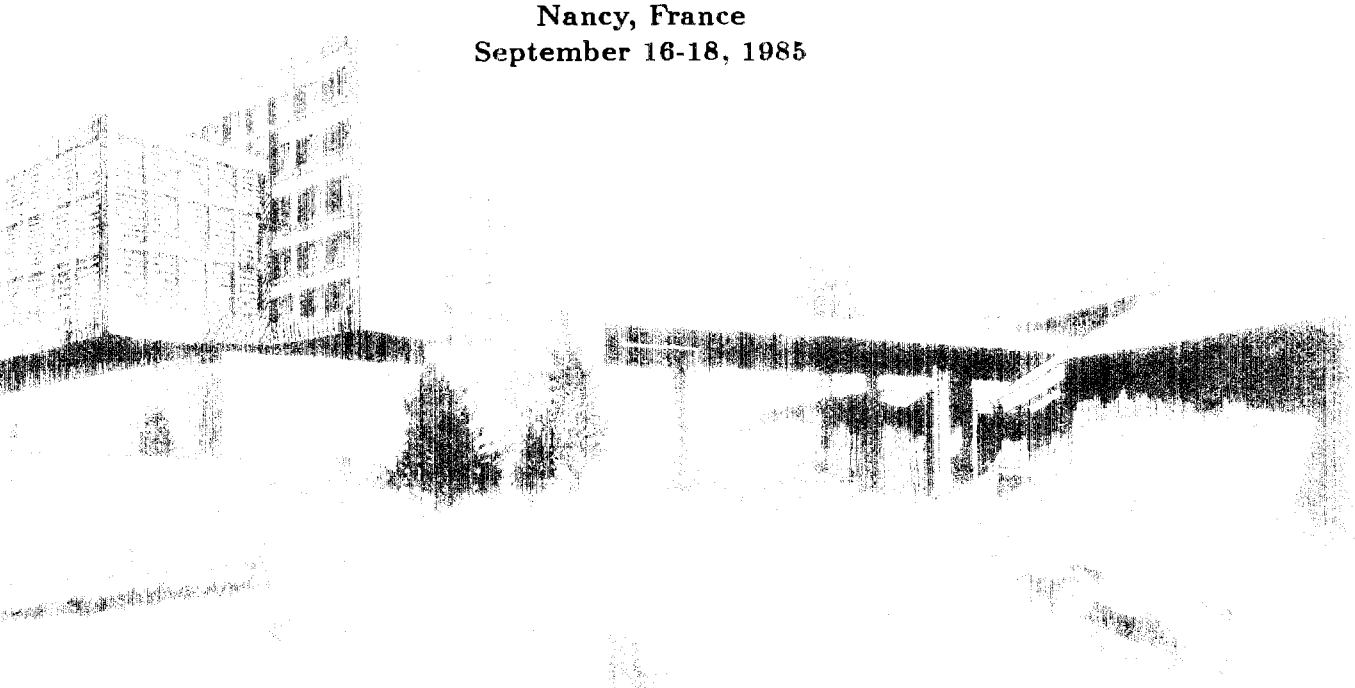
UCRL-92122 Rev. 1

Data Flow Graph Optimization in IF1

December 2, 1987

Presented at the Second
Conference on Functional Programming
and Computer Architectures
Nancy, France
September 16-18, 1985



DATA FLOW GRAPH OPTIMIZATION IN IF1[†]

S.K. Skedzielewski and M.L. Welcome
Computing Research Group
Lawrence Livermore National Laboratory
Livermore, California 94550

ABSTRACT

Optimization techniques are as important when compiling data flow languages as when compiling conventional languages. This paper describes work that has been done on optimizers for SISAL programs that have been translated into IF1 data flow graphs. It shows that conventional optimization algorithms can be easily and efficiently implemented for data flow graphs, and that the payoff for even simple optimizations can be significant.

1. Introduction

Large-scale scientific programming has been synonymous with FORTRAN programming over the past 20 years. One reason for FORTRAN's popularity on supercomputers is that the code produced by the FORTRAN compilers is considered to be "reasonably efficient". Conversely, an argument against the use of dataflow languages is that they cannot be compiled efficiently on a conventional architecture. The latter argument has been neither proven nor disproven; in an attempt to disprove it we are implementing the SISAL language on the Cray-1 architecture.

In order to compare the runtime performance of SISAL and FORTRAN programs we must have comparable implementations. More specifically, we cannot compare a "production quality" FORTRAN compiler with a relatively naive implementation of a SISAL compiler. In order to improve the quality of SISAL code we have implemented some classical code improvement techniques that are common in optimizing compilers for sequential programming languages.

Since many new languages are designed and implemented by small research groups, it is not surprizing that little work has gone into optimization. The work reported here is just the beginning of much more sophisticated code improvements that are needed for the Cray architecture. However, the ease with which all of the improvements were implemented is a big advantage for both the SISAL compiler writer and user, since improved code is very cheap to produce and shows significant runtime benefit.

The remainder of this paper is divided into sections that describe the intermediate form that represents a SISAL program, and each of the code improvers that are now running for SISAL: inline function

---

[†] This work was supported (in part) by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

expansion, common subexpression elimination, loop-invariant expression removal, and loop and test inversion. These techniques are described in [Aho77, Chapter 12] and [Bar79, Chapter 11].

2. Background

SISAL [McG84] is an applicative language that draws on features found in dataflow languages. It could be called a descendant of both VAL [Ack79] and ID [Arv78], but contains some features not found in either. An earlier version of SISAL is described in [McG82].

SISAL is a strongly-typed, procedural, single-assignment language. It is strongly oriented towards "scientific computing" by both its infix expression notation and the many built-in functions for manipulating arrays. Implicit discovery of parallelism guarantees determinate results. It has both parallel and iterative expressions that gather and reduce values produced in the loop bodies. Errors in arithmetic or control return an error value that is well- defined over all of the operations in the language. Finally, streams are provided for interactive input and output, as well as pipeline parallelism.

IF1 [Ske85] is the intermediate form used by all SISAL implementations. It is a hierarchical graph language that describes the dataflow graphs produced from SISAL functions. It also has several properties that make it well-suited for our code improvement analysis. All of the code improvements that we describe in this paper are done on the IF1 form, rather than on the SISAL representation.

SISAL is being implemented on a diverse collection of machine architectures by the following groups

Group	Target Machines
Colorado State University	Denelcor HEP
Digital Equipment Corporation	Vax 11-780
Lawrence Livermore National Laboratory	Cray-1, Cray-X/MP
University of Manchester	Prototype Dataflow Computer

Table 1. Collaborators

The current compilers for SISAL (like many compilers for other languages) perform a straightforward translation. We trade simplicitly in the translator for the use of optimizers. Thus, we depend heavily on good, inexpensive code improvers. Techniques that take hours to run will not be accepted by our users.

We limit our discussion to a few "classic" machine-independent code improvements that we have implemented for IF1. Other problems involving array copying and implementing demand-driven streams will be discussed in future papers.

3. IF1 - an intermediate form

IF1 [Ske85] is a hierarchical graph language that is closely tied to SISAL. It has four elements of interest for these discussions: *graphs, nodes, edges and literal edges.*

An IF1 data flow graph is an acyclic graph in which the nodes represent operations and the directed edges define the data paths between nodes. IF1 nodes are classified as either *simple* or *compound.* Simple nodes represent the basic elements of computation, such as the arithmetic and Boolean operators. Compound nodes represent the more complicated, structured expressions of SISAL. Compound nodes will be discussed shortly.

Each operation node has a collection of input and output ports through which it receives and distributes values. Values enter a node through its input ports, the node performs its operation, and the results are passed out through its output ports. An edge is a directed arc that connects an output port of one node to an input port of another. All edges are strongly typed. Fan-in is not allowed into a node's input ports, but fan-out is allowed (and usually desired) from its output ports. In the figures, nodes are pictured as the small boxes labeled "N d", ports are the small dark pegs on the nodes, and edges are the lines connecting ports.

Constant and literal values are represented by literal edges.. A literal edge is a special kind of edge that does not originate from a node, and always yields the same value. Like normal edges, it is strongly typed. Node 1 of Figure 1.4 has a literal with value "F" on port 1.

An IF1 graph is a collection of nodes interconnected by edges. In order to clearly define the boundary of a graph we use a graph *boundary node* as the source and sink of values entering and leaving the graph. This boundary node contains a set of input and output ports that serve as communication channels between the nodes of the graph and the outside world. Values pass from the outside environment into the graph through input ports. Similarly, values computed inside the graph are passed to the outside world through the boundary's output ports. An input port of a graph boundary node serves as a distribution center for the value it receives. The port supplies this value, via edges, to all nodes within the graph that need it. Fan-out is allowed from a boundary input port to the nodes of the graph, but fan-in is not allowed from the nodes of the graph to their boundary. For simplicity, we will often refer to a graph boundary node as a *graph.* The large boldface boxes in Figures 1.1 through 4.2 represent graphs.

An IF1 graph encapsulates a (possibly) large body of computation. For example, a SISAL function is represented in IF1 by a graph. The input and output parameter lists of the SISAL function header define the input and output ports of its graph boundary node. Graphs in IF1 appear as either function graphs or as subgraphs of compound nodes. Function graphs are at the highest level of the hierarchy; they may contain compound nodes which in turn may contain other compound nodes, etc.

A compound node comprises a number of subgraphs and implicit connections between ports on the subgraphs and ports on the compound node. Each subgraph of a compound node defines a part of a

structured expression. There are five compound nodes defined in IF1 and they mirror the five structured expressions of SISAL.

Name	Use
Select	if-then-else
TagCase	extract values from a discriminated union
LoopA	iteration with termination test after one pass
LoopB	iteration with termination test before one pass
Forall	"apply-to-all"

Table 2. Compound Nodes

The Select node represents a conditional expression. It has subgraphs for the predicate and each of the alternatives.

The TagCase node represents a "case-like" expression used with the SISAL *union* type. It has a subgraph for each tag that the object can assume.

The LoopA and LoopB (see Figure 1.1, Node 1) nodes represent iterative looping expressions. They have subgraphs for initialization, test, body and results.

The Forall node denotes a parallel "apply-to-all" expression. It has generator, body and collector subgraphs.

There are no explicit control nodes or control lines defined within IF1; all control information is implicit within a compound node. Control of subgraph execution is defined by the semantics of each compound node, and may be implemented in different ways on different machines. For example, we will use branch instructions to implement a Select node on the Cray-1, while a static dataflow architecture might use switch and merge operators.

Data values are communicated between a compound node and its subgraphs via specific port assignment conventions. Port assignment conventions are specified for each type of compound node. For example, if a compound node has K inputs then the first K input ports on all of its subgraphs are reserved for these values. Additional ports are reserved for special purposes by the semantics of each compound node.

Example The following SISAL loop computes the sum of the absolute values of the distances between the two functions F and G at the points: (A, B, A), (A, B, A + H), (A, B, A+2H), ... ,(A, B, B).

```
function Example ( A, B, H : real returns real )
    for initial
        X := A;
        V := 0.0
    while X <= B repeat
        X := old X + H;
        V := old V + abs( F( A, B, old X ) - G( A, B, old X ) )
    returns value of V
    end for
end function

function F ( U, V, W : real returns real )
    2.0*U + V*U - 3.0*W*V
end function

function G ( U, V, W : real returns real )
    2.0*U + V*U + 5.0*W
end function
```

The translation of this expression to a LoopB node is shown in Figures 1.1 through 1.5 The LoopB node has four subgraphs:

The INIT graph (Fig. 1.2) initializes the "loop values" X and V.

The TEST graph (Fig. 1.3) evaluates the boolean expression that determines whether the BODY graph will be executed.

The BODY graph (Fig. 1.4) computes the "new" loop values from the "old" loop value and the imported constants.

The RETURNS graph (Fig. 1.5) collects the results to be returned by the LoopB node. In this case, it simply collects the last value that was bound to V before the predicate became false.

Since the LoopB node uses A, B, and H, the first three input ports of each subgraph are reserved for these values. The loop values (X and V) are passed between graphs by exporting and importing them on the next available ports, ports four and five (neither the INIT nor the BODY will output any values on a port corresponding to a loop input). The result of the TEST graph is placed on output port one. The output ports of the RETURNS graph correspond to the output ports of the LoopB node.

Ordering the nodes of a graph by data dependence plays an important role in all of the optimization algorithms presented in this paper.

Definition Two nodes within a graph are said to be *directly data dependent* if an edge exists from one to the other (i.e. one consumes a value produced by the other). The transitive closure of this relation defines the *data dependence* relation.

IF1 graphs are acyclic and hence a node may not be data dependent on itself. Data dependence imposes a partial ordering on the nodes of a graph. We do not worry about data dependence between instances of a loop body, since it falls outside the domain of this discussion. Other analysis routines, such as those that find vector parallelism, or try to update arrays *in situ* will have to deal with more complicated forms of data dependence.

Definition We define a *graph ordering* to be a total ordering on the nodes of a graph, which preserves the partial ordering by data dependence.

Ordering the nodes of an IF1 graph is a simple, linear time algorithm. In the examples of IF1 graphs, the node label will reflect the ordering. If node X depends on node Y, then label(X) > label(Y).

4. Inline function expansion

Function and subroutine calls often impede automatic program analysis in imperative languages. Sharing data by common blocks, call-by-reference parameters, and array aliasing all complicate the global dataflow analysis that is necessary for many optimizations. In SISAL, function calls can have no side effects; all functions are "clean" in that they take and return *values*. Optimization analysis in this environment is simplified by replacing a function call with a copy of the function body. Such "inline expansion" is easily done within IF1. Figures 1.4 and 2.0 show the body subgraph of the example (in Section 3) before and after functions F and G are expanded (the dotted lines in Figure 2.0 outline the function graphs of F and G, they are not part of the expanded graph). Code placed inline can be removed by common subexpression elimination or loop-invariant expression removal (or both).

Inline function expansion both removes the run-time overhead of the function call and creates opportunities for other code improvements. This encourages the programmer to write small, modular SISAL functions that improve readability with no run-time expense. On a sequential machine the expense usually involves parameter evaluation, a context switch and parameter and result copying. On a tagged token data flow machine each token must be retagged as it enters a function.

Although any non-recursive function can be expanded inline, it is not always desirable to do so. For example, very large functions that are called from several locations might explode the code size, and cause extra page faults on a virtual memory machine. Therefore we allow the programmer to identify the functions that should be expanded inline. At this time, we do not allow recursive (or mutually recursive) functions to be expanded. It is a simple matter to relax this constraint and allow recursive functions to be expanded a limitied number of times in order to expand the code size and possibly increase parallelism.

5. Common Subexpression Elimination

It is well known that straight-forward code generation algorithms for compilers often produce redundant code. If a particular operation (e.g. subscripting an array) is used in several places the code to evaluate it

is duplicated. Redundant code may also appear as the result of other optimization passes, such as inline function expansion, or pulling loop-invariant expressions out of a loop. *Common subexpression elimination* (CSE) is the process of finding and removing redundant code, and sharing the value produced by the defining occurrence of the expression. Within IF1, common subexpressions take the form of equivalent operation nodes in a graph.

Definition Two simple nodes are said to be *equivalent* if they have identical operation codes and have equivalent inputs. Two compound nodes are said to be *equivalent* if they have identical operation codes, equivalent inputs, and corresponding subgraphs are isomorphic.

Definition Two graphs are said to be *isomorphic* if there exists a one-to-one and onto mapping from the nodes of one graph to the nodes of the other such that corresponding nodes are equivalent.

Definition Two nodes are said to have *equivalent inputs* if they both have the same number of inputs, and inputs on corresponding ports are both:

(A) Literals with the same type and value *OR*
(B) Edges attached to the same graph (boundary) input port *OR*
(C) Edges attached to corresponding output ports of equivalent nodes.

Definition If two nodes, say N and M, are equivalent then *merging* N into M means that (1) N and all the edges and literals attached to it are removed from the graph, and (2) all the consumers of the values produced by N now get these identical values from the corresponding output ports of M.

In Figure 2.0 of the example, N2 is equivalent to N1, N3 with N4, and N8 with N5. This graph is shown again in Figure 3.0 with these nodes merged. Algorithm 1 illustrates the simplicity of the optimization.

5.1 Observations about our CSE Algorithm

Walking the nodes of the graph in data dependent order simplifies input equivalence testing. By the time node N is checked for equivalence with node M all equivalent nodes preceding N have been merged. Therefore, part (C) of the definition of input equivalence can be replaced by:

(C) Edges attached to the same output port of the same node.

We also observe that the algorithm preserves the ordering by data dependence. We need this property since CSE is often combined with other optimizations that require such an ordering.

```
Input:          A graph G
Output:         The graph with all equivalent nodes merged
Precondition:   The nodes of G are ordered by data dependence

begin
    for each operation code defined within IF1 do
        Create an empty set of nodes
    end for
    { Walk the nodes of the graph in the specified order }
    for each node N do { Depth first search }
        if N is a compound node then
            for each subgraph SG of N do
                call CSE with SG
            end for
        end if
        { Find common subexpressions }
        Let S be the node set with the same operation code as N
        Search S for a node M equivalent to N
        if such an M is found then
            Merge N into M
        else
            Add N to S
        end if
    end for
end
```

Algorithm 1 : Common Subexpression Removal (CSE)

Checking for equivalent compound nodes is potentially expensive; in practice it is not. Two non-equivalent compound nodes usually differ in so many ways that we don't actually compare many nodes before the algorithm terminates.

Certain nodes, such as plus and times, represent commutative operations. Our algorithm will not consider 2*A and A*2 to be equivalent since computer arithmetic is not always commutative. The user can, however, request that we merge such nodes.

CSE does not reassociate operations in an attempt to merge them. The expression (2*A)*B, for example, would not be merged with 2*(A*B) under any circumstances.

5.2 Comparison with other algorithms

A standard common subexpression elimination algorithm for imperative languages appears in [Aho77, chapter 12]. In IF1 the basic unit of analysis is a graph; the basic unit of analysis for an imperative language compiler is a *basic block* (a sequence of statements that contains no branches). The optimizer takes intermediate code produced by the compiler, determines the basic blocks, and constructs a control-flow graph from the blocks. Common subexpression analysis within a basic block is fairly inexpensive, but must be conservative. In the following example,

```
B := A[ I ]
A[ J ] := D
C := A[ I ]
```

A[I] is a common subexpression only if it can be determined that $I \neq J$. If that cannot be determined then the optimizer must assume that I = J and not combine the expressions.

The SISAL compiler directly produces IF1 graphs so no time is spent partitioning the intermediate code. An IF1 graph also represents a more general (and larger) body of computation than a basic block so more common subexpressions might be found. For example, code both before and after a select node can be inspected for common subexpressions without looking at the subgraphs. Finally, any two SISAL expressions that look identical within a scope actually produce the same value.

In order to find and merge common subexpressions from different basic blocks of an imperative code, global data flow analysis must be done (a rather expensive process). Despite this analysis, the optimizer will not have complete information and many common subexpressions will not be merged. Aliasing, call-by-reference parameters and common blocks will force the optimizer to be conservative. In contrast, merging common subexpressions across IF1 graph boundaries is not difficult.

Another optimization, to be discussed shortly, will float loop-invariant operations up out of the subgraphs of loop nodes. It is then possible to merge these new nodes with the nodes already existing within the graph. Similarly, all the nodes within the predicate subgraph of a select node can be moved up into the surrounding graph, again increasing the chances for finding common subexpressions. Finally, equivalent expressions appearing in all the arms of a select node can easily be moved up to the surrounding graph and merged.

6. Loop invariant removal

The detection and removal of loop-invariant expressions is another common compiler optimization. An expression or part of an expression is loop-invariant if it produces the same value on each pass of the loop. Loop-invariant operations may be written by the programmer for clarity, or introduced by another optimizer (e.g. inline expansion), or the code generator (e.g. an array address calculation). We now define what it means for an IF1 node to be invariant within a subgraph of one of the loop nodes (LoopA, LoopB, Forall).

Definition Suppose X is a node within a subgraph G of a loop node L. We say that X is *loop-invariant* in L if each input is

(A) a literal arc *OR*
(B) a port of G reserved for the input values of L *OR*
(C) an edge connected to a producer node that is loop invariant.

Suppose X is loop-invariant and not dependent on any other nodes within G (i.e. all of its inputs satisfy conditions (A) or (B) above). In this case, we promote X out of G and into the graph containing L by the following steps:

(I) Remove X, along with its input edges and literals, from G.

(II) Place X in the graph containing L, at a point directly before L in the data dependent ordering. Reconnect each input edge of X to the source of the corresponding input edge of L and connect X's output ports to newly created input ports on L.

(III) Connect consumer nodes of X in G to the newly created input ports on G. Recall that the input ports on L are implicitly connected to the input ports on G.

(IV) Remove any of L's inputs that were used exclusively by X.

If the nodes of the graph are walked in a data dependent order then condition (C) in the above definition is superfluous when checking for loop-invariance. If Y is loop invariant and satisfies condition (C) then it must be dependent on another loop-invariant node X. By the time Y is reached in the graph walk, X has already been promoted and so all of Y's inputs will satisfy conditions (A) or (B). Notice also that loop-invariant removal preserves the ordering by data dependence.

Figure 3.0 of the example has three loop-invariant nodes: N1, N4, and N5. Figure 4.1 shows the function graph with these nodes promoted from the Loopb body graph and Figure 4.2 shows the improved body graph. LoopB now has four input values so the first four input ports on each of its subgraphs are reserved for these values.

In the process of promoting invariants out of the loop, input ports on L were created and possibly destroyed. This may or may not be desirable, depending on the target machine. On a tagged token dataflow machine, for example, increasing the number of inputs to the loop may actually be more expensive than recomputing the invariant expression each time. As values enter a new scope (the subgraphs) they must be retagged. If the cost of recomputing the loop invariant is "small" then the added loop overhead may exceed the node execution savings.

6.1 Interaction with CSE

Common subexpression elimination and loop-invariant removal can work in lock-step as demonstrated in Algorithm 2. Observe that CSE is called after the invariants have been promoted from all the loop nodes in the graph. Doing so allows the newly promoted nodes to be merged with any existing equivalent nodes. We use depth-first search on loop nodes so that an invariant operation at a deeply nested level can "percolate" up to the point where it is either no longer within a loop or is loop-dependent.

6.2 Comparison with other algorithms

Loop-invariant detection and removal for imperative language compilers can be time consuming. Loop detection not only requires the construction of a control-flow graph for the program, but also the computation of all *dominators* for each basic block within the graph. Finally, loops are detected by

```
Input:          A graph G.
Output:         The graph G with loop invariant nodes
promoted
                out of the loop and all equivalent nodes merged.
Precondition:   The node of the graph are ordered by data
                dependence.

begin { Walk the nodes of the graph in the order defined }
    for each node N do
        if N is a compound node then
            for each subgraph SG of N do {Depth first search }
                call LOOP-CSE with SG
            end for
        end if
        if N is a loop node then
            promote loop invariants out of N
            { See the discussion in Section 6 }
        end if
    end for
    call CSE (non-recursively) with G { this graph only }
end
```

Algorithm 2 : Combined Loop Invariant and CSE Removal

finding "back edges" in the flow graph [Aho77, chapter 13]. In IF1, loop detection is done by simply searching each graph for loop nodes.

Once a loop is detected the code inside its body is searched for loop invariant expressions. The inputs to each expression are checked to see if their definitions are outside the loop. In an imperative language compiler use- definition (u-d) chains must be computed for each variable not previously defined within a block. Moreover, global data flow analysis must be done on the entire program in order to compute u-d chains.

In IF1 the analysis is much simpler; we inspect the input edges on each node in a loop body. If all input edges of the node correspond to input edges of the loop, we have found a loop-invariant operation. This process is very cheap. For example, complete CSE and loop-invariant removal takes only 2.03 CPU seconds on a VAX-11/780 for a 1300 line SISAL program (the Simple benchmark).

7. Loop and test inversion

Another form of loop invariant removal occurs in IF1 when the predicate of a Select node is loop invariant. The loop-invariant removal described in the previous section will not catch such cases, since we must do more than pull an operation out of a loop The predicate for the test may be moved outside of the loop, and each alternative of the Select node executes in its own loop. However, the predicate of the Select node is only executed once. An example should help illustrate the point:

```
for point in Grid at J,K,L
    NewPoint :=
        if J=1 then
            2*Grid[ 2, K, L ] - Grid[ 3, K, L ]
        else
            Point
        end if
returns array of NewPoint
end for
```

Straightforward code generation will emit a Select node buried within three nested Forall nodes (one for each dimension of Grid). It should seem clear that it would be better to have the test only performed at the outermost loop level. The loop-test inversion will do exactly that transformation (at the IF1, rather than SISAL level).

When a loop-invariant predicate is detected the optimizer pulls the Select node out of the loop, and gives each arm of the Select node a modified copy of the loop. It forms the body of each loop by replacing the Select node with the graph of each alternative of the Select node. A smaller example should help show the steps.

```
for Point in Line at J
    NewPoint :=
        if I=1 then
            ( 3*Point[ 2, J ] - Point[ 3, J ] ) / 2.0
        else
            Point
        end if
returns array of NewPoint
end for
```

is transformed first into

```
if I=1 then
    for Point in Line at J
        NewPoint := ( 3*Point[ 2, J ] - Point[ 3, J ] ) / 2.0
    returns array of NewPoint
    end for
else
    for Point in Line at J
        NewPoint := Point
    returns array of NewPoint
    end for
end if
```

Quite often one of the loops produced by the transformation merely copies an array. That loop is collapsed into an edge. The last loop in the example has that form, so it is transformed into the single value "Line".

```
if I=1 then
    for Point in Line at J
        NewPoint := ( 3*Point[ 2, J ] - Point[ 3, J ] ) / 2.0
    returns array of NewPoint
    end for
else
    Line
end if
```

The reason for including such an optimization is that the clearest way to express boundary conditions in SISAL is to use a forall expression that contains a test for the boundary. The Air3D code from NASA Ames Research Center uses seven such **for** expressions to do the boundary conditions on each time step.

The result of running the loop-test inversion on the example at the beginning of the section is:

```
for Plane in Grid at J
    NewPlane :=
        if J=1 then
            for Point in Plane at K, L
                NewPoint := 2*Grid[ 2, K, L ]  - Grid[ 3, K, L ]
            returns array of NewPoint
            end for
        else
            Plane
        end if
returns array of NewPlane
end for
```

8. Experimental Results

The results are shown in the following tables. Table 3 describes the SISAL functions studied. Tables 4-6 give both static and runtime statistics for the smaller programs. Each program is translated into IF1 graphs, and internal functions are expanded inline. Inline expansion causes function bodies to appear at the appropriate nesting level, since no functions are recursive. We counted the nodes at each loop nesting level before and after various optimizers, and compared those ratios to the improvement in execution time by an IF1 interpreter on sample data. In Life and Gauss we see good correlation in program runtime and the node counts at the innermost level. However, when the innermost loop does not dominate the runtime (as occurs in Alpine) the static analysis predicts a much higher improvement than occurred in execution. The functions in Tables 7-9 are from the Air3D benchmark that could not be run on our current interpreter. However, we can see that the code improvers are still removing 20 to 30% of the most deeply nested nodes.

Tables 4, 7 and 9 are particularly interesting since they show the effects of the loop-test inversion. In each case nodes were moved from inner to outer nesting levels, although the overall number of nodes increased. The inversion algorithm creates loops for all alternatives of a Select node as it proceeds, so the code size usually grows. However, the inverted code executes fewer nodes within the loop, so the runtime decreases (cf. Table 4).

9. Summary

We have attempted to show that classical code improvement is very easy to implement for an applicative language. We find that these techniques not only cost very little at compile-time, but also show good performance improvement at runtime. The use of IF1 graphs allows our algorithms to be small, easy to understand, and very fast.

We have only run small examples on our interpreter, but timings from these runs suggest that our static benefit analysis can be used to predict the benefits we will see during the execution of a program. If we can extrapolate to the large codes that we have only analyzed, but not yet run, we can expect 20% improvements on large grid problems at very little cost.

10. Acknowledgements

We wish to thank the groups at Colorado State University and DEC for providing SISAL to IF1 translators. John Glauert is co-author of the IF1 reference manual and strongly influenced its hierarchical structure. We also thank Rea Simpson of LLNL for the SISAL version of the Alpine code, and Cathy Schulbach at NASA Ames Research Center for the copies of the Air3d benchmark.

11. References

[Ack79] Ackerman, W.B. and J.B. Dennis, "VAL--A value-oriented algorithmic language: Preliminary Reference Manual", Tech. Report TR-218, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, MA, June 1979.

[Aho77] Aho, A.V. and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[Arv78] Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine", Tech. Report TR114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[Bar79] Barrett, W.A. and J.D. Couch, *Compiler Construction: Theory and Practice*, Science Research Associates, 1979.

[McG82] McGraw, J.R., S.K. Skedzielewski, "Streams and Iteration in VAL: Additions to a Data Flow Language", *Proc. of the Third International Conference on Distributed Computing Systems*, pp. 730-739, Miami/Ft. Lauderdale, Florida, March 1982, IEEE order number CH-18028.

[McG83] McGraw, J.R., S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas, "SISAL: Streams and Iteration in a Single-Assignment Language", Language Reference Manual, Version 1.2, M-146, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Ske85] Skedzielewski, S.K., and J.R.W. Glauert, "IF1, an Intermediate Form for Applicative Languages", Reference Manual, M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

Life	The game of life — a nice, small 2-D problem
Gauss	Gaussian elimination
Alpine	2-D particle transport function solving Boltzmann's equation. It is interesting because it is also being written in FORTRAN, but it is not a translation of an existing FORTRAN program. It produces a three-dimensional array
Boundary RHSVis Vismat	Subroutines from Air3D, a large three-dimensional grid problem that models fluid flow around a fuselage. The examples from Air3D are interesting because they are part of a large grid problem, for which our optimizations should be very effective.

Table 3. SISAL programs studied

	Node count by static nesting level				Nodes	Improvement	
	0	1	2	3	executed	Static	Dynamic
Raw	7	8	21	98	21913	(1.00)	(1.00)
CSE	6	8	21	71	16620	(0.71)	(0.76)
CSE+Loop	6	8	21	71	16620	0.71)	(0.76)
Loop-Test	6	17	16	71	15901	(0.71)	(0.73)

Table 4. Static and Dynamic results for "Life"

	Node count by static nesting level				Nodes	Improvement	
	0	1	2	3	Executed	Static	Dynamic
Raw	130	117	102	30	7162	(1.00)	(1.00)
CSE	88	105	96	27	6552	(0.87)	(0.91)
CSE+Loop	91	114	75	18	4993	(0.60)	(0.70)
Loop-Test	91	114	75	18	4993	(0.60)	(0.70)

Table 5. Static and Dynamic results for "Gauss"

	Node count by static nesting level							Nodes	Improvement	
	0	1	2	3	4	5	6	executed	Static	Dynamic
Raw	29	4	4	4	4	4	430	52533	(1.00)	(1.00)
CSE	27	4	4	4	4	4	255	41491	(0.59)	(0.79)
CSE+Loop	27	4	4	4	4	4	255	41491	(0.59)	(0.79)
Loop-Test	27	4	4	4	4	4	255	41491	(0.59)	(0.79)

Table 6. Static and Dynamic results for "Alpine"

	Node count by static nesting level			
	0	1	2	3
Raw	73	75	488	409 (1.00)
CSE	71	59	351	371 (0.91)
CSE+Loop	83	96	320	344 (0.84)
Loop-Test	90	106	417	335 (0.82)

Table 7. Static results for "Boundary"

	Node count by static nesting level				
	0	1	2	3	4
Raw	8	5	227	409	9 (1.00)
CSE	7	5	106	244	9 (1.00)
CSE+Loop	21	16	112	208	6 (0.67)
Loop-Test	21	16	112	208	6 (0.67)

Table 8. Static results for "RHSVis"

	Node count by static nesting level			
	0	1	2	3
Raw	19	652	8	58 (1.00)
CSE	16	418	6	49 (0.84)
CSE+Loop	34	395	6	49 (0.84)
Loop-Test	70	855	17	41 (0.71)

Table 9. Static results for "Vismat"

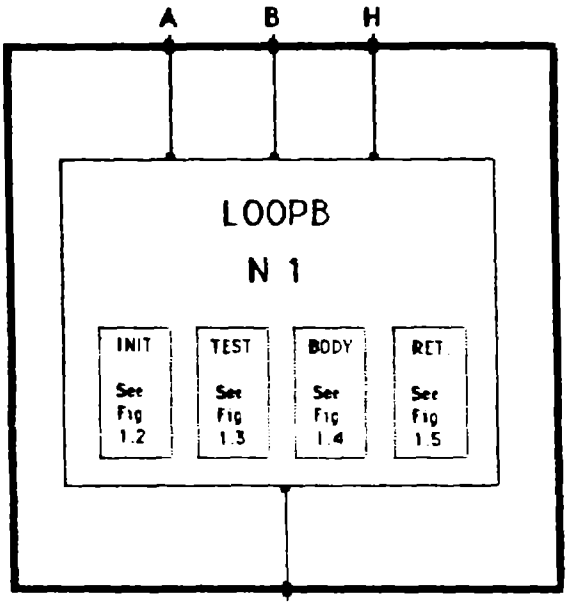


Fig. 1.1 Function Graph of Example

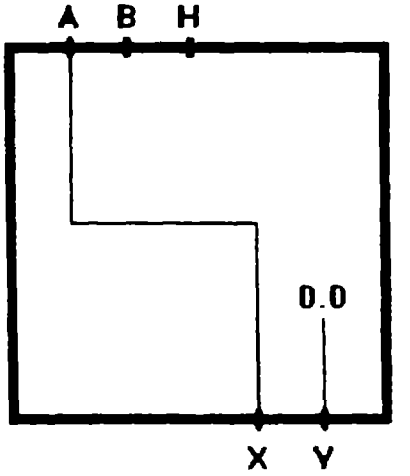


Fig. 1.2 INIT Graph of LoopB Node

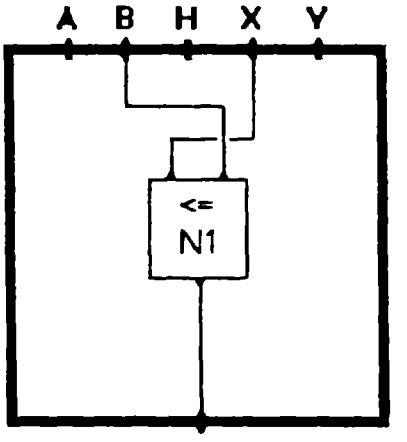


Fig. 1.3 TEST Graph of LoopB Node

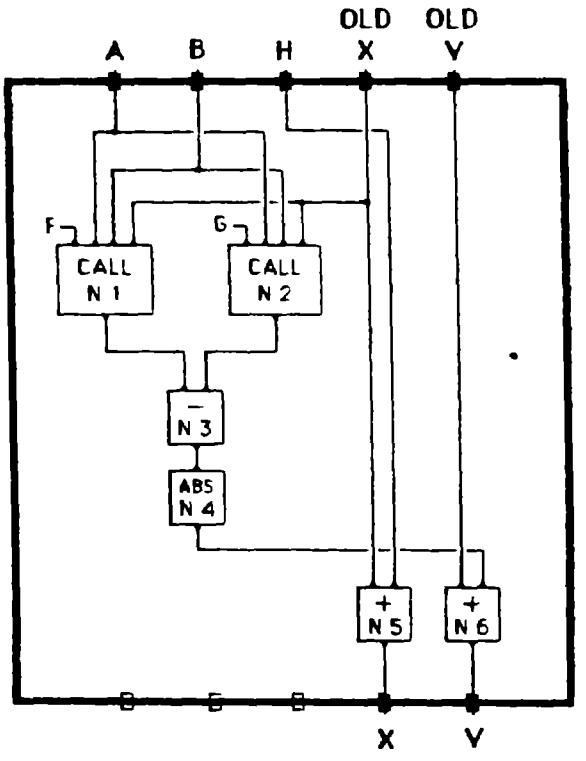


Fig. 1.4 BODY Graph of Example

(Node 3 is a minus)

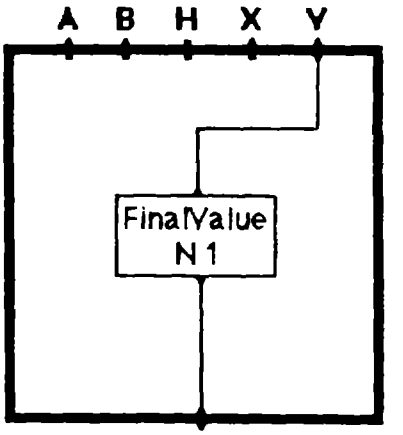


Fig. 1.5 Returns Graph of LoopB Node

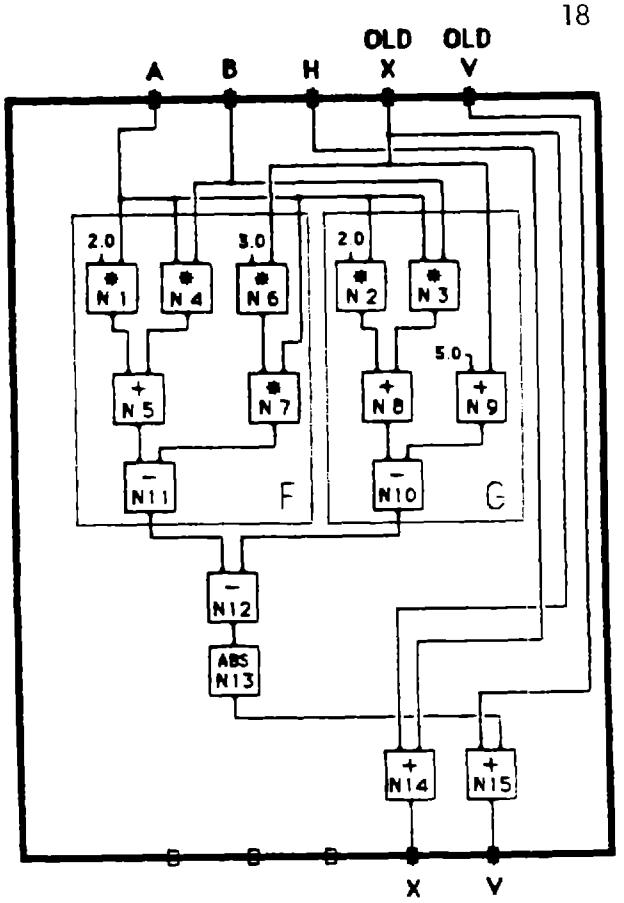


Fig. 2 BODY Graph after Inline Expansion

(Nodes 10, 11, and 12 are minuses)

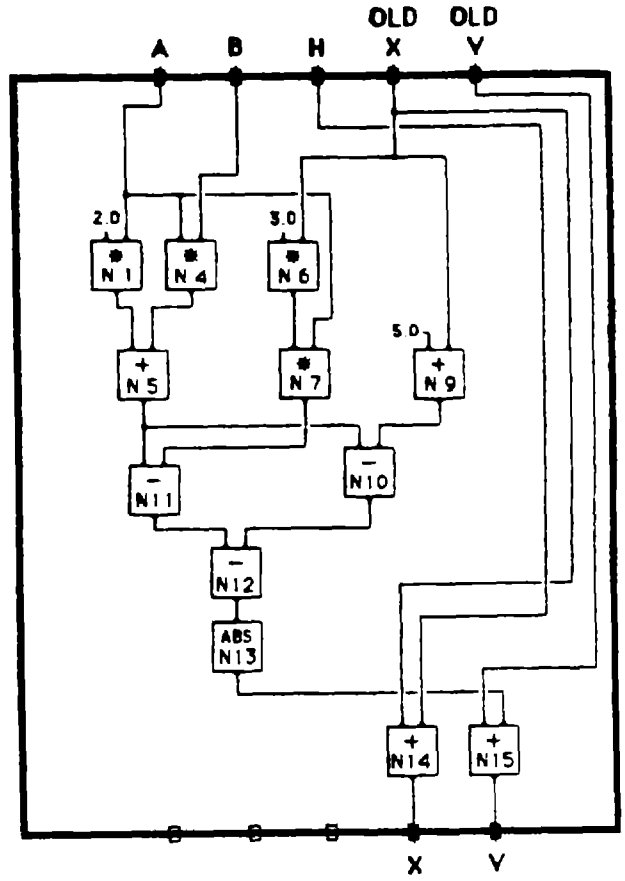


Fig 3 BODY Graph after CSE

(Nodes 10, 11, and 12 are minuses)

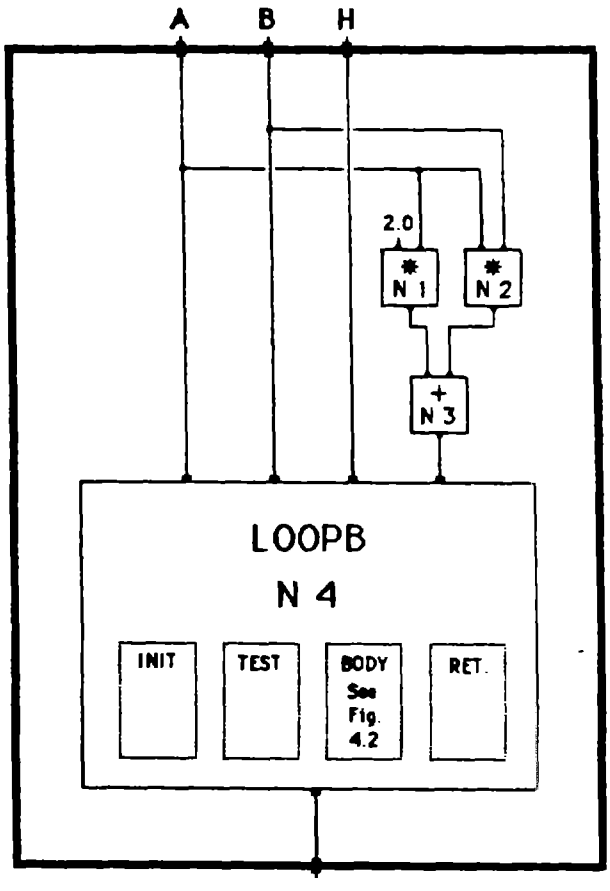


Fig. 4.1 Function Graph After Loop-Invariant Removal

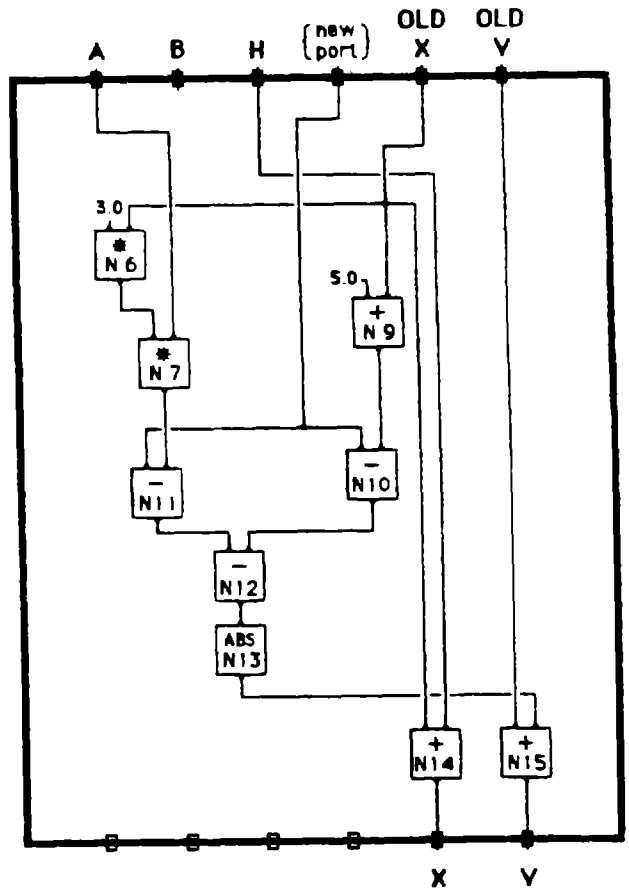


Fig. 4.2 BODY Graph After Loop-Invariant Removal