DESIGN AND IMPLEMENTATION OF A FORTRAN ASSISTANT TOOL FOR VECTOR COMPILERS

CHIH-YUNG CHANG, JIANN-YUAN TZENG and JANG-PING SHEU*

Department of Computer Science and Information Engineering
National Central University, Chung-Li 32054, TAIWAN
sheujp@csie.ncu.edu.tw
FAX: 886-3-4222681

ABSTRACT

In this paper, we present the design and implementation of source-to-source High Performance Fortran assistant Tool (HPFT) in DEC 3000 workstations. For a given sequential program written in Fortran 77, HPFT generates a vectorized, reuse exploited, and/or parallelized version for vector computers. Several new compilation schemes in vectorization, reuse exploitation, and multi-threading are designed in HPFT. Performance evaluator is developed for measuring the system performance. The user interface is also designed for programmer to capture the information related to the compilation and execution of program. Experimental results based on the Convex C3840 vector computer show that the developed HPFT enhances the system performance and usually reduces the program execution time.

Keywords: Data dependence, loop optimization, vector compilers, vector register reuse.

Short title: Will be used by the Publisher as running head.

1. Introduction. Vector computers such as Cray family and Convex series are equipped with hierarchical memory and several CPUs to speed up the program execution. Multiple CPUs can work together to parallelize the execution of vector operations that are defined by programmer or vector compilers. When multiple CPUs concurrently process a program, the parallelism of execution appears in vectorization and parallelization. Synchronizations are needed among these CPUs if their references of array data have dependence relation. The parallelism, memory

* To whom all correspondence should be addressed.

management, and synchronizations are the main factors for determining the execution time of a sequential program. To efficiently utilize the hardware design, these factors should be highly regarded.

In most vector computers, vector compilers are able to analyze the dependence relation existed in program and automatically perform the vectorization and parallelization. However, implicit vector operations or parallelism existed in programs can not be fully exploited by vector compilers. Most current compilers [10, 11, 14] provide compiler directives for user to manually define vector operations and multi-thread in program. An improper use of these directives will cause not only semantic errors but also inefficient execution. Unless that users are skilled in parallel program design, it is difficult to write an efficient program with explicit definitions of vector and/or parallel operations.

Another critical issue to improve the system performance of supercomputers is to reduce the data movements between shared memory and vector registers. Exploiting reuse opportunities of vector register data not only reduces the workload of load/store functional units but also gains the following three advantages. First, arithmetic functional units may avoid waiting for load/store operation and have earlier startup time. Second, time spent on load operations can be saved. The execution time of program is thus significantly improved. Third, fewer load/store operations reduce the traffic of shared memory accessing. Most current compilers, however, are not capable of exploiting the implicit reuse opportunities. This motivates us to design and implement a High Performance Fortran assistant Tool (HPFT) that assists vector compilers in vectorization, reuse exploitation, and synchronization reduction.

Related translators developed in last decade can be found such as Parafrase [18], PFC [2, 3], and SUPERB [23]. Parafrase is a source-to-source translator applied to a Fortran 66 or Fortran 77 program. The system can be retargeted to produce code for different classes of concurrent architectures, including register-to-register and memory-to-memory vector machines, array processors, and shared-memory multiprocessors. The PFC is a source-to-source vectorizer that translates from Fortran 66 or 77 into Fortran 90. Standard transformations are applied in PFC. These include if-conversion, induction variable recognition and substitution, constant propagation and deletion of dead code [2]. SUPERB was developed by Zima and coworkers. It translates Fortran 77 programs into concurrent SUPRENUM Fortran programs for the SUPRENUM machine [23].

Different to these designs, the HPFT is an assistant tool to vectorize, parallelize, and exploit reuse opportunities for vector compilers. HPFT performs the π-*block* decomposition such that maximum benefits can be gained for vectorization and reuse exploitation. In the π-*block* decomposition and vectorization phase, HPFT determines a vectorization vector for each statement in loop body. The vectorization vector is then used to reconstruct the parsing tree such that vector operations are automatically defined. Several procedures are designed and implemented in HPFT to reconstruct the vectorized loop such that maximum benefits can be obtained from the exploitation of the reuse opportunities of vector data. In addition, HPFT provides the multi-threading technique to partition the exploited vector operations into multi-thread such that the reuse opportunities are still preserved and the synchronizations are as few as possible. Sequential program thus

can be translated into a high performance code.

Before execution, performance evaluator is designed for evaluating the performance of the translated code including degree of vectorization, reuse exploitation, and parallelization. In addition, after execution, it summarizes the information including the execution time and speedup. A user interface is also developed for user to easily set options provided by HPFT system and to capture the information offered by performance evaluator. Several benchmarks, libraries, and scientific application programs are taken as the input for measuring the performance improvement. Experimental results based on the Convex C3840 vector computer show that our implementation enhances the system performance and usually reduces the program execution time.

2. Design overview. The HPFT system has been implemented using C language in an X-window based environment. It consists of three main parts: the HPFT kernel, the user interface, and the performance evaluator. The HPFT kernel consists of several modules including the dependence analysis module, the π-*block* decomposition and vectorization module [8], the reuse exploitation module [8], and the multi-threading module [9].

The HPFT kernel is a source-to-source translator, which is implemented in DEC 3000 workstations. It can be roughly divided into six phases as shown in Fig. 1. The outputs of each phase are stored in database and will be the input of the next phase. In the first phase, sequential program written in Fortran 77 is taken as the input and syntax analysis is made to construct the parsing tree. HPFT uses the abstract syntax tree as the intermediate representation. Several procedures are designed for manipulating the abstract syntax tree. Similar design of syntax tree of Adaptor tool [6] is utilized in HPFT. The *Adaptor* is a source-to-source translator designed mainly for message passing system. It transforms data parallel programs written in Fortran into host and node programs with message passing that can be executed on most parallel architectures. We make modification of syntax tree design of Adaptor to guarantee that parsing tree of a source program written in standard Fortran 77 can be generated.

After the syntax analysis, in the second phase, data dependence relation is investigated. In this phase, the GCD [24] and Banerjee [5] tests are used as the principal dependence tests. Then, HPFT constructs the *extended dependence graph* (EDG) for use in latter phases. The EDG is a graphical representation of dependence restriction existed in original program. Different from the conventional data dependence graph [24, 19] (DG), the constructed EDG additionally captures the statement order for each dependence relation existed in program. In latter phase, vectorization can be processed independently for each statement by examining the EDG. In the third phase, HPFT decomposes EDG into several strongly connected components. Then, HPFT extracts vector operations and combines the components that have reuse opportunities into a π-*block* [24]. According to the EDG's restriction, the HPFT then automatically inserts proper compiler directives in parsing tree to define the vector operations. In the fourth phase, HPFT applies some heuristic rules to reconstruct the parsing tree such that implicit reuse can be exploited. Then, in the fifth phase, the HPFT further optimizes the vectorized and reuse-exploited abstract syntax tree and automatically inserts directives to define

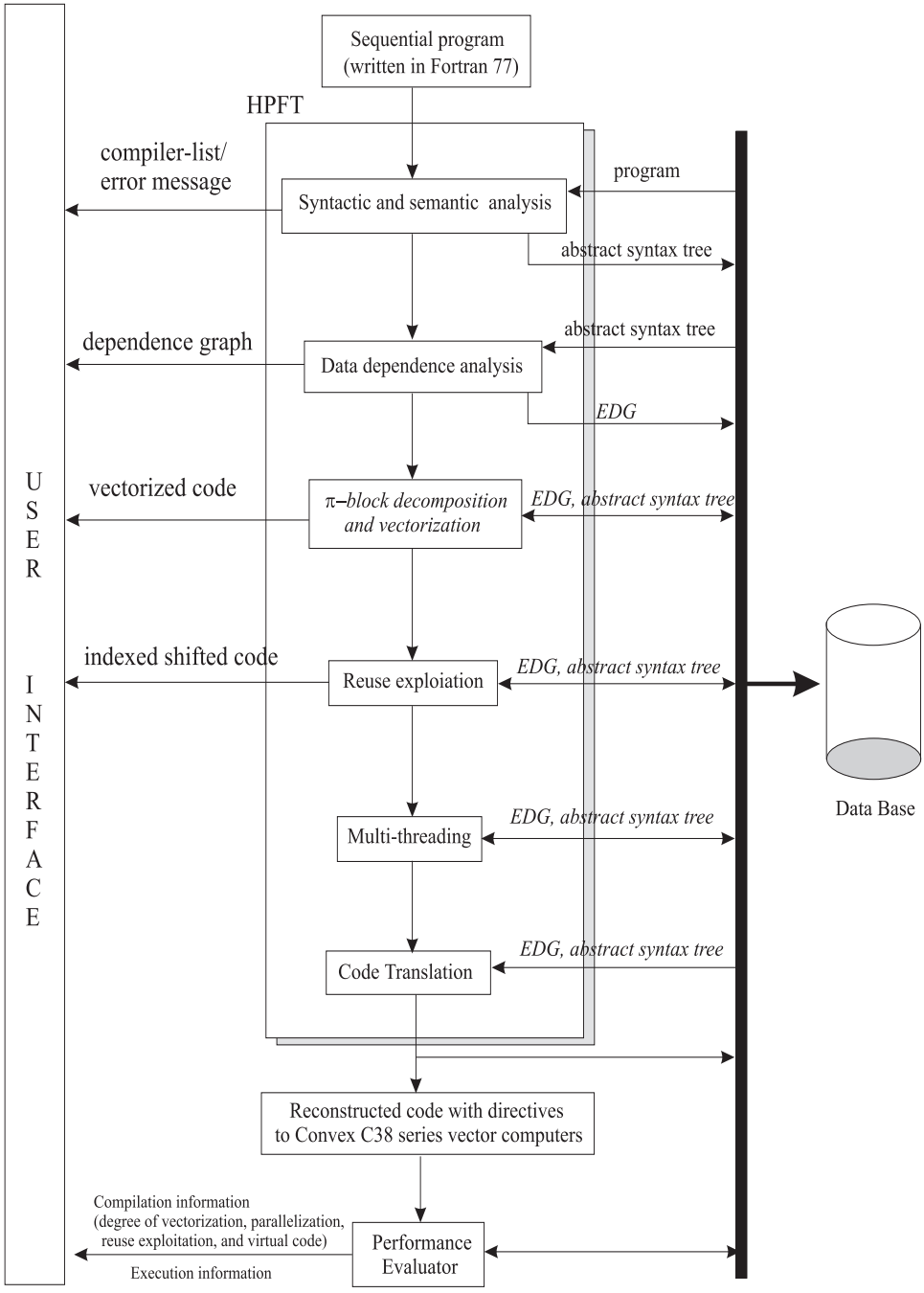


FIG. 1. *The block diagram of HPFT design.*

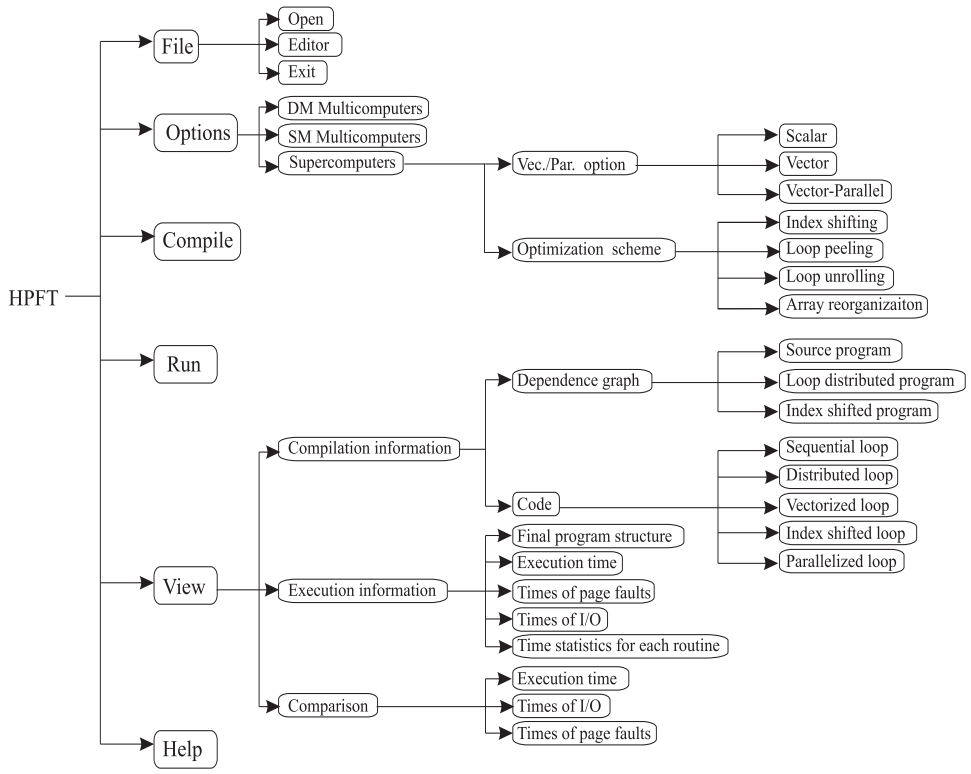


FIG. 2. *Menu-driven design of user interface in HPFT.*

multi-thread. Lastly, in the sixth phase, HPFT transforms the parsing tree into a high performance source code for vector compiler of Convex C3840.

The design of user interface provides a friendly environment for user to select compilation options and to capture the information sent by performance evaluator. Many library functions offered by Motif [13] are used to build the menu-driven procedures. Fig. 2 displays the main functions of user interface design. The user interface of HPFT system provides several switches for user to specify their needs for target machine and source programs. For example, if the target machine is equipped with multiple CPUs, the user may set the 'vector-parallel' switch and specify the number of CPUs for HPFT generating a multi-thread version. Fig. 3(a) shows the snapshots of optimization set by user possibly including index shifting, loop peeling, loop unrolling, and array reorganization. By the user interface design, HPFT system cooperates with the user in determining the best transformation strategy.

Before execution, the performance evaluator offers the information including the estimated CPU time and the degree of vectorization, reuse exploitation, and parallelization for user. After execution, the performance evaluator reports the results and performance information to user. Fig. 3(b)-(f) display some snapshots of

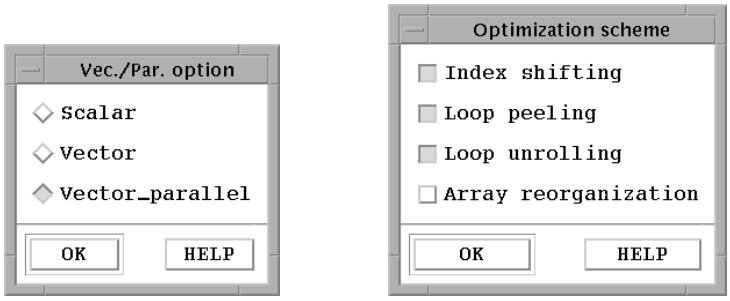
HPFT execution. First, sequential program written in Fortran 77 is analyzed. According to the EDG translation, another version of program is generated by HPFT as shown in Fig. 3(b). By setting the 'compilation information' to user interface as shown in Fig. 3(c), users may view the processing of EDG and the corresponding program in each HPFT processing phase. The menu-driven procedures built in user interface module will be activated and the EDG will be displayed by graphic and tabular viewer. Fig. 3(d) and (e) respectively display the EDG before and after the process of π-*block* decomposition. By setting the 'comparison' switch, HPFT reports the speedup and execution time of both the HPFT version and original version as shown in Fig. 3(f). The design of HPFT system assists vector compiler of Convex C3840 to generate a better object code such that user's programs can be executed in high performance.

3. Data structure and algorithms in HPFT design. In this section, we illustrate the new compilation techniques designed in HPFT for vectorization, reuse exploitation, and multi-threading. To extremely describe the concept of HPFT design, several artificial examples are used. The measure of improvement of HPFT for real applications or benchmarks is located in latter section.

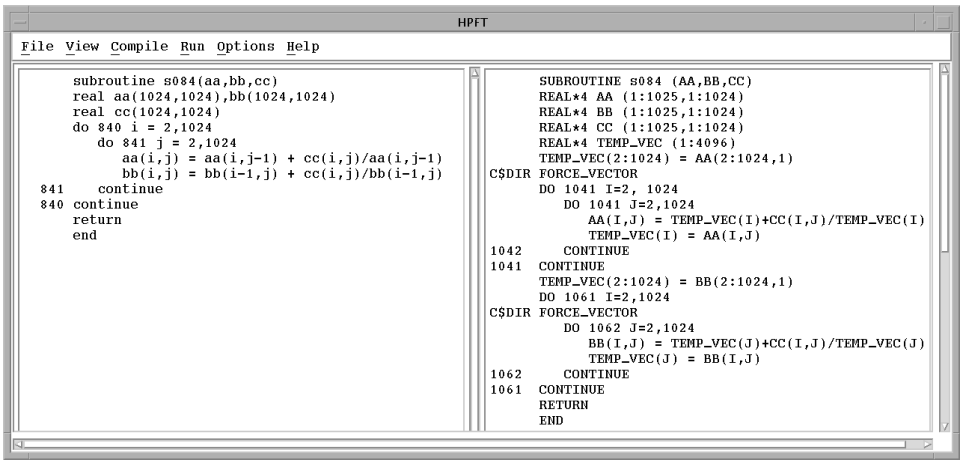
For a sequential program written in Fortran 77, the HPFT parses the program and then constructs the abstract syntax tree. HPFT then analyzes the data dependence relation and constructs the EDG of the original sequential program. Most of the previous designs [18, 2, 23] use *dependence vectors* [19] or *direction vectors* [21] to present the data dependence relation existed in program. In HPFT design, we use EDG to present the data dependence restriction of a loop program. We combine the dependence vector and statement order into an extended dependence vector such that the vectorization can be performed on each statement independently and the reuse exploitation can be easily made. In what follows, we first define the extended dependence graph. To illustrate the design of HPFT in vectorization, reuse exploitation, and multi-threading, algorithm and examples are then described.

Definition 3.1 (Extended Dependence Graph). *An* extended dependence graph $EDG(N, E)$ *of an* n*-nested loop* L *consists of a set of nodes* N *and a set of edges* E*. The node labeled by* S *denotes a statement* S *in loop* L*. An edge labeled by an* extended dependence vector $\bar{d}^e_j = (d^e_{j1}, d^e_{j2}, \ldots, d^e_{j(n+1)})$ *links from node* S_1 *to node* S_2 *if array element is referenced by* S_1 *in iteration* $\bar{i}_1$*, and then referenced again by* S_2 *in iteration* $\bar{i}_2$*, where* $(d^e_{j1}, d^e_{j2}, \ldots, d^e_{jn}) = \bar{i}_2 - \bar{i}_1$ *and* $d^e_{j(n+1)}$ *is 1, 0, or* -1 *according to the textual order of* S_1 *is before, equal to, or after* S_2*, respectively.*

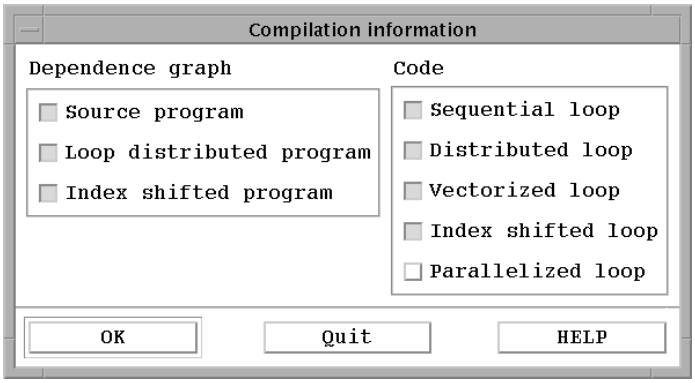
There are input, output, true dependences, and anti-dependences possibly existed in loops program. For the use of vectorization and reuse exploitation, only true dependence and input dependence are considered in EDG. Before the construction of EDG, HPFT uses *variable renaming* and *variable copying* [24] techniques to eliminate the output dependences and anti-dependences. The preprocessing also renames the equivalence declaration for constructing EDG. Program with variable extended dependence vectors has very few opportunities for vectorization, reuse exploitation, and multi-thread extraction. Thus, in HPFT, only loops with constant extended dependence vectors are considered to be reconstructed.


Vec./Par. option
Scalar
Vector
Vector_parallel
OK
HELP
Optimization scheme
Index shifting
Loop peeling
Loop unrolling
Array reorganization
OK
HELP

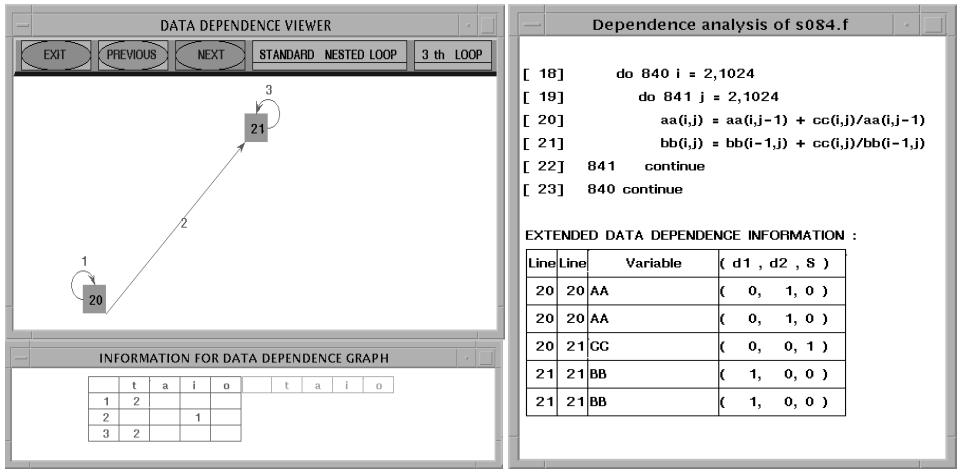
(a)


HPFT
File View Compile Run Options Help
subroutine s084(aa,bb,cc)
real aa(1024,1024),bb(1024,1024)
real cc(1024,1024)
do 840 i = 2,1024
do 841 j = 2,1024
aa(i,j) = aa(i,j-1) + cc(i,j)/aa(i,j-1)
bb(i,j) = bb(i-1,j) + cc(i,j)/bb(i-1,j)
841 continue
840 continue
return
end
SUBROUTINE s084 (AA,BB,CC)
REAL*4 AA (1:1025,1:1024)
REAL*4 BB (1:1025,1:1024)
REAL*4 CC (1:1025,1:1024)
REAL*4 TEMP_VEC (1:4096)
TEMP_VEC(2:1024) = AA(2:1024,1)
C$DIR FORCE_VECTOR
DO 1041 I=2, 1024
DO 1041 J=2,1024
AA(I,J) = TEMP_VEC(I)+CC(I,J)/TEMP_VEC(I)
TEMP_VEC(I) = AA(I,J)
1042 CONTINUE
1041 CONTINUE
TEMP_VEC(2:1024) = BB(2:1024,1)
DO 1061 I=2,1024
C$DIR FORCE_VECTOR
DO 1062 J=2,1024
BB(I,J) = TEMP_VEC(J)+CC(I,J)/TEMP_VEC(J)
TEMP_VEC(J) = BB(I,J)
1062 CONTINUE
1061 CONTINUE
RETURN
END

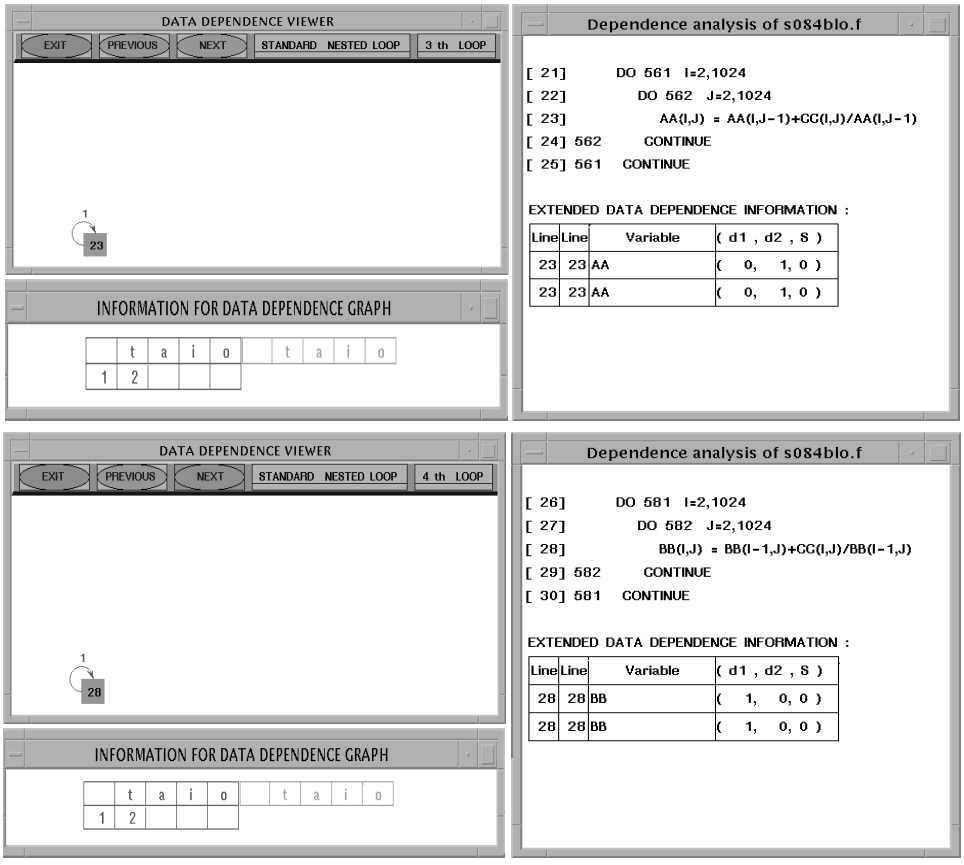
(b)


Compilation information
Dependence graph
Source program
Loop distributed program
Index shifted program
Code
Sequential loop
Distributed loop
Vectorized loop
Index shifted loop
Parallelized loop
OK
Quit
HELP

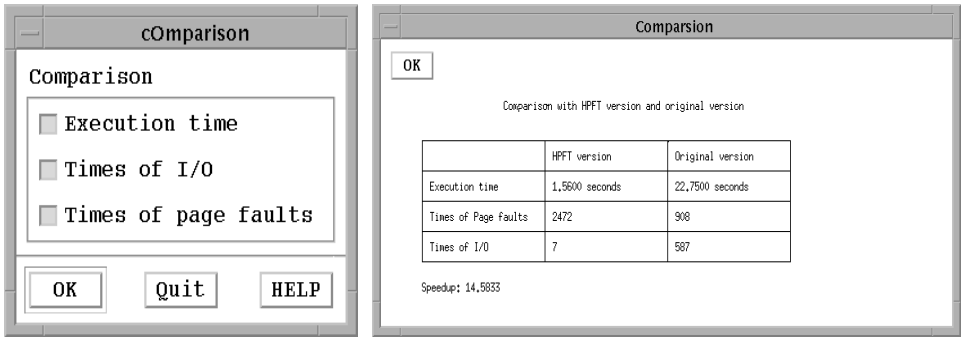
(c)


DATA DEPENDENCE VIEWER
EXIT
PREVIOUS
NEXT
STANDARD NESTED LOOP
3 th LOOP
3
21
2
1
20
INFORMATION FOR DATA DEPENDENCE GRAPH
t a i o t a i o
1 2
2 1
3 2
Dependence analysis of s084.f
[18] do 840 i = 2,1024
[19] do 841 j = 2,1024
[20] aa(i,j) = aa(i,j-1) + cc(i,j)/aa(i,j-1)
[21] bb(i,j) = bb(i-1,j) + cc(i,j)/bb(i-1,j)
[22] 841 continue
[23] 840 continue
EXTENDED DATA DEPENDENCE INFORMATION :
Line Line Variable (d1 , d2 , S)
20 20 AA (0, 1, 0)
20 20 AA (0, 1, 0)
20 21 CC (0, 0, 1)
21 21 BB (1, 0, 0)
21 21 BB (1, 0, 0)

(d)


DATA DEPENDENCE VIEWER
EXIT
PREVIOUS
NEXT
STANDARD NESTED LOOP
3 th LOOP
1
23
INFORMATION FOR DATA DEPENDENCE GRAPH
t a i o t a i o
1 2
Dependence analysis of s084blo.f
[21] DO 561 I=2,1024
[22] DO 562 J=2,1024
[23] AA(I,J) = AA(I,J-1)+CC(I,J)/AA(I,J-1)
[24] 562 CONTINUE
[25] 561 CONTINUE
EXTENDED DATA DEPENDENCE INFORMATION :
Line Line Variable (d1 , d2 , S)
23 23 AA (0, 1, 0)
23 23 AA (0, 1, 0)
DATA DEPENDENCE VIEWER
EXIT
PREVIOUS
NEXT
STANDARD NESTED LOOP
4 th LOOP
1
28
INFORMATION FOR DATA DEPENDENCE GRAPH
t a i o t a i o
1 2
Dependence analysis of s084blo.f
[26] DO 581 I=2,1024
[27] DO 582 J=2,1024
[28] BB(I,J) = BB(I-1,J)+CC(I,J)/BB(I-1,J)
[29] 582 CONTINUE
[30] 581 CONTINUE
EXTENDED DATA DEPENDENCE INFORMATION :
Line Line Variable (d1 , d2 , S)
28 28 BB (1, 0, 0)
28 28 BB (1, 0, 0)

(e)



(f)

FIG. 3. *Snapshots of HPFT translation process. (a) Snapshots of setting compilation options to user interface of HPFT. (b) The sequential program and the HPFT translated code. (c) The setting of views of compilation information. (d) A view of the original EDG provided by data dependence viewer. (e) A view of EDG after* $\pi - block$ *decomposition. (f) Speedup and execution of original version and HPFT version.*

Several vector compilers offer compiler directives for users to manually optimize the program. We implemented our methods for Convex vector compiler. To assist vector compiler of Convex C38 series extracting more or better vector operations from program, HPFT automatically inserts directive C$DIR FORCE_VECTOR. In Convex C38 series, the FORCE_VECTOR directive [10] will force vector compiler to vectorize the loop that follows. If the FORCE_VECTOR directive is applied to an outer loop, the Convex compiler will move the specified loop to the innermost position and run it in vector mode. Similar directives can be also found in CRAY X-MP and IBM ES/9000 supercomputers [11, 14]. In this paper, to display the program in short, sometimes we use the vectorized statement

$$A(1:128) = B(1:128) + C(1:128)$$

to denote the generated code

```
      C$DIR FORCE_VECTOR
      DO  1   I = 1, 128
          A(I) = B(I) + C(I)
1     CONTINUE.
```

The design of phases 2 to 6 of HPFT is described in what follows.

3.1. $\pi - Block$ **decomposition and vectorization.** Before vectorization, HPFT partitions the *EDG* into several strongly connected components. Each component is called a π-*block* [24]. If statement S_i is not strongly connected to other statements, isolating S_i can relax the dependence constraint of S_i. Thus, π-*block* decomposition will benefit to extract more vector operations from sequential loops.

However, π-*block* decomposition possibly distributes two statements that have reuse opportunities into two different π-*blocks*. The reuse opportunities will be lost. In HPFT design, the π-*block* decomposition is mixed with the vectorization such that reuse opportunities can be preserved. Statements that are vectorized in the same loop and have reuse opportunities are not to be decomposed into two π-*blocks*. Two advantages can be obtained from HPFT approach. First, the reuse opportunities can be preserved. Second, additional loop heading tests can be saved.

To improve or make possible the execution of vector operations, loop restructuring techniques such as *loop distribution* [3, 24], *loop interchange* [3, 24, 16], *loop fusion* [24, 16], and *scalar expansion* [24] were introduced in previous study and some of them were implemented in compilers of vector machines. In HPFT design, the π-*block* decomposition and vectorization phase integrates loop distribution, loop interchange, and loop fusion techniques to maintain the reuse opportunities. The π-*block* decomposition design and vectorization design of HPFT are outlined in the following steps.

(1) Find all strongly connected components of EDG.

(2) Topological sort these strongly connected components according to the data dependence relation of these components.

(3) For each statement in EDG, find the best possible loop level for vectorization such that the memory stride is minimal. In this step, HPFT finds a vectorization vector V_i for each statement S_i.

(4) Combine two adjacent strongly connected components if they have reuse opportunities.

The detail algorithm of the π-*block* decomposition and vectorization is located at the latter paragraph. In what follows, we describe how to find the vectorization vector V_i for each statement S_i in a strongly connected component.

In an EDG, if an edge $\bar{d}^e$ links two nodes from S_1 to S_2, then Tail($\bar{d}^e$) denotes node S_1 and Head($\bar{d}^e$) denotes node S_2. Let D_i denote the set of extended dependence vectors $\{\bar{d}_j^e\}$ for all $\bar{d}_j^e$ pointing to or from S_i in EDG and both the Head($\bar{d}_j^e$) and Tail($\bar{d}_j^e$) belonging to the same strongly connected component. That is, all $\bar{d}_j^e$ that satisfy the following two conditions should be collected into the set D_i.

(C1) Head($\bar{d}_j^e$)=S_i or Tail($\bar{d}_j^e$)=S_i

(C2) Both Head($\bar{d}_j^e$) and Tail($\bar{d}_j^e$) belong to the same strongly connected component.

The HPFT determines a vectorization vector $V_i = (v_1, v_2, \ldots, v_{n+1})$, $v_k = 0$ or 1, $1 \leq k \leq n+1$, for each statement S_i in loop body. The value of v_k is 0 or 1 respectively denoting the enable or disable of vectorization on loop in level k, for $1 \leq k \leq n$. The value of v_{n+1} is always 1. Let $sign(\bar{p})$ denote the value of "+" or "−" depending on the first nonzero element of vector $\bar{p}$ is positive or negative, respectively. Let $\bar{d}_j^e = (d_1, \ldots, d_{n+1})$. The operation $V_i \bullet \bar{d}_j^e$ is defined as follows.

$$V_i \bullet \bar{d}_j^e = (v_1 d_1, v_2 d_2, \ldots, v_{n+1} d_{n+1}).$$

To guarantee the semantic correctness, the vectorization vector V_i should satisfy the following vectorization condition

$$sign(V_i \bullet \bar{d}_j^e) = \text{"+"}, \quad \forall \bar{d}_j^e \in D_i.$$

The reason is stated as follows. The *sign* of all dependence vectors in EDG is "+" since HPFT only consider the input and true dependence in EDG construction. The vectorization should maintain the semantic correctness of original sequential program. For a true dependence $\bar{d}_j^e$ linking from S_i to S', data generation of S_i should be performed before the data use of S'. Assume that statement S_i is vectorized in the kth loop level. The value v_k is 0 and the value of other elements v_x is 1, for all $x \neq k$. Thus, the kth element of $V_i \bullet \bar{d}_j^e$ is 0 and other elements of $V_i \bullet \bar{d}_j^e$ are the same as $\bar{d}_j^e$. If the condition $sign(V_i \bullet \bar{d}_j^e) = "+"$ is satisfied, it indicates that vectorization on loop (in level k) maintains the precedence relation for S_i and S'.

Let $D'_{ik} = \{\bar{d'}_j^e\}$ denote the set of n-dimensional vectors that is obtained by masking the kth element of vectors in D_i. To determine the *vectorization vector* V_i for each statement S_i, HPFT constructs D'_{i1} by masking the first element ($k = 1$) of all the extended dependence vectors $\bar{d}_j^e \in D_i$. Then, HPFT checks whether the *sign* of the first nonzero value of $\bar{d'}_j^e$ is "+" or not, for all $\bar{d'}_j^e \in D'_{i1}$. If the answer is 'yes', a zero value is determined to the first element of V_i and the vectorization is made on the outermost loop. Similarly, the value of the second element (k=2), ..., and the nth element ($k = n$) of V_i can be determined. A vectorization vector V_i is then determined perhaps with two or more zeros. Because the Fortran is a column-major language, loop that controls the leftmost position of subscript of array variable will have the property of shortest memory stride. Under the consideration of the shortest memory stride, HPFT preserves one of these zero elements and sets other elements to 1.

Consider the following example.
Example 1:

```
      DO  1  I = 2, 65
          DO  2  J = 2, 65
              DO  3  K = 2, 65
                  S1 :  A(I,J,K) = B(I-1,J+1,K) + A(I,J,K-1)    (L1)
                  S2 :  B(I,J,K) = A(I-1,J+1,K) + B(I,J-1,K)
3             CONTINUE
2         CONTINUE
1     CONTINUE
```

The *extended dependence vectors* in loop $L1$ are

$\bar{d}_1^e = (1, -1, 0, +1)$ for variables $A(I, J, K)$ and $A(I-1, J+1, K)$,
$\bar{d}_2^e = (0, 0, 1, 0)$ for variables $A(I, J, K)$ and $A(I, J, K-1)$,
$\bar{d}_3^e = (1, -1, 0, -1)$ for variables $B(I, J, K)$ and $B(I-1, J+1, K)$, and
$\bar{d}_4^e = (0, 1, 0, 0)$ for variables $B(I, J, K)$ and $B(I, J-1, K)$.

The EDG of loop $L1$ is depicted in Fig. 4. By the previous definition, D_1 is $\{\bar{d}_1^e, \bar{d}_2^e, \bar{d}_3^e\}$ and D_2 is $\{\bar{d}_1^e, \bar{d}_3^e, \bar{d}_4^e\}$. The selected vectorization vectors V_1 for S_1 and V_2 for S_2 are (1, 0, 1, 1) and (1, 1, 0, 1), respectively, where V_1 and V_2 satisfy the *valid vectorizing conditions*

$sign(V_1 \bullet \bar{d}_i^e) = "+"$, for all $\bar{d}_i^e \in D_1$ and
$sign(V_2 \bullet \bar{d}_k^e) = "+"$, for all $\bar{d}_k^e \in D_2$.

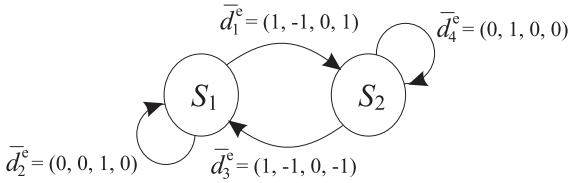


FIG. 4. *The EDG of loop L1.*

Since the second element of V_1 has a zero value, statement S_1 can be vectorized on loop J. Similarly, statement S_2 can be vectorized on loop K. HPFT translates loop $L1$ into the following version $L1'$.

```
      DO  1  I = 2, 65
          C$DIR FORCE_VECTOR
          DO  21  J = 2, 65
              DO  3  K = 2, 65
                S1 :  A(I,J,K) = B(I-1,J+1,K) + A(I,J,K-1)
3             CONTINUE                                          (L1')
21        CONTINUE
          DO  2  J = 2, 65
              C$DIR FORCE_VECTOR
              DO  31  K = 2, 65
                  S2 :  B(I,J,K) = A(I-1,J+1,K) + B(I,J-1,K)
31            CONTINUE
2         CONTINUE
1     CONTINUE
```

The resultant two-way nonperfect loop $L1'$ is vectorized in different loop levels for these two statements. Because that all the extended dependence vectors $\bar{d}_j^e \in D_1$ capture the information of statement order of S_1 and S_2, HPFT can independently perform the vectorization process on S_1 without the effect of vectorization of S_2. Thus, S_1 can be vectorized on loop J, the valid vectorization loop with shortest memory stride.

The *π-block* decomposition and vectorization algorithm is listed in follows.

Algorithm: *π-block* Decomposition and Vectorization of an n-nested loop L
Input: The parsing tree of loop L and the $EDG(N, E)$,
where $N = \{S_i, 1 \leq i \leq s\}$
Output: The reconstructed parsing tree, decomposed *π-blocks*, and
vectorization vector V_i for each statement $S_i, 1 \leq i \leq s$.
Step 1: Decompose the EDG into maximal number of strongly connected components.
Let b denote the number of these components.
Step 2: Topologically sort these components according to dependence relation existed among these components.

Let $SC_1(N_1, E_1)$, $SC_2(N_2, E_2)$, ..., $SC_b(N_b, E_b)$ denote the sorted components, where N_i and E_i respectively denote the node set and edge set of component SC_i, $1 \leq i \leq b$.

Step 3: /* Find vectorization vector V_i for statement S_i */

For each statement $S_i \in N, 1 \leq i \leq s$, do

Let $V_i = (v_1, v_2, \ldots, v_{n+1})$.

Let $D_i = \{\bar{d}_j^e = (d_j^1, d_j^2, \ldots, d_j^{n+1})\}$ for all $\bar{d}_j^e$ pointing to or from S_i in EDG and both the Head($\bar{d}_j^e$) and Tail($\bar{d}_j^e$) belonging to the same strongly connected component.

For k=1 to n do

Let $D_i' = \{\bar{d'}_j^e = (d_j^1, \ldots, d_j^{k-1}, d_j^{k+1}, \ldots, d_j^{n+1})\}$, where $\bar{d'}_j^e$ is obtained by masking the kth element of $\bar{d}_j^e$.

If there exists a $\bar{d'}_j^e \in D_i'$ such that the first nonzero element of $\bar{d'}_j^e$ is negative

set $v_k = 1$

Else

set $v_k = 0$

Endif

Endfor

Set $v_{n+1} = 1$.

Preserve one of zero elements in V_i such that memory stride is minimal and set other zero elements to 1.

Endfor

Step 4: /* Combine strongly connected components that have reuse opportunities into one π-block */

For $k = b$ to 2 step by -1 do

Set flag $Combine = 1$.

For each dependence vector $\bar{d}^e$ linking from SC_{k-1} to SC_k do

Assume $\bar{d}^e = (d_1, d_2, \ldots, d_{n+1})$ pointing from $S_i \in N_{k-1}$ to $S_j \in N_k$.

If one of the following conditions satisfied, set flag $Combine = 0$.

(1) the zero value of V_i and V_j is at the different positions.

(2) Let the lth position of V_i have value 0.

Let $\bar{d'}^e = (d_1, d_2, \ldots, d_{l-1}, d_{l+1}, \ldots, d_{n+1})$.

The first nonzero element of $\bar{d'}^e$ is negative.

Endif

Endfor

If flag $Combine = 1$

Combine components SC_{k-1} and SC_k into a *π-block*.

Endif

Endfor

According to the decomposed EDG, reconstruct the parsing tree by applying the loop distribution and statement reordering techniques.

According V_i, for $1 \leq i \leq s$, insert C$DIR FORCE_VECTOR directives to parsing tree.

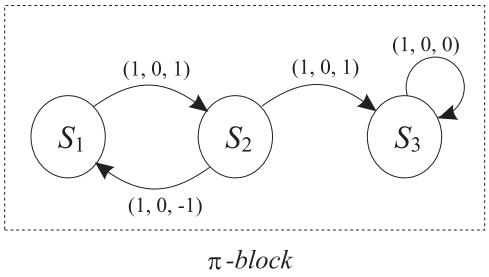


FIG. 5. *The EDG of L2 after performing $\pi - block$ decomposition.*

Consider the following loops program.

Example 2:

$$
\begin{array}{ll}
 & \text{DO } 1 \;\; I = 2, 65 \\
 & \quad \text{DO } 2 \;\; J = 2, 65 \\
 & \quad\quad S_1 : \;\; A(I,J) = B(I-1,J) * 0.5 \\
 & \quad\quad S_2 : \;\; B(I,J) = A(I-1,J) * 0.5 \qquad\qquad (L2) \\
 & \quad\quad S_3 : \;\; C(I,J) = B(I-1,J) * 0.5 + C(I-1,J) \\
2 & \quad \text{CONTINUE} \\
1 & \text{CONTINUE}
\end{array}
$$

In the *π-block* decomposition phase, the HPFT will not decompose two consecutive statements if they will be vectorized in the same loop and have reuse opportunity. The EDG of loop $L2$ will be constructed in phase 2 of HPFT as shown in Fig. 5. The HPFT looks ahead that S_2 and S_3 are both vectorized on loop J and the reuse distance of array B is one iteration gap of loop I. Thus, HPFT does not decompose the EDG into two *π-blocks*. According to the EDG, as described in latter subsection (phase 4), loop $L2$ can be translated into the following vectorized and reuse exploited code by HPFT.

$$
\begin{array}{ll}
 & C(2, 2:65) = B(1, 2:65) * 0.5 + C(1, 2:65) \\
 & B(2, 2:65) = A(1, 2:65) * 0.5 \\
 & C(3, 2:65) = B(2, 2:65) * 0.5 + C(2, 2:65) \\
 & \text{DO } 1 \;\; I = 3, 64 \\
 & \quad S_1 : \;\; A(I-1, 2:65) = B(I-2, 2:65) * 0.5 \\
 & \quad S_2 : \;\; B(I, 2:65) = A(I-1, 2:65) * 0.5 \qquad\qquad (L2') \\
 & \quad S_3 : \;\; C(I+1, 2:65) = B(I, 2:65) * 0.5 + C(I, 2:65) \\
1 & \text{CONTINUE} \\
 & A(64, 2:65) = B(63, 2:65) * 0.5 \\
 & A(65, 2:65) = B(64, 2:65) * 0.5 \\
 & B(65, 2:65) = A(64, 2:65) * 0.5
\end{array}
$$

As a result, loop $L2'$ saves 62 vector loads for $A(I-1, 2:65)$ in S_2 and 62 vector

loads for $B(I, 2:65)$ in S_3. For instance, $A(2, 2:65)$ generated in vector register, say $VR1$, by S_1 at instance $I = 3$ will be immediately reused by S_2 at instance $I = 3$ without loading again from memory to vector register.

Consider another example which is extracted from subroutine S084 of vector benchmark in NETLIB of NCHC (National Center for High Performance Computing). In the *π-block* decomposition phase, HPFT will decompose the EDG.

Example 3:

$$
\begin{array}{ll}
 & \text{DO } 1 \quad I = 2, n \\
 & \quad \text{DO } 2 \quad J = 2, n \\
 & \quad\quad S_1: \; AA(I,J) = AA(I,J-1) + CC(I,J)/AA(I,J-1) \\
 & \quad\quad S_2: \; BB(I,J) = BB(I-1,J) + CC(I,J)/BB(I-1,J) \qquad (L3) \\
2 & \quad \text{CONTINUE} \\
1 & \text{CONTINUE}
\end{array}
$$

The EDG of $L3$ and the data structure of EDG are respectively shown in Fig. 6(a) and (b). Since S_1 and S_2 can be vectorized in different loops, the HPFT decomposes the EDG as shown in Fig. 6(c). In phase 3, HPFT will perform vectorization and modify the parsing tree equivalent to the program $L3'$ as follows.

$$
\begin{array}{ll}
 & \text{DO } 11 \quad J = 2, n \\
 & \quad S_1: \; AA(2:n,J) = AA(2:n,J-1) + CC(2:n,J)/AA(2:n,J-1) \\
11 & \text{CONTINUE} \qquad (L3') \\
 & \text{DO } 2 \quad I = 2, n \\
 & \quad S_2: \; BB(I,2:n) = BB(I-1,2:n) + CC(I,2:n)/BB(I-1,2:n) \\
2 & \text{CONTINUE}
\end{array}
$$

The decomposition of the set of statements $\{S_1, S_2\}$ into two sets $\{S_1\}$ and $\{S_2\}$ causes statement S_1 vectorized in loop I. Benefits of shorter memory stride accessing and vectorization are gained when executing S_1. There are reuse opportunities of arrays AA and BB existed in S_1 and S_2, respectively. These reuse opportunities will be further exploited in the reuse exploiting phase of HPFT. The HPFT automatically determines how to decompose the original EDG into several *π-blocks* such that maximum benefits from vectorization and reuse exploitation can be obtained.

3.2. Reuse exploitation of vector data. A considerable amount of research has been done on this topic [4, 7, 12]. In HPFT design, loops with reuse distance equal to or larger than one iteartion gap can be further reduced to zero. The reduction of reuse distance is determined by several heuristic rules operated on EDG. In what follows, we will use an example to illustrate the process of reuse exploitation in HPFT. Improvements in real application programs are measured in latter section. In phase 4, the HPFT exploits the reuse opportunities existed in the parsing tree. Consider the following vectorized loop $L4$ whose parsing tree is the input of phase 4 of HPFT.

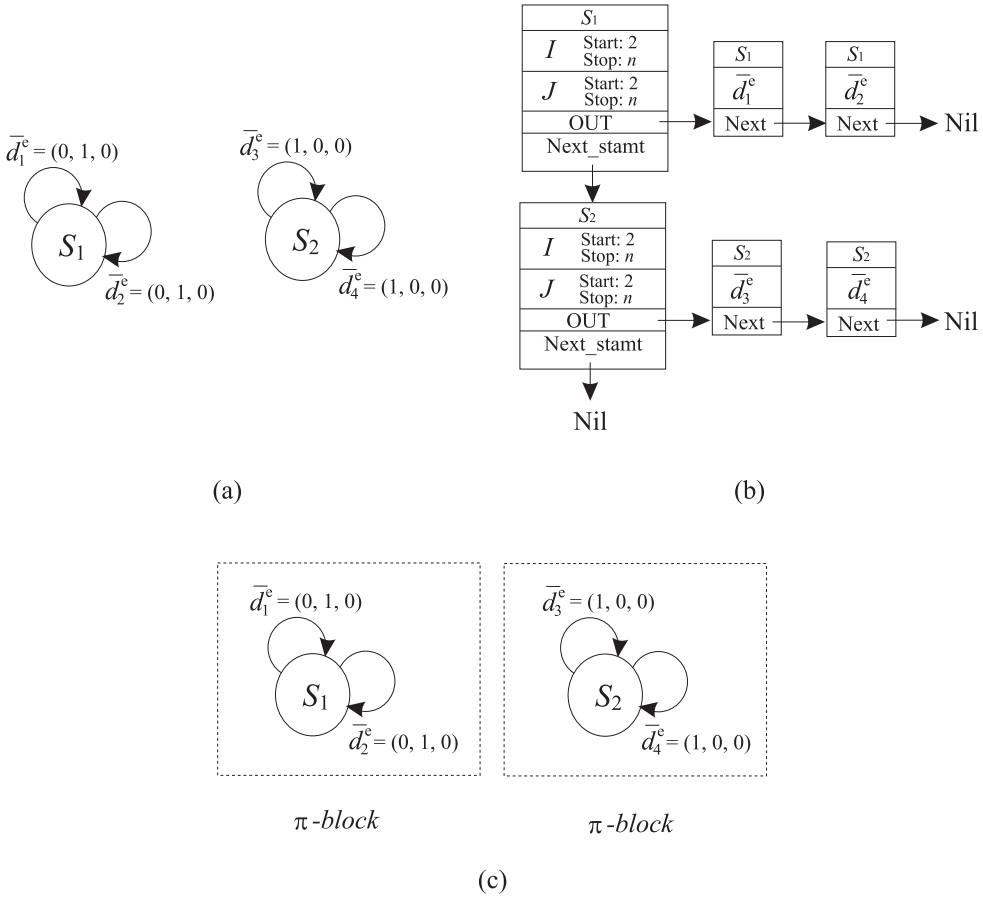


FIG. 6. *Data structure of EDG and π − block decomposition of loop L3. (a) The EDG of L3. (b) Data structure representing EDG in HPFT. (c) The decomposed EDG of L3.*

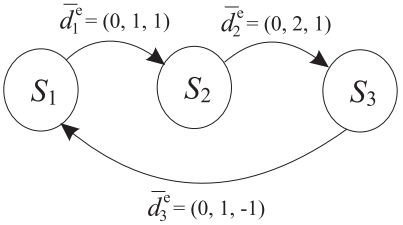


FIG. 7. *The EDG of L4.*

Example 4:

$$
\begin{array}{lll}
\text{DO} & 2 \quad J = 3, 642 & \\
 & S_1: \quad A(1:64, J) = C(1:64, J-1) + X & \\
 & S_2: \quad B(1:64, J) = A(1:64, J-1) * Y & (L4) \\
 & S_3: \quad C(1:64, J) = B(1:64, J-2) - Z & \\
2 & \text{CONTINUE} &
\end{array}
$$

The EDG of loop $L4$ is shown in Fig. 7. In loop $L4$, the *extended dependence vectors* are $\bar{d}_1^e = (0,\ 1,\ 1)$, $\bar{d}_2^e = (0,\ 2,\ 1)$, and $\bar{d}_3^e = (0, 1, -1)$. Data elements $A(1:64, 3)$ generated by statement S_1 at instance $J = 3$ will be used by S_2 at instance $J = 4$. However, most supercomputers generate $A(1:64, 3)$ in vector register, say $VR1$, at $J = 3$ and then load $A(1:64, 3)$ again in another vector register, say $VR2$, at $J = 4$. This is because that $VR1$ should swap out the data $A(1:64, 3)$ before executing $J = 4$ for keeping the generated data $A(1:64, 4)$. Likewise, arrays B and C generated by S_2 and S_3 will be used by statements S_3 and S_1, respectively. The reuse distance can be reduced by applying index shift method [15] or loop alignment method [1]. HPFT performs the index shift [15] procedure **Shift** on EDG of $L4$ to exploit the reuse opportunities of arrays A, B, and C.

Let sets IN(S) and OUT(S) respectively denote the set of edges incoming to and outgoing from the node S in EDG. In the following, we will first give the definition of *shift vector* $\bar{d}_{max}^e$ which will be used to reduce the dependence distance between two references.

DEFINITION 3.2 (SHIFT VECTOR AND REUSE VECTOR). *Let there be l extended dependence vectors $\bar{d}_i^e \in IN(S)$ and r extended dependence vectors $\bar{d}_j^e \in OUT(S)$ for $1 \le i \le l, 1 \le j \le r$. A* shift vector $\bar{d}_{max}^e$ *of IN(S) is the maximum vector that satisfies the condition $\forall\ \bar{d}_i^e \in IN(S)$, $\bar{d}_i^e - \bar{d}_{max}^e > 0^{n+1}$ where 0^{n+1} is an $(n+1)$-dimensional zero vector, for $1 \le i \le l$. Extended dependence vectors $\bar{d}_i^e \in IN(S)$ that are equal to $\bar{d}_{max}^e$ are called* reuse vectors $\bar{d}_{reuse}^e$ *of IN(S), for $1 \le i \le l$. Similarly, a* shift vector $\bar{d}_{max}^e$ *of OUT(S) is the maximum vector that satisfies the condition $\forall\ \bar{d}_j^e \in OUT(S)$, $\bar{d}_j^e - \bar{d}_{max}^e > 0^{n+1}$, for $1 \le j \le r$. Extended dependence vectors $\bar{d}_j^e \in OUT(S)$ that are equal to $\bar{d}_{max}^e$ are called* reuse vectors $\bar{d}_{reuse}^e$ *of OUT(S).*

Note that there are possibly more than one dependence vector $\bar{d}_i^e \in$ IN(S) being reuse vectors of IN(S). Let $\bar{d}^e = (d_1^e, \ldots, d_{n+1}^e)$ be the reuse vector of IN(S). Two cases of $\bar{d}^e$ will be discussed as follows. First, if the value d_{n+1}^e is 1, $\bar{d}^e - \bar{d}_{max}^e$ is

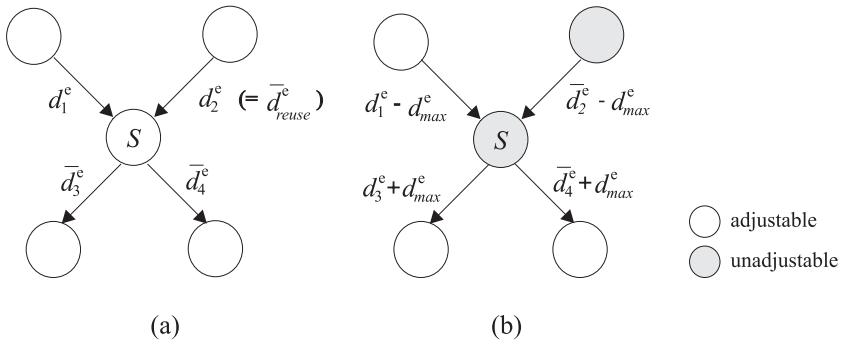


FIG. 8. *Index shift operation on node S. (a) Original EDG. (b) Applying $Shift(S, \bar{d}_{max}^e, IN)$.*

equal to $(0, 0, \ldots, 0, 1)$. In another case, if the value d_{n+1}^e is -1, $\bar{d}^e - \bar{d}_{max}^e$ is equal to $(0, \ldots, 0, 1, -1)$. When a node S is selected to be adjusted in an IN direction, all the dependence vectors $\bar{d}_i^e \in \text{IN}(S)$ can be reduced to $\bar{d}_i^e - \bar{d}_{max}^e$. The semantic correctness is maintained since condition $\bar{d}_i^e - \bar{d}_{max}^e > 0^{n+1}$ is satisfied. Let *dir* be the variable with value of IN or OUT. When applying index shift operation on node S, the value of *dir* is IN or OUT can be determined by several heuristic rules[8]. Let the complement direction $\overline{dir}$ denote the value IN or OUT if *dir* is OUT or IN respectively. We now formally define the *index shift operation* on S as follows:

Shift(S, $\bar{d}_{max}^e$, dir):
  $\forall$ edges $\bar{d}^e \in dir(S)$, $\bar{d}^e = \bar{d}^e - \bar{d}_{max}^e$.
  $\forall$ edges $\bar{d}^e \in \overline{dir}(S)$, $\bar{d}^e = \bar{d}^e + \bar{d}_{max}^e$.
  Mark node S with "unadjustable".
  Mark adjustable nodes in set Tail($\bar{d}_{reuse}^e$) or Head($\bar{d}_{reuse}^e$) with "unadjustable" depending on that the value of *dir* is IN or OUT respectively.

Fig. 8 depicts the index shift operation performing on node S in an IN direction. Assume that $\bar{d}_2^e$ is the reuse vectors of IN(S). In Fig. 8(b), the index shift operation is applied on node S with dir = IN such that the reuse distance between statements S and S' can be reduced, for all $S' \in$ Tail($\bar{d}_i^e$), $\forall\ \bar{d}_i^e \in$ IN(S). Dependence vectors $\bar{d}_i^e$ $\in$ IN(S), for i = 1 and 2, can get benefits by decreasing their dependence distance value. The decreasing of the dependence value indicates that the reuse distance is shortened when running program. However, the reuse distance of dependence vectors $\bar{d}_j^e \in$ OUT(S), for j = 3 and 4, has been increased by value $\bar{d}_{max}^e$ after applying Shift(S, $\bar{d}_{max}^e$, IN) on node S. Initially, all nodes in EDG are adjustable. When we apply the index shift operation Shift(S, $\bar{d}_{max}^e$, dir) on node S, the reuse distance of edges $\bar{d}_{reuse}^e$ is optimal. All nodes linked by edges $\bar{d}_{reuse}^e$ can not be further adjusted to maintain the optimal distance.

In Example 4, HPFT first applies the **Shift**(S_2, (0, 1, 0), IN) on statement S_2. That is, HPFT adjusts the variables' subscript of S_2 such that value of the

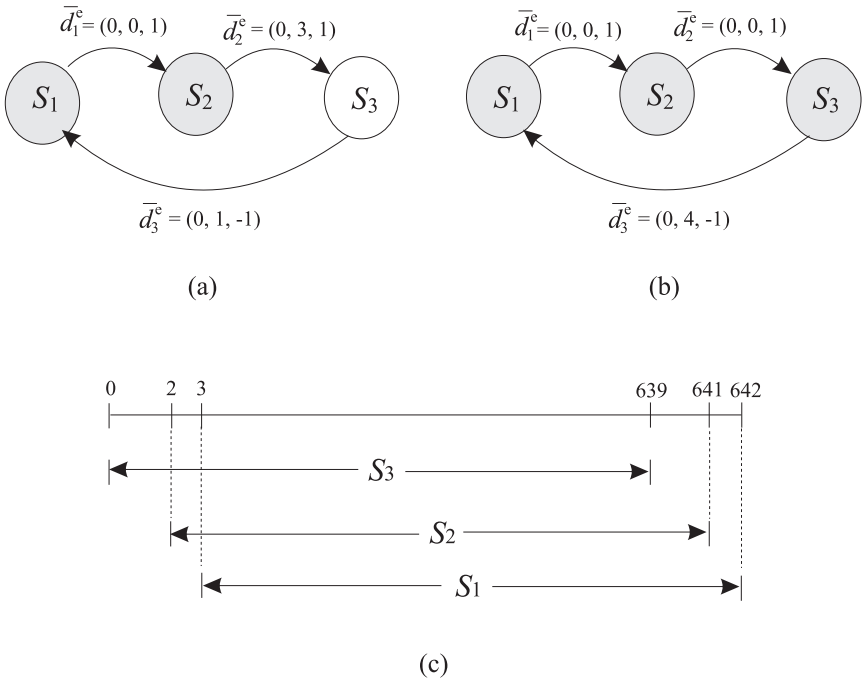


FIG. 9. *Index shift operation applying on loop L4. (a) The EDG after applying* $Shift(S_2, (0,1,0), IN)$. *(b) The EDG after applying* $Shift(S_3, (0,3,0), IN)$. *(c) Iteration range of statements after applying index shift operations.*

extended dependence vector $\{\bar{d}_1^e\} = (0,\ 1,\ 1) \in \mathrm{IN}(S_2)$ can be reduced to (0, 0, 1). Statement S_2 is then changed to

$$S_2 : B(1:64, J+1) = A(1:64, J) * Y, \quad \text{for} \quad 2 \leq J \leq 641.$$

To preserve the exploited reuse, statements S_1 and S_2 can not be further adjusted. The HPFT marks them with 'unadjustable' as shown the shaded nodes in Fig. 9(a). The dependence distance 1 of $\bar{d}_1^e$ in Fig. 7 is then accumulated to $\bar{d}_2^e$ since the reference $B(1:64, J+1)$ of the adjusted S_2 and the reference $B(1:64, J-2)$ of S_3 in $L4$ have an extended dependence vector (0, 3, 1) as shown in Fig. 9(a). Continuously, HPFT applies **Shift**(S_3, (0, 3, 0), IN) procedure on node S_3 such that $\bar{d}_2^e \in \mathrm{IN}(S_3)$ can be reduced to (0, 0, 1). Statement S_3 can be changed to

$$S_3 : C(1:64, J+3) = B(1:64, J+1) - Z, \quad \text{for} \quad 0 \leq J \leq 639.$$

The dependence distance of $\bar{d}_2^e$ is accumulated to $\bar{d}_3^e$ as shown in Fig. 9(b). The iteration range of J on statements S_1, S_2, and S_3 is shown in Fig. 9(c). HPFT then gather the intersection of these statements' running iteration range into one loop.

Translated by HPFT, parsing tree of loop $L4$ will be reconstructed into another parsing tree equivalent to the following loop $L4'$.

```
      DO  21  J = 0, 1
          S3 :  C(1 : 64, J + 3) = B(1 : 64, J + 1) − Z
   21 CONTINUE
      S2 :  B(1 : 64, 3) = A(1 : 64, 2) ∗ Y
      S3 :  C(1 : 64, 5) = B(1 : 64, 3) − Z
      DO  2  J = 3, 639
          S1 :  A(1 : 64, J) = C(1 : 64, J − 1) + X
          S2 :  B(1 : 64, J + 1) = A(1 : 64, J) ∗ Y          (L4′)
          S3 :  C(1 : 64, J + 3) = B(1 : 64, J + 1) − Z
   2  CONTINUE
      DO  22  J= 640, 641
          S1 :  A(1 : 64, J) = C(1 : 64, J − 1) + X
          S2 :  B(1 : 64, J + 1) = A(1 : 64, J) ∗ Y
   22 CONTINUE
      S1 :  A(1 : 64, 642) = C(1 : 64, 641) + X
```

This makes that arrays A and B are reused in the same iteration and the reuse of array C occurs after 4 running iterations in loop J labeled by 2. To reduce the reuse distance of array C incurred by the extended dependence vector $(0,\ 4,\ -1)$, HPFT invokes the procedure **Spreading** to further apply loop spreading technique [24] on loop J labeled by 2 in $L4'$. The reconstructed syntax tree is equivalent to the following loop $L4''$:

```
   DO  23  K = 3, 6
       DO2  J = K, 639, 4
           S1 :  A(1 : 64, J) = C(1 : 64, J − 1) + X
           S2 :  B(1 : 64, J + 1) = A(1 : 64, J) ∗ Y          (L4″)
           S3 :  C(1 : 64, J + 3) = B(1 : 64, J + 1) − Z
2      CONTINUE
23 CONTINUE
```

The other fragments of loops labeled by 21 and 22 can be further unrolled by HPFT to reduce the loop head overhead. In loop $L4''$, the elements of arrays A and B generated in vector registers are immediately reused in the same iteration instance and the reuse of array C occurs in the next running iteration. To exploit the reuse of array C, HPFT further reconstructs $L4''$ into the following program.

```
   DO  23  K = 3, 6
       TEMP(1 : 64) = C(1 : 64, K − 1)
       DO2  J = K, 639, 4
           S1 :  A(1 : 64, J) =  TEMP(1 : 64) + X
           S2 :  B(1 : 64, J + 1) = A(1 : 64, J) ∗ Y          (L4‴)
           S3 :  C(1 : 64, J + 3) = B(1 : 64, J + 1) − Z
           S3′ :  TEMP(1 : 64) = C(1 : 64, J + 3)
2      CONTINUE
23 CONTINUE
```

The HPFT automatically replaces array C in S_1 by an additional array variable $TEMP$. The Convex C3840 vector compiler is capable of eliminating the load operation of S_1 and the store operation of S_3' for array $TEMP$ since the variable $TEMP$ is loop invariant. Thus, the load operation for $TEMP$ of S_1 and the store operation for $TEMP$ of S_3' will only be performed one time during the execution of loop J running from K to 639. Loop $L4'''$ is superior than $L4''$ since the load operation of array C in statement S_1 of $L4''$ is saved in $L4'''$. The improvement of loop $L4'''$ in execution time is 14.65% compared to $L4''$ in Convex C3840. As a result, reuse opportunities of arrays A, B, and C are fully exploited by HPFT. Compared to the version of $L4$, loop $L4'''$ saves 45.71% execution time.

For a given complex EDG, the order of each node (or statement) applied by the index shift operation may effect the degree of reuse exploitation. Several heuristic rules [8] have been developed in **Select** procedure. These rules are designed for achieving the following two principles. First, HPFT selects node S that applying index shift operation Shift(S, $\bar{d}^e_{max}$, dir) on S will reduce the maximum number of edges to a distance of zero or one iteration gap and mark the least nodes with "unadjustable". Since applying index shift operation on an adjustable node can exploit at least one reuse opportunity, the adjustable node can be treated as a resource of reuse exploitation. Second, in vector register reuse consideration, we hope that the dependence distance can be reduced to zero such that the memory load operation can be saved. Even if some of the reuse distance can not be reduced to zero when applying the index shift operation on node S, the reduced nonzero distance can benefit cache or memory accessing since the reuse distance has been shortened. Thus, when applying the index shift operation on node S, we hope that the operation can benefit more dependence vectors in set $dir(S)$ by shortening their reuse distance and accumulate their distance on less dependence vectors belonging to $\overline{dir}(S)$.

The HPFT calls **Select** procedure to determine which node is suitable to be first selected to apply the shift operation. For example, consider the EDG shown in Fig. 10(a). HPFT first selects S_2 to apply index shift operation. Applying the index shift operation on S_2 to minimize dependence vector $\bar{d}^e_1$ first will exploit the reuse opportunity occurred by $\bar{d}^e_1$ and mark node S_2 with 'unadjustable'. Statement S_3 is still adjustable. However, applying index shift operation on S_3 first to minimize $\bar{d}^e_2$ will exploit the reuse opportunity occurred by $\bar{d}^e_2$ and mark nodes S_2 and S_3 with 'unadjustable'. This will result statements S_1, S_2, and S_3 are unadjustable. Since the adjustable nodes are the resources of reuse exploitation, the HPFT will first select node S_2 to apply the index shift operation. Similarly, in Fig. 10(b), selecting S_2 first to apply the index shift operation will exploit two reuse opportunities introduced by $\bar{d}^e_1$ and $\bar{d}^e_2$. In another way, selecting S_3 first to apply the index shift operation will exploit only one opportunity. Thus, HPFT selects the S_1 first.

In the reuse exploitation phase, HPFT translates the parsing tree into a form that reuse distance can be reduced. Other techniques related to reuse exploitation such as *loop unrolling* [24], *loop rerolling* [24], and *temp variable replacement* [7] are also designed to cooperate with the index shift process. The reuse exploitation designed in HPFT kernel also benefits to the multi-threading extraction as discussed in the latter subsection.

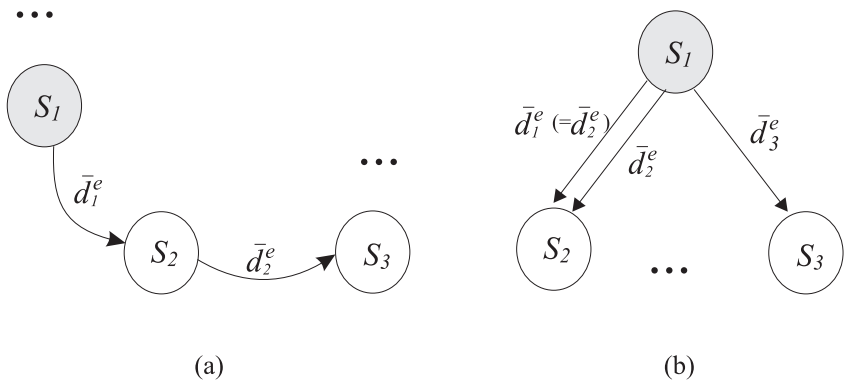


FIG. 10. *Applying index shift operation first on S_2 is better than on S_3. (a) Selecting S_2 first is better. (b) Selecting S_2 first is better.*

3.3. Multi-threading. Recent vector computers are equipped with several CPUs. Most of them offer both vectorization and multiprocessing capabilities. Several independent vector operations can be concurrently performed by multiple CPUs. Synchronizations are needed among these CPUs if their references of vector data have dependence relation. It is important to carefully partition the vector operations into threads such that these threads can be concurrently performed by multiple CPUs with fewer synchronizations and more reuse opportunities.

Let FORCE_PARALLEL be the compiler directive that forces compiler to parallelize the loop that follows. The directive introduced here are recognizable to compiler of Convex C38 series [10]. Similar instructions also can be found in compilers of other vector machines such as CRAY families [11]. As an example, consider the following program.

```
      DO  1  I1 = 1, n
          C$DIR FORCE_PARALLEL
          DO  2  I2 = 1, 4
              Loop Body
2         CONTINUE
1     CONTINUE
```

The directive FORCE_PARALLEL will force compiler to distribute four instances $I_2 = 1$, I_2=2, I_2=3, and I_2=4 on 4 CPUs to concurrently execute the loop body. Since loop I_1 is a sequential loop, synchronizations are needed among these CPUs to guarantee the sequential execution of loop I. In this example, there are n synchronizations needed among 4 CPUs.

Assume there are 4 CPUs in vector computers. Consider again the example of $L4$. One possible multi-threading version can be easily derived by partitioning the vector operations into 4 subsets as follows.

Example 5:

```
      DO  1  J = 3, 642
          C$DIR FORCE_PARALLEL
          DO  2  I = 1, 64, 16
              S1 :  A(I : I + 15, J) = C(I : I + 15, J − 1) + X
              S2 :  B(I : I + 15, J) = A(I : I + 15, J − 1) ∗ Y          (L5)
              S3 :  C(I : I + 15, J) = B(I : I + 15, J − 2) − Z
2         CONTINUE
1     CONTINUE
```

The partitioned 4 sets, $I = 1$, $I = 17$, $I = 33$, and $I = 49$ can be assigned to 4 CPUs for concurrent execution as shown in Fig. 11(a). Since the outermost loop J is a sequential loop, the 4 CPUs should be synchronized in each instance of J. Two disadvantages can be found in the multi-thread version of loop $L5$. First, there is no reuse opportunity exploited in vector register of each CPU. Second, there are 640 synchronizations which can be reduced in another multi-thread version translated by HPFT.

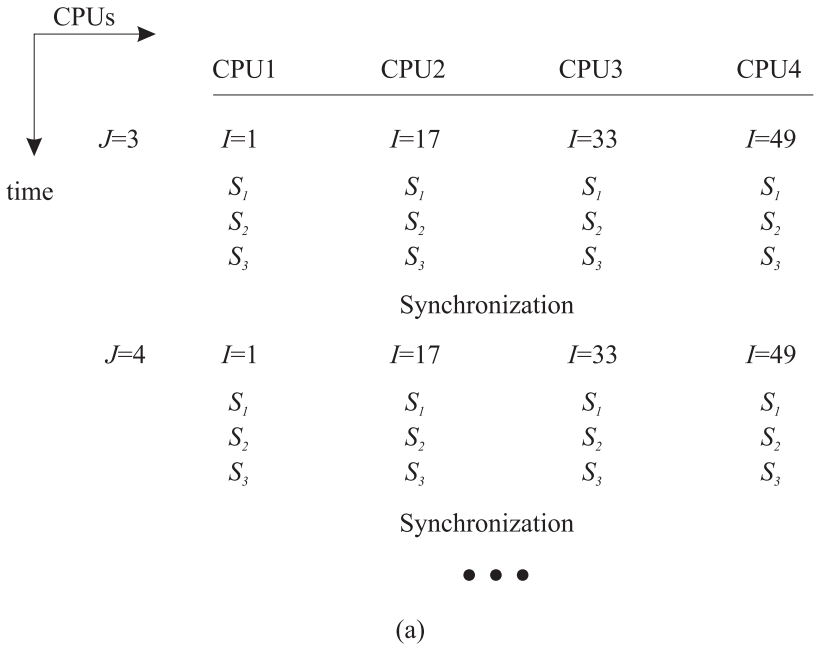
The HPFT first exploits the reuse opportunities of arrays A, B, and C into another version of loop $L4'''$ as discussed before. Then, HPFT transforms abstract syntax tree of loop $L4'''$ into another syntax tree equivalent to following multi-thread version.

```
      C$DIR FORCE_PARALLEL
      DO  23  K = 3, 6
          TEMP(1 : 64) = C(1 : 64, K − 1)
          DO2  J = K, 639, 4
              S1 :  A(1 : 64, J) = TEMP(1 : 64) + X
              S2 :  B(1 : 64, J + 1) = A(1 : 64, J) ∗ Y          (L5′)
              S3 :  C(1 : 64, J + 3) = B(1 : 64, J + 1) − Z
              S3′ :  TEMP(1 : 64) = C(1 : 64, J + 3)
2         CONTINUE
23    CONTINUE
```

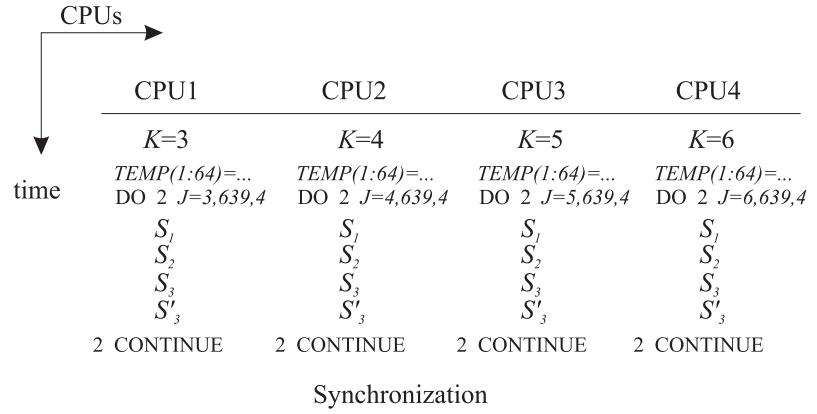
The execution of loop $L5'$ is shown in Fig. 11(b). The version of $L5'$ is superior than $L5$ in degree of reuse exploitation and the number of synchronizations. In loop $L5'$, reuse of arrays A, B, and C has been exploited. In total, there are 3*(639-3+1)=1911 vector loads saved. In addition, loop $L5'$ has only one synchronization. Compared to $L5$, loop $L5'$ saves 69.6% execution time.

If compiler applies loop interchange technique to loop $L5$, the number of synchronizations can be reduced to one. However, even if the loop interchange is possibly applied during multi-threading, in the following example, HPFT can also improve the execution time by reducing the number of synchronizations. Consider the following program which is extracted from vector benchmark of NETLIB in NCHC.

Example 6:



(a)



(b)

FIG. 11. *Multi-thread execution of loops L5 and L5′. (a) Execution of loop L5. (b) Execution of loop L5′.*

```
      EQUIVALENCE (EQV4(1,1), CC4(1,2))
      DO  1  J = 2, 80
          DO  2  I = 1, 90
              EQV4(I,J) = CC4(I+2, J+5) + Y(I,J)          (L6)
2         CONTINUE
1     CONTINUE
```

By setting '-O3' optimization switch to Convex C3840 vector compiler, we have the assembly code equivalent to the following code.

```
      DO  1  J = 2, 80
          C$DIR FORCE_PARALLEL
          DO  2  I = 1, 90, 23
              EQV4(I : I+22, J) = EQV4(I+2 : I+24, J+4) + Y(I : I+22, J)   (L6')
2         CONTINUE
1     CONTINUE
```

An attempt to interchange loops I and J will cause semantic error. The reason is stated as follows. The vector data $EQV4(3:25,6)$ are used by a CPU, say P_1, at the execution of $J = 2$. The vector data $EQV4(24:46,6)$ will be generated by another CPU, say P_2, at the execution of $J = 6$. To guarantee that the data element $EQV4(24,6)$ used by P_1 is old, the loop J should be kept in the outermost position to ensure the sequential execution. Partitioning vector length 90 into 4 sets produces 79 synchronizations. In the vectorization phase, the HPFT will translate the loop $L6$ into the following vector form.

```
      DO  1  J = 2, 80
          EQV4(1:90, J) = EQV4(3:92, J+4) + Y(1:90, J)
1     CONTINUE
```

Then, in the multi-threading phase, the HPFT translates the loop $L6$ into multi-thread execution form as follows.

```
      C$DIR FORCE_PARALLEL
      DO  1  K = 2, 5
          DO  2  J = K, 80, 4
              EQV4(1:90, J) = EQV4(3:92, J+4) + Y(1:90, J)        (L6'')
2         CONTINUE
1     CONTINUE
```

Only one synchronization is needed in loop $L6''$. Compared to $L6'$, the loop $L6''$ has fewer synchronizations. The translated HPFT version avoids the dependence relation caused by sectioning the length of vector operations.

In what follows, we list the algorithm of reuse exploitation and multi-threading for each *π-block*.

Algorithm: Reuse exploitation and multi-threading

Input: The *π-block* $B(N,E)$ where $N = \{S_i, 1 \le i \le m\}$ and the associated parsing tree.

Output: The reduced *π-block* and a reconstructed parsing tree.

/* reuse exploitation */
For each node $S_i, 1 \leq i \leq m$, do
  Set Adjustable(S_i) = True.
Endfor
While there exists an adjustable node in EDG do
  Call **Select**(B) to select an adjustable node S from B and
  determine the value of dir.
  Apply **Shift**(S, $\bar{d}^e_{max}, dir$) on node S.
  Modify the running iteration range and variables' subscript of node S
  according to the vector $\bar{d}^e_{max}$.
Endwhile
According to the intersection of running iteration range of S_i, $1 \leq i \leq m$, apply
loop distribution and loop unrolling to statements $S_i, 1 \leq i \leq m$, in parsing tree.
/* Loop spreading and multi-threading */
Let B' denote the reuse exploited π-$block$.
Let f be the level of loops that all statements $S_i \in N, 1 \leq i \leq m$, are enclosed.
Let there be h extended dependence vectors $\bar{d}^e_j = (d_{j1}, d_{j2}, \ldots, d_{j(n+1)}), 1 \leq j \leq h$,
existed in B'.
For $k = 1$ to f
  $p_k = \gcd(d_{jk})$, for all $1 \leq j \leq h$.
Endfor
Select maximum value p_m from set $\{p_1, p_2, \ldots, p_f\}$ such that interchanging loop
in level m to
the outermost position is valid.
Interchange the mth level loop to the outermost loop in parsing tree.
Apply loop spreading on the parsing tree and automatically insert $TEMP$
variable to increase the degree of reuse exploiting.
Insert multi-threading directive in parsing tree such that the generated program
has directive FORCE_PARALLEL before the outermost loop.

For a given program, the HPFT first performs the π-$block$ decomposition and vectorization such that maximum benefits can be obtained for reuse exploitation. Then, HPFT further inserts the directives to explicitly define more or better vector operations from sequential program. In the reuse exploitation phase, the HPFT reconstructs the original program such that the number of vector load operations can be reduced. In final, the HPFT inserts directives to define the multi-thread with less number of synchronizations and generates another high performance version of loops for vector compilers. As discussed in the next section, with the assistance of HPFT, compiler of Convex C3840 usually obtains a better performance.

4. Experimental results. In this section, the performance improvement of sequential programs translated by HPFT is measured on Convex C3840 supercomputer. The current version of Fortran compiler on Convex C3840 system is 7.0. Two versions of program are compared. For the first version, we take original program written in Fortran 77 as the input of vector compiler of Convex. The object code generated by vector compiler is referred to the *original version*. Instead, another version is generated by the following two steps. First, the sequential program is

taken as the input of HPFT. The high performance code translated by HPFT is then taken as the input of vector compiler of Convex C3840 in the second step. The generated object code by these two steps is referred to the *HPFT version*.

Loops selected as the original programs for test can be roughly cataloged into three classes. The first class is the vector benchmarks that are extracted from NETLIB of NCHC (National Center for High Performance Computing). Two benchmarks are measured. The first benchmark (referred as benchmark 1) consists of 107 subroutines of loops that are originally designed for testing the vectorization capability of PFC [2, 3]. In total, there are 65 subroutines can be vectorized by vector compiler of Convex. The 65 subroutines are considered as the original programs and the execution time of two versions, the original version and the HPFT version, is compared. In total, there are 11 subroutines improved by applying the HPFT system.

The second benchmark (referred as benchmark 2) consists of 45 simple vectorization tests, 50 subscript tests, 8 tests of rearranging the structure of nested loops, 36 tests involving branching, 35 ambiguity checking, 28 tests for external routines, and 49 tests for others. Because that the 45 tests of simple vectorization focus mainly on the testing of general vectorization capabilities, for most current vector compiler, they can obtain a good performance without the assistance of HPFT. Only 8 rearranging the structure of nests of loops that are related to the main focus of HPFT design are selected to be tested. Two loops of the 8 selected loops have significant improvement in execution time for HPFT versions.

The second class selected as the original programs consists of several libraries including subroutines of BLAS1 and BLAS2. The level 1 BLAS (Basic Linear Algebra Subprograms) and level 2 BLAS respectively perform the vector/vector and matrix/vector operations. All subroutines of BLAS1 and BLAS2 are designed in libraries for calls in most supercomputers. The subroutines of BLAS1 and BLAS2 used as the original 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 [17, 22]. In what follows, only those that are restructured or the directives are inserted by HPFT are listed in Table 1.

The experimental results of execution time and speedup for these three classes of programs are summarized in Table 2. For the three measured classes, two versions of several subroutines that have the same execution time (HPFT is not activated when translating these subroutines) are not listed in Table 2.

Compared with the original version, the main factors that the HPFT version improves are classified into vectorization, memory conflict, locality, and the number of synchronizations. The effect of these factors is analyzed as shown in Table 3. In vectorization, the HPFT assists vector compiler of Convex C3840 to extract more or better vector operations from several subroutines. For instance, two statements S_1 and S_2 of subroutine S084 will be decomposed into two π-*blocks* by HPFT process. Then the HPFT vectorizes the first dimension and second dimension of S_1 and S_2, respectively. However, these two statements of original version are not decomposed. The dependence relation existed in S_2 thus makes S_1 sequential execution.

Some programs such as S029, S030, S084, Loop 1 and Loop 2 of vector benchmark 2, FLSA, SLEGE2, and FDFT are vectorized in the second dimension of array

TABLE 1
Applications and it's abbreviation.

Applications	Abbreviation
Fourier Least-Squares Approximation [17]	FLSA
Jacobi Method for Solution of Linear Equations [17]	Jacobi
Barycentric Form of Lagrange Interpolation [17]	BFLI
Accumulating a Sum (A. Sum) [22]	ASum
Solving Linear Equation by Gaussian Elimination (loop 1) [22]	SLEGE1
Solving Linear Equation by Gaussian Elimination (loop 2) [22]	SLEGE2
Computing the Uniform Norm of Matrix A and A Inverse [22]	Uniform norm
Gauss-Seidel Iterations [22]	GSI
Comparing Compound Simposon's and Newton-Cotes Integration [22]	CSNCI
Computing The Value of A Filtered Discrete Fourier Transform [22]	FDFT

operations. The HPFT performs the memory conflict reduction scheme to reorganize the memory allocation. Consequently, array accessing when performing vector operations has low frequency of memory conflict. The speedup of these programs thus has significant improvement as found in Table 2.

In the reuse exploitation, the HPFT performs *index shifting*, *temp vector variable*, and *loop spreading* schemes to exploit the reuse opportunities. The reuse exploitation reduces the number of loads from shared memory to vector registers and frees the load/store functional units for other use. Execution time of subroutines S022, S023, S029, S030, S047, S048, S049, S084, S100, loops 1 and 2 in benchmark 2, BLAS2, and application programs are thus improved. In synchronization, the execution of HPFT version has fewer synchronizations among CPUs than one of the original version for some subroutines and programs.

The HPFT system extracts not only the implicit vector operations but also the reuse opportunities for a sequential program written in Fortran 77. The exploited vector operations and multi-thread are explicitly defined for vector compilers by automatically inserting proper compiler directives in the translated code. Experimental results show that vector compiler cooperated with the HPFT system usually produces a more efficient code for users to early complete their program execution.

5. Conclusion. In this paper, we have described the design and implementation of HPFT. The HPFT performs source-to-source translation and code tuning from a sequential program into a favorite execution form. Execution of sequential programs thus can be earlier completed due to the benefits of more vector operations, higher degree of reuse exploitation, and fewer synchronizations among CPUs. Performance evaluator and menu-driven user interface are also designed in HPFT system to offer users a friendly environment.

TABLE 2
Comparisons of original version and HPFT version for benchmarks, libraries and applications.

loop programs		Problem Size	Original	HPFT	speedup
Bench 1	S022	1024 × 1024	8.5 ms	5.3 ms	1.6
	S023	1024 × 1024	12.8 ms	9.5 ms	1.3
	S029	1024 × 1024	7.5 ms	0.3 ms	25
	S030	1024 × 1024	23.88 ms	3.64 ms	6.56
	S044	1048576	5.55 ms	3.0 ms	1.85
	S045	1048576	8.51 ms	5.10 ms	1.66
	S047	1024 × 1024	7.12 ms	5.72 ms	1.24
	S048	1024 × 1024	6.99 ms	5.75 ms	1.22
	S049	1024 × 1024	6.96 ms	5.63 ms	1.23
	S084	1024 × 1024	23.9 ms	1.58 ms	15.1
	S100	1024 × 1024	219 ms	153 ms	1.43
Bench 2	LOOP 1	1024 × 1024	72 ms	3 ms	24
	LOOP 2	1024 × 1024	78 ms	7 ms	11.14
BLAS1	SAXPY	1048576	10 ms	4 ms	2.5
	SCOPY	1048576	10 ms	3 ms	3.33
	SSCAL	1048576	6 ms	2 ms	3
	DAXPY	1048576	19 ms	9 ms	2.1
	DCOPY	1048576	17 ms	6 ms	2.8
	DSCAL	1048576	12 ms	5 ms	2.4
BLAS2	SGEMV	1024 × 1024	6 ms	5 ms	1.2
	SGBMV	1024 × 1024	6 ms	5 ms	1.2
	DGEMV	1024 × 1024	11.5 ms	9.8 ms	1.17
	DGBMV	1024 × 1024	12.1 ms	9.9 ms	1.2
App.s	FLSA	1024	32.1 ms	5.1 ms	6.29
	Jacobi	1024 × 1024	16 ms	13 ms	1.23
	BFLI	1024 × 1024	31 ms	27 ms	1.15
	ASum	1024	3.8 ms	2.6 ms	1.46
	SLEGE1	1024 × 1024	7.8 ms	6.5 ms	1.2
	SLEGE2	1024 × 1024	55 ms	7 ms	7.85
	Uniform norm	1024 × 1024	74 ms	58 ms	1.27
	GSI	1024 × 1024	6.3 ms	4.8 ms	1.31
	CSNCI	1024 × 1024	53 ms	5.7 ms	9.29
	FDFT	1024 × 1024	57 ms	5 ms	11.4

TABLE 3
Factors responsible for the effect of HPFT version.

loop programs	Vectorization	mem. conflict	Localities	Synchronizations
S022			√	
S023			√	
S029		√	√	√
S030		√	√	
S044	√			
S045	√			
S047			√	√
S048			√	√
S049			√	√
S084	√	√	√	√
S100			√	
Loop 1		√	√	
Loop 2		√	√	
SAXPY	√			
SCOPY	√			
SSCAL	√			
DAXPY	√			
DCOPY	√			
DSCAL	√			
SGEMV			√	√
SGBMV			√	√
DGEMV			√	√
DGBMV			√	√
FLSA	√	√	√	√
Jacobi			√	
BFLI			√	
ASum			√	
SLEGE1			√	
SLEGE2		√	√	√
Uniform norm			√	
GSI			√	√
CSNCI			√	
FDFT		√	√	√

Experiments with benchmarks, libraries, and scientific applications show that HPFT enhances the capabilities of current vector compiler of Convex C3840 supercomputer in vectorization, reuse exploitation, and multi-threading. Further measuring of improvement of HPFT is currently studied for these benchmarks and real applications on other machines such as CRAY family and IBM ES/9000. The developed HPFT system plays a role of assistant of current vector compilers. In cooperation with the HPFT system, vector compilers usually generate more efficient object code for user to speed up their program execution.

Acknowledgements. This work was supported by the National Science Council of the Republic of China under grant Nsc 84-2213-E-008-010. This paper has received the 1994 Long-Term Thesis Award.

REFERENCES

[1] J. R. Allen, D. Callahan, and K. Kennedy, *Automatic decomposition of scientific programs for parallel execution*, in Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages, 1987, pp. 63–76.

[2] R. Allen and K. Kennedy, *PFC: a program to convert Fortran to parallel form*, in Proceeding of IBM Conference on Parallel Computing and Scientific Computation, 1982.

[3] ———, *Automatic translation of Fortran programs to vector form*, ACM Transactions on Programming Languages and Systems, 9 (1987), pp. 491–542.

[4] ———, *Vector register allocation*, IEEE Transactions on Computers, 41 (1992), pp. 1290–1317.

[5] U. Banerjee, *Dependence Analysis for Supercomputing*, Kuwer Academic Publishers, Norwell, Massachusetts, 1988.

[6] T. Brandes, *Automatic translation of data parallel programs to message passing programs*, Internam Report Adaptor 93-1, Jan. 1993.

[7] 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, June 1990, pp. 53–65.

[8] C. Y. Chang and J. P. Sheu, *Vectorizing do-loops and exploiting data reuse for vector computers*, Technical Report 93-6, Department of Computer Science and Information Engineering, National Central University, 1993.

[9] ———, *Compile-time scheduling of multithread with data localities on multiple vector processors*, Concurrency: Practice and Experience, 7 (1995), pp. 349–369.

[10] Convex Computer Corporation, Richardson, FX, *Fortran optimization Guide*, 1990.

[11] Cray Research Inc., *CF77 volumn 4: parallel processing guide*, 1981.

[12] J. J. Dongarra and S. C. Eisenstat, *Squeezing the most out of an algorithm in Cray Fortran*, ACM Transactions on Mathematical Software, 10 (1984), pp. 221–230.

[13] D. Heller, *Motif programming manual*, O'Reilly and Associates, Inc., 1991.

[14] IBM corporation, *IBM AIX VS FORTRAN/ESA programming guide, Version 1, Release 1*, 1992.

[15] L. S. Liu, C. W. Ho, and J. P. Sheu, *On the parallelism of nested for-loops using index shift method*, in Proceedings of International Conference on Parallel Processing, vol. II, Aug. 1990, pp. 119–123.

[16] D. A. Padua and M. J. Wolfe, *Advanced compiler optimizations for supercomputers*, Communications of the ACM, 29 (1986), pp. 1184–1201.

[17] S. M. Pizer and V. L. Wallace, *To compute numerically concepts and strategies*, Little, Brown and Company.

[18] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leng, and D. Schouten, *Parafrase-2: an environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors*, in Proceedings of International Conference on Parallel Processing, vol. II, Aug. 1989, pp. 39–48.

[19] J. P. Sheu and C. Y. Chang, *Synthesizing nested loop algorithms using nonlinear transformation method*, IEEE Transactions on Parallel and Distributed Systems, 2 (1991), pp. 304–317.

[20] R. E. Tarjan, *Depth first search and linear graph algorithms*, SIAM. J. Computing, 1 (1972), pp. 146–160.

[21] 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, June 1991, pp. 30–44.

[22] S. Yakowitz and F. Szidarovszky, *An introduction to numerical computations*, Macmillan Publishing Company, New York, second edition ed., 1989.

[23] H. P. Zima, H.-J. Bast, and M. Gerndt, *SUPERB: A tool for semi-automatic MIMD/SIMD parallelization*, Parallel Computing, 6 (1988), pp. 1–18.

[24] H. P. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1991.