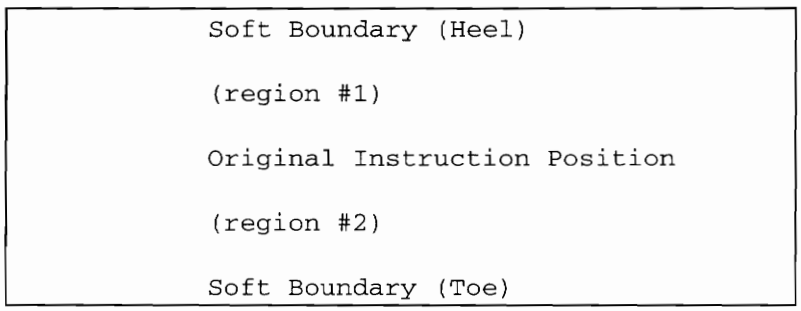


Figure 3-15. Soft boundaries relative to original instruction position.

Suppose that the model, in determining the soft boundaries for instruction **i**, ignored pointers altogether. The figure above shows two cases: when instruction **i** is moved to the heel and when it is moved to the toe. In either case, the question "*What can I do at run-time to ensure that when* ***i*** *is executed it does not conflict with any pointer references or dereferences for an instruction in the region (either region 1 or 2 depending on the case)?*" needs to be asked and answered. Run-time extensions can possibly be added to dynamically check any pointer (de)reference that can potentially conflict with **i**; this may, however, be costly with respect to performance if many checks have to be made (e.g. within a loop).

3.11 Summary

The instructional footprint model (IFM) is a model for defining and reasoning about programs in the CL language and in conjunction with an application framework can be used to determine the footprint of any instruction. Programs and instructions in CL are defined so that the necessary computational and control information is captured.

An application framework is also presented that includes functions to aggregate and deaggregate blocks of code. These functions are used to aid the process of determining footprints. The algorithm for determining an instruction's footprint is presented which relies on the use of soft and hard boundaries. The hard and soft boundaries define the potential and actual limits of an instruction's footprint.

Finally, footprinting issues related to function calls, gotos, and pointers are discussed. Certain assumptions, such as the extent of inter-procedural analysis to be performed, have to be made in order to handle these footprinting caveats. Compile-time analysis along

with run-time extensions can be used to solve many of the problems presented by function calls, gotos, and pointers.

CHAPTER 4 - Linda Primitive Transposition

4.1 Introduction

In support of enhanced performance, instructional footprinting can be used as an optimization technique with the goal of speeding up the execution of Linda programs. When applied to Linda programs, the optimization attempts to overlap (or parallelize) the normal computation of Linda programs with attendant processing associated with the TS manager (a separate process). This is achieved by initiating any **IN** or **RD** primitive earlier in the code and then receiving the returned tuple immediately before it is needed. The span of code between the early initiation of an **IN/RD** and the delayed receipt of the returned tuple is called the footprint of the instruction. An important part of footprinting in Linda is the instructional decomposition of the **IN** and **RD** primitives. For the purpose of footprinting, these primitives are decomposed into **INIT** and **RECV** operations to initiate a non-blocking request for a tuple and then to receive the returned tuple values, respectively. The following example illustrates how an **IN** is transformed into its two component operations.

```
IN("Matrix", i, j, ?element) ======> INIT("Matrix", i, j, ?element)
                                      RECV("Matrix", i, j, ?element)
```

The optimization goal then is to move the **INIT** instruction *backward* in the code so as to initiate the **IN** as early as possible. The **RECV** instruction is then moved *forward* in the code so that the (blocking) request for the tuple (and its values) is made as late as possible. This achieves maximum parallel activity between the TS manager and a Linda process.

The problem one faces when moving **INIT** and **RECV** instructions around in program code, however, is the ability (or inability) to preserve program semantics. There are two aspects to this problem:

1) The impact of moving **INIT/RECV** instructions past computational code, and

2) The impact of moving **INIT/RECV** instructions past Linda operations.

The first impact deals with the conflict of the read and write sets associated with program statements. Chapter 3 addresses this issue in detail. This chapter, however, focuses on the second aspect of code motion, i.e., the effect of moving **INIT/RECV** instructions past other Linda operations.

In [LANDR92a], one of the example programs that is optimized is a Linda program that solves the dining philosophers problem. This particular solution, adapted from a solution appearing in [CARRI89a], spawns (using the **EVAL**) `Num_Phil` philosopher processes, each of which executes `ProcessCycles` life cycles where a life cycle is thinking, sitting down at the table, eating, and then getting up from the table. Each life cycle requires three **IN** operations (one room ticket[3] and two chopsticks) to be performed before the philosopher can eat, and then three **OUT**s (putting the **IN**ed tuples back into TS) to be executed after the philosopher is finished eating. Figure 4-1 shows the main code that a philosopher process executes.

---

[3] The "room ticket" is used to ensure against deadlock. If there are N seats at the table (i.e. N philosophers), then N-1 room tickets are issued, and therefore only allowing N-1 philosophers to eat at the same time. This prevents the situation where all philosophers want to eat at the same time, and each pick up their left chopstick and wait forever for their right chopstick.

```
while (ProcessCycles > 0) {
      think();
      IN("Room Ticket");
      IN("Chopstick", Phil_ID);
      IN("Chopstick", (Phil_ID + 1) % Num_Phil);
      eat();
      OUT("Chopstick", Phil_ID);
      OUT("Chopstick", (Phil_ID + 1) % Num_Phil);
      OUT("Room Ticket");
      --ProcessCycles;
}
```

Figure 4-1. Code from the Linda program solving the dining philosophers problem.

In this program, three **IN**s are performed in order to gain access to the table and to allow a philosopher to eat. The optimized (and instructionally decomposed) version of this code segment is show below in Figure 4-2.

```
while (ProcessCycles > 0) {
      INIT_IN("Room Ticket");
      INIT_IN("Chopstick", Phil_ID);
      INIT_IN("Chopstick", (Phil_ID + 1) % Num_Phil);
      think();
      RECV_IN("Room Ticket");
      RECV_IN("Chopstick", Phil_ID);
      RECV_IN("Chopstick", (Phil_ID + 1) % Num_Phil);
      eat();
      OUT("Chopstick", Phil_ID);
      OUT("Chopstick", (Phil_ID + 1) % Num_Phil);
      OUT("Room Ticket");
      --ProcessCycles;
}
```

Figure 4-2. Optimized Linda program solving the dining philosophers problem.

In Figure 4-1, each **IN**, acting as a semaphore, is initiated one right after another before the `think()` routine is called. Effectively, there is an implied sequence of operations

associated with and among each of the **IN** primitives. For each **IN**, a request for a tuple is first made, with the Linda process blocking until the tuple is returned. This sequential nature is violated when **INIT**s for **IN**s (and **RD**s) are pushed past non-corresponding **RECV**s. The problem is that the TS manager does not guarantee that requests will be satisfied (and hence returned) in the same order requested. Suppose for example that the first **INIT** for the "Room Ticket" fails to find a matching tuple in Tuple Space. The request is shelved until potentially matching tuples arrive. Because an **INIT** does not block, the next **INIT** is processed, and if a matching tuple is found, the requested tuple is sent back to the Linda process *before* the first request for a tuple is satisfied. The requesting Linda process eventually blocks on the first **RECV** which is for the "Room Ticket". In this example, because there is no actual data being returned in the three tuples, and because Linda primitives are still serviced in the order of their request in our implementation of the TS manager, the program still works properly.

Although the dining philosophers program described above does work, other implementations of Linda using Instructional Footprinting on **IN** and **RD** primitives that return data can, and will, cause the semantics of the associated Linda program to be changed. The issue at hand, however, is broader than just whether an **INIT** operation can *cross over* a **RECV** operation (or vice versa) in order to maximize the footprint of an **IN** or a **RD**. The question that should be asked is when can an **INIT** or **RECV** operation cross over *any* Linda operation without changing the intended semantics of the original program. A footprinting technique called Linda Primitive Transposition (LPT) that addresses this question is proposed.

Recall that the goal of Instructional Footprinting is to speedup Linda programs. This speedup, however, must not come at the expense of sacrificing program semantics. So,

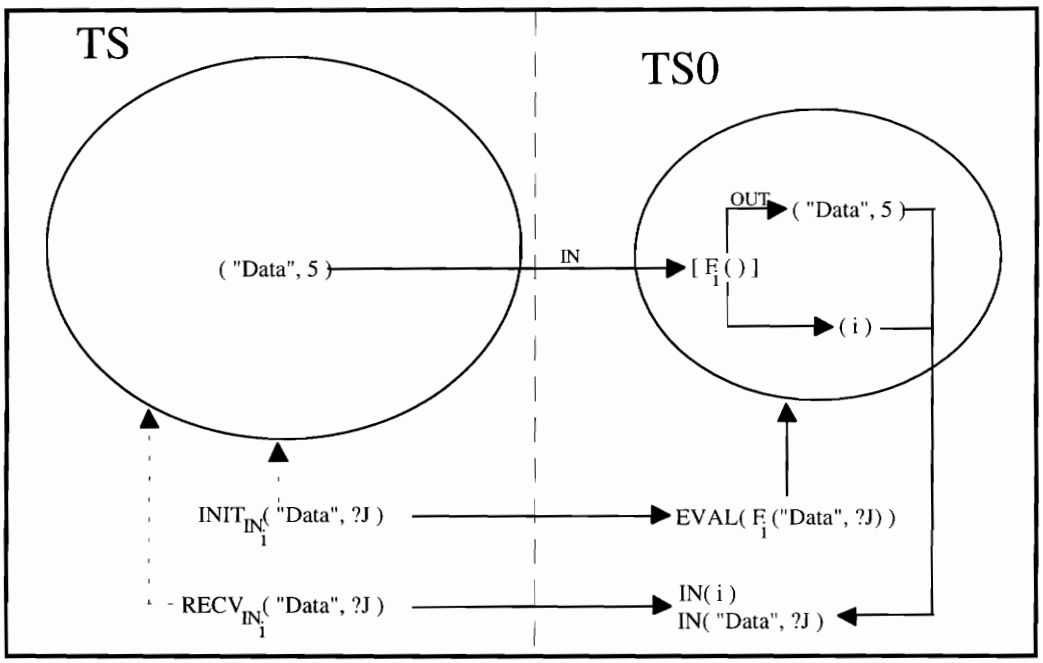
why not simply disallow **INIT** and **RECV** operations from crossing over any Linda operation (or at least other **INIT**s and **RECV**s)? As it turns out, when **INIT**s are executed one right after another (as opposed to alternating **INIT**s and **RECV**s) significant speedup can be achieved. The reason is the ability to exploit the multi-processing capabilities of the network. In our experiments, speedups as high as 64% have been experienced when footprinting a series of **IN** operations results in the grouping together of the **INIT**s and **RECV**s. Therefore, it is to our advantage to determine under what conditions one can push **INIT**s and **RECV**s past other Linda primitives (or possibly, their decomposed counterparts) and exploit the benefits of maximizing the distance between **INIT** and **RECV** pairs.

The remainder of this chapter is organized into five sections, each expanding on various aspects of LPT. The next section formally describes the semantics of the **INIT** and **RECV** operations based upon the TS formalization work of Jensen [JENSE90]. Section 4.3 describes the details of LPT by presenting the reasons why **INIT**s and **RECV**s cannot be naively moved past certain Linda operations and presents when it is safe to do so. Section 4.4 defines Tuple Sequencing and Tuple Identification, and describes their use in conjunction with LPT in preserving program semantics. Resultant speedups of several programs are shown in Section 4.5 followed by conclusions in Section 4.6.

4.2 The Semantics of INIT and RECV

In order to accurately and precisely discuss the movement of **INIT**s and **RECV**s past other Linda operations we must first provide a formal semantic definition describing their behavior [LANDR93a]. We base our definitions on earlier work by Jensen [JENSE90], where he formally characterizes Tuple Space and the Linda primitives that operate on it.

Before giving a formal definition of **INIT** and **RECV**, however, it is first appropriate to examine the operations informally and within a framework simplifying the Tuple Space (TS) model. Assuming that there is a single TS where all regular Linda primitives operate, let us define a complimentary tuple space (TS0) for the purpose of explaining the semantics of **INIT**s and **RECV**s. To the Linda programmer, only TS exists. For the purpose of illustration, however, when **IN**s and **RD**s are transformed into their component **INIT** and **RECV** operations, TS0 is needed to describe their operation.




```
Fi ( )
{
      INTS( "Data", ?Value );
      OUTTS0( "Data", Value );
      RETURN i;
}
```

Figure 4-3. Description of INIT and RECV operations for an IN.

Figure 4-3 shows an example of how the **INIT** and **RECV** for an **IN** are implemented with the use of TS0. The **INIT** performed on TS can be implemented as an **EVAL** executed on TS0, while the accompanying **RECV** is implemented as two **IN** operations executed on TS0. The subscript i is used, for convenience, to uniquely identify all **IN**s and **RD**s in a program. The function being **EVAL**ed, `F_i()`, has the task of **IN**ing a tuple from TS (using the same tuple template) and then **OUT**ing it to TS0. For example, in Figure 4-3 the tuple `("Data",5)` is found in TS and then **OUT**ed to TS0 by `F_i()`. By virtue of returning i, a second tuple is also placed in TS0 by `F_i()` and has as its contents the unique designator value associated with i.

The **RECV** operation is implemented as a series of two **IN** operations performed on TS0. The first **IN** retrieves the uniquely valued data tuple and the second **IN** removes the resultant tuple produced by the **EVAL**. In our example, the **RECV** operation first performs an **IN**`(i)` to retrieve the synchronizing data tuple, and thereby *forces* it to block until the corresponding **INIT** has been completed. This is followed by an **IN**`("Data",?J)` to retrieve the data tuple (in this case `("Data",5)`).

Implementing **INIT**s and **RECV**s in terms of other Linda operations allows us to define their semantics using existing formalization machinery already in place. The following definitions provide a formal description of the **INIT** and the **RECV** for **IN** and **RD** operations based on Jensen's work and our "modified" view of Tuple Space.

DEFINITIONS

INIT$_{IN}$

$$\frac{p \xrightarrow{INIT_{in}(Si)} p'}{TS \cup \{t'[p]\} \xrightarrow{INIT_{in}(Si)} TS \cup \{t'[p']\}}$$

$$TS_0 \xrightarrow{INIT_{in}(Si)} TS_0 \cup \{t''\}$$

When a process state transition from P to P' occurs, Tuple Space (which is TS unioned with the live tuple t' containing process state P) is transformed into a new Tuple Space consisting of TS unioned with the live tuple t' with the new process state P'. In addition, the live tuple t'' is added to TS0.

INIT$_{RD}$

$$\frac{p \xrightarrow{INIT_{rd}(Si)} p'}{TS \cup \{t'[p]\} \xrightarrow{INIT_{rd}(Si)} TS \cup \{t'[p']\}}$$

$$TS_0 \xrightarrow{INIT_{rd}(Si)} TS_0 \cup \{t'''\}$$

When a process state transition from P to P' occurs, Tuple Space (which is TS unioned with the live tuple t' containing process state P) is transformed into a new Tuple Space consisting of TS unioned with the live tuple t' with the new process state P'. In addition, the live tuple t''' is added to TS0.

RECV$_{IN}$ / RECV$_{RD}$

$$\frac{p \xrightarrow{RECV(Si)} p'}{TS \cup \{t'[p]\} \xrightarrow{RECV(Si)} TS \cup \{t'[p']\}} match_{TS_0}(S_i, t)$$

$$TS_0 \cup \{t\} \cup \{i\} \xrightarrow{RECV(Si)} TS_0$$

When a process state transition from P to P' occurs, Tuple Space (which is TS unioned with the live tuple t' containing process state P) is transformed into a new Tuple Space consisting of TS unioned with the live tuple t' with the new process state P'. In addition, two tuples, t and i, are removed from TS0.

where

S denotes a tuple template,

p denotes a processing environment,

t denotes a tuple,

t[p] denotes the process environment p executing within the active tuple t,

t'' is an active tuple whose job is to perform an **IN**(S_i) from **TS**, place the new tuple in **TS0**, and then terminate leaving tuple { i } in **TS0**, and

t''' is the same as t'' but it **RD**s from TS instead of **IN**s.

The **IN**s, **RD**s, and **OUT**s associated with t'' and t''' (i.e. the function F_i ( )) have slightly different semantics than found in [JENSE90], but only in that they are dealing with both TS and TS0 rather than TS alone.

4.3 LPT and Program Semantics

Given the formalization of **INIT** and **RECV** operations in Section 4.2, it is now possible to better describe the code motion of **INIT**s and **RECV**s relative to the preservation of program semantics. In this section, we first present two examples of code motion that cause program semantics to be altered, followed by a description of a bifurcated classification scheme for these semantic violations. We also present a discussion of the cases in which it is safe to perform code motion.

Example 1

Moving an $\mathbf{INIT_{IN}}$ up past a $\mathbf{RECV_{IN}}$ of a different Linda operation can alter program semantics by potentially allowing multiple access to critical regions delineated by other Linda operations. The following example illustrates this potential problem.

```
   :                                          :
   "spawn task 1 to update                    "spawn task 1 to update
    current checking balance"                  current checking balance"
   :                                          :
IN("Task 1")                                INIT("Task 1")
IN("Checking", ?bal)              ====>     INIT("Checking", ?bal)
   :                                        RECV("Task 1")
   :                                        RECV("Checking", ?bal)
   :                                          :
```

In the unoptimized code above, suppose that a "checking balance" tuple has previously been **OUT**ed and that a task (task 1) has been spawned to update it before the current program segment accesses it. Without any optimization, the current program segment will wait for task 1 to finish by performing an **IN**("Task 1"). Effectively, the first **IN** will block until a matching tuple is found, meaning that Task 1 has finished. The second **IN** will then remove the "checking balance" tuple from TS. Recall that in Instructional Footprinting, **INIT**s behave like **EVAL**s and can be serviced in any order. In the optimized code, if the **INIT** for the "checking balance" tuple is satisfied before the **INIT** for the "Task 1" tuple then it is possible for the "checking balance" tuple to be taken out of TS before the new value is placed in it (assuming task 1 is not finished and has not written an updated checking balance tuple). The reason program semantics are altered is because the second **IN** (in the unoptimized code segment) operates under the assumption that the first **IN** has completed, meaning Task 1 is complete. However, this assumption is violated in the optimized version because the **INIT** requesting check balance data has been moved up past the intended synchronizing **RECV** for task 1 completion. In effect, the second **IN** in the unoptimized version has the potential for being serviced before the first **IN**.

Example 2

Suppose Instructional Footprinting allows us to move a $\mathbf{RECV_{RD}}$ down past an **OUT**. It turns out that performing this move can alter program semantics also. The following example shows that, when optimized, a **RD** request can be satisfied by an **OUT** operation that would otherwise (in an unoptimized scenario) be impossible.

```
RD("Data", ?x)                      INIT("Data", ?x)
OUT("Data", y)      ========>       OUT("Data", y)
                                    RECV("Data", ?x)
```

If the code above is not optimized, then the **RD** will block if there are no "Data" tuples in TS. It is impossible for the **RD** to be satisfied by the **OUT** following it. However, such is not the case when the code is optimized. In the optimized code segment, suppose that when the **INIT** is executed there are no "Data" tuples in TS, in which case, the service is delayed. The next TS operation performed is the **OUT** which does place a "Data" tuple in TS. Once this happens, the **INIT** request that was delayed can and will be satisfied. Clearly, such is *not* the intent of the original unoptimized code segment.

Both of the above examples have one aspect in common - a *temporal influence* has inadvertently provided the potential for program semantics to change. In particular, program semantics can change because lexically prior code that should finish execution before successor code is encountered does not finish (as in Example 1), or because the execution of successor code is started before prior code has completed execution (as in Example 2).

Code motion of an instruction can be viewed as performing a series of two instruction swaps, thereby, propagating an instruction up or down in a program. Consider, for example, a code segment containing instructions **A** and **B** where **A** immediately precedes **B** lexicographically. In normal execution, instruction **A** would initiate, execute, and then finish before instruction **B** is reached. Initiating instruction **B** before **A** can potentially modify the process state that instruction **A** is expecting and therefore alter the operational effects of **A**. Similarly, the effect of instruction **A** can be altered by delaying **A**'s completion until after instruction **B** starts executing. If the operation of instruction **A** is affected, the code motion performed is said to have an *Anterior Impact* on program semantics because instruction **A** is temporally anterior with respect to the two instructions.

It is also possible for instruction **B** to be operationally affected by the same two types of code motion, that is, initiating instruction **B** before **A** or allowing **B** to start executing before instruction **A** is finished. The process state that instruction **B** expects is one in which instruction **A** has completed executing. Therefore, the effects of instruction **B** can change if the process state that is expected by **B** is different because instruction **A** has not completed execution. Therefore, if instruction **B** is affected, the code motion performed has had a *Posterior Impact* on program semantics because instruction **B** is temporally posterior with respect to the two instructions.

For further discussion, it is important to examine the components of code motion in terms of moving **INIT**s and **RECV**s. Each movement involves two component instructions - a *primary* and a *secondary* component. The primary component is the instruction being moved (i.e. the **INIT** in the first example and the **RECV** in the second), and the secondary component is the instruction being moved over (the "other" Linda operation). In Figures 4-4a and 4-4b, the two instructions being moved, the **INIT** and the **RECV**, respectively, are the primary components.

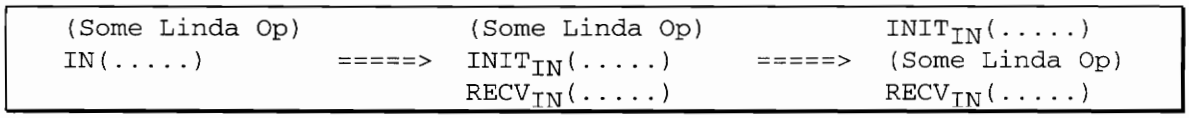
(a) (b)

Figure 4-4. The code motion basics of INITs and RECVs.

As Figure 4-4 shows, there are only two types of code motion being considered. The first type (4-4a) is moving an **INIT** up past some Linda operation and the second type (4-4b)

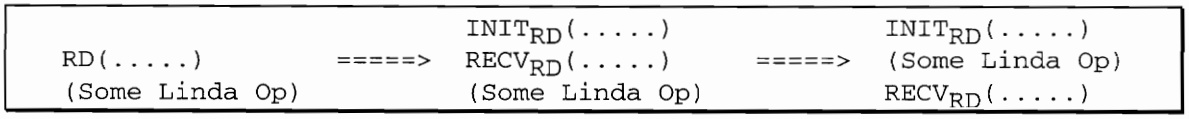
is moving a **RECV** down past some Linda operation. The Linda operation (also called the secondary component instruction) can be an $\mathbf{INIT_{IN}}$, $\mathbf{INIT_{RD}}$, $\mathbf{RECV_{IN}}$, $\mathbf{RECV_{RD}}$, **OUT**, **EVAL**, **INP**, or a **RDP**.

The following example using an **IN** (a **RD** can be used just as well) shows unoptimized code on the left, instructionally decomposed code in the middle, and optimized code on the right.



In the unoptimized code, the **IN** contains the primary component instruction (the **INIT**) and the `Linda Op` is the secondary component instruction. If the code is optimized by performing the **INIT** for the **IN** before the `Linda Op`, an anterior impact can occur if the operation of the `Linda Op` is affected. Likewise, a posterior impact can occur if the **IN** is semantically altered because of the code motion.

Both anterior and posterior impacts can also occur when a **RECV** is moved down past a Linda operation. The example below using a **RD** shows unoptimized, instructionally decomposed, and optimized code respectively.



In this case, the **RD** contains the primary component instruction (the **RECV**) and the `Linda Op` is the secondary component instruction. An anterior impact can occur when the operation of the **RD** is affected. If the `Linda Op` has been operationally affected then a posterior impact has occurred.

An example of a posterior effect occurring is when an **INIT** for a **RD** is moved above both the **INIT** and **RECV** for an **IN**. The following example shows the posterior effect.

```
IN("Some Tuple")                        INIT_RD("Some Tuple")
RD("Some Tuple")      =======>          INIT_IN("Some Tuple")
                                        RECV_IN("Some Tuple")
                                        RECV_RD("Some Tuple")
```

If there is only one matching tuple in Tuple Space (TS) then the original code will block on the **RD** because the **IN** removed the tuple from TS. In the optimized code, the $\mathbf{INIT_{RD}}$ removes the matching tuple intended for the **IN** (i.e. the $\mathbf{INIT_{IN}/RECV_{IN}}$ pair). This results in the **RD** being adversely affected by a previous instruction that did not execute but should have.

An example of an anterior effect happens when a **RECV** for an **IN** is moved past an **OUT** operation. The following example shows the anterior effect.

```
IN("Some Tuple")                        INIT_IN("Some Tuple")
OUT("Some Tuple")      ======>          OUT("Some Tuple")
                                        RECV_IN("Some Tuple")
```

In the unoptimized code, if there are no matching tuples in TS when the **IN** is executed then the **IN** will block. However, in the optimized code the **OUT** places a matching tuple in TS that will satisfy the $\mathbf{INIT_{IN}}$. This results in the **IN** being adversely affected by an instruction (supposedly) yet to be executed.

Although it turns out that most movements of **INIT** and **RECV** operations across Linda operations can potentially alter program semantics, there are three situations where it is safe. Interestingly enough, 2 of the 3 cases involve moving **INIT**s (for both **IN**s and **RD**s) up past **EVAL** operations.

Why is moving an **INIT** up past an **EVAL** so different from other movements that it has no effect on program semantics? Part of the answer lies in the fact that the impact of an **EVAL**, by definition, is not time constrained. Recall, all that is guaranteed by an **EVAL** operation is that it will create a process tuple in tuple space; it does not guarantee when it will execute, and thereby, create a data tuple. Consider the following example.

```
                                          INIT_IN("Data", ?x)
EVAL("Data", F())          ======>        EVAL("Data", F())
IN("Data", ?x)                            RECV_IN("Data", ?x)
```

In the unoptimized version above, the **EVAL** produces a live tuple in TS that consists of two fields - "Data" and a process evaluating function `F()`. Control returns to the **IN** operation as soon as the live tuple is placed in TS (i.e. it does not wait for the process evaluating `F()` to finish).

If the code is optimized by executing the **INIT** before the **EVAL**, program semantics are not altered because the **EVAL** is non blocking. In other words, it is not guaranteed that the **EVAL** will finish processing before the **IN** is executed. In fact, it is not even guaranteed that the process created by the **EVAL** will be started before the **IN** is reached. Therefore, executing the **INIT** for the **IN** before the **EVAL** is consistent with original program semantics.

The third safe movement involves moving an **INIT** for a **RD** up past a **RDP** operation. The reason this movement is considered safe stems from the systematic elimination of the three possibilities that can alter program semantics, i.e.

1) The alteration of TS,

2) The detection of tuple presence in TS, and

3) The blocking nature of certain Linda operations.

Effectively, when trying to determine if a movement does or does not alter program semantics, it is necessary and sufficient to explore these three possibilities. In the case of moving an $\mathbf{INIT_{RD}}$ up past a **RDP**, the following questions are asked:

1) Is TS altered by either operation?

2) Is the detection of tuple presence affected?

3) Is unnecessary blocking(or a lack thereof) causing an adverse effect?

These questions must be asked from the point of view of both the **RD** and the **RDP**. As it turns out, the initiation of a **RD** does not alter TS or block, and therefore does not affect the detection of tuple presence in TS for other operations. In addition, the **RD** is not affected by not previously executing the **RDP** because the **RDP** does not affect TS, does not block and therefore does not affect the detection of tuple presence in TS for other operations.

The following table summarizes which **INIT**s and **RECV**s for **IN**s and **RD**s can be safely moved past other Linda operations. For example, the first entry in the last row is a NO, which indicates that a $\mathbf{RECV_{RD}}$ cannot be moved down past an $\mathbf{INIT_{IN}}$ without possibly altering program semantics. We can also view the same scenario as moving an $\mathbf{INIT_{IN}}$ up past a $\mathbf{RECV_{RD}}$. Its corresponding entry in the table also indicates such a move can

have an adverse impact on program semantics. The justifications for each entry can be found in Appendix B.

Table 4-1. Summary of acceptable INIT/RECV movements.

	$INIT_{IN}$	$RECV_{IN}$	$INIT_{RD}$	$RECV_{RD}$	OUT	EVAL	INP	RDP
$INIT_{IN}$	NO	NO	NO	NO	NO	YES	NO	NO
$RECV_{IN}$	NO	NO	NO	NO	NO	NO	NO	NO
$INIT_{RD}$	NO	NO	NO	NO	NO	YES	NO	YES
$RECV_{RD}$	NO	NO	NO	NO	NO	NO	NO	NO

4.4 Tuple Sequencing and Tuple Identification

Table 4-1 paints a fairly bleak picture about the prospects of code motion as it relates to the movement of **INIT**s and **RECV**s across Linda operations. In fact only 9% (3 out of 32) of the possible movements are safe. Is there a way to increase this percentage without sacrificing program semantics and still achieve speedup? The answer is a qualified - yes.

Recall that the motivation behind moving **INIT**s and **RECV**s across Linda operations is to gain significant speedups by maximizing the footprint size. In our implementation of Linda, a distributed system using a separate process as a TS manager that employs sockets as the communication mechanism, it is possible to increase the percentage of safe movements from 9% to 72% through the use of two proposed techniques called *Tuple Sequencing* and *Tuple Identification* [LANDR94b]. Tuple Sequencing is a technique that ensures that Linda operations in a particular process are executed (and completed) in the

order they are *sent* to TS. Tuple Identification ensures returned tuples are matched up with the correct **RECV**s. The following table summarizes which movements are safe if Tuple Sequencing and Tuple Identification are used.

Table 4-2. Summary of code motion using Tuple Sequencing and Tuple Identification.

	$INIT_{IN}$	$RECV_{IN}$	$INIT_{RD}$	$RECV_{RD}$	OUT	EVAL	INP	RDP
$INIT_{IN}$	NO	YES	NO	YES	NO	YES	NO	NO
$RECV_{IN}$	YES	YES	YES	YES	YES	YES	YES	YES
$INIT_{RD}$	NO	YES	NO	YES	NO	YES	NO	YES
$RECV_{RD}$	YES	YES	YES	YES	YES	YES	YES	YES

To illustrate the impact of Tuple Sequencing and Tuple Identification, it is helpful to analyze which cases of code motion are now safe to perform and which ones are still unsafe. It is also important to recognize that the basic problem encountered with both anterior and posterior temporal influences is that the order of operations explicitly laid out by the programmer in the unoptimized code is violated by the optimized code, which in turn causes program semantics to be altered. Notice that in all cases where a NO (in Table 4-1) is **not** changed to a YES (in Table 4-2) involves moving an **INIT**. In all other cases it is changed to a YES because Tuple Sequencing and Tuple Identification enables us to guarantee that the original order of operations (or at least the original order of initiation) is preserved. For example, consider the following code.

```
IN("Task 1")                                    INIT("Task 1")
IN("Task 1", "Checking", ?bal)   ====>         INIT("Task 1", "Checking", ?bal)
                                                RECV("Task 1", "Checking", ?bal)
                                                RECV("Task 1")
```

Moving one **RECV$_{IN}$** down past another **RECV$_{IN}$** is "safe" when Tuple Sequencing and Tuple Identification is employed because the original order of the **INIT**s has not changed. That is, Tuple Sequencing and Tuple Identification guarantee that **INIT**`("Task 1")` will be serviced before **INIT**`("Task 1","Checking",?bal)` - as it should be. Whereas moving an **INIT$_{IN}$** up past another **INIT$_{IN}$** is not safe because the original order of the **INIT**s is altered.

The way in which Tuple Sequencing helps is that when multiple Linda requests are made to the TS manager from the same process, the requests are serviced one at a time and in the order in which they are received by the TS manager. This means that if two **INIT**s are sent to the TS manager from the same process, the second **INIT** is not processed until the first **INIT** finds a matching tuple. This ensures the original, intended order of Linda operations is maintained while still realizing substantial speedups by allowing the **RECV**s to be moved maximally downward. Two **INIT**s from separate processes need not be considered because ordering is not implied by asynchronous processes.

Tuple Identification allows tuples returned to a process to be tagged with a unique identifier to indicate for which **RECV** it is intended. This is necessary because if Tuple Sequencing is used in the code segment above, the first tuple being sent back must reflect that it is intended for the second **RECV** executed and not the first one. Moreover, when Tuple Identification is employed, not only must each tuple be uniquely tagged but provisions must be made for returned tuples to be stored (most likely on the process side) in the event the returned tuple is not the one currently being requested (**RECV**ed).

4.5 Results using LPT

In order to show the effectiveness of Instructional Footprinting, three programs were executed with and without Instructional Footprint optimization. Tuple Sequencing and Tuple Identification were used, as needed, to preserve program semantics. The three programs are the dining philosophers problem, a distributed banking simulation, and a raytrace program. In all three cases, the reason for our observed speedup is because **INIT**s and **RECV**s cross over other Linda operations. We also note that in each case, our footprinting algorithm groups several **INIT**s together which appears to have an additional positive impact on speedup. The primary reason why grouping **INIT**s together for execution causes (sometimes dramatic) program speedup is because of the compounding effect of overlapping footprints with the multi-processing characteristics of the network.

Experiments were performed on two machines networked together. One machine is used to execute all Linda processes and the second is used to run the TS manager and house TS itself. The two machines are Commadore Amigas running Unix System V Release 4. Communication between the Linda processes and the TS manager is accomplished through the use of sockets at the TCP/IP level and data is transferred without delay.

4.5.1 Dining Philosophers Problem

The dining philosophers is a classic problem used to illustrate the expressiveness (or lack thereof) of a programming language. Although the dining philosophers problem does not represent a typical "real world" problem in terms of utility, it is used for two reasons:

1) the solutions are generally known to most researchers, and

2) the programming structures and techniques used are common to solutions of real world problems.

This particular solution, described in Section 4.1, is optimized by taking the three **IN** operations and performing their **INIT**s in immediate succession followed by their three **RECV**s. The following graph provides a comparison of the execution times for various runs of the original solution to those of their corresponding optimized versions.

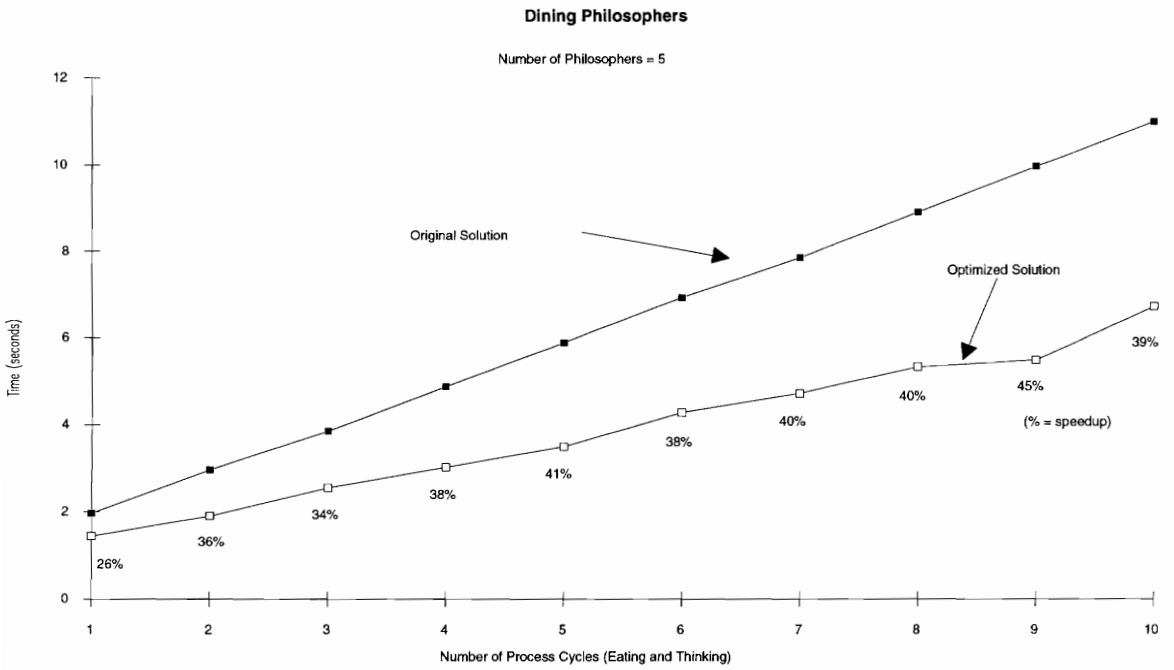


Figure 4-5. Graph of Dining Philosopher's execution times vs. number of life cycles.

Each run reflects a different number of life cycles (varying from 1 to 10); the speedups range from a low of 26% to a high of 45%. Moreover, when tests were run varying the number of philosophers from 1 to 10 (and holding the number of life cycles constant at 5), execution speedup leaped as high as 64%.

The runs reflected in Figure 4-5 were performed with no processing being performed in the eating and thinking routines. Suspecting that performance might suffer if the eating and thinking routines have unequal processing times, other experiments were performed varying the processing times of the thinking routine. As a result of increasing the processing times of the thinking routine, subsequent speedup dropped as the processing time increased. The speedup dropped to zero but never was a negative speedup (a decrease in performance) experienced.

4.5.2 Distributed Banking Simulation

This Linda program simulates a distributed database of checking accounts, with pieces at each of three different banks[4] . The simulation takes several checking accounts (database records) and duplicates each at the different sites. The simulation then reads in a series of transactions for each site and posts them against the databases. The simulation spawns processes to manage the data at each site and to handle the transactions. The following graph shows the speedup achieved when optimized as compared to the original (unoptimized) solution.

---

[4] This problem is similar to the one presented in [LANDR92a]. In fact it solves the same problem but is written by another programmer and hence is slightly different. This difference allowed for better optimization than did the one presented in [LANDR92a].

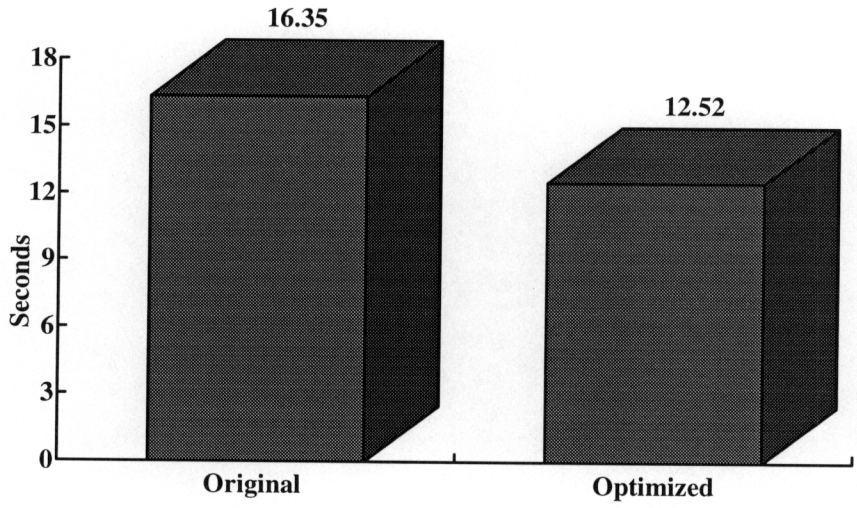


Figure 4-6. Execution times for the Distributed Banking Simulation.

Through the use of Instructional Footprinting, **IN**s and **RD**s are decomposed into **INIT**s and **RECV**s, of which, 25 of the **INIT**s were executed in groups ranging from 2 to 4 in size. The resulting program executed on the average about 23% faster than the original code.

4.5.3 RayTrace Program

The raytrace program reads in an ASCII file describing a scene to be traced. The program generates a file containing the raytrace image in Utah Raster RLE format. Worker processes are **EVAL**ed to compute individual scan lines for the raytrace image. The following graph shows execution times for the original code as compared to code optimized by Instructional Footprinting.

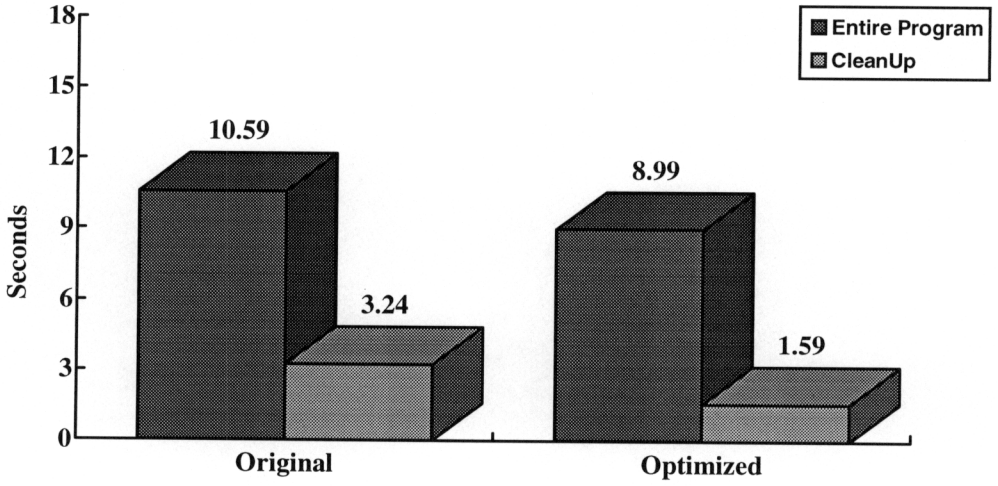


Figure 4-7. Execution times for the Raytrace Problem.

The chart above shows timings for total program execution and for the execution of the raytrace cleanup routine. The reason for showing the timings for the cleanup routine is because this is where all of the optimizations were performed (and hence where all the speedup really comes from). For each worker process, the cleanup routine **IN**s several tuples from TS containing statistics. These **IN** operations are optimized to execute the **INIT**s in one group and the **RECV**s in another. The resulting speedups averaged about 15% for the entire execution of the program and about 51% for the cleanup routine.

4.6 Summary

Instructional Footprinting is an optimization technique used in Linda systems to speedup the execution of Linda programs. **IN** and **RD** operations are decomposed into two parts - an initiation (**INIT**) and a receive (**RECV**). The initiation is executed as early as possible

while the receive is executed as late as possible. This span of code between the initiation and the receive is the footprint of the **IN** or **RD**.

There are many difficulties in assuring that program semantics remain unaltered when moving **INIT**s and **RECV**s around in code. One such difficulty pertains to identifying whether program semantics are altered when an **INIT** or a **RECV** is moved past a Linda operation. By defining detailed semantics for **INIT** and **RECV** operations, it is possible to identify which movements can potentially alter program semantics. Anterior and Posterior Temporal Influences provide a means of classifying movements which can cause semantics to change. We have shown that, in many cases, the use of Tuple Sequencing and Tuple Identification in conjunction with LPT can ensure that program semantics are not changed.

Results from several programs show significant speedups with the use of Instructional Footprinting augmented with LPT. In each case, the use of LPT permitted an increased amount of optimization, and subsequently, played a critical role in the amount of speedup observed. Therefore, we contend that Instructional Footprinting, together with LPT, contributes significantly to increased performance of programs written from the Linda perspective.

CHAPTER 5 - Footprint Quantity

5.1 Introduction

The goal of Instructional Footprinting is to enhance execution performance of Linda programs by overlapping the normal computation of Linda programs with attendant processing associated with the TS manager. In a world where all things are equal, it would follow that larger footprints imply larger performance improvements. However, two footprints of equal size may or may not provide equal performance improvement. The reason is that not all instructions within a footprint take the same amount of time to execute. In other words, the *quality*, or execution time, of instructions vary. In addition, the lexical proximity of certain instructions may provide additional performance improvements. The goal of producing footprints with a particular level of performance improvement can be achieved in one of two ways. Footprints can include either

1. A large *quantity* of average *quality* instructions, or

2. A smaller *quantity* of high *quality* instructions.

In either case, it is desirable to maximize the quantity of instructional footprints. Larger footprint sizes not only provide more instructions to execute in parallel with the TS manager, but also increase the probability of having higher quality instructions within a footprint.

With this in mind, the question turns to *"What can be done to actively work towards increasing the quantification of instructional footprints?"* The answer can be stated from two perspectives - from the researcher and from the programmer. The first perspective deals directly with the footprinting process itself. It focuses attention on improvements to the footprinting process that can be made by the researcher to increase the size of footprints by breaking down and stretching soft boundaries [LANDR94a]. The

programmer's perspective focuses on programming styles and how they affect footprint size.

This chapter discusses instructional footprint quantification from both the researcher's and programmer's perspective. Section 5.2 discusses the point of view of the researcher by first presenting footprinting results for the basic footprinting model (Section 5.2.1) described in Chapter 3. The model is then improved upon in Section 5.2.2 by applying the Linda Primitive Transposition (LPT) technique proposed in Chapter 4 and observing how footprint sizes are affected. This is followed by Section 5.2.3 which describes the application of instruction piggybacking to the basic model. Instruction piggybacking is another proposed footprinting technique aimed at speeding up Linda programs by increasing footprint sizes. Section 5.2.4 shows the results when both improvement techniques are simultaneously applied to the basic footprinting model. Section 5.3 discusses what programming styles a programmer might adopt in an attempt to increase footprint sizes. Section 5.3 also highlights the results of some statistical tests that were performed to measure the relative effects of different factors on footprint sizes. Section 5.4 provides some concluding remarks about the quantification of instructional footprints.

5.2 The Researcher's Perspective

As an implementor of the Instructional Footprinting optimization, the researcher has direct control over the potential power of the footprinting process. With the goal of safely increasing the size of footprints, the researcher is tasked with breaking down and removing limitations associated with the footprinting process. For example, given a particular **IN** or **RD** instruction, there are two types of boundaries placed on the movement of the sub-instructions (the **INIT** and **RECV**). They are the *hard* and *soft* boundaries. The hard boundary represents the outer limits of where the **INIT** or **RECV**

can be moved. The beginning and ends of functions or control structures are examples of hard boundaries. Soft boundaries, on the other hand, represent semantic limitations associated with moving an **INIT** or **RECV**. A variable conflict is one example of a soft boundary. The hard and soft boundaries are just two examples where researchers can focus attention in an attempt to increase the size of footprints.

This section primarily focuses on the soft boundary. Results from the use of a basic footprinting model are presented. This is followed by a discussion of two improvements that can be made to the basic footprinting algorithm. Results are presented for each improvement when applied individually and then when used together.

5.2.1 The Basic Model

The specific details of the basic instructional footprinting model can be found in Chapter 3. For the purpose of maximizing footprint quantity, the relevant parts of the model are the definition of soft boundaries. Recall that the soft boundary is what directly defines the size of the associated footprint. There are basically 3 types of soft boundaries. They are defined by:

1) Variable conflicts,

2) Linda conflicts, and

3) Construct conflicts.

Suppose an **INIT** is being moved up in the code as far as possible. A variable conflict may occur between the **INIT** and a neighbor instruction. A typical variable conflict may be that the neighbor instruction is writing a variable that is being read by the **INIT**. Because of this conflict, the two instructions cannot be safely swapped and the soft

boundary is then determined. A Linda conflict occurs when the neighbor instruction happens to be a Linda primitive. Because it is possible for program semantics to be altered by moving the **INIT** past the Linda primitive, the swapping does not take place and the soft boundary is found. The last conflict occurs when a variable or Linda conflict occurs and the neighbor instruction is in fact an aggregate instruction, meaning a collection of simple or aggregate instructions (e.g., a while loop and a block of code). If a construct conflict occurs, the aggregate instruction and the **INIT** are not swapped and the soft boundary is determined.

In order to determine the effectiveness of Instructional Footprinting, Linda programs were solicited from academia and from industry. From the response, 12 Linda programs were assembled for the purpose of collecting statistics about footprint sizes for **IN** and **RD** primitives found in the programs. The 12 programs were chosen because they were examples of "real world" applications. Table 5-1 contains some statistics collected for each of the 12 programs.

Table 5-1. Selected statistics for the 12 programs in the test bed.

Program	% Functions to be Optimized	AVG Hard Boundary Size	Number of INs and RDs to be Optimized	AVG # Linda Operations in Functions to be Optimized
DNA	29	12.65	4	4.50
Dist. Memory Manager	29	22.60	9	4.33
Boyer-Moore	38	26.90	11	7.00
Dist. Banking Simulation	71	12.30	40	11.86
Parallel FFT	25	5.00	3	3.50
Parallel Sort	75	27.50	7	5.33
Parallel Sum	100	12.90	8	5.33
Bug Trudger	40	28.60	7	5.33
Spanning Tree	40	19.00	3	5.00
Dining Philosophers	40	3.00	6	3.33
Raytrace	36	26.40	11	5.14
Matrix Multiply	40	4.50	3	3.50

The statistics collected were based upon a ratio of the actual footprint size for an **IN** or a **RD** to its footprint potential (determined by the hard boundary). Strictly speaking, a footprint is the sum of the upward mobility of the **INIT** and the downward mobility of the **RECV**. For the purpose of simplicity, a footprint in this chapter refers to either the distance that an **INIT** can move up or the distance that a **RECV** can move down. In other words, there are two relevant footprints for each **IN/RD**, the one for the **INIT** and the one for the **RECV**.

From the 12 programs examined, there were 112 **IN**s and **RD**s from which 224 footprints were calculated. With respect to the basic footprinting model, **IN**s and **RD**s are treated the same. However, later improvements do distinguish between the two. For the purpose of comparison, statistics in four categories of footprints (i.e., $\mathbf{INIT}_{\mathbf{IN}}$, $\mathbf{INIT}_{\mathbf{RD}}$,

RECV$_{IN}$, and **RECV$_{RD}$**) are presented. The following table shows average footprint ratios for these four categories.

Table 5-2. Footprint ratios for the basic footprint model.

	INIT	RECV	INIT/RECV
IN	0.17	0.12	0.10
RD	0.41	0.11	0.12
IN/RD	0.22	0.12	0.10

These average ratios only include footprints for which there is a potential greater than zero. For example, if there was no room to move an **INIT$_{IN}$** (say, it is at the top of a `while`) then it is not included in the average of 0.17. The average footprint ratios for the **INIT/RECV** column include only those where the overall potential (up and down) is not zero. Also, the numbers in the **IN/RD** row are a weighted average of the two numbers above it. Notice that the averages are weighted towards **IN**s. This is because 83 of the 112 Linda primitives optimized are **IN** operations.

Other than the average footprint ratio for **INIT$_{RD}$**'s, the averages ranged from .10 to .22. *What caused the average footprint ratio for **INIT$_{RD}$** to be 0.41, dramatically higher than the others?* There are a several reasons for this. First of all, there were only 29 **RD**s compared to 83 **IN**s, which can have an impact on average footprint ratios if outliers exist. In addition, about half of the **INIT$_{RD}$**s had a footprint potential of zero and therefore were not included in the average. Second, all of the footprint potentials for the **INIT$_{RD}$**s were relatively small -- the average was 4 lines. This means that the only possible footprint ratios are 0, .25, .50, .75, and 1. These are big increments compared to those for potentials of say 10, 20, or 30. It is interesting to note that half of the **INIT$_{RD}$**

potentials are zero and the remainder ranged between 2 and 7. It turns out that in almost every case, the **RD**s were being used to retrieve some parameters from TS before performing some process. Therefore, the chance of variable conflicts occurring when footprinting is significantly reduced because there are very few variable references prior to the $\mathbf{INIT_{RD}}$. Another interesting point is that the only types of conflict that occurred with the $\mathbf{INIT_{RD}}$ were Linda conflicts. This is because, in several cases, **RD**s were grouped together.

Of the 179 footprints that had a potential greater than zero, there were 23 (13%) that maximized to the length of their potential. Therefore, roughly 1/8th of the footprints with non-zero potentials did not have any Linda, variable, or construct conflicts. Considering only the 155 footprints that did have a conflict occur, the following graph shows the distribution among the three types of conflicts.

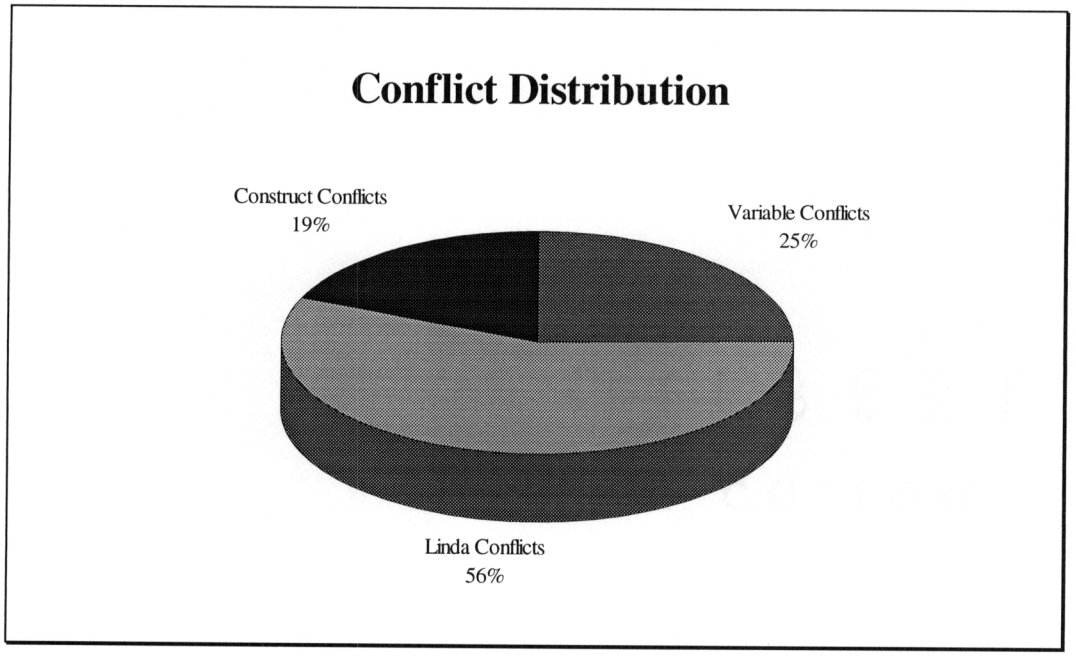


Figure 5-1. Conflict distribution for the basic footprint model.

It is interesting to note that well over half of the conflicts are Linda conflicts. There were three types of construct conflicts found in the analyzed programs. The conflicts found were with `IF`s, `FOR`s, and `WHILE`s. It is also helpful, not only to look at conflict distribution, but also the average footprint ratios recorded for each of the conflict types. The following table shows the five conflict types and their average footprint ratio.

Table 5-3. Number of conflicts and average footprint ratios by conflict type.

Conflict Type	Number of Conflicts	Average Footprint Ratio
IF	12	0.05
FOR	16	0.01
WHILE	1	0.00
Linda	88	0.11
Variable	39	0.14

These numbers show that the average footprint ratio for the 88 Linda conflicts is 0.11. The lowest footprint ratio is for the `WHILE` conflict and the highest is for variable conflicts. From the researcher's perspective, this table shows three possible paths for improvement to the basic footprinting model. The first is to take the conflict type with the largest number of conflicts in the table, the Linda conflict, and work to reduce this number. The other path is to take the conflict type with the smallest average footprint ratio, the `WHILE` conflict, and work to increase this number. The third path is to select a conflict type based upon a measure combining both the number of conflicts and the average footprint ratio. The following figure shows the *improvement measure* for each of the five conflict types.

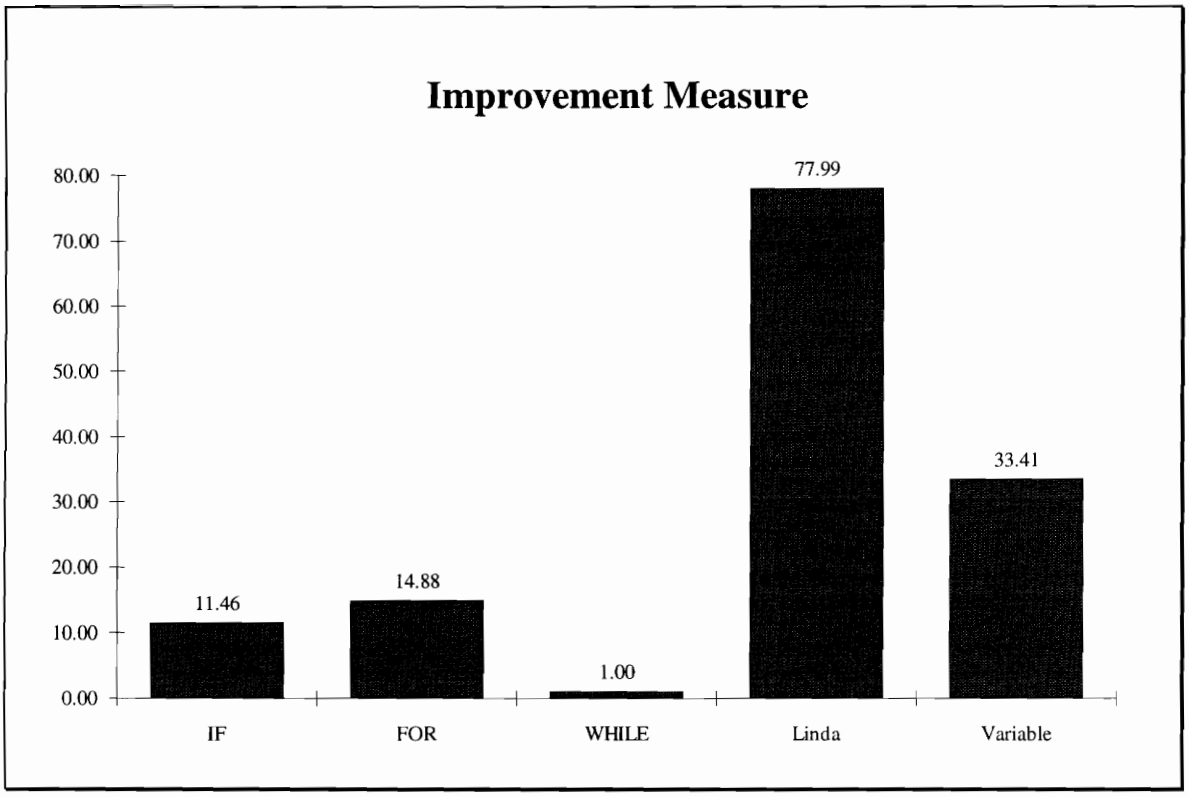


Figure 5-2. Improvement measure for each conflict type.

The graph above shows improvement measures based upon the following formula.

Improvement Measure = (Number of Conflicts * (1 - Average Footprint Ratio))

The higher the number, the higher the potential for improvement. The graph, therefore, indicates that improving the basic footprinting model with respect to Linda conflicts offers the researcher the best possibility for increasing footprint ratios. The next section discusses an improvement to the basic footprinting model called *Linda Primitive Transposition* (LPT). LPT addresses the issue of moving **INIT**s and **RECV**s safely past other Linda operations for the purpose of increasing footprint ratios.

5.2.2 Linda Primitive Transposition (LPT)

The goal of Linda Primitive Transposition (LPT) is to safely remove many of the soft boundaries associated with moving **INIT**s and **RECV**s for an **IN** and **RD** past Linda primitives (i.e. $\mathbf{INIT_{IN}}$, $\mathbf{RECV_{IN}}$, $\mathbf{INIT_{RD}}$, $\mathbf{RECV_{RD}}$, **OUT**, **EVAL**, **INP**, and **RDP**). A formal description of LPT is detailed in Chapter 4.

Several problems arise when **INIT**s and **RECV**s are naively moved past other Linda primitives, all of which relate to the altering of program semantics. The following example from Chapter 4 illustrates how program semantics can be potentially altered when a $\mathbf{RECV_{IN}}$ is moved down past an **OUT**.

```
RD("Data", ?x)                          INIT_RD("Data", ?x)
OUT("Data", y)        ========>         OUT("Data", y)
                                        RECV_RD("Data", ?x)
```

In the unoptimized code above on the left, the **RD** will wait for a "Data" tuple to arrive in TS if one is not present. It is impossible for the **RD** to be satisfied by the following **OUT** because the **RD** is a blocking primitive. However in the optimized version on the right, the $\mathbf{INIT_{RD}}$ is non-blocking. It is possible that the $\mathbf{INIT_{RD}}$ can be satisfied by the **OUT** when control is returned to the Linda program and the **OUT** is executed. Clearly, this is not the intent of the original program.

Preserving program semantics is a problem with nearly all transpositions of **INIT**s and **RECV**s with other Linda primitives. In fact, there are only three transpositions that can be safely performed. They are moving an $\mathbf{INIT_{RD}}$ or an $\mathbf{INIT_{RD}}$ up past an **EVAL**, and moving an $\mathbf{INIT_{RD}}$ up past a **RDP**. Only 3 out of 32 possible transpositions are safe; the other 29 can potentially alter program semantics. However, through the use of two code motion techniques called *Tuple Sequencing* and *Tuple Identification,* it possible to

increase the number of safe transpositions from 3 to 23. The two techniques are fully described in Chapter 4. Table 4-2 shows the transpositions that are semantically safe to perform.

As evident in Table 4-2, it is possible to increase footprint quantities in cases when the soft boundary is associated with a Linda conflict. In order to find out just what effect LPT has on improving footprint quantification, the basic footprint model was modified to include the LPT technique. The test programs were optimized again with the new footprinting process and data was collected about footprint ratios for the 224 optimized **IN**s and **RD**s. Table 5-3 shows new average footprint ratios for **INIT**s and **RECV**s.

Table 5-4. Footprint ratios for the LPT footprint model.

	INIT	RECV	INIT/RECV
IN	0.35	0.41	0.28
RD	0.80	0.29	0.34
IN/RD	0.44	0.38	0.30

When compared to Table 5-1, the table above shows good improvement in footprint ratios across the board. The average footprint ratios in Table 5-3 are 18% to 39% higher than those in Table 5-1. The biggest increase in average footprint ratios occurred with **INIT$_{RD}$**s. This is not surprising because the soft boundary encountered for the **INIT$_{RD}$** using the basic footprinting model was associated only with Linda conflicts. Again, the average ratios in Table 5-3 only include footprints where the potential is greater than zero (the ratios in the **INIT/RECV** column do not include footprint ratios for which the combined up and down potential is zero). Also, the numbers in the **IN/RD** row are a weighted average of the two numbers above it.

Using LPT, the number of cases where footprints reached their potential doubled from 13% to 26% of the 179 footprints having non-zero potential. The following figure shows the distribution among the three conflict types for the remaining 132 footprints that had a conflict occur.

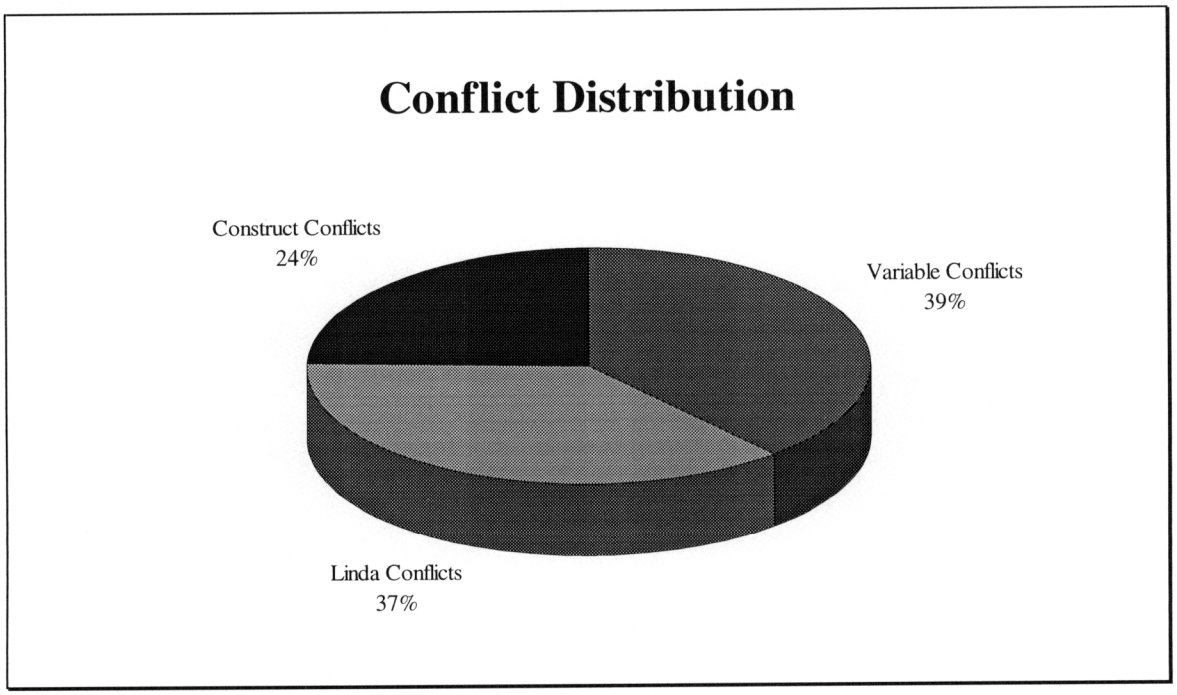


Figure 5-3. Conflict distribution for the LPT footprint model.

Note that the percentage of Linda conflicts dropped from 56% down to 37%. Even more compelling is the fact that 67 of the original 88 footprint ratios with Linda conflicts were increased through the use of LPT. This 76% improvement is consistent with the percentage of safe transpositions (see Table 4-2) which is 72%. It is also interesting to note that Linda conflicts for $\mathbf{RECV_{IN}}$s and $\mathbf{RECV_{RD}}$s were entirely eliminated due to

LPT. This is expected because Table 4-2 shows that it is safe to move **RECV$_{IN}$**s and **RECV$_{RD}$**s past any Linda primitive.

Another point of comparison is the average footprint ratios for each of the five conflict types. This information for the LPT footprint model is detailed in the following table.

Table 5-5. Number of conflicts and average footprint ratios by conflict type using LPT.

Conflict Type	Number of Conflicts	Average Footprint Ratio
IF	14	0.10
FOR	17	0.02
WHILE	1	0.00
Linda	49	0.36
Variable	51	0.20

Comparing these numbers to those in Table 5-2, notice that the number of footprints with Linda conflicts dropped a total of 39 (from 88 to 49) and the average footprint ratio increased from 0.11 to 0.36. Recall that 67 of the original 88 footprint ratios with Linda conflicts were increased with the use of LPT. Of these 67 footprints, 28 continue to have Linda conflicts for their soft boundary. Among the remaining 39 footprints no longer having Linda conflicts, 2 are now associated with IF construct conflicts, 1 with FOR construct conflicts, 12 with variable conflicts, and 24 maximized to their fullest potential. Notice that in every case, except for the `WHILE`, the average footprint ratio increased. This is due to the redistribution of the 39 larger footprint ratios affected by LPT.

The use of LPT in the footprinting process tripled the average footprint ratio for **IN**s and **RD**s. In addition, Linda conflicts were eliminated with respect to soft boundaries for **RECV**s. Consider now the improvement measures with the use of LPT in Figure 5-4.

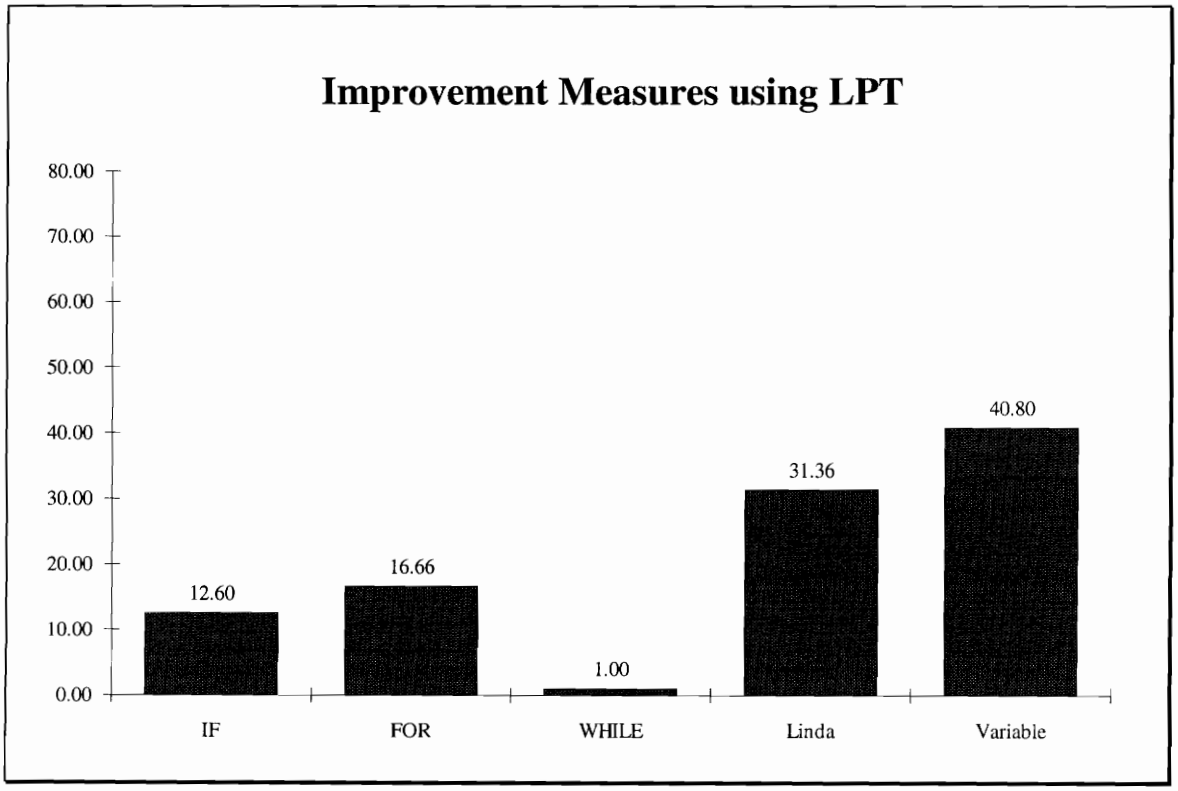


Figure 5-4. Improvement measures after using LPT.

The measures for `IF`, `FOR`, and `WHILE` have not changed significantly from those in Figure 5-2. However, the improvement measure for Linda conflicts was more than twice that for variable conflicts (which is what made it a clear choice for improvement before). The measure for variable conflicts (40.8) is now only slightly larger than that for Linda conflicts (31.36).

In the absence of a dominant improvement measure from which to work, another approach can be taken in finding ways to increase footprint ratios. Instead of identifying a select group of footprints that have the most potential for improvement, another approach is to apply improvement optimization across the board. In other words, the objective would be to increase the footprint ratios in each of the five conflict categories. This approach to footprint optimization is explored in detail in the following section.

5.2.3 Instruction Piggybacking

Fundamentally, there are only two types of conflicts: Linda and variable. The construct conflicts (IF, FOR, and WHILE) are simply constructs with component instructions that contain Linda and/or variable conflicts. Any across-the-board improvement made to increase footprint ratios has to be applicable to both Linda and variable conflicts.

Instruction piggybacking is an optimization technique for instructional footprinting aimed at increasing footprint ratios in all conflict categories without altering the soft boundary. To illustrate the idea, consider the following example.

```
while (x<100) {                          while (x<100) {
 x = x + 1;                                x = x + 1;
 y = y - 1;              ========>         y = y - 1;
 z = readvalue();                          z = readvalue();
 in("Data", z, ?item);                     init_in("Data", z, ?item);
}                                          recv_in("Data", z, ?item);
                                         }
```

Using the basic footprinting model, the **INIT$_{IN}$** in the above code could not be optimized. There is a variable conflict with `z` and therefore the footprint ratio is zero. However, the first two instructions of the `while` loop are not in conflict with the assignment of `z` or with the **IN**. A problem with the basic footprinting model is that it is short sighted. In other words, the footprinting process stops when the first conflict is

encountered. Instruction piggybacking looks past the first conflict and continues to try to optimize. For example, it is possible to continue to optimize the code above by continuing to move the **INIT$_{IN}$** and the assignment of z as a single instruction. The read and write sets of the new instruction are now the combined read and write sets of the two individual instructions, respectively. This piggyback instruction can then be moved past the first two instructions in the `while` because no conflicts exist.

```
while (x<100) {                              while (x<100) {
 x = x + 1;                                    z = readvalue();
 y = y - 1;                  ========>         init_in("Data", z, ?item)
 z = readvalue();                              x = x + 1;
 in("Data", z, ?item);                         y = y - 1;
}                                              recv_in("Data", z, ?item);
                                             }
```

The footprint ratio has now risen from 0 to 0.66 through the use of instruction piggybacking. In this example, piggybacking occurred once with the **INIT$_{IN}$** and the assignment of z. However, the new piggyback instruction could have encountered another conflicting instruction along the way to the hard boundary. If this had happened, then the piggyback instruction would be combined with the new conflicting instruction and the two moved as one instruction. The process of piggybacking and moving continues until the hard boundary is reached.

In the basic footprinting model, **INIT**s and **RECV**s are not allowed to be moved past Linda primitives in order to preserve program semantics. This is still true with the use of instruction piggybacking. No matter how many instructions an **INIT** or a **RECV** are piggybacked onto, the semantics of the piggyback instruction still includes the semantics of the **INIT** or **RECV** and therefore cannot be moved past other Linda primitives without the use of LPT.

To show the effectiveness of instruction piggybacking on increasing footprint ratios, this optimization technique was added to the basic footprinting model and footprint statistics were collected on the same set of Linda programs. Table 5-5 shows the new footprint ratios.

Table 5-6. Footprint ratios for the piggyback footprint model.

	INIT	RECV	INIT/RECV
IN	0.25	0.21	0.15
RD	0.47	0.14	0.15
IN/RD	0.29	0.19	0.15

The average footprint ratios improved in every category. In addition, nearly 30% of the footprints in the averages above were optimized due to instruction piggybacking. Notice, in Table 5-6 the number of conflicts in each category remained the same compared to the basic footprinting model. The footprint ratios in the different categories are what changed.

Table 5-7. Number of conflicts and average footprint ratios by conflict type using instruction piggybacking.

Conflict Type	Number of Conflicts	Average Footprint Ratio
IF	12	0.07
FOR	16	0.04
WHILE	1	0.00
Linda	88	0.21
Variable	39	0.16

Although the use of instruction piggybacking increased nearly 30% of the footprint ratios, this optimization technique did not come near the results of LPT. The reason lies in the basics of the technique. LPT concentrated on increasing footprint ratios by breaking down conflicts between instructions. In other words, it allowed **INIT**s and **RECV**s to be safely moved past other Linda primitives through the use of Tuple Sequencing and Tuple Identification. Instruction piggybacking, on the other hand, takes the approach of generalizing the basic footprint process by allowing movement of the **INIT** or the **RECV** to continue even after the first conflict is found.

This distinction is important in understanding the fundamental difference between the two optimization techniques. With instruction piggybacking, there is an effect of *diminished returns* as instructions get piggybacked. Realize that whenever two instructions are piggybacked the resulting instruction is potentially harder to move toward the hard boundary. This is because every time piggybacking occurs, the subsequent movement of the new piggyback instruction is effectively the movement of two instructions. This substantially decreases the mobility of the new instruction. This is precisely why LPT (which does not suffer from diminished returns) provides better footprint ratios than instruction piggybacking.

The previous two sections described two different types of optimizations to the basic footprint model. The optimizations were applied in isolation in order to measure their effectiveness. The next section presents results of applying both optimizations to the footprinting process.

5.2.4 The Complete Model

The two optimizations, LPT and instruction piggybacking, work to increase footprint ratios from two different perspectives. LPT works at *removing* soft boundaries, that is

eliminating certain instruction conflicts. Instruction piggybacking directs its attention at *extending* soft boundaries by generalizing the footprinting process.

The simple fact that these two optimizations work so differently is the exact reason why they are able to work together. Consider the basic footprint model with LPT added to it. This new model has far fewer rules associated with Linda conflicts. Now suppose we add instruction piggybacking which produces a complete model with both optimizations. The difference between this model and one with only instruction piggybacking is the need for piggybacking because Linda conflicts occur less frequently. This is because LPT has reduced the number of conflicts to a minimum. The following table shows average footprint ratios for the complete model.

Table 5-8. Footprint ratios for the complete footprint model.

	INIT	RECV	INIT/RECV
IN	0.39	0.48	0.32
RD	0.80	0.36	0.40
IN/RD	0.47	0.45	0.34

Notice that the ratios above are greater than or equal to those for the two individual optimizations. This should be of no surprise considering that the two optimizations are complimentary in nature. However, being complimentary does not mean the effects of the two optimizations are disjoint. In other words, the net improvement for both optimizations is not the sum of the individual improvements. Consider how instruction piggybacking works. The optimization is applied only when a conflict occurs when an instruction is being moved toward a hard boundary. So, the benefits are seen when the conflicting instructions are piggybacked and their movement toward the hard boundary

continues. LPT, on the other hand, is proactive in the sense that it actively breaks down conflict barriers between Linda instructions. This has an effect on the application of instruction piggybacking. The fewer the conflicts, the less the need to invoke instruction piggybacking. It was our experience that LPT, in many cases, eliminates the need to perform instruction piggybacking because by removing certain Linda conflicts the **INIT/RECV** was free to move all the way to the hard boundary. In these cases (and other similar ones), the sole effect of LPT in a sense subsumes the effect of instruction piggybacking.

LPT, by reducing the number of times piggybacking occurs, can also increase the effectiveness of instruction piggybacking . Consider the following code segment.

```
if (xpos <> 0)
  rd("Data", xpos, ?x);
  x++;
  rd("Data", ypos, ?y);
  y++;
}
```

Suppose, we want to footprint the first **RD** operation. In particular, we want to move the **RECV** for the first **RD** down in the code as far as possible. Using the footprint model with LPT, the $\mathbf{RECV_{RD}}$ cannot be moved at all because of a variable conflict with x++. Using the basic footprint model with instruction piggybacking, the $\mathbf{RECV_{RD}}$ gets piggybacked with x++ because of a variable conflict. Piggybacking occurs again with the second **RD** because of a Linda conflict. The next instruction is piggybacked because it has a variable conflict not with the original $\mathbf{RECV_{RD}}$ but with the second **RD** that was piggybacked. The net effect is that the $\mathbf{RECV_{RD}}$ cannot be safely moved. Using instruction piggybacking in the presence of LPT, the $\mathbf{RECV_{RD}}$ still gets piggybacked with x++, but it is able to be moved past the **RD** through the use of LPT. In addition, there is now no conflict with the

last instruction because the second **RD** was not piggybacked. The net effect is that the $\mathbf{RECV_{RD}}$ (along with x++) is able to be moved down past two instructions to the bottom of the IF.

An interesting caveat to footprinting with both optimizations involves the question of safety when moving Linda primitives past other Linda primitives. Using LPT in the presence of instruction piggybacking requires that the safety of moving any Linda primitive (not just **INIT**s and **RECV**s as is addressed with LPT) past any other Linda primitive be considered. Consider the following example.

```
if (condition) {
  eval(worker());
  out("Worker", total);
  in("Worker Result", ?result);
}
```

In moving the **INIT** for the **IN** up in the code, it is piggybacked with the **OUT** because there is a Linda conflict. Next, the question of whether the piggyback instruction (the $\mathbf{INIT_{IN}}$ and the **OUT**) can be moved past the **EVAL**. With respect to the $\mathbf{INIT_{IN}}$, the answer is yes. With respect to the **OUT**, the answer is maybe. Recall, LPT only addresses the movement of **INIT**s and **RECV**s past Linda primitives. In our experiments, it was implicitly assumed that any Linda primitive transposition is considered *unsafe* unless explicitly determined otherwise. The only safe movements that were allowed appear in Figure 5-5. Therefore in the code above, the piggybacked instruction could not move past the **EVAL** because it is assumed unsafe to move the **OUT** past the **EVAL**.

5.3 The Programmer's Perspective

From the researcher's perspective, the footprinting process can be optimized through the use of LPT and instruction piggybacking. There is another perspective in achieving

higher footprint ratios; this is the perspective of the programmer. This section addresses issues surrounding the question - *What can the Linda programmer do or not do to achieve higher footprint ratios?* The suggestions presented in this section pertain to programming styles and techniques and are based upon intuition, common sense, and informal results obtained by combing through thousands of lines of Linda code.

The results presented in the previous section are based upon a collection of Linda programs collected across internet. After examining these programs and performing many different variations of instructional footprinting, a number of programming observations were made with respect to increasing footprint ratios. These observations can be summarized into four categories.

1) The use of local/global variables.

2) The use of Linda primitives.

3) The use of `RETURN`, `BREAK`, `EXIT`, and `CONTINUE`.

4) The size of control structures.

The first observation deals with the use of local and global variables. The use of local variables (or the reduced use of globals) can lead to better footprint ratios. The use of global variables increases the coupling (i.e., conflict) between functions. This can inhibit footprint ratios by blocking the movement of **`INIT`**s and **`RECV`**s past function calls due to (global) variable conflicts.

The second observation addresses the use of Linda primitives and the negative impact it has on footprint ratios. In the 12 programs examined, only 42% of the functions contained an **`IN`** or a **`RD`** to be optimized. In fact, 80% of the Linda primitives were

found to be in only 30% of the functions examined. These statistics point to the fact that Linda primitives are often used in groups. As seen in Section 5.2, Linda primitives can have a negative effect on the movement of **INIT**s and **RECV**s. The use of LPT can significantly reduce the number of Linda conflicts that arise, but not completely. In fact, using both LPT and instruction piggybacking raises issues associated with moving any Linda primitive past any other Linda primitive. The bottom line is that, although LPT can reduce the negative impact that groups of Linda primitives have on footprinting, it cannot eliminate them. By reducing the number of Linda primitives used in a function, the ability to achieve higher footprint ratios is strengthened.

Another observation made with respect to programming style impacting footprint ratios has to do with the use of control instructions - RETURN, BREAK, EXIT, and CONTINUE. All four of these control instructions break normal control flow of a program and therefore cause conflicts in footprinting. Although the use of these control instructions affected only a few footprints, the effect was severe in most cases. Instruction piggybacking, in particular, was affected the most. Because moving a control instruction around is unsafe, piggybacking onto a control instruction is fruitless. By reducing the use of control instructions, optimizations such as instruction piggybacking can be maximized and larger footprint ratios can be produced.

A final observation addresses the size of control structures used in a function. In particular, **INIT**s and **RECV**s have less of a tendency to conflict with control structures that are small in size. This observation, as witnessed in the suite of Linda programs examined, seems reasonable considering that it only takes one Linda or variable conflict in a control structure to cause a control structure conflict with an **INIT** or **RECV**. The larger the control structure, the larger the probability a conflict exists. This is of

particular importance because, with respect to moving an **INIT** or **RECV** past a control structure, it is all or nothing. A conflict with a large control structure has a larger impact on the footprint ratio than does a conflict with a small control structure. By keeping control structures reduced in size (e.g., replacing a large if statement with several smaller ones), it is possible to increase footprint ratios.

With the exception of control instructions, the observations made in this section relate directly to the types of soft boundary conflicts - Linda, variable, and control structure conflicts. In an attempt to quantify the effect of the three main factors affecting the process of footprinting, a regression analysis was performed based upon the basic footprinting model. For each footprint ratio, the following data were collected and used in the regression analysis:

- the number of Linda conflicts,
- the number of variable conflicts, and
- the number and size of control structure conflicts.

After performing several transformations on the dependent and independent variables, the largest R^2 value obtained was 0.40 (a value in the 0.70 or 0.80 range was expected). Two reasons come to mind that can explain the low R^2 value. The first is that the position of conflicts relative to the instruction being footprinted was not included in the statistical model. Using positional information may strengthen the model because conflict positions determine the footprint for an **IN** or a **RD**. The second reason addresses the fact that, in the basic footprint model, one and only one conflict uniquely determines a footprint ratio. If for a footprint ratio, there exists a total of 10 conflicts, only the one conflict closest to the instruction being moved determines the footprint ratio. The fact that only one conflict

directly determines the footprint ratio can have an impact on the results of the regression analysis. Although the results of the regression analysis were not as expected, it did shed light on how much of an impact that conflict positions and their singular effect have on footprint ratios.

5.4 Summary

Instructional footprinting is a model for exploiting asynchronous speedups in Linda programs. The model is based on decomposing **IN** and **RD** primitives into **INIT** and **RECV** component instructions and then migrating these instructions. The **INIT** is executed as early as possible, while the **RECV** is moved forward in the code as far as possible. The amount of movement is called the footprint of the **INIT** or **RECV**. By separating the **INIT** and **RECV** for an **IN** or **RD**, the request can be serviced by the TS manager in parallel with normal computation of the requesting Linda program.

There are two key factors in determining the amount of speedup a Linda program experiences with instructional footprinting. These factors are the *quantity* and *quality* of the footprints. Footprint quantity refers the size (in number of instructions) of the footprint, while quality refers to the amount of speedup attributed to the individual instructions in a footprint.

This chapter addresses the issues surrounding footprint quantity from the researcher's and programmer's perspective. From the researcher's perspective, experiments were run using the basic footprint process as well as using two improvements to the process called Linda Primitive Transposition (LPT) and Instruction Piggybacking. Results showed average footprint ratios of 0.17 for the basic footprinting model, 0.41 using LPT, 0.24 using Instruction Piggybacking, and 0.46 using both LPT and Instruction Piggybacking.

This chapter also discussed several observations from the programmer's perspective about how programming styles can affect the size of footprint ratios. These observations dealt with how minimizing the use of global variables, Linda operations, control instructions, and large control structures has a tendency to increase footprint ratios.

CHAPTER 6 - Footprint Quality

6.1 Introduction

Recall that the goal of Instructional Footprinting is to reduce the execution time of Linda programs. This is accomplished by parallelizing the execution of **IN** and **RD** requests made to the TS manager (a separate process on another processor) with normal, useful computation of the requesting Linda process. The parallelism comes from initiating the **IN** or **RD** early in the code (non-blocking) and receiving the tuple data later in the code when it is needed (blocking). This span of code is called the footprint of the **IN** or **RD**.

Footprints can be measured with two different yardsticks. The first is how big the footprint is, i.e. it's *quantity*. The other yardstick is the *quality* of the footprint. The goal of footprint quantity is to maximize the size of each footprint so as to increase the potential for parallelism. The goal of footprint quality is to identify the contribution (individually and in tandem with other instructions) different instructions make to the speedup delivered by a footprint. The previous chapter discusses footprint quantity and related issues such as Linda Primitive Transposition (LPT) and Instruction Piggybacking. This chapter discusses in detail the goals and issues surrounding the measurement of footprint quality.

The measurement of a footprint's quality requires that two issues be addressed. The first is the *quality extent* of a footprint. *Extent* addresses the fact that there are many factors which contribute to the quality (or speedup) of a footprint. For example, one factor is understanding that all instructions are not created equal. In other words, a footprint containing 10 assignment statements is not of the same quality as a footprint with 10 Linda **EVAL**s. Different instructions have different execution speeds than other instructions and therefore will contribute in varying amounts to the quality of a footprint and hence to the speedup of a program. Another factor contributing to quality extent is

looping found in footprints. Often it is unknown how many times the instructions of the loop body are to be executed. This affects the extent of a footprint's quality. One way to measure some of the factors of quality extent is to rate the quality that is contributed by different instructions. Section 6.1 discusses quality ratings for instructions and how it can be used.

The second issue surrounding the measurement of footprint quality is the effect of, what I call, the *Lexical Proximity* of **INIT**s on execution speedup. Substantial performance improvements are witnessed when **INIT**s are moved past **RECV**s (or visa versa) so that groups of **INIT**s are executed in close proximity to one another (in other words, there are no **RECV**s intermingled). What is unique about lexical proximity is that there is overlap of footprints. This overlap, in conjunction with the multi-processing characteristics of networks, allows for dramatic speedups in Linda programs. Section 6.3 discusses the details of lexical proximity, when it happens, and why.

6.2 Instructional Quality Ratings

As mentioned before, not all instructions are created equal. Some instructions such as a simple assignment statement execute extremely fast compared to an **IN** operation that must travel across the network to retrieve a tuple. Knowing the relative "quality" of an instruction can be beneficial in many respects.

First of all, the assignment of quality ratings to instructions creates a quality yardstick by which footprints can be measured and compared. Looking at the size or quantity of footprints is not sufficient for a quality comparison. The assignment of ratings to each instruction within a footprint allows one to gauge the relative effectiveness of a particular footprint. In some cases, one can calculate what speedup is contributed by a footprint.

Why is it important to be able to say something about the potential speedup contributed by a footprint? Not all programs that are footprinted can or will be run. One might want to run a series of programs through the footprinting process for the purpose of analysis. Without running the programs to observe resultant speedup, the results of footprinting (i.e., the optimized source code) can be analyzed for potential speedup by looking at the quality ratings of instructions in the footprints.

Another goal in developing a quality rating scheme is to provide the Linda researcher with a tool that can be used in deciding where the fruitful paths are for future research. For example, if a researcher wants to increase the quality of instructional footprints then quality ratings can be used as a *measurement of potential* indicating that it is more beneficial to have Linda operations in a footprint than computational instructions.

It is also possible to apply quality ratings to the footprinting process. This new information can be used to detect the point at which adding more instructions to a footprint provides no more speedup. This break-even point comes when the time it takes to execute the **IN** (or **RD**) is equaled by the processing time within the footprint. Knowing the quality ratings of instructions, a better decision could be made if a choice has to be made as to what goes in a footprint.

Quality ratings are obtained by recording timings on a representative set of instructions. These timings are system/environment dependent in many respects. First of all, the timings are dependent on the efficiency of the C compiler and of the Linda system. Second, Linda programs were run on a network of two machines. However, we would expect the quality ratings to be similar on different machines or network configurations. One machine is used to execute all Linda processes and the second is used to run the TS manager and house TS itself. The two machines are Commadore Amigas running Unix

System V Release 4. Communication between the Linda processes and the TS manager is accomplished through the use of sockets at the TCP/IP level and data is transferred without delay.

Another system dependency is the timing facilities provided by the machines. The resolution of the timings is to the microsecond but the granularity is 0.016666 seconds. Therefore, timings can only be recorded in increments of 0.016666 seconds. The limitation of the timing granularity introduces an error factor into the recording of the instruction timings. In order to minimize the effect of the granularity error, timings for multiple instructions were taken and then averaged. This has the effect of minimizing the granularity error to the point where it is negligible. By recording and averaging the execution times of multiple instructions, the granularity error was reduced to less than 2% of the individual instruction time.

Timings are recorded for four categories of instructions - integer manipulation, floating point manipulation, function calls, and Linda operations. Ratings are assigned to instructions relative to the fastest instruction. Therefore, integer and floating point assignment (the fastest instruction recorded) has a rating of 1.0. All other quality ratings are a factor relative to the speed of an assignment statement. The formula for the quality ratings of instruction A is simply ($Time_A$ / $Time_{Assignment}$). The instructions that execute slowly produce a high quality rating. Therefore, quality of an instruction is directly proportional to its speed. The following table shows the timings and quality ratings for instructions in the four different categories.

Table 6-1. Instruction Timings and Quality Ratings.

	Integer		Floating Point	
Instruction	Timing	Rating	Timing	Rating
Assignment	0.000000682	1.00	0.000000682	1.00
Addition	0.000001096	1.61	0.000005099	7.47
Subtraction	0.000001099	1.61	0.000006024	8.82
Multiplication	0.000002986	4.37	0.000005102	7.47
Division	0.000004916	7.20	0.000008095	11.85
Casting	0.000005281	7.73	0.000004299	6.30
Printf	0.005233323	7662.34	0.005433334	7955.18

(a)

	Function Calls	
Parameters	Timing	Rating
None	0.000009066	13.27
3	0.000010406	15.24

(b)

Linda Operation	Timing	Rating
INIT	0.004899999	7174.30
RECV	0.004533333	6637.46
IN	0.187000000	273795.10
RD	0.188333400	275747.39
INP	0.186666680	273307.07
RDP	0.189333340	277211.45
OUT	0.003055557	4473.78
EVAL	0.102333340	149830.84

(c)

In Table 6-1a, assignment statements (both integer and floating point) rated the fastest, posting a time of less than a microsecond. The other computational instructions such as addition, subtraction, and multiplication also involve assignment as well. The integer casting is an integer assignment of a float being casted to an integer. The `printf` printed out (to stdout) the string "Data:" followed by two integers (or two floating points). The quality ratings for integer and floating point manipulation are not surprising. However, it is interesting to note that the amount of time it takes to perform a `printf` is seven to eight

thousand times that of a simple assignment statement. This would indicate that including device I/O statements in a footprint is more beneficial than simple computations. However, including I/O statements in footprints may change program semantics in a unique way. Consider the following example.

```
:
:
:
IN("Account Balance", "Johnson", ?balance);
printf("Account Balance for Johnson has been retrieved\n");
:
:
:
```

Suppose that the **IN** is decomposed into its component **INIT** and **RECV** instructions and the **RECV** is moved down past the `printf`. When the program is run, the `printf` would be executed and indicate to the user that the account balance has been retrieved when in fact it may still be waiting for the balance tuple to be placed in TS. This alteration of semantics is only temporary until the tuple is actually returned. The issue of when and how **INIT**s and **RECV**s can be safely moved past I/O statements is not addressed in detail in this report. The quality rating of `printf`s, however, does indicate that it would benefit execution speedup by having I/O statements in footprints. This is an example of how future research can be directed in potentially fruitful directions with the use of quality ratings.

Table 6-1b shows timing and ratings for two function calls - one with no parameters and the second with three parameters. The function calls have empty bodies and therefore the timings indicate the *overhead* of a function call. The timings indicate that the overhead of a function call is just slightly greater than a floating point division. Obviously, what matters is the amount of time it takes to execute the body of a function and not the

overhead of making the call. Depending on the function, it may be possible to compute a timing for a function and therefore the quality rating for a particular function can be known and used at compile time.

The most interesting information is in Table 6-1c which has to do with the ratings for Linda operations. The **IN**, **RD**, **INP**, and **RDP** all rated about the same. This is because the timings for all retrieval operations were performed with matching tuples present in TS. It would take the execution of about 275,000 assignment statements to equal the time it takes to execute one tuple retrieval operation. The primary reason for such a rating discrepancy is due to the network. Recall that TS and the TS manager both reside on a machine separate from that of the Linda processes. Each tuple request sends a tuple template across the network to the TS manager and then the matched tuple is sent back across the network to the requesting Linda process. This round trip consumes most of the time needed to perform an **IN**, **RD**, **INP** or a **RDP**. Each **IN** operation takes about 0.19 seconds and about 0.010 seconds total to perform an **INIT** and a **RECV**. The timings for a **RECV** reflect the fact that the returned tuple has already been sent back and is waiting to be received by the requesting Linda process. This means that it takes 0.18 seconds for the tuple template to reach the TS manager, the tuple to be taken out of TS, and sent back to the requesting process. This makes up 95% of the processing time for an **IN**. Because TS is partitioned into separate tuples spaces, the code to search for and retrieve a tuple in TS is rather straightforward and relatively efficient. The network, therefore, is the primary source of processing time when it comes to tuple retrieval operations.

Notice that an **OUT** is significantly faster than an **IN**. This is expected because an **OUT** is a non-blocking operation unlike the **IN** operation and therefore simply sends a tuple across the network to the TS manager and does not wait for a reply.

The execution of an **EVAL** also exhibits a high quality rating, roughly 150,000. The significant processing performed by an **EVAL** is the creation of a new process. The timing reflects only the creation of the new process and not the time it takes to execute the **EVAL**ed process.

Linda operations are the slowest instructions timed and hence recorded the highest quality ratings because they offer the best potential for program execution speedup when placed inside footprints. However, it is not sufficient to just look at the timings of footprint instructions when trying to determine the processing time of a footprint. As mentioned before, there are some unknowns. One of the biggest unknowns is the use of conditional and looping control structures inside footprints.

It is difficult to determine the exact processing time of a footprint when IF statements are used because it cannot (in most cases) be determined at compile time which path will be taken. The best that can be done is to make an educated guess at the percentage of times the THEN path will be take as apposed to the ELSE path.

There is a similar problem with looping constructs such as WHILEs, FORs, and REPEATs. The unknown is the number of times the loop will be processed. It may be executed zero times or it may be an infinite loop with BREAK or EXIT statements inside to break out of the loop. Although it would be advantageous to always compute an number for the processing time or quality rating of a footprint, it is not always possible. It is possible, however, to derive an expression describing the processing time or quality of a footprint. Suppose we use the quality ratings in Table 6-1 to describe the speed of individual instructions. Furthermore, let us assume for the sake of simplicity that THEN and ELSE paths on the IF are executed equally. If we use variables for the amount of iterations a

loop will execute, then it is possible to describe in simple terms the "quality" of a footprint. For example, suppose the following is the code inside a footprint.

```
x = 1;
y = y * y;
for (i=1; i<=k; i++) {
  x = x + 1;
  for (j=1; j<=k; j++) {
    y = y / 2;
  }
}
```

A quality expression for the footprint can be written as:

$$footprint_{quality} = 1 + 1.61 + k\ (\ 1.61 + k\ (\ 7.20\ )\ )$$

or more succinctly as:

$$footprint_{quality} = 7.20k^2 + 1.61k + 2.61$$

where k is the number times that the two loops will be executed. Being able to derive quality expressions for footprints allows one to analyze footprints without having to execute the footprinted programs and observe THEN/ELSE percentages, loop iterations, and program speedup.

Instructional quality ratings address the issues surrounding the fact that not all instructions contribute the same to speedup within a footprint. The next section addresses another issue of quality with respect to footprints - the order of instructions in a footprint.

6.3 Lexical Proximity

The idea behind lexical proximity is that the order of instructions in a footprint is just as important, if not more, than which instructions that comprise the footprint. More

specifically, the order of **INIT**s and **RECV**s within a footprint is particularly important. Dramatic reductions in execution times can be obtained when **INIT**s are in lexical proximity. In other words, it is beneficial to have **INIT**s executed close together and not intermingled with **RECV**s.

The reason why dramatic performance improvements can be made by executing **INIT**s in succession (or at least without any **RECV**s intermingled) is due to the network. It is interesting to note that the network plays such a significant role in slowing down Linda operations because of the time it takes to transfer data as compared to the computational time of a CPU. Because it is so slow relative to CPU speed, the network is the source of the greatest performance improvements with instructional footprinting.

By exploiting the multi-processing characteristics of the network, the blocking time of **RECV**s is reduced and in many cases eliminated. Consider the following simple segment of Linda code.

```
IN("Name",  SSN, ?name);
IN("Addr",  SSN, ?addr);
IN("City",  SSN, ?city);
IN("State", SSN, ?state);
```

Suppose the above **IN** operations were decomposed into their respective **INIT**s and **RECV**s. Furthermore, suppose that the first **IN** is footprinted so that the **RECV** is moved down past the next two **IN**s. The following is the resulting optimized code.

```
INIT("Name",  SSN, ?name);
INIT("Addr",  SSN, ?addr);
RECV("Addr",  SSN, ?addr);
INIT("City",  SSN, ?city);
RECV("City",  SSN, ?city);
INIT("State", SSN, ?state);
RECV("State", SSN, ?state);
RECV("Name",  SSN, ?name);
```

Consider what happens when this segment of code is executed and all 3 tuples are present in TS. First, the request for the "Name" tuple is sent across the network to the TS manager. Second, the request for the "Addr" tuple is sent followed by the execution of a blocking **RECV** for the "Addr" tuple. Once the "Name and the "Addr" tuples are sent back, the blocking **RECV** for the "Addr" tuple is satisfied and the request for the "City" tuple is sent out and the corresponding **RECV** blocks while waiting for it to return. The request is then sent out for the "State" tuple and the corresponding **RECV** blocks while waiting for it to return. Finally, the **RECV** picks up the returned "Name" tuple that has already been sent back. This **RECV** does not block and therefore program execution is sped up. Is this the best that can be done?

The answer is NO. There is still wasteful blocking that is being performed by the other 3 **RECV**s. Suppose that the footprint is reordered by performing the footprinting process on the 3 remaining **IN** operations. Because there are no variable conflicts and LPT (Linda Primitive Transposition) allows **INIT**s and **RECV**s to be swapped, the footprint code can be reordered to look like the following.

```
INIT("Name",  SSN, ?name);
INIT("Addr",  SSN, ?addr);
INIT("City",  SSN, ?city);
INIT("State", SSN, ?state);
RECV("Name",  SSN, ?name);
RECV("Addr",  SSN, ?addr);
RECV("City",  SSN, ?city);
RECV("State", SSN, ?state);
```

For simplicity of explanation the **RECV**s are placed in the same order as the **INIT**s. The **RECV**s can be executed in any order with the use of Tuple Identification. Consider what happens when the code above gets executed and TS holds all four tuples. All four **INIT**s are executed (without blocking) one right after another. The first **RECV** is executed and blocks until the "Name" tuple is returned. The next **RECV** is executed to retrieve the "Addr" tuple. The blocking time for this **RECV** is minimal (if any at all) because the "Addr" tuple is right behind the "Name" tuple. The "City" and "State" tuples are following behind as well. Remember, the significant amount of time spent on a tuple retrieval is the voyage back and forth across the network (see Figure 6-2). Because all four tuples can pass over the network at the same time, the usual blocking time of each **RECV** (when **INIT**s are performed one at a time) can now be mostly consumed by one **RECV**.

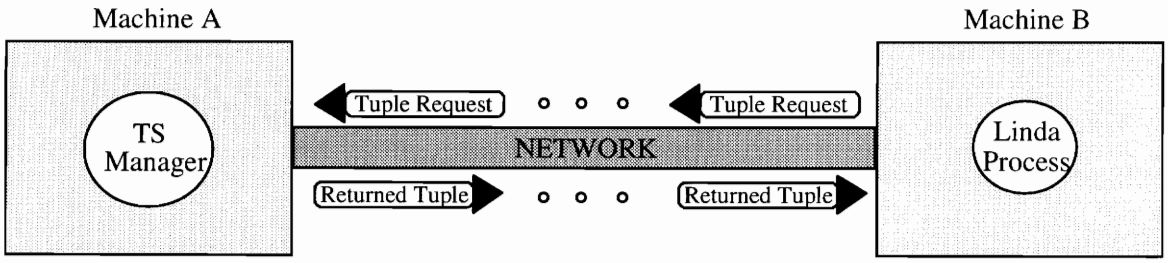


Figure 6-2. Multi-processing characteristics of the network in a Linda environment.

It is interesting to note that the significant speedup due to instructional footprinting is attributed to exploiting the multi-processing characteristics of the network and **not** the exploitation of computational concurrency between a requesting Linda process and the TS manager. Although the network imposes the biggest performance penalty for a Linda program, it is also the greatest source of improvement for instructional footprinting.

To illustrate the utility of lexical proximity, a simple Linda program is constructed to show the potential for speedup by exploiting the inherent characteristics of the network. The code for the Linda program is show below.

```
main() {
  OUT("Data", i, j);                INIT("Data", i ,j);
  OUT("Data", i, j);                INIT("Data", i ,j);
  OUT("Data", i, j);                INIT("Data", i ,j);
  OUT("Data", i, j);                RECV("Data", i ,j);
  OUT("Data", i, j);                timer_split("After 1st RECV");
  OUT("Data", i, j);                RECV("Data", i ,j);
  OUT("Data", i, j);                RECV("Data", i ,j);
  OUT("Data", i, j);                RECV("Data", i ,j);
  OUT("Data", i, j);                RECV("Data", i ,j);
  OUT("Data", i, j);                RECV("Data", i ,j);
  start_timer()                     RECV("Data", i ,j);
  INIT("Data" i, j);                RECV("Data", i ,j);
  INIT("Data" i, j);                RECV("Data", i, j);
  INIT("Data" i, j);                RECV("Data", i, j);
  INIT("Data" i, j);                RECV("Data", i, j);
  INIT("Data" i, j);                timer_split("Finished");
  INIT("Data" i, j);                print_times();
  INIT("Data" i, j);              }
```

This simple program **OUT**s 10 tuples to TS and then performs 10 **INIT**s followed by 10 **RECV**s. Timings were taken after the first **RECV** is performed to show that it is blocking. A final timing is taken at the end of the last **RECV** to show the total time required to perform the 10 **INIT**s and 10 **RECV**s. The time required to perform the 10 **INIT**s and the one **RECV** was 0.06666 seconds. Given that an **INIT** takes 0.005 seconds (0.05 seconds for 10), it took 0.01666 seconds to perform the first **RECV**. A **RECV** takes about 0.005 seconds to execute if the tuple has already been received, therefore the blocking time of the **RECV** was 0.01166 seconds. What is significant here is that it took 0.12333 seconds to perform all the **INIT**s and **RECV**s. This leaves 0.05666 seconds to execute 9 **RECV**s. This is an average of 0.006 seconds; fairly close to the time it takes to execute a normal, non-blocking **RECV**.

The same program was executed without being optimized. That is, the **INIT**s and **RECV**s were executed in pairs. The unoptimized version executed the 10 **INIT/RECV** pairs in 1.93999 seconds. The speedup of the optimized version over the unoptimized version is a dramatic 93.6%.

To demonstrate the effect of Lexical Proximity on execution performance, experiments were performed using two machines networked together. One machine is used to execute all Linda processes and the second is used to run the TS manager and house TS itself. The two machines are Commadore Amigas running Unix System V Release 4. Communication between the Linda processes and the TS manager is accomplished through the use of sockets at the TCP/IP level and data is transferred without delay.

Several of the Linda programs used in the test bed described in Chapter 5 exhibited speedups due to lexical proximity. The following table shows execution times in seconds for a couple of these programs.

Table 6-2. Speedups of Linda programs through the use of Lexical Proximity.

Application	Original	Paired	Grouped
Banking Simulation	16.35	16.35	12.52
Raytrace	10.52	10.44	8.99
Raytrace (cleanup routine)	3.42	3.19	1.59

The first Linda program is a simulation of a distributed banking system. The simulation was run unoptimized (Original), optimized without the use of lexical proximity (Paired), and optimized with the use of lexical proximity (Grouped). Speedup occurred only when the **INIT**s were grouped together; speeding up execution by 23.43%. The second

program is a Raytrace system that generates traced images in Utah Raster RLE format. Two timings were recorded for this system. The first is for the execution of the entire system and the second is the execution of only the cleanup routine that collects statistics and data from its worker processes. The reason for two sets of timings is because most of the optimization occurred in the cleanup routine which only accounts for a small portion of the total execution time. Speedup is experienced both when lexical proximity is and is not used. However, the speedup was less than 2% for both the cleanup and the total run of the program. When the **INIT**s are grouped together, a speedup of over 15% results for total program execution and a speedup of over 51% for the cleanup routine.

The following table shows average footprint ratios and speedups for the 12 program test bed used in Chapter 5. Some programs, for various reasons, were not able to be executed. They are indicated with NA for percent speedup.

Table 6-3. Program speedups versus footprint ratios for the 12 program test bed.

Program	Average Footprint Ratio	Percent Speedup
DNA	0.17	NA
Dist. Memory Manager	0.68	NA
Boyer-Moore	0.34	0
Dist. Banking Simulation	0.46	23
Parallel FFT	0.37	0
Parallel Sort	0.18	0
Parallel Sum	0.12	0
Bug Trudger	0.16	0
Spanning Tree	0.00	0
Dining Philosopher	0.40	26 - 64
Raytrace (and cleanup routine)	0.28	15 (51)
Matrix Multiply	0.21	0

6.4 Summary

While Chapter 5 points out the need for large footprints and demonstrates techniques for achieving it, this chapter deals with the quality of instructions and footprints. There are two aspects of quality that are addressed. The first is the development of a quality rating scheme for instructions and the second is an explanation of the role that lexical proximity plays in obtaining performance speedups.

The development and use of an instructional quality rating scheme is advantageous for several reasons. First of all, a common yardstick can be used for measuring the speedup potential of footprints. This may be particularly useful when a footprint analysis of programs is needed to understand potential speedup and it is impossible to run the resulting optimized programs. Second, knowing the relative quality of different instructions helps the researcher decide which areas of research (such as developing techniques that allow more Linda operations to reside in footprints) are potentially more fruitful than others. Finally, the knowledge of instruction quality could be used during the footprinting process as a heuristic in determining the best path to take when producing a footprint. The notion of instruction quality may also be used to measure when a footprint is "good enough." In other words, when the point is reached when the addition of more instructions to a footprint adds no more speedup.

Lexical proximity looks at footprint quality from the point of view of instruction order. It turns out that the primary source of speedup comes from exploiting the multi-processing characteristics of the network. This is accomplished by, whenever possible, ordering the **INIT/RECV** pairs within a footprint so that **INIT**s are executed in succession followed by the execution of the **RECV**s. Speedup is obtained by allowing the first **RECV** to bear

the brunt of blocking for the rest of the **RECV**s. Enormous speedups can be achieved as evidenced by the banking and raytrace applications.

CHAPTER 7 - Conclusions and Future Research

7.1 Conclusions

The research presented in this report focuses on calls to functions that are implemented as separate processes. These calls are inherently synchronous in nature and, as such, cause the calling process to block while the independent process is servicing the request. Minimizing the wasteful blocking time of the calling process, and hence speeding up program execution, is the primary goal of this research effort.

The approach taken to minimize this unnecessary blocking time is to convert the synchronous calls to independent functions to asynchronous calls. By doing so, useful computation of the calling process can overlap the computation performed by the independent process in fulfilling the request. This is accomplished by recognizing the presence of calls to independent function calls (e.g., a call to the operating system to read a record from a file) and then automatically *initiating* the call earlier in the code (a non-blocking operation) and then *receiving* the returned data later when it is needed (a blocking operation). The span of code between the initiation of the request and receipt of the returned data is called the *footprint* of the function call.

There are two aspects of instructional footprinting - instruction decomposition and code motion. The first issue addresses how an independent function call is decomposed into two sub-operations - initiation and receipt. The initiation of the function call is a non-blocking operation that sends the request to the independent process. The second sub-operation receives the returned data and is blocking. The second aspect of instructional footprinting focuses on automatically moving the two sub-operations in opposite directions. The initiation is moved back in the code as far as possible, while the receipt is moved as far forward in the code as it can. This code motion is tempered with the need to preserve program semantics.

The goal of this research effort is two-fold:

1) Develop an instructional footprint model and application framework that can be used to determine footprints of any instruction, not just independent function calls, and

2) Augment the footprinting process with techniques and tools that will aid the researcher and programmer in producing programs that exhibit enhanced execution performance.

Instructional Footprint Model

The Instructional Footprint Model (IFM) presented in Chapter 3 is an abstract model that is applicable to many operational domains. A simplified language, called the Canonical Language (CL), is used in the model to capture the necessary aspects of a procedural language (such as conditionals, loops, and instructions). For the purpose of footprint determination, the only semantic information necessary for instructions needed in the IFM is knowing which variables are read from and written to by each instruction.

The IFM, which models procedural programs, is complemented with an application framework that provides the power in footprint determination. Included in the framework are functions used to aggregate and deaggregate groups of instruction for the purpose of simplifying the footprinting process. A detailed description of the footprint algorithm is given along with definitions for when conflicts between instructions exist. Finally, issues dealing with instructional footprinting in the presence of function calls, gotos, and pointers are addressed and solutions offered ranging from compile-time analysis to instrumenting the optimized code with run-time extensions. The IFM and its application

framework are important aspects of this research effort because they provide a domain-free model for reasoning about and determining the footprints of any type of instruction.

Applying Instructional Footprinting to the Linda Domain

In order to prove the effectiveness of instructional footprinting as an optimization technique, the IFM and its framework was applied to an application domain. The Linda language was chosen as the domain because of its criticized lack of performance due to its high-level approach. In Linda, the **IN** and **RD** primitives that retrieve tuples from Tuple Space (TS) are examples of independent function calls when TS is managed by a separate process (as is the case with our implementation of Linda). The mechanisms for initiating **IN**s and **RD**s (called **INIT**s) and receiving the returned tuple (called **RECV**s) were added as sub-primitives to the Linda language. In other words, the **INIT** and **RECV** primitives are available to the Linda compiler but not to the Linda programmer. The footprinting optimization was then added to the Linda compiler to automatically determine instructional footprints and perform code motion. The footprint optimization was applied to several "real world" Linda applications ranging from a raytracing system to a program that performs FFT calculations. The resulting speedups were recorded as high as 64% in some cases. However, in many cases the potential for speedup was not fully exploited because the IFM is designed to be used in all application domains.

The second goal of this research effort is to maximize the potential for execution speedup of Linda programs through the application of instructional footprinting. This is accomplished by equipping the basic footprinting process with techniques that are aimed at maximizing footprint *quantity* and *quality*. Footprint quantity refers to the number of instructions in a footprint, while footprint quality refers to the amount of speedup contributed by footprint instruction both individually and in groups. Maximizing

footprint quantity increases the amount of execution concurrency between Linda programs and the TS manager and therefore has the effect of increasing execution performance. Maximizing footprint quality addresses the fact that not all instructions are created equal. Some instructions in a footprint contribute more to speedup than others do. By knowing which instructions are the *performers*, the footprinting process can be optimized to identify opportunities where these instructions can be included in footprints.

Linda Primitive Transposition

Chapter 4 presents a technique called Linda Primitive Transposition (LPT) that increases both footprint quantity and quality. LPT addresses the issues surrounding the movement of **INIT**s and **RECV**s past other Linda operations and its resulting effect on program semantics. Through the use of Tuple Sequencing and Tuple Identification, LPT increases the percentage of safe movement from 9% (these are inherently safe movements) to 72%. The result is substantial increases in footprint quantity and quality which leads to significant execution speedups.

Footprint Quantity

As mentioned before, maximizing footprint quantities improves performance speedups by increasing the degree of concurrency between Linda programs and the TS manager. The issues surrounding footprint quantity are presented in Chapter 5. The goal of maximizing footprint quantity can be viewed from two perspectives - the Linda researcher and the Linda programmer. From the researcher's perspective, the footprinting process can be enhanced with techniques such as LPT and instruction piggybacking. The latter technique allows the soft boundary of a footprint to be pushed back by aggregating instructions that are in conflict and then moving the aggregated instruction toward the footprint's hard boundary. Together these two techniques increased footprint quantities

nearly 40%. The results presented in Chapter 5 show that enhanced speedups are indeed the result of increased footprint quantities.

From the programmer's perspective, programming styles that affect footprint quantity are discussed. The observations made with respect to programming style deal with the use of local/global variables, the use of Linda primitives, the use of instructions that break flow of control, and the size of control structures. The observations made in Chapter 5 allow the Linda programmer to take a proactive role in increasing the amount of speedup achieved through the use of instructional footprinting.

Footprint Quality

As described in Chapter 6, the measurement of a footprint's quality requires two issues to be addressed. The first is quality extent which addresses the many factors which contribute to a footprint's quality. The primary factor is the individual quality of a footprint's instructions. The second issue surrounding the measurement of footprint quality is the effect of lexical proximity of **`INIT`**s on execution speedup.

Instructional Quality Ratings directly address the primary factor relating to quality extent. Through the use of quality ratings, instructions can be rated relative to execution speed. Having knowledge of quality ratings for different instructions can be beneficial in many respects. First of all, it provides a means to perform footprint analysis and gauge results without having to execute the optimized programs. Second, instructional quality ratings provide the Linda researcher with a tool for measuring how fruitful different research paths are. Finally, it is also possible to use the knowledge of quality ratings in the footprinting process as a means of improving the quality of footprints. Results showed that the ratings for normal integer and floating point calculations are far outranked by I/O instructions and even more by Linda operations. The results of these ratings strongly

indicates that the path to increased footprint quality is through techniques such as LPT and instruction piggybacking that concentrate on including Linda operations in footprints.

The second aspect of footprint quality is the effect of lexical proximity of **INIT**s on execution speedup. The order of instructions within a footprint is just as important to speedup as the type of instructions that comprise the footprint. This is especially true of **INIT** operations. In unoptimized code, **INIT**s and **RECV**s are alternated where the **RECV** operation blocks awaiting the returned tuple. However, through the use of LPT, it is possible for **INIT**s to be moved past **RECV**s and **RECV**s to be moved past **INIT**s. This allows **INIT**s within a footprint to be executed together without intervening **RECV**s. By executing many **INIT**s together, they can be transported across the network to the TS manager at the same time. By exploiting the multi-processing characteristics of the network, significant speedups are realized. Results shown in Chapter 6 demonstrate the effectiveness of LPT and lexical proximity. Experiments recorded speedups over 50% in just a couple of Linda programs. The use of LPT in exploiting lexical proximity takes an important step towards the goal of maximizing the benefits of instructional footprinting.

7.2 Future Research

There are several areas where the groundwork of instructional footprinting can be extended. Many of the next steps are within the domain of Linda. For example, the Linda group in the CS department here at Virginia Tech has developed a network version of Linda that distributes **EVAL**ed processes to available workstations on a network. One next step is to incorporate the footprinting optimization into the next version of LindaLAN. There are many opportunities for exploiting concurrency in LindaLAN and thereby providing Linda programs with enhanced execution performance.

Eliminating Linda Conflicts

Other steps can also be taken to improve the footprinting process. The first is continuing the work of LPT by working to eliminate all Linda conflicts. Currently LPT eliminates 72% of the possible conflicts of moving **INIT**s and **RECV**s past other Linda operations. Future work can concentrate on the remaining 28% as well as address the safety in moving any Linda operation past any other Linda operation (which is an issue with instruction piggybacking).

Extending LPT

The notion of LPT and lexical proximity can also be taken one step further. It is not uncommon to see loops of **IN** operations as in the code below.

```
for (i=1; i<=10; i++)
   IN("Data Array", i, ?vector[i]);
```

The potential for improving execution performance in situations such as this is enormous. The problem is that the IFM in its current form cannot optimize the code above because the FOR acts as the hard boundary for the **IN**. However, it might be possible to recognize the situation such as the one above and split the FOR into two separate FORs - one for **INIT**s and one for **RECV**s. The resulting optimized code might look like the following.

```
for (i=1; i<=10; i++)
   INIT("Data Array", i, ?vector[i]);


for (i=1; i<=10; i++)
   RECV("Data Array", i, ?vector[i]);
```

By splitting the **INIT**s and **RECV**s into separate loops, the requests for tuples can travel across the network at the same time while reducing the blocking time for the **RECV**s in the next FOR loop. This exploits lexical proximity in another dimension.

Optimizing INPs and RDPs

This research effort concentrated on footprinting **IN**s and **RD**s only. Another research area is to extend the optimization to **INP**s and **RDP**s as well. This will allow for more potential in speedup by being able to decompose and move more blocking operations.

Breaking the Functional Boundary

The IFM defines functional boundaries as the outer limits of hard boundaries. In other words, footprints cannot span outside functional boundaries. One future direction can be to improve the IFM by allow footprints to step outside of functions and thereby increase footprint quantities. This will increase resultant speedups by allowing more instructions to be incorporated into footprints.

Other Application Domains

A final area of future research is in applying instructional footprinting to other application domains such as operating systems and networking. This research effort concentrated on applying instructional footprinting in the Linda domain and dramatic execution speedups were the result. By using the Linda domain as the proving ground, it is now time for other application domains to benefit from instructional footprinting as well.

BIBLIOGRAPHY

[ARTHU91] J. D. Arthur, G. Cline and K. Landry, "Linda-LAN: A Distributed Parallel Processing Environment Based Upon The Linda Paradigm," *A Research Proposal*, Computer Science Department, Virginia PolyTechnic Institute and State University.

[ASHCR89] C. Ashcraft, N. Carriero and D. Gelernter, "Is Explicit Parallelism Natural? Hybrid DB search and sparse LDL^T factorization using Linda," Yale University, Department of Computer Science, *Tech Memo,* January 1989.

[BALAS89] V. Balasundaram and K. Kennedy, "A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations," *Sigplan Notices,* Vol. 24, No. 7, July 1989, pp. 41-53.

[BANER76] U. Banerjee, "Data Dependence in Ordinary Programs," M.S. Thesis, University of Ill. at Urbana-Champaign, DCS Tech Report # UIUCDCS-R-76-837, October 1976.

[BANER79] U. Banerjee, "Speedup of Ordinary Programs," PhD. Thesis, University of Ill. at Urbana-Champaign, DCS Tech Report # UIUCDCS-R-79-989, October 1979.

[BANSA89] A. Bansal and L. Sterling, "Transforming Generate-and-Test Programs to Execute Under Committed-Choice AND-Parallelism," *International Journal of Parallel Programming,* Vol. 18, No. 5, 1989, pp. 401-446.

[BAXTE89] W. Baxter and H. Bauer, III, "The Program Dependence Graph and Vectorization," *Sixteenth Annual ACM Symposium on Principles of Programming Languages,*" January 1989, pp. 1-11.

[BIRRE84] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Volume 2, Number 1, February 1984, Pages 39 - 59.

[BJORN89a] R. Bjornson, N. Carriero, and D. Gelernter, "The Implementation and Performance of Hypercube Linda," *Research Report YALEU/DCS/RR-690*, March 1989.

[BJORN89b] R. Bjornson, "Experience with Linda on the iPSC/2," *Research Report YALEU/DCS/RR-698*, March 1989.

[BJORN92] R. Bjornson, "Linda on Distributed Memory Multiprocessors," *Research Report YALEU/DCS/RR-698*, November 1992.

[BLOSS88] A. Bloss, P. Hudak and J. Young, "Code Optimizations for Lazy Evaluation," *Lisp and Symbolic Computation*, 1, 1988, pp. 147-164.

[BORRM88] L. Borrmann, M Herdieckerhoff and A. Klein, "Tuple Space Integrated into Modula-2, Implementation of the Linda Concept on a Hierarchical Multiprocessor," *CONPAR88*, 1988, pp. 659-666.

[BROOK75] F. Brooks, The Mythical Man-Month, Addison Wesley, 1975.

[BURKE86] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelism," *Sigplan Notices*, Vol. 21, No. 7, July 1986, pp. 162-175.

[BURKE90] M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, July 1990, pp. 341-395.

[CALLA87] D. Callahan, and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," Proceedings of the First International Conference on Supercomputing, Athens, Greece, 1987. Available as Rice University, Department of Computer Science Technical Report TR87-57, July 1987.

[CALLA90] D. Callahan and B. Smith, "A Future-based Parallel Language for a General-purpose Highly-parallel Computer," *Languages and Compilers for Parallel Computing*, MIT Press, 1990, pp. 95-113.

[CARRI86a] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986, Pages 110-129.

[CARRI86b] N. Carriero and D. Gelernter, "Linda on Hypercube Multicomputers," *Hypercube Multiprocessors 1986*, Siam, pp. 45-56.

[CARRI87] N. Carriero, "Implementation of Tuple Space Machines," *Research Report YALEU/DCS/RR-567* (PhD thesis), December 1987.

[CARRI88] N. Carriero and D. Gelernter, "Applications Experience with Linda," *Proc. ACM Symp. Parallel Programming*, July 1988.

[CARRI89a] N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, Vol. 32, No. 4, April 1989.

[CARRI89b] N. Carriero and D. Gelernter, "Coordination Languages and their Significance," *Yale Tech Report,* YALEU/DCS/RR-716, July 1989.

[CARRI90] N. Carriero and D. Gelernter, "Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler," *Languages and Compilers for Parallel Computing*, MIT Press, 1990, pp. 115-125.

[CARRI92] N. Carriero, D. Gelernter and T.G. Mattson, "Linda in Heterogeneous Computing Environments," *Proceedings of the Workshop on Heterogeneous Processing*, IEEE, March 1992.

[CHASE90] D. R. Chase, M. Wegman and F. K. Zadeck, "Analysis of Pointers and Structures," *Sigplan Notices,* Vol. 25, No. 6, June 1990, pp. 296-310.

[CHATT89] A. Chatterjee, "FUTURES: A Mechanism For Concurrency Among Objects," Proceedings of SuperComputing '89, November, 1989, pp. 562-567.

[CHIBA92] S. Chiba, K. Kato and T. Masuda, "Exploiting a Weak Consistency to Implement Distributed Tuple Space," *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992, pp. 416-423.

[DOWLI90] M. Dowling, "Optimal code parallelization using unimodular transformations," Parallel Computing, Vol. 16, No. 2&3, December 1990, pp. 157-171.

[EBCIO90] K. Ebcioglu and T. Nakatani, "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture," *Languages and Compilers for Parallel Computing,* 1990, pp. 213-229.

[FERRA87] J. Ferrante, K. Ottenstein and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems,* Vol. 9, No. 3, July 1987, pp. 319-349.

[FRADE91] P. Fradet and D. Le Metayer, "Compilation of Functional Languages by Program Transformation," ACM Transactions on Programming Languages and Systems, Vol. 13, No. 1, January 1991, pp. 21-51.

[GELER85a] D. Gelernter, "Generative Communication in Linda," ACM *Transactions on Programming Languages and Systems*, Vol. 7, No. 1, January 1985, Pages 80-112.

[GELER85b] D. Gelernter, N. Carriero, S. Chandran and S. Chang, "Parallel Programming in Linda," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp.255-263.

[GELER90] D. Gelernter, "Ada-Linda: Motivation, Informal Description and Examples," *Yale Technical Report.*

[GELER92] D. Gelernter and N. Carriero, "Coordination Languages and their Significance," Communications of the ACM, Vol. 35, No. 2, February 1992, pp. 97-107.

[HALST85] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, October 1985, pp. 501-538.

[HORWI88] S. Horwitz, J. Prins, and T. Reps, "On the Adequacy of Program Dependence Graphs for Representing Programs," *Fifteenth Annual ACM Symposium on Principles of Programming Languages,* January 1988, pp. 146-157.

[HORWI89] S. Horwitz, P. Pfeiffer and T. Reps, "Dependence Analysis for Pointer Variables," *Sigplan Notices*, Vol. 24, No. 7, July 1989, pp. 28-40.

[HUDAK89] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," ACM Computing Surveys, Vol. 21, No. 3, September 1989, pp. 359-411.

[IWANO90] K. Iwano and S. Yeh, "An Efficient Algorithm for Optimal Loop Parallelization (Extended Abstract)," *Lecture Notes in Computer Science,* No. 450, August 1990, pp. 201-210.

[JELLI90] R. Jellinghaus, "Eiffel Linda: An Object-Oriented Linda Dialect," ACM SIGPLAN Notices, Vol . 25, No. 12, December 1990, pp. 70-84.

[JENSE90] K. K. Jensen, "The Semantics of Tuple Space and Correctness of an Implementation," *Yale Tech Report,* YALEU/DCS/RR-788, April 1990.

[KAMBH91] S. Kambhatla, "Replication Issues for a Distributed and Highly Available Linda Tuple Space," *Master's Thesis*, Department of Computer Science, Oregon Graduate Institute, 1991.

[KRISH87] V. Krishnaswamy, S. Ahuja, N. Carriero and D. Gelernter, "The Linda Machine," *1987 Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation*, Chapter 36, Princeton University, September 30 - October 1, 1987.

[KRISH88] V. Krishnaswamy, S. Ahuja, N. Carriero and D. Gelernter, "The Architecture of a Linda Coprocessor," *Conference Proceedings of The 15th Annual International Symposium on Computer Architecture*, May 30 - June 2, 1988, pp. 240 - 249.

[LANDI90] W. Landi and B. Ryder, "Pointer-induced Aliasing: A Problem Classification," *Eighteenth Annual ACM Symposium on Principles of Programming Languages,* January 1991, pp. 93-103.

[LANDR92a] Landry K., and Arthur J. D., "Instructional Footprinting: A Basis for Exploiting Concurrency Through Instructional Footprinting and Code Motion: A Research Prospectus," *Tech Report # TR 92-33*, Department of Computer Science, Virginia Tech, June 1992.

[LANDR92b] Landry K., and Arthur J. D., "Instructional Footprinting: A Basis for Exploiting Concurrency Through Instructional Footprinting and Code Motion," *1992 Virginia Computer Users Conference*, Virginia Tech, September 1992.

[LANDR93a] Landry K., and Arthur J. D., "Instructional Footprinting and Semantic Preservation in Linda," Submitted for publication in *Concurrency: Practice and Experience*.

[LANDR93b] Landry K., and Arthur J. D., "Instructional Footprinting: A Basis for Exploiting Concurrency Through Instructional Footprinting and Code Motion," Submitted for publication in the *1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[LANDR94a] Landry K., and Arthur J. D., "Boundary Analysis in the Instructional Footprint Model: An Implementation in Linda," Submitted for publication in the *1994 International Conference on Parallel Processing*.

[LANDR94b] Landry K., and Arthur J. D., "Achieving Asynchronous Speedup While Preserving Synchronous Semantics: An Implementation of Instructional Footprinting in Linda," To appear in the *1994 IEEE International Conference on Computer Languages*, May 1994.

[LARUS88a] J. Larus and P. Hilfinger, "Detecting Conflicts Between Structure Accesses," *Sigplan Notices*, Vol. 23, No. 7, July 1988, pp. 21-34.

[LARUS88b] J. Larus and P. Hilfinger, "Restructuring Lisp Programs for Concurrent Execution," *Sigplan Notices*, Vol. 23, No. 9, September 1988, pp.100-110.

[LEICH89] J. Leichter, "Shared Tuple Memories, Shared Memories, Buses and LAN's -- Linda Implementation Across the Spectrum of Connectivity," *PhD Thesis*, Yale University, Computer Science Department, July 1989.

[LI90] Z. Li and P. Yew, "Some Results on Exact Data Dependence Analysis," *Languages and Compilers for Parallel Computing,* 1990, pp. 374-401.

[LISTO86] B. Listov M. Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for distributed Computing," *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, June, 1988, pp. 150-159.

[LISTO88] B. Listov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," *Proceedings of the '88 Conference on Programming Language Design and Implementation*, June, 1988, pp. 260-267.

[LUCCO86] S. Lucco, "A heuristic Linda kernel for hypercube multiprocessors," *Proceedings of the 1986 Workshop on Hypercube Multiprocessors*, September 1986.

[MAEKA87] M. Maekawa, A.E. Oldehoeft and R. R. Oldehoeft, *Operating Systems: Advanced Topics*, 1987.

[NEIRY87] A. Neirynck and A. Demers, "Computation of Aliases and Support Sets," *Fourteenth Annual ACM Symposium on Principles of Programming Languages,* January 1987, pp. 274-283.

[NOBAY89] H. Nobayashi and C. Eoyang, "A Comparison Study of Automatically Vectorizing Fortran Compilers," Proceedings Supercomputing '89, November 1989, pp. 820-825.

[PATTE93] L.I. Patterson, R.S. Turner, R.M. Hyatt and K.D. Reilly, "Construction of a Fault-Tolerant Distributed Tuple-Space," *Proceedings of the 1993 Symposium on Applied Computing*, ACM/SIGAPP, February 1993, pp. 279-285.

[POLYC90] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran," *Languages and Compilers for Parallel Computing,* 1990, pp. 423-453.

[RAMKU89] B. Ramkumar and L. Kale, "Compiled Execution of the Reduced-Or Process Model on Multiprocessors," *Technical Report UIUCDCS-R-89-1513*, University of Illinois at Urbana-Champaign, May 1989.

[RAMSH88] L. Ramshaw, "Eliminating goto's while Preserving Program Structure," *Journal of the Association for Computing Machinery,* Vol. 35, No. 4, October 1988, pp. 893-920.

[SALTZ89] J. Saltz, R. Mirchandaney and D. Baxter, "Run-Time Parallelization and Scheduling of Loops," *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures,* June 1989, pp. 303-312.

[SCHWI91] U. Schwiegelshohn, F. Gasperoni and K. Ebciouglu, "On Optimal Parallelization of Arbitrary Loops," Journal of Parallel and Distributed Computing, Vol. 11, No. 2, February 1991, pp. 130-134.

[SCHUM91] C. Schumann, K. Landry and J. D. Arthur, "Comparison of Unix Communication Facilities Used in Linda," *Proceedings of the 1991 Virginia Computer Users Conference.*

[TSUDA90] T. Tsuda and Y. Kunieda, "V-Pascal: An Automatic Vectorizing Compiler for Pascal with No Language Extensions," *The Journal of Supercomputing,* Vol. 4, No. 3, September 1990, pp. 251-275.

[WEIHL89] W. E. Weihl, "Remote Procedure Call," *Distributed Systems*, 1989, pp. 65-85.

[WHITE88] R. Whiteside and J. Leichter, "Using Linda for Supercomputing On a Local Area Network," in *Proc. Supercomputing '88*, November 1988.

[WOLFE90] M. Wolfe, "Data Dependence and Program Restructuring," *The Journal of Supercomputing,* Vol. 4, No. 4, January 1991, pp. 321-344.

[ZENIT90] S. E. Zenith, "Linda Coordination Language; subsystem kernel architecture (on transputers)," *Research Report YALEU/DCS/RR-794*, May 1990.

[ZIMA90] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers,* ACM Press, New York, 1991.

APPENDIX A - Proof of Correctness

Given two instructions, i and j, instruction i is being moved up past instruction j. The proof of the Instructional Footprinting algorithm in Figure 3-13 can be broken down into two parts:

1. Prove the CanSwap() routine which means proving Theorem 1 with respect to instructions i and j, and

2. Prove that instruction i executes the same number of times as in its original position.

Part 1 - Proof of Theorem 1

The proof of Theorem 1 can be accomplished by performing a mapping between the definitions of instructions in Instructional Footprinting and processes associated with Bernstein's Conditions.

Instructional Footprinting Definitions:

For a given instruction i, the following defines the read and write sets respectively:

ρ_i - The *read set* is an unordered set of variables. A variable **x** is a member of ρ_i iff **x** is non-destructively referenced in instruction **i**.

ω_i - The *write set* is an unordered set of variables. A variable **x** is a member of ω_i iff **x** is destructively referenced in instruction **i**.

Theorem 1 used in the CanSwap() routine:

> Two instructions **i** and **j** can be positionally swapped if for every variable **x** that is an element of ρ_i **U** ω_i, **x** is not an element of ω_j. Similarly, for every variable **x** that is an element of ρ_j **U** ω_j, **x** is not an element of ω_i.

Theorem 1 can be rewritten as:

> Two instructions can be positionally swapped if they are mutually non-interfering:
>
> $(\rho_i \cup \omega_i) \cap \omega_j = \varnothing$ **and**
>
> $(\rho_j \cup \omega_j) \cap \omega_i = \varnothing$

Bernstein's Conditions and Associated Definitions and Theorems:

For a given instruction i, the Range and Domain are indicated as:

> Range = R_i = Write Set
>
> Domain = D_i = Read Set

Given a system of processes, and given P_i and P_j are two of the processes, then P_i and P_j are mutually non-interfering if

$$(R(P_i) \cap R(P_j)) \cup (R(P_i) \cap D(P_j)) \cup (D(P_i) \cap R(P_j)) = \varnothing$$

In addition, a theorem associated with the above condition is:

> A mutually non-interfering system of processes is determinate.

Mapping between Instructional Footprinting and Bernstein's Conditions:

1. An instruction in Instructional Footprinting is a process in Bernstein's Conditions.
2. Instructions i and j map to processes P_i and P_j and are considered the system of processes.
3. R = Range = Write Set = ω
4. D = Domain = Read Set = ρ
5. Stating instructions i and j can be positionally swapped means that system of processes, P_i and P_j, are determinate.

We now need to map the definitions for mutually non-interfering in Theorem 1 to the definitions for mutually non-interfering in Bernstein's conditions.

The definitions in Theorem 1 can be rewritten as follows:

$$((\rho_i \cup \omega_i) \cap \omega_j) \cup ((\rho_j \cup \omega_j) \cap \omega_i) = \varnothing$$

or $$(\rho_i \cap \omega_j) \cup (\omega_i \cap \omega_j) \cup (\rho_j \cap \omega_i) \cup (\omega_j \cap \omega_i) = \varnothing$$

Removing the duplicate terms, we get:

$$(\rho_i \cap \omega_j) \cup (\rho_j \cap \omega_i) \cup (\omega_j \cup \omega_i) = \varnothing$$

Replacing ρ with D, ω with R, i with P_i, and j with P_j, we the same definition as with Bernstein's conditions:

$$(R(P_i) \cap R(P_j)) \cup (R(P_i) \cap D(P_j)) \cup (D(P_i) \cap R(P_j)) = \varnothing$$

Part 2 - Proof of Execution Cardinality

Given that instruction i is to be moved up past instruction j, instruction i must be executed the same number of times in its new position as in its original position. The same can be proven for instruction i being moved down past instruction j.

Let's assume without loss of generality that i is executed exactly 1 time in its original position. It must be proven that the Instructional Footprinting algorithm does not violate the execution cardinality rule.

There are essentially 3 type of aggregate instructions that instruction i encounters as it moves:

1. Blocks
2. Conditionals
3. Loops

Let's consider each aggregate instruction individually.

Blocks:

Every instruction in a block is executed exactly one time. If instruction i is move to another position in a block of code, then it still will be executed exactly 1 time. The footprinting algorithm deaggregates blocks of code when a conflict is found and allows the instruction to move past the individual instructions of the block of code. Therefore, the footprinting algorithm does not violate the execution cardinality rule with respect to blocks of code.

Conditionals:

Case 1: Instruction i resides within either the THEN or the ELSE parts.

In this case, instruction i cannot be moved outside the confines of the THEN or ELSE block of code. Because i resides within a BLOCK of code, i is guaranteed to execute exactly once (see execution within a BLOCK above).

Case 2: Instruction i is being moved past a CONDITIONAL and there is conflict.

In a conditional, either the THEN or the ELSE block of code is executed, but not both. Because the THEN and ELSE parts are blocks of code, each instruction is executed exactly once. In the footprinting algorithm, aggregate instructions that are conditionals are deaggregated when a conflict occurs. To ensure that the instruction being moved is executed exactly once, it is propagated through both the THEN and ELSE parts. Therefore, the footprinting algorithm does not violate the execution cardinality rule with respect to moving past CONDITIONALS.

Case 3: Instruction i is being moved past a CONDITIONAL and there is no conflict.

In this case, no conflict exists between instruction i and the CONDITIONAL. Instruction i and the CONDITIONAL effectively execute as single instructions within a 2-instruction BLOCK of code. Therefore instruction i can be moved past the CONDITIONAL without violating the execution cardinality rule.

Looping:

Case 1: Instruction i resides within a LOOP.

In this case, instruction i cannot be moved outside the confines of the LOOP. Because instruction i resides within the LOOP body which is a BLOCK of code, i is guaranteed to executed the same number of times regardless of the position within the LOOP body.

Case 2: Instruction i is being moved past a LOOP.

There is never a situation when instruction i is allowed to step into a LOOP. Therefore if a conflict exists between instruction i and a LOOP, the footprinting process stops. If there is no conflict between instruction i and the LOOP, then instruction i can be moved past the LOOP while still maintaining the execution cardinality rule. This is because instruction i and the LOOP effectively execute as single instructions in a 2-instruction BLOCK of code.

APPENDIX B - LPT Justifications

The following pages present justifications for why, in the absence of Tuple Sequencing and Tuple Identification, **INIT**s and **RECV**s for **IN**s and **RD**s can or cannot be moved past other Linda operations. For each movement, a YES or NO is given for whether the movement is safe along the reason in parenthesis. This is followed by a justification.

$\mathbf{INIT_{IN}}$ PAST $\mathbf{INIT_{IN}}$ **NO (Posterior Temporal Influence)**

Moving an $\mathbf{INIT_{IN}}$ up past an another $\mathbf{INIT_{IN}}$ can alter program semantics by potentially allowing multiple access to critical regions defined by the use of Linda operations. The following example illustrates this potential problem.

```
IN("Task 1")                                INIT("Task 1", "Check Bal", ?bal)
IN("Task 1", "Check Bal", ?bal) =====> INIT("Task 1")
                                            RECV("Task 1")
                                            RECV("Task 1", "Check Bal", ?bal)
```

Suppose the above code has previously **OUT**ed checking balance tuples and then spawned off tasks (using **EVAL**s) and it is now waiting for task 1 to finish by performing an **IN**("Task 1") and getting the checking balance with the next **IN**. If the **INIT** for the checking balance is performed before the **INIT** for the task 1 then it is possible for the checking balance tuple to be taken out of TS before the new value is placed in it (assuming task 1 is not finished and has not written a new checking balance tuple).

NOTE: Moving an $\mathbf{INIT_{IN}}$ past an $\mathbf{INIT_{IN}}$ implies that the $\mathbf{INIT_{IN}}$ is moved up past a $\mathbf{RECV_{IN}}$. Neither Tuple Sequencing or Tuple Identification can be used to preserve program semantics.

INIT$_{IN}$ PAST RECV$_{IN}$ **NO (Posterior Temporal Influence)**

This case is identical to the case of moving an **INIT$_{IN}$** past an **INIT$_{IN}$**. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. With this in mind, moving an **INIT$_{IN}$** past a **RECV$_{IN}$** can act like the case when an **INIT$_{IN}$** is moved past an **INIT$_{IN}$**.

NOTE: Tuple Sequencing can be used to preserve program semantics.

RECV$_{IN}$ PAST INIT$_{IN}$ **NO (Posterior Temporal Influence)**

This case is identical to the case of moving an **INIT$_{IN}$** past an **INIT** for another **IN**. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. Moving a **RECV** past an **INIT** causes two **INIT**s to be executed one right after the other. With this in mind, moving a **RECV$_{IN}$** past an **INIT$_{IN}$** can act like the case when an **INIT$_{IN}$** is moved past an **INIT** for another **IN**.

NOTE: Tuple Sequencing can be used to preserve program semantics.

$\texttt{RECV}_{\texttt{IN}}$ PAST $\texttt{RECV}_{\texttt{IN}}$ **NO (Posterior Temporal Influence)**

This case is identical to the case of moving an $\texttt{INIT}_{\texttt{IN}}$ past an **INIT** for another **IN**. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. Moving a **RECV** past another **RECV** causes two **INIT**s to be executed one right after the other. With this in mind, moving a $\texttt{RECV}_{\texttt{IN}}$ past a $\texttt{RECV}_{\texttt{IN}}$ can act like the case when an $\texttt{INIT}_{\texttt{IN}}$ is moved past an **INIT** for another **IN**.

NOTE: Moving a $\texttt{RECV}_{\texttt{IN}}$ past a $\texttt{RECV}_{\texttt{IN}}$ implies that the $\texttt{RECV}_{\texttt{IN}}$ is moved past an $\texttt{INIT}_{\texttt{IN}}$. Tuple Sequencing and Tuple Identification can be used to preserve program semantics.

$\mathbf{INIT_{IN}}$ PAST $\mathbf{INIT_{RD}}$ **NO (Anterior Temporal Influence)**

Moving an $\mathbf{INIT_{IN}}$ past an $\mathbf{INIT_{RD}}$ can alter program semantics. Suppose that both **INIT**s are looking for tuples of the same form. It is possible for the $\mathbf{INIT_{IN}}$ to remove the last matching tuple from TS before the $\mathbf{INIT_{RD}}$ has a chance to make a copy of it. The following example illustrates this.

```
                                                  INIT_IN("Data", ?y)
    RD("Data", ?x)              ======>           INIT_RD("Data", ?x)
    IN("Data", ?y)                                RECV_RD("Data", ?x)
                                                  RECV_IN("Data", ?y)
```

In the above code, the unoptimized version would (assuming a single matching tuple in TS) would return a copy to the **RD** and remove the tuple and return it for the **IN**. In the optimized version, it is possible for the **IN** to remove the tuple first and then block on the **RD**. Therefore, program semantics are altered.

NOTE: Moving an $\mathbf{INIT_{IN}}$ past an $\mathbf{INIT_{RD}}$ implies moving the $\mathbf{INIT_{IN}}$ past a $\mathbf{RECV_{RD}}$. Program semantics cannot be preserved even with the use of Tuple Sequencing and Tuple Identification.

$\texttt{INIT}_{\texttt{IN}}$ PAST $\texttt{RECV}_{\texttt{RD}}$ **NO (Anterior Temporal Influence)**

This case is identical to the case of moving an $\texttt{INIT}_{\texttt{IN}}$ past an $\texttt{INIT}_{\texttt{RD}}$. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. With this in mind, moving an $\texttt{INIT}_{\texttt{IN}}$ past a $\texttt{RECV}_{\texttt{RD}}$ can act like the case when an $\texttt{INIT}_{\texttt{IN}}$ is moved past an $\texttt{INIT}_{\texttt{RD}}$.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing.

$\mathbf{RECV_{IN}}$ PAST $\mathbf{INIT_{RD}}$ **NO (Posterior Temporal Influence)**

Moving a $\mathbf{RECV_{IN}}$ past an $\mathbf{INIT_{RD}}$ can alter program semantics. Suppose that both **INIT**s are looking for tuples of the same form. It is possible for the $\mathbf{INIT_{RD}}$ to make a copy of the last matching tuple in TS before the $\mathbf{INIT_{IN}}$ has a chance to remove it from TS. The following example illustrates this.

```
                                          INIT_IN("Data", ?x)
    IN("Data", ?x)          ======>       INIT_RD("Data", ?y)
    RD("Data", ?y)                        RECV_IN("Data", ?x)
                                          RECV_RD("Data", ?y)
```

In the above code, the unoptimized version would (assuming a single matching tuple in TS) would remove the tuple from TS and return it to the **IN** and the **RD** would block waiting for another matching tuple to arrive in TS. In the optimized version, it is possible for the **RD** to make a copy of the tuple before it is removed and sent back to the **IN**. This would give both **INIT**s the same tuple and the code would not block. Therefore, program semantics are altered.

NOTE: Moving an $\mathbf{INIT_{IN}}$ past an $\mathbf{INIT_{RD}}$ implies moving the $\mathbf{INIT_{IN}}$ past a $\mathbf{RECV_{RD}}$. Program semantics can be preserved with the use of Tuple Sequencing.

$\texttt{RECV}_{\texttt{IN}}$ PAST $\texttt{RECV}_{\texttt{RD}}$ **NO (Posterior Temporal Influence)**

This case is identical to the case of moving a $\texttt{RECV}_{\texttt{IN}}$ past an $\texttt{INIT}_{\texttt{RD}}$. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. Moving a $\texttt{RECV}_{\texttt{IN}}$ past a $\texttt{RECV}_{\texttt{IN}}$ causes two **INIT**s to be executed back to back. With this in mind, moving an $\texttt{INIT}_{\texttt{IN}}$ past a $\texttt{RECV}_{\texttt{RD}}$ can act like the case when an $\texttt{INIT}_{\texttt{IN}}$ is moved past an $\texttt{INIT}_{\texttt{RD}}$.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing and Tuple Identification.

$\mathbf{INIT_{IN}}$ PAST OUT **NO (Posterior Temporal Influence)**

An **$\mathbf{INIT_{IN}}$** cannot be moved up past an **OUT** primitive without compromising program semantics. In order to illustrate this, the following example points out how results can change in an optimized version.

```
OUT("Data", y)                          INIT("Data", ?x)
      :                                 OUT("Data", y)
      :              ========>               :
      :                                      :
      :                                      :
IN("Data", ?x)                          RECV("Data", ?x)
```

Suppose that when the **INIT** is executed, only one "Data" tuple is in TS. The tuple would be removed from TS and sent back. Suppose that before the **OUT** is performed, a **RDP** is executed by another process resulting in a false being sent back because TS is void of "Data" tuples. This situation would not happen in the unoptimized version because the request for the "Data" tuple would occur after the **OUT**. This demonstrates that program semantics can be altered if an **$\mathbf{INIT_{IN}}$** is moved up past an **OUT**.

NOTE: Program semantics cannot be preserved with the use of Tuple Sequencing or Tuple Identification.

RECV$_{IN}$ PAST OUT **NO (Anterior Temporal Influence)**

Moving a **RECV$_{IN}$** down past an **OUT** primitive can cause the **IN** to be affected by the **OUT** which should not happen. The following example illustrates this fact.

```
IN("Data", ?x)                          INIT("Data", ?x)
     :                                       :
     :                ==========>            :
     :                                       :
OUT("Data", ?x)                         OUT("Data", y)
                                        RECV("Data", ?x)
```

Suppose there are no "Data" tuples in TS, the unoptimized code above would block on the **IN** ("Data" could be used as semaphores). In the optimized code, the **INIT** is affected by the **OUT** by taking the tuple it put out in TS. Therefore, moving **RECV$_{IN}$** past **OUT** can alter program semantics.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing.

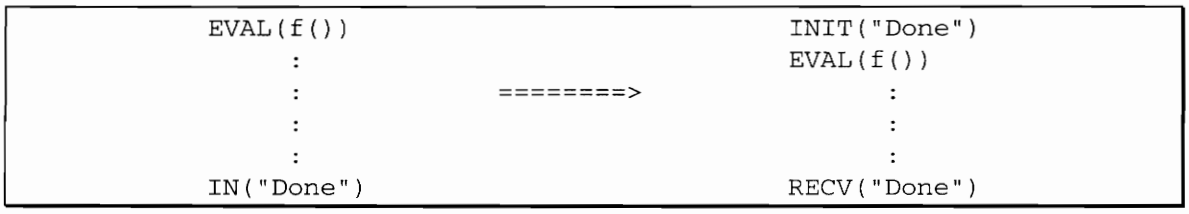
INIT$_{IN}$ PAST EVAL **YES (TS Semantics)**

The reason an **INIT$_{IN}$** can be moved up past an **EVAL** is because of the definition of TS, **INIT**, **RECV** and **EVAL**. By this I mean 2 things:

1) **EVAL** is not guaranteed to place a data tuple in TS before it returns, and

2) TS does not guarantee which matching tuple will be returned to an **IN**.

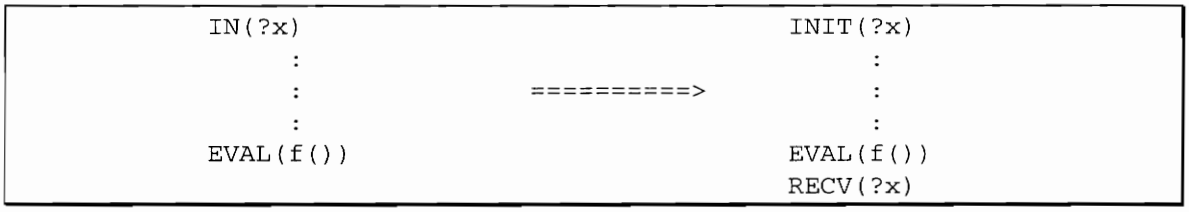
These two facts about TS semantics are important because it is not guaranteed that the **EVAL** will finish before **IN** is executed (if unoptimized). In addition, even if the **EVAL** does deliver a matching tuple into TS before the **IN** is executed it is not guaranteed that the **IN** will get that particular tuple. The following example will illustrate these points.



If no optimization is made, it is possible that no tuple activity from the **EVAL**ed function `f()` will occur before the **IN** is reached and executed. This is precisely the case in the optimized version. The initiation for the **IN** is made and then the function `f()` is **EVAL**ed. Therefore because this scenario is allowable in the unoptimized version it is allowable to move **INIT$_{IN}$** up past an **EVAL**.

RECV$_{IN}$ PAST EVAL **NO (Anterior Temporal Influence)**

Moving a **RECV$_{IN}$** down past an **EVAL** primitive can cause the **IN** to be affected by the **EVAL** which should not happen. The following example illustrates this fact.



Suppose there are no matching tuples in TS, the unoptimized code above would block on the **IN**. In the optimized code, the **INIT** is affected by the **EVAL** by taking the tuple the **EVAL** placed in TS. Therefore, moving **RECV$_{IN}$** past **EVAL** can alter program semantics.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing.

$\mathbf{INIT_{IN}}$ PAST INP **NO (Anterior & Posterior Temporal Influence)**

Moving **$\mathbf{INIT_{IN}}$** past **INP** can alter program semantics in both an anterior and a posterior fashion. By this I mean that both the primary and the secondary component instructions are affected. The following example illustrates this fact.

```
rtn = INP("Data", ?x)                    INIT("Data", ?y)
IN("Data", ?y)              ====>        rtn = INP("Data", ?x)
                                         RECV("Data", ?y)
```

With only one matching "Data" tuple in TS, the **INP** of the unoptimized code above would remove the singular tuple giving `rtn` a value of true and the **IN** would subsequently block. In the optimized version, the **INIT** could remove the singular matching tuple before the **INP** reaches TS. This would mean that `rtn` would have a value of false and the code would not block. Therefore, moving an **$\mathbf{INIT_{IN}}$** past an **INP** can alter program semantics. Both the primary and secondary component instructions are affected.

NOTE: Program semantics cannot be preserved with the use of Tuple Sequencing or Tuple Identification.

$\mathbf{RECV_{IN}}$ PAST `INP` **NO (Posterior & Anterior Temporal Influence)**

Moving an **$\mathbf{RECV_{IN}}$** past an **`INP`** can alter program semantics in both an anterior and posterior fashion. By this I mean that both the primary and the secondary component instructions are affected. The following example illustrates this fact.

```
IN("Data", ?y)                          INIT("Data", ?y)
rtn = INP("Data", ?x)        ====>      rtn = INP("Data", ?x)
                                        RECV("Data", ?y)
```

With only one matching "Data" tuple in TS, the **`IN`** operation in the unoptimized code above would remove the singular tuple and the **`INP`** would subsequently fail to find a matching tuple and return false to `rtn`. In the optimized version, the **`INIT`** could not remove the singular matching tuple before the **`INP`** reaches TS. This would mean that `rtn` would have a value of true and the code would block on the **`RECV`**. Therefore, moving an **$\mathbf{INIT_{IN}}$** past an **`INP`** can alter program semantics. Both the primary and secondary component instructions are affected.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing.

$\mathbf{INIT_{IN}}$ PAST RDP **NO (Anterior Temporal Influence)**

Moving **$\mathbf{INIT_{IN}}$** past **RDP** can alter program semantics in an anterior fashion. The following example illustrates this fact.

```
rtn = RDP("Data", ?x)                    INIT("Data", ?y)
IN("Data", ?y)              ====>        rtn = RDP("Data", ?x)
                                         RECV("Data", ?y)
```

With only one matching "Data" tuple in TS, the **RDP** in the unoptimized code above would return a copy of the singular tuple giving `rtn` a value of true and the **IN** would remove the same tuple from TS. In the optimized version, the **INIT** could remove the singular matching tuple before the **RDP** reaches TS. This would mean that `rtn` would have a value of false. Therefore, moving an **$\mathbf{INIT_{IN}}$** past a **RDP** can alter program semantics.

NOTE: Program semantics cannot be preserved with the use of Tuple Sequencing or Tuple Identification.

$\mathbf{RECV_{IN}}$ PAST RDP **NO (Posterior Temporal Influence)**

Moving a **$\mathbf{RECV_{IN}}$** past a **RDP** can alter program semantics in a posterior fashion. The following example illustrates this fact.

```
IN("Data", ?y)                              INIT("Data", ?y)
rtn = RDP("Data", ?x)          ====>        rtn = RDP("Data", ?x)
                                            RECV("Data", ?y)
```

With only one matching "Data" tuple in TS, the **IN** in the unoptimized code above would remove the singular tuple and the **RDP** would fail to find a matching tuple and would return false to `rtn`. In the optimized version, the **INIT** could fail to remove the singular matching tuple before the **RDP** reaches TS. This would mean that `rtn` would have a value of true. Therefore, moving an **$\mathbf{INIT_{IN}}$** past a **RDP** can alter program semantics.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing.

$\texttt{INIT}_{\texttt{RD}}$ PAST $\texttt{INIT}_{\texttt{IN}}$ **NO (Posterior Temporal Influence)**

This is identical to moving a $\texttt{RECV}_{\texttt{IN}}$ down past an $\texttt{INIT}_{\texttt{RD}}$. The only difference is that the **INIT**s are executed in reverse order. This is irrelevant because it is not guaranteed (according to the semantics of the **INIT**) which **INIT** will reach TS first. Therefore, moving an $\texttt{INIT}_{\texttt{IN}}$ up past an $\texttt{INIT}_{\texttt{RD}}$ can alter program semantics.

NOTE: Moving an $\texttt{INIT}_{\texttt{RD}}$ past an $\texttt{INIT}_{\texttt{IN}}$ implies moving the $\texttt{INIT}_{\texttt{RD}}$ past a $\texttt{RECV}_{\texttt{IN}}$. Program semantics cannot be preserved even with the use of Tuple Sequencing and Tuple Identification.

$\mathbf{INIT_{RD}}$ PAST $\mathbf{RECV_{IN}}$ **NO (Posterior Temporal Influence)**

This is identical to the case of moving a $\mathbf{RECV_{IN}}$ down past an $\mathbf{INIT_{RD}}$. Therefore moving $\mathbf{INIT_{RD}}$ past $\mathbf{RECV_{IN}}$ can cause program semantics to be altered.

NOTE: As with moving a $\mathbf{RECV_{IN}}$ down past an $\mathbf{INIT_{RD}}$, program semantics can be preserved with the use of Tuple Sequencing.

$\texttt{RECV}_{\texttt{RD}}$ PAST $\texttt{INIT}_{\texttt{IN}}$ **NO(Anterior Temporal Influence)**

This is identical to the case of moving an $\texttt{INIT}_{\texttt{IN}}$ up past a $\texttt{RECV}_{\texttt{RD}}$. Therefore moving $\texttt{RECV}_{\texttt{RD}}$ past $\texttt{INIT}_{\texttt{IN}}$ can cause program semantics to be altered.

NOTE: As with moving an $\texttt{INIT}_{\texttt{IN}}$ down past a $\texttt{RECV}_{\texttt{RD}}$, program semantics can be preserved with the use of Tuple Sequencing.

$\mathbf{RECV_{RD}}$ PAST $\mathbf{RECV_{IN}}$ **NO(Anterior Temporal Influence)**

This is similar to moving an $\mathbf{INIT_{IN}}$ up past a $\mathbf{RECV_{RD}}$. The only difference is that the resulting code has the **RECV**s in different order which has no effect. Therefore, moving $\mathbf{RECV_{RD}}$ PAST $\mathbf{RECV_{IN}}$ can alter program semantics.

NOTE: As with moving an $\mathbf{INIT_{IN}}$ down past a $\mathbf{RECV_{RD}}$, program semantics can be preserved with the use of Tuple Sequencing and Tuple Identification.

$\mathbf{INIT_{RD}}$ PAST $\mathbf{INIT_{RD}}$ **NO (Posterior Temporal Influence)**

Moving an $\mathbf{INIT_{RD}}$ up past an another $\mathbf{INIT_{RD}}$ can alter program semantics by potentially allowing multiple access to critical regions defined by the use of Linda operations. The following example illustrates this potential problem.

RD("Task 1") RD("Task 1", "Checking Bal", ?bal)	=====>	INIT("Task 1", "Checking Bal", ?bal) INIT("Task 1") RECV("Task 1") RECV("Task 1", "Checking Bal", ?bal)

Suppose the above code has previously **OUT**ed checking balance tuples and then spawned off tasks (using **EVAL**s) and it is now waiting for task 1 to finish by doing an **RD**("Task 1") and the getting the checking balance with the next **RD**. If the **INIT** for the checking balance is performed before the **INIT** for the task 1 then it is possible for the checking balance tuple to be copied out of TS before the new value is placed in it (assuming task 1 is not finished and has not written a new checking balance tuple).

NOTE: This implies that an $\mathbf{INIT_{RD}}$ is moved up past a $\mathbf{RECV_{RD}}$ is allowable and does preserve program semantics. Neither Tuple Sequencing or Tuple Identification can be used to preserve program semantics.

INIT$_{RD}$ PAST RECV$_{RD}$ **NO (Posterior Temporal Influence)**

This case is identical to the case of moving an **INIT$_{RD}$** past an **INIT$_{IN}$**. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. With this in mind, moving an **INIT$_{RD}$** past a **RECV$_{RD}$** can act like the case when an **INIT$_{RD}$** is moved past an **INIT$_{RD}$**.

NOTE: Tuple Sequencing can be used to preserve program semantics.

RECV$_{RD}$ PAST INIT$_{RD}$ **NO (Posterior Temporal Influence)**

This case is identical to the case of moving an **INIT$_{RD}$** past an **INIT$_{IN}$**. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. Moving a **RECV** past an **INIT** causes two **INIT**s to be executed one right after the other. With this in mind, moving an **RECV$_{RD}$** past a **INIT$_{RD}$** can act like the case when an **INIT$_{RD}$** is moved past an **INIT$_{RD}$**.

NOTE: Tuple Sequencing can be used to preserve program semantics.

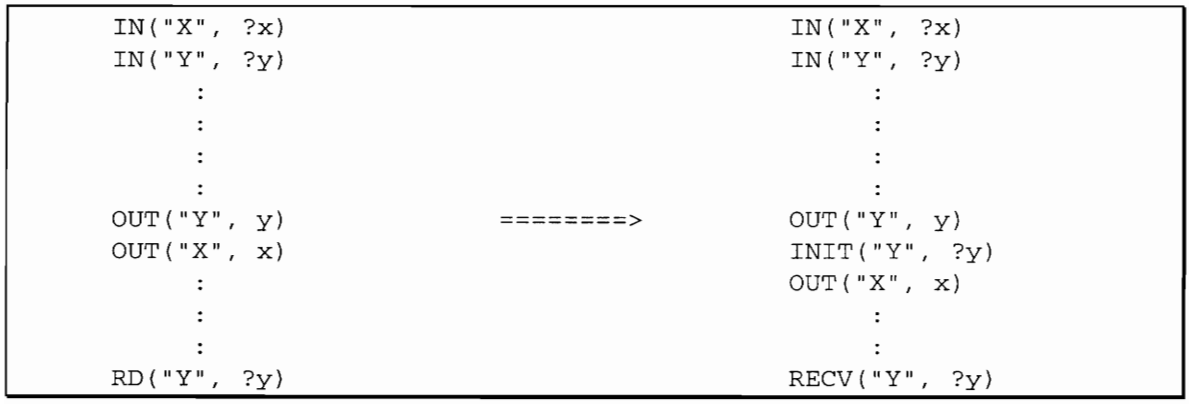
$\texttt{RECV}_{\texttt{RD}}$ PAST $\texttt{RECV}_{\texttt{RD}}$ **YES (Posterior Temporal Influence)**

This case is identical to the case of moving an $\texttt{INIT}_{\texttt{RD}}$ past an $\texttt{INIT}_{\texttt{IN}}$. The reason is that when two **INIT**s are executed one right after another it is not guaranteed which **INIT** will reach (affect) tuple space first. Moving a **RECV** past a **RECV** causes two **INIT**s to be executed one right after the other. With this in mind, moving a $\texttt{RECV}_{\texttt{RD}}$ past a $\texttt{RECV}_{\texttt{RD}}$ can act like the case when an $\texttt{INIT}_{\texttt{RD}}$ is moved past an $\texttt{INIT}_{\texttt{RD}}$.

NOTE: Tuple Sequencing and Tuple Identification can be used to preserve program semantics.

$\mathbf{INIT_{RD}}$ PAST OUT **NO (Posterior Temporal Influence)**

Moving an **$\mathbf{INIT_{RD}}$** up past an **OUT** can change program semantics. Program semantics change because in effect the assumptions under which the **INIT** for the **RD** is executed is different if optimized -- a posterior temporal influence. The following example illustrates this.



In the above example, the **$\mathbf{INIT_{RD}}$**, when optimized, is placed before the **OUT** for the "Y" semaphore tuple. Assuming that there is only one set of "X" and "Y" semaphore tuples, the **INIT** is guaranteed to get the "Y" semaphore and the value that is put out in TS by the just executed **OUT** (in the same process). This guarantee does not hold true in the original unoptimized code. The "Y" semaphore could be from any number of processes that have since passed through the critical region.

NOTE: Tuple Sequencing or Tuple Identification cannot be used to preserve program semantics.

$\mathbf{RECV_{RD}}$ PAST OUT **NO (Anterior Temporal Influence)**

Moving a **$\mathbf{RECV_{RD}}$** down past an **OUT** can alter program semantics. The following example shows how an optimized **RD** request can be satisfied by an **OUT** operation that would otherwise (unoptimized) be impossible.

```
RD("Data", ?x)                          INIT("Data", ?x)
OUT("Data", y)      ========>           OUT("Data", y)
                                        RECV("Data", ?x)
```

Suppose that when the **INIT** is executed there are no "Data" tuples in TS in which case the request is shelved. The next TS operation performed is the **OUT** which places a "Data" tuple in TS. Once this happens, the **INIT** that was shelved can now be satisfied. In the unoptimized code above, it is impossible to satisfy the **RD** with the following **OUT**. Therefore, program semantics can be altered when moving a **$\mathbf{RECV_{RD}}$** down past an **OUT**.

NOTE: Tuple Sequencing can be used to preserve program semantics.

$\texttt{INIT}_{\texttt{RD}}$ PAST EVAL **YES (TS Semantics)**

This case is the same as moving an $\texttt{INIT}_{\texttt{IN}}$ up past an **EVAL**. The only difference is that **IN**s remove tuples from TS. While **RD** operations only make a copy of returned tuples. The fact that **RD** operations do not alter TS only strengthens the conclusion that moving an $\texttt{INIT}_{\texttt{RD}}$ up past an **EVAL**. As stated before, two important points should be made:

1) **EVAL** is not guaranteed to deliver a data tuple in TS before returning, and

2) TS does not guarantee which matching tuple is returned.

These two facts about TS semantics are important because it is not guaranteed that the **EVAL** will finish before **RD** is executed (if unoptimized). In addition, even if the **EVAL** does deliver a matching tuple into TS before the **RD** is executed it is not guaranteed that the **RD** will get that particular tuple.

RECV_{RD} PAST EVAL **NO (Anterior Temporal Influence)**

This optimization is similar to a **RECV_{RD}** being moved down past an **OUT**. An **EVAL** differs from an **OUT** in that the **OUT** places a tuple in TS before it returns which is not true of **EVAL**s. However, an **EVAL** can functionally resemble an **OUT** if there is no TS activity until the resulting tuple from the **EVAL** is placed in TS. Because of this and the fact that program semantics are compromised when a **RECV_{RD}** is moved past an **OUT**, moving a **RECV_{RD}** down past an **EVAL** can alter program semantics.

NOTE: Tuple Sequencing can be used to preserve program semantics.

$\texttt{INIT}_{\texttt{RD}}$ PAST `INP` **NO (Posterior Temporal Influence)**

Moving an **$\texttt{INIT}_{\texttt{RD}}$** up past an **`INP`** can alter program semantics. To see this consider the following example.

```
rtn = INP("Data", ?x)                    INIT("Data", ?y)
RD("Data", ?y)          ======>          rtn = INP("Data", ?x)
                                         RECV("Data", ?y)
```

Suppose that there is a single "Data" tuple in TS before the above code is executed. In the original code, the "Data" tuple would be returned to the **`INP`** operation while the **`RD`** operation blocks. In the optimized code, the **`INIT`** is satisfied by the singular "Data" tuple in TS as well as satisfying the **`INP`** operation.

NOTE: Tuple Sequencing or Tuple Identification cannot be used to ensure program semantics are unaltered.

$\mathbf{RECV_{RD}}$ PAST INP **NO (Anterior Temporal Influence)**

Moving a **$\mathbf{RECV_{RD}}$** down past an **INP** operation can alter program semantics. In the following example, the boolean value returned by the **INP** can change when optimized.

```
RD("Data", ?x)                          INIT("Data", ?x)
rtn = INP("Data", ?y)      =====>       rtn = INP("Data", ?y)
                                        RECV("Data", ?x)
```

Suppose one "Data" tuple is present in TS. In the original code above, the **RD** would get a copy of the tuple and the **INP** would remove it returning a true to rtn. In the optimized version, it is possible that the **INP** removes the tuple from TS (returning true to rtn) before the **RD** could return a copy of the tuple. Therefore, program semantics are altered when moving a **$\mathbf{RECV_{RD}}$** down past an **INP**.

NOTE: Tuple Sequencing can be used to ensure program semantic preservation.

$\mathbf{INIT_{RD}}$ PAST RDP **YES (TS Semantics)**

Moving an **$\mathbf{INIT_{RD}}$** up past a **RDP** operation does not change program semantics. Recall that **RD** and **RDP** operations do not alter TS. In order to show that program semantics do not change, consider the following code.

```
                                          INIT("Some Tuple")
rtn = RDP( "Some Tuple")      =====>      rtn = RDP("Some Tuple")
RD("Some Tuple")                          RECV("Some Tuple")
```

In the optimized version of the above code, TS at the point the **INIT** is executed is identical to the TS at the point the **RD** is executed in the unoptimized version. This is because the **RDP** does not alter TS or block. The reverse is also true. The TS at the point the **RDP** is executed in the optimized version is the same as the TS at the point the **RDP** is executed in the unoptimized version. This is because the **INIT** for the **RD** does not alter TS nor does it block. Therefore, program semantics are not altered by moving an **$\mathbf{INIT_{RD}}$** up past a **RDP**.

$\mathbf{RECV_{RD}}$ PAST RDP **NO (Posterior Temporal Influence)**

Moving a **$\mathbf{RECV_{RD}}$** down past a **RDP** can cause program semantics to be altered. A posterior temporal effect can be seen in the following example.

```
RD("Data", ?x)                          INIT("Data", ?x)
rtn = RDP("Data", ?y)   =====>          rtn = RDP("Data", ?y)
                                        RECV("Data", ?x)
```

Assume that there are no matching tuples in TS. In the unoptimized code above, the **RD** would block until a matching tuple arrived satisfying both the **RD** and the **RDP** (returning a true to rtn). In the optimized version, it is possible for the **RDP** to reach TS before the matching tuple arrives in TS (as in the unoptimized version). If this happens, the **RDP** fails and returns a value of false to rtn. Therefore, program semantics are altered when moving a **$\mathbf{RECV_{RD}}$** down past a **RDP**.

NOTE: Program semantics can be preserved with the use of Tuple Sequencing.

APPENDIX C - Linda Primitives

This Appendix defines all the Linda primitives, describes the structure of tuples, and also gives examples of how each primitive is used.

OUT

The **OUT** primitive is used to deposit a tuple into TS. A tuple can have one or more fields. The tuple field types may vary depending on the base language in which Linda is inserted. Typically, a field can be of type integer, float, character, array (which includes string), or record. The following are examples of **OUT** operations:

```
out( aFloatValue )

out( "Matrix", i, j, element )

out( vector:len )
```

The first example is **OUT**ing to Tuple Space a tuple with a single floating point field. The second is a **OUT**ing a four-tuple represents an element of a matrix where `i` and `j` are the indices and `element` is the value. The third operation is placing in Tuple Space a tuple containing one field that holds an array whose length is `len` elements.

EVAL

The **EVAL** operations is used to spawn new Linda processes by placing "live" tuples into Tuple Space that are eventually replaced with a normal data tuple when the process is finished. The structure of the **EVAL** is identical to **OUT** operation. The only operational difference between an **OUT** and an **EVAL** is that an **EVAL** operation spawns a new process

to evaluate the tuple fields before the resulting tuple is placed into Tuple Space. The following illustrates how the **EVAL** operation can be used.

```
eval( "Matrix Multiply Results", MatrixMult( x, y ) )
```

A live tuple would be created in Tuple Space whose job is to evaluate the two fields in the matrix multiply tuple. The evaluation of the first field is trivial, while the evaluation of the second field requires a call to MatrixMult. The resulting data tuple is a string followed by the result of MatrixMult - in this case a matrix.

IN

The **IN** primitive is a tuple retrieval operation that uses a tuple template to find matching tuples in Tuple Space. Once a matching tuple is found, it is removed from Tuple Space and send back to the requesting operation. A tuple template resembles a tuple in most ways. For example,

```
in( "Data", i, f )
```

would be used to find a tuple in Tuple Space that has the string `"Data"` as its first field, an integer having the value `i` as its second field, and a float having the value `f` as its third field. The **IN** is a blocking operation. If, for example, no tuple were found matching the template in the above **IN** operation then the **IN** would block until a matching tuple is placed in Tuple Space.

Field values can be returned from matching tuples. This is accomplished by placing a question mark before a field indicating that it is a formal field. Matching is still performed on that field but only on its type. For example,

```
in( "Pixel", i, j, ?color )
```

would return a value for `color` upon finding a matching tuple in Tuple Space. A matching tuple would have `"Pixel"` as the first field, the values `i` and `j` as the next two, and the type of `color` for the fourth.

RD

A **RD** operation is identical to an **IN** except that it does not remove the matching tuple from Tuple Space. The **RD** simple makes a copy of the matching tuple, allowing the original tuple to remain in Tuple Space.

INP

The **INP** is the predicate version of the **IN** operation. In other words, it does not block when a matching tuple is not found in Tuple Space. The **INP** returns, as its return value, a boolean indicating whether a matching tuple is found. Consider the following example.

```
if (inp( "Start Multiply Process") {
     /* tuple found, start the multiply process now */
} else {
     /* tuple NOT found */
     /* Don't start the multiply process yet */
}
```

The **INP** operation would look for a tuple containing `"Start Multiply Process"` as its only field and would return `TRUE` if one is found and `FALSE` if one is not found.

RDP

The **RDP** is identical to the **INP** operation except that it does not remove the matching tuple from Tuple Space. If a matching tuple is found, The **RDP** simple makes a copy of the tuple, allowing the original tuple to remain in Tuple Space.

VITA

The author, Ken Landry, was born to Charles and Janice Landry in Bartlesville, Oklahoma. Up to the age of 9, Ken experienced growing up in many different states as well as living overseas. The rest of his youth was spent in Louisville, Kentucky where he attended Trinity High School.

Feeling the need to flee, Ken attended the University of Alabama in Tuscaloosa. Roll Tide Roll! There he pursued a bachelors degree in computer science in the College of Engineering and graduated in 1986.

Wanting to pursue graduate work, Ken entered the masters program in computer science at Virginia Tech. After two years of study, specializing in artificial intelligence and compilers, Ken received his masters degree in 1988.

The experience of conducting research convinced Ken to stay at Virginia Tech and pursue his doctorate in computer science. Ken spent the next six years working in the area of parallel programming languages. During that time, Ken married his wife Jennifer and together are raising their son, Joshua.

Ken is interested in continuing to pursue his research interests in the areas of programming languages, either in academia or in industry.