The Journal of Supercomputing, 17, 187–204, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

# Improving Memory Traffic by Assembly-Level Exploitation of Reuses for Vector Registers

CHIH-YUNG CHANG* changcy@email.au.edu.tw
*Department of Computer and Information Science, Aletheia University, 32 Chen-Li St., Tamsui, Taipei, Taiwan*

TZUNG-SHI CHEN chents@mail.cju.edu.tw
*Department of Information Management, Chang Jung University, Tainan, Taiwan*

JANG-PING SHEU sheujp@csie.ncu.edu.tw
*Department of Computer Science and Information Engineering, National Central University, Chung-Li, Taiwan*

Final version accepted December 20, 1999

**Abstract.** In this paper, we propose a compilation scheme to analyze and exploit the implicit reuses of vector register data. According to the reuse analysis, we present a translation strategy that translates the vectorized loops into assembly vector codes with exploitation of vector reuses. Experimental results show that our compilation technique can improve the execution time and traffic between shared memory and vector registers. Techniques discussed here are simple, systematic, and easy to be implemented in the conventional vector compilers or translators to enhance the data locality of vector registers.

**Keywords:** data dependence, vector register, partial reuse, vector compilers, reuse distance, vectorization, supercomputer

## 1. Introduction

In today's supercomputers, memory is organized as a hierarchy which may consist of several levels, ranging from a number of scalar and vector registers, a small and high speed cache, to the shared memory. Given that a program has been written or transformed into a vector form, one of the most critical issues in improving the performance of supercomputer is to decrease the amount of data movement in a memory hierarchy. Much work [2, 3, 7, 8, 11, 14] has been done in automatically transferring loops into a form that reduces the frequency of loading or storing data between different memory levels. A lot of researchers [6, 10] developed prefetching schemes to predict the needed data and prefetch them from memory to cache for improving the performance of data access. Techniques such as *tiling* and *blocking* [11, 14, 15] are the well known transformation schemes that improve the data locality of numerical algorithms. Callahan, Carr, and Kennedy [3] present the

* To whom all correspondence should be addressed.

method of *scalar replacement*, which uses scalar registers to rotate the scalar data for reuse in time. Moreover, Allen and Kennedy [2] report the source-to-source translation scheme to exploit the reuse of vector data in vector register. All these schemes significantly improve the localities of loops execution. Although a lot of effort is being spent on improving the reuse exploitation of vector data, the efficient and effective methods have yet to be developed for fully exploiting the reuse opportunities of vector data.

In loops program, there may exist *partial reuse* opportunities of the vector register data in the execution of successive iteration. That is, only parts of data generated in vector register can be reused in the next computation. For example, elements $A(1:64)$ are generated in vector register $VR1$ by current vector operation and elements $A(2:65)$ will be referenced in the execution of next vector operation. However, only 63 elements $A(2:64)$ of $VR1$ can be reused. The lack of one element $A(65)$ will cause current compiler or assembler to generate a vector load operation which loads 64 elements $A(2:65)$ from main memory to vector register [2]. Allen and Kennedy [2] apply *loop alignment* technique to skillfully exploit the partial reuse existed between two statements. However, two types of partial reuse widely found in application programs are difficult to be exploited up to now. First, the partial reuse opportunities that exist in the same statement of loop program are difficult to be exploited. Second, if there are two or more partial reuses that exist between two statements, techniques such as loop alignment or index shifting [9] can exploit only one of these partial reuses. To save more vector load operations, we develop a scheme to translate vectorized program into assembly code such that the two types of partial reuse can be completely exploited. We discuss how to systematically apply the vector shift operation together with the prefetching technique to exploit the partial reuses. The scheme proposed in this paper can be easily integrated in the current vector compilers design.

The rest of this paper is organized as follows. The primary concept of partial reuse exploitation is briefly discussed in Section 2. A translation scheme is proposed in Section 3 to generate the assembly code whose partial reuse opportunities can be fully exploited. Performance analysis is made in Section 4 to measure the improvement of our compilation scheme. The conclusion will be finally given in Section 5.

## 2. Basic concept of extracting partial reuses of vector data

In this section, we first introduce the *data dependence* [2, 12, 17] for a loop $L$. The basic concept of exploiting partial reuses is then introduced.

In loops execution, two references may access a common set of variables in a manner that requires preserving their relative order. This gives rise to *data dependence*. There are three important classes of dependence identified for two statements $S_1$ and $S_2$ in previous research [2, 17]. Statements $S_1$ and $S_2$ are said to have a *true dependence* if $S_1$ stores into a variable which $S_2$ later uses. If two statements $S_1$ and $S_2$ both store into the same variable, we say that $S_1$ and $S_2$ have an *output dependence*. Statements $S_1$ and $S_2$ are said to have an *input dependence* if $S_1$ and $S_2$ both read the same variable. The dependence of a program represents the statement orderings that must be preserved in order to preserve the valid output of the program. If a variable is generated by $S_1$ in iteration $\bar{i}_1$, and then used by $S_2$ in iteration $\bar{i}_2$, we say that statements $S_1$ and $S_2$ have a *dependence vector* $\bar{d} = (d_1, d_2, \dots, d_n) = \bar{i}_2 - \bar{i}_1$. Dependence vector is used to measure the iteration distance of a true dependence. In this paper, information such as dependence vector, input, output, and true dependence is used to exploit the partial reuse opportunities.

The problem on reuse exploiting is that there may exist *partial reuse* in loop. That is, only parts of data generated in vector register can be reused in the next computation. For example, two references $A(1:64)$ and $A(2:65)$ have partial reuse opportunity. The 64 elements $A(1:64)$ loading from memory to vector register can be partially reused for reference $A(2:65)$. However, only 63 elements $A(2:64)$ can be reused in vector register. The lack of one element $A(65)$ will cause compiler or assembler to generate a vector load operation which loads 64 elements from main memory to vector register. In this paper, we present a method to exploit the partial reuse opportunities of a loop program.

Allen and Kennedy [2] proposed a *loop alignment* technique to exploit partial reuse opportunity existed between different statements. Consider the following vectorized loop program (borrowed from [2])

```
   DO   2 $I = 1, 8$
        $S_1$ :   $A(1:64, I) = B(1:64, I) + 1.0$                    ($L1$)
        $S_2$ :   $C(1:64, I) = A(2:65, I) + 2.0$
 2 CONTINUE
```

Here, the vector data $A(2:65, I)$ loaded in $S_2$ are offset by 1 from the vector data $A(1:64, I)$ stored in $S_1$. The reuse opportunities can be exploited by applying techniques such as *loop alignment* [2] or *index shifting* [9]. The transformed loop program is shown as follows.

```
   DO   2 $I = 1, 8$
        $A(1, I) = B(1, I) + 1.0$
        $S_1$ :   $A(2:64, I) = B(2:64, I) + 1.0$                    ($L1'$)
        $S_2$ :   $C(1:63, I) = A(2:64, I) + 2.0$
        $C(64, I) = A(65, I) + 2.0$
 2 CONTINUE
```

In the transformed loop program $L1'$, a vector load operation from memory to vector register is saved in each iteration.

The partial reuse opportunities that exist in different statement can be exploited by applying loop translation techniques such as loop alignment or index shifting methods. However, if the reuse opportunities are existed in the same statement, these two techniques cannot be applied. Consider the following vectorized loop program.

```
DO    2 J = 2, 65
      S_1 :  A(2 : 65, J) = B(2 : 65, J) * A(1 : 64, J - 1)          (L2)
2   CONTINUE
```

$$S_1: \underbrace{A(2:65,J)}_{VR3} = \underbrace{\underbrace{B(2:65,J)}_{VR1} * \underbrace{A(1:64,J-1)}_{VR2}}_{VR3} \qquad (L2)$$

In loop $L2$, there is a true dependence between references $A(2:65,J)$ and $A(1:64,J-1)$. The *dependence vector* is $\bar{d} = (1, 1)$. The first dimension of array $A$ is vectorized in loop $L2$. A nonzero value of the second dimension of dependence vector indicates that there is a reuse opportunity in statement $S_1$. Moreover, a nonzero value of the first dimension of dependence vector indicates that the reuse type is partial reuse. For example, data elements $A(2:65,2)$ generated in vector register $VR3$ at instance $J = 2$ will be partially reused by vector register $VR2$ at instance $J = 3$. The partially reused 63 elements $A(2:64,2)$ of $VR2$ can be obtained from $VR3$. However, current vector compilers fail to exploit the partial reuse opportunity and will load vector elements $A(1:64,J-1)$ from memory to $VR2$ in each running iteration.

The main idea of partial reuse exploitation is to apply the *vector shift* operation together with the prefetch scheme on vector register to exploit the partial reuses existing in the same or different statements. In what follows, we define the *vector shift* operation which will be used to exploit the partial reuse opportunities.

**Definition 1.** For a given vector computer, assume that a vector register can store 64 array elements. The vector register can be treated as 64 scalar registers. The *vector shift-left* (or *shift-right*) operation *Lshift* $VR(i), VR'$ (or *Rshift* $VR(i), VR'$) is defined by shifting all the data elements in scalar registers of $VR$ to the left (right) $i$ positions and storing the resultant vector data in the corresponding position of vector register $VR'$.

Figure 1 shows an example of the operations *Lshift* $VR(1), VR'$ and *Rshift* $VR(1), VR'$. In Figure 1(a), the *Lshift* $VR(1), VR'$ operation shifts all elements of vector register $VR$ to the left one position and stores the resultant vector data in the corresponding position of vector register $VR'$. Similarly, Figure 1(b) displays the *Rshift* $VR(1), VR'$ operation. We apply the *vector shift* operation together with the prefetch scheme on vector register to exploit all the partial reuses existed

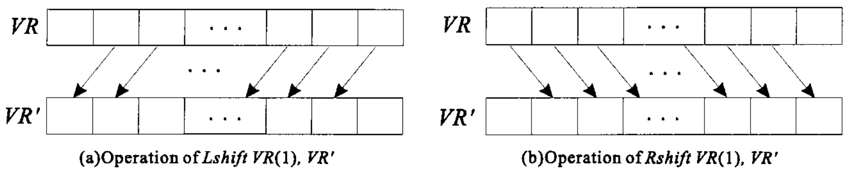
VR
VR'
(a)Operation of Lshift VR(1), VR'
VR
VR'
(b)Operation of Rshift VR(1), VR'

*Figure 1.* An example of vector shift operations.

in the same or different statements. Compilers may predict the missing elements $A(1, J-1)$, for $2 \leq J \leq 65$, and preload them into an extra vector register $VR4$. Then compilers apply the *vector shift* operations to shift elements in vector registers $VR3$ and $VR4$ for supporting the reused elements of $VR2$ in time. Let instruction *Multiply*/$i$ perform vector multiply if the corresponding bit of mask register is $i$, for $i = 0$ or 1. Loop $L2$ can be rewritten as the following form (noted as PRA).

```
/* Partial Reuse of array $A$ exploited (PRA): */
Vector Preload $A(1, 1:64)$ to $VR4$    /* preload the predicted missing elements */
Vector Load $A(1:64, 1)$ to $VR2$
Set mask register to value $00 \cdots 001$
Perform $J$ from 2 to 65 on the following operations:
$OP1$:  Vector Load $B(2:65, J)$ to $VR1$
$OP2$:  Vector Multiply/1 $VR1$ by $VR4$ into $VR3$   /* compute $B(2, J)$
                                                     $*A(1, J-1)$ */
$OP3$:  Rshift $VR4(1), VR4$                          /* shift next predicated
                                                     element for use */
$OP4$:  Vector Multiply/0 $VR1$ by $VR2$ into $VR3$   /* compute $B(3:65, J)$
                                                     $*A(2:64, J-1)$ */
$OP5$:  Lshift $VR3(1), VR2$                          /* 63 elements of $VR2$ are
                                                     supported by $VR3$ */
$OP6$:  Vector Store $VR3$ to $A(2:65, J)$
```

The operations of vector registers are depicted in Figure 2. Data needed for reuse by vector register $VR2$ are then obtained from the register operations including vector preloading, $OP3$, and $OP5$ instead of loading from memory. Assume that there is one Load/Store functional unit. In loop $L2$, the load of elements $B(2:65, J)$ will occupy the Load functional unit. The 64 clock cycles waiting for the available Load/Store functional unit to load array elements $A(1:64, J-1)$ can be reduced to 2 clock cycles in program PRA for executing the operations $OP3$ and $OP5$, in each iteration, in a pipelining and chaining fashion [17]. In program PRA, there are $2 + 64 \times 2 = 130$ Load/Store vector operations in total. Compared to loop $L2$, 1/3 Load/Store vector operations are saved in PRA. Moreover, the reuse exploitation of vector data in $VR2$ reduces the bus traffic and benefits the data accessing of other processors to shared memory.

In the next section, we will propose an algorithm and related data structure to implement the prefetching operation and to exploit the total/partial reuse opportunities. The reuse opportunities with a distance of zero or one iteration can be exploited in the assembly code translation phase.

## 3. Assembly-level transformation for reuse exploitation

In this section, we present the transformation algorithm to exploit reuse opportunities by assembly-level transformation. To illustrate our reuse exploitation algorithm, we define the following two types of distance.

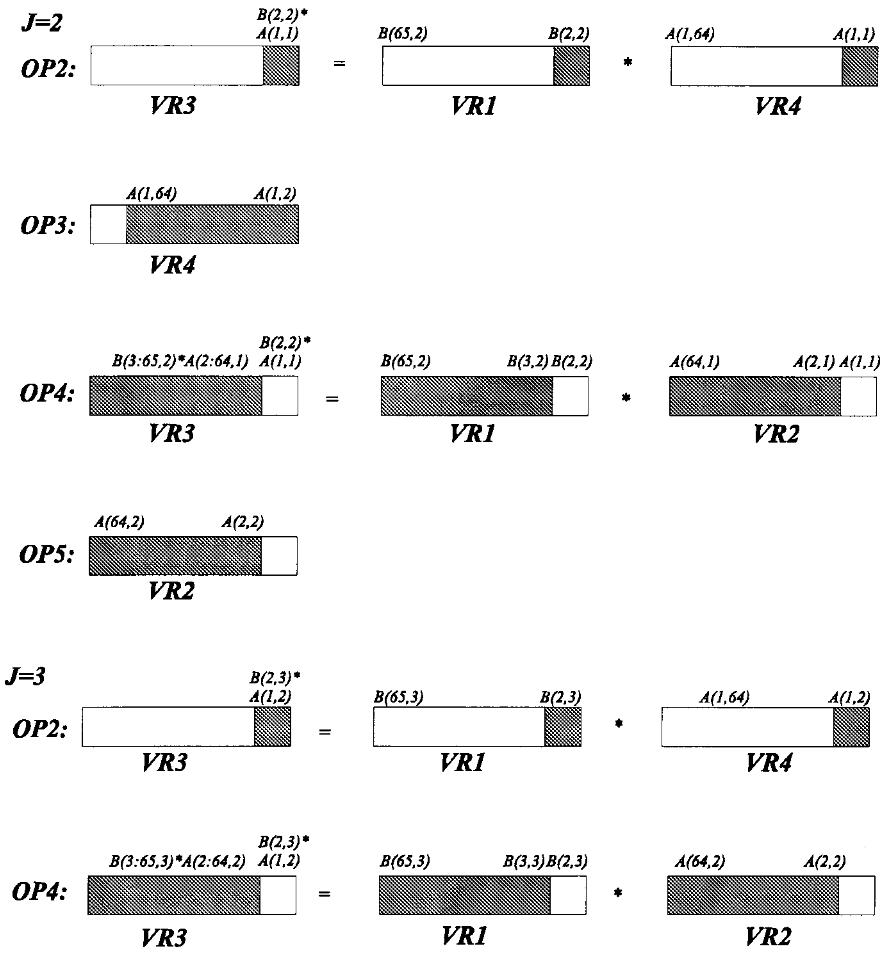

*Figure 2.* Operations of program PRA: combining prefetching and vector shift operations to exploit the partial reuse for loop $L2$.

**Definition 2.** The *missing distance* is defined as the number of missing elements in a partial reuse. The *reuse distance* is defined as the iteration difference between generation and use of the vector elements in a loop program.

Consider the following example.

```
      DO   2   $J = 2, 65$
               $S_1$:  $A(2:65, J) = A(2:65, J-1) * A(1:64, J-1)$
   2  CONTINUE
```

The references $A(2:65, J)$ and $A(2:65, J-1)$ have total reuse opportunity. The *reuse distance* is one since there is one iteration distance between the generation of vector array $A(2:65, J)$ and the use of vector array $A(2:65, J-1)$. Since the reuse type is total reuse, all 64 elements can be found in the vector register after one iteration, provided that the vector data are preserved in vector register. Because there is no missing element, the *missing distance* is zero. For the references $A(2:65, J)$ and $A(1:64, J-1)$, they have partial reuse opportunity. The *reuse distance* is one since there is one iteration difference between the generation of vector array $A(2:65, J)$ and the use of vector array $A(1:64, J-1)$. After one iteration, only 63 elements $A(2:64, J)$ of $A(2:65, J)$ in vector register can be reused by the reference $A(1:64, J-1)$. Since there is one missing element between these two references, the value of *missing distance* is one. To exploit a reuse opportunity, a vector register should be reserved for storing elements for reuse in time. The exploitation of a reuse opportunity whose reuse distance is larger than one has an overhead that one vector register should be scheduled for storing elements for a long time. To prevent the exhaustive use of vector registers provided by vector computer, only those reuse opportunities whose reuse distance is zero or one iteration are considered to be exploited.

The input of our transformation algorithm is a vectorized program together with the dependence vectors of this program. Our exploitation technique consists of two stages. In the first stage, the binding stage, data structure should be created for keeping the binding information of dependence vector and the original vectorized program. Each array reference of vectorized statements in program is assigned a unique label $l$ from 1 to $j$, where $j$ is the number of array references of vectorized statements in program. For each dependence vector, the *dependence structure* $DS = (S, D,$ *type*, *dependence vector*$)$ containing $(n+3)$ elements should be constructed to present the reuse information. The first two elements, $S$ and $D$, of *dependence structure* denote labels of the references of source and destination respectively, associated to the dependence relation. The value of the third element is one of the three symbols $i$, $t$ or $o$, denoting the dependence type of this dependence vector is input, true, or output dependence respectively. The last $n$ elements record values of the dependence vector. In the second stage, we will transform the program into assembly code to explicitly exploit the reuse opportunities, including total reuse and partial reuse.

Each dependence structure will be classified into one of the following five reuse cases: (1) total reuse with reuse distance of zero iteration, (2) partial reuse with reuse distance of zero iteration, (3) total reuse with reuse distance of one iteration, (4) partial reuse with reuse distance of one iteration, and (5) reuse with reuse distance larger than one iteration. Here, we only consider the reuse exploitation in cases (1) to (4). The classification of reuse type for compilers is easy. For checking the reuse of a $DS$ is total or partial reuse, compilers only need to check the value of element in vectorized dimension of the dependence vector in $DS$ is zero or not, respectively. The distance of reuse can be examined by checking the value of nonvectorized elements of dependence vector in $DS$.

In the second stage, the transformation stage, compilers transform the program into assembly code by explicitly exploiting the reuse opportunities. Consider the dependence structure $DS = (S, D,$ *type*, *dependence vector*$)$. Let $S$ and $D$ be the labels of references of arrays $A(f(I^n))$ and $A(g(I^n))$, respectively, where $I^n$ is the iteration space $(I_1, I_2, \ldots, I_n)$ and $f$ and $g$ are intrinsic functions. Let $Reg(S)$

and $Reg(D)$ respectively denote the vector registers assigned to references $A(f(I^n))$ and $A(g(I^n))$. The transformation algorithm for 4 cases of reuse is now described in the following.

### *(1) The total reuse with reuse distance of zero iteration*

Examples of this case can be considered as the following three loops.

(a) type "$o$"
```
    DO 1 $J = 1, 64$
        $\underbrace{A(1:64, J)}_{Reg(S)} = \cdots$
        $\underbrace{A(1:64, J)}_{Reg(D)} = \cdots$
1   CONTINUE
```

(b) type "$i$"
```
    DO 2 $J = 1, 64$
        $\cdots = \underbrace{A(1:64, J)}_{Reg(S)}$
        $\cdots = \underbrace{A(1:64, J)}_{Reg(D)}$
2   CONTINUE
```

(c) type "$t$"
```
    DO 3 $J = 1, 64$
        $\underbrace{A(1:64, J)}_{Reg(S)} = \cdots$
        $\cdots = \underbrace{A(1:64, J)}_{Reg(D)}$
3   CONTINUE
```

The following reuse exploiting steps can be integrated in the vector compilers.

```
If the type is "$o$."        /* output dependence */
    Omit the Store operation of register $Reg(S)$ to $A(f(I^n))$.
else                         /* input or true dependence */
    Omit the Load operation from $A(g(I^n))$ to register $Reg(D)$.
    Replace $Reg(D)$ by $Reg(S)$ in assembly program.
endif
```

### *(2) The partial reuse with reuse distance of zero iteration*

The exploitation of output dependence with partial reuse is not discussed here since the benefit is not significant in implementation. The partial reuse is exploited only if the dependence type is input or true dependence. Let us consider the following fragments of loop with partial reuse.

(a) type "$t$"
```
    DO 1 $J = 1, 64$
        $\underbrace{A(2:65, J)}_{Reg(S)} = \cdots$
        $\cdots = \underbrace{A(1:64, J)}_{Reg(D)}$ op ...
1   CONTINUE
```

(b) type "$i$"
```
    DO 2 $J = 1, 64$
        $\cdots = \underbrace{A(2:65, J)}_{Reg(S)} + \underbrace{A(1:64, J)}_{Reg(D)}$ op ...
2   CONTINUE
```

Compilers should verify the vectorized dimension of dependence structure to predict the missing elements during the running iterations ranging from 1 to 64. In this example, the set of missing elements is $M = \{A(1, J) \mid 1 \le J \le 64\} = A(1, 1:64)$. The compilers then assign a vector register, say $Reg(M)$, to prefetch the missing elements and prepare the right shift operation to support one missing element for use in each running iteration. The following reuse exploiting steps can be included in the compilers to handle this case of reuse:

```
Insert a prefetch statement "Load $M$ into vector register $Reg(M)$" before the
   loop head.
Omit the operation "Load $A(g(I^n))$ to $Reg(D)$" in loop body.
Replace the arithmetic operation "op" by the following operations:
   Lshift $Reg(S)(1), Reg(D)$.
   Set mask to $0\cdots01$.
   Operate/0 $Reg(D)$ with $\cdots$
   Operate/1 $Reg(M)$ with $\cdots$
   Rshift $Reg(M)(1), Reg(M)$.
   Set mask to $1\cdots11$.
```

For those reuses with missing distance 2, we may prepare two vector registers; each preloads 64 elements with a stride value 2. Another approach to save the number of vector registers is that compilers may prepare one vector register to preload two times of 64 elements. In each time, the 64 elements are prefetched for supporting 32 iterations. In order to avoid the exhaustive use of vector registers, we do not consider the number of missing elements larger than 2 in each running iteration. Similar discussion can be made in the case that the missing elements are in the highest positions of vector register, that is, $M = \{A(65, J) \mid 1 \le J \le 64\}$.

*(3) The total reuse with reuse distance of one iteration*

The following reuse exploiting steps can be integrated in the compilers:

```
If the type is "o."        /* output dependence */
    Omit the Store operation of register $Reg(S)$ to $A(f(I^n))$.
else                       /* input or true dependence */
    Insert "Load $A(g(I_0^n))$ to $Reg(D)$" before loop head,
      where $I_0^n$ is the first running iteration index in loops.
    Omit the Load operation from $A(g(I^n))$ to register $Reg(D)$ in loop body.
    Insert statement "rotate $Reg(S)$ to $Reg(D)$" in loop tail.
endif
```

*(4) The partial reuse with reuse distance of one iteration*

Let us consider the following fragments of program.

(a) type "*t*"

DO 1 $J = 2, 65$

$\cdots = \underbrace{A(1:64, J-1)}_{Reg(D)}$ op ...

$\underbrace{A(2:65, J)}_{Reg(S)} = \underbrace{A(1:64, J-1)}_{Reg(D)}$

op...

(b) type "*i*"

DO 2 $J = 2, 65$

$\cdots = \underbrace{A(1:64, J-1)}_{Reg(D)}$ op ...

$\cdots = \underbrace{A(2:65, J)}_{Reg(S)} + \underbrace{A(1:64, J-1)}_{Reg(D)}$

op...

$\cdots = \underbrace{A(1:64, J-1)}_{Reg(D)}$ op ... $\qquad \cdots = \underbrace{A(1:64, J-1)}_{Reg(D)}$ op ...

```
1   CONTINUE                    2   CONTINUE
```

In this example, the set of missing elements is $M = \{A(1, J-1) \mid 2 \leq J \leq 65\} = A(1, 1:64)$. The following reuse exploiting steps can be integrated in the compilers:

Insert a prefetch statement "Load $M$ into vector register $Reg(M)$" before the loop head.
Insert an initial load statement "Load $A(g(I_0^n))$ to $Reg(D)$" before the loop head.
Omit the operation "Load $A(g(I^n))$ to $Reg(D)$" in loop body.
Replace the arithmetic operation "op" by the following operations:
    Set mask to $0\cdots01$.
    Operate/1 $Reg(M)$ with $\cdots$
    Insert statement "Rshift $Reg(M)(1), Reg(M)$."
    Operate/0 $Reg(D)$ with $\cdots$
    Set mask to $1\cdots11$.
    Insert statement "Lshift $Reg(S)(1), Reg(D)$" to loop tail.

The consideration of *missing distance* larger than one is the same as partial reuse with distance zero. Consider the example of loop $L2$ introduced in Section 2. Program PRA can be obtained by applying case (4) transformation to loop $L2$.

When there are two or more partial reuses, it seems that exploiting each partial reuse needs to pay the cost of two vector shift and three mask setting instructions on saving one vector load. In fact the consecutive pairs of mask setting instructions can be saved. All the arithmetic operations can be done by sharing the first pair of mask setting instructions. Thus, only one pair of mask setting instructions is needed for two or more consecutive partial reuse exploiting.

For a given vectorized program, compilers may firstly analyze the dependence relation and evaluate the dependence vector. Then the partial and total reuses with reuse distance zero or one iteration gap can be explicitly exploited in assembly code by the described transformation algorithm. In the next section, performance analysis for the proposed reuse exploiting technique is presented. The performance improvement by applying the proposed reuse exploiting strategy on real applications is also discussed.

## 4. Performance analysis

The advantage of total reuse exploitation is obvious since the Load/Store operations can be saved without any extra overhead. The performance improvement of total reuse exploitation is measured on Convex C3840 supercomputer. Two versions of assembly code are compared. For the first version, we take original program written in Fortran 77 as the input source of vector compiler of Convex. The vectorized assembly code automatically generated by vector compiler of Convex is referred to the *original version*. To generate the second version, we exploit the total reuse opportunities existed in the input source by applying cases (1) and (3) algorithms proposed in Section 3. Then, we obtain another version referred to the *reuse exploited version*.

Loops selected as the source can be roughly cataloged into three classes. The first class is the vector benchmark that is extracted from NETLIB of NCHC (National Center for High Performance Computing). The selected benchmark consists of 107 subroutines of loops that are originally designed for testing the vectorization capability of PFC [1]. In total, 65 subroutines can be vectorized by vector compiler of Convex. The 65 subroutines are considered as the source and the execution time of two versions, the original version and the reuse exploited version, is compared. In total, 11 subroutines are improved by applying the proposed scheme.

The second class, selected as source, consists of several subroutines of BLAS2 (Basic Linear Algebra Subprograms) which perform the matrix/vector operations. All subroutines of BLAS2 are designed in libraries for calls in most vector computers. The subroutines of BLAS2 used as the source are also stored in NETLIB of NCHC. In addition, the third class consists of several application programs. Most of these programs are selected from the numerical computation programs [13, 16]. We list the abbreviation of the application programs in Table 1.

The experimental results of execution time and speedup for these classes of programs are summarized in Table 2. Only subroutines of benchmark that have total reuse opportunities are listed in the table. Compared with the original version, the total reuse exploitation reduces the number of loads from shared memory to vector registers and frees the load/store functional units for other use. For a column-major based compiler such as Fortran, successive elements of array data are stored in memory in a column-major fashion. Assume that there is a declaration $A(128, 128)$ in a vectorized program. Consider two references $A(1, 1:128)$ and $A(1:128, 1)$ that are respectively vectorized in the second and the first dimension. The time cost for moving 128 elements $A(1, 1), A(1, 2), \ldots,$ and $A(1, 128)$ from memory to vector register is much more than one for moving 128 elements $A(1, 1), A(2, 1), \ldots,$ and $A(128, 1)$. This is because that elements $A(1, 1:128)$ have a memory stride of 128 whereas elements $A(1:128, 1)$ have a memory stride

*Table 1.* Applications and their abbreviations

| Application | Abbreviation |
|---|---|
| Fourier least-squares approximation [13] | FLSA |
| Jacobi method for solution of linear equations [13] | Jacobi |
| Barycentric form of Lagrange interpolation [13] | BFLI |
| Accumulating a sum (A. Sum) [16] | ASum |
| Solving linear equation by Gaussian elimination (loop 1) [16] | SLEGE1 |
| Solving linear equation by Gaussian elimination (loop 2) [16] | SLEGE2 |
| Computing the uniform norm of matrix $A$ and $A$ inverse [16] | Uniform norm |
| Gauss-Seidel iterations [16] | GSI |
| Comparing compound Simposon's and Newton-Cotes integration [16] | CSNCI |
| Computing the value of a filtered discrete Fourier transform [16] | FDFT |

*Table 2.* Comparisons of original version and reuse exploited version for benchmark, libraries and applications

| Loop programs | Problem size | Original version (ms) | Reuse exploited version (ms) | Speedup |
|---|---|---|---|---|
| Benchmark | | | | |
| S022 | 1024 × 1024 | 8.5 | 5.3 | 1.6 |
| S023 | 1024 × 1024 | 12.8 | 9.5 | 1.3 |
| S029 | 1024 × 1024 | 7.5 | 0.3 | 25 |
| S030 | 1024 × 1024 | 23.88 | 3.64 | 6.56 |
| S044 | 1048576 | 5.55 | 3.0 | 1.85 |
| S045 | 1048576 | 8.51 | 5.10 | 1.66 |
| S047 | 1024 × 1024 | 7.12 | 5.72 | 1.24 |
| S048 | 1024 × 1024 | 6.99 | 5.75 | 1.22 |
| S049 | 1024 × 1024 | 6.96 | 5.63 | 1.23 |
| S084 | 1024 × 1024 | 23.9 | 1.58 | 15.1 |
| S100 | 1024 × 1024 | 219 | 153 | 1.43 |
| BLAS2 | | | | |
| SGEMV | 1024 × 1024 | 6 | 5 | 1.2 |
| SGBMV | 1024 × 1024 | 6 | 5 | 1.2 |
| DGEMV | 1024 × 1024 | 11.5 | 9.8 | 1.17 |
| DGBMV | 1024 × 1024 | 12.1 | 9.9 | 1.2 |
| Applications | | | | |
| FLSA | 1024 × 1024 | 32.1 | 5.1 | 6.29 |
| Jacobi | 1024 × 1024 | 16 | 13 | 1.23 |
| BFLI | 1024 × 1024 | 31 | 27 | 1.15 |
| ASum | 1024 | 3.8 | 2.6 | 1.46 |
| SLEGE1 | 1024 × 1024 | 7.8 | 6.5 | 1.2 |
| SLEGE2 | 1024 × 1024 | 55 | 7 | 7.85 |
| Uniform norm | 1024 × 1024 | 74 | 58 | 1.27 |
| GSI | 1024 × 1024 | 6.3 | 4.8 | 1.31 |
| CSNCI | 1024 × 1024 | 53 | 5.7 | 9.29 |
| FDFT | 1024 × 1024 | 57 | 5 | 11.4 |

one. A vector load operation for elements with a larger stride will spend much more time than vector load operation for elements with a stride one [5]. Exploiting the reuse opportunity of a reference that is vectorized in the second dimension will save a vector load operation and have a significant improvement in time cost. This effect can be found in speedup of benchmarks S030, S084, FLSA, SLEGE2, CSNCI, and FDFT.

In the partial reuse exploitation, compilers use vector shift operations to save Load operations. The time cost of vector shift operation is less than one of the load operation since that the vector-to-vector operation is faster than memory-to-vector operation. The time cost of load operation depends on various factors including the data format defined in program, the stride of data accessing determined by vectorization, the number of Load/Store function units in system, the number of consecutive load and store operations in program, the number of memory banks for load interleaving, and whether the operation followed by load can be chained or not. In Convex C3840, there is one load/store functional unit supported by system.

The memory is organized as 32 banks for interleaving. The comparison for vector shift and load operation in time cost is made in Convex C3840 by using the following 4 types of program to show that vector shift operation is cost effective.

```
(1) Pure load
      Perform $J$ from 2 to 129 on the following operation:
        Load $A(2:129, J)$ to $VR0$   /* use shift and preloading operations
                                        to compare with */

(2) A load of the same array followed by another load of partial reuse array
      Perform $J$ from 2 to 129 on the following operations:
        Load $A(1:128, J)$ to $VR0$
        Load $A(2:129, J)$ to $VR1$   /* use shift and preloading operations
                                        to compare with */

(3) A load of partial reuse elements followed by a store operation
      Perform $J$ from 2 to 129 on the following operations:
        Load $A(2:129, J-1)$          /* use shift and preloading operations
          to $VR0$                      to compare with */
        Store $A(1:128, J)$ from $VR1$

(4) Two consecutive loads followed by a store
      Perform $J$ from 2 to 129 on the following operations:
        Load $B(2:129, J)$ to $VR0$
        Load $A(2:129, J-1)$          /* use shift and preloading operations
          to $VR1$                      to compare with */
        Store $A(1:128, J)$ from $VR2$
```

The reason why we take these 4 types of program to measure the comparison in time cost for vector shift and vector load operations in application programs is described in the following. Consider the following program that contains a partial reuse introduced by an input dependence vector.

```
  DO  1 $J = 2, 129$
        ...
        $OP1$: Load $A(1:128, J)$ to $VR0$
        ...
        $OP2$: Load $A(2:129, J)$ to $VR1$
        ...
1 CONTINUE
```

If there are many arithmetic or logical operations *OP*s between *OP*1 and *OP*2, the occupying of load functional unit by performing *OP*1 has no effect to *OP*2. Then, exploiting partial reuse that uses vector shift and preloading operations instead of *OP*2 can be measured by program type (1). On the other hand, if the *OP*2 has a bottleneck on the lacking of load/store functional unit, the exploitation of partial reuse can be measured by program type (2).

If the partial reuse occurs in a true dependence vector in which vector elements used by a statement are partially generated by another statement, the similar discussion can be made and we may measure them by program types (1), (3), or (4). Two classes, one is vectorized in the first dimension and another is vectorized in the second dimension, of the above 4 program types are measured in Convex C3840 supercomputer. We compare the above 4 program types with the corresponding partial reuse exploited program. Each program is running 50,000 times controlled by an outermost loop to scale their execution time measured in seconds. The comparisons in time cost for load operation (original program noted by origin) and vector shift and preloading operations (partial reuse exploited program noted by p.r.e.) are depicted in Table 3. As shown in Table 3, performance improvement on loops vectorized in the second dimension is more significant than one vectorized in the first dimension due to the fact that the time cost of load operation is higher when stride is lengthened.

In what follows, two examples are considered to illustrate that the combination of total reuse and partial reuse exploitation has significant improvement in execution time. Consider the following example.

```
  DO        1 $I = 2,129$
              $A(I, 2:129) = A(I-1, 3:130)$
                             $+A(I-1, 2:129) * B(I, 2:129)$          ($L3$)
1 CONTINUE
```

In loop $L3$, there exists total reuse between references $A(I, 2:129)$ and $A(I-1, 2:129)$ and partial reuse between references $A(I, 2:129)$ and $A(I-1, 3:130)$. Loop $L3$ is first compiled to assembly code, noted by Noreuse.s, by compiler of Convex C3840 supercomputer. Then, we modify the assembly code Noreuse.s into a total reuse exploited assembly code, noted by Total.s, according to the translation algorithm of case (3) discussed in Section 3. Finally, we further rewrite the Total.s into TotPar.s by the translation algorithm of case (4) discussed in Section 3 such that both the total and partial reuses are exploited. The performance improvements of Total.s and TotPar.s, compared with Noreuse.s, are 31.5% and 53.8% respectively.

*Table 3.* Comparisons of 4 program types with and without partial reuse exploited

| | 1th dim. vectorized | | | 2nd dim. vectorized | | |
|---|---|---|---|---|---|---|
| | Origin (sec) | p.r.e. (sec) | Improvement (%) | Origin (sec) | p.r.e. (sec) | Improvement (%) |
| Type (1) | 16.50 | 15.53 | 5.88 | 24.96 | 15.60 | 37.50 |
| Type (2) | 33.39 | 30.54 | 8.54 | 55.87 | 37.68 | 32.56 |
| Type (3) | 35.07 | 33.27 | 5.13 | 49.23 | 40.10 | 18.55 |
| Type (4) | 52.42 | 48.27 | 7.91 | 78.81 | 60.61 | 23.09 |

We also consider the example of the sequential SOR loop program (borrowed from [8, 14, 15]) as shown in follows.

```
      DO  1 $I = 2, n - 1$
          DO  2 $J = 2, m - 1$
              $A(I, J) = 0.2 * (A(I - 1, J) + A(I, J - 1)$
                             $+ A(I, J) + A(I + 1, J) + A(I, J + 1))$
  2       CONTINUE
  1   CONTINUE
```

Two dependence vectors (0, 1) and (1, 0) can be found as shown in Figure 3. The sequential loop program can be vectorized into the following loop form by applying *loop skewing* together with *loop interchange* techniques [14, 15] or supernode partitioning [8].

```
      DO  1 $J = 4, n + m - 2$
          C$DIR FORCE_VECTOR
          DO  2 $I = \max(2, J - m + 1), \min(n - 1, J - 2))$
              $A(I, J - I) = 0.2 * (A(I - 1, J - I) + A(I, J - I - 1)$
                              $+ A(I, J - I) + A(I + 1, J - I)$
                              $+ A(I, J - I + 1))$                    (L4)
  2       CONTINUE
  1   CONTINUE
```

where C$DIR FORCE_VECTOR denotes the compiler directive instruction that specifies the vectorization of loop $I$. Dependence vectors (0, 1) and (1, 0) originally existing in sequential SOR program are thus respectively transformed into (0, 1) and (1, 1) in $L4$ as shown in Figure 4. Since the vector length of Convex C3840 is 128, we apply strip mining [17] technique to partition loop $I$ into several bands with size $128 * (m - 2)$. Within each band, vector operations can be performed along

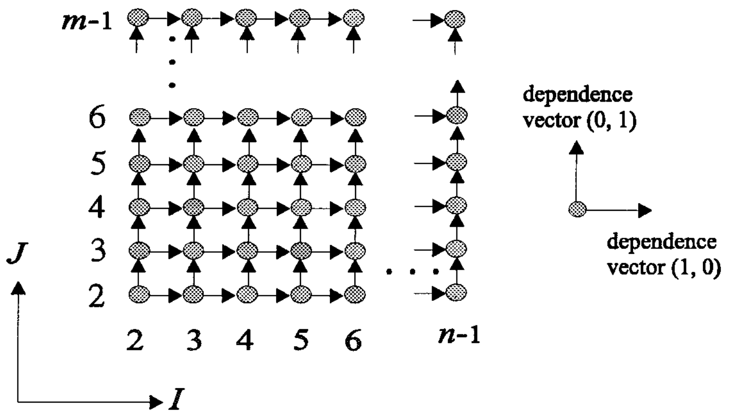

*Figure 3.* Dependence among iterations of sequential SOR algorithm.

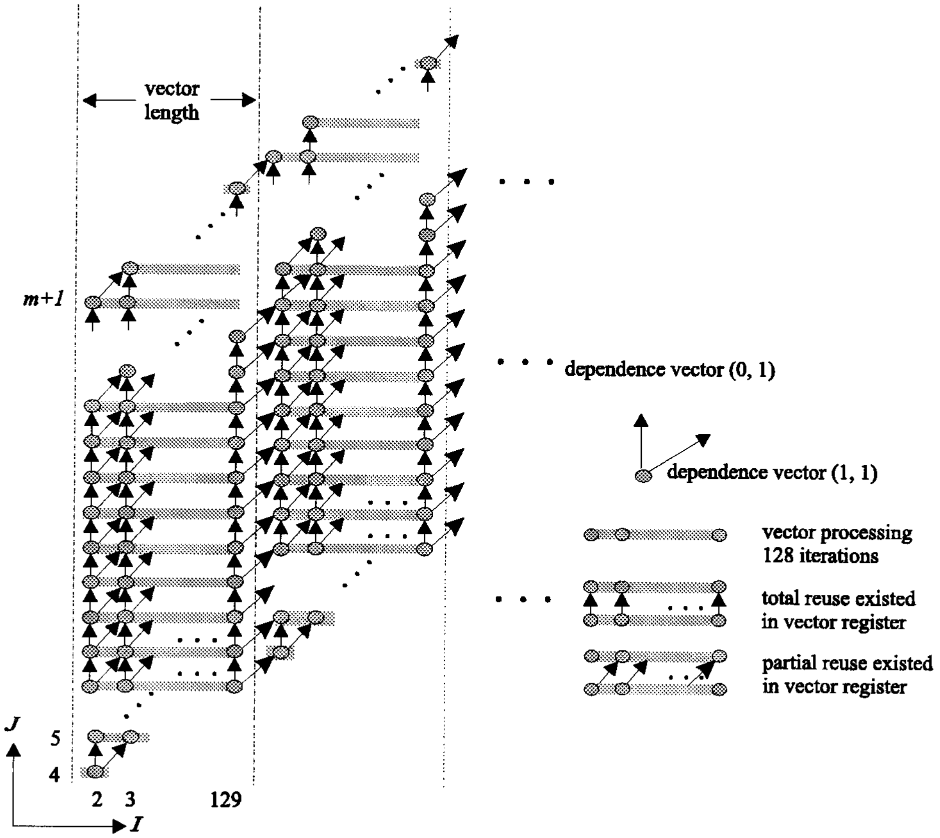
vector length
m+1
dependence vector (0, 1)
dependence vector (1, 1)
vector processing 128 iterations
total reuse existed in vector register
partial reuse existed in vector register
J
5
4
2 3 129
I

*Figure 4.* Dependence among iterations of vectorized SOR program.

dimension $J$ as shown in Figure 4. The total and partial reuse opportunities can be found between two contiguous vector operations.

Previous studies [2, 8] can exploit the total reuse opportunity with a reuse distance of one iteration. The partial reuse that occurs in the same statement cannot be exploited by applying their reuse exploitation scheme. The proposed techniques in this paper can exploit not only the total reuses but also the partial reuses existed in original loop program. In addition, all the partial reuses existing in the same or different statements can be exploited by applying our proposed techniques.

We perform loop $L4$ on Convex C3840 in three different versions of assembly code. The first version of assembly code, denoted by Sor.s, is generated by compiling $L3$ on vector compiler of Convex C3840. The Convex compiler fails to exploit both the total and partial reuses existed in $L4$. Then, we apply techniques proposed in [2, 8] to exploit the total reuse opportunity in $L4$ and generate the second version, denoted by SorTot.s, of assembly code of $L4$. The third version, denoted by SorTotPa.s, is generated by applying our translation algorithm of cases 3 and 4 discussed in Section 3 to exploit both the total and partial reuses existed in $L4$. Three versions, Sor.s, SorTot.s, and SorTotPa.s, of assembly code are running on

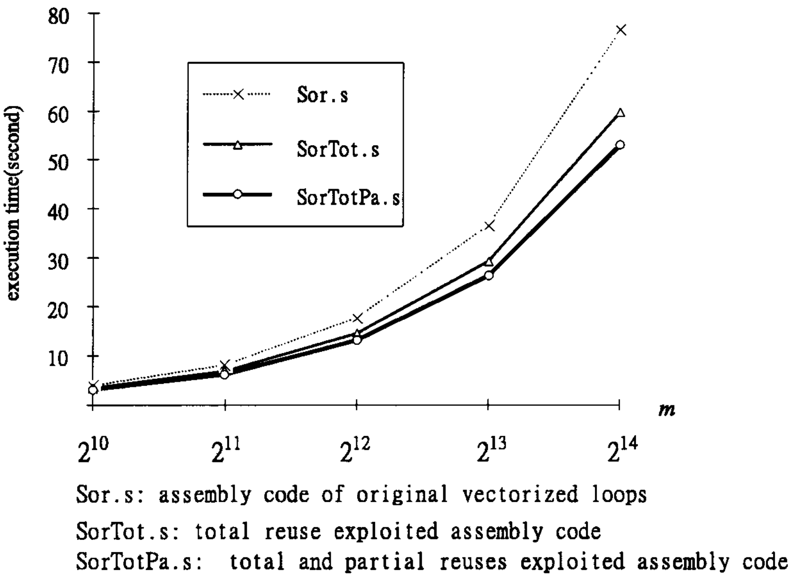

*Figure 5.* Execution time of three versions of SOR algorithm.

Convex C3840 supercomputer to evaluate the improvement of the proposed strategies. We set the value of $n$ to $2^{10}$ and vary the value of $m$ ranging from $2^{10}$ to $2^{14}$. As shown in Figure 5, the larger the value of $m$ we set, the more the improvement we gain. This is due to the fact that the number of load operations saved by exploiting total and partial reuses is increased when value of $m$ is increased. Compared to Sor.s, in average, the improvements of the SorTot.s and SorTotPa.s are respective 18.54% and 26.19% in execution time. The exploitation of partial reuse in SorTotPa.s additionally saves 7.65% of total execution time compared to SorTot.s.

## 5. Conclusions

In this paper, we develop compilation techniques to exploit the total and partial vector reuses. By analyzing data dependence relation of a vectorized loop program, we derive the set of missing elements of a partial reuse and prefetch the set into vector register for use in time. By applying the vector shift operation, we exploit the partial reuse opportunities such that vector load operations can be saved. An assembly-level translation algorithm is developed to transform the vectorized loop program into reuse exploited form. Techniques discussed in this paper are simple, systematic, and easy to be implemented or integrated with conventional vector compilers, assembler, or translator. Performance analysis and experimental results show that the proposed methods make significant improvements in execution time and bus traffic between vector registers and memory.

## Acknowledgments

This work was supported by the National Science Council of the Republic of China under grant NSC 88-2213-E-156-002.

## References

1. R. Allen and K. Kennedy. PFC: a program to convert Fortran to parallel form. In *Proceedings of IBM Conference on Parallel Computing and Scientific Computation*, 1982.
2. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, 1992.
3. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 53–65, June 1990.
4. H. Cheng. Vector pipelining, chaining, and speed on the IBM 3090 and Cray X-MP. *IEEE Computer*, 10:31–46, 1989.
5. Convex. *CONVEX FORTRAN Optimization Guide*. CONVEX Computer Corporation, Richardson, TX, 1990.
6. F. Dahlgren and P. Stenstrom. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pp. 385–398, April 1996.
7. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, 1988.
8. F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 319–329, January 1988.
9. L. S. Liu, C. W. Ho, and J. P. Sheu. On the parallelism of nested for-loops using index shift method. In *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. II, pp. 119–123, August 1990.
10. N. Manjikian. Compiling loop fusion with prefetching on shared-memory multiprocessors. In *Proceedings of 1997 International Conference on Parallel Processing*, pp. 78–82, 1990.
11. N. Mitchell, L. Carter, J. Ferrante, and K. Hogstedt. Quantifying the multi-level nature of tiling iterations. In *The 10th International Workshop on Languages and Compilers for Parallel Computing*, pp. 1–15, 1997.
12. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.
13. S. M. Pizer and V. L. Wallace. *To Compute Numerically Concepts and Strategies*. Little, Brown and Company, Boston, 1993.
14. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.
15. M. J. Wolfe. More iteration space tiling. *Proceedings of the ACM International Conference on Supercomputing*, pp. 655–664, November 1989.
16. S. Yakowitz and F. Szidarovszky. *An Introduction to Numerical Computations*. 2nd ed. Macmillan Publishing Company, New York, 1989.
17. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley Publishing Company, Reading, Mass., 1990.