On Techniques in Vectorizing Compilers and Optimizing Program Transformations for Supercomputers

MASAAKI SHIMASAKI*

Techniques in vectorizing compilers and optimizing program transformations for supercomputers are described. First we give a brief overview of programming languages for supercomputers, especially a historical review of languages for early vector processors in Japan in relation to languages for supercomputers in the present day. Then we give conditions for vectorization of basic loop programs. The order of vector instructions does not correspond, in general, to that of source statements in the program in order to attain high vectorization ratio. This can be considered as the reordering of statements in the source program. For dedugging a program, however, there are cases where it is convenient to force a compiler to keep the order of instructions with that of source statements, even if vectorization ratio is affected to some extent. We give conditions for vectorization with/without reordering of statements, in a easily verifiable way. Finally we describe an analysis of loop unrolling techniques for nested loops. This programming technique is one of few general and effective optimizing transformations. Reasons why the technique is effective depend on architecture of computers and compilers used. It is shown that the situation can be well understood using a rather simple machine model.

1. Introduction

Because of the great success of CRAY-1 supercomputers and of an ever-increasing demand for large scale scientific and engineering computation, new series of supercomputers have recently been developed and much more attention has been paid to supercomputers and their applications. The wider the scope of applications, the more important the system support for program development suitable for supercomputers becomes. A good vectorizing compiler plays a crucial role in program development of efficient programs. In fact recent supercomputers provide user with an automatic vectorizing compiler, the language specification of which is fully compatible with that for conventional scalar computers. This is effective to make supercomputers popular and in fact nowadays, supercomputers become indispensable to scientific and engineering computations of a wide range. Although supercomputers have become relatively easy to use recently, some skill in programming is still now required if one wants to use the full computation power of supercomputers. In this paper we try to describe the status of compiling techniques and discuss related problems. In section 2, we give a brief overview of programming languages for supercomputers, especially a historical review of languages for early vector processors in Japan. In section 3, we describe basic features of automatic vectorization of FORTRAN programs. Conditions for vectorization are given for basic loop programs. We give condition for vectorization with and without statement reordering. This is important because in the debugging phase of programs, vectorization without statement reordering is useful. Some other problems are also briefly reviewed. In section 4, we describe an analysis of loop unrolling techniques for nested loops and discuss problems in vectorizing compilers.

*Data Processing Center, Kyoto University, Kyoto, Japan

2. Brief Overview of Programming Languages for Supercomputers

Programming languages for supercomputers can be generally classified into the following categories:

(1) Assembly languages

(2) High level programming languages dedicated for vector/parallel processing

(3) Standard programming languages (e.g. FORTRAN) with language extensions for vector /parallel processing

(4) Standard programming languages (e.g. FORTRAN) and automatic vectorization by a compiler

(i) Automatic vectorization by a compiler and system-defined functions for vector processing

(ii) Full compatibility with programming languages on conventional computers.

Assembly languages are not used in general, except for development of system-defined functions, because description of vector/parallel processing by an assembly language is highly machine dependent and its debugging is extremely difficult. Dedicated language approach for vector/parallel processing is adopted only in cases of research or experiments because it is difficult to make use of vast amount of existing software so far developed. In 1970's, automatic vectorization technique was in a primitive state and various language extensions to describe vector/parallel processing were introduced to standard languages, mainly to FORTRAN. In case of Japanese computers, AP-FORTRAN for FACOM 230–75 AP was among them[1, 2]. Statements for array processing were introduced to AP-FORTRAN. We give several examples in AP-FORTRAN to show the outline of the language:

```
      DIMENSION V(100), X(100), Y(100)
      . . .
      V(*)=S*X(*)+Y(*)
      . . .
```

The meaning of the statement is almost self-explanatory. The above statement is equivalent to the following:

```
      DO 10 I=1, 100
            V(I)=S*X(I)+Y(I)
   10 CONTINUE
```

In order to apply operations to a part of an array, 'Index Declaration Statement' was introduced:

```
            INDEX J/2, 100, 2/
            . . .
            V(J)=S*X(J)+Y(J)
            . . .
```

The meaning of the above statement is as follows:

```
      DO 10 I=2, 100, 2
            V(J)=S*X(J)+Y(J)
   10 CONTINUE
```

It should be noted that the indirect addressing was already made available in vector processing in AP-FORTRAN:

```
     DIMENSION V(100), X(100), Y(100), IX(100)
     . . .
     V(*)=S*X(*)+Y(IX(*))
     . . .
```

The above statement is equivalent to the following DO loop.

```
      DO 10 I=1, 100
            V(I)=S*X(I)+Y(IX(I))
   10 CONTINUE
```

Besides the above kind of vector operations, system functions for vector operations: sum of vector elements, inner product of vectors, search for maximum (minimum) values in a vector elements or its index. Techniques for these macro operations in AP-FORTRAN lead to yield current vectorizing FORTRAN compilers which can detect and vectorize the above kinds of macro operations by pattern matching of statements. Vector version of basic external functions including elementary functions were also provided in AP-FORTRAN. Only two FACOM 230–75 AP systems were installed at users' sites. The considered reason for the few system is as follows: The maximum performance of FACOM 230–75 AP was 22 MFLOPS and was not enough; absence of automatic vectorization in the AP-FORTRAN compiler required rewriting existing software and users were not willing to do so; the number of application programs provided for the machine was small. It is often said that the experiences of FACOM 230–75 AP formed the basis for the design of FACOM VP series. There are certain similarities between array processing functions in AP-FORTRAN and those in the proposed FORTRAN 8X. In fact features for array processing in FORTRAN 8X are already introduced into the extended language of FORTRAN 77/VP.

Automatic vectorization was first realized in the CRAY FORTRAN compiler. Vectorization in early CRAY FORTRAN compilers was very restricted and many system-defined functions for vector operations were provided. The first Japanese vectorizing FORTRAN compiler was for HITAC M-180 IAP (Integrated Array Processor)[3, 4]. In the HITAC IAP FORTRAN, a certain kinds of macro operations such as inner product operation in FORTRAN could be vectorized together with assignment statements in the innermost DO loop. It should be noted that although there were restrictions, simple IF statements in FORTRAN could also be vectorized in the IAP FORTRAN compiler. IAP FORTRAN was reviewed in reference[5]. Fully automatic vectorization without resorting to the use of system defined special functions is indispensable to popularization of supercomputers and this was first realized in recent Japanese models of supercomputers such as HITAC S-810, FACOM VP and NEC SX. It should be noted that this approach was adopted and developed in HITAC IAP and NEC IAP.

3. Advanced Vectorization Techniques and its Formalization

3.1 Relative Execution Order of Statements and Collision of Variable References

In 3.1, 3.2, we consider basic features in automatic vectorization and we discuss the DO loops with the following restrictions:

[Restrictions]

1) The loop body consists of assignment statements only. The variable on the left-hand side is neither a simple variable nor an array variable with constant index.

2) There is no function call in the expression.

3) No variable used in the loop under consideration are connected with other variables by Equivalence statements.

If there are several statements in a vectorized loop, the relative execution order of statements in vector execution is different from that in scalar execution. Let us consider the following DO loop:

```
DO $I=I_1, I_2, \ldots, I_n$
    $S_1$
    $S_2$
    .
    .
    .
    $S_m$
END DO
```

We denote the scalar execution of the statement S_i for the control variable $I=I_j$ by $(S_i, I_j)_s$. The total scalar execution of the DO loop is carried out in the following order:

$$[[(S_i, I_j)_s, i=1, \ldots, m], j=1, \ldots, n].$$

On the contrary if we denote the vector execution of the statement S_i for the control variable $I=I_j$ by $(S_i, I_j)_v$, then the total execution of the DO loop is carried out in the following order:

$$[[(S_i, I_j)_v, j=1, \ldots, n], i=1, \ldots, m]$$

[Definition] relative execution order of statements

I) In scalar execution mode, the relative execution order

$<_s$ (precede) or $=_s$ (same time) or $>_s$ (succeed) between $(S_i, I_j)_s$ and $(S_{i'}, I_{j'})_s$ is defined as follows:

$$(S_i, I_j)_s <_s (S_{i'}, I_{j'})_s \rightleftharpoons j<j' \text{ or } (i<i' \text{ and } j=j') \quad (1)$$

$$(S_i, I_j)_s =_s (S_{i'}, I_{j'})_s \rightleftharpoons j=j' \text{ and } i=i' \quad (2)$$

$$(S_i, I_j)_s >_s (S_{i'}, I_{j'})_s \rightleftharpoons j>j' \text{ or } (i>i' \text{ and } j=j') \quad (3)$$

where $\rightleftharpoons$ denotes if and only if.

II) In vector execution mode the relative execution order between $(S_i, I_j)_v$ and $(S_{i'}, I_{j'})_v$ is defined as follows:

$$(S_i, I_j)_v <_v (S_{i'}, I_{j'})_v \rightleftharpoons i<i' \text{ or } (i=i' \text{ and } j<j') \quad (4)$$

$$(S_i, I_j)_v =_v (S_{i'}, I_{j'})_v \rightleftharpoons i=i' \text{ and } j=j' \quad (5)$$

$$(S_i, I_j)_v >_v (S_{i'}, I_{j'})_v \rightleftharpoons i>i' \text{ or } (i=i' \text{ and } j>j') \quad (6)$$

We now give an example. In scalar execution mode the execution of the statement S_2 for the control variable $I=I_1$ precedes the execution of the statement S_1 for the control variable $I=I_2$ and in fact we have

$$(S_2, I_1)_s <_s (S_1, I_2)_s.$$

In vector execution mode, however, the execution of the statement S_2 for the control variable $I=I_1$ succeeds the execution of the statement S_1 for the control variable $I=I_2$. In this case we have

$$(S_2, I_1)_v >_v (S_1, I_2)_v.$$

The computation result in vector execution mode must be the same as that in scalar execution mode even if there are changes in the order of execution of statements. In order to check this we have to check the "reference relation of variables" and we introduce some technical terms:

[Definition] Definition and Use of a variable

If a variable appears on the left-hand side of an assignment statement, the occurrence of the variable is called a definition of the variable. If a variable appears on the right-hand side of an assignment statement, the occurrence of the variable is called a use of the variable. The definition or the use of a variable is also called a reference to the variable.

[Definition] Collision of variable references

When the address of a certain variable v in $(S_i, I_j)_s$ coincides with the address of a variable v' in $(S_{i'}, I_{j'})_s$, it is said that there is a collision in variable references.

We now describe how to detect collisions of references of variables. In case of simple variables, for any two occurrences of the same identifier, there is a collision of references. There is no problem in detection of collision of references for simple variables. Let us consider a case of a k-dimensional array identifier. We denote the index set of an array variable v in $(S_i, I_j)_s$ and that of an array variable v' in $(S_{i'}, I_{j'})_s$ by $(f_1, f_2, \ldots, f_k)$ and $(f_1', f_2', \ldots, f_k')$, respectively, where f_p $(1 \leq p \leq k)$ is a function of I_j and f_p' $(1 \leq p \leq k)$ is a function of $I_{j'}$. There is a collision of references when there is a set of integer solutions valied as control values satisfying the following Diophantine equations:

$$\begin{aligned} f_1 &= f_1' \\ f_2 &= f_2' \\ &\cdots \\ f_k &= f_k' \end{aligned} \quad (7)$$

In case of control values of DO loop and values of f_p and f_p' $(1 \leq p \leq k)$ can be computed at compile time, it is possible to verify the existence/nonexistence of valid solution of the equation by computation at compile time. If the upper and/or lower bound of the DO loop control variable are/is given by variables and cannot be determined at compile time, then the Diophantine equation must be solved symbolically. If the form of expressions for array index is restricted, e.g. a linear form, there are cases where the Diophantine equation can be solved symbolically. In actual programs expressions of

Table 1 Cases of Collision of References

case	v in $(S_i, I_j)_s$	v' in $(S_{i'}, I_{j'})_s$
I	Definition	Use
II	Definition	Definition
III	Use	Definition
IV	Use	Use

Table 2 Dependence Relations

case	$(S_i, I_j)_s$:$(S_{i'}, I_{j'})_s$		
	(1) $<_s$	(2) $=_s$	(3) $>_s$
I) v in $(S_i, I_j)_s$: Def v' in $(S_{i'}, I_{j'})_s$: Use	Flow Dependence?*)	Anti Dependence	Anti Dependence?*)
II) v in $(S_i, I_j)_s$: Def v' in $(S_{i'}, I_{j'})_s$: Def	Output Dependence?*)	—	Output Dependence?*)

array index have simple forms in many cases where detection of collisions of references can be done relatively easily. If nonexistence of solution can be verified, then there is no collision of references and the loop can be safely vectorized. It should be noted that this is a sufficient condition and even when it is difficult to prove nonexistence of solution at compile time, there are many cases where there is no solution in practice. In those cases a compiler must stand on the safer side and abandon vectorization of the loop. In order to cope with this situation, a compiler usually provides compiler option of 'forced vectorization' by user-supplied information on reference relation of variables.

3.2 Dependence Analysis and Condition for Vectorization with/without Statement Reordering

When there are collisions of variable references, we must analyze the situation more precisely. We use dependence analysis technique discussed by the research group of Kuck. It should be noted that we will pay careful attention to statement reordering in vectorization, different from the treatment described in Pauda[6].

Let us assume that there is a collision of references between v in $(S_i, I_j)_s$ and v' in $(S_{i'}, I_{j'})_s$, namely the address of v is equal to that of v'. We must consider the following cases shown in Table 1:
For case IV, there is no problem from the view point of vectorization, because the computation result cannot be affected by the changes in the execution order. Cases I and III can be treated in the same way and without loss of generality, we can only consider case I as a representative. Thus in the following we consider cases I and II. The relative execution order between $(S_i, I_j)_s$ and $(S_{i'}, I_{j'})_s$ is either $<_s$ or $=_s$ or $>_s$ and we can further carry out case analysis which can be summarized as the following Table 2:

We now give some definitions of technical terms:

[Definition] Flow Dependence

If the value defined by the statement S_i is used in the statement $S_{i'}$, then it is said that there is a Flow Dependence relation from the statement S_i to the statement $S_{i'}$. This relation is denoted by

$$S_i \,\delta\, S_{i'}. \tag{8}$$

[Definition] Anti Dependence

If the old value used in the statement $S_{i'}$ is redefined in the statement S_i, then it is said that there is an Anti Dependence relation from the statement S_i to the statement $S_{i'}$. This relation is denoted by

$$S_i \,\bar{\delta}\, S_{i'}. \tag{9}$$

[Definition] Output Dependence

If the value defined in the statement S_i is redefined in the statement $S_{i'}$, then it is said that there is an Output Dependence relation from the statement S_i to the statement $S_{i'}$. This relation is denoted by

$$S_i \,\delta^0\, S_{i'}. \tag{10}$$

In the following we consider cases given in Table 2.
Case I-(1): v: Definition; v': Use and $(S_i, I_j)_s < (S_{i'}, I_{j'})_s$

If there is no assignment statement for the variable v between S_i and $S_{i'}$*), then the value of v defined in $(S_i, I_j)_s$ is actually used in $(S_{i'}, I_{j'})_s$ and there is a Flow Dependence from the statement S_i to the statement $S_{i'}$. Depending on the relation between i and i', the relation can be represented graphically as shown in Fig. 1.
For the case (a), Eq. (1) and $(i<i')$ give

$$(i<i') \text{ and } (j \leqq j') \tag{11}$$

and from Eq. (4), we have

$$(S_i, I_j)_v <_v (S_{i'}, I_{j'})_v.$$

Since we are now analyzing the case of $(S_i, I_j)_s <_s (S_{i'}, S_{j'})_s$, the relative execution order in vector execution mode is the same as that in scalar execution mode, as far as definition and use of variable v and v' are concerned. In this case DO-loop can be vectorized. Let us give a simple example:

```
      DO 10 I=1, N
            C(I+1)=A(I)+B(I)
            E(I)=C(I)-D(I)
   10 CONTINUE
```

In fact for the array variable C, there is a collision of references between $(S_1, I_j)_s$ with $I_j=j+1$ and $(S_2, I_{j'})_s$ with $I_{j'}=j'$. The Diophantine equation to be solved is given by

*If there is an assignment statement S_i'' for the variable v between S_i and S_i', then the value defined in S_i is killed by the definition in S_i'' and cannot reach to the statement S_i', and therefore there is no dependence relation between S_i and S_i'. The marks'?' in Table 2 denote the necessity of carrying out this kind of flow analysis.

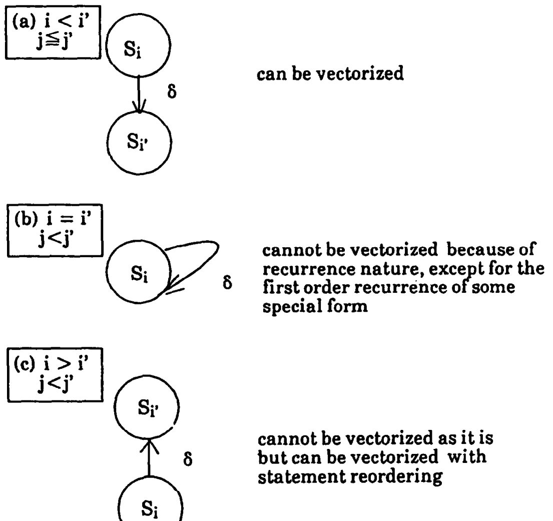

(a) i < i'
j≦j'
S_i
δ
$S_{i'}$
can be vectorized
(b) i = i'
j<j'
S_i
δ
cannot be vectorized because of recurrence nature, except for the first order recurrence of some special form
(c) i > i'
j<j'
$S_{i'}$
δ
S_i
cannot be vectorized as it is but can be vectorized with statement reordering

Fig. 1 Flow Dependence Relation.
v in $(S_i, I_j)_s$: Def; v' in $(S_{i'}, I_{j'})_s$: Use

$$j+1=j'$$

and this has the solution set $(j, j')=\{(1, 2), (2, 3), \ldots (N-1, N)\}$. For this set of solutions we can verify that *Eqs.* (1) and (4) hold.

For the case (b), *Eq.* (1) and $(i=i')$ give

$$(i=i') \text{ and } (j<j') \tag{12}$$

and in this case, too, we have

$$(S_i, I_j)_v <_v (S_{i'}, I_{j'})_v.$$

We give an example:

```
      DO 10 I=1, N
            C(I+1)=C(I)/B(I)
   10 CONTINUE
```

From the logical point of view, the relative execution order is preserved but owing to the nature of pipeline arithmetic, this case is inadequate for vector operation. In fact except for the first order recurrence in a simple form, this kind of recurrence loop cannot be vectorized.

For the case (c), Eq. (1) and $(i>i')$ give

$$(i>i') \text{ and } (j<j') \tag{13}$$

and in this case, we have from Eq. (6)

$$(S_i, I_j)_v >_v (S_{i'}, I_{j'})_v.$$

Since we are now analyzing the case of $(S_i, I_j)_s <_s (S_{i'}, I_{j'})_s$, the relative execution order in vector execution mode is different from that in scalar execution mode. In this case DO-loop cannot be vectorized. Let us give a simple example:

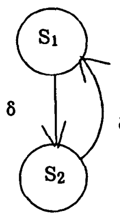


Fig. 2 Recurrence Flow Dependence Relation.

```
      DO 10 I=1, N
            E(I)=C(I)-D(I)
            C(I+1)=A(I)+B(I)
   10 CONTINUE
```

It is easy to see that without changing the meaning of the program, this DO-loop can be transformed into the DO-loop we considered as the case (a). So if we allow a compiler to reorder source statements for vectorization, then the compiler can vectorize this loop.

Let us consider the following example:

```
      DO 10 I=1, N
            C(I)=A(I)+B(I)            S1
            A(I+1)=C(I)-D(I)          S2
   10 CONTINUE
```

In this case there are two collisions of variable references. One is for the variable A and the other is for the variable C. It is easy to verify that both of them are flow dependence relations. There is a cycle in the dependence graph shown in Fig. 2 and this loop cannot be vectorized because of its essentially recurrence nature.

Case I-(2): v: Definition; v': Use and $(S_i, I_j)_s =_s (S_{i'}, I_{j'})_s$

From Eq. (2) and (5), we have

$$i=i', j=j' \tag{14}$$

and

$$(S_i, I_j)_v =_v (S_{i'}, I_{j'})_v.$$

We give a simple example for this case:

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

The value of $C(I)$ is used first in the evaluation of the right-hand side and then the store operation is carried out (anti dependence relation). There is no problem in vectorization.

Case I-(3): v: Definition; v': Use and $(S_i, I_j)_s >_s (S_{i'}, I_{j'})_s$

If there is no assignment statement for the variable v between $S_{i'}$ and S_i, then after the value of v' is used in

$(S_{i'}, I_{j'})_s$, the variable v is defined in $(S_i, I_j)_s$. There is an Anti Dependence relation from the statement $S_{i'}$ to the statement S_i. Depending on the relation between i and i', the relation can be represented graphically as shown in Fig. 3.

For the case (a), Eq. (3) and $(i<i')$ give

$$(i<i') \text{ and } (j>j') \tag{15}$$

and from Eq. (4) we have

$$(S_i, I_j)_v <_v (S_{i'}, I_{j'})_v.$$

Since we are now analyzing the case of $(S_i, I_j)_s >_s (S_{i'}, I_{j'})_s$, the relative execution order in vector execution mode is different from that in scalar execution mode. In this case DO-loop cannot be vectorized without statement reordering. We give a simple example:

```
      DO 10 I=1, N
            C(I)=A(I)+B(I)              S1
            E(I)=C(I+1)-D(I)            S2
   10 CONTINUE
```

In fact for the array variable C, there is a collision of references between $(S_1, I_j)_s$ with $I_j=j$ and $(S_2, I_{j'})$ with $I_{j'}=j'+1$. The Diophantine equation to be solved is given by

$$j=j'+1$$

and this has the solution set $(j, j')=\{(2, 1), (3, 2), \ldots, (N, N-1)\}$. For this set of solutions we can verify that Eqs. (3) and (4) hold. If we are allowed to introduce a temporary array, then the above loop can be transformed to the following loop which is vectorizable without statement reordering.

```
      DO 10 I=1, N
            TEMP(I)=C(I+1)              S0
            C(I)=A(I)+B(I)              S1
            E(I)=TEMP(I)-D(I)           S2
   10 CONTINUE
```

The dependence graphs for the original loop and modified loop are given in Fig. 4.

For the case (b), Eq. (3) and $(i=i')$ give

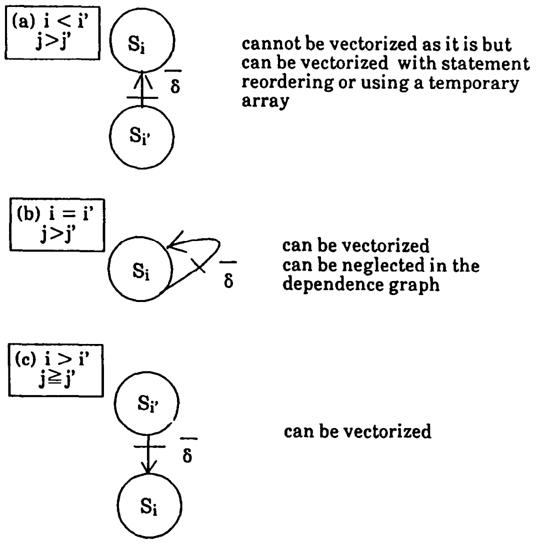
$$(i=i') \text{ and } (j>j') \tag{16}$$

and in this case, we have

$$(S_i, I_j)_v >_v (S_{i'}, I_{j'})_v.$$

There is no problem for vectorization. We give an example:

```
      DO 10 I=1, N
            C(I)=C(I+1)/B(I)
   10 CONTINUE
```



(a) i < i' j>j'
S_i
$\bar{\delta}$
$S_{i'}$
cannot be vectorized as it is but can be vectorized with statement reordering or using a temporary array
(b) i = i' j>j'
S_i
$\bar{\delta}$
can be vectorized can be neglected in the dependence graph
(c) i > i' j≧j'
$S_{i'}$
$\bar{\delta}$
S_i
can be vectorized

Fig. 3 Anti Dependence Relation.
v in $(S_i, I_j)_s$: Def; v' in $(S_{i'}, I_{j'})_s$: Use

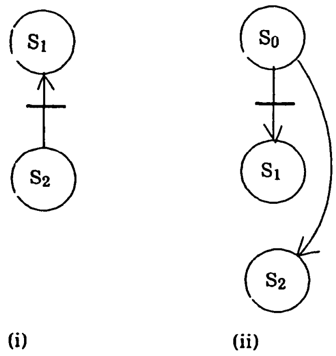


Fig. 4 Vectorization using a temporary array.

For the case (c), Eq. (3) and $(i>i')$ give

$$(i>i') \text{ and } (j \geqq j') \tag{17}$$

and in this case, we have from Eq. (6)

$$(S_i, I_j)_v >_v (S_{i'}, I_{j'})_v.$$

Since we are now analyzing the case of $(S_i, I_j)_s >_s (S_{i'}, I_{j'})_s$, the relative execution order in vector execution mode is the same as that in scalar execution mode. In this case DO-loop can be vectorized. We give a simple example:

```
      DO 10 I=1, N
            E(I)=C(I+1)-D(I)
            C(I)=A(I)+B(I)
   10 CONTINUE
```

As far as cases II-(1) and II-(3) are concerned, we can

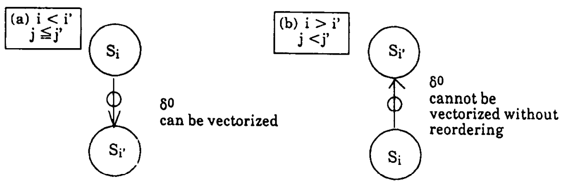
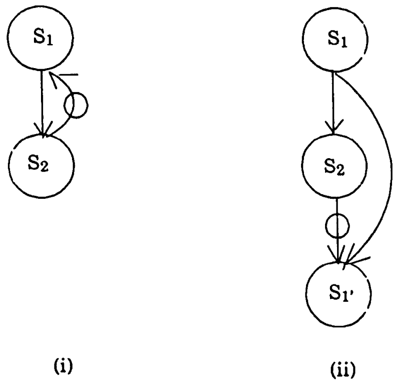


Fig. 5 Output Dependence Relation.
v in $(S_i, I_j)_s$: Def; v' in $(S_{i'}, I_{j'})_s$: Def



(i) (ii)

Fig. 6 Vectorization by reordering statements and using a temporary array.

only consider the case II-(1) without loss of generality.

Case II-(1): v: Definition; v': Definition and $(S_i, I_j)_s <_s (S_{i'}, I_{j'})_s$

If there is no assignment statement between S_i and $S_{i'}$, then there is an output dependence relation, as shown in Fig. 5.

(a) $i<i'$

For this case, we have from Eq. (1) and ($i<i'$)

$$i<i' \text{ and } j \leqq j' \tag{18}$$

Since $(S_i, I_j)_s <_s (S_{i'}, I_{j'})_s$ and $i<i'$, the relative execution order is preserved in vectorization. Thus there is no problem for vectorization. Although not an interesting example, we give an example:

```
   DO 10 I=1, N
         C(I+1)=A(I)+B(I)
         C(I)=E(I)-D(I)
10 CONTINUE
```

(b) $i>i'$

Eq. (1) and ($i>i'$) gives

$$i>i' \text{ and } j<j'. \tag{19}$$

In this case the relative execution order is changed by vectorization, therefore this loop cannot be vectorized without statement reordering. We give an example:

```
   DO 10 I=1, N
         C(I)=A(I)+B(I)
         C(I+1)=E(I)-D(I)
10 CONTINUE
```

Let us consider the following example:

```
   DO 10 I=1, N
         C(I)=A(I)+B(I)
         C(I+1)=C(I)-D(I)
10 CONTINUE
```

This loop cannot be vectorized by simple statement reordering, because there is a cycle in the dependence graph shown in Fig. 6(i). Introducing a temporary array, the loop can be transformed into the following loop which can be vectorized:

```
   DO 10 I=1, N
         TEMP(I)=A(I)+B(I)
         C(I+1)=TEMP(I)-D(I)
         C(I)=TEMP(I)
10 CONTINUE
```

The dependence graph of this loop is shown in Fig. 6(ii).

Results given in Figs. 1-6 and Eqs. (11)-(19) can be summarized in Table 3.

Table 3 Vectorizability of statements. V: Vectorizable; S: Not Vectorizable because of recurrency; VR: Vectorizable with Reordering Statements; VT: Vectorizable using Temporary; VR*: Vectorizable with Reordering Statements and using Temporary

v in $(S_j, I_j)_s$ v' in $(S_{i'}, I_{j'})_s$	i, i' j, j'	Dependency	Vectorizability
v: Def	$i<i'$; $j \leqq j'$	Flow	V
v': Use	$i=i'$; $j<j'$	Flow	S
	$i>i'$; $j<j'$	Flow	VR
	$i<i'$; $j>j'$	Anti	VT, VR
	$i \geqq i'$; $j \geqq j'$	Anti	V
v: Def	$i<i'$; $j \leqq j'$	Output	V
v': Def	$i>i'$; $j<j'$	Output	VR*

We can now summarize the analysis that we have so far described:

If we carry out analysis of collision of variable references and dependence analysis using compiler flow analysis techniques if necessary, and make a dependence graph. We can say the following:

[Vectorization without Reordering Statements]

1) The loop can be vectorized if there is no upward-directed edge in the dependence graph.

2) The loop can be vectorized if there is no upward-directed edges except for those of anti dependence rela-

tions, because upward-directed edges related to anti dependence can be eliminated by introducing temporary arrays. This case can thus be reduced to case 1).

[Vectorization with Reordering Statements]

The loop can be vectorized if there is no cycle (strongly connected components) which consists of only edges related to flow dependence in the dependence graph.

Vectorization can be done by the following steps:

1) Elimination of cycles in the dependence graph
Even if there is a cycle in the dependence graph, at least one edge of the cycle is related to anti dependence or output dependence. By introducing a temporary array and reordering statements if necessary, the cycle can be broken. This procedure is repeatedly applied until there is no cycle in the dependence graph.

2) Elimination of upward-directed edge in the dependence graph
Using the topological sort, as pointed out in the reference[7], the dependence graph can be transformed (reordered) in such a way that there is no upward-directed edge in it.

3) Vectorization
The program can now be vectorized.

It should be noted that except for cases of essentially recurrence algorithms, the loop can be vectorized if we allow a compiler to reorder statements and this is realized in recent Japanese FORTRAN 77 compilers. Although the maximum degree of vectorization is good for production programs, there are cases where it is good to restrict vectorization by prohibiting a compiler to reorder statements; it makes it easier to identify a statement causing a run-time error. It is convenient for users to have this kind of control on the degree of vectorization by a compiler option. (e.g. NOADV option in FACOM FORTRAN 77/VP). Table 3 gives conditions of vectorization with/without reordering statements. It should be noted that the conditions in Table 3 can be verified rather easily.

3.3 Vectorization in Other Cases

In this section we briefly summarize vectorization of loops which were not treated in the previous section. So far we assumed the variable on the left-hand side is neither a simple variable nor an array variable with a constant index. Important cases involving simple variables are cases of macro operations, such as vector sum and inner product operations, and cases where a temporary array variable must be used for vectorization as in the following example:

```
DO 10 I=1, N              DO 10 I=1, N
     ....                      ....
   X= ...                   TEMP(I)= ...
     ....                      ....
   = ..X                    =TEMP(I)...
     ....                      ....
10 CONTINUE               10 CONTINUE
                             X=TEMP(I)
```

These cases are well treated by recent vectorizing compilers and there seems to be no problem.

In vectorization of IF statements, there are three kinds of patterns of object codes supported by the hardware in case of FACOM VP, and one of them can be selected depending on conditions such as how often the condition of the IF statement becomes true. Detailed description of them is given in reference[8]. The compiling algorithm of IF statements on HITAC S-810 is described in reference[9]. In case of NEC SX system, readers are referred to reference[10]. Vectorization of the loop involving 'GO TO statement going outside the loop' is not easy in general but such a loop can be vectorized in Japanese FORTRAN 77 compilers under certain rather restrictive conditions. In case of multiply nested DO loops, techniques in existing compilers are i) the loop interchange, ii) reduction of tightly nested loops into a single loop, and iii) splitting a non-tightly nested loops into separate tightly nested loops[10]. The loop interchange technique is used if the original innermost loop has a recurrence data reference relation, but by the loop interchange, the innermost loop becomes vectorizable. It is also used in such a way that vectorization becomes most efficient from the view point of memory access pattern. Continuous memory access is generally most efficient compared with constant stride access or indirect access, and if both I and J can be the control variable of the vectorized loop involving $A(I, J)$, then a compiler tries to select I as the control variable of vectorization. As far as techniques ii) and iii) are concerned, the situation is not so easy and the condition imposed on current compilers are very restrictive. Tsuda *et al.*[11] give a method to treat general multiply nested DO loops by vector indirect addressing.

4. Some Analysis of Loop Unrolling Transformation and Related Codes

Although the peak performance of supercomputers is very high, e.g. several hundreds MFLOPS or one or two GFLOPS, one of the problems with this type of vector supercomputers is that it is not necessarily easy to attain such high performance with ordinary programs. In fact it is often said that in case of register-register type of vector supercomputers, such as CRAY-1, there are typical three performance levels[12, 13].:

Scalar: a performance level when a vector processing unit is not utilized in the computation

Vector: a performance level when a vector processing unit is effectively used in the most CPU-time consuming part of the computation

Super-vector: a performance level when vector processing units are fully utilized. Chaining is effectively used.

In order to attain high performance, there are many things to be considered. Among them are whether a computational algorithm used is well vectorized or not, characteristics of hardware and compiler, programming techniques such as loop unrolling and hidden performance bottleneck caused by various details such as memory bank conflict. Supercomputers have inherently specific characters and characteristics differ greatly from computer to computer. It is often the case that a slight modification in an algorithm or in a program causes transition from one performance level to higher one. The loop unrolling technique for nested loops is one of few general and powerful programming techniques. Loop unrolling applied to the inner loop is one of common optimizing transformation which is effective on scalar computers. On the other hand, loop unrolling for the outer loop is useful for supercomputers. It is often quite effective to attain super-vector performance. Although conceptually simple, and generally effective, the reason why it is effective in performance improvement is not so simple. It depends on computer architecture and the compiler used. For example, if there are sets of the same kind of independent pipeline arithmetic units, as in the case of HITAC S-810, then the loop unrolling transformation enables the compiler to make full use of many independent pipeline arithmetic units. This immediately leads to effective performance improvement. Even if there is only one set of the same kind of arithmetic pipeline unit as in the case of CRAY-1 and FACOM VP, the loop unrolling technique is still effective. In order to carry out quantitative analysis, we consider a model of a register-register type vector supercomputer to model CRAY-1 or FACOM VP etc, which is similar to that used by Dongarra[14]. We assume the following virtual vector instructions

VL r, M	Vector Load $r \leftarrow M$	Load from memory to vector register including a constant vector
VST r, M	Vector Store $r \rightarrow M$	Store from vector register to memory
VAD $r3, r1, r2$	Vector Add $r3 \leftarrow r + r2$	Vector elementwise addition Vector
VMS $r2, r1, s$	Vector MultiplyScalar $r2 \leftarrow r1^* s$	Multiply each element of vector by scalar
VTR $r2, r1$	Vector Register Transfer $r2 \leftarrow r1$	Transfer from vector register to vector register

where r or r_i denotes a vector register, s denotes a floating point scalar register and M denotes the memory address.

Following Dongarra, we assume that the time chart

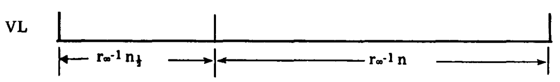


Fig. 7 Time chart of instruction execution.

of instruction execution can be represented by Fig. 7:

The parameters r_∞^{-1} and $n_{1/2}$ are Hockney's parameters and r_∞^{-1} denotes the maximum performance and $n_{1/2}$, N-half gives the vector length for which a half of the maximum performance is attained[14]. We assume chaining may occur after the time r_∞^{-1} $n_{1/2}$ has elapsed. For simplicity, we assume that all the instruction can produce one result per chime and the values of $n_{1/2}$ are all equal. Different from conventional scalar computers, it is not allowed to use a single vector register both as an operaned register and as the resultant register on vector computers, and we assume the VTR instruction.

Let us consider the following program for matrix multiplication using SAXPY operation. The method is called the middle product method. For comparison of the method with other method such as the inner product method, readers are referred to[14–16].:

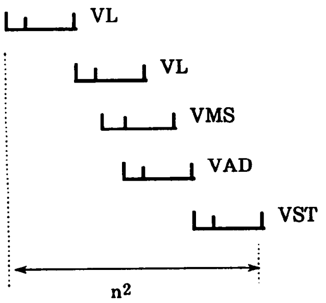
```
      DO 40 J=1, N
      DO 10 I=1, N
            X(I)=0
10    CONTINUE
      DO 30 K=1, N
      DO 20 I=1, N
            X(I)=X(I)+B(I, K)*C(K, J)
20    CONTINUE
30    CONTINUE
      DO 40 I=1, N
            A(I, J)=X(I)
40    CONTINUE
```

If the compiler cannot produce object codes to keep the value of $X(I)$ in a vector register during the execution of loop 30, and only one load/store pipeline is available, then we can give the execution chart in Fig. 8: The cpu time can be estimated by the following equation, where n denotes the value of the variable N:

$$T_1 = r_\infty^{-1} n^2 (3n + 5n_{1/2}) \tag{20}$$

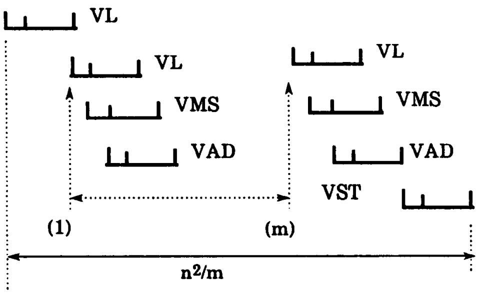
Now let us consider the program in which loop unrolling of depth m is applied to the loop 30: The relevant portion of the program is given as follows:

```
      NN=(N/m)*m
      DO 30 K=1, NN, m
      DO 20 I=1, N
            X(I)=X(I)+B(I, K)*C(K, J)+
```



```
··········· inner loop ($n^2$) ···········
VL     r1, X       r1←X
VL     r2, B       r2←B
VMS    r3, r2, fc  r3←r2*fc
VAD    r4, r1, r3  r4←r1+r3
VST    r4, X       r4→X
··········· inner loop end ···········
```

Fig. 8 Time chart of the program without unrolling (1). $X(I)$ is not kept in a vector register in this case. The vectorized instructions for the innermost loop 20 are repeated for n^2 times ($J=1$, N; $K=1$, N), where n denotes the value of variable N.



```
········ inner loop ($n^2/m$) ········
         VL     r1, X
         VL     r2, B
         VMS    r3, r2, fc
         VAD    r4, r1, r3
··············································
         VL     r2, B
         VMS    r3, r2, fc
         VAD    r1, r4, r3
         VST    r1, X
··········· inner loop end ···········

m: even
```

Fig. 9 Time chart of the loop program with unrolling (1). $X(I)$ is not kept in a vector register.

```
              . . .+B(I, K+m−1)
              *C(K+m−1, J)
   20 CONTINUE
   30 CONTINUE
      IF (MOD(N, M).NE.0)THEN
        . . . (*post processing of remaining loop for
              K=NN+1, N*)
      ENDIF
```

For this case we can give the time chart shown in Fig. 9. In this case the cpu time can be estimated by the following equation:

$$T_{1u}=r_\infty^{-1}n^2/m\{(m+2)n+(3m+2)n_{1/2}\} \qquad (21)$$

where we assumed that mod $(n, m)=0$ for simplicity. If we define the effect of the loop unrolling in this case by $E_u=T_1/T_{1u}$, then we have

$$E_u=m(3+5n_{1/2}/n)/\{(m+2)+(3m+2)n_{1/2}/n\}$$
$$\rightarrow 3m/(m+2) \qquad n\gg n_{1/2} \qquad (22)$$

We give values of E_u for values of m usually used in Table 4.

If we compute E_u from the MFLOPS data reported by Dongarra[14], we have for $m=2$, 4, 8, 16, $E_u=60/39(=1.54)$, $83/39(=2.13)$, $101/39(=2.59)$, $111/39(=2.85)$, respectively for CRAY-1M. It can be said that the result is in fairy good agreement with Table 4. For the data of CRAY-1S, E_u for $m=2$, 4, 8, 16, are $53/40(=1.33)$, $72/40(=1.80)$, $86/40(=2.15)$ and $96/40(2.4)$, respectively. In this case it is not as good in agreement as in the case of CRAY-1M.

Table 4 Effect of Loop Unrolling computed by Eq. (22)

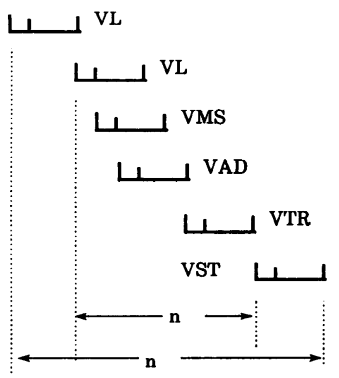
m	E_u
2	1.50
4	2.00
8	2.40
16	2.67

Let us consider the case where the compiler can produce object codes to keep $X(I)$ in a vector register*). Furthermore we assume that there are two load/store pipes. We can give the instruction sequence and its timing chart for the unrolled program on our model shown in Fig. 10. On vector computers, vector register $r4$ must not be equal to $r1$ in the VAD $r4$, $r1$, $r3$ instruction and we must use the VTR instruction.
In this case the cpu time can be estimated by

$$T_2=2r_\infty^{-1}n\{n(n+2n_{1/2})+(n+n_{1/2})\} \qquad (23)$$

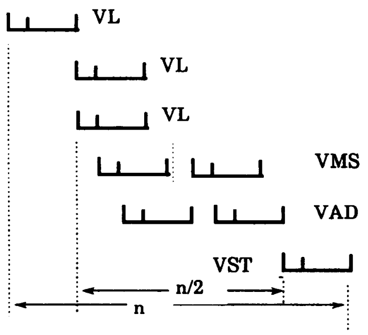
If we apply the loop unrolling to the loop program once, we have the instruction sequence and timing chart given in Fig. 11, where we assume that m is even. It should be noted that it is not necessary to use VTR in-

*For the simplicity of analysis, we give an analysis for the case when $n\leq n_r$, where n_r is the length of a vector register. It should be noted that even in the case when $n>n_r$, $X(I)$ can be kept in a vector register during execution of the nested DO loop, if the loop segmentation technique is used. We give discussion on the effect of n_r in Appendix.



```
VL     r1, X          r1←X
··········· inner loop (n) ···········
VL     r2, B          r2←B
VMS    r3, r2, fc     r3←r2*fc
VAD    r4, r1, r3     r4←r1+r3
VTR    r1, r4         r1←r4
··········· inner loop end ···········
VST    r1, X          r1→X
```

Fig. 10 Time chart of the loop program without unrolling (2). $X(I)$ is kept in a vector register.



```
VL     r1, X          r1←X
··········· inner loop (n²) ···········
VL     r2, B          r2←B
VL     r3, B          r3←B
VMS    r4, r2, fc     r4←r2*fc
VAD    r5, r1, r4     r5←r1+r4
VMS    r6, r3, fc     r6←r3*fc
VAD    r1, r5, r6     r1←r5+r6
··········· inner loop end ···········
VST    r1, X          r1→X
```

Fig. 11 Time chart of the loop program with unrolling (2). $X(I)$ is kept in a vector register.

struction. The cpu time can be estimated by

$$T_{2u}=r_\infty^{-1}n\{(n/2)(2n+5n_{1/2})+(2n+2n_{1/2})\} \qquad (24)$$

Therefore in this case we have for the effect of loop unrolling $E_{2u}=T_2/T_{2u}$

$$E_{2u}=2\{n(n+2n_{1/2})+(n+n_{1/2})\}$$
$$/\{n(n+5/2n_{1/2})+(2n+2n_{1/2})\} \qquad (25)$$

In order to check the validity of Eqs. (24) and (25), we show the result of experiments on FACOM VP 200 at Data Processing Center of Kyoto University.

The graph of E_u vs n the order of matrix is shown in Fig. 12. If $n \gg \sqrt{n_{1/2}}$, we can consider the higher order terms of n in Eq. (25) and we have from Eq. (25)

$$E_{2u}\approx 2-2/(2n/n_{1/2}+5) \qquad \to 2(n/n_{1/2}\gg 1). \qquad (26)$$

The graph of Eq. (26) is also shown in Fig. 12, and it can be said that the result can be explained by the analysis and Eq. (26).

Although the same unrolling technique is used, the situation varies depending on whether $X(I)$ is kept in a vector register or not. In the following we give more discussion on this point. Judging from the time chart shown in Fig. 11, even if we apply loop unrolling with the depth of $m>2$, we cannot expect significant improvement in performance. On FACOM VP 200 (clock cycle:7.5 nsec; FORTRAN 77/VP V10L20) at Data Processing Center of Kyoto University, we obtained the following data given in Table 5:

Table 5 Matrix multiplication of FACOM VP 200 (clock cycle 7.5 ns, FORTRAN 77/VP Compiler V10L20) ($N=300$)

m	MFLOPS
1	245.2
2	434.2
4	450.6

In fact with $m=2$, the super-vector performance was already obtained and the improvement for $m>2$ was not so significant.

In the above measurement, even in the case of $m=1$, the DO statement

DO 30 $K=1$, NN, m

was used to avoid the effect of the compiler optimization effect. If we use the DO statement

DO 30 $K=1$, N

then the compiler detects the situation where the loop unrolling is applicable, and it really carries out the loop unrolling optimization in the form of duplicating the body statement. Unfortunately in this case, unnecessary register transfer code is issued and the measured perfor-

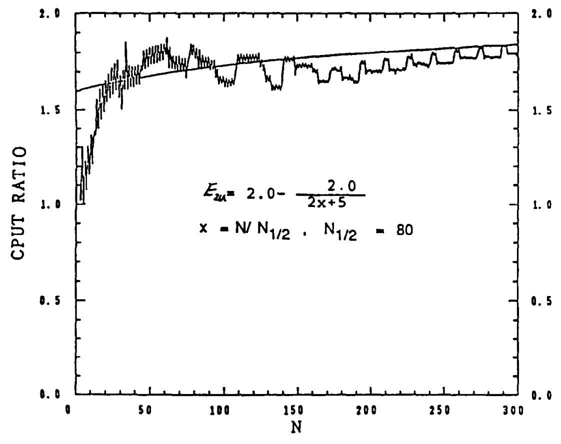


Fig. 12 The loop unrolling effect.

mance was 251.5 MFLOPS (vector performance). Although it is better than the performance without unrolling in Table 5 ($m=1$) but not as good as the result of $m=2$ (super-vector performance). If we replace the loop body of the unrolled loop by

$$X(I)=X(I)+B(I,K)^*C(K,J)$$
$$X(I)=X(I)+B(I,K+1)^*C(K+1,J)$$
$$\dots$$
$$X(I)=X(I)+B(I,K+m-1)^*C(K+m-1,J)$$

then the performance obtained on FACOM VP is only vector performance, while the super-vector performance was attained before. The reason for the poor performance is that extra register transfer codes are issued. This shows that on FACOM VP, the loop unrolling technique in a general form is not so effective, but if it is carefully coded, the program with unrolling depth 2 gives a satisfactory performance. It is advised to use explicit loop unrolling carefully on FACOM VP.

On HITAC S-810, the loop unrolling is generally effective up to the depth of approximately 10. In fact compiler supports option for automatic loop unrolling up to depth 10. The above two forms of the unrolled body give quite similar performance and good code is produced by the compiler from both forms of the source code, although load/store instructions are issued in the inner loop. The important part of the generated codes are:

```
VLD
VLD
VMAD
VLD
VMAD
. . .
VSTD
```

Table 6 Effect of Loop Unrolling on HITAC S-810

m	MFLOPS	E_u
1	184.7	1.0
2	325.6	1.76
4	404.4	2.19
8	428.4	2.32
10	446.9	2.42

We give the results of the unrolled program with the second form of the unrolled body on HITAC S-810 at Computer Center at University of Tokyo in Table 6. Measurement was done on August 7, 1987 and the compiler used was FORTRAN 77/HAP V20-1C. It seems performance improvement comes from the fact that relative weight of load/store instructions decrease as the unrolled depth increases.

5. Conclusion

We gave a detailed description of basic features of vectorization. Using the dependence analysis technique, we made it clear under what condition a loop program can be vectorized without/with reordering of source statements, or using a temporary array. In the latter part of the paper, we gave some performance analysis of the loop unrolling technique which is one of few general and effective programming techniques. Without using machine specific details, e.g. details concerning chain slot time, performance behavior of the method become clear to some extent. It became clear that the features of object code generated by compilers affect the performance behavior of the method greatly. Recently compiler technology made a progress to handle this problem but in some cases, there is still room to be improved. If one wants to make use of the computation power of vector processors fully, he must pay careful attention to the problem.

Supercomputers become much more accessible and easy to use than those of the past generation, owing to a progress in vectorizing compilers and support software systems, such as interactive software tools for vectorization. However, current systems do not give enough information to users who want to use fully the computation power of supercomputers, e.g. information on possibility of memory bank conflict. It will become an important problem to future support systems how more detailed technical information can be presented to users for effective program improvement in a user-friendly way.

References

1. Miwa, O., Inui, N., Kume, N. and Uchida, K. *FACOM 230-75 Array Processor (in Japanese)*, *Johoushori* **18**, 4 (1977), 410-415.
2. Miwa, O. Kume, N. Uchida, K., Suzuki, S., Tanakura, Y. and Isobe, F. *FACOM 230-75 Array Processor system* (in Japanese), *Fujitsu* **29**, 1 (1978), 94-128.
3. Takanuki, R., Nakata, I., Umetani, Y. Some Compiling Algorithms for an Array Processor, *3rd USA-JAPAN Comp. Conf.*

(1977), 273-279.
4. Umetani, Y., Kawabe, S., Horikoshi, H. and Okada, T. An Analysis on Applicability of the Vector Operations to Scientific Programs and the Determination of an Effective Instruction Repertoire, *3rd USA-JAPAN Comp. Conf.* (1978), 331-335.
5. Metcalf, M. FORTRAN Optimization, Academic Press, London, New York (1982), 176-181.
6. Pauda, D. A. and Wolfe, M. J. Advanced Compiler Optimizations for Supercomputers, *Comm. ACM* **29**, 12 (1986), 84-1201.
7. Allen, J. R. and Kennedy, K. Automatic Loop Interchange, *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* **19**, 6 (1984), 233-246.
8. Kamiya, S., Isobe, F., Takashima, H. and Takiuchi, M. Practical Vectorizing Techniques for the *FACOM VP, Information Processing* 83 (ed. R. E. A. Manson), North Holland (1983), 389-394.
9. Yasumura, M., Tanaka, Y., Kanada, Y. and Aoyama, A. Compiling Algorithms and Techniques for the S-810 Vector Processor, *ICPP'84* (1984), 285-290.
10. Tsukakoshi, M., Katayama, H., Abe, K. and Yamamoto, K. The Supercomputer SX System: FORTRAN 77/SX and Support Tools, *Proc. of the second International Conference on Supercomputing* **I** (1987), 72-79.
11. Tsuda, T. and Kunieda, Y. Mechanical Vectorization of Multiply Nested DO Loops By Vector Indirect Addressing, *Information Processing* 86 (ed. H. J. Kugler), North Holland (1986), 785-790.
12. Hockney, R. W. and Jesshope, C. R. Parallel Computers, Adam Hilger, Bristol, 1981.
13. Dongarra, J. J. and Eisenstat, S. C. Squeezing the Most out of an Algorithm in CRAY FORTRAN, *ACM Trans. Math. Software* **10**, (1984), 219-230.
14. Dongarra, J. J., Gustavson, F. G. and Karp, A. Implementing linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine, *SIAM Review* **26** (1984), 91-112.
15. Matsuura, T. and Miura, K. Vector Coding for Supercomputers, *Record of the 27th National Convention of Inf. Proc. Society of Japan*, 7P-4, 1983 (in Japanese).
16. Shimasaki, M. Performance Analysis of Vector Supercomputers by Hockney's Model, *Proceedings of the second International Conference on Supercomputing*, **III** (1987), 359-368.

Appendix Analysis of the effect of the length of a vector register

Without loss of generality we can write

$$n=p\cdot n_r+q \quad 0\leqq p,\ 0<q\leqq n_r \tag{A1}$$

If a vector operation with the vector length n is carried out as successive p vector operations with the vector length n_r and the vector operation with the vector length q, then the CPU time for the total operation can be estimated by the following equation:

$$\begin{aligned} t&=p\cdot r_\infty^{-1}(n_r+n_{1/2})+r_\infty^{-1}(q+n_{1/2})\\ &=r_\infty^{-1}\{n+(p+1)\cdot n_{1/2}\} \end{aligned} \tag{A2}$$

Corresponding to Eq. (22), we have

$$\begin{aligned} E'_u&=m(3+5(p+1)n_{1/2}/n)\\ &\quad /\{(m+2)+(3m+2)(p+1)n_{1/2}/n\}\\ &\to m(3+5pn_{1/2}/n)\\ &\quad /\{(m+2)+(3m+2)pn_{1/2}/n\} \quad n\to\infty \end{aligned}$$

From Eq. (A1), we have $p/n=1/n_r-(q/n_r)/n\to 1/n_r$, if $n\to\infty$. And we have

$$\begin{aligned} E'_u&\to m(3+5n_{1/2}/n_r)\\ &\quad /\{(m+2)+(3m+2)n_{1/2}/n_r\} \quad n\to\infty \end{aligned}$$

If $n_{1/2}/n_r$ is small enough compared with 1.0, then $E'_u\approx E_u$.

Let us consider the case where the compiler tries to keep $X(I)$ in a vector register, and $n>n_r$. We can use the loop segmentation technique. If NR is set to n_r, the length of the vector register, the DO loop 30 can be replaced by the following loop:

```
     DO 35 II=1, (N-1)/NR+1
           IIS=(II-1)*NR+1
           IIE=MIN (N, II*NR)
     DO 30 K=1, N
     DO 20 I=IIS, IIE
           X(I)=X(I)+B(I, K)*C(K, J)
20   CONTINUE
30   CONTINUE
35   CONTINUE
```

Corresponding to Eqs. (23) and (24), we have

$$T'_2=2r_\infty^{-1}n[n\{n+2(p+1)n_{1/2}\}+\{n+(p+1)n_{1/2}\}] \tag{23'}$$

and

$$\begin{aligned} T'_{2u}&=r_\infty^{-1}n[(n/2)\{2n+5(p+1)n_{1/2}\}\\ &\quad +\{2n+2(p+1)n_{1/2}\}] \end{aligned} \tag{24'}$$

Therefore in this case we have for the effect of loop unrolling $E'_{2u}=T'_2/T_2u'$

$$\begin{aligned} E'_{2u}&=2[n\{n+2(p+1)n_{1/2}\}+\{n+(p+1)n_{1/2}\}]\\ &\quad /[n\{n+5/2(p+1)n_{1/2}\}+\{2n+2(p+1)n_{1/2}\}]\\ &\to 2\{1+2n_{1/2}(p+1)/n\}/\{1+5/2n_{1/2}(p+1)/n\}\\ &=2-\{n_{1/2}(p+1)/n\}/\{2+5n_{1/2}(p+1)/n\}\\ &\to 2-(n_{1/2}/n_r)/(2+5n_{1/2}/n_r). \end{aligned}$$

In case of FACOM VP-200, $n_r=1024$ and $n_{1/2}\approx 80$ and $n_{1/2}/n_r$ is neglisible small. E'_{2u} tends to 2 as in the case of our previous analysis.

(Received August 26, 1987)