Instructional Footprinting and Semantic Preservation in Linda

Kenneth Landry and James D. Arthur

TR 93-22

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

June 14, 1993

Instructional Footprinting and Semantic Preservation in Linda

Kenneth Landry
James D. Arthur
Computer Science Department
Virginia Polytechnic Institute and State University

Abstract
Linda is a coordination language designed to support process creation and inter-process communication within conventional computational languages. Although the Linda paradigm touts architectural and language independence, it often suffers performance penalties, particularly on local area network platforms. Instructional Footprinting is an optimization technique with the primary goal of enhancing the execution speed of Linda programs. The two main aspects of Instructional Footprinting are instructional decomposition and code motion. This paper addresses the semantic issues encountered when the Linda primitives, `IN` and `RD`, are decomposed and moved past other Linda operations. Formal semantics are given as well as results showing significant speedup (as high as 64%) when Instructional Footprinting is used.

1. Introduction and Motivation

Parallel programming languages typically provide two process oriented activities - computation and coordination. Historically, parallel programming languages have been architectural and language specific. Concomitantly, equivalent process coordination primitives are not always available in such languages, and are definitely not the same on different machines. Linda [CARRI89b], on the other hand, does not suffer the architectural and language limitations of its predecessors. More specifically, Linda itself is a coordination language [CARRI89b, GELER92 and ZENIT90] rather than a conventional parallel programming language. This means that the semantics of Linda primarily address process creation and communication.

Additionally, Linda is not tied to any particular computational language; it can be (and has been) introduced into many different base languages to provide parallel programming capabilities. In particular, Linda has been embedded in a wide variety of languages including C++, Fortran, various Lisps, PostScript, Joyce, Modula-2, and Ada [BORRM88, CARRI90 and GELER90]. Not only is Linda language independent, it is also architecturally independent. Currently, many different architecture platforms are hosts for Linda [ARTHU91], including workstations such as Sun, DEC, Apple Mac II and Commodore AMIGA 3000UX, as well as a network of DEC VAX machines. In addition, Linda has also been ported to many parallel machines like the Sequent, S/Net and the Hypercube [BJORN89a, BJORN89b, CARRI86a, CARRI86b, CARRI87 and LUCCO86], including a machine directly supporting the Linda paradigm [KRISH87 and KRISH88].

Tuples and Tuple Space (TS) are the basis for process creation and communication in Linda. TS is an associative, process shared repository of information used to store data and process specifications [CARRI87, CARRI89a, GELER85a and GELER85b]. By virtue of being associative, multiple copies of identical tuples can co-exist in TS. Retrieval of such tuples is based upon the matching of types and values of tuple fields. Tuples are placed in TS with the Linda primitive `OUT`, are removed from TS with the `IN` operation, and copied with the `RD`. Processes are created and placed in TS with the `EVAL` primitive. The `IN` and the `RD` are blocking operations, and as such, will cause the requesting process to wait until matching tuples are found in TS. Linda also provides two non-blocking versions of the `IN` and the `RD`, called `INP` and `RDP`. These predicate versions return a status indicating whether a matching tuple is found, and if so, the tuple value(s) also.

The implementation of Linda (and in particular the Tuple Space component) may vary from machine to machine. Some systems have implemented TS in shared memory with each Linda process performing its own operations on TS. Other implementations utilize a separate process in order to manage TS [CARRI87 and SCHUM91] -- this is particularly true of network implementations of Linda [CARRI87, SCHUM91 and WHITE88].

Several implementations of Linda, on both shared and distributed memory parallel (MIMD) machines, have shown good performances [BJORN88, BJORN89 and CARRI87]. Nonetheless, performance often suffers on full-scale local area network configurations. The primary goal of the research described in this paper is to provide Linda systems (and particularly those implemented on network platforms) with the means of producing programs with acceptable, and often enhanced performance.

In support of enhanced performance, Instructional Footprinting [LANDR92] is introduced as an optimization technique with the goal of speeding up the execution of Linda programs. When applied to Linda programs, the optimization attempts to overlap (or parallelize) the normal computation of Linda programs with attendant processing associated with the TS manager (a separate process). This is achieved by initiating any `IN` or `RD` primitive earlier in the code and then receiving the returned tuple right before it is needed. The span of code between the early initiation of an `IN/RD` and the delayed receipt of the returned tuple is called the footprint of the instruction. An important part of footprinting in Linda is the instructional decomposition of the `IN` and `RD` primitives. For the purpose of footprinting, these primitives are decomposed into `INIT` and `RECV` operations to initiate a non-blocking request for a tuple and then to receive the returned tuple values, respectively. The following example illustrates how an `IN` is transformed into its two component operations.

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

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

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

1) The impact of moving `INIT/RECV` instructions past computational code, and
2) The impact of moving `INIT/RECV` instructions past Linda operations.

The first impact deals with the conflict of the read and write sets associated with program statements. To a certain extent, Berstein's conditions [MAEKA87] enable us to address this issue. Even so, pointers, function calls, and gotos, continue to complicate matters. A closer examination of this problem can be found in [LANDR92]. This paper, however, focuses on the second aspect of code motion, i.e., the effect of moving `INIT/RECV` instructions past other Linda operations.

In [LANDR92], one of the example programs that is optimized is a Linda program that solves the dining philosophers problem. Figure 1 shows the main code that a philosopher process executes.

```
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;
}
```

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

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

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

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

In Figure 1, each IN, acting as a semaphore, is initiated one right after another before the think() routine is called. Effectively, there is an implied sequence of operations associated with and among each of the IN primitives. For each IN, a request for a tuple is first made, with the Linda process blocking until the tuple is returned. This sequential nature is violated when INITs for INs (and RDs) are pushed past non-corresponding RECVs. The problem is that the TS manager does not guarantee that requests will be satisfied (and hence returned) in the same order requested. Suppose for example that the first INIT for the "Room Ticket" fails to find a matching tuple in Tuple Space. The request is shelved until potentially matching tuples arrive. Because an INIT does not block, the next INIT is processed, and if a matching tuple is found, the requested tuple is sent back to the Linda process *before* the first request for a tuple is satisfied. The requesting Linda process eventually blocks on the first RECV which is for the "Room Ticket". In this example, because there is no actual data being returned in the three tuples, and because Linda primitives are still serviced in order of their request in our implementation of the TS manager, the program still works properly.

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

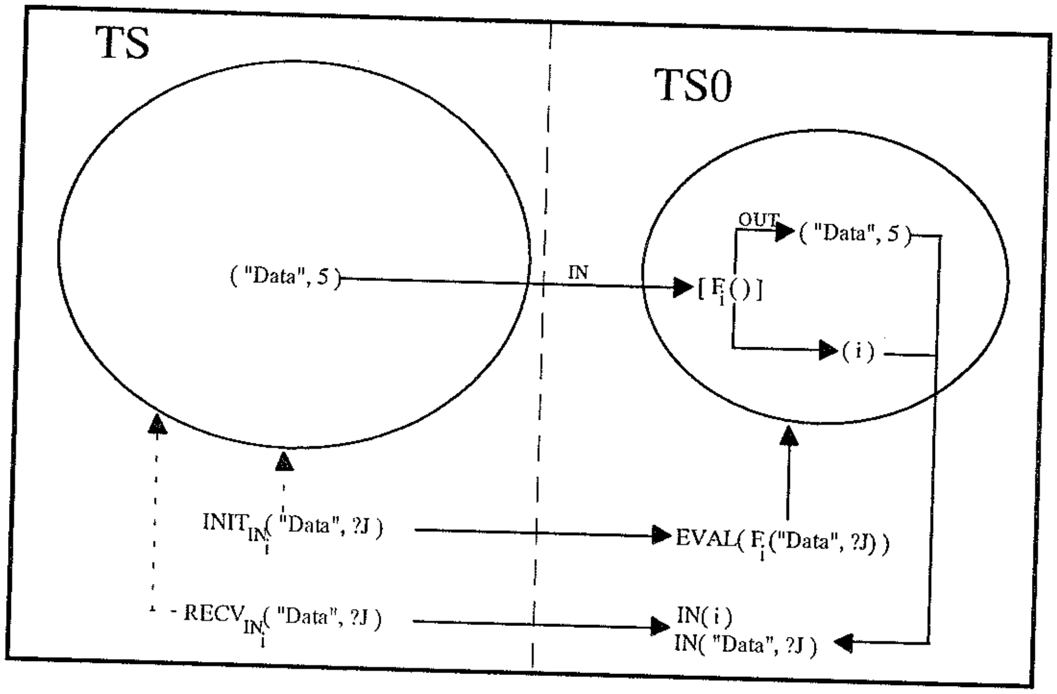
Recall that the goal of Instructional Footprinting is to speedup Linda programs. This speedup, however, must not come at the expense of sacrificing program semantics. So, why not simply disallow INIT and RECV operations from crossing over any Linda operation (or at least other INITs and RECVs)? As it turns out, when INITs are executed one right after another (as opposed to alternating INITs and RECVs) significant speedup can be achieved. The primary reason is implementation specific, and is directly related to the buffering mechanisms of sockets (many distributed systems implementing Linda use sockets for communication) [SCHUM91], and secondarily to the reduced blocking time of RECVs. In our experiments, speedups as high as 64% have been experienced when footprinting a series of IN operations results in the grouping together of the INITs and RECVs. Therefore, it is to our advantage to determine under what conditions one can push INITs and RECVs over other Linda primitives (or possibly, their decomposed counterparts) and exploit the benefits of maximizing the distance between INIT and RECV pairs.

The remainder of this paper is organized into 5 sections, each expanding on various aspects of the above premise. The next section formally describes the semantics of the INIT and RECV operations based upon the TS formalization work of Jensen [JENSE90]. Section 3 describes reasons why INITs and RECVs cannot be naively moved past certain Linda operations and presents when it is safe to do so. Section 4 defines Tuple Sequencing and Tuple Identification, and describes their use in preserving program semantics. Resultant speedups of several programs are shown in Section 5 followed by conclusions in Section 6.

2. The Semantics of INIT and RECV

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

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



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

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

Figure 3 shows an example of how the INIT and RECV for an IN are implemented with the use of TS0. The INIT behaves like an EVAL, while the RECV is perceived as two IN operations. The function being EVALed, $F_i()$, has the task of INing a tuple from TS (using the same tuple template) and then OUTing it to TS0. For example, in Figure 3 the tuple ("Data",5) is found in TS and then OUTed to TS0 by $F_i()$. By virtue of returning i, a second tuple is also placed in TS0 by $F_i()$ and has as its contents the unique designator value associated with i.

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

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

DEFINITIONS

$\mathtt{INIT}_{\mathtt{IN}}$

$$\frac{p \xrightarrow{INIT_{in}(Si)} p'}{TS \cup \{t'[p]\} \xrightarrow{INIT_{in}(Si)} TS \cup \{t'[p']\}}$$
$$TS_0 \xrightarrow{INIT_{in}(Si)} TS_0 \cup \{t''\}$$

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

$\mathtt{INIT}_{\mathtt{RD}}$

$$\frac{p \xrightarrow{INIT_{rd}(Si)} p'}{TS \cup \{t'[p]\} \xrightarrow{INIT_{rd}(Si)} TS \cup \{t'[p']\}}$$
$$TS_0 \xrightarrow{INIT_{rd}(Si)} TS_0 \cup \{t'''\}$$

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

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

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

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

where

S denotes a tuple template,

p denotes a processing environment,

t denotes a tuple,

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

t" is an active tuple whose job is to perform an IN(S_i) from **TS**, place the a new tuple in **TS0**, and then terminate leaving tuple { i } in **TS0**, and

t"' is the same as t" but it RDs from TS instead of INs.

The INs, RDs, and OUTs associated with t" and t"' (i.e. the function $F_i()$) have slightly different semantics than found in [JENSE90], but only in that they are dealing with both TS and TS0 rather than TS alone.

3. Code Motion and Linda Operations

Given the formalization of INIT and RECV operations in Section 2, it is now possible to better describe the code motion of INITs and RECVs relative to the preservation of program semantics. In this section we first present two examples of code motion that cause program semantics to be altered, followed by a description of a bifurcated classification scheme for these semantic violations. We also presents a discussion of the cases in which it is safe to perform code motion.

Example 1

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

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

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

Example 2

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

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

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

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

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

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

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

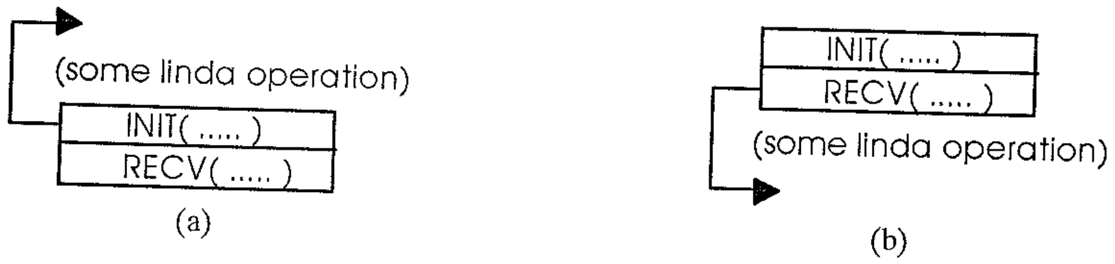


Figure 4. The code motion basics of `INIT`s and `RECV`s.

As Figure 4 shows, there are only two types of code motion being considered. The first type (4a) is moving an `INIT` up past some Linda operation and the second type (4b) is moving a `RECV` down past some Linda operation. The Linda operation (also called the secondary component instruction) can be an $\texttt{INIT}_{\texttt{IN}}$, $\texttt{INIT}_{\texttt{RD}}$, $\texttt{RECV}_{\texttt{IN}}$, $\texttt{RECV}_{\texttt{RD}}$, `OUT`, `EVAL`, `INP`, or a `RDP`.

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

```
(Some Linda Op)              (Some Linda Op)               INIT_IN(.....)
IN(.....)          =====>    INIT_IN(.....)     =====>     (Some Linda Op)
                             RECV_IN(.....)                RECV_IN(.....)
```

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

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

```
                             INIT_RD(.....)                INIT_RD(.....)
RD(.....)          =====>    RECV_RD(.....)     =====>     (Some Linda Op)
(Some Linda Op)              (Some Linda Op)               RECV_RD(.....)
```

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

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

```
IN("Some Tuple")                        INITRD("Some Tuple")
RD("Some Tuple")       =======>         INITIN("Some Tuple")
                                        RECVIN("Some Tuple")
                                        RECVRD("Some Tuple")
```

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

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

```
IN("Some Tuple")                        INITIN("Some Tuple")
OUT("Some Tuple")        ======>        OUT("Some Tuple")
                                        RECVIN("Some Tuple")
```

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

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

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

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

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

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

The third safe movement involves moving an INIT for a RD up past a RDP operation. The reasoning involved in considering this movement as safe stem from the systematic elimination of the three possibilities that can alter program semantics, i.e.

1) The alteration of TS,
2) The detection of tuple presence in TS, and
3) The blocking nature of certain Linda operations.

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

1) Is TS altered by either operation?

2) Is the detection of tuple presence affected?

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

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

The following table summarizes which INITs and RECVs for INs and RDs can be safely moved past other Linda operations. For example, the first entry in the last row is a NO, which indicates that a RECV_{RD} cannot be moved down past an INIT_{IN} without possibly altering program semantics. We can also view the same scenario as moving an INIT_{IN} up pas a RECV_{RD}. Its corresponding entry in the table also indicates such a move can have an adverse impact on program semantics.

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

Figure 5. Summary of acceptable INIT/RECV movements.

4. Tuple Sequencing and Tuple Identification

The table in Figure 5 paints a fairly bleak picture about the prospects of code motion as it relates to the movement of INITs and RECVs across Linda operations. In fact only 9% (3 out of 32) of the possible movements are safe. Is there a way to increase this percentage without sacrificing program semantics and still achieve speedup? The answer is a qualified - yes.

Recall that the motivation behind moving INITs and RECVs across Linda operations is to gain significant speedups by maximizing the footprint size. In our implementation of Linda - a distributed system using a separate process as a TS manager which employs sockets as the communication mechanism, it is possible to increase the percentage of safe movements from 9% to 75% through *Tuple Sequencing* and *Tuple Identification.* Tuple Sequencing is a technique which ensures that Linda operations in a particular process are executed (and completed) in the order they are *sent* to TS. Tuple Identification ensures returned tuples are matched up with the correct RECVs. The following table summarizes which movements are safe if Tuple Sequencing and Tuple Identification are used.

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

Figure 6. Summary of code motion using Tuple Sequencing and Tuple Identification

To illustrate the impact of Tuple Sequencing and Tuple Identification, it is helpful to analyze which cases of code motion are now safe to perform and which ones are still unsafe. It is also important to recognize that the basic problem encountered with both anterior and posterior temporal influences is that the order of operations explicitly laid out by the programmer in the unoptimized code is violated by the optimized code, which in turn, causes program semantics to be altered. Notice that in all cases where a NO (in Figure 5) is **not** changed to a YES (in Figure 6) involves moving an INIT. In all other cases it is changed to a YES because Tuple Sequencing and Tuple Identification enables us to guarantee that the original order of operations (or at least the original order of initiation) is preserved. For example, consider the following code.

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

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

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

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

5. Results

In order to show the effectiveness of Instructional Footprinting, three programs were executed with and without Instructional Footprint optimization. Tuple Sequencing and Tuple Identification were used, as needed, to preserve program semantics. The three programs are the dining philosophers problem, a distributed banking simulation, and a raytrace program. In all three cases, the reason for our observed speedup is because INITs and RECVs cross over other Linda operations. We also not that in each case, our footprinting algorithm groups several INITs together which apprears to have an additional positive impact on speedup. The reason why grouping INITs together for execution causes (sometimes dramatic) program speedup may be one of several reasons. Recall that these Linda programs are being executed on a network platform with TS implemented as a separate process using sockets for communication. We hypothesize that total speedup is due to (1) exploiting socket buffering parameters by executing several INITs together , and (2) reducing the blocking time of each RECV by maximizing the distance (of the footprint) between each RECV and its respective INIT in the program execution profile.

Dining Philosophers Problem

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

1) the solutions are generally known to most researchers, and
2) the programming structures and techniques used are common to solutions of real world problems.

This particular solution spawns (using the EVAL) N philosopher processes, each of which executes M life cycles where a life cycle is thinking, sitting down at the table, eating, and then getting up from the table. Each life cycle requires three IN operations (one room ticket[1] and two chopsticks) to be performed before the philosopher can eat, and then three OUTs (putting the INed tuples back into TS) to be executed after the philosopher is finished eating.

The code is optimized by taking the three IN operations and performing their INITs in immediate succession followed by their three RECVs. The following graph provides a comparison of the execution times for various runs of the original solution to those of their corresponding optimized versions.

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

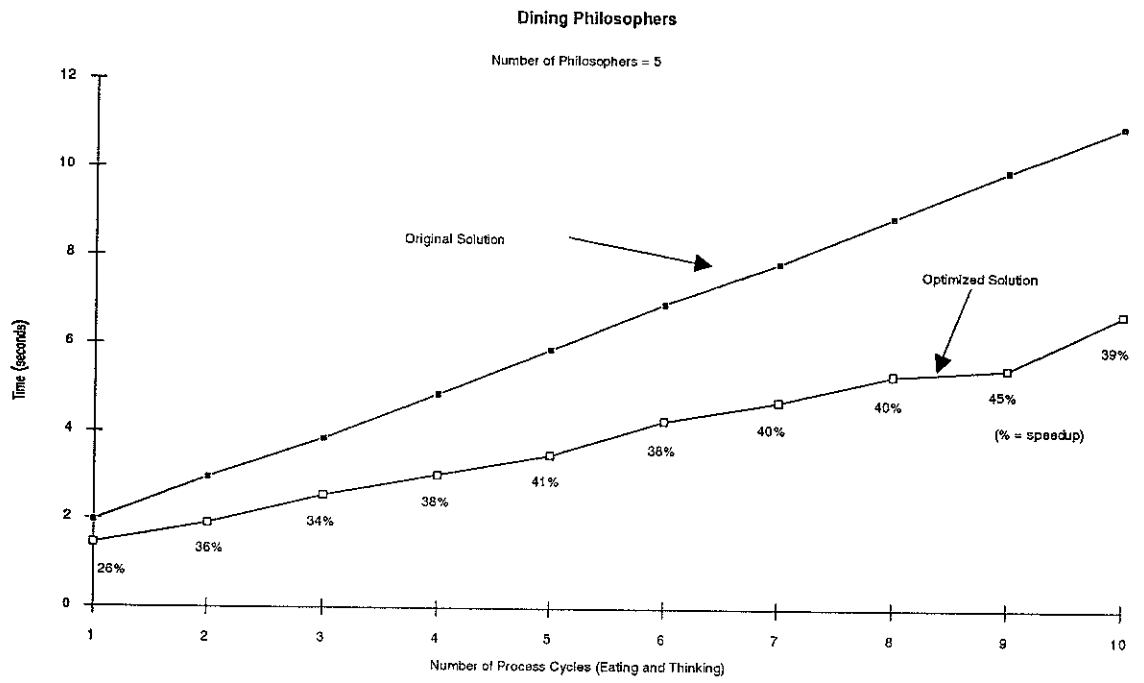


Figure 7. Graph of Dining Philosopher's Execution Times vs. Number of Life Cycles.

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

Distributed Banking Simulation

This Linda program simulates a distributed database of checking accounts, with pieces at each of three different banks[2]. The simulation takes several checking accounts (database records) and duplicates each at the different sites. The simulation then reads in a series of transactions for each site and posts them against the databases. The simulation spawns processes to manage the data at each site and to handle the transactions. The following graph shows the speedup achieved when optimized as compared to the original (unoptimized) solution.

2 This problem is similar to the one presented in [LANDR92]. In fact it solves the same problem but is written by another programmer and hence is slightly different. This difference allowed for better optimization than did the one presented in [LANDR92].

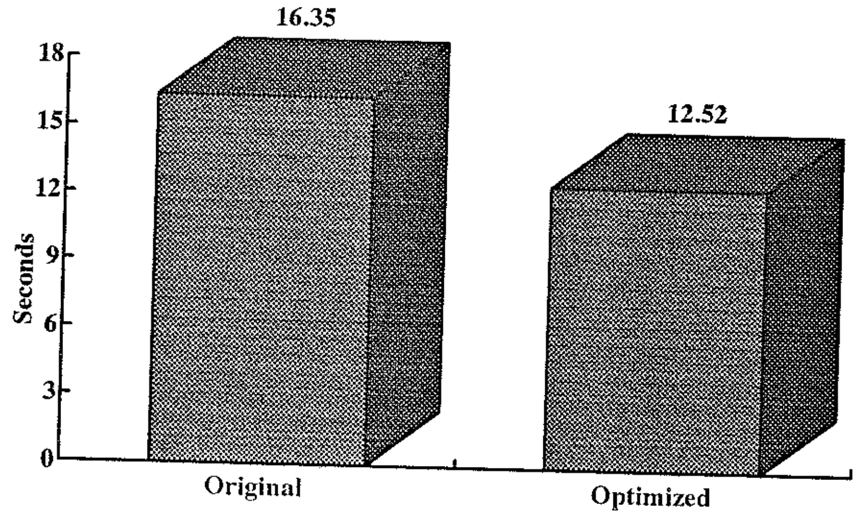


Figure 8. Execution Times for the Distributed Banking Simulation.

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

RayTrace Program

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

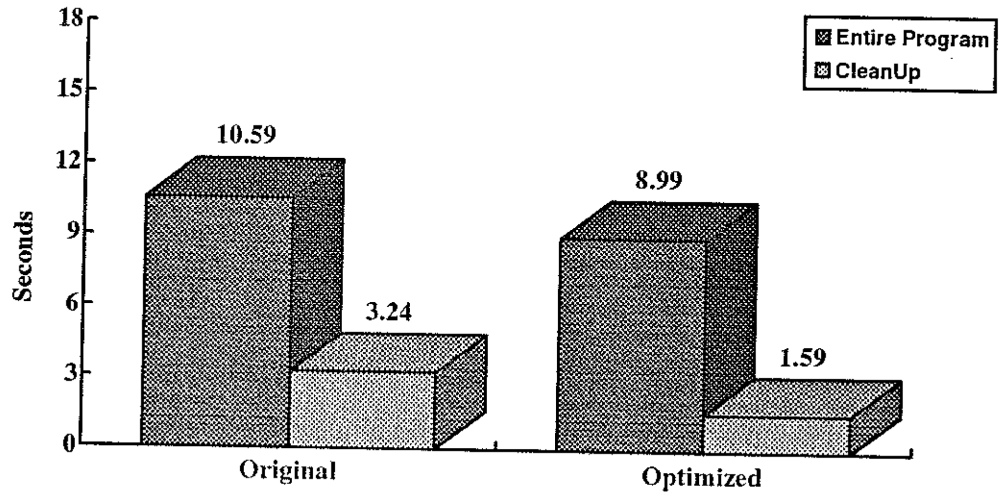


Figure 9. Execution Times for the Raytrace Problem.

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

6. Conclusions

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

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

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

REFERENCES

[LANDR92] Landry K., and Arthur J. D., "Instructional Footprinting: A Basis for Exploiting Concurrency Through Instructional Footpringing and Code Motion: A Research Prospectus," *Tech Report # TR 92-33*, Department of Computer Science, Virginia Tech, June 1992.

[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.

[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.

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

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