COMPILING ALGORITHMS AND TECHNIQUES FOR THE S-810 VECTOR PROCESSOR

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

Akio Aoyama
Software Works
Hitachi Co. Ltd.
Totsuka, Yokohama 244
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 R. Takanuki, 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 680MFLOPS 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 mainframes. 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 (ACOS1000 IAP) and Mitsubishi (MELCOM COSMO700 IAP). The conceptual base of IAP is transparency and high cost/performance. Transparency is achieved through the interruptable 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 super computer with vector registers. Other companies in Japan have developed or have been developing this type as well, e.g., Fujitsu (VP-200 and VP-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 M280H IAP compiler[4],[5],[12]. 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.

In this paper, we first briefly describe basic algorithms for data dependency analysis. The data dependency analysis under IF statements is then explained. Finally, the issues on vector object optimization are described.

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, one 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		Vector mode
DO 10 i=1,N		
A(i)=B(i)+C(i)	=>	$(A_i=B_i+C_i, i=1,N)$
10 D(i)=A(i)*E(i)		$(D_i=A_i*E_i, i=1,N)$

Fig. 1 Loop Distribution(Vectorization)

Whether or not loop distribution is permissible depends on the value defined and its usage for each data. This dependency is generally 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 dependent, 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-1)=B(i)+C(i)
10 D(i)=A(i)*B(i)

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

Macro Operation	Example
Vector Sum	S=S+A(i)
Vector Product	S=S*A(i)
Inner Product	S=S+A(i)*B(i)
Vector Iteration	A(i+1)=A(i)*B(i)+C(i)
Vector Max	S=MAX(S,A(i))
Vector Min	S=MIN(S,A(i))

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_i is "less than" the value of subscript F_j iff $\exists\, p,q(1 \le p < q \le n)\ f_i q = f_j p$

where $F_i = (f_i 1, f_i 2, \ldots\ldots, f_i n)$ and

$F_j = (f_j 1, f_j 2, \ldots\ldots, f_j n)$

are ordered set 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 unsuitable dependency. Two statements are exchangeable iff 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 unvectorizable parts. The loop splitting algorithm is as follows:

(1) Let an assignment statement or a conditional statement be a 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, such as indirect addressing is quite difficult. Therefore, an array with a non-linear index can be vectorized if its occurrences are use only or it appears only once. Otherwise it should be declared independent by a user (Fig. 3).

```
*VOPTION VEC
      DO 10 i=1,N
      A(L(i))=A(L(i))+B(i)
   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          Unrolled loop

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

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(e1) GOTO 2
 1       s1
       GOTO 3
 2     IF(e2) THEN
          s2
        ELSE
          s3
          GOTO 1
        END IF
 3    s4
 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 occurence appears on every path of the flow graph. Otherwise it is called partially defined.

```
Case A:  IF(  ) THEN
           S=...
         ELSE
           ..=S
         ENDIF
          ...=S

Case B:  IF(  ) THEN
           S=...
         ELSE
           S=...
         ENDIF
          ...=S

Case C:  IF(  ) THEN
           S=...
           ..=S
         ELSE
           ...
         ENDIF
```

Fig. 6 Data Dependency Under IF

Thus the data dependency condition is modified as follows:

(1)' A variable is unsuitably 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, $\cap^*$ and $\cup^*$.

For each variable v and each index i in a flow graph, there are three data flow variables $IN_i(v)$, $OWN_i(v)$, $OUT_i(v)$ defined as follows:

$$IN_i(v) = \bigcap^*_{j \in Pred(i)} OUT_j(v)$$

$$OWN_i(v) = \begin{cases} -1 & \text{(use precedes)} \\ 0 & \text{(not appear)} \\ +1 & \text{(def precedes)} \end{cases}$$

$$OUT_i(v) = IN_i \cup^* OWN_i(v)$$

Here, $IN_i(v)$ is the input status for the variable v in vertex i, $OWN_i(v)$ is the own status for the variable v in vertex i, $OUT_i(v)$ is the output status for the variable v in vertex i.

Semantics of ∩* and ∪* 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 neutral state.

Table 2. Data flow operators.

∩*

	-1	0	+1
-1	-1	-1	-1
0	-1	0	0
+1	-1	0	+1

∪*

		OWN		
		-1	0	+1
IN	-1	-1	-1	-1
	0	-1	0	+1
	+1	+1	+1	+1

Unary ∩* is defined by binary ∩* as follows:

$$\bigcap^*_{j=1..n} X_j = X_1 \cap^* X_2 \cap^* \ldots \cap^* X_n$$

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

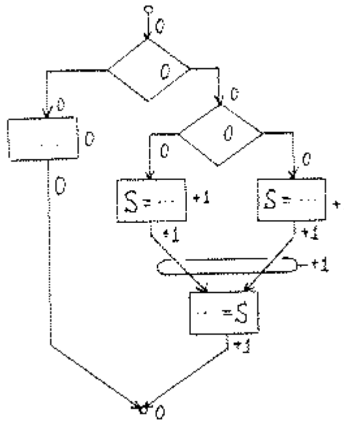

Fig. 7 An Example of the Data Flow Analysis

This algorithm is efficient in the sense that each vertex is traversed only once. In general this algorithm is quite useful for various data dependency of variables, such as variables under nested IF statements or partially defined variables.

Vectorization of IF statements can be further extended based on the above algorithm. For example, if semantics analysis is employed, some special cases can be vectorized as well. Such an example is shown in Fig. 8.

```
    DO 10 i=1,N
    IF(A(K).EQ.X) S=B(I)
    C(I)=S*D(I)
 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 technique 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(1)=0.0
     A(i)=0.0                DO 10 i=2,N
    ELSE             =>       A(i)=S*B(i)
     A(i)=S*B(i)          10 CONTINUE
    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 scalar and Add) instruction instead of VM(Vector Multiply) with scalar and VA(Vector Add) instructions.
(2) Parallel execution of vector instructions with their preparing instructions(Fig. 10).

Here TestVp instruction, which will wait the end of vector intructions, is moved out from the outer DO loop.

```
      DO 10 j=1,N
      A(1,j)=0.0
        DO 10 i=2,N
 10     A(i,j)=B(i,j)+C(i,j)
```

```
+---------->|
|           |
|    +-------------+
|    |Sequence of  |    +--------------+
|    | Preparing   |  / |Sequence of   |
|    |Instructions|/   |   Vector     |
|    +-------------+    |Instructions  |
|      EXecuteVP <----- +--------------+
|           |
<-----------+
         TestVP
           |
           v
```

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
```

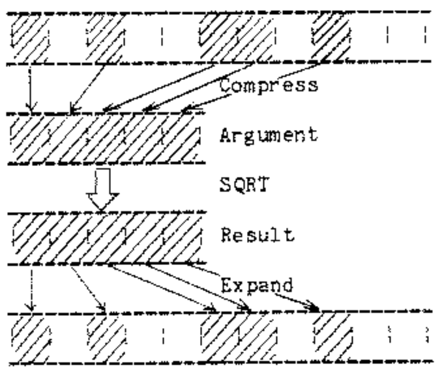

Fig. 11 A Compression of A Function Argument

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 IAP, a lot of techniques of enhancing vectorization are used for the S-810 compilers. With these enhancements, the vectorization ratio of typical large scientific FORTRAN programs has increased about 30%. And the performance ratio of S-810 vector mode to scalar mode is about 10-100. 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 bredth-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 Sakae Takahashi for his helpful comments during the preparation of this paper.

References

[1] R. Takanuki, Y. Umetani and I. Nakata, "Some Compiling Algorithms for an Array Processor", Proceedings of 3rd USA-JAPAN Computer Conference, (1978), pp.273-279.

[2] Y. Umetani, S. Kawabe, H. Horikoshi and T. Odaka, "An Analysis on Applicability of the Vector Operations to Scientific Programs and the Determination of an Effective Instruction Repertoire", ibid, (1978), pp.331-335.

[3] R. Takanuki and Y. Umetani, "Optimizing FORTRAN77", Hitachi Review, vol. 30,No. 5 , (1981).

[4] Y. Umetani and M. Yasumura, "A Vectorization Algorithm for Control Statements", Journal of

Information Processing, To be published.
[5] M. Yasumura, Y. Umetani and H. Horikoshi, "Partial Vectorization Method for Automatic Vector Compilers", Journal of Information Processing (in Japanese), Vol. 24, No. 1, (1983).
[6] D. Kuck, R. Kuhn, D. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimizations", Proceedings of the 8th ACM Symposium on Principles of Programming Languages, (1981), pp.207-218.
[7] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques", IEEE Transaction on Computer, Vol. C-29, (Sept. 1980), pp.763-776.
[8] D. J. Kuck, "Parallel Processing of Ordinary Programs", Advances in Computer, Academic Press, Vol. 15, (1976).
[9] A. V. Aho and J. D. Ullman, "Principles of Compiler Design", Addison-Wesley, (1977), 604p.
[10] D. B. Loveman, "Program Improvement by Source-to-Source Transformation", Journal of the ACM, Vol. 20, No. 1, (Jan. 1977).
[11] R. L. Sites, "An Analysis of the CRAY-1 Computer", Proceedings of the 5th Annual Symposium on Computer Architecture, (1978), pp.101-106.
[12] M. Yasumura, T. Matsunaga, Y. Tanaka, and Y. Umetani, "A Method of Control Structure Analysis for a Vector Compiler", Proceedings of National Conference of Information Processing Japan (in Japanese), (Oct. 1981), pp.211-212.
[13] Y. karaki, "Effective Use of the Supercomputer", Centre News, Computer Centre, University of Tokyo, Vol. 15, No. 9 10, (1983), pp.39-62.