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

Kenneth D. Landry and James D. Arthur

TR 92-33

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

June 4, 1992

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

Kenneth D. Landry
James D. Arthur

Department of Computer Science

May 1992

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 simple as a service request made to the operating system or more 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 the request for data *earlier* in the code and *retrieving* the returned data later when it is needed. In order to facilitate this concurrency of processing, an instructional footprint model is developed which formally describes movement of instruction. This paper presents a proposal for research that involves the development of the instructional footprint model and an algorithmic framework in which to exploit concurrency in programming languages.

Contents

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 proposed in this report focuses on calls to functions that are implemented as a *independent* processes, but, for the most part are synchronous in nature causing the calling process to block while waiting for data to be returned. The called function (or independent process) 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 to 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 request information is transferred to the operating system, 2) the service is performed, and 3) the record is returned back 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. The span of code from the point where the initiation of the data request is made for a independent function call to the point where the return data is received is called 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.

As with any research effort, there are issues that need to 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 answered. 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.

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 and GELER90]. In addition, Linda has been 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 Hypercube [BJORN89a, BJORN89b, CARRI86a, CARRI86b, CARRI87 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 CARRI88a].

The Linda approach supports process creation and intercommunication through a shared data/process repository called Tuple Space (TS) [CARRI87, CARRI89a, GELER85a and GELER85b]. Linda provides operations to generate data tuples (OUT), to read data tuples (RD), and to remove them from TS (IN). Tuple Space not only contains data tuples but also process tuples (created with the `eval` operation) which are often called "live tuples." These process tuples are instantiated and are 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.

---

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.

Several implementations of Linda utilize 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 program. 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. This is due to the taxing communications overhead of transferring tuple structures and data across a network.

It becomes apparent rather quickly 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 explicit primitives - one for *initiating* 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 is to:

1) Provide the primitives for the initiation and retrieval routines for `IN` operations,
2) 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 concentrates on determining two pieces of information for an independent function call:

1) The earliest point in the code to safely initiate a data request for the call and,
2) The latest point that the call's return data can be safely retrieved.

Section 2 provides background research relevant to this investigation. Section 3 presents the instructional footprint model which describes how instructions in general can be safely moved around in a program. Sections 4 and 5 provide some preliminary results and describe the research plan for this investigation.

2 Background

2.1 Related Work

This research effort involves the use of similar techniques 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 as:

> *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 this proposed research on instructional footprinting 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 of means a 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 research 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 functional languages, is similar in some respects to futures and our research as well. With lazy evaluation, certain 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 were *lazily evaluated* (i.e. the evaluation was postponed until needed) do not need to be performed.

For example,

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

is a Lisp 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 *promises* to be used in conjunction with an asynchronous communication mechanism called *call-streams* [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 been applied to Lisp and Prolog [BANSA89, FRADE91, LARUS88b and RAMKU89], Fortran is primarily the 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 what iterations of a loop are independent of each other. Independent iterations of loops are placed on different processors and executed in parallel. However, different vectorization and loop parallelization are both areas that rely on extensive control and data flow analysis in order to create vector/parallel programs from serial ones.

Similarly, the determination of instructional footprints must rely on the same types of control and data dependency analysis. Much work has been done to determine data dependencies between instructions [BANER76, BANER78, 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 and FERRA87] are often used to accurately 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 initially considered in earlier research efforts, subsequent research has been directed towards interprocedural side-effects and procedure-induced aliasing, 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 [NIERY87] has on data flow analysis. Because the use of GOTOs are 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.

---

2 Horowitz 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."

3 Instructional Footprint Model

3.1 Introduction

The Instructional Footprint Model (**IFM**) is a tool for analyzing the interaction of program instructions. Similar to Berstein's conditions that are used to determine the interdependence (or independence) of processes [MAEKA87], **IFM** can be used to ascertain the *footprint* of instructions (i.e. how far back or forward in a program an instruction 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 increase 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 an instruction can be *safely* executed. The question is *what is safely executed.* The following example illustrates where the heel and toe of a given instruction can be *safely* placed for program execution.

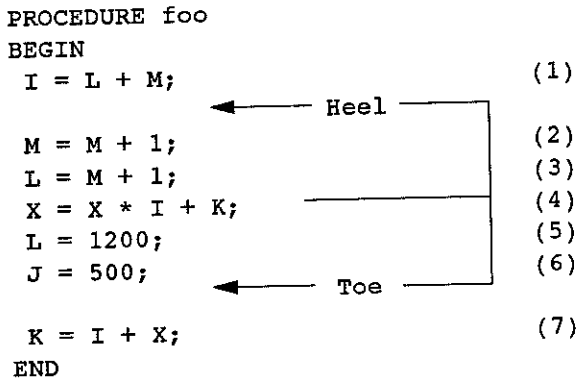

Figure 1. Example of Instruction Footprint.

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 criteria 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 1. The placement of instruction 4 obviously cannot leave 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 2. Example of a Complicating IF.

For example, in Figure 2 the heel of instruction 4's footprint cannot be placed between instruction 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 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 program 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 heel and toe of the footprint 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, GOTOs and pointers are handled both by compile-time analysis and with run-time extensions.

3.2 The CL Language

Because the IFM model 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, a generic language is employed. 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.

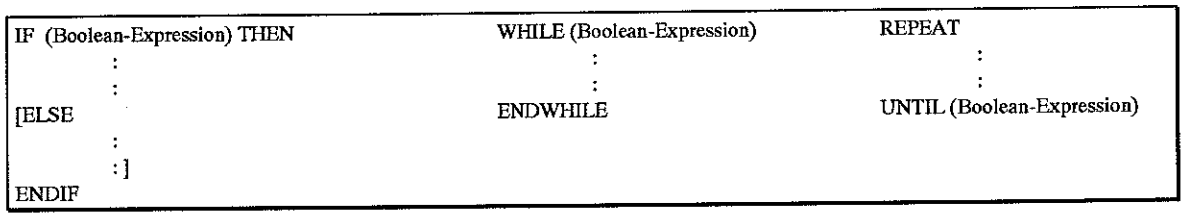

Figure 3. Constructs of CL.

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 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.

The figure in Appendix A 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 located. The IFM model does not require the use of an actual programming language (such as C), but rather one can reason about safety with respect to code movement 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 model.* 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 5 shows examples of instructions (represented as nodes) as they relate to the BLOCK, IF, WHILE and REPEAT statements in CL.

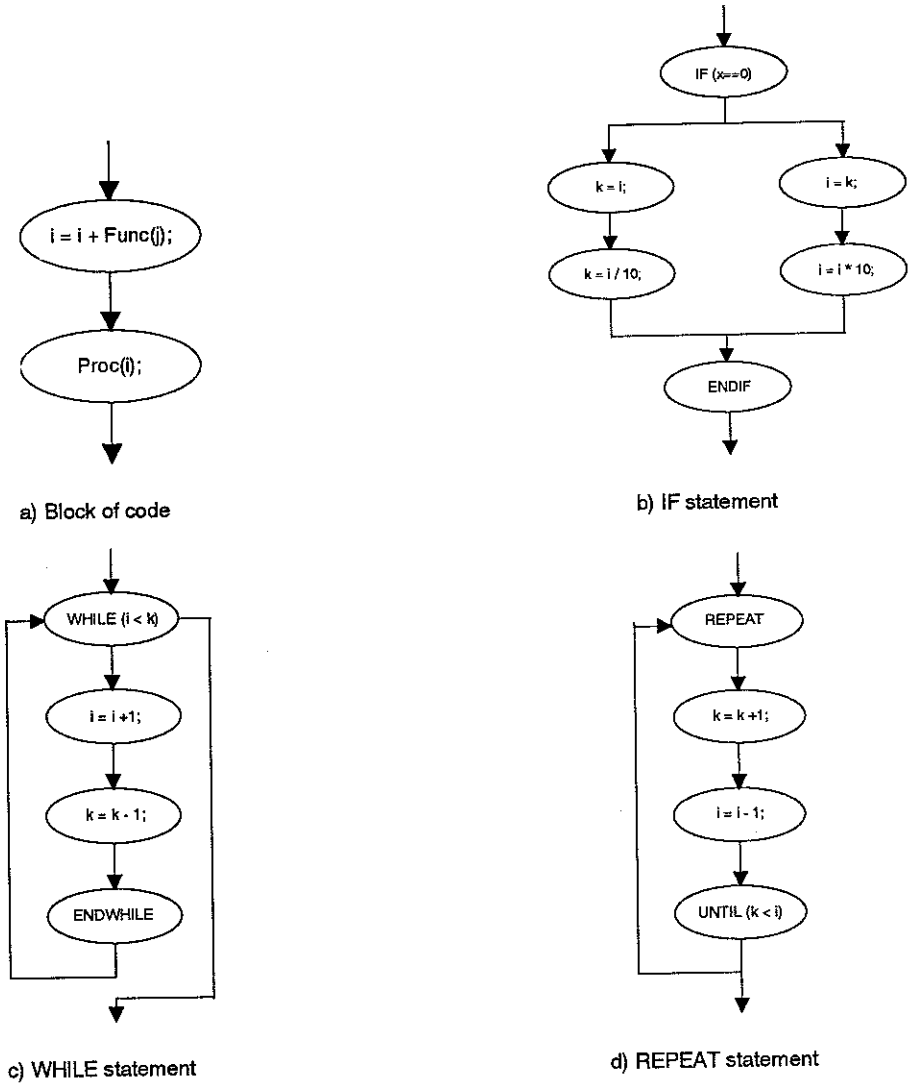

Figure 4. Sample Instruction Graphs of Code Segments.

As seen in Figure 4a, 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 4b. The reason is that the distinction between the THEN and the ELSE parts is not important to the development or use of the **IFM** model. 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 both **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 focused to define only 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 the instructions in program **P**.

C - An unordered set of control flow pairs. These control flow pairs describe the entire flow of control for **P**. For example, a control flow pair (**1 2**) means that instruction **2** can be executed immediately after instruction **1**.

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 5. 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 all components of program **P** that are considered an instruction. Recall that all assignment statements and procedure calls are considered instructions, as well as, 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 instruction immediately following **i** in lexographical order. This construction changes when IFs, WHILEs and REPEATs 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 **i** and the set **C**, these functions will return a set of instructions. 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 instructions that can immediately precede **i** in execution. The following are the formal definitions of τ and ϕ.

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

Consider the example procedure in Figure 5. In applying τ and ϕ to instructions 2 and 5 respectively, $\tau(2,C)$ returns {3 5} and $\phi(5,C)$ 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** model (see Figure 4 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** model. An instruction in the **IFM** model 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 set 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 identifications are elements of the component instruction set. These three set are defined as:

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

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

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

In the **IFM** model, 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 instructions that are being considered a single instruction.

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

WHILE - For the WHILE instruction, π_{WHILE} contains three instructions. The first instruction is the WHILE condition. This is followed by a single (possibly aggregate) instruction representing the WHILE body. The last instruction is the ENDWHILE.

REPEAT - For the REPEAT instruction, π_{REPEAT} contains three instructions. The first instruction is the REPEAT. This is followed by a single (possibly aggregate) instruction representing the REPEAT body. The last instruction is 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 { **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. A formal definition of the aggregation function is provided in Appendix B.

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 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.

```
PROCEDURE bar                                  I = { 1  2.4  5 }
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; (footprint inst)   (5)            π2.4 = { 2 3 4 }
--- Hard Boundary ---
END
```

Figure 6. Example of the Aggregation of the WHILE from Figure 5.

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) }. A formal definition of the deaggregation function is provided in Appendix C.

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 notedly are the boundaries imposed by functions. The footprint of an instruction must remain within the BEGIN-END boundaries of the defining function. This class of boundaries is called hard boundaries and is 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 cannot conflict with the write set of another. 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. Conversely, 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 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 7 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;  (footprint Instruction)           (6)
      L = K * J;                                (7)
---- Hard Boundary ----
END
```

Figure 7. Example of a soft boundary placed inside and IF statement.

Suppose now that an instruction to be footprinted conflicts (by use of Theorem 1) with an instruction within a WHILE or REPEAT (see Figure 6). 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 the instruction (depending on the direction the instruction is being moved). Figure 8 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.

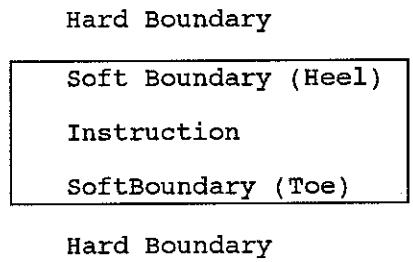

Figure 8. Relative Positions of an Instruction to its Boundaries .

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 between 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.

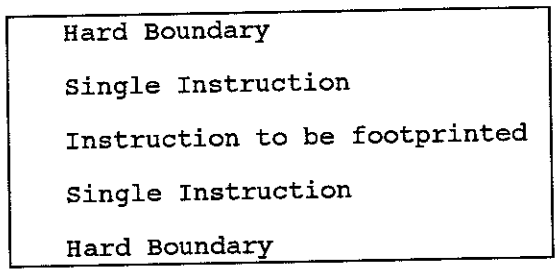

Figure 9. Aggregation of Code Segments to Single Instructions .

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 sequence of instructions are aggregated is critical. Given the set of instructions shown in Figure 10, 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; (Instruction to be footprinted)
      Next = X + 1;-------------------------------------+
      Prev = Prev - 1;                                  (1)
      NewOne = Value + 12;-----------------------------+
----- Hard Boundary -----
END
```

Figure 10. 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 11 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 11. 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 12 outlines the footprinting algorithm that determines soft boundaries (i.e. the heel and the toe) 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 π_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 12. 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.

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 of referencing 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** model (with respect to function calls) is in determining what side-effects and potential aliasing are cause 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 minor exceptions of the `IF`, `REPEAT` and `WHILE`, the instructional footprint model so far has assumed a sequential flow of control. In other words, `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 13, 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 13. 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 positions 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, if the two instructions were to be swapped, it is not known whether the integer pointer `ip` points to the integer `i`.

```
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. Therefore, it cannot be determined where a pointer variable is pointing. In order to see what, if any, run-time extensions can be provided, consider Figure 14 which shows the soft boundaries (the heel and the toe) relative to the original instruction position.

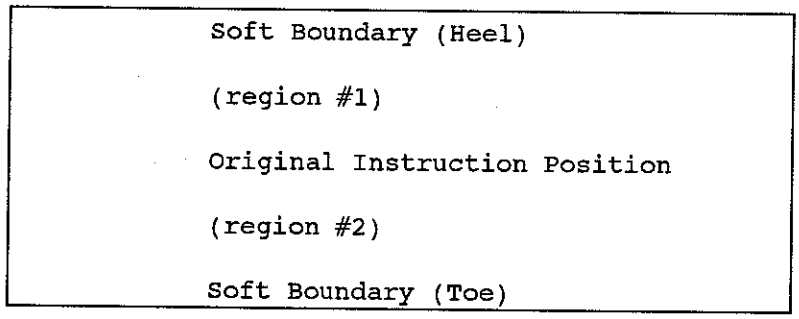

Figure 14. Soft Boundaries Relative to Original Instruction Position.

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).

4 Preliminary Results

4.1 Linda and the IFM model

Recall that this research is motivated by the lack of speed in Linda programs. The primary bottleneck in Linda programs is accessing tuple space. Frequently, the tuple space manager (the kernel) is implemented as a separate process with services being requested by the Linda program and its EVALed processes. Therefore, when an IN is executed, a template is sent to tuple space requesting a matching tuple; the requesting process must block until the tuple is returned. The RD operation is similar to the IN, but differs in that the returned tuple is left in tuple space. Predicate versions (INP and RDP) exist for both the IN and the RD operations. INP and RDP are non-blocking operations (unlike IN and RD) that provide a return value indicating whether a matching tuple was found or not.

The **IFM** model can be applied to the IN, RD, INP and RDP operations of Linda in an attempt to initiate the request for a tuple early and postpone the receipt of the tuple until it is actually needed. Unlike the IN and RD operations, the INP and the RDP operations return whether the operation was successful or not.

4.2 Description of the Experiments

Two experiments have been performed in order to prove the feasibility of this research effort. The intent of performing these two experiments is to show that some Linda programs can realize significant speedups while other do not. As one of the primary goals, this research investigation will endeavor to understand which Linda programs are and are not amenable to speedup and why.

In both experiments, programs are analyzed (according to the **IFM** model) and timings are taken for the original (un-optimized) program and then again for the optimized one. The first experiment is the dining philosophers problem and the second is a program simulating a distributed and redundant database system. Both programs were written as class projects in the graduate operating systems class in the Fall of 1991.

Two AMIGA 3000UX workstations are used for both experiments. One of the machines is used solely for the tuple space and its manager (the kernel), while the other workstation is used to run the Linda programs and the EVALed Linda processes. Communication between the kernel and the Linda programs

(processes) occurs through sockets across a network. In order to obtain consistent and meaningful results, the experiments are run when the workload on both machines was low - no other users logged on.

4.3 The Dining Philosophers Problem

The dining philosophers problem is a classic problem often used to measure the expressiveness (or lack thereof) of a parallel programming language. Although this research is not intended to prove the expressive capabilities of the Linda language, the dining philosophers problem is used for two reasons. First of all, the dining philosophers problem and its solutions are generally known and readily understood by many researchers. The second reason is that program structures or techniques found in solutions to the dining philosophers problem typically can be found in solutions to more "real world" problems.

In this experiment, each philosopher is an EVALed Linda process that simulates a series of process cycles. A process cycle consists of thinking, acquiring a "room ticket,"[3] picking up two chopsticks, eating and then replacing the room ticket and the chopsticks. The code for the philosopher routine is

```
while (ProcessCycles > 0) {
      think();
      in("Room Ticket");
      in("Chopstick", Phil_ID);
      in("Chopstick", Phil_ID % Num_Phil);
      eat();
      out("Chopstick", Phil_ID);
      out("Chopstick", Phil_ID % Num_Phil);
      out("Room Ticket");
      --ProcessCycles;
}
```

The time spent thinking and eating by each philosopher is configurable in terms of how many seconds to sleep, and for this experiment was set to zero. In the optimized version of the above routine, the three `IN`s are initiated before thinking because the `IN`s do not conflict with the code in the `think()` routine. Although the 3 `IN`s do not conflict with the `eat()` routine (according to the **IFM** model), the receive operations are not moved past the eating routine in order to preserve the intent of the problem.Initially, the dining philosophers program is run with 5 philosophers and 5 process cycles. In this run, the three `IN`s of the philosopher routine are executed 25 times accounting for a total of 75 `IN` requests

[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 of where all philosophers want to eat at the same time, and each pick up their left chopstick and wait forever for their right chopstick.

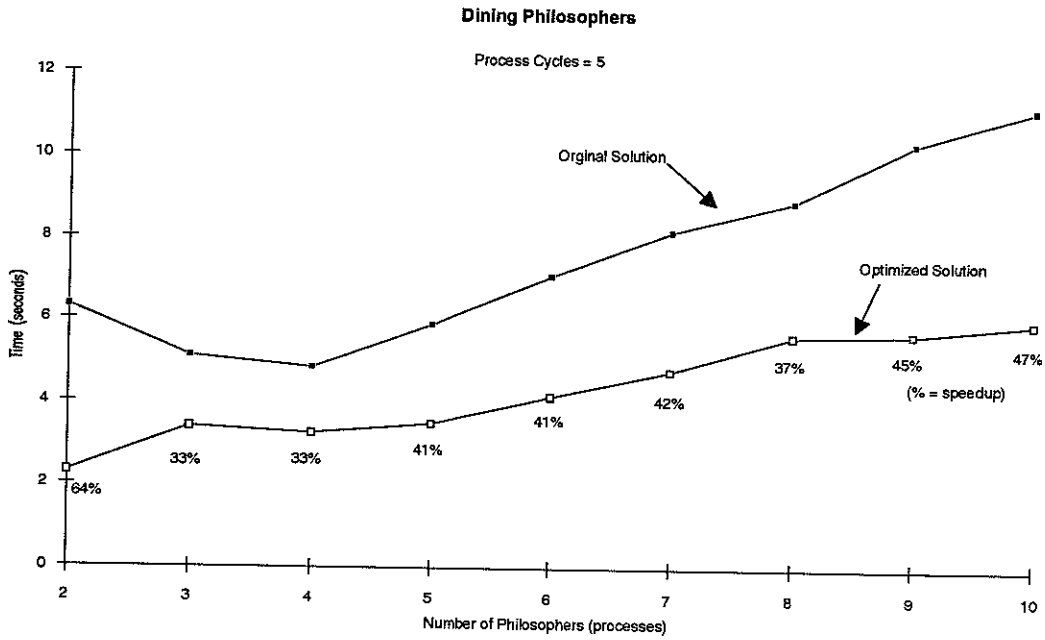

Figure 15. Graph of Dining Philosopher's Execution Times vs Number of Philosophers.

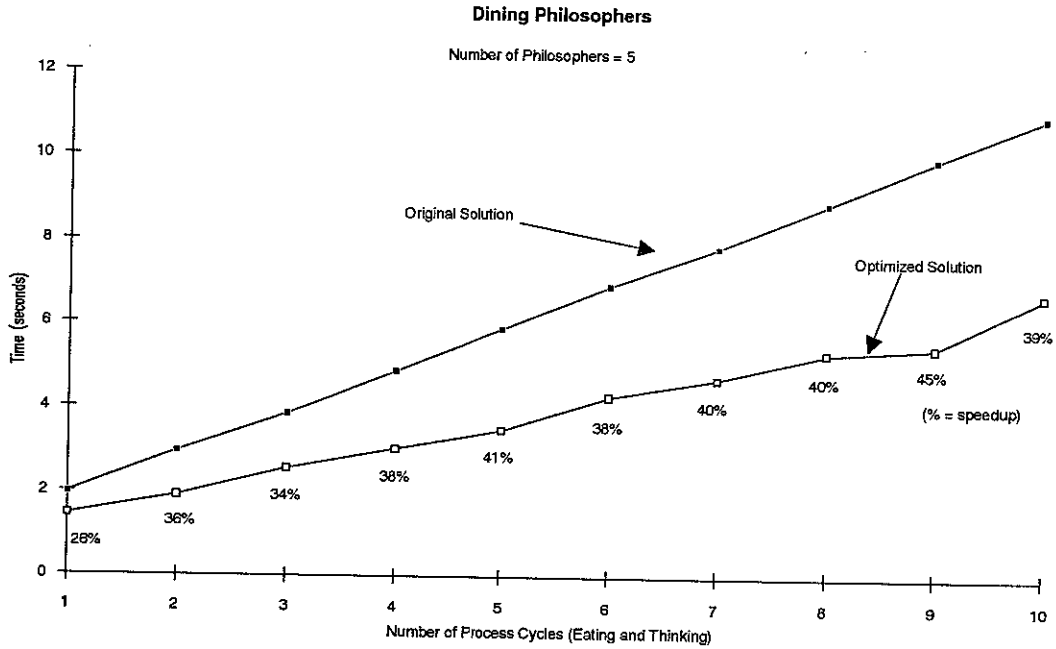

Figure 16. Graph of Dining Philosopher's Execution Times vs Number of Life Cycles.

made to the kernel. The program is run 10 times and the average execution time is taken. The speedup from the original program to the optimized program is approximately 41%. This speedup shows a positive effect of initiating IN operations early. At this juncture, however, we must ask: could early initiation of IN operations have a detrimental effect on performance? The easy answer is maybe. Consider the philosophers routine, it may be the case that a philosopher successfully makes the requests for the INs and then thinks, and thinks and thinks. Meanwhile, another philosopher who only thinks for a short period of time becomes blocked because the other philosopher has the chopsticks and is still thinking. In this scenario, it entirely possible to have a detrimental effect on performance. Although, this did not happen in the experimental run. Nevertheless, this issue needs to be analyzed further in this research.

Several other runs were made with the dining philosophers program where the number of philosophers and the number of process cycles were changed. Figure 15 shows a graph plotting the execution times versus the number of philosophers. The number of process cycles for this series of runs is held constant at 5. The relative speedups are show below the optimized solution line. Notice that as the number of philosophers (EVALed processes) increases, the speedup also increases. Also, notice that both lines have "inconsistencies." The line representing the original solution shows a drop in execution time in going from two philosophers to four and then proceeds to rises in a linear fashion afterward. In contrast, the line representing the optimized solution does not show this initial drop but rather contains a couple of plateaus. Further analysis is needed (with possibly more data) in order to explain the sometimes non-linear results.

Figure 16 shows a graph plotting the execution times versus the number of process cycles. The number of philosophers for this series of runs is held constant at 5. It is interesting to note that these plotted lines have far fewer inconsistencies than do the lines in Figure 15. This could be due, in part, to the consistent number of processes in this series of runs. In addition, the speedup percentages increased at a slower rate in Figure 16 than in Figure 15 although the number of executed INs were the same at respective data points.

4.4 The Distributed Database Problem

Another experiment performed is a distributed and redundant database simulation written for a term project in the graduate operating system class in Fall of 1991. The simulation consisted of three sites (EVALed processes) each holding a database. The system consisted of a total of three records each of which were duplicated at exactly two sites. The system provided ten different types of transactions (a

mixture of reads and writes of different records) and each site performed exactly two transactions each. The types of transactions performed by each site are random, as well as, the simulated processing time of each site.

The source code for this simulation is analyzed and it is found that only two routines could be optimized - the routines for gaining exclusive and shared record locks. In each lock routine, approximately 1 `IN` operation was optimized by initiating the `IN` before a `printf` routine. Due to the randomness present in the program, the speedup varied; ranging from no speedup to approximately 10%. The algorithm for the two record locking routines is

```
x_lock(int record, int site) {
    get a site ticket.
    wait for ticket to be called.
    gain an exclusive lock.
    update site ticket counter.
}
```

```
s_lock(int record, int site) {
    get a site ticket.
    wait for ticket to be called.
    if record is already shared
       update share counter in TS.
    else
        update lock counter in TS.
        update share counter in TS.
    endif
    update site ticket counter.
}
```

Why such a discrepancy in speedups between this experiment and the one performed for the dining philosophers problem? First of all, it is not the realistic expectation of this research investigation to achieve speedup with every program. Moreover, we fully expect to find programs for which speedup cannot be achieved. In fact, one of the goals of this research effort is to understand why it is that certain programs are amenable to optimization and why other programs are not. For the database experiment, any number of reasons could account for the lack of speedup, such as:

- The number of `IN`s executed in this experiment was relatively low compared to the number executed in the dining philosophers problem (approximately 6 compare to 75 for 5 philosophers and 5 process cycles).
- The percentage of processing time spent in the lock routines (where the `IN` operations are) was only about 10% relative to the total processing time of the simulation. Compare this to the dining philosophers experiment where the processing time of the philosopher routine is the vast majority of the total processing time.
- In this experiment, only a single `IN` operation is initiated early in the lock routines whereas three `IN` operations were initiated early, one right after another, in the philosopher routine.

- The `IN` operation in the lock routines contained an extra parameter that had tuple data returned in it. None of the INs in the philosopher routine had tuple data returned.

These two experiments point out two important facts. First, speedup can be achieved in programs that apply the **IFM** model. This shows the feasibility of this proposed research. Secondly, performance increases are not realized in all cases; some programs, for any number of reasons, simply cannot be optimized. However, as exemplified by our two experiments, what is important is not only achieving speedup in some programs, but realizing that some programs or classes of programs obtain performance increases while other do not, and understanding why such behavior occurs.

5 Research Plan

5.1 Review of Preliminary Results

The results of the preliminary experiments described in Section 4 are encouraging. The dining philosophers experiment shows on average a speedup of 40%. The results from the distributed database experiment, however, are not as impressive. The speedup for this experiment ranged from 0% to approximately 10%. A number of possible reasons have been suggested in Section 4 for this discrepancy and part of this proposed research is to determine the actual causes of such disparity.

A conservative approach is taken in performing the preliminary experiments in order to obtain results in a timely manner. In the two experiments performed, it may be possible to increase the "level of detail" of the footprinting optimization. For example, the initiation or receipt of an IN/RD/INP/RDP operation never crosses other Linda operations. There may be times when it is "safe" to do so. Clearly, this previously considered "hard boundary" needs to be reassessed relative to such possibilities.

Given that some programs are susceptible to speedup and others are not, two primary avenues of research will be explored. The first area involves analyzing programming structures and techniques used in programs that provide significant or insignificant speedups. The second area of research is analyzing what classes of problem solutions tend to produce better speedup than others.

5.2 Programming Structures and Techniques

In deciding what program characteristics allow for significant (and insignificant) speedup, it is necessary to understand how certain programming structures and techniques affect the amount of speedup a program can obtain. If one can determine what features of a programming language greatly influence the ability to apply the **IFM** model, we will go a long way in being able to identify, apriori, programs that are and are not amenable to optimization. Some programming structures and techniques that will be addressed in this research include:

- the size and number of functions and function calls,
- array and structure references,
- the use of `IN` and `RD` operations in looping constructs,
- the relationship between `INP`/`RDP` operations and `IF`s, and
- the effect intrinsic functions have on the amount of speedup achieved.

Functional boundaries correctly limit the placement of the heel and toe of an instruction. Many programmers follow the software engineering practice of writing many short functions rather than a small number of large functions. This means the use of more functions and more function calls. This practice is good for maintainability and readability but can have a negative effect on the placement of an instruction's heel and toe. This also means that more inter-procedural analysis needs to be performed because more function calls are present in the program. On the other hand, improved speedups might be possible if the initiation and data receipt of Linda operations (e.g. `IN` and `RD`) can reach outside of the enclosing function. For example, if an `IN` operation in a function is being initiated early, it might be possible to initiate the `IN` before the function is even called. Thus, increasing the concurrency of computation between the Linda program and the kernel. In addition, if an `IN` cannot be initiated before a particular function call due to conflicts, it might be possible to step into the function and initiate the `IN` operation within the function (in effect, *deaggregating* the function call).

Often, arrays and structured variables hamper the ability to initiate a Linda operation early and receive the return data later. This is due to the inability to determine, at compile time, what parts of the array or structure are being referenced. Some researchers have ignored this problem while others have assumed the worst case scenario (i.e. there is always a conflict). This research will address this problem and propose an equitable solution.

The presence of `IN` and `RD` operations within loops is common. As noted in the description of the **IFM** model, loops imposes a hard boundary restriction, limiting the placement of the heel and toe for the

IN/RD operation to be within the loop. There are two common situations when an IN/RD operation is used in loops. The first is when the only instruction in a loop is an IN or RD operation. No optimization can be performed in this case. However, the entire loop can be footprinted as a single instruction. This can allow for the early initiation or postponement (of receiving the return data) of the loop that contain IN/RD operations. The second situation involves an IN or RD operation performed at the beginning or end of the loop. This limits the placement of the heel and toe respectively. In this case, the detrimental position of the IN/RD operation severely restricts the size of the their footprints.

INP and RDP predicate operations are often used as the condition for an IF statement. This is usually preceded by "setup" code for both the THEN and ELSE parts. Frequently, none or only a small portion of the setup code conflicts with the INP or RDP operation, thus, allowing for the early initiation of the INP or RDP operation. This is a case where potentially propitious speedups may occur.

Intrinsic functions (e.g. printf and strcpy) are, in some ways, easier to handle than user defined functions. In particular, instrinsic function have side effects that only affect the reference parameters. No global variables are referenced. This observation has significant beneficial ramifications when determining the heel and toe positions for Linda operations because of their non-coupling nature.

5.3 Classes of Problem Solutions

In [CARRI88b], Carriero and Gelernter describe three major conceptual classes of problem solutions used in parallel programming. These classes differ in the approach taken in developing a problem solution and hence, differ in the programming structures and techniques used in the solution. Because programmers start (or should start) thinking about a solution to a problem at the *conceptual approach* level and not at the programming structure/technique level, it is important to understand what impact this has on the applicability of the **IFM** model.

The three conceptual classes described by Carriero are the *result, agenda* and *specialist* classes. These three classes attack a problem from different angles or perspectives. Carriero describes these three classes as:

> *Result parallelism focuses on the shape of the finished product; specialist parallelism focuses on the make-up of the work crew [the functionality available in the problem]; agenda parallelism focuses on the list of task to be performed.*

A part of this proposed research is to analyze different classes of problem solutions in order to determine the typical programming structures and techniques used and what degree of speedup a program written using a particular conceptual class can be expected. Throughout this research effort, it is not only important to know which classes of problem solutions are amenable to the IFM speedup and which ones are not, but *why* they are or are not. Although the three classes mentioned by no means encompass all problem solutions, they do capture the majority and therefore, offer a good starting point.

5.4 Other Topics

The previous two sections described the two major components of this proposed research, however, additional topics also need to be addressed. The first topic is the order in which Linda operations are footprinted. The processing order does, in fact, have an impact on where the heel and toe of Linda operations are placed. It may be the case that two or three *passes* are needed in order to maximize the size of all or some footprints. In addition, it is quite possible in some situations to increase the size of an instruction's footprint by moving other instructions around in a program. For example, if an `IN` cannot be initiated before instruction **X**, then it might be beneficial to move (if possible) instruction **X** back in the code in order to initiate the `IN` operation as early as possible.

REFERENCES

[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. Tesis, University of Ill. at Urbana-Champaign, DCS Tech Report # UIUCDCS-R-76-837, October 1976.

[BANER79] U. Banerjee, "Speedup of Ordinary Programs," PhD. Tesis, 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 Perfomance of Hypercube Linda," *Research Report YALEU/DCS/RR-690*, March 1989.

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

[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.

[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.

[CARRI88a] N. Carriero and D. Gelernter, "Applications Experience with Linda," *Proc. ACM Symp. Parallel Programming*, July 1988.

[CARRI88b] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *Research Report YALEU/DCS/RR-628*, November 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.

[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.

[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.

[HORWI88a] 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.

[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.

[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.

[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.

[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 go to'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 - Construct Transformations from C to CL

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;

APPENDIX B - Aggregation Function

The aggregation function, `aggr()`, takes as input a set of instructions S, as well as, the sets **I** and **C** which represent the program containing the instructions 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
BEGIN
--- Hard Boundary ---
  I = K;                          (1)
  IF (I=0)                        (2)
  THEN
     I = 1;                       (3)
  ELSE
     K = I;                       (4)
  ENDIF                           (5)
  J = I; (footprint inst) (6)
  L = K * J;                      (7)
--- Hard Boundary ---
END
```

==========>

```
PROCEDURE foobar
BEGIN
--- Hard Boundary ---
  J = K;                  (1)
  IfInst;                 (2.5)
  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 instruction 2.5 must be added to **I**, while the set of instructions $\pi_{2.5}$ must be removed. For **C**, the set of control flow pairs { (2 3) (2 4) (3 5) (4 5) } representing the structure of the `IF` must 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 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 in the set **S**
ω_{AI} = union of all ω_i where **i** is any instruction 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 **not** is 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. In addition, S must be a sequence of either aggregate, assignment or procedure call instructions.

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

`WHILE` — The input set **S** is a set of three instructions. The first is the instruction for the `WHILE` condition. This is followed by a single (possibly aggregate) instruction representing the `WHILE` body. The last instruction in **S** represents the `ENDWHILE`.

`REPEAT` — The input set **S** is a set of three instructions. The first is the instruction for the `REPEAT`. This is followed by a single (possibly aggregate) instruction representing the `REPEAT` body. The last instruction in **S** represents the `UNTIL` condition.

APPENDIX C - Deaggregation Function

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

`deaggr( i, I, C )`

C_i refers to the set of control flow pairs for the component instructions 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

$I = I - \{ i \} + \pi_i$

for all pairs (**i** **x**) where **x** is an element of **I**

$C = C - (i\ x) + C_i$

also,

- for all pairs (**x** **i**) where **x** is an element of **I**

  $C = C + \{ (x\ F) \}$
- case 1: **i** is a `WHILE`

  for all pairs (**x** **i**) where **x** is an element of **I**

  $C = C + \{ (F\ x) \}$
- case 2: **i** is **not** a `WHILE`

  for all pairs (**x** **i**) where **x** is an element of **I**

  $C = C + \{ (L\ x) \}$