Advanced Vectorization Techniques for Supercomputers

Shizuo Gotou*, Yoshikazu Tanaka*, Kyoko Iwasawa*, Yasusi Kanada* and Akio Aoyama**

A new FORTRAN 77/HAP compiler for Hitachi's supercomputers S-810 and S-820 has been implemented featuring new compiling techniques to enable users to easily obtain higher performance. The most important element of this compiler is an advanced global data flow analysis method which determines whether vectorization as well as optimization can be applied or not. Also important are powerful program transformation techniques for vectorization and optimization to vectorize more portions of the programs and produce more efficient code. The performance improvement of this new compiler is compared with the performance of the previous compiler. It is also pointed out that these method and techniques can be applicable to other supercomputers as well.

1. Introduction

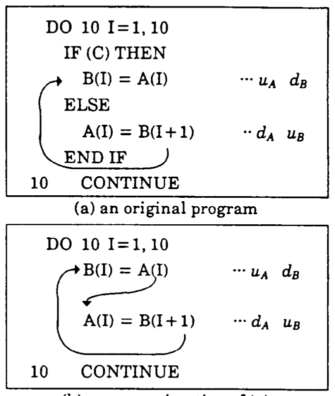
Supercomputers have been providing users in many scientific and engineering fields with large computational power. In this regard, many compiling techniques have been developed[3, 4, 6, 9, 11, 13, 14] to provide powerful automatic vectorizing functions necessary to obtain higher performance from supercomputers. This paper describes advanced automatic vectorization techniques implemented in the new version of the FORTRAN 77/HAP compiler for Hitachi's supercomputers S-810 and S-820.

The most important objective of automatic vectorizing compilers is to increase the vectorization ratio (the ratio of the vectorized portion to the entire program) and the vector acceleration ratio (the ratio of the scalar-mode execution time of the vectorized portion to the vector-mode execution time of the same portion).

In order to increase the vectorization ratio, it is desirable to analyze and investigate the original program to vectorize as many portions of the program as possible. Earlier compilers for supercomputers including our old version compiler could vectorize only the innermost loops, but recent compilers have been able to vectorize multiple nested loops whose structures are rather simple. It is now necessary to remove more restrictions and expand the vectorization to even more complicated control structures.

For example, Fig. 1 shows a weakness in conventional methods. Because the program in Fig. 1(a) has been treated as the program in Fig. 1(b) by the conventional methods, a data dependence relation for array A from use u_A to definition d_A has been detected, which does not exist in fact. Another data dependence relation for array B can really exist from use u_B to definition d_B. As these two data dependence relations construct a cycle, the program has been determined to be unvectorizable. A more detailed data flow analysis method is necessary to vectorize this program so that it proves that no data dependence relation for array A exists.

In order to increase the vector acceleration ratio, it is desirable to develop as many and more powerful vectorization techniques as possible. Fig. 2 shows an example of a vectorization technique called outer loop unrolling. The program of Fig. 2(a) can be transformed to the form in (b). The statement in the loop is copied with the array subscripts changed from J to $J+1$, where J is the loop index variable of the outer loop, and with the loop



```
   DO 10 I=1,10
     IF (C) THEN
       B(I) = A(I)          ··· u_A  d_B
     ELSE
       A(I) = B(I+1)        ·· d_A  u_B
     END IF
10   CONTINUE
```

(a) an original program

```
   DO 10 I=1,10
       B(I) = A(I)          ··· u_A  d_B

       A(I) = B(I+1)        ··· d_A  u_B

10   CONTINUE
```

(b) an approximation of (a)

Fig. 1 An example showing the weakness of the conventional data flow analysis method.

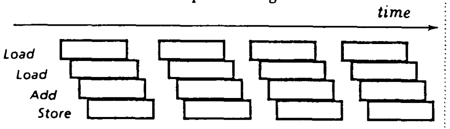
*Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo, 185, Japan.
**Software Works, Hitachi, Ltd., Yokohama, Kanagawa, 244, Japan.

```
      DO  10  J=1,100
      DO  10  K=1,100
         A(K,J)=B(K,J)+C(K,J)
10    CONTINUE
```

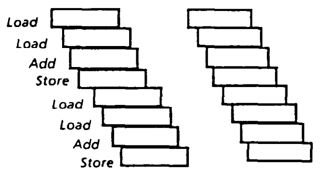
(a) an original program

```
      DO  10  J=1,100,2
      DO  10  K=1,100
         A(K,J)=B(K,J)+C(K,J)
         A(K,J+1)=B(K,J+1)+C(K,J+1)
10    CONTINUE
```

(b) result of outer loop unrolling



(c) execution process of (a)



(d) execution process of (b)

Fig. 2 An example showing the effectiveness of a technique to increase the vector acceleration ratio.

incremental value changed from 1 to 2. This program transformation does not affect the result of the program. Fig. 2(c) and (d) show the execution processes of (a) and (b) respectively. Many hardware resources possessed by supercomputers can be used more efficiently. It is also necessary to develop an analysis method which verifies the correctness of the transformation.

In summary, in order to attain higher performance it is necessary to develop a more detailed data flow analysis method and more vectorizing techniques.

In chapter 2, we will describe the method of global data flow analysis we have developed for the new version of the FORTRAN 77/HAP compiler. In chapter 3, we will describe the vectorization techniques using the results of global data flow analysis. Finally, in chapter 4, we will present a performance comparison of a world-wide used benchmark called the Livermore Loops and show the effectiveness of our method.

2. Global Data Flow Analysis Method

We have developed a new powerful global data flow analysis method that constructs an extended data dependence graph for arrays as well as for simple variables. The extended data dependence and the method of analysis originally reported in reference[5] are presented in this chapter.

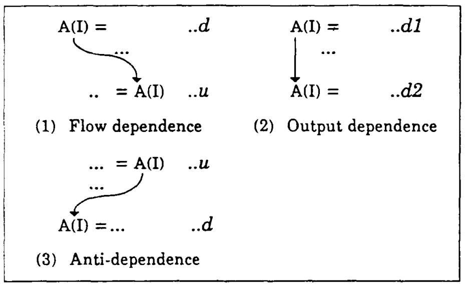


Fig. 3 Kinds of data dependences.

2.1 Data dependences

2.1.1 Kinds of data dependences

Data dependence analysis is required to determine whether portions of a program are vectorizable by nature and, if so, how they are transformed to a vectorizable form. There are three well-known kinds of data dependences[6]. Examples are shown in Fig. 3.

Def-1.1 Flow dependence

If a value of a variable or an array element defined at a definition *d* may be used at a use *u*, a flow dependence from *d* to *u* exists.

Def-1.2 Output dependence

If a value defined at a definition *d*1 may be redefined at a definition *d*2, an output dependence from *d*1 to *d*2 exists.

Def-1.3 Anti-dependence

If a value used at a use *u* may be redefined at a definition *d*, an anti-dependence from *u* to *d* exists.

These terms are derived from reference[6] but the definitions are slightly different. For example, when there are three definitions *d*1, *d*2 and *d*3 in order and there is no path from *d*1 to *d*3 without travelling through *d*2, output dependence from *d*1 to *d*3 exists by the definition of reference[6], but does not exist by our definition.

2.1.2 Classification of data dependence based on loop dependence

It is useful to classify data dependences based on their relations to loops (named loop dependences) in order to examine whether the loops can be vectorized or not. Each of the above data dependences can be classified as follows.

Def-2.1 *L*-loop independent data dependence

If a value at a reference (a use or a definition) or a variable or an array element in each of the *i*-th iteration of a loop *L* is referred to at another reference solely in the *i*-th iteration, the data dependence between the two references is called an *L*-loop independent data dependence.

Def-2.2 *L*-loop carried data dependence

If a value at a reference in each of the *i*-th iteration of a loop *L* is referred to at another reference in some *j*-th

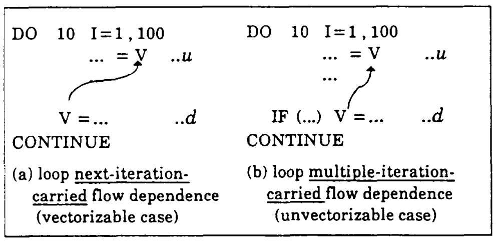

DO 10 I=1,100
... = V ..u
V = ...d
CONTINUE
(a) loop next-iteration-carried flow dependence (vectorizable case)
DO 10 I=1,100
... = V ..u
...
IF (...) V = ...d
CONTINUE
(b) loop multiple-iteration-carried flow dependence (unvectorizable case)

Fig. 4 Examples of loop dependence.

iteration ($j>i$), the data dependence is called an L-loop carried data dependence.

These definitions are derived from reference[3]. We have enhanced it so that L-loop carried data dependences could be classified further as follows.

Def-2.2.1 L-loop next-iteration-carried data dependence

If $j=i+1$ in Def-2.2, the data dependence is called an L-loop next-iteration-carried data dependence.

This kind of data dependence is used to facilitate vectorization using the macro vector instructions as presented in section 3.3.

Def-2.2.2 L-loop multiple-iteration-carried data dependence

If $j>i+1$ in Def-2.2, the data dependence is called an L-loop multiple-iteration-carried data dependence.

Similarly, the L-loop multiple-iteration-carried data dependence could be classified such as 3rd-iteration-carried data dependence, 4th-iteration-carried data dependences, and so on. However, since they are not very useful in optimization of usual programs for numerical calculations, it is convenient to distinguish only three kinds of loop dependences (loop independent data dependence, loop next-iteration-carried data dependence and loop multiple-iteration-carried data dependence).

2.1.3 Examples of loop dependence

Loop dependence analysis is useful for vectorization and optimization. Fig. 4 shows examples of loop dependences on variables. It is noticed that the analysis on variables is as important as on arrays. In Fig. 4(a), a value defined at d is surely used at u in the next iteration, so that the data dependence from d to u is a loop next-iteration-carried flow dependence. If data dependences on the other variables and array elements (not shown in the figure) are suitable, the loop is determined to be vectorizable. But for the loop in Fig. 4(b) it is unknown how many times subsequently the value of variable V defined at d will be used as u, so that the data dependence from d to u is a loop multiple-iteration-carried flow dependence. Because there is no vector function to process this case, the loop is determined to be unvectorizable. The vectorizability is thus determined by the loop dependence analysis.

2.2 Global data flow analysis method

Our global data flow analysis method determines existing data dependence relations by scanning the statements in the program. It is processed in combination with the array subscript comparison and it constructs a data dependence graph from the data dependence relations. We explain at first the array subscript comparison, followed by two kinds of sets called the "reaching definition" and "exposed use" used in the global data flow analysis, and finally the process of the analysis method itself.

2.2.1 Array subscript comparison

Every pair of two array references $r1$ and $r2$ in a loop L is analyzed and determined whether relations defined as follows exist or not.

Def-3.1 L-loop n-time complete coincidence (n: a non-negative integer)

If, for all i, an array element at an array reference $r1$ referred to in the i-th iteration of loop L is necessarily referred to at another array reference $r2$ in the $(i+n)$-th iteration, it is considered that an L-loop n-time complete coincidence exists between $r1$ and $r2$.

Def-3.2 L-loop n-time possible coincidence

If, for some i, an array element at an array reference $r1$ referred to in the i-th iteration of loop L is possibly referred to at another array reference $r2$ in the $(i+n)$-th iteration, it is considered that an L-loop n-time possible coincidence exists between $r1$ and $r2$.

Def-3.3 L-loop n-or-more times possible coincidence

If, for some i, an array element at an array reference $r1$ referred to in the i-th iteration of loop L is possibly referred to at another array reference $r2$ in some j-th iteration ($j \geqq i+n$), it is considered that an L-loop n-or-more-times possible coincidence exists between $r1$ and $r2$.

2.2.2 Reaching definition and exposed use

For each point (such as between statements), the following two sets are defined.

Def-4.1 Reaching definition

Reaching definition at a point P is the set of definitions whose defined values may be used at point P. This definition is the same as in reference[1].

Def-4.2 Exposed use

Exposed use at a point P is the set of uses whose used values may be reused at point P. This definition is newly given by the authors.

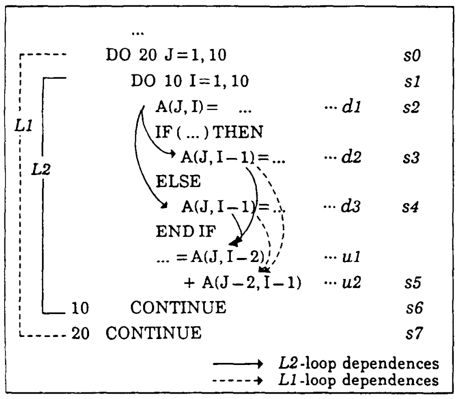
While reaching definitions are used to determine flow dependences and output dependences, exposed uses are used to determine anti-dependences.

Def-5.1 Reaching Definition In: RIN

RIN(s) is the set of reaching definitions at the point immediately preceding statement s.

Def-5.2 Reaching Definition Out: ROUT

ROUT(s) is the set of reaching definitions at the point immediately succeeding statement s.



```
      ...
      DO 20 J=1,10                        s0
        DO 10 I=1,10                      s1
          A(J,I)=  ...            ...d1   s2
          IF(...)THEN
            A(J,I-1)=...          ...d2   s3
          ELSE
            A(J,I-1)=...          ...d3   s4
          END IF
          ... =A(J,I-2)           ...u1
             + A(J-2,I-1)         ...u2   s5
 10     CONTINUE                          s6
 20   CONTINUE                            s7
```

Fig. 5 A program example.

These sets are calculated three times for a loop in the analyzing process as shown in the next paragraph, and are distinguished as follows.

(1) RIN and ROUT used for analyzing loop independent data dependences are written as RIN_i and ROUT_i.

(2) RIN and ROUT used for analyzing loop next-iteration-carried data dependences are written as RIN_n and ROUT_n.

(3) RIN and ROUT used for analyzing loop multiple-iteration-carried data dependences are written as RIN_m and ROUT_m.

To calculate these reaching definitions, the following two sets are introduced.

Def-5.3 RGEN

RGEN(s) is the set of definitions at statement s.

Def-5.4 RKILL

RKILL(s) is the set of definitions whose defined values are surely redefined at statement s.

To calculate the set RKILL(s), the array subscript comparison is used. If there is n-time complete coincidence between a definition d which reaches statement s and the definition in statement s, definition d belongs to the set RKILL(s), where $n=0$ for the loop independent analysis, $n=1$ for the loop next-iteration-carried analysis and $n=2$ for the loop multiple-iteration-carried analysis.

RIN and ROUT are calculated so that the following data flow equations are satisfied.

$$\mathrm{RIN}(s) = \bigcup_{p \in \mathrm{Pred}(s)} \mathrm{ROUT}(p),$$

where Pred(s) is the set of preceding statements of statement s.

$$\mathrm{ROUT}(s) = (\mathrm{RIN}(s) - \mathrm{RKILL}(s)) \cup \mathrm{RGEN}(s)$$

Similarly, the exposed uses are distinguished and calculated.

2.2.3 Analyzing process

Fig. 6 shows a process of the global data flow analysis method where the analysis type is indicated for each process step.

The loops are thus scanned three times. In each step, data dependence analysis is performed as follows.

(a) Calculation of the reaching definitions

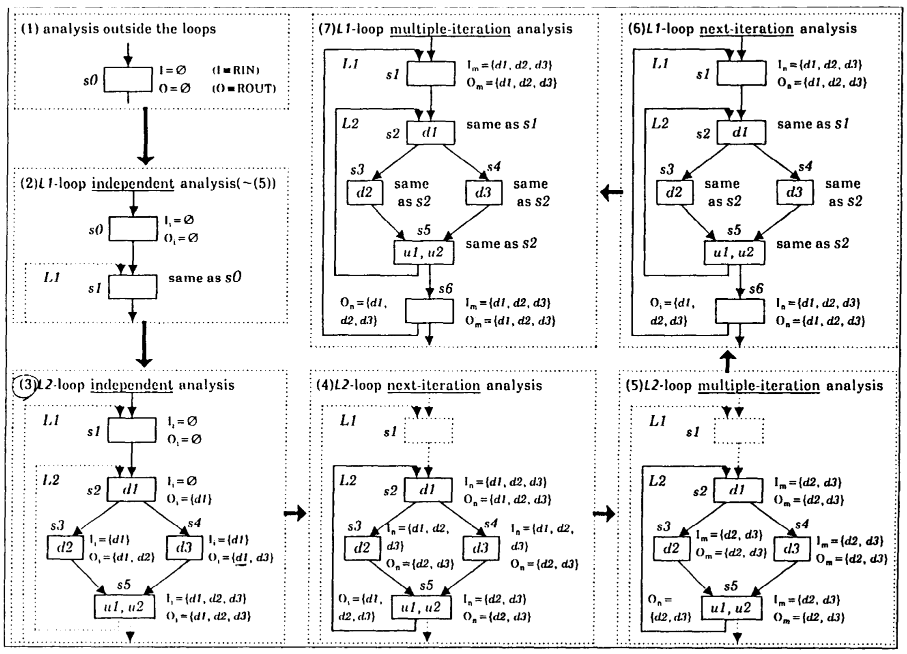


Fig. 6 Analyzing process of the program in Fig. 5.

As shown in the above paragraph, the sets of RGEN and RKILL are obtained for each statement in the loop and the sets of RIN and ROUT are calculated to satisfy the data flow equations presented in the same paragraph.

(b) Calculation of the exposed uses

The exposed uses are calculated in a similar way.

(c) Detection of data dependences

We explain here the case of flow dependences (others are obtained similarly). For each statement s having a use u, every definition d belonging to RIN(s) is analyzed individually as follows.

(i) In a process of loop independent analysis, if there is a 0-time possible coincidence between d and u, a loop independent flow dependence exists.

(ii) In a process of loop next-iteration-carried analysis, if there is a 1-time possible coincidence between d and u, a loop next-iteration-carried flow dependence exists.

(iii) In a process of loop multiple-iteration-carried analysis, if there is a 2-or-more-times possible coincidence between d and u, a loop multiple-iteration-carried flow dependence exists.

We explain steps (3) and (4) in detail.

Step (3), that is, $L2$-loop independent analysis scans all statements in loop $L2$. Three array element definitions $d1$, $d2$ and $d3$ appear in the loop and the subscript comparisons for pairs ($d1$, $d2$) and ($d1$, $d3$) are made. As there are no 0-time complete coincidences between them and $d2$ does not reach $d3$ and vice versa, RKILL(s)$=\emptyset$ for all s. RIN_i and ROUT_i are then calculated as shown in Fig. 6(3).

Next, data dependences are calculated as follows. As far as flow dependences are concerned, for each of two uses $u1$ and $u2$, subscript comparisons are made between every pair of three definitions $d1$, $d2$ or $d3$ in $\mathrm{RIN}_i(s)$ and $u1$ or $u2$. As a result of the subscript comparison, it is proved that no $L2$-loop 0-time possible coincidence exists for any pair. So, no loop-independent flow dependences are detected. Other kinds of loop independent data dependences are examined similarly. As a result, neither output nor anti-dependences are detected.

Step (4), that is, $L2$-loop next-iteration-carried analysis scans the same range of step (3). When scanning statement $s2$, $\mathrm{RIN}_n(s2)$ is initialized to be equal to $\mathrm{ROUT}_i(s5)$ and the subscript comparisons of definition $d1$ with definitions $d1$, $d2$ and $d3$ in $\mathrm{RIN}_n(s2)$ are made. As a result, the comparisons show that no $L2$-loop 1-time complete coincidence exists. Thus RKILL($s2$)$=\emptyset$. Similarly, statements $s3$, $s4$ and $s5$ are scanned and analyzed. The analysis proves that there are two $L2$-loop 1-time complete coincidences: one between $d1$ and $d2$ and the other between $d1$ and $d3$, and hence RKILL($s3$)$=$RKILL($s4$)$=\{d1\}$, RKILL($s5$)$=\emptyset$. Next, RIN_n and ROUT_n are calculated as shown in Fig. 6(4).

Detection of data dependences is performed as follows. For statement $s3$, subscript comparisons are performed between definition $d3$ and definitions $d1$, $d2$ and $d3$ in $\mathrm{RIN}_n(s3)$; it is proved that there is an $L2$-loop 1-time possible coincidence between $d1$ and $d3$ and hence there exists an $L2$-loop next-iteration-carried output dependence from $d1$ to $d2$. Analysis on statement $s4$ proves that there exists another $L2$-loop next-iteration-carried output dependence. Similarly, analysis on statement $s5$ proves the existence of two $L2$-loop next-iteration-carried flow dependences: from $d2$ to $u1$ and from $d3$ to $u1$.

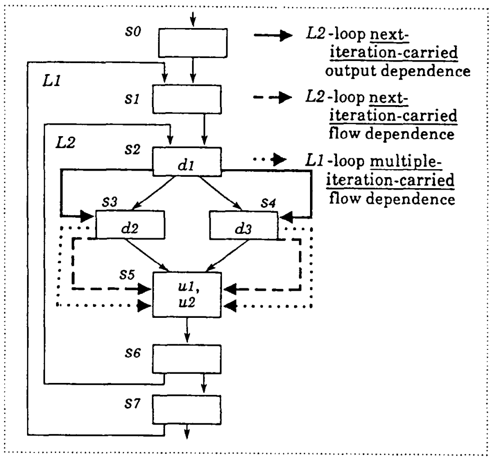

s0
s1
s2
s3
s4
s5
s6
s7
L1
L2
d1
d2
d3
u1, u2
L2-loop next-iteration-carried output dependence
L2-loop next-iteration-carried flow dependence
L1-loop multiple-iteration-carried flow dependence

Fig. 7 Flowgraph and the data dependence graph of the program in Fig. 5.

Fig. 7 shows the data dependence graph constructed by the analysis on the program in Fig. 5. The following results are obtained.

(1) $L2$ loop next-iteration-carried output dependences ($\longrightarrow$) from $d1$ to $d2$ and $d3$ exist.

(2) $L2$ loop next-iteration-carried flow dependences ($--\rightarrow$) from $d2$ and $d3$ to $u1$ exist.

(3) $L1$ loop multiple-iteration-carried flow dependencies ($\cdots\rightarrow$) from $d2$ and $d3$ to $u2$ exist.

(4) No other data dependence exists.

2.2.4 Significant features of the data flow analysis method

The significant features of our global data flow analysis method are summarized as follows.

(a) Multiply nested loops can be uniformly analyzed by travelling from outer loops to inner loops. The subscript comparison method we have developed absorbs the differences by loop nest level by taking the ranges of the loop index variables into consideration.

(b) Data dependences can be classified such as loop independent, loop next-iteration-carried and loop multiple-iteration-carried owing to the subscript comparison facility that determines the n-time complete or possible coincidences. As a result, advanced vectorization using the macro vector instructions was realized as will be explained in section 3.3.

(c) Loops with complicated control structures can be analyzed by taking control structures into consideration in formulating data flow equations for RIN, ROUT, etc.

3. Vectorization Techniques

The old version of the FORTRAN 77/HAP compiler can vectorize only the innermost loops. The new one can vectorize multiply nested loops with more complicated structures by using the results of the global data flow analysis presented in chapter 2; thus more portions of programs can be processed effectively in vector processing mode. The vectorization techniques newly developed are presented in this chapter.

3.1 Multiply nested loop vectorization techniques

Four kinds of techniques have been developed. They are outer loop splitting, loop interchange, loop collapse and outer loop unrolling. Their applicabilities are checked uniformly by using the results of the global data flow analysis.

```
   DO 10 J=K+1,N
     A(K,J)=W                         splitting line
  ========================================
  :DO 20 I=K+1,N                         :
  :  A(I,J)=A(I,J)+A(I,K)*A(K,J)         : (A)
20:CONTINUE                              :
10 CONTINUE
```

(a) an original loop

```
  :DO 10 J=K+1,N                         :
  :  A(K,J)=W                            : (B)
10:CONTINUE                              :
   DO 11 J=K+1,N
  :DO 20 I=K+1,N                         :
  :  A(I,J)=A(I,J)+A(I,K)*A(K,J)         : (A)
20:CONTINUE                              :
11 CONTINUE
```

(b) result of outer loop splitting

[- - -] : parts to be vectorized

Fig. 8 An example of outer loop splitting.

3.1.1 Outer loop splitting

Vector processing can be basically applied only to the innermost loops. In an example of Fig. 8(a), only part (A) can be vectorized. But if the original loop in Fig. 8(a) can be transformed to the form in (b), both of parts (A) and (B) can be vectorized. This transformation is called outer loop splitting. The condition for data flow relations to permit this transformation is that there are no cycles of data dependence relations related to this outer loop. The noticeable features of our outer loop splitting method are as follows.

(a) Even when part (A) in Fig. 8(a) is not vectorizable because of the existence of some data dependence, outer loop splitting can be made and part (B) in Fig. 8(b) is checked if it can be vectorized.

(b) The applicabilities of all other vectorizing techniques (such as statement reordering, inner loop splitting, loop interchange, loop collapse, outer loop unrolling and so on) are examined for the new loops which are transformed to the innermost loops or tightly nested loops by outer loop splitting. Examples will be shown later.

3.1.2 Loop interchange

For tightly nested loops, an outer loop may be exchanged with the innermost loop if permitted. This is called loop interchange, and the new innermost loop may be vectorized. Fig. 9 shows an example of loop interchange.

Loop interchange is applied in either of the following cases.

(a) The innermost loop has a data dependence which prevents vectorization and one of the outer loops has no such data dependence. The sufficient applicability condition of data flow relations is that some outer loop (L) has no L-loop carried data dependences.

```
   DO 10 J=K+1,N
     DO 20 I=K+1,N
       A(I,J)=A(I-1,J)+A(I-2,J)
20   CONTINUE
10 CONTINUE
```

(a) an original loop

```
   DO 20 I=K+1,N
  :DO 10 J=K+1,N                         :
  :  A(I,J)=A(I-1,J)+A(I-2,J)            :
10:CONTINUE                              :
20 CONTINUE
```

(b) result of loop interchange

[- - -] : parts to be vectorized

Fig. 9 An example of loop interchange.

(b) The loop length of the vectorizable innermost loop is too short to be efficiently vectorized and one of the outer loops which is also vectorizable when exchanged has a longer loop length.

Vectorization ratio is increased in the former case, and the vector acceleration ratio is increased in the latter case.

3.1.3 Loop collapse

Tightly nested loops may be transformed to the form of a collapsed loop if permitted in order to increase the effect of vectorization. There are the following two cases.

(a) Continuous array reference case

This case has been well-known. If all of the arrays in a tightly nested loop are continuously referred to, the loop can be collapsed to one loop by use of alias arrays. Fig. 10 shows an example of this case.

```
      REAL  A(10,10)
      DO  10  J=1,5
        DO  20  K=1,10
          A(K,J)=1.0
 20     CONTINUE
 10   CONTINUE
```

(a) an original loop

```
      REAL  A(10,10), ￥A(50)
      EQUIVALENCE  (A(1,1), ￥A(1))
      DO  10  JK=1,50
        ￥A(JK)=1.0
 10   CONTINUE
```

(b) result of loop collapse

Fig. 10 An example of loop collapse (continuous reference case).

```
      REAL  A(10,10)
      DO  10  J=1,3
        DO  20  K=1,J
          A(K,J)=K
 20     CONTINUE
 10   CONTINUE
```

(a) an original loop

```
      REAL  A(10,10)
      INTEGER  ￥K(6)/1,1,2,1,2,3/
      INTEGER  ￥J(6)/1,2,2,3,3,3/
      DO  15  L=1,6
        A(￥K(L), ￥J(L))=￥K(L)
 15   CONTINUE
```

(b) result of loop collapse

Fig. 11 Another example of loop collapse by using indirect vector addressing (non-continuous reference case).

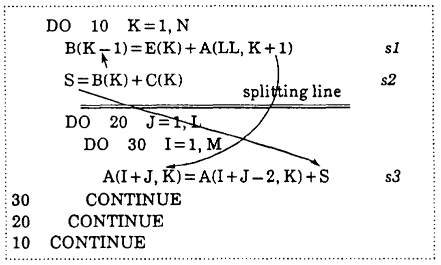
(b) Non-continuous array reference case

Even when some arrays in a tightly nested loop are not continuously referred to, the loop can be collapsed to one loop if the array references can be changed to indirect addressing references, such that references can be determined by a loop index variable newly introduced by the compiler (Tsuda [12]). Fig. 11 shows an example of this case. We enhanced this technique to be applicable even when conditional statements exist.

3.1.4 Outer loop unrolling

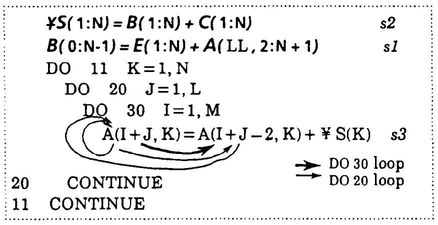
The Hitachi's supercomputers have multiple load-pipelines, adder-pipelines and multiplier-pipelines. To use these pipelines effectively, an innermost vectorizable loop which uses only a small amount of these pipelines may be unrolled (if permitted) such that the body of the innermost loop is expanded twice, 4 times or 8 times and the loop index variable of the outer loop is incremented by 2, 4 or 8 times of the initial incremental value respectively (most suitable number of times is automatically determined by the compiler).

The condition of data flow relations to apply outer loop unrolling is the same as loop interchange. This



```
      DO  10  K=1,N
        B(K-1)=E(K)+A(LL,K+1)                s1
        S=B(K)+C(K)                          s2
                            splitting line
        DO  20  J=1,L
          DO  30  I=1,M
            A(I+J,K)=A(I+J-2,K)+S            s3
 30       CONTINUE
 20     CONTINUE
 10   CONTINUE
```

(a) an original program



```
      ￥S(1:N)=B(1:N)+C(1:N)                  s2
      B(0:N-1)=E(1:N)+A(LL,2:N+1)            s1
      DO  11  K=1,N
        DO  20  J=1,L
          DO  30  I=1,M
            A(I+J,K)=A(I+J-2,K)+￥S(K)    s3
                                   DO 30 loop
                                   DO 20 loop
 20     CONTINUE
 11   CONTINUE
```

(b) result of outer loop splitting

```
      ￥S(1:N)=B(1:N)+C(1:N)
      B(0:N-1)=E(1:N)+A(LL,2:N+1)
      DO  20  J=1,L
        DO  30  I=1,M
          A(I+J,1:N)=A(I+J-2,1:N)+￥S(1:N)
 30     CONTINUE
 20   CONTINUE
```

(c) result of loop splitting and loop interchange

Fig. 12 An example of multiply nested loop vectorization.

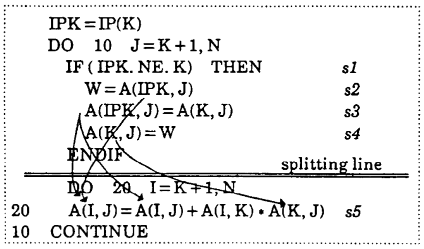
technique is also applicable to a non-tightly nested loop. We have already shown an example and effectiveness of this technique in Fig. 2.

3.2 Examples of multiply nested loop vectorization

Two examples of multiply nested loop vectorization processes which succesively use the techniques presented above are described. Fig. 12 shows an example where two techniques for multiply nested loop vectorization, outer loop splitting and loop interchange, are applied. The old version could not vectorize this program.

In Fig. 12(a), there are two data dependences related to the DO 10 loop for array A and variable S across the splitting line. But they do not form a cycle of data dependences and thus satisfy the applicable condition. Therefore outer loop splitting can be done. The data dependence for array B is next-iteration-carried anti-dependence of the DO 10 loop but it is not cyclic; thus statements $s1$ and $s2$ can be vectorized by interchanging them. This is a well-known technique called statement reordering[9]. In addition variable S is changed to an array ￥S. This is the so-called scalar expansion technique[13].

The result of outer loop splitting is shown in Fig. 12(b), where there are four data dependences, one for DO 30 and three for DO 20. The DO 30 loop cannot be vectorized because statement $s3$ has a recurrence which



```
    IPK=IP(K)
    DO  10  J=K+1,N
      IF(IPK.NE.K)  THEN                 s1
        W=A(IPK,J)                       s2
        A(IPK,J)=A(K,J)                  s3
        A(K,J)=W                         s4
    ENDIF
                                splitting line
    DO  20  I=K+1,N
20    A(I,J)=A(I,J)+A(I,K)*A(K,J)        s5
10  CONTINUE
```

(a) part of LU decomposition for a dense matrix

```
    IPK=IP(K)
    IF(IPK.NE.K)  THEN
      W=A(IPK,K+1:N)
      A(IPK,K+1:N)=A(K,K+1:N)
      A(K,K+1:N)=W
    ENDIF
    DO  11  J=K+1,N,2
      DO  20  I=K+1,N
      A(I,J)=A(I,J)+A(I,K)*A(K,J)
20    A(I,J+1)=A(I,J+1)+A(I,K)*A(K,J+1)
11  CONTINUE
```

(b) transformed form of outer loop unrolling

```
    IPK=IP(K)
    IF(IPK.NE.K)  THEN
      W=A(IPK,K+1:N)
      A(IPK,K+1:N)=A(K,K+1:N)
      A(K,K+1:N)=W
    ENDIF
    DO  11  J=K+1,N,2
      A(K+1:N,J)=A(K+1:N,J)
               +A(K+1:N,K)*A(K,J)
      A(K+1:N,J+1)=A(K+1:N,J+1)
               +A(K+1:N,K)*A(K,J+1)
11  CONTINUE
```

(c) resultant vectorized form

Fig. 13 Another example of multiply nested loop vectorization.

```
   DO  10  J=1,N
10 X(J)=A(J)*X(J-1)+B(J)
```

(a) First order iteration

```
   DO  20  J=1,N
20 X(J)=X(J-1)+B(J)
```

(b) Partial summation

```
   DO  30  J=1,N
30 S=S+X(J)
```

(c) Summation

```
   DO  40  J=1,N
40 S=S+X(J)*Y(J)
```

(d) Inner product

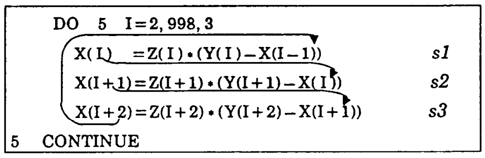
```
   DO  50  J=1,N
     IF(X(J).GT.P)THEN
       I=J
       P=X(J)
     ENDIF
50 CONTINUE
```

(e) Maximum value search (a sample pattern)

```
   DO  60  J=1,N
     IF(Y(I).GT.Y(J))THEN
       P=Y(J)
       I=J
     ENDIF
60 CONTINUE
```

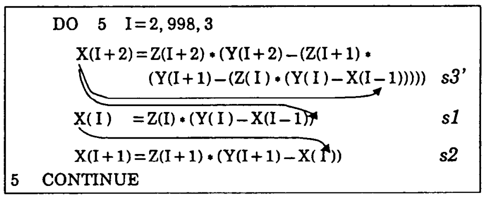
(f) Minimum value search (a sample pattern)

Fig. 14 Functions of macro vector instructions.



```
   DO  5  I=2,998,3
     X(I)  =Z(I)*(Y(I)-X(I-1))                s1
     X(I+1)=Z(I+1)*(Y(I+1)-X(I))              s2
     X(I+2)=Z(I+2)*(Y(I+2)-X(I+1))            s3
5  CONTINUE
```

(a) a source program (Livermore Loop No. 5)



```
   DO  5  I=2,998,3
     X(I+2)=Z(I+2)*(Y(I+2)-(Z(I+1)*
               (Y(I+1)-(Z(I)*(Y(I)-X(I-1)))))) s3'
     X(I)  =Z(I)*(Y(I)-X(I-1))                s1
     X(I+1)=Z(I+1)*(Y(I+1)-X(I))              s2
5  CONTINUE
```

(b) result of substitution of X(I), X(I+1) and reordering

Fig. 15 Inter-statement recurrence vectorization.

is not applicable to the macro vector instruction.

Loop interchange between DO 20 and DO 30 is not applicable because DO 20 has unsuitable data dependences, that is, the three data dependences for DO 20 are loop carried data dependences. The DO 11 loop has no carried data dependences; therefore loop interchange between DO 11 and DO 30 can be applied. The resultant vectorized form is shown in (c).

Fig. 13 shows another example. The program in Fig. 13(a) is a part of an LU decomposition program of a dense matrix. All pairs of array references are analyzed and no unsuitable data dependences are found in the program. The analysis answers that there are some data dependences across the splitting line. These data dependences appear because the value of *IPK* is not known at compile time. Between *A*(*IPK*, *J*) and *A*(*I*, *K*) or *A*(*K*, *J*), there are no dependences because the value range of the loop index variable *J* is from *K*+1 to *N* and the relation *IPK* not equal to *K* exists. Also there are no dependences between *A*(*K*, *J*) and *A*(*I*, *J*) or *A*(*I*, *K*) because *K* is less than *I*.

As a result of these analyses, loop splitting can be made applicable. In the new tightly nested loops (DO 11 and DO 20), there are no carried data dependences for the DO 11 loop and the condition for outer loop unrolling is satisfied. Fig. 13(c) shows the vectorized form of this program.

3.3 Vectorization technique using macro vector instructions

3.3.1 Macro vector instructions

There are some programs which have so-called strongly connected parts where data dependence relations constitute a cycle. These parts are not usually vectorized, but some of them can be vectorized by using macro vector instructions which some supercomputers including Hitachi's are equipped with. For example, reduction operation, first order linear iteration and maximum or minimum value searching are very important because they often appear in numerical calculation programs. Hitachi's supercomputers S-810 and S-820 have the following six kinds of macro vector instructions.

(1) First order iteration
(2) Partial summation
(3) Summation
(4) Inner product
(5) Maximum value searching
(6) Minimum value searching

Fig. 14(a) ~ (f) show the functions of these six instructions in FORTRAN form.

```
 DO  5  I=2, 998, 3
   TEMP1=Z(I+2) * Z(I+1) * Z(I)
   TEMP2=Z(I+2) * Y(I+2)
            -Z(I+2) * Z(I+1) * Y(I+1)
            +Z(I+2) * Z(I+1) * Z(I) * Y(I)
  [X(I+2)=TEMP1 *  X(I-1)+TEMP2 ](iteration)
   X(I)=Z(I) * (Y(I)-X(I-1))
5  X(I+1)=Z(I+1) * (Y(I+1)-X(I))
```

(c) resultant vectorizable loop form

```
TEMP1 = Z(4:1000:3) * Z(3:999:3) * Z(2:998:3)
TEMP2 = Z(4:1000:3) * Y(4:1000:3)
        - Z(4:1000:3) * Z(3:999:3) * Y(3:999:3)
        + Z(4:1000:3) * Z(3:999:3)
        * Z(2:998:3) * Y(2:998:3)
X(4:1000:3) = TEMP1 * X(1:997:3) + TEMP2
X(2:998:3) = Z(2:998:3) * (Y(2:998:3) - X(1:997:3))
X(3:999:3) = Z(3:999:3) * (Y(3:999:3) - X(2:998:3))
```

(d) resultant vector instruction form

Fig. 16 Result of inter-statement reccurrence vectorization.

Table 1 Performance Comparison of the Livermore Loops.

Hardware Compiler Year/month	S810-20 V02-30(old) 1986/3	S810-20 V20-1A(new) 1987/6
No. 1	156.3	304.8
No. 2	259.1	276.7
No. 3	216.9	341.6
No. 4	66.7	66.8
No. 5	2.4	23.6
No. 6	3.2	26.2
No. 7	290.9	288.9
No. 8	8.8	116.8
No. 9	263.2	271.1
No. 10	67.1	65.6
No. 11	9.9	16.8
No. 12	110.8	113.2
No. 13	5.0	6.1
No. 14	8.5	12.5
Average	104.9	137.9
Harm. mean	10.6	31.2

(in MFLOPS)

3.3.2 Vectorization technique of inter-statement recurrence

We have developed a new method to apply the first order iteration instruction to program parts which have an inter-statement recurrence, that is, a recurrence between multiple statements. Fig. 15 shows an example. This is the No. 5 loop of the Livermore benchmark. Two loop independent flow dependences exist from *s*1 to *s*2 and from *s*2 to *s*3, and a loop next-iteration-carried flow dependence exists from *s*3 to *s*1. These three data dependences constitute an inter-statement recurrence.

Vectorization is processed as follows. First, the right part of *s*1 is substituted into statement *s*2 which includes the destination of flow dependence relation from *s*1, and the right part of *s*2 is succesively substituted into statement *s*3. Next, *s*1, *s*2 and *s*3 are reordered so that except for the data dependence in statement *s*3 there are no data dependences which prevent vectorization. This transformation dose not affect the result of the program. Statement *s*3 now satisfies the condition of applicability of a first order iteration vector instruction. Fig. 16 shows the resultant vectorized form.

The effectiveness of this inter-statement recurrence vectorization will be shown in the next chapter. Livermore Loop No. 6 is also vectorized in a similar way.

4. Performance for the Livermore Loops

The Livermore Loops have been widely used for a benchmark job to measure the performance of many supercomputers. It has been noticed that for this type of measurement the vectorization and optimization capability of the FORTRAN compiler are important as well as the hardware speed[10].

Table 1 gives the speed in MFLOPS (million floating operations per second) for the Livermore Loops. The performance data for the old version compiler on Hitachi S-810 model 20 are taken from reference[10]. The speed of the new version compiler on the same hardware model was measured by the authors.

Comparison of the data for the old version and the new one reveals the following points.

(1) Average performance was increased by 1.31 times from 104.9 to 137.9 MFLOPS. This increase was obtained mainly by the speed up of loops No. 1 and No. 3. In these loops, inner loop unrolling technique was effective.

(2) The harmonic mean was greatly improved from 10.6 to 31.2 MFLOPS. This improvement was obtained by the vectorization of loops No. 5, No. 6 and No. 8. Loops No. 5 and No. 6 became vectorized by using the inter-statement recurrence vectorization technique shown in chapter 3. Loop No. 8 was vectorized by the advanced global data flow analysis shown in chapter 2.

5. Conclusions

We have described the global data flow analysis method and the vectorization techniques based upon it which are implemented in the new version of the FORTRAN 77/HAP compiler for Hitachi's supercomputers S-810 and S-820. Many of these method and techniques are also applicable to other supercomputers.

The effectiveness of the method and techniques has also been evaluated. A performance comparison of the old and new compilers using the Livermore Loops benchmark job on S-810 model 20 has shown a 1.31 increased performance on the same hardware. This result shows that the vectorizing and optimizing ability of the compiler is important as well as the hardware speed.

Supercomputers with a large amount of processors have recently been appearing. It is also necessary to provide users with an easy-to-use programming environment for these new types. Compilers for these multiple processor systems will be required to have the ability to transform programs automatically to forms that multiple processor systems can execute efficiently. In particular, parallelism detection in the program, detection of obstructions which prevents parallel processing, debugging and tuning facilities will be needed by these compilers.

We believe that the global data flow analysis method and some of the vectorization techniques presented here will become the basis for the future software systems utilized by these types of supercomputers.

References

1. Aho, A. V., Sethi, R. and Ullman, J. D. Compilers—Principles, Techniques, and Tools, Addison Wesley, Massachusetts (1986).
2. Allen, J. R., Kennedy, K., Porterfield, C. and Warren, J. Conversion of Control Dependence to Data Dependence, *10-th Annual ACM symposium on Principles of Prog. Lang.* (Jan. 1983), 177-189.
3. Allen, J. R. and Kennedy, K. Automatic Loop Interchange, *ACM SIGPLAN '84 Symposium on Compiler Constructions, SIGPLAN Notices*, **19**, 6 (1984).
4. Banerjee, U. Speed up of Ordinary Programs, Ph. D. Thesis, *Univ. of Illinois at Urbana Champaign, Dept. of Comput. Sci.* (1979).
5. Kanada, Y., Ishida, K. and Nunohiro, E. A Method of Global Dataflow Analysis for Arrays, *Trans. of IPS Japan*, **28**, 6 (Jun. 1987), 567-576 (in Japanese).
6. Kuck, D. J., Kuhn, R. H., Padua, D. H., Leasure, B. and Wolfe, M. Dependence Graphs and Compiler Optimizations, *8th Annual ACM Symposium on Principles of Prog. Lang.* (Jan. 1981), 207-218.
7. Loveman, D. B. Program Improvement by Source-to-Source Transformation, *J. ACM*, **24**, 1 (Jan. 1977), 121-145.
8. Nagashima, S., Inagami, Y., Odaka, T. and Kawabe, S. Design Consideration for a High-speed Vector Processor-The Hitachi S-810, *IEEE International Conference on Computer Design; VLSI in computers. ICCD 84, IEEE Press*, New York (1983), 238-243.
9. Padua, D. A. and Wolfe, M. J. Advanced Computer Optimizations for Supercomputers, *Comm. ACM*, **29**, 12 (Dec. 1986), 1184-1201.
10. Steen, A. Van Der. Results on the Livermore Loops on some new supercomputers, *SUPERCOMPUTER*, 12 (Mar. 1986), 13-14.
11. Takanuki, R., Umetani, Y. and Nakata, I. Some Compiling Algorithms for an Array Processor, *3rd USA-Japan Computer Conference* (1978), 273-279.
12. Tsuda, T. and Kunieda Y. Mechanical Vectorization of Multiple Nested DO Loops by Vector Indirect Addressing, *IFIP 10th World Computer Congress* (Sep. 1986), 785-790.
13. Wolfe, M. J. Optimizing Supercompilers for Supercomputers, Ph. D. Thesis, *Univ. of Illinois at Urbana Champaign, Dept. of Comput. Sci.* (1982).
14. Yasumura, M., Tanaka, Y., Kanada, Y. and Aoyama, A. Compiling Algorithms and Techniques for the S-810 Vector Processor, *the 1984 International Conference on Parallel Processing, IEEE Press*, New York (Aug. 1984), 285-290.

(Received August 12, 1987)