Instructional Footprinting: A Model for Exploiting Concurrency through Instructional Decomposition and Code Motion

Kenneth D. Landry and James D. Arthur

TR 92-35

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

July 22, 1992

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

Kenneth D. Landry
James D. Arthur

Department of Computer Science
Virginia Polytechnic Institute & State University

Abstract

In many languages, the programmer is provided the capability of communicating, through the use of function calls, with other separate, independent processes. This capability can be as simple as a service request made to the operating system, or 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 instructions. This paper presents research findings that entails the development of the instructional footprint model, an algorithmic framework in which to exploit concurrency in programming languages, and some preliminary results from applying the instructional footprint model.

1. Introduction

Many languages offer the capability to communicate with a separate, independent process to perform a service or a computation. The investigation discussed 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.

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

1 Linda is the product of a research project conducted by Gelernter and Carriero at Yale 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.

[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.

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) Automatically determine the optimally safe positions for the initiation and retrieval of an `IN`, and then
3) Place the initiation and retrieval routines at these positions.

To summarize, the focal point of this research is to aid in the transformation of synchronous calls to independent functions into asynchronous calls. Assuming the availability of mechanisms for initiating an independent function call and for later retrieving the return data, this research 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.

3. Background

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, remote procedure calls and code transformations such as vectorization and loop parallelization.

3.1 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 ability, given that an `in` or `rd` downstream must be executed, to initiate that `in` or `rd` operation early.

Tuple pre-fetch is often applied to `in` or `rd` operation in loops. Once it is known that an `in` or `rd` in a loop is guaranteed to be executed, say in the next iteration, the `in` or `rd` operation is initiated. The primary difference between tuple pre-fetch and instructional footprinting is the extent at which flow analysis is performed. Tuple pre-fetch is only interested in initiating Linda operation early, while instructional footprinting also addresses the benefits of delaying the receipt of tuple data until it is needed.

3.2 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 their synchronization. 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*.

3.3 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.

3.4 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. Much work has been done to determine data dependencies between instructions [BANER76, BANER79, BURKE90, LI90 and WOLFE90] including in-depth analysis of reference patterns involving subscript and structure variables [BALAS89, BURKE86, CHASE90 and LARUS88]. Array vectorization and loop parallelization are two research 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 relies on the same types of control and data dependency analysis.

4. Instructional Footprint Model

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.

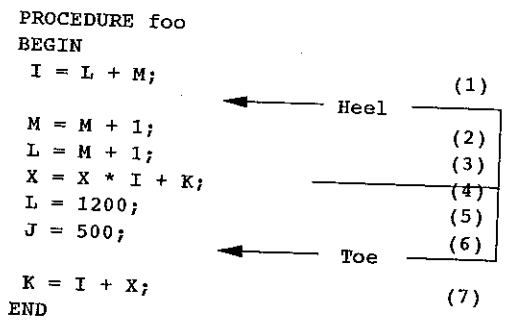

Figure 1. Example of Instruction Footprint.

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. 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 4.1.

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 4.2.

4.1 Model Description

In order to adequately model control and data flow in a program, the **IFM** model includes representations for instructions as well as programs. This section describes how programs and instructions are modeled as well as provides definitions for hard and soft boundaries. In addition, the aggregation and deaggregation functions are described.

4.1.1 A Program in the IFM Model

A program in the **IFM** model 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 4 shows examples of instructions (represented as nodes) as they relate to the `BLOCK`, `IF`, `WHILE` and `REPEAT` statements.

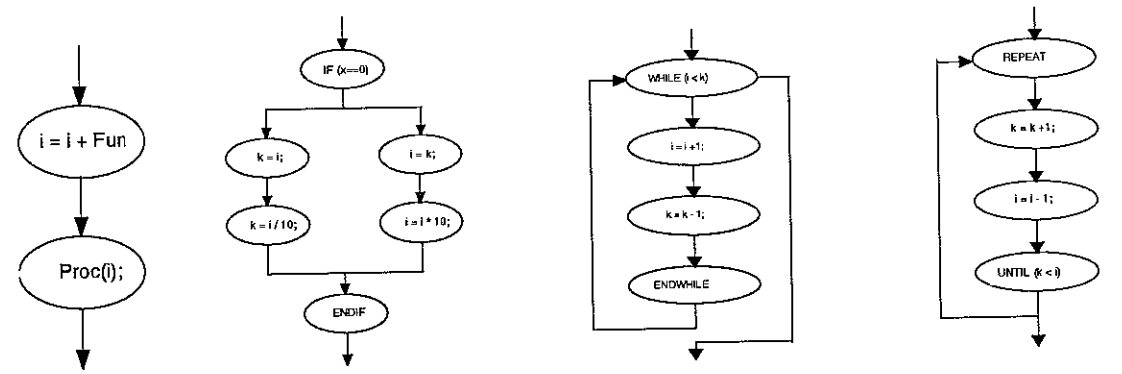

Figure 4. Sample Instruction Graphs of Code Segments.

As seen in Figure 4a, assignment and procedure calls are considered instructions in the **IFM** model. 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 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.

Effectively, for a given program **P**, the set **I** is constructed by including all components of program **P** that are considered an instruction. 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.

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}.

4.1.2 An Instruction in the IFM Model

Recall that 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.

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 the 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 }.

4.1.3 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 `IF`s, `WHILE`s and `REPEAT`s. 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.

4.1.4 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.

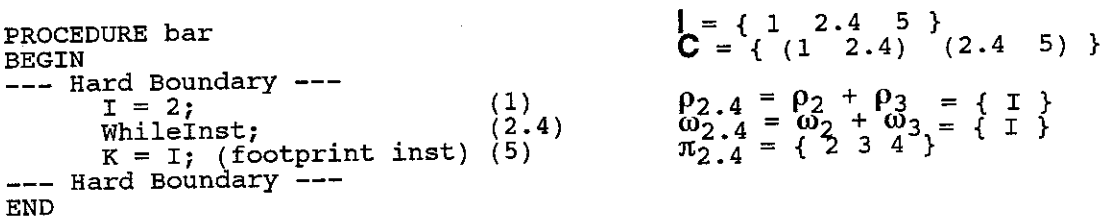

Figure 6. Example of the Aggregation of the WHILE from Figure 5.

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.

4.1.5 Hard and Soft Boundaries

In searching for the footprint of an instruction, the placement of the heel and toe is limited by certain boundaries. Most notably are the boundaries imposed by functions. The footprint of an instruction must remain within the BEGIN-END boundaries of the defining function. 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. In addition, other restrictions must be observed when determining a soft boundary involving `IF`s, `WHILE`s or `REPEAT`s. 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 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.

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). Effectively, hard and soft boundaries exist on both sides of the instruction being footprinted, and thereby, restrict the size of its footprint.

4.1.6 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.

With respect to function calls, the difficulty in the **IFM** model is in determining what side-effects and potential aliasing are caused by a function call. At compile-time, the **IFM** model can either assume the worst case scenario -- there is always a conflict -- or perform some form of interprocedural side-effect and alias analysis at either compile-time, run-time or both.

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. However, the use of GOTOs can be ignored. According to Ramshaw [RAMSH88], it is possible to transform a program with `GOTO`s into a functionally equivalent one without `GOTO`s. Therefore, any program with `GOTO`s can be transformed into a `GOTO`less program before applying the IFM model.

So far the model has disregarded the use of pointers, assuming all references are made to non-pointer variables. With pointers, it is impossible in most cases to determine if two instructions conflict when pointers are involved and therefore poses a difficult problem. The amount of compile-time analysis that can be achieved for pointers is minimal because the values of pointers are not known at compile-time. However, it may be possible to instrument the program being analyzed with run-time extensions in order to provide some form of dynamic pointer analysis.

4.2 Algorithmic Framework

The **IFM** model alone is not enough to model code motion; an algorithmic framework is needed to provide a basis for determining the footprint (the soft boundaries) of instructions. This section describes the aggregation setup process needed for determining an instruction's footprint. In addition, the algorithm for determining an instruction's footprint is also discussed.

4.2.1 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 IFs, WHILEs and REPEATs 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.

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 7, the aggregation order of the setup process proceeds as shown.

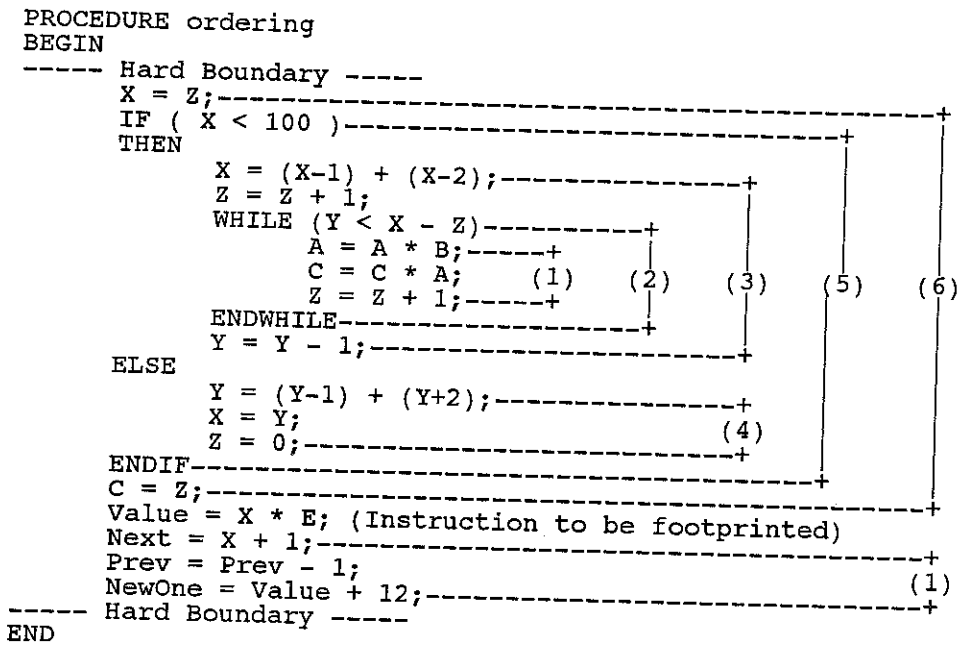

Figure 7. Example of the Aggregation Setup Process Order.

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`.

4.2.2 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, rules regarding conflicts of variables and the rules regarding soft boundaries within `IF`s, `WHILE`s and `REPEAT`s (described in Section 4.1) are applied and the footprint of an instruction is determined. In the footprint determination algorithm, the instruction being footprinted and its neighbor instruction are checked for variable conflicts to see if they can be swapped. If the swap is performed, then the footprinting process continues with the instruction to be footprinted and its new neighbor. Otherwise, the instruction type of the neighbor instruction is checked. If it is not an aggregate instruction, then the soft boundary is found. If the neighbor instruction is a `REPEAT` or `WHILE` aggregate instruction, however, the processing stops - the soft boundary is found. Otherwise, deaggregation occurs for the neighbor instruction and the footprinting process continues with the footprint instruction and its new neighbor. In the case where the neighbor instruction is an aggregate `IF` instruction, the footprinting process is performed twice - first for the `THEN` part and then for the `ELSE` part.

5. Preliminary Results

Two experiments have been performed in order to show the utility of instructional footprinting as a performance optimization. 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.

5.1 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 `EVAL`ed Linda process that simulates a series of process cycles. A process cycle consists of thinking, acquiring a "room ticket,"[2] picking up two chopsticks, eating and then replacing the room ticket and the chopsticks.

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 dining philosophers routine, three `IN`s (representing the room ticket and the two chopsticks) are initiated before thinking because the `IN`s do not conflict with the code in the `think()` routine. 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 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

---

2 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.

optimized program is approximately 41%. This speedup shows a positive effect of initiating `IN` operation early.

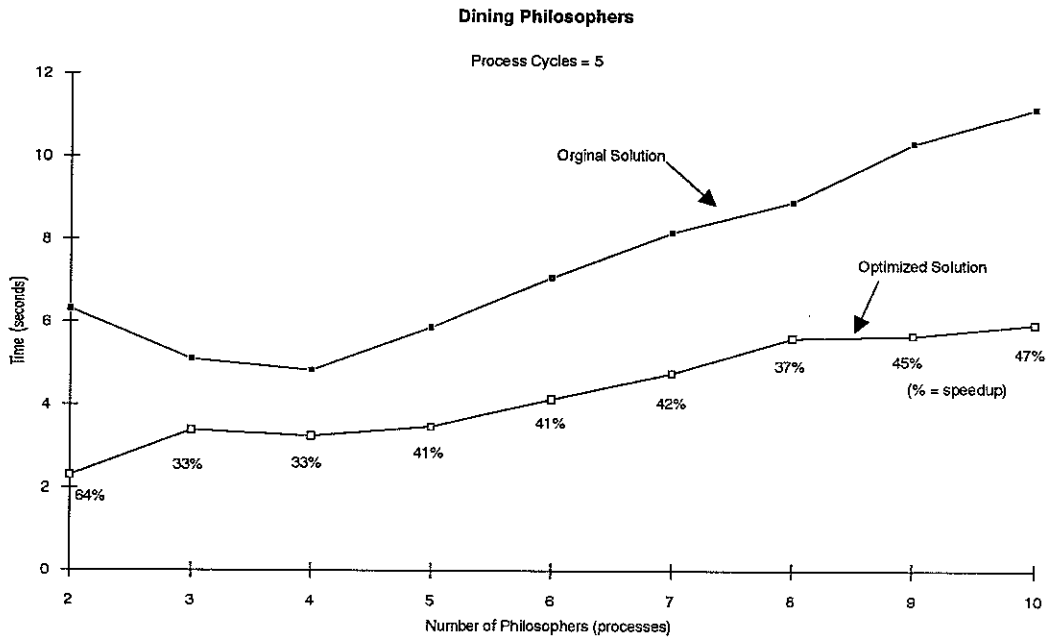

Figure 8. Graph of Dining Philosopher's Execution Times vs Number of Philosophers.

Several other runs were made with the dining philosophers program where the number of philosophers and the number of process cycles were changed. Figure 8 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 (`EVAL`ed 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.

5.2 The Distributed Database Problem

Another experiment performed is a distributed database simulation. 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 was 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%. 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, ranging from the number of times the `IN`s were executed to the fact that the `IN`s in the database program had return data while `IN`s in the dining philosophers program did not.

These two experiments point out two important facts. First, speedup can be achieved in programs that apply the **IFM** model. 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.

6. Conclusions and Future Work

This paper addresses the issue of program performance improvements through the use of instruction decomposition and code motion. Essentially, the goal is to transform synchronous, independent function calls to asynchronous ones. The approach presented in this paper uses an instructional footprint model to determine an instruction's footprint. This footprint can then be used to automatically transform a synchronous call into an asynchronous one. The preliminary results of this research effort demonstrate the utility of instructional footprinting. The dining philosophers experiment, experiencing speedups ranging from 33% to 64%, demonstrates the feasibility of the instructional footprinting approach. However, not all programs show such promising speedups. The distributed database experiment exhibited less than a 10% speedup when optimized, indicating that not all programs are amenable to optimization.

As future work, the effect that different programming structures and techniques has on achieving optimization speedup needs to be addressed. In addition, how speedup is affected by the use of different parallel programming paradigms needs to be addressed as well. Another area of future work has to do with the footprinting algorithm itself. Given that there are several instructions in a program to footprint, it is not clear what the "optimal" footprinting order is and whether or not it would require multiple passes. Not only should the relationship among instructions to be footprinted be analyzed, but the relationship between an instruction to be footprinted and all other instructions must be analyzed as well. Finally, the question of whether *program intent* is preserved when a program is optimized using instructional footprinting needs to be addressed.

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.

[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.

[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.

[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.

[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.

[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.

[LARUS88] J. Larus and P. Hilfinger, "Detecting Conflicts Between Structure Accesses," *Sigplan Notices,* Vol. 23, No. 7, July 1988, pp. 21-34.

[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.

[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.

[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.*

[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.