REPORT DOCUMENTATION PAGE

1. REPORT NUMBER
   AFIT/CI/NR 85-13T

2. TITLE (and Subtitle)
   Sisal Optimizations

3. AUTHOR(s)
   Jody Dejonghe Acres

4. PERFORMING ORGANIZATION NAME AND ADDRESS
   AFIT STUDENT AT: Colorado State Univ

5. TYPE OF REPORT & PERIOD COVERED
   THESIS/DY93BH7811YN

6. PERIOD COVERED

7. AUTHOR(S)
   Jody Dejonghe Acres

8. CONTRACT OR GRANT NUMBER(S)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
   AFIT STUDENT AT: Colorado State Univ

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. REPORT DATE
   1984

12. NUMBER OF PAGES
   36

13. SECURITY CLASS. (of this report)
   UNCLASSIFIED

14. SECURITY CLASS. (of this report)
   UNCLASSIFIED

15. DISTRIBUTION STATEMENT (of this report)
   RESTRICTED

16. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

17. SUPPLEMENTARY NOTES
   APPROVED FOR PUBLIC RELEASE: IAW AFR 190-

18. KEY WORDS (Continue on reverse side if necessary and identify by block number)

19. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

UNCLASSIFIED

DTIC FILE COPY

AD-A151 615
ABSTRACT

The goal of this master's project was to provide several optimizations for the SISAL compiler being developed at Colorado State University. SISAL is a data flow language intended for use on a variety of multiprocessor architectures. Since SISAL is compiled into the intermediate form IFI, which is a common intermediate form for data flow languages, this project concentrated on optimizations that are unique to the characteristics of SISAL, rather than on more traditional optimizations. In particular, the optimizations developed concentrated on making array operations more efficient by including operations where possible and by performing "live analysis" which would tell if data values will ever be needed as it is in a program.
AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AFIT). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: Sisal Optimiizations

AUTHOR: ACRES, Jody DeJonghe

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?
   ( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?
   ( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?
   ( ) a. MAN-YEARS ( ) b. $

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?
   ( ) a. HIGHLY ( ) b. SIGNIFICANT ( ) c. SLIGHTLY ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME

GRADE

POSITION

ORGANIZATION

LOCATION

STATEMENT(s):
SISAL OPTIMIZATIONS

JODY DEJONGHE ACRES
CAPT. USAF
FALL 1994

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
MASTER OF SCIENCE
COMPUTER SCIENCE
COLORADO STATE UNIVERSITY

(36 PAGES)
1.0 OVERVIEW

The purpose of this master's project was to provide several optimizations needed for the SISAL compiler being developed at Colorado State University. In particular, the optimizations considered were loop invariant removal, array optimizations and stream optimizations. The majority of the standard optimizations applied to compilers will be provided by other sources. This project's main goal, therefore, was to provide "non-standard" optimizations that are required by some of the more unique characteristics of the SISAL compiler.

2.0 BACKGROUND

SISAL (Streams and Iteration in a Single Assignment Language) is a data flow language intended for use on a variety of multiprocessor architectures. The current project at CSU is to produce a compiler for a Denelcor HEP multiprocessor. The basic characteristics of SISAL are: 1) no side effects; 2) locality of effect; 3) parallelism constrained only by data dependencies; and 4) single assignment. [Cobb, 1984] The main objective of SISAL is to provide a programming language for expressing algorithms which will make it easy to detect and exploit any implicit parallelism. The main application area being numerical computations that are straining current high performance machines. Being a "data flow" language, the statements in SISAL are not necessarily executed sequentially. Instead, they are executed whenever their input data is available. This parallelism is obtained since all instructions
whose data is available can theoretically be executed simultaneously. See [McGraw, 1983] for more information on SISAL.

The SISAL compiler produces an intermediate form referred to as "IF1". IF1 is an intermediate form designed specifically for data flow languages. It describes a program as a data flow graph with nodes representing operations, edges representing data and types associated with an edge identifying the characteristic of the data being passed. Figure 1 shows an example of the graphical representation of the expression $(a + b) + c$ and the IF1 description of this graph. The numbers in the boxes are port numbers that are used to identify the values. See [Staller, 1984] for a complete description of IF1.

![Graphical representation of the expression $(a + b) + c$ and its IF1 description.](image)

**Figure 1**
To make the IFI easier to traverse during compilation, it is encoded into a graphical format using pointers. The nodes, ports, and edges are all represented by a "C" structure containing information about each. They are linked together via pointers which allow one to traverse the graph in either a forward or backward direction. To follow the graph forward one goes from a node to a port (representing an output value), to the edge (representing a use of this value), to a node that uses this data. To follow the graph backward, one goes from a node to an edge representing an input value (physically different than the edge followed in the forward direction) to the port that this value came from, back to the node that produced the value.

Figure D shows this graphical encoding of the previous example in * to *7. Going forward from a port and following the linked list of edges will give you all uses of this value. Going backward from a node and following this linked list of edges will give you all the input values for this node. The optimizations performed in this project were applied to this graphical encoding of a SISAL program.
FIGURE 2
2. LOOP OPTIMIZATIONS

The loop optimization considered in this project was loop invariant removal. This involves finding any instruction inside of a loop whose computation stays constant for each iteration of the loop and moving it outside of the loop. This way it is only executed once saving execution time. Because of the characteristics of GIGAL and the graphical encoding of IFI, identifying loop invariants is simpler than classical approaches to solving this problem. In particular, basic blocks and loops are already identified in the intermediate form and information is contained in the graphical encoding. Moving the code outside of the loop, however, is more difficult.

2. LOOP INVARIANT REMOVAL ALGORITHM

The steps involved in removing loop invariants from IFI are:

1. Traverse the graph recursively backwards and for each "while" or "for all" node found perform steps 2-6.

2. Traverse the nodes in the body of the loop recursively backwards treating each node as invariant if all of its input values are invariant. An input value is considered invariant if:
   a) it is a constant;
   b) its source node is node 0 meaning it comes from outside the loop or
   c) its source node has been previously marked as invariant. If a loop node is encountered during this traversal, steps 1-6 are performed on this node before it is processed.

3. Once all invariant nodes have been identified, the one
moved immediately before the loop node. This may cause nodes to be renumbered.

4) The input edges to the nodes moved in 3 must be modified to point to the proper source port. This will be undefined if the source was another moved node or a constant. If the source port was from node C, however, the corresponding source from outside the loop must be found and the edge wired into it.

5) Any outputs from the nodes moved in 3 that are still needed inside the loop must be channeled into node A of the loop and all nodes that use these values must be hooked up to these new ports.

6) Finally, the ports of node A may need to be compacted. Some of the only users of a particular input value may have been nodes moved outside the loop. Some ports can be deleted and the remaining ports compacted.

3.2. STATUS OF LOOP OPTIMIZATIONS

Steps 1-7 identified above had been completed when it was discovered that this process had already been done at another site. We received copies of the code and executable object and after running several tests and analyzing the code I verified that it did indeed remove loop invariants properly. It was decided to stop any further work on this effort and continue working on optimizations that were unique to the work being done at CSH. Some documentation explaining the loop optimization code we received is contained in Appdx A. Although the code probably correctly I did have some questions about the code
itself. First it seems to only consider an input value as important if its source node is the graph node model 1. In our system, however, constants have no source node. I did not see these with constants and they were optimized correctly, so their meaning must just be different than ours. Also I don't think the code we received was actually executable. The only thing we needed to do was to make them different and not correct. In addition, I couldn't find any place in the code that fixed the inputs to match those that weren't moved to reflect the correct location of inputs that were moved. There were, however, many routines that were not updated with the code and this part could have been handled or one of them.

10. INTERNAL REPRESENTATIONS

The next task was to analyze the array processing in SISAL and to determine if there were any optimizations that could be performed to make it more efficient. This involved visualizing the interpretation of SISAL into IF1 and then into the various libraries that actually perform the array actions to see if a 'IF1' was needed, if any, was needed.

11. ARRAY INTERPRETATION

This section outlines how each array operation in SISAL is
implemented in the various routines that actually perform the
array operation.

Brian, please let me know if there will be any high-level
modules.

Okay, let me know which ones.
Select - returns a pointer to the array element at index J

\[
\text{SISAL: } A[J] \\
\text{IF1:}
\begin{align*}
\text{array(T)} & \quad \text{integer} \\
\text{T} & \quad \text{V}
\end{align*}
\]

Runtime - select

Params:

\begin{align*}
\text{NAME} & \quad \text{MEANING} & \quad \text{SOURCE} \\
\text{dvptr} & \quad \text{ptr to dops vector to start dereferencing at} & \quad \text{from symbol table} \\
\text{size} & \quad \text{element size} & \quad \text{array type} \\
\text{ndim} & \quad \text{number of levels in the array} & \quad \text{array type} \\
\text{boundarr} & \quad \text{ptr to subscripts to apply to array} & \quad \text{input edges}
\end{align*}

Replacement - replaces the array element at index J with value V

\[
\text{SICAL: } A[J;V] \\
\text{IF1:}
\begin{align*}
\text{array(T)} & \quad \text{int} & \quad \text{value(s)} \\
\text{T} & \quad \text{V} & \quad \text{V}
\end{align*}
\]

Runtime - replace

Params:

\begin{align*}
\text{NAME} & \quad \text{MEANING} & \quad \text{SOURCE} \\
\text{dvptr} & \quad \text{ptr to dops vector of input array} & \quad \text{symbol table} \\
\text{newdvptr} & \quad \text{new dops vector} & \quad \text{output param} \\
\text{dflag} & \quad \text{boolean indicating if element to replace is an array} & \quad \text{array type} \\
\text{numval} & \quad \text{number of elements to replace} & \quad \text{input edges}
\end{align*}
concatenation - concatenates two or more arrays

\[ \text{SIGMA} = \text{A} \oplus \text{B} \]

\[
\begin{align*}
\text{IF} & : \\
\text{array}(T) & \text{array}(T) \\
& \text{ACATENATE} \\
\text{array}(T)
\end{align*}
\]

Runtime - concat

Params:
- `dflag`: boolean indicating if input arrays contain dope vectors
- `numarr`: number of arrays being concatenated
- `size`: element size
- `newdvec`: new dope vector
- `array`: array of ptrs to dope vectors for input arrays symbol table

Add high/low - appends a single value to either end of an array

\[ \text{SIGMA} = \text{array_addl}(A, V) \quad \text{array_addl}(A, V) \]

\[
\begin{align*}
\text{IF} & : \\
\text{array}(T) & \text{value} \\
& \text{ADDL} \text{ADDD4} \\
\text{array}(T)
\end{align*}
\]
Functions:

NAME       MEANING                      SOURCE
array      ptr to array vector        symbol table
newarray   new array vector          output param
value      ptr to value to append    input edge
rectype    rectype of edge           edge type
flags      boolean indicating if     edge type
           value is an array

Remove high low element - returns the array A with its high
index decreased by one or its low index increased by one

SISAL - array_reml(A)      array_reml(A)

IF:
array(T)

v

\[ \text{AREML/AREMH} \]

v
array(T)

Routine - array_reml, array_remh

Parameters:

NAME       MEANING                      SOURCE
dypt       ptr to input array         symbol table
copy       new copy vector            output param
elem       element size of array      array type

Set low limit - adds LD - array_lim(A) to all elements thus
shifting the origin of the array

SISAL - array_setl(A,LO)

IF1 -
array(T)   LO

v

\[ \text{ASETL} \]

v
array(T)
Runtime - array_user

Params:
<table>
<thead>
<tr>
<th>NAME</th>
<th>MEANING</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>dpyr</td>
<td>ptr to input array</td>
<td>symbol table</td>
</tr>
<tr>
<td>newdpyr</td>
<td>new dope vector</td>
<td>output param</td>
</tr>
<tr>
<td>lobound</td>
<td>lower bound for new array</td>
<td>input edge</td>
</tr>
</tbody>
</table>

Set bounds - returns an array with range (LO,Hl) with the same data as the input array, where possible. If LO > array_liml(A) or HI > array_limh(A) elements will be missing.

SISAL - array_adjust(A,LO,HI)

IF 1 -
array(T) LO HI

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>AADJUST</th>
</tr>
</thead>
</table>

array(T)

Runtime - array_adjust

Params:
<table>
<thead>
<tr>
<th>NAME</th>
<th>MEANING</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo</td>
<td>lower bound</td>
<td>input edge</td>
</tr>
<tr>
<td>hi</td>
<td>upper bound</td>
<td>input edge</td>
</tr>
<tr>
<td>size</td>
<td>element size of array</td>
<td>array type</td>
</tr>
<tr>
<td>dpyr</td>
<td>pointer to input array</td>
<td>symbol table</td>
</tr>
<tr>
<td>newdpyr</td>
<td>new dope vector</td>
<td>output param</td>
</tr>
</tbody>
</table>

4.1 ANALYSIS RESULTS

There were several observations from the above analysis:

1) The majority of the input parameters needed by the runtime routines are directly available from either the IFI or the symbol table. The only exception to this was parameter "replace" used in the replace routine to determine if the input arra was ever used again. If it isn't, the replacement can take
plan without having to copy the array. Some kind of "live" memory needs to be done to supply this value.

3) Although both SISAL and the runtime routines allow for slicing in referencing a multi-dimensional array, the IFI breaks them down into indexing by only one index at a time. This is done to allow loop invariant removal and common sub-expression optimizations to be performed, however, it means some of the capabilities of the runtime routine will never be used. The options taken for one call to the runtime routine versus several calls to dereference an element are probably not much different. However, a routine that would combine the referenced into one IFI node after all optimizations have been performed could be useful.

4) A similar thing happens with concatenates as they are broken down into concatenating only two arrays at a time by the intermediate form. Both SISAL and the runtime routine will allow multiple arrays to be concatenated at once. In this case, however, the results of several calls to the runtime routine versus just one can result in less efficiency. This happens because for each call a new logical array will be allocated to hold the results of the concatenate. One call to the runtime routine would only have to allocate one logical array. An optimization that would go through the IFI and combine any group of concatenates into one could increase the efficiency of the runtime routine.

4) There is an IFI node available called AllEmpty, which returns a boolean indicating if there are any elements in the array. I couldn't find any SISAL statement that used this node.
There is a runtime routine which deallocates arrays. I was unable to find any documentation that outlined when an array would be deallocated.

From those observations it was decided that routines were needed to combine a series of concatenates into one concatenate or a series of selects into one select. A routine was also needed to do a "live" analysis on an array input to a replace node. Instead of writing a specific routine for this purpose, however, a generalized routine which given any input edge will determine if the value is ever needed again would satisfy this requirement and would also be useful in other areas of code generation.

4.2 COMBINING CONCATENATES

As mentioned above, a routine was written to combine a series of concatenate nodes in IFI into a single node when possible. Figure 7 shows an example of an IFI graph with several concatenate nodes and the graph that would result from this routine. The code for this routine is contained in appendix B and described below.

```
array1  array2
   |       |
   v       v
ACatenate  ACatenate
   |       |
   v       v
array3  array

FIGURE 7
```
The main strategy is to start with a concatenate node from the IF1 graph and then by following its inputs recursively backwards through the graph to determine if any of the inputs to this node are also concatenates. If so, the input edges of the higher concatenate are linked into the lower concatenate node instead. Before the edges are linked to the lower node, however, its inputs are also checked so that multiple levels of concatenates can ultimately be combined together.

The main routine is called "combinecats" and expects as input a pointer to a concatenate node. It first initializes a linked list which will be used to hold the input edges to the final combined concatenate node. It then calls a routine "followedges" which will cycle back through all the input edges to the node and place the inputs to all nodes that can be combined in the linked list. After it returns from this routine it sets the backptr in the original node to point to this new list of input edges and sets the next pointer of the last edge to NULL.

Routine "followedges" cycles through each of the input edges to a node. For each edge it looks at the port that produced this value and checks if there is more than one use of the value. If so, it does not combine the nodes since this intermediate value is needed somewhere else. If the value is not used on any of the ports back to the source node, if it is a concatenate node, the nodes can be combined and it calls itself to start checking the inputs for this node. Once an input edge is found whose source node can not be combined the input edge is placed in the linked list of input edges for the combined node.
In addition, the corresponding "forward" edge for this input edge must be made to point to the combined node instead of the original node that may have already been combined. Once all edges for the input node have been processed the routine returns. Because of the order in which edges will be processed in this routine, the edges placed on the linked list of edges will end up in the correct order.

Figure 4a is a detailed example of the IF1 graphical encoding of two concatenates that can be combined into one. Concatenates are called with a pointer to node 5. It in turn calls followedges with node 5. Followedges looks at the first input edge. By going back up to the source port and then looking at the linked list of edges pointed to by this port it sees that there is only one use of this value. It also sees that the source node is a concatenate and so it calls itself with a pointer to node 7. The first input to node 7 is checked and although there is only one use of this value the source node is not a concatenate. This means this input edge must be placed on the linked list and the edge pointed to by node 1 port 1 must be node 7 point to node 5 instead of node 3. Now the second input to node 7 is checked. It is similar to edge 1 and so it is also added to the linked list and its corresponding forward edge made to point to node 5. This completes the processing of node 7. Next, the second input edge to node 5 is processed. The first edge is found that the input array value is needed by another node and thus the source node can not be combined. The edge is therefore put on the linked list of edges and its forward edge made to point to node 5 (which it always did).
This is the last input edge for node 5 and so "followed" returns to "combineSel". The table for node 5 is set in the linked list of edges and the next pointer for the last edge set to NULL. The resulting graph is shown in figure 4b.

4.1 COMBINE SELECTS

The routine to combine selects is very similar to that described above for concatenate above. The goal being to combine the dereferencing of several dimensions of a multidimensional array into one runtime call. The code for this routine is contained in appendix C and described below.

Again, the strategy is to start with a select node from the IF graph and then by following its inputs recursively backwards through the graph to determine if any of its inputs are other selects. If they are and there is no other use of the calculated index for this dimension of the array is added to a list of indexes to be used in dereferencing an element.

The main routine is called "combineSel" and expects as input a pointer to a select or "aalement" node. It initializes a linked list which will hold all the indexes for the final select node. It then calls routine "followback" to collect all the input edges for this node and to place all the indexes that can be combined in the linked list. After it returns from this routine it gives up the pointers to the node of the pointer into the array to be dereferenced followed by all the indexes. The array to be dereferenced is the first up to the input node of the graph that could be dereferenced.
Routing "followback" looks at the first input to the node which is the case that is input to the select. It looks at the source code for this input and if it is also a select and there is an output of this intermediate value then the node can be continued. It then calls itself with this new code to see if the input to this node can also be continued. Once a node that needs to be continued is found, the routine adds the node for the current input into the linked list of nodes on. The corresponding "forward" edge is also made to point to the combined node. As the routine returns from all its recursive calls, all indexes will be added to the linked list.

4.5 LIVE ANALYSIS

Given a pointer to an edge, this routine will return 0 if the value is not used again or 1 if it is. The basic strategy here is to determine if there are any other uses of the value produced by the source port of this edge. Because of the parallel manner in which the IF nodes are executed, any other use of a value could possibly be a later use of the value. The only exception would be if there was a data dependency that required the other use of this value to occur before the particular use being analyzed.

The code for live analysis is contained in Appendix D and described here. The main routine is called "livel" and expects as input a pointer to the edge whose "liveness" is in question and a pointer to the node that this edge is input to. The edge must be the "back edge" that one would find by following the edge one input from a node to the "forward edge" one would get by
following the same pointer from a node. The routine first initializes list to be false (0). It then finds the first node that produced this value and makes a list of all other nodes that use this value. Associated with each node on the list is a flag which is initially set to 1 which means that the node is "live." If there are any nodes on the list, i.e., there are other nodes, it calls "checkdown" to see if there is a data dependency. This will require the code using the edge in question to appear after the other nodes that use this value. If no node on the list has such a dependence, then the value is considered to be live. If a live use of this value has been found, the next step is to determine if the edge in question is embedded in a compound node and was passed in from outside the node. If so it is possible that there is another use of the value outside of the compound node. The graph node for the compound node and the corresponding input edge is found. The liveness of the original edge is then determined by calling "islive" with this corresponding edge.

Routine "checkdown" takes as input a pointer to a node that has been checked and a pointer to a list of nodes against which it is to be checked for data dependence. The routine cycles through each input edge to the node to be checked and finds the edge with this value. It then cycles through the linked list of nodes and sets its flag to 0 if a match is found. This shows that there is a data dependence and the node is no longer live. If the node is not code in the graph code it immediately exits it with the same error to continue checking for data dependence.

80
3.3 STREAM OPTIMIZATIONS

The next objective of this project was to analyze the various instructions in SIGAL to determine if any of them were efficient. If any were, then some of the optimization that could be done to the code would include the optimization of streamlining the instructions. Unfortunately, it
Please provide the necessary information to understand the stream functions mentioned in the document. Currently, waiting for some of this information and thus the entire analysis could not be updated. The SICL-IF relationship was analyzed and is shown in the next section.

**STREAM ANALYSIS**

Create a stream - creates a stream of the desired type with no elements.

```plaintext
SICL: stream type-name[]

IF1: a stream type is created. No node is generated.
```

Append - appends "v" to the end of stream "g".

```plaintext
SICL: stream.append(g,v)

IF1

``` steam(T)  value
```

```
AADDH
```

```
stream(T)
```

Credit -

Delete first element - returns the first element of a stream.

```plaintext
SICL: stream.first(g)

IF1

``` steam(T)  value
```

```
AIFEMENT
```

```
T
```
Runtime -

Select all but first element - returns a stream identical to the input stream except with the first element removed

**SISAL** - *stream_rest*(G)

**IF1** -

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>AREML</td>
<td>AREML</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>stream(T)</td>
<td>stream(T)</td>
</tr>
</tbody>
</table>
```

Runtime -

Test for empty - returns true if there are no elements in the stream and false otherwise

**SISAL** - *stream_empty*(G)

**IF1** -

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>AISEMPTY</td>
<td>AISEMPTY</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>
```

Runtime -

Number of elements - returns the number of elements in G

**SISAL** - *stream_size*(G)
IFI -

```
stream(G)

<table>
<thead>
<tr>
<th>ASIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Runtime -

```

Concatenate - returns a stream of all elements of G followed by all elements of H

SISAL - G !! H

IFI -

```
stream(T)  stream(T)

<table>
<thead>
<tr>
<th>ACATENATE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Runtime -

5.2 STREAM ANALYSIS CONCLUSIONS

Not much can be said until the runtime routines are analyzed. It seems though that the same sort of things done for arrays may need to be done with streams. In particular, concatenation could possibly be combined if the runtime routine can handle more than two at a time. Also some kind of live analysis may be useful in determining when a stream is no longer needed.
4.2. PROJECT CONCLUSIONS

The goal of this project was to provide some optimizations for the SISAL compiler which would allow it to produce more efficient code. Since SISAL is compiled into the intermediate form IFI which is a common intermediate form for data flow languages, we were looking for optimizations that were unique to the characteristics of SISAL rather than the more traditional optimizations. As we found out with loop optimization, these more traditional optimizations are being worked on at other sites. Arrays were one area of SISAL that could be a potential problem area if no optimizations were applied. In particular, replace operations could quickly eat up a lot of memory if the array had to be copied each time. The combining of concatenate and select nodes and the live analysis provided by this project will hopefully increase the efficiency of array operations in SISAL.

As for my personal gains from this project, I learned a lot about data flow machines and data flow programming languages. The concepts of data flow and parallel execution add another dimension to the way one traditionally thinks about programming. Looking at the loop optimizations also forced me to analyze those methods more thoroughly.
REFERENCES


APPENDIX A

IFILoop Documentation

IFILoop is a command that can be executed to perform loop
optimizations on an IFI file. This documentation describes the
routines that perform this optimization. The main routine is
Unloop.

AddInputPorts:

This function will add input ports to a loop graph. This
includes adding ports to the graph as well as its subgraphs
(both the test, body, and results subgraphs).

Inputs:
- Pointer to the node to add ports to
- Number of ports to add

Called by:
- None

Called by:
- None

CompactPorts:

This routine cycles through the input ports for a loop node
and deletes any that are no longer used.

Inputs:
- Pointer to node whose ports will be compacted

Output:
- None

Called by:
- ImproveCompoundNode

Called by:
- None

ImproveCompoundNode:

This function cycles through the nodes of a loop and call
IsMoveable to determine if a node is invariant. If so, it call
MoveNodeOut to move the node into the graph surrounding the
loop. After processing all nodes in the loop, if any nodes were
moved, it calls CompactPorts to renumber the inputs to the loop
to reflect deletions and additions of inputs caused by the
move.

Inputs:
- Pointer to the compound node to be checked for
  invariant

Output:
- None

Called by:
- ImproveGraph

Called:
- IsMoveable
  - MoveNodeOut
  - CompactPorts
ImproveGraph -

This function goes through the graph of a function looking for compound nodes. When one is found it calls itself to improve the subgraphs of this graph. It then checks whether the node is a loop construct and calls ImproveCompoundNode to remove the loop invariants. After the entire graph has been processed it calls RemoveGraphCSE to remove common subexpressions.

Inputs - Pointer to the first node in graph
Calls - None
Called by - LoopImprove
Calls - ImproveGraph
- ImproveCompoundNode
- RemoveGraphCSE

LoopImprove -

This is the main procedure for the loop optimizations. It cycles through the functions of a program calling other routines to perform the optimizations on each function.

Inputs - Pointer to a function in the program
Outputs - Indirectly, an improved IFI graph
Called by - None
Calls - ImproveGraph
- RemoveGraph

MoveNodeOut -

This function moves a node from one location to another by first changing the input edges to the node to reflect the environment of the outer graph. Next, AddInputPorts is called to create new input ports into the loop to hold the outputs of those moved nodes and their outputs are wired into these ports.

Inputs - Pointer to the node to be moved
- Pointer to subgraph in compound node that the node to
be moved resides in
Pointer to compound node that the node is to be moved out of

Output: Current number of input ports to the compound node
Called by: ImproveCompoundNode
Calls: AddInputPorts
APPENDIX E
CODE TO COMBINE CONCATENATE NODES

```
#include "basenode.h"
struct edgetype *eliststart, *elistend;
    /* Used to keep a linked list of input edges for the */
    /* combined node */
struct *masternode;
    /* Points to the combined node */

COMBINECATS
    /* This routine will combine a series of concatenate nodes. */
    /* by following the input edges of a concatenate node and */
    /* and linking the input edges of all nodes that can be */
    /* combined into the node. */

edgetype *edges[nodedepth];
    /* Points to node to start */
    /* combining at */

masternode = edgetype;
    /* Remember this starting node */
eliststart = ellistend = NULL;
    /* Initialize list of edges */
edgedata [3];
    /* Used to renumber edges */
allwedges[nodedepth];
    /* Call followedge to do the */
    /* actual combining */
edgedata = backptr = elliststart;
    /* Make node point to this */
    /* new list of edges */
edgestart = ellistend = NULL;
    /* Fix last edge on list */

FOLLOWEDGES
    /* This function will cycle recursively backwards through */
    /* depth determining if nodes can be combined. When it */
    /* reaches the last node that can be combined on a given */
    /* path it puts its input edges on a list of edges for the */
    /* final combined node. */

followedgedes[nodedepth];
struct *edgetype
    /* points to the node whose edges */
    /* are to be followed */;
struct edgetype *sourceedge, *destedge;
struct edgetype *sourcepart;
sourceedge = nodedepth backptr;
    /* get first input edge */
    /* now cycle through all its input edges checking if the */
* Input: source node can be combined */

while (sourceedge != NULL) {
    sourceport = sourceedge->ptr.up;
    /* if this value is not needed elsewhere and the source */
    /* node is a concatenate it can be combined */
    if (sourceport->usage->nextedge == NULL &&
        (sourceport->source->typenode == concatenate)) {
        sourceport->source->beenhere = 1;
        followedges(sourceport->source);
    } else {    /* the source node can not be combined and */
        /* this edge is put on the edge list */
        if (eliststart == NULL)
            eliststart = sourcedge;
        else
            elistend->nextedge = sourcedge;
        elistend = sourcedge;
    } /* the corresponding "forward edge" must be made to */
    /* point to the final combined node */
    destedge = sourcedge->ptr.up->usage;
    while (destedge->ptr.dn != nodeptr)
        destedge = destedge->nextedge;
    destedge->ptr.dn = masternode;
    destedge->port = sourcedge->port = edgeno++;
}
sourcedge = sourcedge->nextedge;
APPENDIX C
CODE TO COMBINE SELECT NODES

#include "nodesif,encode/encode.h"
struct edgetype *eliststart, *elistend;
   /* Used to keep a linked list of input edges for the */
   /* combined node */
struct node *masternode;
   /* Points to the combined node */
struct node *topnode;
   /* Points to the highest node in the graph */
   /* ** be combined */

/* combinerese */

/* This routine will combine a series of select or */
/* "element" nodes by following the array input to the */
/* node and linking the indexes of all nodes that can be */
/* * Set node to void one node. */

combine(nodeptr)
   struct node *nodeptr; /* Points to node to start */
   /* combining at */
{
    masternode = nodeptr; /* Remember this starting node */
    toplevel = nodeptr;  /* initialize the highest node */
    eliststart = elistend = NULL; /*Initialize list of indexes*/
    edgenum = 0;  /* Used to renumber edges */
    followed = nodeptr; /* Call follow back to do the */
    /* actual combining */
    depth = &depthptr = toplevel->backptr; /* make node point to*/
    /* input array */
    nodeptr->depthptr->nextedge = eliststart; /*link in list of*/
    /* indexes */
    eliststart->nextedge = NULL; /* fix last edge on list */
    /* now fix up first input edge */
    edgetptr = toplevel->backptr->ptr.u->usage;
    while (edgetptr->ptr.dn != toplevel)
       edgetptr = edgetptr->nextedge;
    edgetptr->ptr.fr = nodeptr;
    edgetptr->ptr.fr = 1;
}

78
followback(nodeptr)
struct IFnode *nodeptr; /* points to the node whose edges */
/* are to be followed */

struct edgetype *sourceedge, *destedge;
struct portptr *topnode;

/* this way if the input's source node can be obtained */
sourcexport = nodeptr->backptr->p4rt.up;
/* if this value is not needed elsewhere and the source */
/* node is an element it can be combined */
if ((sourcexport->usage->nextedge == NULL) &&
    (sourcexport->type->node == arra,element)) {
    /* node->sourcexport->source; */
    /* node->backhere = 1; */
    followback(topnode);
}

sourceedge = nodeptr->backptr->nextedge;
/* put the index on the linked list */
if (elliststart == NULL)
    elliststart = sourceedge;
else
    ellistend = sourceedge;

if (the corresponding "forward edge" must be made to */
/* point to the final combined node */
destedge = sourceedge->ptr.up->usage;
while (destedge->ptr.in = nodeptr)
    destedge = destedge->nextedge;
/* destedge->ptr.in = masternode; */
destedge->p4rt = sourcexedge->p4rt = edgeno++;
}
sourcexedge = sourcexedge->nextedge;
APPENDIX D

CODE TO TEST IF AN EDGE IS LIVE

```
#include <sys/encode/syndef.h>

struct nodelist {  
    struct Encode *nodeptr;
    int depends;
    struct nodelist *nextnode;
}; /* The structure will be used to hold a list nodes that */
/* also use the value being tested */

/* ISLIVE */

/* This routine checks if a given data value represented */
/* by an edge will ever be used again. */

islive(innode, inedge)
struct Encode *innode
    /* points to the node that the */
    /* edge is question is input to*/
    struct edgetype *inedge;
    /* points to the "backedge" */
    /* whose liveness is to be */
    /* tested */
/* contains a linked */
/* list of other nodes that use */
/* this data value */
struct edgetype *edgeptr;
int live, found;

live = 0; /* initialize live to false */

/* build list of all other uses */

liststart = NULL;
edgeptr = inedge->ptr.up->usage;
while (edgeptr != NULL) {
    if ((edgeptr->ptr.dn != innode) &&
        (edgeptr->port != inedge->port)) {
        nptr = (struct nodelist *)malloc(sizeof
            (struct nodelist));
        nptr->nodeptr = edgeptr->ptr.dn;
        nptr->depends = 1; /* initialize it to be live */
        nptr->nextnode = liststart;
        liststart = nptr;
    }
    edgeptr = edgeptr->nextedge;
```
Pseudocode for the next step of the algorithm:

1. **Input:** A graph, a root node, a destination node

2. **Initialization:** Initialize the current node as the root node and the current path as an empty list.

3. **Path Construction:**
   - While the current node is not the destination node:
     - For each neighbor of the current node:
       - If the neighbor is not in the current path:
         - Add the neighbor to the current path.

4. **Path Adjustment:**
   - While the next node in the current path is not the destination node:
     - If there is an edge from the next node to the destination node:
       - Add the edge to the current path.
     - Else:
       - Remove the last node from the current path.

5. **Output:** The current path as the shortest path from the root to the destination.

Note: This algorithm assumes a graph with a single source and destination node, and each edge has a non-negative weight. The algorithm works by iteratively adding the next node to the path until it reaches the destination node or finds a path that cannot be improved by adding another node.
If selecting edges:

edge = edge + 1

if edge > next input edge:
    go to

else:
    add edge to next input edge

end

if next input edge:
    add edge

else:
    exit

end

edge = edge + 1

if edge > next input edge:
    go to

else:
    add edge to next input edge

end

if next input edge:
    add edge

else:
    exit

end

edge = edge + 1

if edge > next input edge:
    go to

else:
    add edge to next input edge

end

if next input edge:
    add edge

else:
    exit

end