COMPILING ALGORITHMS AND TECHNIQUES
FOR THE S-810 VECTOR PROCESSOR

Michiaki Yasumura, Yoshikazu Tanaka, Yasushi Kanada
Central Research Laboratory
Hitachi Co. Ltd.
Kawaguchi, Tokyo 185
Japan

Aiko Koyama
Software Works
Hitachi Co. Ltd.
Tatsuka, Yokohama 224
Japan

Abstract — This paper describes the compiling algorithms and techniques for the Hitachi S-810 vector processor. The data dependency analysis method presented here is based on the algorithm by K. Takeuchi, et al.[1]. The results are similar to but the approach different from the loop distribution algorithm by D. Kuck, et al.[6]. A new data flow algorithm for data dependency of variables under IF statements is described. For this purpose new data flow operators are introduced. Some program transformation techniques are shown to be useful for enhancing vectorization. The issues on vector object optimization techniques are also described. With these algorithms, typical vectorization ratio of the S-810 vector compiler is about 30% higher than that of Hitachi's previous vector compiler and more than 600MLOPS were attained for one FORTRAN program. Most of these algorithms and techniques are easily adaptable to other vector processors.

Introduction

Over the past decade Hitachi has been developing two types of vector processors and their compilers. One, called IAP (Integrated Array Processor), is integrated in general purpose miniframes. Examples are the Hitachi M180 IAP, M200H IAP, and M280H IAP. Other companies in Japan also make this kind of vector processor, for example, Nippon Electric Company (ACOS/1000 IAP) and Mitsubishi (MELCOM COSMO7000 IAP). The conceptual base of IAP is transparency and high cost/performance. Transparency is achieved through the interruptible vector instructions (thus they are memory-to-memory instructions) and automatic vector compilers. Therefore, IAP can be used not only for FORTRAN programs in batch mode but also for APL programs in interactive mode. The IAP system can be put together with a small amount of extra hardware. However, the performance ratio of IAP to scalar processor is generally not so high. A typical performance ratio for IAP is about 2 to 3, with a maximum ratio of about 10.

The other type of vector processor is the Hitachi S-810 model 20 and model 10. This type is a dedicated scientific supercomputer with vector registers. Other companies in Japan have developed or have been developing this type as well, e.g., Fujitsu (VF-200 and VF-100), and Nippon Electric Company (SX-2 and SX-1). The primary goal of these processors is high performance. But user friendliness is also important, and the automatic vector compiler is the key to achieving these goals.

A basic data dependency algorithm has been developed for the M180 and M200H IAP compiler[1]. Techniques for vectorizing IF statements and program transformation techniques have been developed for the M200H IAP compiler[2,5,6,11]. Based on the above algorithm and techniques, we have developed extended vectorization techniques and algorithms for the S-810 compiler to enhance vectorization and to generate more efficient vector objects.

Data Dependency Analysis

To vectorize FORTRAN programs automatically, the order of operation execution in DO loops has to be changed. In the scalar processing mode, each operation is executed for each index value. After all operations in a DO loop are executed for an index value, the index value is incremented, and each operation is executed again for the new index value, and so on. In the vector processing mode, each operation is executed for all index values, and the next operation is executed for all index values, and so on. This change in execution order is called loop distribution or vectorization (Fig. 1).

FORTRAN DO Loop

\[
\begin{align*}
\text{DO} & \ 10 \ i=1,N \\
A(1)=B(1)+C(1) \Rightarrow \ (A_{i}, B_{i}, C_{i}, i=1, N) \\
10 & \ B(1)=A(1)+E(1) \quad (D_{i}, A_{i-1}, B_{i}, i=1, N)
\end{align*}
\]

Fig. 1 Loop Distribution (Vectorization)
Whether or not loop distribution is permissible depends on the value defined and its usage for the data. This dependency is usually called data dependency. Data dependency analysis has been studied for many years for variables in the field of scalar object optimization. However, the study of data dependency analysis of arrays for vectorization is relatively new. One study publicized in this field is the work done by D. Kuck et al. [6].

The method of a data dependency analysis described here is slightly different from theirs. In this method, data dependency relations are classified into five categories: the first is suitably independent, the second unsuitably dependent, the third specially dependent, the fourth unknown dependent, and the fifth independent. Examples of the unsuitably dependent case are shown in Fig. 2. Array A is unsuitably dependent in DO 10, whereas variable S is unsuitably dependent in DO 20.

DO 10 I=1,N
  A(I)+B(I)+C(I)

DO 20 I=1,N
  A(I)+S=B(I)
  C=C+I

Fig. 2 Unsuitably Dependent Case

The specially dependent case is a variation of unsuitably dependent case. Examples of the specially dependent case are shown in Table 1. These special operations can be vectorized with special hardware support.

Table 1 Specially Dependent Case

<table>
<thead>
<tr>
<th>Macro Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Sum</td>
<td>S=S+A(I)</td>
</tr>
<tr>
<td>Vector Product</td>
<td>S=S*A(I)</td>
</tr>
<tr>
<td>Inner Product</td>
<td>S=S+A(I)*B(I)</td>
</tr>
<tr>
<td>Vector Iteration</td>
<td>A(I+1)=A(I)*B(I)+C(I)</td>
</tr>
<tr>
<td>Vector Max</td>
<td>S=MAX(S,A(I))</td>
</tr>
<tr>
<td>Vector Min</td>
<td>S=MIN(S,A(I))</td>
</tr>
</tbody>
</table>

Our basic dependency analysis is done for variables and for arrays according to the following two rules:

(1) A variable is unsuitably dependent if there is a defining occurrence and its first occurrence is not a defining occurrence.

(2) An array is unsuitably dependent if one of the two occurrences is a defining occurrence and the preceding occurrence contains a subscript, the value of which is "less than"(*) the value of the subscript of the succeeding occurrence.

(*) The value of subscript F is "less than" the value of subscript F whenever

\[ F_2 \leq F_1 \]

where \( F_1 = (f_1, f_2, ..., f_n) \) and \( F_2 = (f_1', f_2', ..., f_n') \) are ordered sets of subscript values.

For more details on the basic data dependency algorithm, see [1].

The vectorization analysis is based on this basic algorithm[1] and is enhanced by some program transformation techniques. At least three program transformation techniques are related to the data dependency analysis. These are statement exchanging, loop splitting, and loop unrolling for a cyclic index.

During the data dependency analysis, if two statements are exchangeable, they are exchanged to reduce unsuitably dependency. Two statements are exchangeable if variables/arrays in two statements are mutually unsuitably dependent or independent but not suitably dependent nor unknown.

In the course of the data dependency analysis, a loop is split into vectorizable parts and unsuitably vectorizable parts. The loop splitting algorithm is as follows:

(1) Let an assignment statement or a conditional statement be an S-Block (Split-Block).

(2) Analyze the data dependencies within a vectorizable S-Block and among S-Blocks and mark unvectorizable on the S-Block if it contains unsuitably dependent or unknown dependent variables/arrays.

(3) Combine adjacent unvectorizable S-Blocks. Repeat step 2 until all S-Blocks are checked.

(4) Combine adjacent vectorizable S-Blocks.

The resultant S-Block is quite similar to the PI-Block produced by the data dependency graph[6].

Data dependency analysis is effective only for linear indexes. Data dependency analysis for non-linear indexes is quite difficult. Therefore, an array with a non-linear index can be vectorized if its occurrences are used only or it appears only once. Otherwise it should be declared independent by a user (Fig. 3).
* OPTION VEC
  DO 10 I=1,N
    A(L(1))=A(L(1))+B(1)
  10 CONTINUE

Fig. 3 Forced to Vectorize Case

A cyclic index is a non-linear index. A program with a cyclic index, however, can be vectorized through a loop unrolling technique. In Fig. 4 cyclic index j is removed by loop unrolling.

Original loop

\[ \text{s} \]
\[ \text{DO 10 I=1,N} \]
\[ A(I)=B(I)+C(I) \]
\[ \text{END} \]

Unrolled loop

\[ \text{s} \]
\[ \text{DO 10 I=1,N} \]
\[ A(I)=B(I)+C(I) \]
\[ \text{END} \]

Fig. 4 Loop Unrolling for A Cyclic Index

Vectorizing IF Statements

To vectorize IF statements, control flow and data flow or data dependency under IF statements are first analyzed.

The main purposes of control flow analysis are to detect anomalies, such as internal loops, or branches into control structures (see Fig. 5) and clarify control and controlled relationships.

DO 10 I=1,N
  IF(1) GOTO 2
  \[ s1 \]
  GOTO 3
  \[ s2 \]
  IF(3) THEN
  \[ s2 \]
  ELSE
  \[ s3 \]
  GOTO 1
  \[ s4 \]
  END IF
  \[ s5 \]
  10 CONTINUE

Fig. 5 An Anomalous Control Structure

Control flow analysis is relatively easy and little is new to vector compilers.

The situation is different for data flow analysis. Data dependency of arrays within IF statements is the same as that without IF statements. However data dependency of variables with IF statements is different (Fig. 6).

Case A is unvectorizable, even though the definition of variable \( s \) precedes its use textually. Case B is vectorizable, since the variable is totally defined and both definitions precede its use. Case C is vectorizable, though the variable is partially defined. Here, a variable is totally defined if its defining occurrence appears on every path of the flow graph. Otherwise it is called partially defined.

Case A: IF(1) THEN
  \[ s \]
  ELSE
  \[ s \]
  ENDIF

Case B: IF(1) THEN
  \[ s \]
  ELSE
  \[ s \]
  ENDIF

Case C: IF(1) THEN
  \[ s \]
  ELSE
  \[ s \]
  ENDIF

Fig. 6 Data Dependency Under IF

Thus the data dependency condition is modified as follows:

(1) A variable is unsuitable dependent if there is a defining occurrence and there is a path on which the defining occurrence does not precede the other occurrence.

To check the above condition, the depth-first traverse approach was first attempted for the IAP compiler. This method is simple and there is no extra memory except for backtracking. However it was too slow to analyze a fairly large DO loop with many IF statements. Therefore we introduced if-then-else reduction method which reduces IF-then or if-then-else branches and that makes the depth-first method practical.[12] Nevertheless, the depth-first method is intrinsically time consuming process.

So we have developed the breadth-first data flow method for the S-810 compiler. To facilitate this method we have introduced the data flow operators, \( \bigoplus \) and \( \bigotimes \).

For each variable \( v \) and each index \( i \) in a flow graph, there are three data flow variables \( \text{IN}_i(v) \), \( \text{OWN}_i(v) \), and \( \text{OUT}_i(v) \) defined as follows:

\[ \text{IN}_i(v) = \bigoplus \text{OUT}_i(v) \]
\[ \text{OWN}_i(v) = \bigotimes \text{IN}_i(v) \]
\[ \text{OUT}_i(v) = \text{OWN}_i(v) \]

Here, \( \text{IN}_i(v) \) is the input status for the variable \( v \) in vertex \( i \), \( \text{OWN}_i(v) \) is the own status
for the variable \( v \) in vertex \( i \). \( \text{OUT}(v) \) is the output status for the variable \( v \) in vertex \( i \).

Semantics of \( \text{IF}^* \) and \( \text{IM} \) is defined in Table 2. The value \(-1\) is interpreted as the unsuitably dependent state, whereas the value \(+1\) as the suitably dependent state. And the initial state of each variable is set to \( 0 \) as a neutral state.

<table>
<thead>
<tr>
<th>( \text{OR}^* )</th>
<th>( \text{OWN} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{X} )</td>
<td>( \text{X} )</td>
</tr>
<tr>
<td>( \text{X} )</td>
<td>( \text{X} )</td>
</tr>
<tr>
<td>( \text{X} )</td>
<td>( \text{X} )</td>
</tr>
</tbody>
</table>

Binary \( \text{IF}^* \) is defined by binary \( \text{IF} \) as follows:

\[
\text{IF}^*(X_1, X_2, ..., X_n) = X_1 \text{IF}(X_2, X_3, ..., X_n)
\]

\( j=1..n \)

Using these operators and status variables, the flow graph is traversed in breadth-first order. If the value \( \text{OUT} \) of the final vertex is not \(-1\), then it is suitably dependent or independent. (See an example in Fig. 7)

```
DO 10 i=1,N
  IF(A(i).EQ.1) THEN
    A(i)=0.0
    A(i)=S*B(i)
  ELSE
    A(i)=S*B(i)
  END IF
10 CONTINUE
```

Fig. 8 Semantics analysis of an IF statement

This example is an unsuitably dependent case by definition, since there is a definition of a variable and there is a path on which there is no preceding definition of the usage. However, when the semantics of IF statement is considered, this IF expression is index independent (we call this type of IF statement loop-invariant IF statement). Therefore this definition of the variable is either always executed or not executed at all for all index values. Thus this type of variables can be vectorized.

The other techniques for enhancing vectorization of IF statements is related to the program transformation techniques. One example is the loop unrolling of edge conditions. (See Fig. 9) IF statement of an edge condition is removed by the loop unrolling.

```
DO 10 i=1,N
  IF(i.EQ.1) THEN
    A(i)=0.0
    A(i)=S*B(i)
  ELSE
    A(i)=S*B(i)
  END IF
10 CONTINUE
```

Fig. 9 Loop Unrolling for an Edge Condition

Vector Object Optimizations

Object optimization techniques for vector compilers are vector text optimizations, vector register assignments, vector memory managements, and other machine dependent optimizations.

Vector text optimization is a common technique for the IAP and the S-810 vector processor and it is similar to scalar text optimization. Some of the vector text optimization techniques are common expression elimination, invariant expression moveout, and dead code elimination. Among them the first two are most effective for vector processors.

Vector register assignment is the one of the important tasks for the S-810 type vector processor. Vector memory management is the important task for IAP type vector processors. The main target of vector memory management is the efficient use of temporary vector in memory. Examples of machine dependent optimization for the S-810 are:

1. Use of VMA(Vector Multiply with scalars and Add) instruction instead of VM(Vector Multiply) with scalar and VA(Vector Add) instruction.

2. Parallel execution of vector instructions with their preparing instructions (Fig. 10).
Here, the TestVF instruction, which will wait for the end of vector instructions, is moved out from the outer DO loop.

```
DO 10 i=1,N
A(1,j)=D(1)
DO 10 i=2,N
10 A(i,j)=B(i,j)+C(1,j)
```

![Sequence of Instructions](image)

**Fig. 10** Scalar/vector Parallel Execution

(3) Compression of vector arguments for intrinsic functions under IF statements (Fig. 11). The argument is composed of those elements, each of which corresponds to the true case of IF-expression.

```
DO 10 i=1,N
IF (A(i).NE.0) THEN
B(i)=SQRT(A(i))
ENDIF
10 CONTINUE
```

![Compression of A Function Argument](image)

**Fig. 11** A Compression of A Function Argument

Among these optimization techniques, the vector register assignment is the most important and most difficult one. Little has been reported on vector register assignment. The strategy should be different from scalar register assignment, since vector processors like S-810 execute multiple vector instructions in parallel and the access to the same vector register by different instructions may hinder their parallel execution. The vector instruction specification also imposes some restrictions on the vector register assignment for each instruction. Thus we employ tabulated LRU (Least Recently Used) method to assign vector registers in place of simple Round-Robin.

**Results**

Though the basic algorithm of the data dependency analysis of the S-810 is the same as that of the IA64, a lot of techniques of enhancing vectorization are used for the S-810 compilers. With these enhancements, the vectorization ratio of typical scientific FORTRAN programs has increased about 30%. And the performance ratio of S-810 vector mode to scalar mode is about 10^4-10^6. Maximum speed which was attained for a thermal conduction program written in FORTRAN program compiled by the S-810 compiler is 687 MFLOPS (Million Floating Operations Per Second) [13].

**Conclusion**

We have described some basic data flow algorithms and some vectorizing techniques. Among them, the algorithm of analyzing data flow of variables under IF statements utilizes the breadth-first approach. For that algorithm, we have introduced the concept of data flow operator. The algorithms and techniques developed for the S-810 vector compiler are effective as well as practical. Some program transformation techniques are especially useful for enhancing vectorization. We believe that program transformations by vector compilers should be further extended to vectorize many more ordinary programs.

**Acknowledgement**

The authors would like to acknowledge Seiichi Takehashi for his helpful comments during the preparation of this paper.

**References**


Information Processing. To be published.


