LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Loop Unfolding for a Static Dataflow Machine

Computation Structures Group Memo 262
May 1986

Edward H. Gornish

Submitted to the Department of Electrical Engineering and Computer Science on May 9, 1986 in partial fulfillment of the requirements for the Degree of Bachelor of Science.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Loop Unfolding
for a Static Dataflow Machine

by

Edward H. Gornish

Submitted to the Department of Electrical Engineering and Computer Science on May 9, 1986 in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Abstract

Loop unfolding is an important optimization for parallel computers. Loop unfolding involves copying the body of a loop one or more times, with the intention that the copied bodies execute simultaneously. A compiler uses several criteria to decide if multiple copies of a loop body can and should execute concurrently. These criteria are presented along with a discussion of loop unfolding. The Computation Structures Group of the MIT Laboratory for Computer Science has been designing a static dataflow parallel processing supercomputer, and a compiler has been written for the functional language VAL. This report describes an implementation of loop unfolding for the VAL compiler.

Acknowledgments

I would like to thank my thesis advisor, Professor Jack B. Dennis, for his guidance and support during the time I spent with the Computation Structures Group of the MIT Laboratory for Computer Science.

I would like to thank Bill Ackerman who worked closely with me on many aspects of my thesis.

Other members of the Computation Structures Group, including Natalie Tarbet, Andy Boughton, and David Marcovitz, have been an important source of information, and I would like to thank them for all the help they have given me.

Table of Contents

Table of Figures

Chapter One

Introduction

The Computation Structures Group of the MIT Laboratory for Computer Science has been designing a high speed static dataflow supercomputer. The front end of a compiler for the functional language VAL has been written, and current work focuses on the compiler's optimizer. In this thesis, I discuss the optimization of loop unfolding and its specific design and implementation for the VAL compiler. Loop unfolding is the process whereby an optimizer copies the main body of a loop one or more times and modifies the framework of the loop, in such a way that the loop still computes its original function. Using this optimization on a parallel computer means that if data dependencies between cycles of the loop are few or nonexistent, then most or all of the reproduced code can execute simultaneously.

Parallel computing is becoming the modern trend in computer science. To effectively exploit the capabilities of a parallel machine, compilers must be able to perform transformations such as loop unfolding. In particular, loop unfolding is one of the more fundamental optimizations of a dataflow computer [2]. Therefore, it is an important issue in the context of the static dataflow project.

Chapter 2 presents relevant background to the topic of loop unfolding, including a discussion of other loop optimizations that work closely with loop unfolding. Chapter 3 examines loop unfolding in-depth and describes various types of unfolding. The major section of the thesis is Chapter 4, which discusses the specific design and implementation of loop unfolding for the VAL compiler. Chapter 5 describes further, related work.

Conventions:

boldface	**Boldface** indicates constructs and nodes in VAL and Intermediate Graph Format.
italics	*Italics* indicates variables and terms being introduced for the first time.

Graphs may contain one of the following descriptors in their captions.

VAL	The figure contains VAL code.
IGF	The figure is displayed in the Intermediate graph format.
SF	The figure is displayed in the Simple Format.

Chapter Two

Background

I will give some background information on several areas of the static dataflow project, stressing how these areas relate to my particular research project.

2.1 Static Dataflow Computer

The Computation Structures Group of the MIT Laboratory for Computer Science has been designing a static dataflow supercomputer under the leadership of Professor Jack B. Dennis. The computer's specifications call for a hardware that supports highly parallel computations. While my particular research stems from this static dataflow project, its applications can be extended to a compilation system for any applicative language.

2.2 VAL

The VAL programming language [1] was developed primarily as a source language for highly parallel data driven machines, such as the MIT static dataflow computer. VAL is a *functional* or *applicative* language; i.e., it is value oriented and free of side effects. Such features make VAL an ideal source language for a compiler that generates parallel code. The VAL compiler, and in turn, the static dataflow computer attempt to exploit these features of VAL to the fullest extent.

The semantics of a VAL program are fairly obvious and unambiguous; I will, however, describe briefly some of the VAL constructs necessary to understand my thesis. Figure 2-1 shows a typical VAL **for** loop. Most of the loops that we will be

```
for X1, x2 := I1, I2 do
   if P1(X1, X2) then R1(X1, X2)
   elseif P2(X1, X2) then R1(X1, X2)
   elseif P3(X1, X2) then R3(X1, X2)
   else iter X1, X2 := S(X1, X2) enditer
   endif
endfor
```

Figure 2-1:VAL - Typical Loop

concerned with are **for** loops, with a body consisting of an **if** expression. In the loop in Figure 2-1, if $P_n(X_1, X_2)$ is true, then the whole **for** expression evaluates to $R_n(X_1, X_2)$. The arity of $R_n(X_1, X_2)$ and the **for** expression must be the same; in general, their arities can be greater than one. X_1 and X_2 are loop variables; their values are updated, and the **for** is instructed to repeat, via the **iter** expression. The **iter** expression takes a subset of the loop variables on the left side of the = and the new values to assign to them on the right side. In the example in Figure 2-1, $S(X)$ is assumed to return two values.

2.3 Compiler

The VAL compiler consists of the following components:

- parser
- linker
- optimizer
- code generator

- cell allocator.

Recent work has focused on the optimizer and code generator*. My research deals with issues in the optimizer.

2.4 Intermediate Graph Format

I use the VAL Intermediate Graph Format to illustrate many of the examples in my thesis. This format is used internally by the VAL optimizer, and it was developed with the aim of facilitating the optimizations that we wish to perform. The format is described in detail in [3]; here I will discuss the features that I will use in my illustrations.

A program displayed in the Intermediate Graph Format consists of a set of nodes (where each node represents a certain construct) and links representing data paths between the nodes. Inputs to a node consist of a number of *arms*, where each arm is comprised of a number of *args*. Outputs from a node consist of a number of *results*, where each result is comprised of a number of *arcs*. Each link connects an arc of one node to an arg of another node (these can be the same node). Figure 2-2 shows a sample graph. All simple constructs (such as **addition**, **array−select**) have one arm with two args, and one result with a number of arcs. The node in Figure 2-3 represents the expression $X+Y$. The result corresponds to the sum of X and Y, and there is one arc for each place the result is used in the graph.

The complex constructs that we will be concerned with are:

- **iterif**
- **for**.

---

*Dr. William B. Ackerman has been working on optimization, and Charles A. Goldman has been working on code generation.

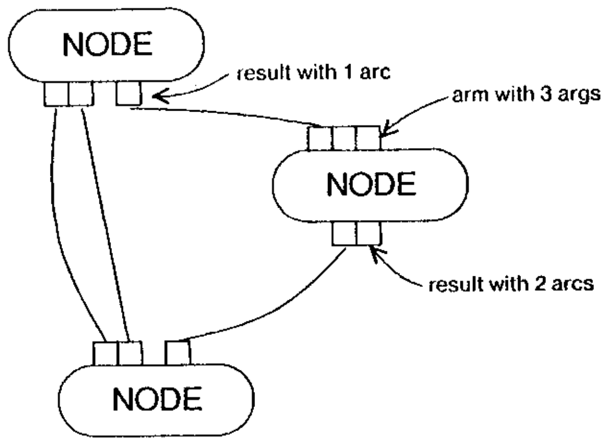

Figure 2-2:IGF - Sample Graph

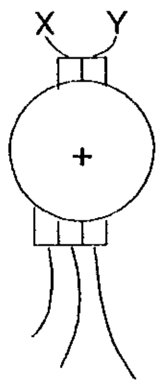

Figure 2-3:IGF - Addition Node

These constructs are explained in the examples in Section 2.4.1 and Section 2.4.2.

2.4.1 Iterif

Figure 2-4 shows a fragment of VAL code and the corresponding graph.

```
if P1(X, F2) then R1(X, F2)
elseif P2(X, F2) then R2(X, F1)
else iter X := S(X, F3) enditer
endif
```

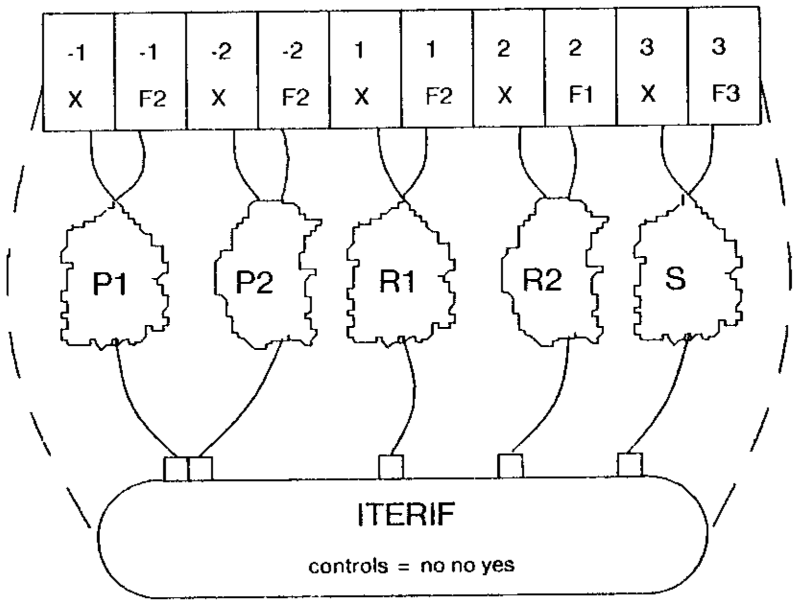

Figure 2-4: VAL, IGF - Typical **Iterif**

The VAL **if** and **iter** are combined into one construct, the **iterif**, when transformed into a graph. The boxes at the top of the graph are called *gates*, the numbers inside gates are called *cases*, and the alphanumeric characters in the gates represent the variables that can pass through them. The entire graph of Figure 2-4 is referred to as the **iterif** *body*. All inputs to the **iterif** body must pass through the gates; no links may cross the dashed lines that connect the gates and the **iterif** node. All inputs to the subgraph for the n^{th} clause of the **iterif** pass through gates with case n, and all inputs to the subgraph for the predicate enabling the n^{th} clause pass through gates with case $-n$. The outputs from the predicates, *P1* and *P2*, go to the leftmost arm of the **iterif** node. The predicates have arity one, and their outputs are either true or false. Each clause, *R1*, *R2*, and *S*, uses an arm of the **iterif** node. Each clause also has an associated *control*, which is one of the following labels:

yes — A *yes* clause or arm is one that will always iterate, such as the *S* clause. It is produced by an **iter** in the VAL source program. The arity of a yes arm is equal to the number of loop variables, and the values going into it are the new values to be assigned to the loop variables. I will also refer to a yes arm as an *iterating* arm.

no — A *no* clause or arm, such as *R1* or *R2*, is one that returns, or indicates that the loop is to exit. The arity of a no arm is equal to the arity of the **for** loop, and the values going into it are the values to be returned from the loop. I will also refer to a no arm as a *return* arm.

maybe — A *maybe* clause or arm may or may not iterate. The link to a maybe arm comes from another **iterif** node. This results from a nested **if** in the VAL source program, where the **iter** is contained within the inner **if**. The arity of a maybe arm is one. This single link is special; it represents either the new values to be assigned to the loop variables or the values to be returned from the loop.

In the internal representation of the Intermediate Graph Format, the gates, both inputs and outputs, are part of the **iterif** node. The inputs to the gates comprise arm 1 of the **iterif** node, the outputs from the predicates arm 2, and each clause comprises an arm starting with arm 3. The output of gate 1 comprises result 0, the output of gate 2 comprises result -1, the output of gate 3 comprises result -2, etc... Every **iterif** has an output result at 1. This output goes to either the maybe arm of another **iterif** node or to a **for** node.

2.4.2 For

Every **iterif** node is contained within the body of a **for** node. **For** nodes are similar to **iterif** nodes. They also have gates, all inputs must pass through the gates, and no links may cross a dashed line. The internal representation of the gates is the same as for an **iterif** node. **For** node gates can have one of two cases.

1. *1* - The input to the gate is a loop variable. Its value is updated each loop cycle.

2. *0* - The input is a free variable. Its value remains constant throughout the entire execution of the loop.

For nodes have one other input, which comes from result 1 of an **iterif** node. This input is arm 3; arm 2 is left blank for implementation reasons. Figure 2-5 shows a complete VAL **for** loop and the equivalent **for** body graph.

2.5 Other Loop Optimizations

The loop unfolding optimization works intimately with other loop optimizations. It is necessary to discuss two particular loop optimizations, for a full understanding of the theory behind and the power of loop unfolding:

1. progressing loop variables

2. successor variables

2.5.1 Progressing Loop Variables

A progressing loop variable is a loop variable that increases (or decreases) by some fixed increment in each loop cycle (e.g., a variable that increases by five each cycle, a variable that is divided by two each cycle). If such a variable exists, it is important for an optimizing compiler, such as the VAL compiler, to find this variable and incorporate the following information into the code for the program:

```
for X := I do
   if P1(X, F2) then R1(X, F2)
   elseif P2(X, F2) then R2(X, F1)
   else iter X := S(X, F3) enditer
   endif
endfor
```

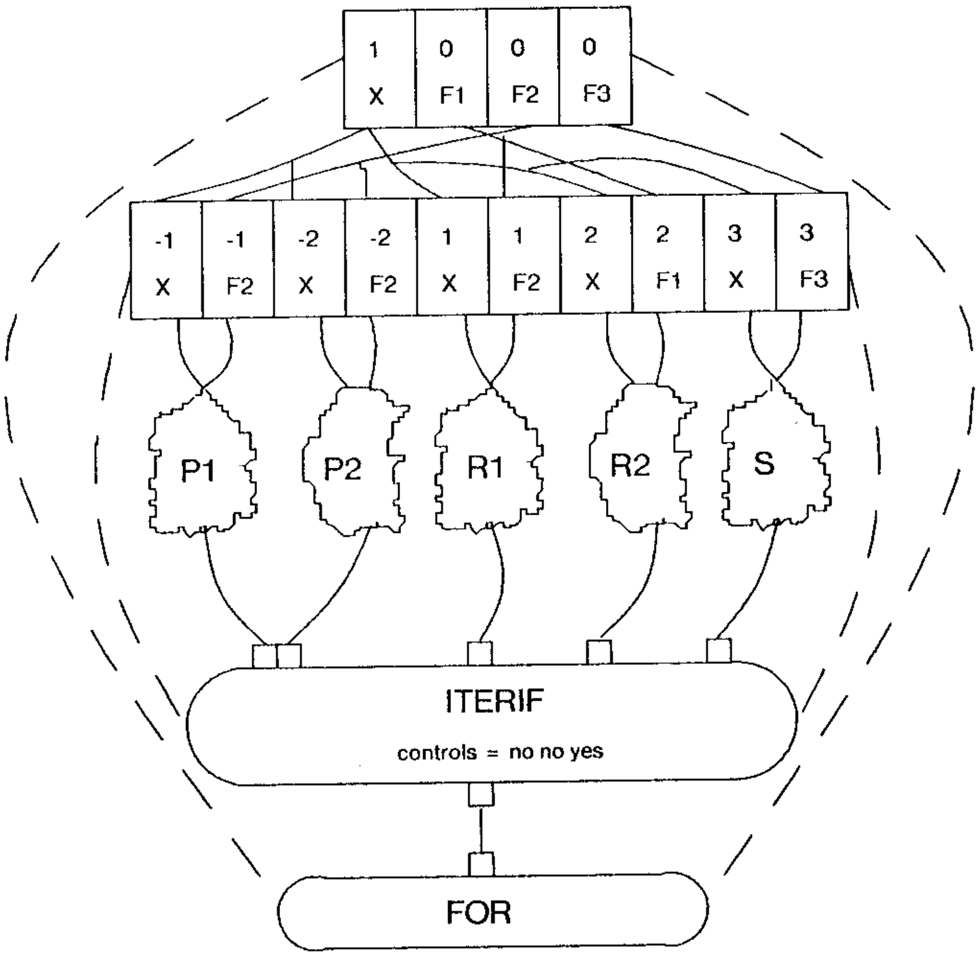

Figure 2-5: VAL, IGF - Typical For Loop

• the fact that it is a progressing loop variable

- the type of progression, e.g., addition, etc ...
- the increment by which it progresses

The VAL optimizer encodes this information through the transformation shown in Figure 2-6.

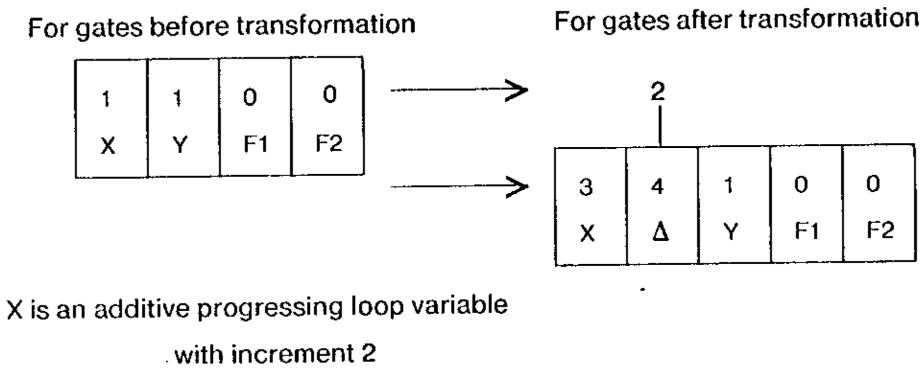

Figure 2-6:IGF - Progressing Loop Variable Optimization

This optimization is important for the following reasons.

- If all the loop variables are progressing loop variables, then we have eliminated *ALL DATA DEPENDENCIES* between successive cycles of our loop. The loop is now susceptible to total parallelism. The individual copies of the loop that execute in parallel are generated by loop unfolding. For this reason, probably more than 90 percent of the loops that we want to unfold have progressing loop variables.

- If the progressing loop variable is the index to an array access, then the code generator can turn the chain of array accesses into a chain of auto increments, provided that the targeted machine supports auto incrementing by the amount that the variable is progressing. Currently, the specifications for the MIT static dataflow machine call for auto incrementing by one; however, both auto incrementing by powers of two and auto decrementing have been suggested**.

- If the progressing loop variable is an index to an array reference, then the knowledge that this is a progressing loop variable is useful for a code generator that attempts to turn array references into streams.

2.5.2 Successor Variables

Two variables, *v1, v2*, are successor variables if the value of *v1* in pass *n* of a loop is equal to the value of *v2* in pass *n+1* of the loop (e.g., *v1* = *j, j+c, j+2c, ...* and *v2* = *j-c, j, j+c, ...*). Successor variables are not necessarily loop variables; they are sometimes generated from a progressing loop variable (e.g. *X* is an additive progressing loop variable with increment two, *I*=*X*+1, and *J*=*X*-1). *I* and *J* are, therefore, successor variables. If *I* and *J* are used as indexes to array *A*, then *A[I]* in cycle *n* of the loop is equal to *A[J]* in cycle *n+1* of the loop. If we save the value of *A[I]* from one loop cycle to the next, then we only have to perform one array access in all but the first loop cycle.

The VAL optimizer performs this transformation by making the output of *A[I]* a loop variable. However, since this type of loop variable must be treated specially, the optimizer assigns it a unique case (case=2). Because successor loop variables have a special case, the loop unfolding optimization must know how to deal with these variables correctly.

**This past summer, with the help of Dr. William B. Ackerman, I wrote a low-level dataflow program that solved a tridiagonal matrix using the cyclic reduction algorithm. I wrote the program twice, once assuming auto incrementing of one, and once assuming auto incrementing by any power of two between 0 to 32. The addition of auto incrementing by powers of two allowed approximately 33% of the *cells* (low-level dataflow code) from the inner loop to be moved to the outer loop, with the addition of a relatively small overhead. With a matrix of size 127, this meant the execution of 30% fewer cells. Since the total number of cells executed in either case is $n*out+(2^n-1)*in$ (where *out* is the number of cells in the outer loop, *in* is the number of cells in the inner loop, and 2^n-1 is the size of the matrix), this efficiency increases with larger matrices.

Current work on the successor variable optimization, for the VAL optimizer, is incomplete. It is, therefore, not possible at this point to determine the correct way for the loop unfolding optimization to handle successor variables. In my implementation, I will not the allow the unfolding of any loop that has successor loop variables.

2.6 Graph Conventions used in Thesis

I use two models to display dataflow graphs in this thesis. When a lot of detail is necessary, I use the Intermediate Graph Format described in Section 2.4. When less detail is required, I use the *Simple Model.* Figure 2-7 demonstrates the differences between these two models. The simple model suppresses information about exact gates and links in a graph, and is used primarily when the relative positions of different components of a graph are important.

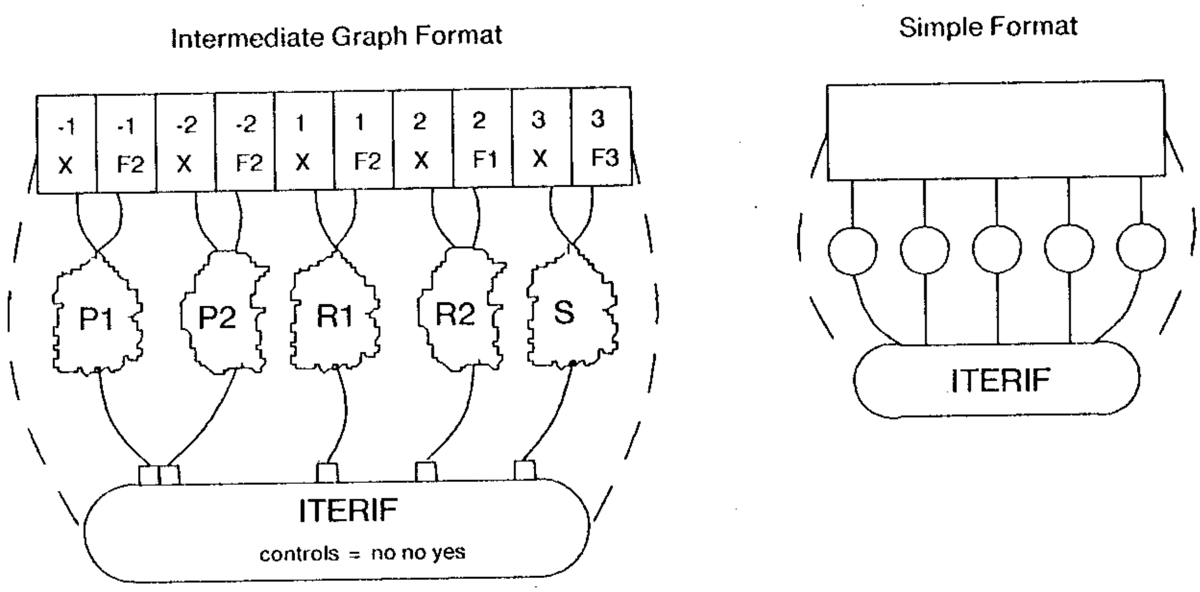

Figure 2-7:IGF, SF - Comparison of Intermediate graph format and Simple Format

Chapter Three

Loop Unfolding

Loop unfolding is the process where a compiler generates one or more copies of a loop and modifies the framework of the loop. The *unfolding factor* is the number of copies being created. The intent is that if a loop must execute 10 times, it will only have to execute five times if the loop is copied once. In Chapter 4, I will discuss an implementation of loop unfolding for the VAL compiler described in Section 2.3; here, I will be concerned with a general description of loop unfolding and a discussion of its benefits. The material in this chapter is discussed in greater detail in [2].

3.1 Different Types of Loop Unfolding

I will discuss four cases of loop unfolding:

1. Loops without a known number of cycles
2. Loops with a known number of cycles
3. Initial Unfolding
4. Final Unfolding

Examples of these four types of unfolding will be illustrated with VAL.

3.1.1 Loops Without a Known Number of Cycles

If we have no knowledge about how many times a loop will cycle, then we must execute the exit test with each copy of the loop. Figure 3-1 shows the transformation of such a loop unfolded once.

```
for X := I do
  if P(X) then R(X)
  else
    let NEW: int := S(X) in
      if P(NEW) then R(NEW)
      else
        iter X := S(NEW) enditer
      endif
    endlet
  endif
endfor
```

Figure 3-1: VAL - Unfolding Factor = 1, Unknown Number of Cycles

3.1.2 Loops With a Known Number of Cycles

If we know that the number of times a loop will cycle is a multiple of *n*, then we can unfold the loop up to *n-1* times, while only having to execute the exit test once. Figure 3-2 shows the transformation of such a loop.

```
for X := I do
   if P(X) then R(X)
   else
      iter X := S(S(X)) enditer
   endif
endfor
```

Figure 3-2: VAL - Unfolding Factor = 1, Known Number of Cycles

3.1.3 Initial Unfolding

Initial unfolding consists of copying the iterating function *n* times and first executing these *n* cycles before entering the loop. Figure 3-3 shows the effect of an initial unfolding of three.

```
for X := S(S(S(I))) do
   if P(X) then R(X)
   else
      iter X := S(X) enditer
   endif
endfor
```

Figure 3-3: VAL - Initial Unfolding Factor = 3

We would want to perform an initial unfolding of *n* on a loop which was going to execute at least *n* times, or when the number of times the loop was going to cycle was equal to *n* mod *m*. In the latter case, we would also perform a regular loop unfolding of *m*, where the unfolding is of the form where we know that the number of cycles will be a multiple of *m*.

3.1.4 Final Unfolding

Final unfolding is similar to initial unfolding, except that the copies execute after the loop terminates. The reasons for a final unfolding are analogous to the reasons for an initial unfolding.

3.2 Benefits of Loop Unfolding

The primary reason for performing loop unfolding is so that the copies of a loop can execute simultaneously. Thus loop unfolding is beneficial for a parallel computer. I described, in the discussion of progressing loop variables, how to achieve maximum concurrency capabilities (see Section 2.5.1). It is believed that many loops have the complete data independence between loop cycles, necessary to achieve this maximum concurrency. All loops generated by the VAL **forall** structure [1] have this feature; all **forall** loops can, therefore, be optimized by the progressing loop variable optimization. One general use of the **forall** structure is to apply some function to every element of an array. However, even when there is not complete data independence between cycles of a loop, we can still achieve a large degree of concurrency. The optimizer determines the minimum *critical path* of data dependencies between loop cycles. Other benefits of loop unfolding relate to areas discussed in Chapter 5.

Chapter Four

Implementation

VAL contains two loop constructs, **for** and **forall** [1]; however, **foralls** are converted to **fors** via other optimizations, so I will only discuss **fors** in this chapter. To simplify the discussion of implementation, I will divide this chapter into two sections:

1. Base Model

2. Extensions.

In actual use, some of the items discussed under *Extensions* are as important as the items discussed under *Base Model.* However, for purposes of presentation, it is expedient to present the implementation this way.

4.1 Base Model

In this section, I will describe how to perform a single unfolding on a simple loop. A simple loop, such as the one shown in Figure 4-1, contains an **iterif** body which has three main units:

1. n predicate clauses, or bodies

2. n return clauses, or bodies

3. one iterating clause, or body

The corresponding VAL program is shown in Figure 4-2. I refer to the **iterif** that existed before the unfolding as the *original* **iterif**, and the **iterif** that was created when the loop was unfolded as the *new* **iterif.**

There are two main parts to unfolding a simple loop:

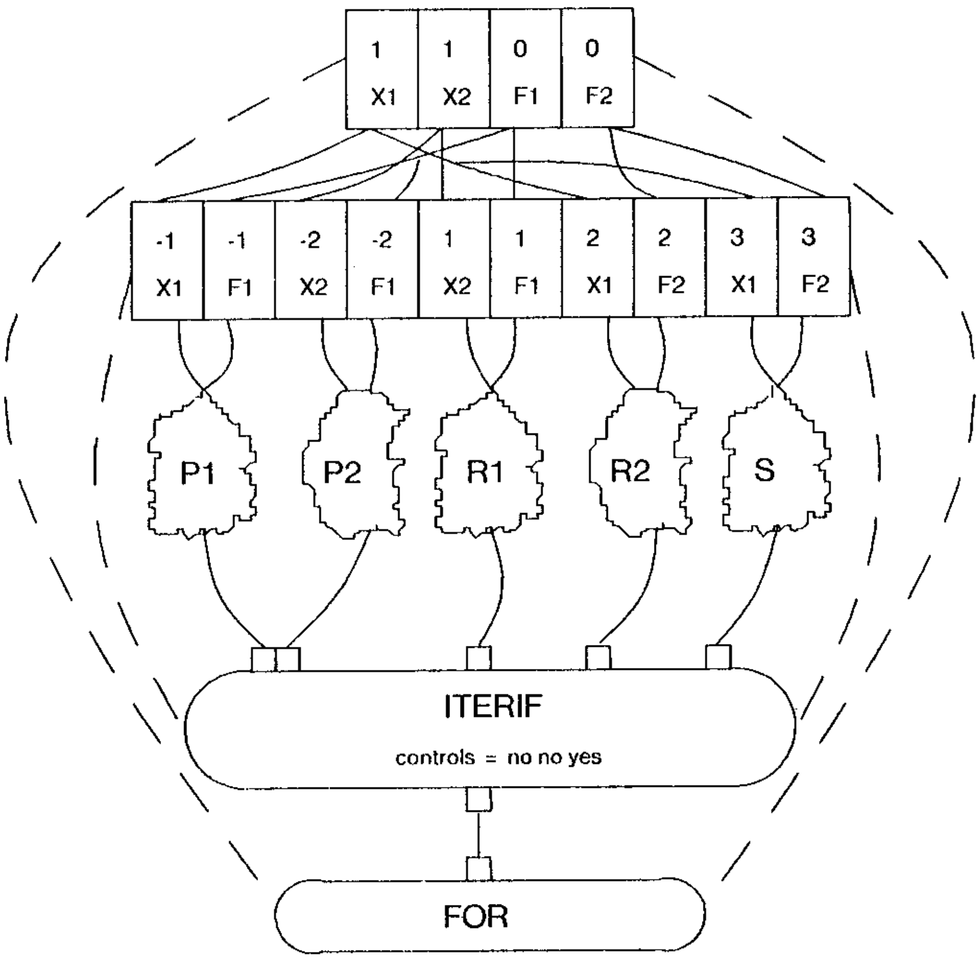

Figure 4-1:IGF - Simple Loop

1. reproducing or copying the **iterif** body

2. splicing in the copied **iterif** body.

```
for X1, X2 := I1, I2 do
   if P1(X1, F1) then R1(X2, F1)
   elseif P2(X2, F1) then R2(X1, F2)
   else iter X := S(X1, F2) enditer
   endif
endfor
```

Figure 4-2: VAL - Simple Loop

4.1.1 Reproducing an Iterif Body

A typical **iterif** body is shown in Figure 4-1. For reproduction purposes, it does not contain the inputs to the gates (however, it contains the outputs from the gates) or the one output from the node.

No links may cross dashed lines in a graph. The advantage of this rule becomes very relevant here. Because of this restriction, we can *blindly* copy all the nodes residing between the outputs of the gates and the inputs to the **iterif** node, without worrying about copying extra material. Otherwise, we would need some knowledge of the meaning to the subgraph we were copying. Besides being more difficult to implement, this latter process would also takes much longer to execute.

We use three data structures in the copying process:

1. A node queue: The need for a node queue will be shown in Section 4.1.1.2. Enqueueing and dequeuing have their usual meanings.

2. A node copy pointer: Each node in the original subgraph has a pointer to the node that is its copy. *Node-copying* refers to both creating a copy of a node and placing a pointer to the new node in the copy pointer of the original node.

3. A node network: A network containing pointers to all nodes that have been copied is needed to clear, at the end of the copying process, the copy pointers described above.

The copying process can be split chronologically into two parts:

1. initial links copying

2. main section copying.

All links are bi-directional and are implemented as two separate links in the software. For clarity, I refer to the node that the copying routine is acting upon as the *from-node*, and the node at the other end of the link as the *to-node*.

4.1.1.1 Initial Links Copying

For each link in the sets of gate outputs and **iterif** node inputs, the copying process executes the following:

1. If the to-node does not have its copy pointer set, then the process node-copies and enqueues the to-node.

2. A similar link is created between the new **iterif** node and the copy of the to-node.

Figure 4-3 shows a typical section of a loop that would be copied during one call to the copying process. The highlighted links are the initial links.

4.1.1.2 Main Section Copying

At this point in the copying routine, there are some nodes in the local queue. Until the queue is empty, the routine recursively dequeues a node (i.e., the current from-node) from the queue. For each of the from-node's links, the process executes the following actions:

1. If the to-node does not have its copy set, then the process node-copies and queues the to-node.

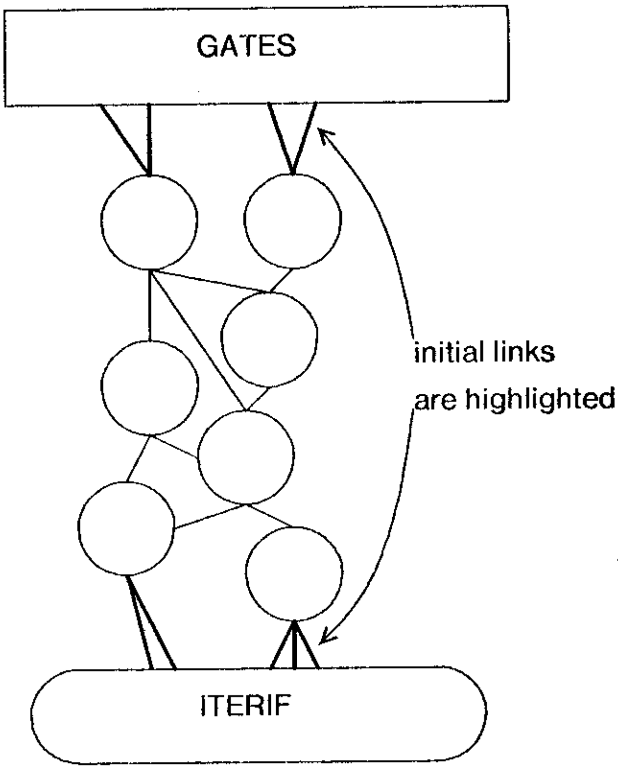

Figure 4-3:SF - Initial Links

2. If the to-node has not been netted, then a similar link is created between the copies of the from-node and the to-node.

Figure 4-4 shows the same subgraph as shown in Figure 4-3, except now the main links are highlighted. The process then places the from-node in the node network. Thus, if a to-node does not have its copy set, it means that this is the first time the to-node has been encountered. If a to-node has its copy set but is not netted, it means that the node has been encountered but has not yet been a from-node. Therefore the to-node does not necessarily have all its links copied. Finally, if a to-node is netted, it means that it has been a from-node and, therefore, all its links

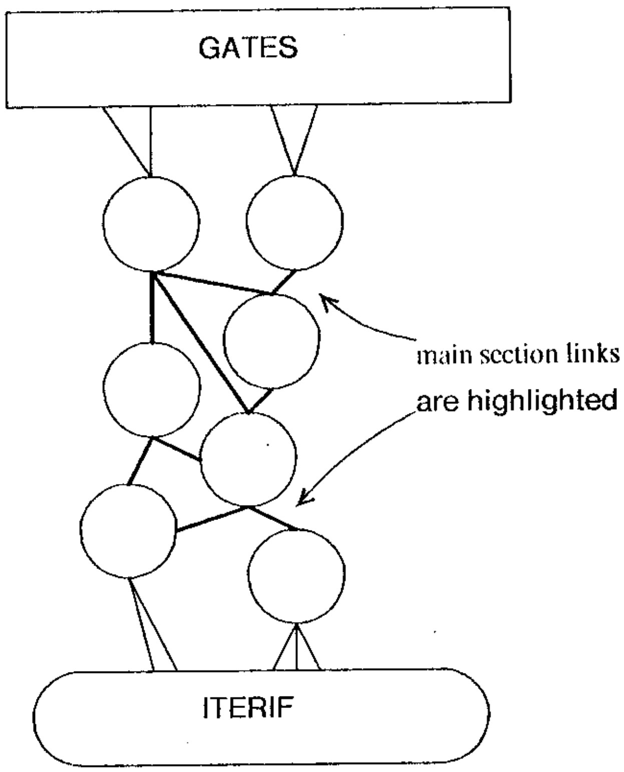

Figure 4-4:SF - Main Links

have been copied.

4.1.2 Splicing

The copied **iterif** body must be spliced into the main graph at two locations:

1. the one output of the new **iterif** node

2. the gate inputs of the new **iterif.**

The first operation is relatively simple. As shown in Figure 4-5, the inputs, from the iterating body, that fed the iterating arm of the original **iterif** node are replaced by

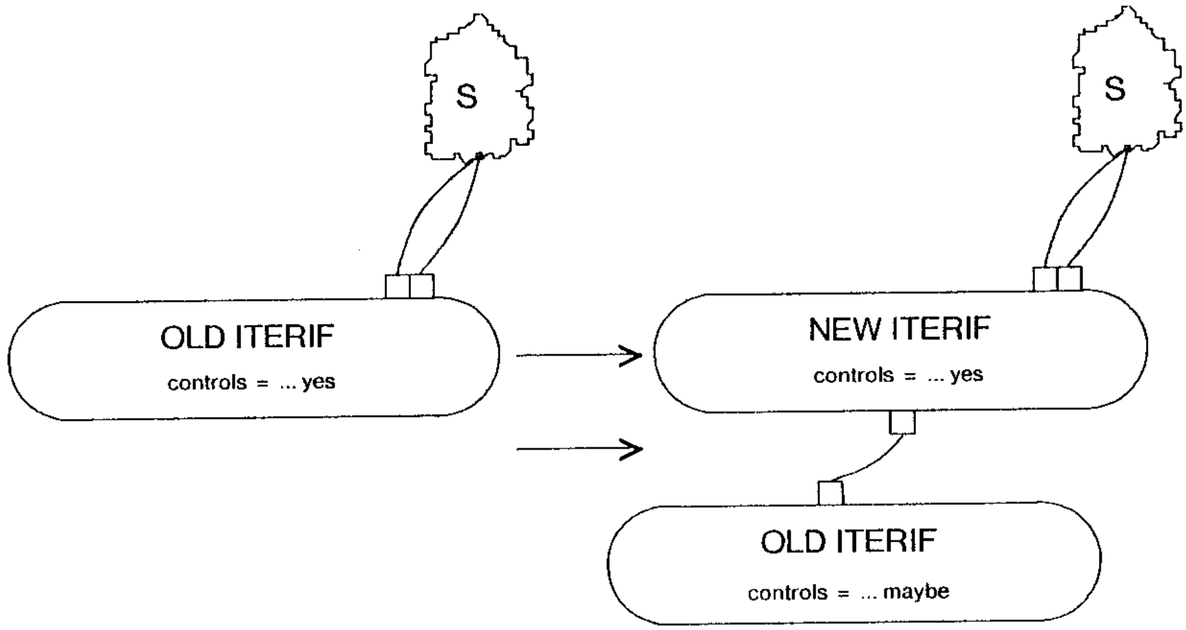

Figure 4-5:IGF - Splicing in Iterif Output

the one output from the new **iterif** node. We must change the control corresponding to this arm of the original **iterif** node from yes to maybe.

The second operation involves one of two actions, depending on the case of the **for** gate that is the input to the particular **iterif** gate. The case of the **for** gate can be either:

1. *1*, representing a loop variable

2. *0*, representing a free variable.

For each gate input to the original **iterif**, we perform one of the two actions.

4.1.2.1 Loop Variable

If the input to the **iterif** gate comes from a loop variable, then we want the input to the equivalent gate on the new **iterif** to be what the value of the loop variable would have been in the next cycle. More specifically, if the input to the **iterif** gate comes from the n^{th} loop variable, then the equivalent input on the new **iterif** comes from whatever currently feeds the n^{th} arg of the iterating arm of the original **iterif** node (see Figure 4-6).

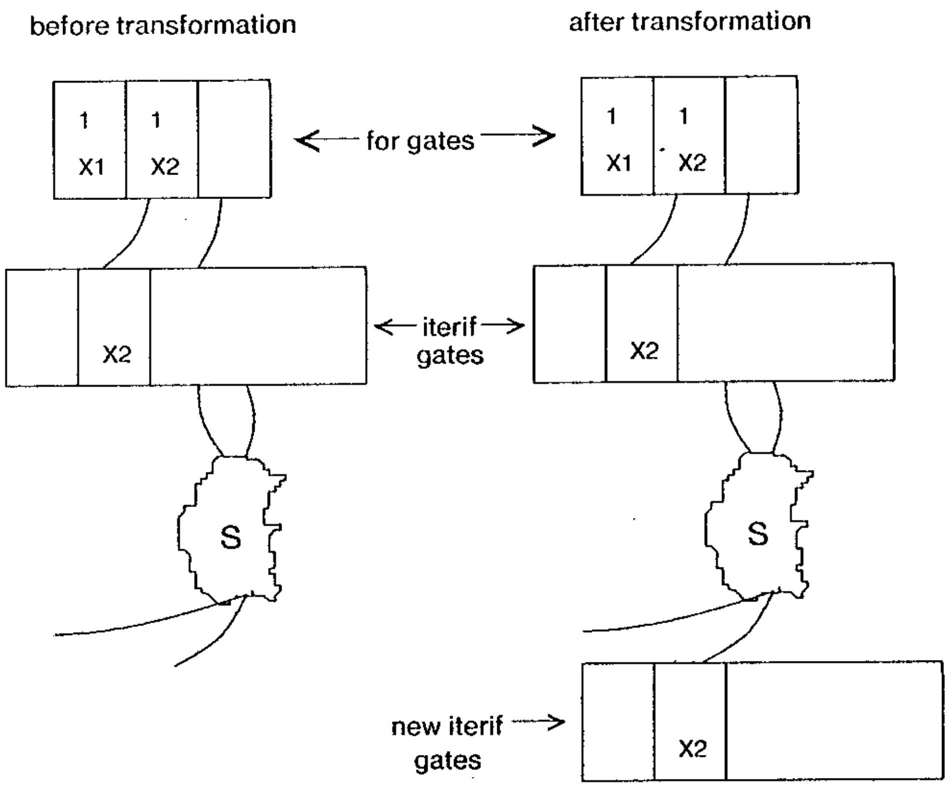

Figure 4-6:IGF - Splicing in Loop Variable

The case of the new gate is the same as the case of the equivalent gate on the old

iterif.

4.1.2.2 Free Variable

If the input to the **iterif** gate comes from a free variable, then we want to build a link between the free variable and the equivalent gate on the new **iterif.** However, we cannot build this link directly. Since the new **iterif** is contained within the body of the old **iterif**, building a direct link from a **for** gate to a new **iterif** gate would require passing through the dashed lines of the original **iterif.** This, as noted before, is illegal. Therefore, we must first build a new gate on the original **iterif.** The case of this gate is the case of the iterating arm of the **iterif.** We then build a link between the free variable of the **for** and the new gate. Now we can build the new gate on the new **iterif.** The case of this latter gate is the same as the case of the equivalent gate on the old **iterif**, and the input to this gate comes from the new gate that we built on the old **iterif.**

We cannot simply make the input to the gate on the new **iterif** come from the output of the equivalent gate on the old **iterif**, since in general, the case of the gate on the old **iterif**, will not be the same as the case of the iterating arm of the old **iterif.** We do not have to worry about the correct cycle of the loop as we did with loop variables, because a free variable remains constant throughout the execution of a loop. When creating new gates, it is very likely that some of the gates on the same **iterif** are similar, i.e., they have the same input and case. Another optimization combines similar gates into one gate, after the unfolding has been completed. Figure 4-7 shows the transformation involving a free variable. Figure 4-8 shows the overall unfolding of the loop in Figure 4-1.

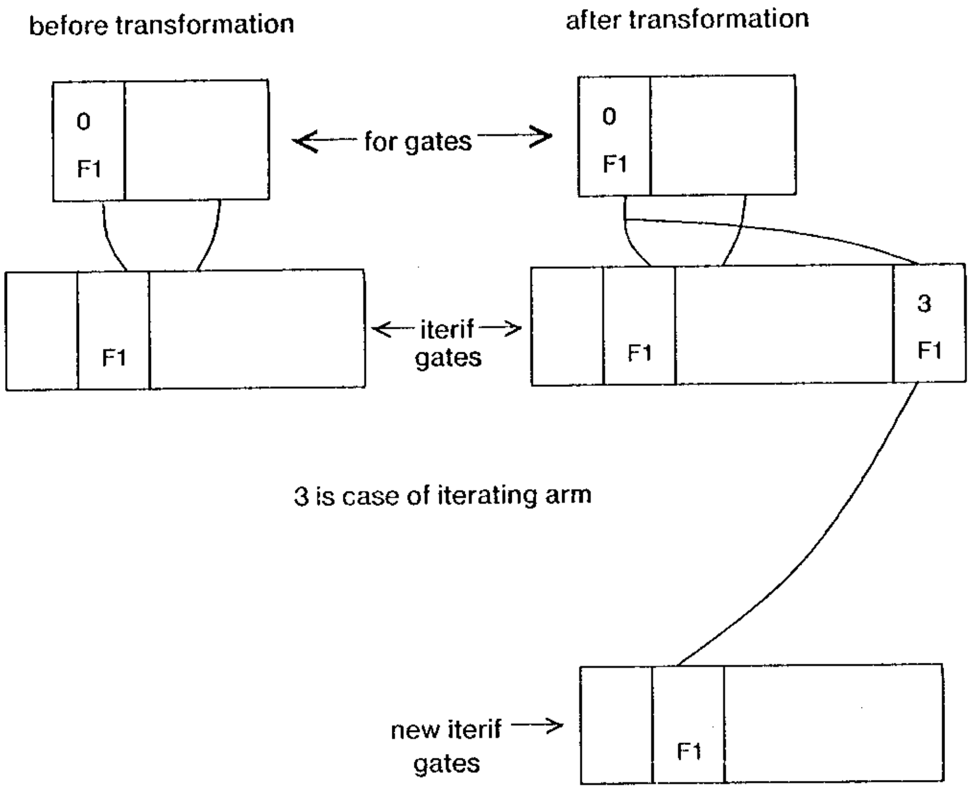

Figure 4-7: IGF - Splicing in Free Variable

4.2 Extensions

We must add several extensions to the basic model, in order for it to be able to unfold any loop. The extensions are:

- multiple unfoldings
- dealing with progressing loop variables
- loops with nested **iterifs**

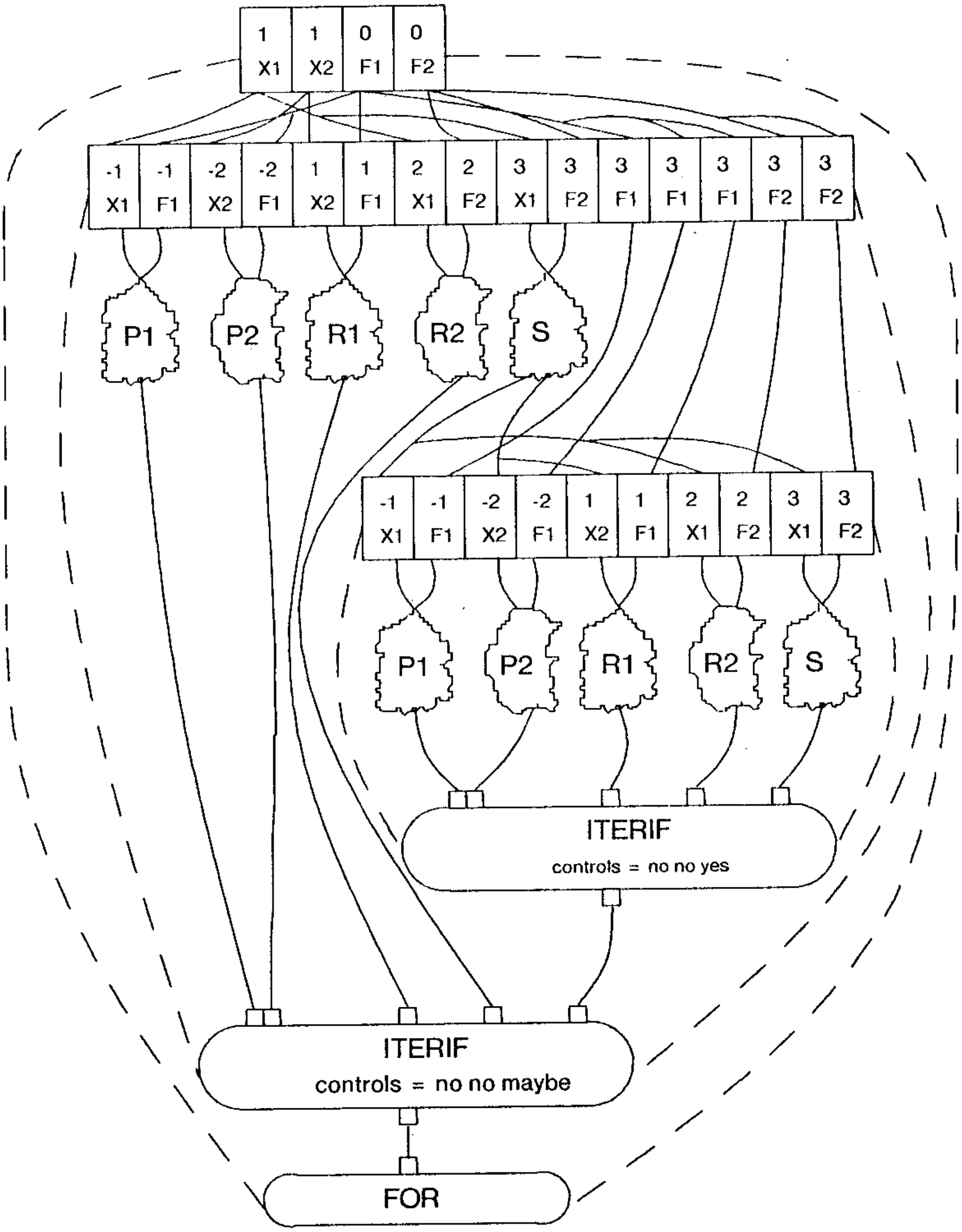

Figure 4-8: IGF - Unfolding, Basic Loop

- loops with a **let** statement between the **for** and **if** statements
- loops with multiple yes arms
- nested loops.

4.2.1 Multiple Unfoldings

In general we want to unfold a loop more than once. Adding this capability to the base model consists of performing the base operations on an arbitrary number of copies in parallel. There are four places in the base model where it necessary to discuss the effects of this extension:

1. reproducing the **iterif** body
2. **iterif** node output splicing
3. loop variables
4. free variables.

In the following discussion, I use *NEW-ITERIFS* to refer to all the copies of the original **iterif**; that is, every **iterif** except for the original one. I use *OLD-ITERIFS* to refer to every **iterif**, including the original one, but not including the last copy. The variable n refers to the number of times we are unfolding the loop.

4.2.1.1 Reproducing the Iterif body

The algorithm for reproducing the body of an **iterif** is basically the same as for a single unfolding. A node's copy-pointer now points to n nodes: one node for each copy of the original **iterif** being made. Wherever we built a link in the basic model, we now build n equivalent links: one for each copy of the **iterif** body.

4.2.1.2 Iterif Output Splicing

We perform the base model operations, described in Section 4.1.2, n times. The new link is created between the n^{th} node of *OLD-ITERIFS* and the n^{th} node of *NEW-ITERIFS* (see Figure 4-9).

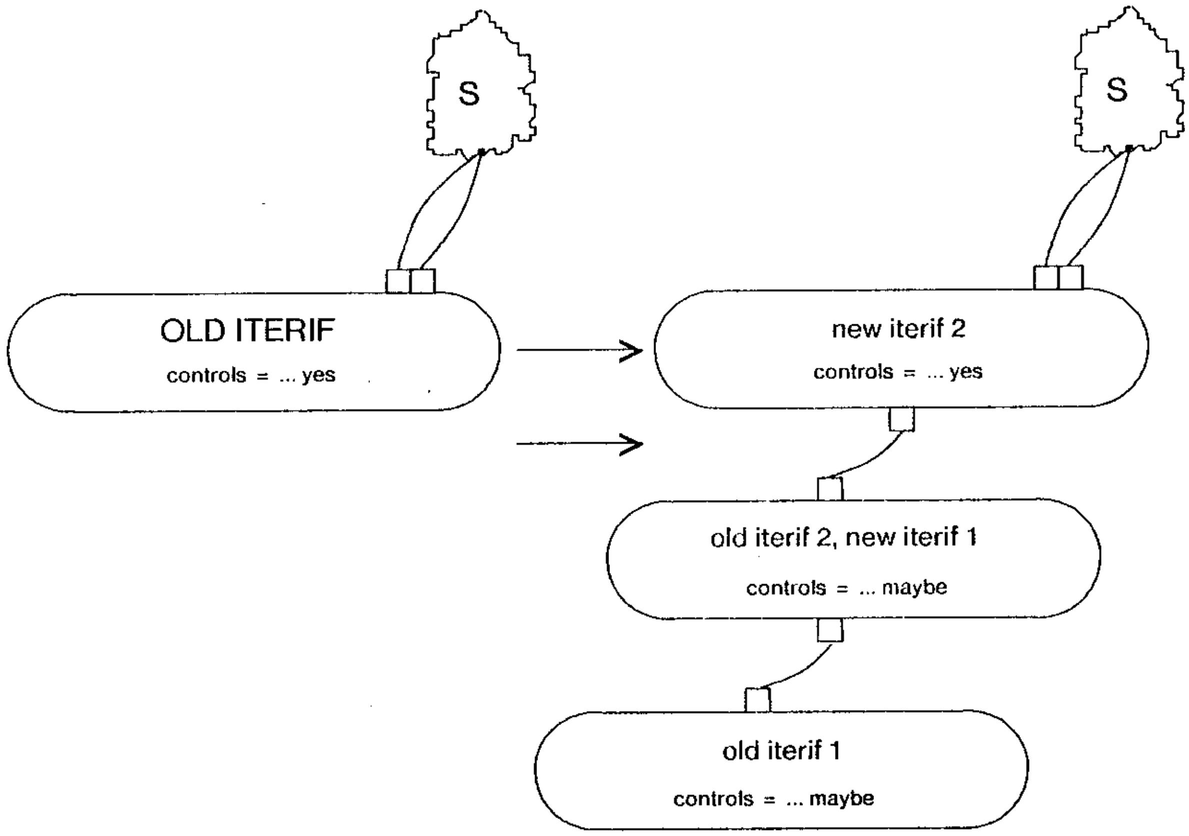

Figure 4-9: IGF - Multiple Unfolding, Splicing in Iterif Output

4.2.1.3 Loop Variables

As with output splicing, each time we encounter a loop variable, we perform the base model operations, described in Section 4.1.2.1, n times. The new link is built between the n^{th} **iterif** body of *OLD-ITERIFS* and the n^{th} element of *NEW-ITERIFS* (see Figure 4-10).

4.2.1.4 Free Variables

Again, we perform the base model operations, described in Section 4.1.2.2, n times, whenever we encounter a free variable. In general, the new links are built between the $n\text{-}1^{th}$ element of *OLD-ITERIFS*, the n^{th} element of *OLD-ITERIFS*, and the n^{th} element of *NEW-ITERIFS*. However, when $n = 1$, the source of the free variable is the **for** (see Figure 4-11). Figure 4-12 shows the simple format unfolding, with a factor of two, of the loop shown in Figure 4-1.

4.2.2 Loops with Progressing Loop Variables

The transformation performed on a loop by the *Progressing Loop Variable* optimization and its importance in interacting with loop unfolding were discussed in Section 2.5.1. When unfolding a loop and splicing in the copied subgraphs at the **iterif** gates, we perform the following two actions whenever we encounter a gate whose input is from a progressing loop variable:

1. initializing Δ, i.e. the amount that the progressing loop variable is incremented by each cycle

2. updating X, i.e. the name by which we will refer to the progressing loop variable.

Unless stated otherwise, we assume that we are dealing with an additive progressing loop variable.

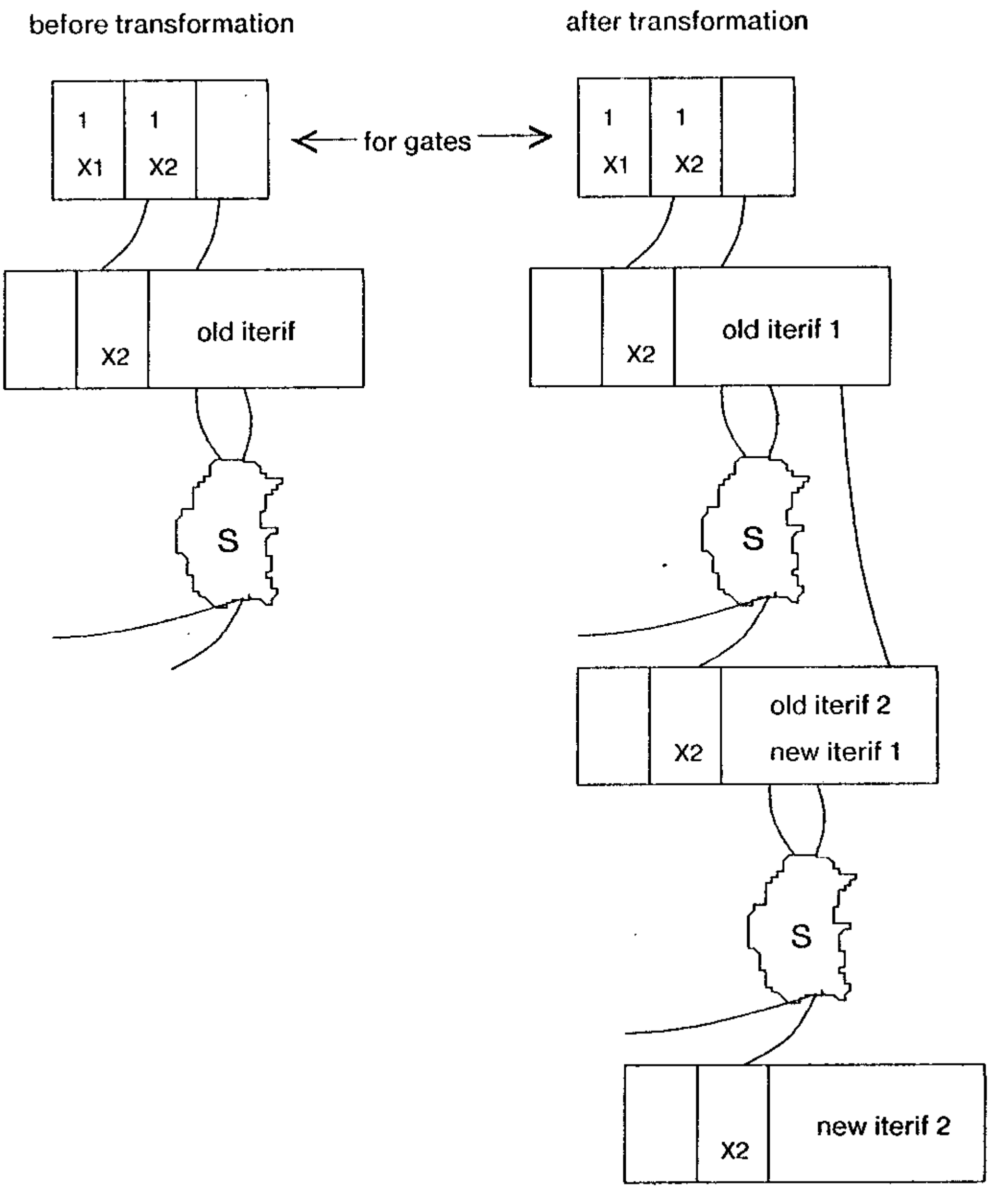

Figure 4-10:IGF - Multiple Unfolding, Splicing in Loop Variable

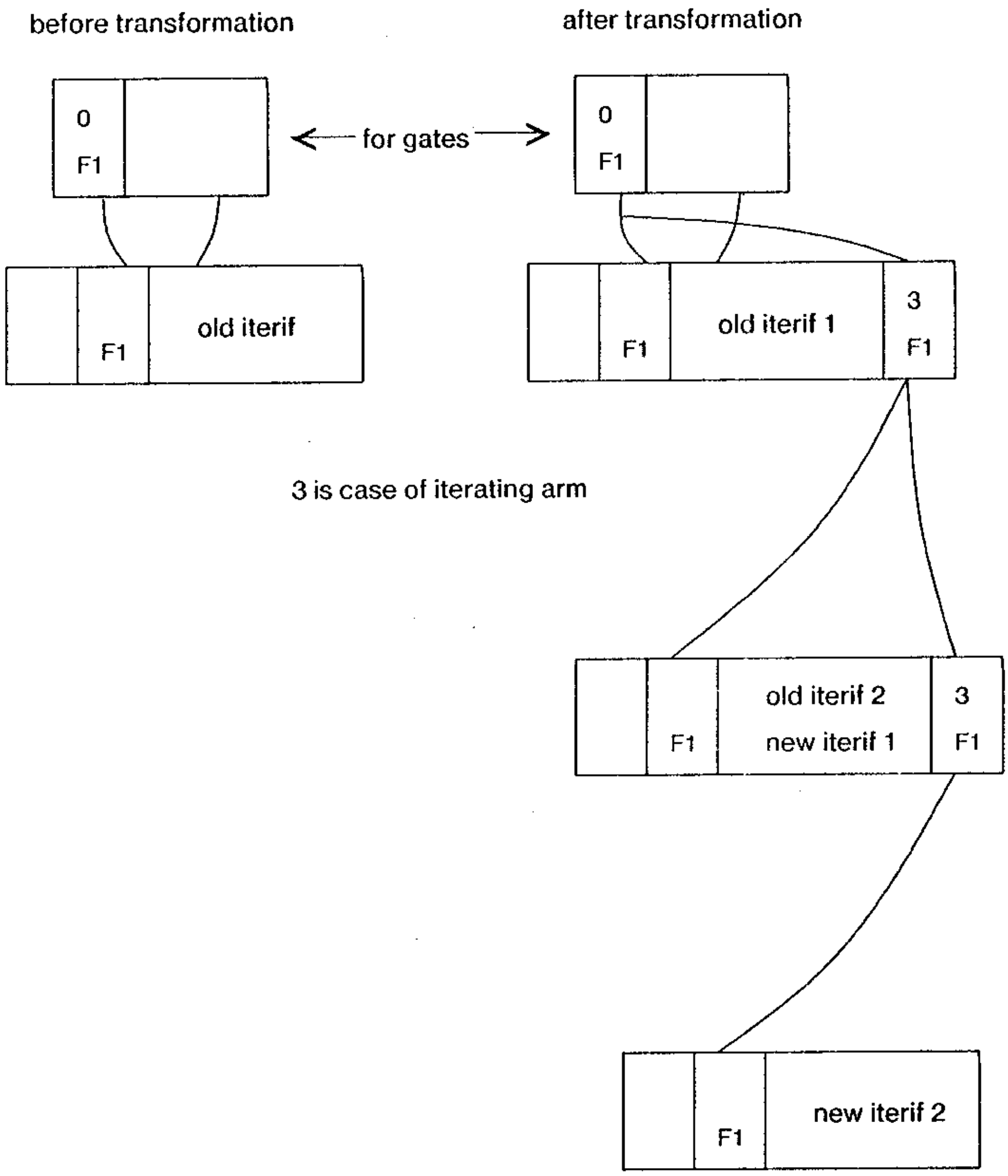

Figure 4-11:IGF - Multiple Unfolding, Splicing in Free Variable

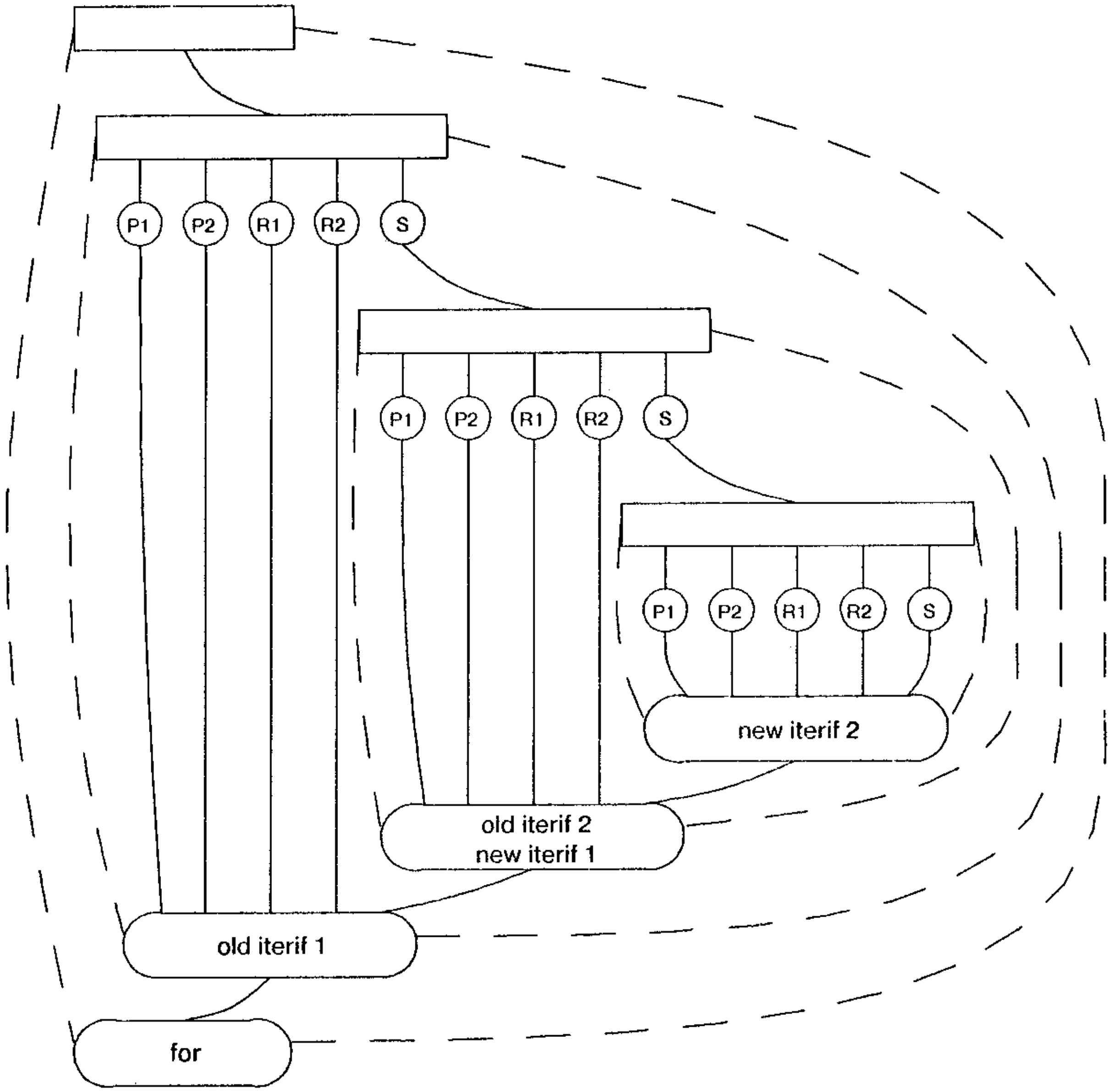

Figure 4-12:SF - Multiple Unfolding, Factor = 2

4.2.2.1 Initializing Delta

As noted above, Δ is the amount by which we would normally increment our X each loop cycle. However, if we unfold the loop n times, we must increment X by $\Delta * (n + 1)$ (where * is multiplication). Thus we add a **multiplication** node to the graph. Its inputs are Δ and $n+1$ (the latter is a constant at compile time), and its output, the new Δ, feeds into the **for** gate that the old Δ used to feed into (see Figure 4-13).

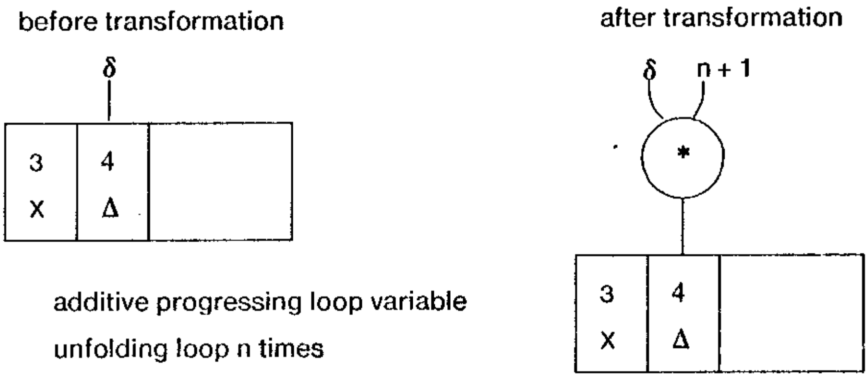

Figure 4-13:IGF - Initializing Δ

4.2.2.2 Updating *X*

When updating X, we perform the following operations n times.

- create two new gates on the n^{th} element in *OLD-ITERIFS*, whose respective inputs are the *current* Δ and the *current* X.

  We determine the current Δ and X as follows:

  - If $n=1$, then we create a new gate on the **for** whose input is the original Δ. This will be our current Δ. The **for** gate that corresponding to X is our current X.

  - If $n \neq 1$, then the current Δ is the output of the Δ gate that we created on the n-1^{th} element of *NEW-ITERIFS*, and the current X is the output of the last addition node we created.

- create an **addition** node whose inputs are the outputs of the gates that were just created

- create a new gate on the n^{th} element in *NEW-ITERIFS*. The new gate corresponds to the gate on the original **iterif** where we are currently splicing in the subgraphs.

- feed the output of the **addition** node into the newly created gate (see Figure 4-14).

We only have to initialize Δ and update X the first time we encounter an original **iterif** gate whose input is from a particular progressing loop variable. More than one gate on the original **iterif** node can have its input come from the same gate of the **for**. The first time we encounter the particular progressing loop variable we save pointers to the n **addition** nodes created. When we later encounter this progressing loop variable, we simply make the output of the *ith* **addition** node be the input to the *ith* element of *NEW-ITERIFS*.

4.2.3 Loops with Nested Iterifs

We can unfold a loop with a nested **iterif** structure, where the yes arm resides on one of the inner **iterifs**. As before, we have to place the copied **iterif** body, or bodies, between the iterating body and iterating arm input of the original **iterif**. The **iterif** reproducing process does not require any changes. We still copy all the nodes within the dashed lines of the original **iterif**. Nested **iterifs** within the original **iterif** body do not affect the copying process. The main difference is that we now have an arbitrary number of **iterifs** in each copy of the original **iterif** body. Before, we were

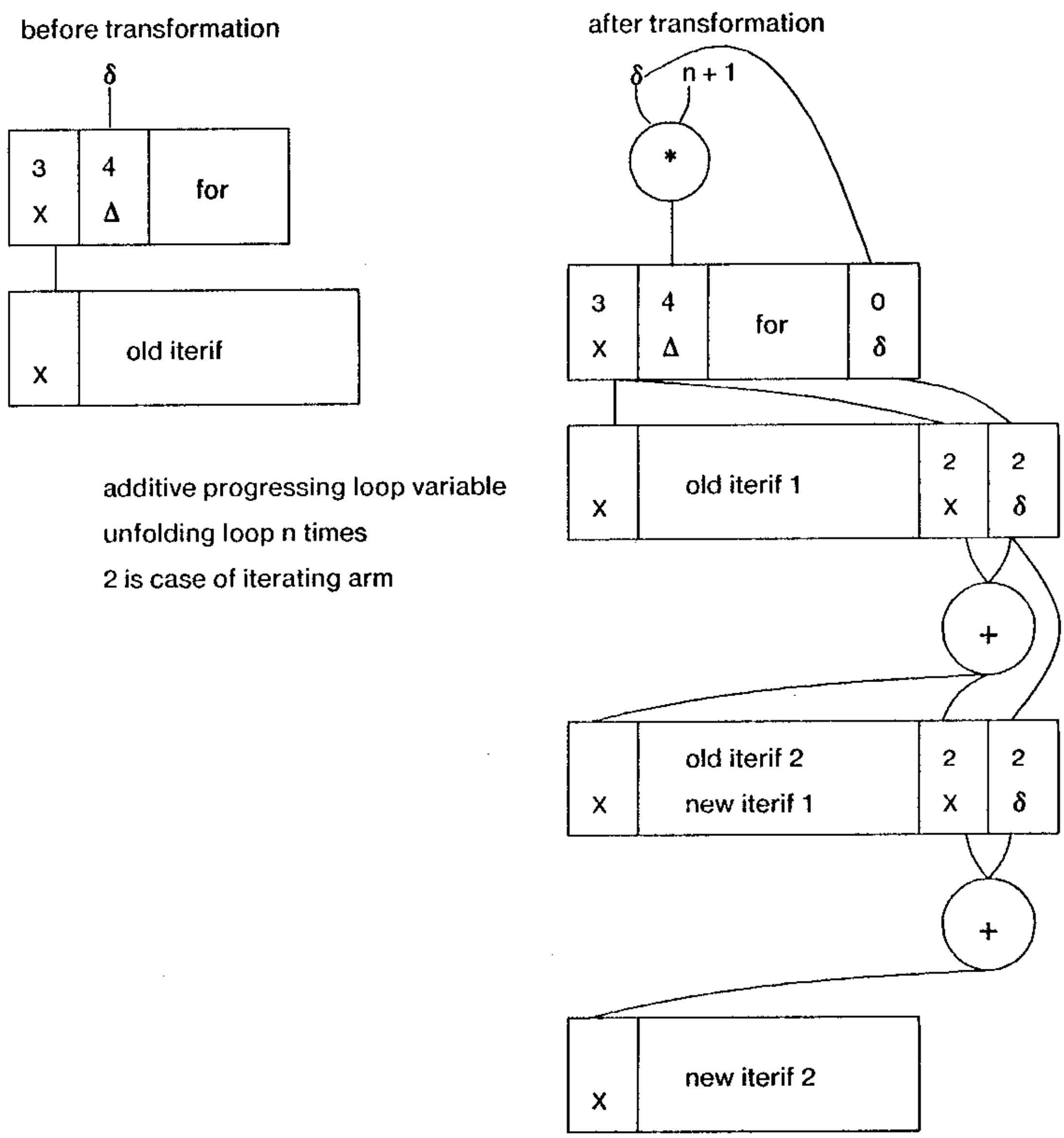

Figure 4-14: IGF - Updating X

only concerned with the one **iterif** in each copy; now we must keep track of the first and last **iterif** in each copy. Thus the procedure for splicing in the copied **iterif** bodies is similar to the our previous method. The main change is that sometimes we are concerned about the first **iterifs** (i.e., the first **iterif** in each copy), and sometimes we are concerned about the last **iterifs** (i.e., the last **iterif** in each copy).

4.2.3.1 Iterif Output Splicing

In the previous model, the output of the $n+1^{th}$ **iterif** node fed into what was originally the iterating arm of the n^{th} **iterif** node. Now, we want the output of the $n+1^{th}$ *first* **iterif** node to feed into what used to be the iterating arm of the n^{th} *last* **iterif** node. The controls that must be changed from yes to maybe are those corresponding to the yes arms of the *last* **iterif** nodes (see Figure 4-15).

4.2.3.2 Loop Variables

In the previous model, we wanted the appropriate input to the n^{th} **iterif** node to be fed to the appropriate gate on the $n+1^{th}$ **iterif**. Now we want the appropriate input to the n^{th} *last* **iterif** node to be fed to the appropriate gate on the $n+1^{th}$ *first* **iterif** (see Figure 4-16).

4.2.3.3 Free Variables

In keeping with the restriction that no links may cross a dashed line, we must feed free variables through the gates of each **iterif** in a nested **iterif** body. In Sections 4.1.2.2 and 4.2.1.4, we discussed how three **iterifs** are necessary to describe the path of the links that feed a free variable into an **iterif** gate. We use the same convention here. In general, we build a link between the n-1^{th} *next-to-last* **iterif** and the n-1^{th} *last* **iterif**. We then feed the variable through the **iterifs** of the next copy to the appropriate gate of the n^{th} *last* **iterif** (see Figure 4-17).

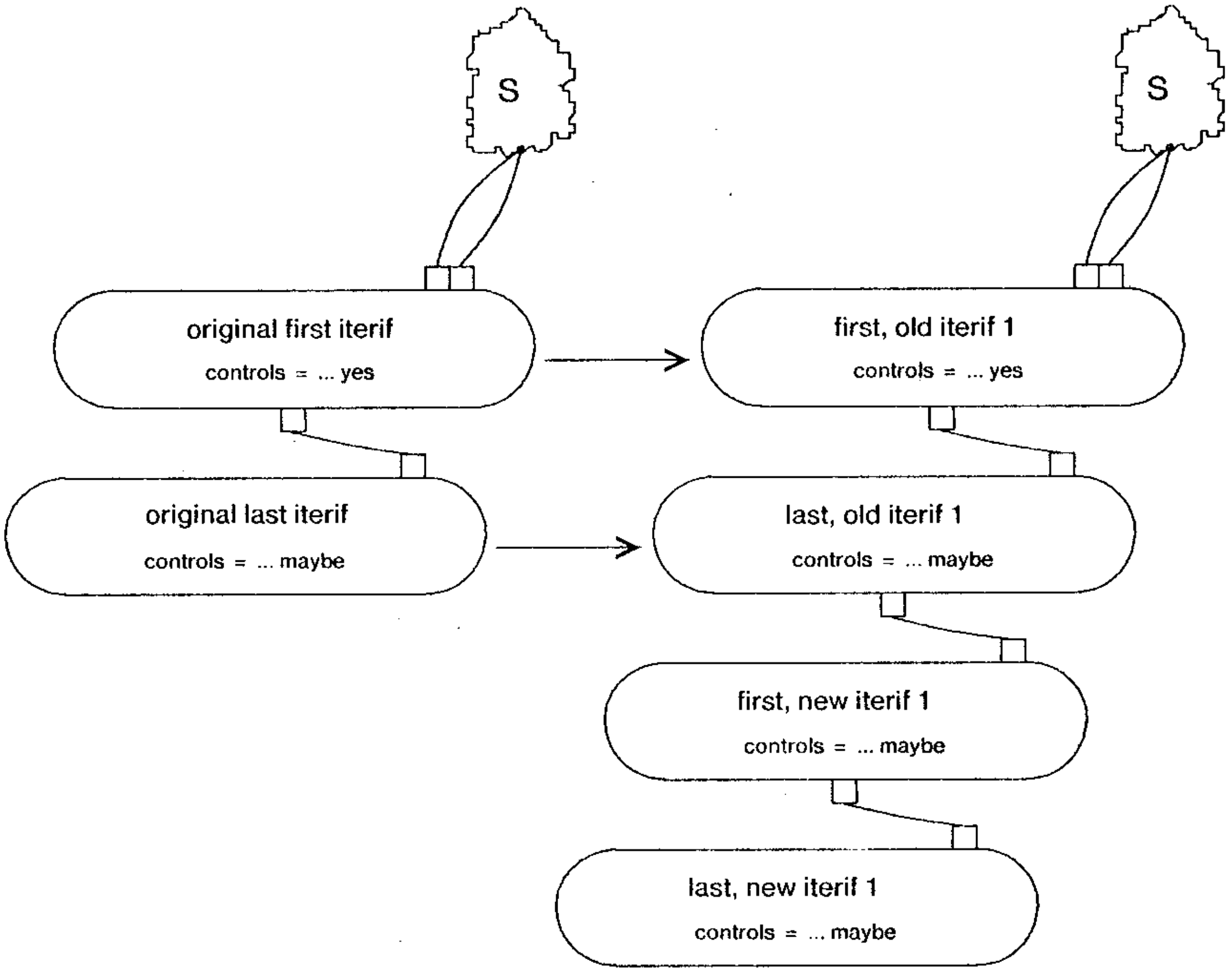

Figure 4-15: IGF - Nested Iterif Unfolding, Iterif Output Splicing

4.2.3.4 Progressing Loop Variables

As with free variables, we now have to feed X and Δ (described in Section 2.5.1) through the gates of all the **iterifs** of a given copy. In general, X comes from the n^{th} **addition** node, and Δ comes from the n^{th} *last* **iterif.** We feed these two values

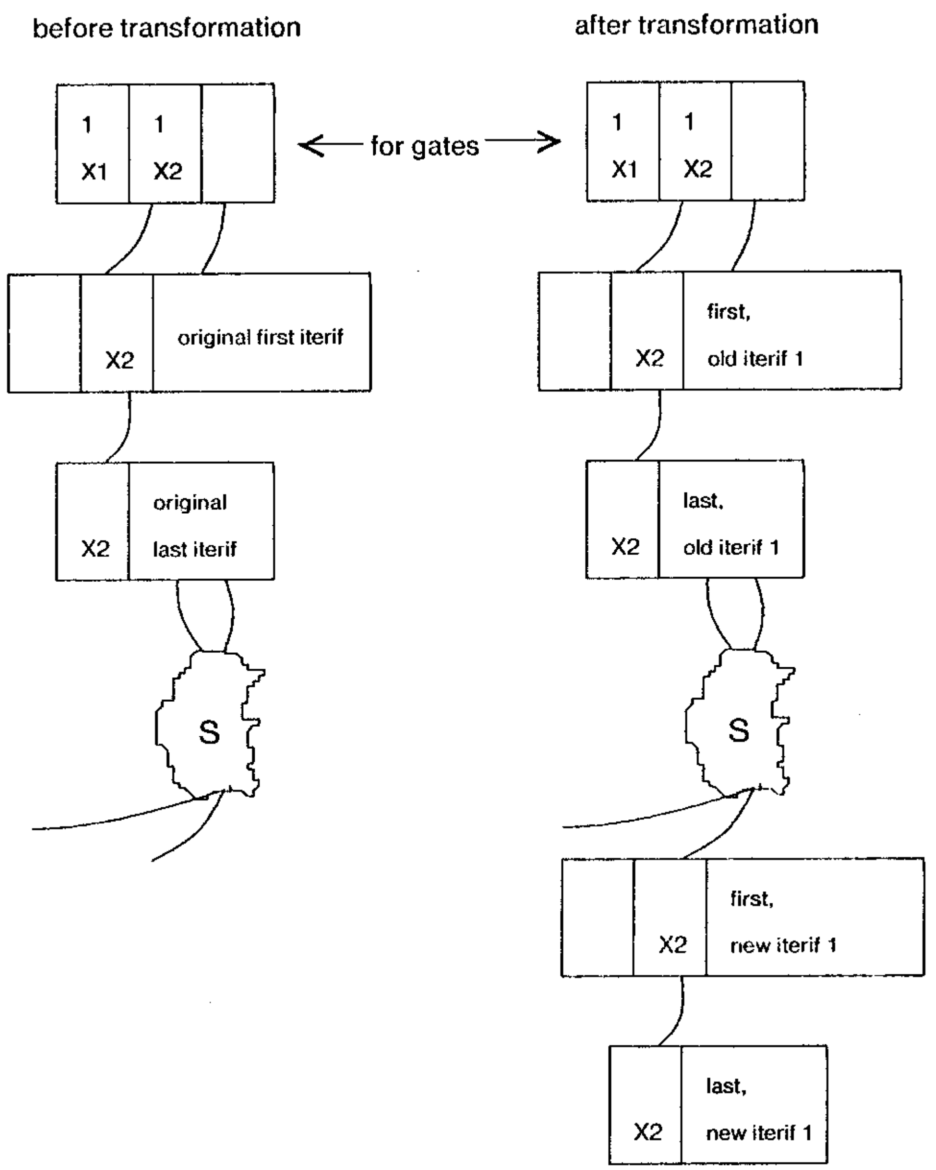

Figure 4-16:IGF - Nested Iterif Unfolding, Splicing in Loop Variables

through the gates of all the **iterifs** of the n^{th} copy. At this point we proceed as in Section 2.5.1 (see Figure 4-18). Figure 4-19 shows the overall unfolding of a nested

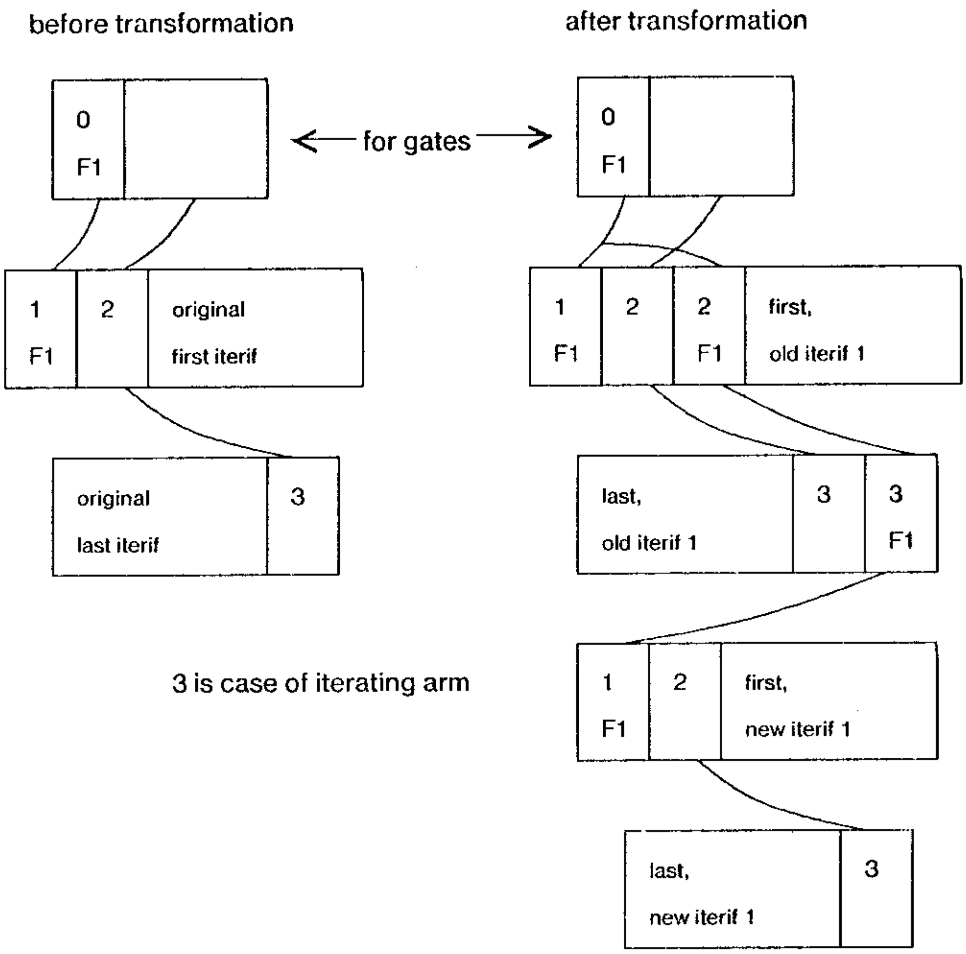

Figure 4-17:IGF - Nested Iterif Unfolding, Splicing in Free Variables

loop.

4.2.4 For Loop with a Let Structure

A **for** loop in VAL that has a **let** between the **for** and **if** statements, such as the VAL fragment in Figure 4-20, translates into the intermediate graph also displayed in

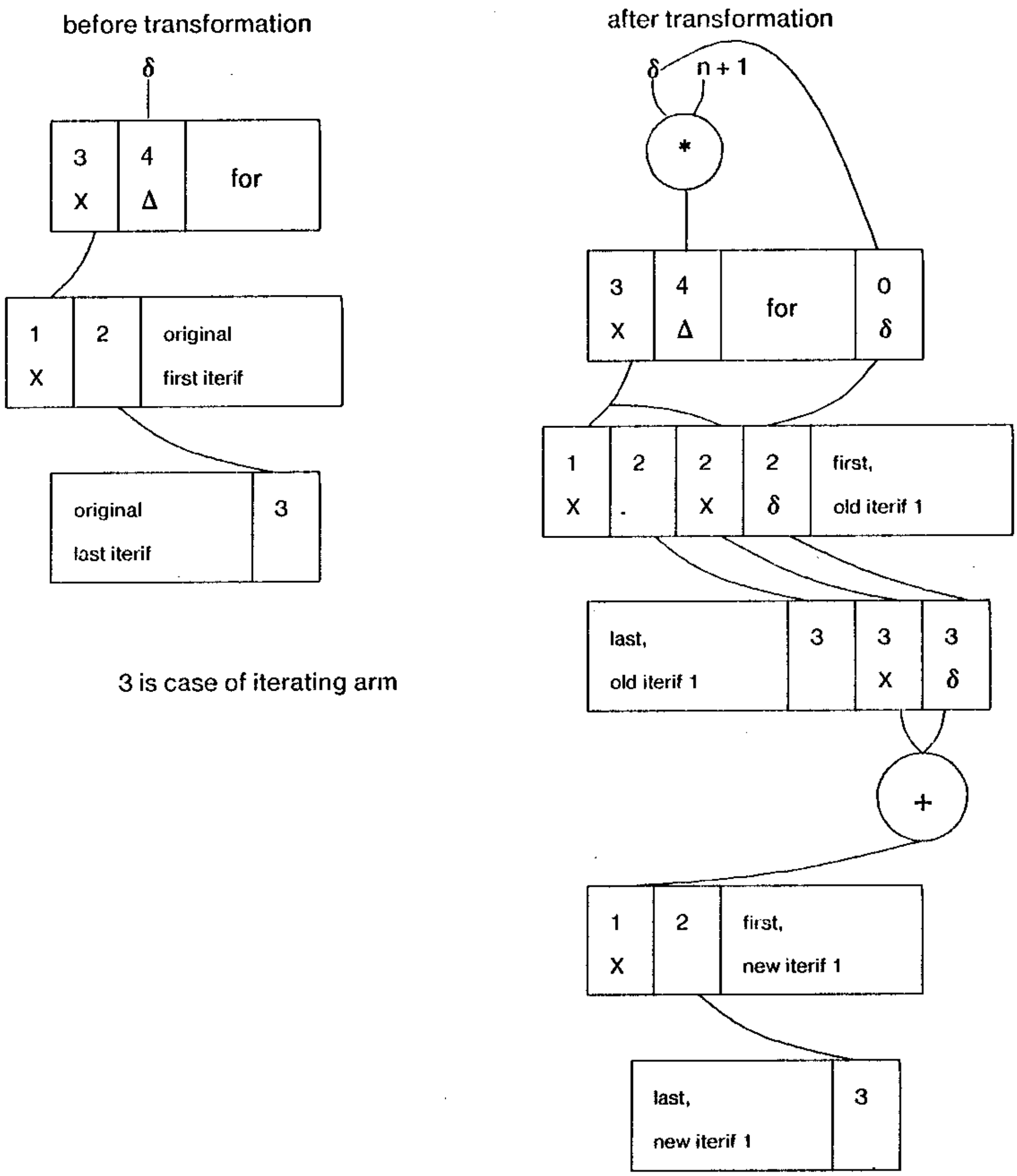

Figure 4-18:IGF - Nested Iterif Unfolding, Splicing in Progressing Loop Variables

Figure 4-20. Such a loop can be unfolded; however, we must take special care in

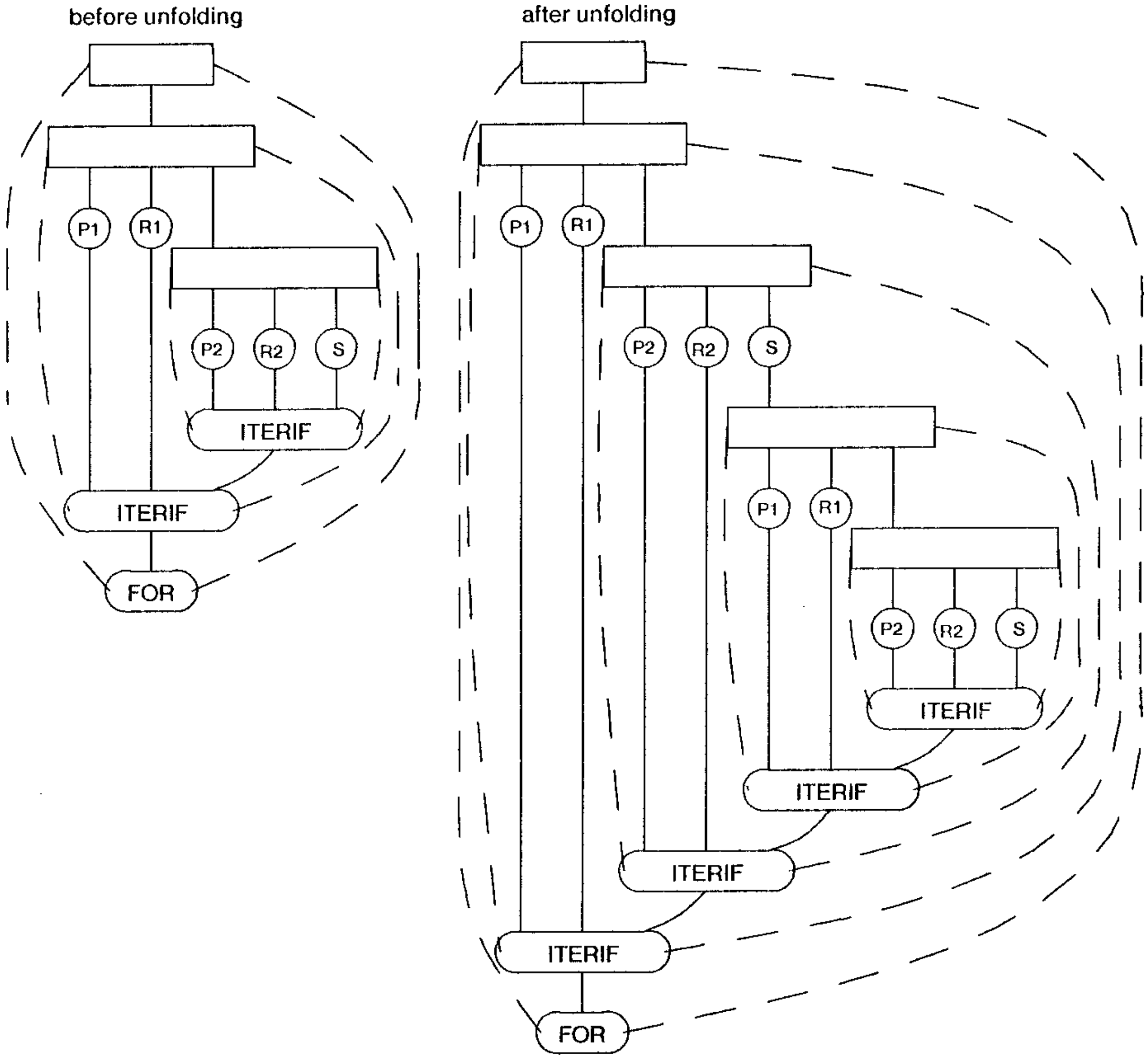

Figure 4-19:SF - Unfolding, Nested Loop

dealing with the **let**, since it occurs at a critical location in the loop. We must

```
for X := I do
   let F1 := LET1(X, F1) in
      if P then R
      else iter X := S enditer
      endif
   endlet
endfor
```

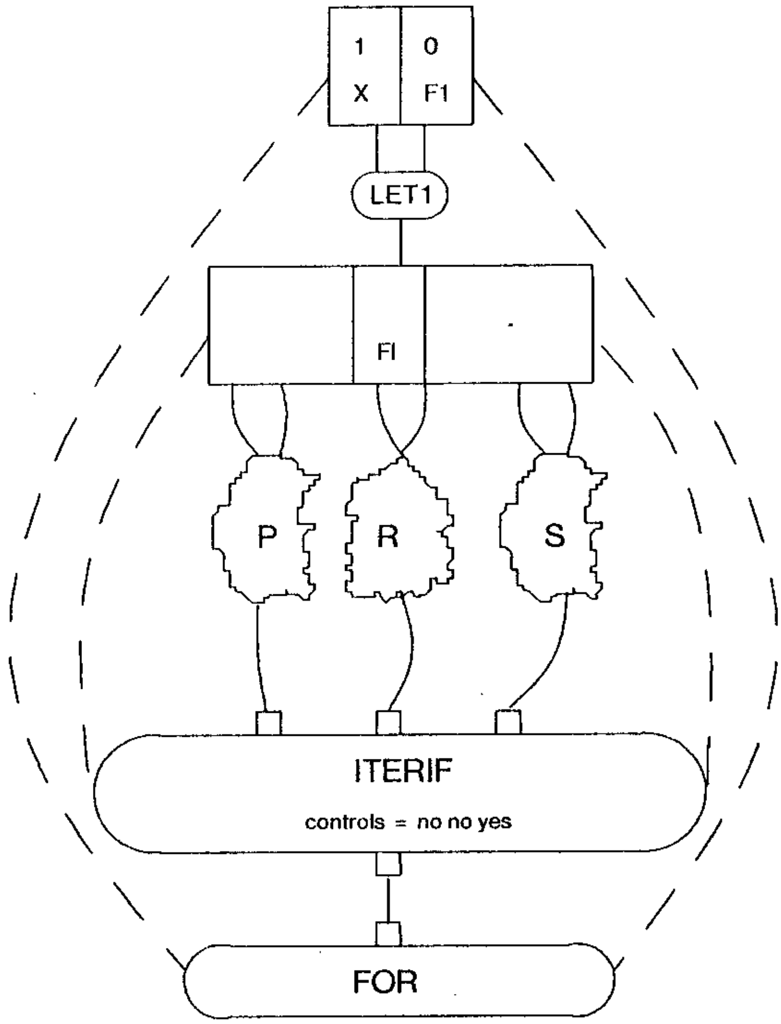

Figure 4-20:VAL, IGF - For Loop with a Let

reproduce the body of the **let** adjacent to the first **iterif** of each copy of the original

iterif.

The general method for copying the body of the **let** is the same as for copying the body an **iterif**. With an **iterif**, we made a copying specification, i.e., we said that we were copying what was between the gate outputs and node inputs of one **iterif**, to the gate outputs and node inputs of another **iterif**. We do the same with a **let**; however, our copying specification is more complicated than it was with an **iterif**. First, the subgraph that we copy is specified by more than one node, and second, we are not reproducing the **let** body at the exact equivalent location, as we did with an **iterif**. With a **let**, we copy what is in between the *from outputs* and *from inputs* to the *to outputs* and *to inputs*. These are defined as follows:

from outputs — All the outputs of the **for** gates that do not go directly to the **iterif** gates, feed into the **let** body. These outputs comprise the *from outputs*.

from inputs — All the inputs to the **iterif** gates not coming directly from the **for** gates, come from the **let** body. These inputs comprise the *from inputs*.

to outputs — For each loop variable in the *from outputs*, we find where the variable is iterated in the iterating body of the **iterif** (similar to what we do when we encounter a loop variable, while splicing in an **iterif** at its gates). We add the output that represents this iteration point to the *to outputs*.

For each free variable in the *from outputs*, we feed this variable through all the **iterif** gates until we reach the *last* **iterif** adjacent to where we are reproducing the **let** body. We add the output of the gate that we build on this *last* **iterif** to the *to outputs*.

For all progressing loop variables in the *from outputs*, we find the addition node (adding X and Δ as described in Section 2.5.1) that is adjacent to where we are reproducing the **let** body. The output of this node is the correct current value of our progressing loop variable. We add this output to the *to outputs*.

to inputs	We take the inputs in *from inputs* and find where the equivalent inputs are located on the *first* **iterif** that is adjacent to where we are reproducing the body of the *LET*. These latter inputs comprise the *to inputs*.

Figure 4-21 shows the unfolding of the loop in Figure 4-20.

4.2.5 Loops with Multiple Yes Arms

A loop that has more than one yes, or iterating, arm within its **iterif** structure (whether nested or not) can be unfolded. However, it is usually not beneficial to do so. When unfolding a loop, it is necessary to reproduce the copy of the original **iterif** body at the location of each yes arm. However, the number of yes arms grows exponentially (with base equal to the number of original yes arms) with each unfolding. For example, if we originally had two yes arms, then we have to copy the **iterif** body at *two* locations. Since each of the original yes arms is copied twice, we now have four yes arms. The original two yes arms have now become maybe arms; therefore, they do not count in the *current* number of yes arms. The next time we unfold the loop we will have to copy the original **iterif** body at *four* locations, yielding eight yes arms, and so on.

We will probably want to unfold certain loops somewhere in the order of 2^8 times. Unfolding a loop that has only one additional yes arm is impractical, let alone unfolding a loop that has more than two yes arms. Even if we were to only unfold loops on the order of 2^3 times, unfolding a loop with more than one yes arm would probably still be impractical.

4.2.6 Nested Loops

The complete model that I have presented so far can unfold nested loops without explicitly adding any more capabilities to what already exists. Consider a one-level

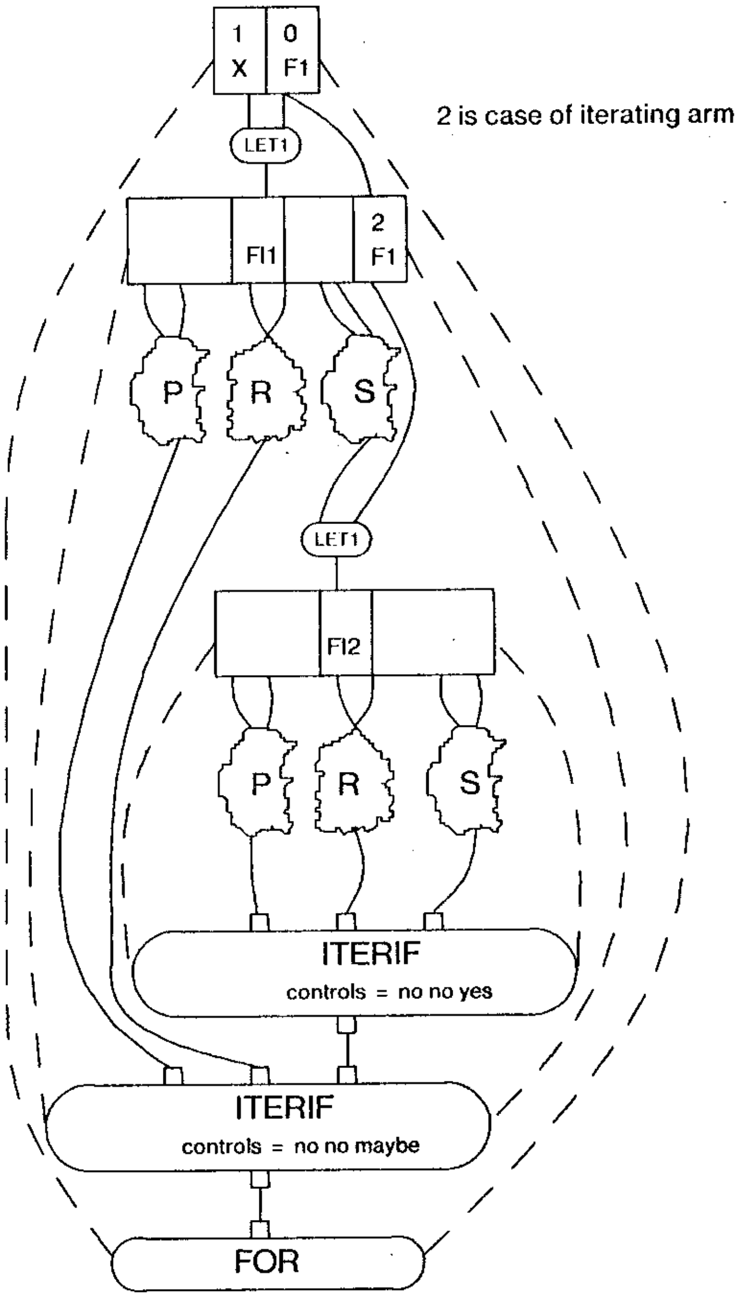

Figure 4-21:IGF - Unfolding, For Loop with a Let

nested loop. The inner loop has no knowledge that it is nested; thus, it will be unfolded just like any other loop. We can also unfold the outer loop normally, since the inner loop resides completely in one of the predicate, return, or iterating arms of the outer loop.

Our model cannot handle the outer loop of a nested loop such as the one shown in Figure 4-22.

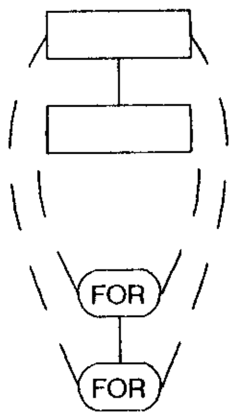

Figure 4-22:SF - Degenerate Nested Loop

Currently, all loops that we unfold must have the one input to the **for** coming from an **iterif** node. However, a loop such as the outer loop shown in Figure 4-22 is not worth unfolding. Because of the semantics of VAL, such a loop is *degenerate*, i.e., it only executes once [1].

Chapter Five

Further Work

The work that would most immediately follow my research project involves completing the design and implementation of the successor variable optimization and updating the unfolding optimization so that they work together. There is also a much wider range of research related to loop unfolding, most of which is discussed in [2]. These issues include:

- array interlace
- combining array interlace and loop unfolding
- how much to interlace and unfold

While an in-depth presentation of these issues is beyond the scope of this thesis, it is worthwhile to discuss them briefly.

5.1 Array Interlace

Interlace is the process of separating an array into n slices. The i^{th} slice of an array contains all elements equal to i mod n. For example, if a nine-element array is divided into three slices, the first slice has elements 1, 4, and 7, the second slice has elements 2, 5, and 8, and so on. Different slices, and the operations performed on them can be assigned to different processors. As we will see in section 5.2, array interlace is used in conjunction with loop unfolding.

5.2 Combining Interlace and Unfolding

Many loops perform a certain operation on every element of an array. If there are no data dependencies, then as noted in section 2.5.1, there will probably be progressing loop variables in the loop and we will want to unfold it. If we were to unfold the loop four times, then the i^{th} copy of the loop would access the i mod 4 elements of the array. Therefore, we would also want an array interlacing of four slices. The i^{th} copy of the loop accesses its array elements from the i^{th} slice of the array.

5.3 Coordinating Interlace and Unfolding Factors

In general, a loop does not access every element of an array. Suppose only the even elements are accessed. The *reference interval* is then equal to two. (If only every third element were accessed, the reference interval would be equal to three). Our desired interlace is equal to reference interval * unfolding. When the reference interval is not equal to one, only the array slices that are a multiple of the reference interval are used. For example, suppose we have an array with 24 elements, but only the even elements are used. An unfolding of three and an interlace of six would be appropriate. In the first cycle the three copies of the loop would access array elements 2, 4, and 6 respectively. In the second cycle, they would access elements 8, 10, and 12 and so on. We see that the odd slices are not used.

It is not always possible to have interlace = unfolding * reference interval. In such cases it is more complicated to transform the program correctly; however, I will not discuss how such a transformation is actually performed. One of the constraints on the amount of unfolding is how much memory is available. Thus, when performing loop unfolding, we have to consider the time/space tradeoff between faster execution and more memory usage. A compiler has to weigh many factors before determining the amount of unfolding and interlace. If a compiler does not have the capabilities to make these decisions, then the use of user input or advice has been proposed. Since the VAL compiler does not currently have these capabilities, the unfolding optimization uses a simple user interface when determining the unfolding factor.

References

1. Ackerman, William B. and Dennis, Jack B. VAL--A Value-Oriented Algorithmic Language: Preliminary Reference Manual. 218, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, jun, 1979. .

2. Ackerman, William B. The VAL Intermediate Graph Format. Memo 235, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, jan, 1984. .

3. Ackerman, William B. Efficient Implementation of Applicative Languages. 323, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, mar, 1984. .