Common Optimization Techniques [6]

Inlining

Pushing a loop in a subroutine or function

Loop unwinding

Loop segmentation

Loop jamming

Inverting loops to get maximum vector length

Inverting loops to avoid a recurrence

Linearizing loops

Converting scalar recurrence to vector temporary

Positioning conditional blocks in a scalar loop

Inlining
6.1

In "modular" programs, some subroutines and functions may perform few computations before returning to the calling routine. In many of these cases, the overhead to transfer control to the subprogram takes longer than the actual subprogram execution. For example, it can take up to 75 clock periods to call a subroutine with no arguments passed. With just one argument, that overhead nearly doubles. Therefore, for very small subprograms it is better to move the code in line. Three programming techniques that can significantly improve performance in these situations are illustrated in this section; they are:

Pulling the contents of a subprogram into the calling program

Replacing a one line function with a statement function or a preprocessor macro

Pushing the looping structure into the called subprogram

Also, note that the SCC and CF77 provide command–line options and directives for controlling inline code expansion of called subprograms.

Note to C Programmers Regarding Preprocessor Macros
6.1.1

Recall that when using preprocessor macros, the code is expanded inline and the arguments are replaced token by token, thus the use of auto–increment and auto–decrement operators may yield unexpected results. Also, liberal use of parentheses is recommended.

Consider the erroneous side effects of defining a preprocessor macro to cube a single argument:

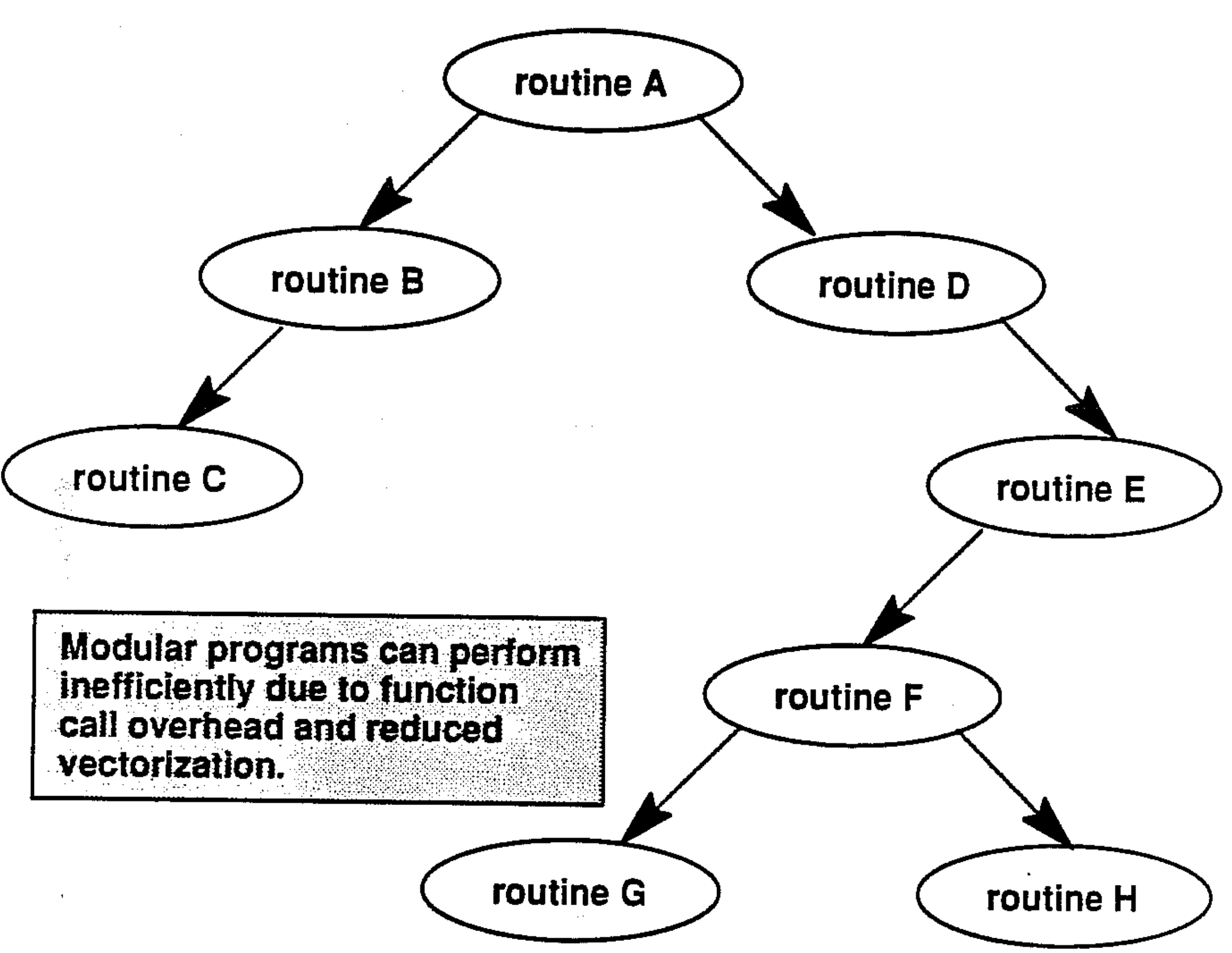
```
#define cube(x)   ((x)*(x)*(x))
   cube(i++);
```

The reference to cube is expanded as follows:

```
((i++)*(i++)*(i++));
```

With the preceding macro expansions, the cube of `i`, where `i=3` for example, would be `3*4*5` (or 60) instead of `3*3*3 (27)`.

Subprogram Inlining



Inlining, automatic or manual, reduces function call overhead and increases the possibility for vectorization.

The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000005	0.000001	8.8058
64	0.000066	0.000001	70.7949
65	0.000067	0.000001	55.8600
500	0.000515	0.000004	127.0489
1000	0.001029	0.000008	135.5032

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000001	6.7550
64	0.000050	0.000001	65.7650
65	0.000051	0.000001	68.3799
500	0.000389	0.000002	208.1161
1000	0.000774	0.000003	243.9461

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000025	0.000003	10.0238
64	0.000312	0.000004	75.9197
65	0.000318	0.000005	58.2033
500	0.002427	0.000022	109.1795
1000	0.004855	0.000043	112.0672

CRAY-2 timings

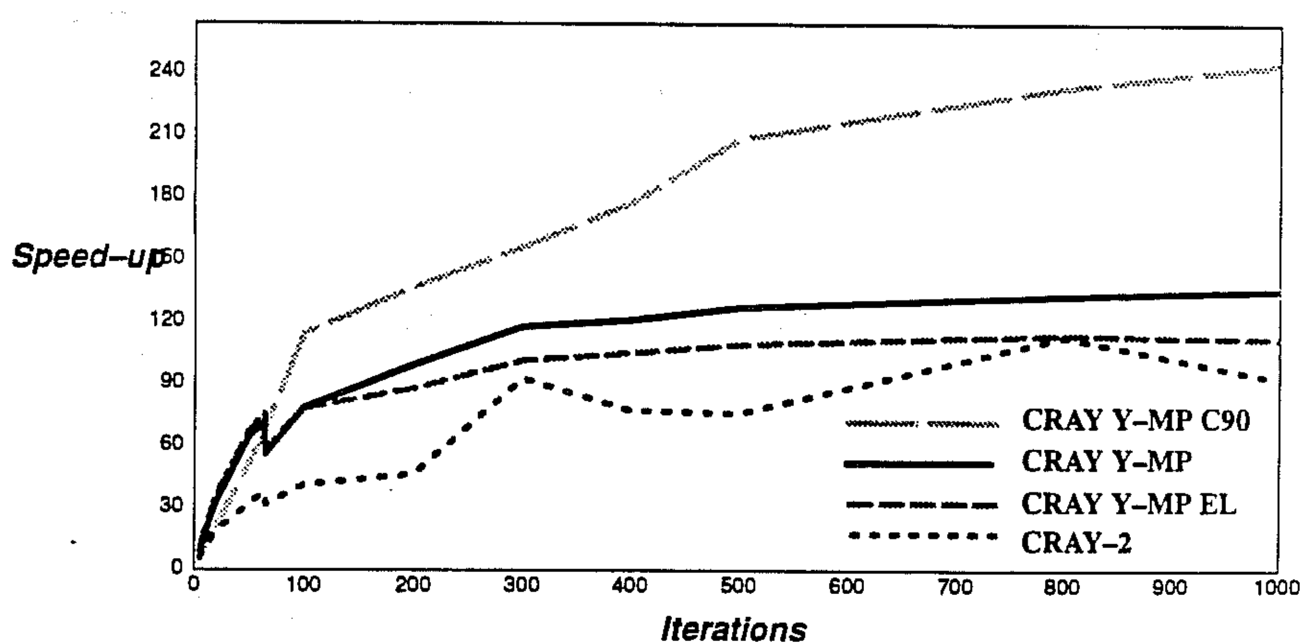
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000001	5.4206
64	0.000082	0.000002	37.2910
65	0.000083	0.000003	31.4969
500	0.000634	0.000008	75.8177
1000	0.001269	0.000014	92.6092

Inlining a Function *(in Fortran with "-o inline1")*

```
      REAL FUNCTION ORIGINAL(N)
      REAL    A(1000),B(1000),CELSIUS1
CDIR$ NOINLINE
      DO 10 I=1,N
10      CALL CELSIUS(A(I), B(I))
      END
cccccccccccccccccccccccccccccccccccccccccccccc
      SUBROUTINE CELSIUS(C, F)
      REAL    C,F
      C = (F - 32.0) * 5.0/9.0
      END
cccccccccccccccccccccccccccccccccccccccccccccc
      REAL FUNCTION MODIFIED(N)
      REAL    A(1000), B(1000), CELSIUS2, F
CDIR$ INLINE
      DO 10 I=1,N
10      CALL CELSIUS(A(I), B(I))
      END
```

Not vectorized due to subroutine call.

Vectorized; no subroutine call overhead.



The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000005	0.000001	6.5738
64	0.000059	0.000001	58.2738
65	0.000060	0.000001	44.1733
500	0.000457	0.000004	110.6802
1000	0.000914	0.000008	119.2725

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000001	7.3226
64	0.000046	0.000001	72.4868
65	0.000046	0.000001	79.0355
500	0.000356	0.000002	190.5848
1000	0.000710	0.000003	226.4867

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000023	0.000003	7.0841
64	0.000280	0.000005	61.9007
65	0.000285	0.000006	46.5294
500	0.002188	0.000022	97.4933
1000	0.004359	0.000043	101.8143

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000001	4.8358
64	0.000079	0.000002	35.1376
65	0.000080	0.000003	27.9297
500	0.000615	0.000009	69.2406
1000	0.001225	0.000014	90.5073

Inlining a Function *(in C)*

```
int a[1000], b[1000];
float original(n)
int n;
{
int i;
    for(i=0;i<n;i++)
        celcius(i);
}
/*******************************************/
celcius(i)
int i;
{
    a[i] = (b[i] - 32.0) * 5.0/9.0;
}
/*******************************************/
float modified(n)
int n;
{
int i;
#pragma inline celcius
    for(i=0;i<n;i++)
        celcius(i);
}
```

Not vectorized due to function call overhead.

Vectorized; no function call overhead.

Speed–up (0–240) vs. *Iterations* (0–1000)

CRAY Y-MP C90
CRAY Y-MP
CRAY Y-MP EL
CRAY-2

Pushing a Loop Into a Subroutine or Function
6.2

The examples shown on the previous pages illustrated an optimization technique that can be applied to loops that do not vectorize due to a subroutine or function call. The technique, called subprogram inlining, effectively "pulled" all the work performed by the called function into the loop, thereby allowing vectorization.

An alternative to subprogram inlining, or pulling the work performed by the called function into the loop, is pushing the loop into the called function. With this technique, routine A still contains a function call. However, the called function effectively does all the work, and the loop within the function may vectorize.

The example on the next page illustrates this technique.

The benefits of this technique are twofold:

1) The work within the loop may vectorize when the subroutine or function reference is removed.

2) The modified code has fewer external references. In the original code, each iteration of the (non–vectorizing) loop makes an external reference. In the modified code, there is only one external reference (to a routine that does all the work).

Fortran and C versions of this technique follow, along with comparative timing comparisons.

Pushing a loop into a subroutine or function

Loop with 1 function call per iteration; the loop in the calling routine is not vectorized.

routine A

```
...
DO I=1,N
   ...
   CALL SUB(...)
   ...
END DO
...
```

routine A

```
...
for (i=0;i<N;i++) {
...
   SUB(...);
...
}
```

SUB

```
...
vectorizable work
...
```

1 function call with entire loop in the function. All the work then vectorizes.

The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000006	0.000001	4.0087
64	0.000067	0.000002	41.2325
65	0.000068	0.000002	36.4887
500	0.000522	0.000005	101.2070
1000	0.001044	0.000009	113.0851

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000001	3.2727
64	0.000049	0.000001	42.6259
65	0.000050	0.000001	43.2878
500	0.000384	0.000002	175.0456
1000	0.000767	0.000003	222.0434

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000025	0.000007	3.4403
64	0.000314	0.000008	40.5765
65	0.000319	0.000009	34.6546
500	0.002439	0.000026	93.0590
1000	0.005270	0.000048	109.8125

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000002	3.2011
64	0.000084	0.000003	30.2311
65	0.000085	0.000003	25.9038
500	0.000650	0.000012	52.7496
1000	0.001300	0.000023	55.8405

Pushing a Loop into a Subroutine *(in Fortran)*

```
      REAL FUNCTION ORIGINAL(N)
       REAL    A(1000),B(1000)
       DO 10 I=1,N
10       CALL CELSIUS1(A(I), B(I))
      END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE CELSIUS1(C, F)
       REAL    C,F
       C = (F - 32.0) * 5.0/9.0
      END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION MODIFIED(N)
       REAL    A(1000), B(1000)
       CALL CELSIUS2(N, A, B)
      END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE CELSIUS2(N, C, F)
       INTEGER N
       REAL    C(N), F(N)
       DO 10 I=1,N
10        C(I) = (F(I) - 32.0) * 5.0/9.0
      END
```

Not vectorized due to function call overhead.

Work is vectorized.

CRAY Y-MP C90
CRAY Y-MP
CRAY Y-MP EL
CRAY-2

Speed-up: 0, 30, 60, 90, 120, 150, 180, 210

Iterations: 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000

The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000005	0.000001	3.6082
64	0.000066	0.000002	27.6491
65	0.000067	0.000003	24.3565
500	0.000516	0.000012	44.0697
1000	0.001032	0.000022	46.0696

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000001	3.9073
64	0.000050	0.000001	38.9739
65	0.000051	0.000001	39.3981
500	0.000388	0.000004	87.8763
1000	0.000776	0.000008	95.9263

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000026	0.000006	4.6126
64	0.000324	0.000009	37.6794
65	0.000329	0.000010	32.7791
500	0.002535	0.000031	80.8593
1000	0.005055	0.000058	87.2189

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000003	2.6016
64	0.000084	0.000004	22.9786
65	0.000085	0.000005	18.6310
500	0.000650	0.000014	47.5402
1000	0.001300	0.000022	59.6714

Pushing a Loop into a Function *(in C)*

```
float original(n)
int n;
{
float celsius1();
int i;
   for (i=0;i<n;i++)
        a[i] = celsius1(b[i]);
}
/*******************************************/
float celsius1(f)
float f;
{return((f -32.0) * 5.0/9.0);}
/*******************************************/
float modified(n)
int n;
{
void celsius2();
     celsius2(n, a, b);
}
/******************************************/
void celsius2(n, c, f)
int n;
float c[], f[];
{
int i;
#pragma _CRI ivdep
   for(i=0;i<n;i++)
      c[i] = (f[i] - 32.0) * 5.0/9.0;}
```

Not vectorized due to function call overhead.

Work is vectorized.

Speed-up

100
80
60
40
20
0

0 100 200 300 400 500 600 700 800 900 1000

Iterations

CRAY Y-MP C90
CRAY Y-MP
CRAY Y-MP EL
CRAY-2

CF77 Loop Unwinding

6.3

Given a nest of loops, if the trip count of the inner loop(s) is at most 5, then CF77 will unwind (eliminate) the inner loops and replaces them with straight–line code. This enables the outer loop, to vectorize. CF77 will issue messages when a loop is unwound. For example, consider the following code segment:

```
      DO 20 J = 1 , 100
         DO 10 I = 1 , 4
            A(I,J) = I * J
10       CONTINUE
20    CONTINUE
```

CF77 will unwind the `DO 10` loop to produce the following psuedocode:

```
      DO 20 J = 1 , 100
         A(1,J) = 1 * J
         A(2,J) = 2 * J
         A(3,J) = 3 * J
         A(4,J) = 4 * J
20    CONTINUE
```

Now, loop 20 can vectorize with a vector length of 100, whereas in the original form, loop 20 could not have vectorized.

Note

CF77 does not perform source–to–source transformation for unwinding. You can only view the transformed code from the assembly program generated by CF77.

Loop Unwinding

Outer loop that does not vectorize:

DO I=1,N

Inner loop that does vectorize with vector length = 4:

DO J=1,4

END DO

END DO

DO I=1,N

Unwound (eliminated) inner loop; resulting loop vectorizes with vector length = N:

4 iterations unwound

END DO

Loop Unrolling
6.4

Loop unrolling is similar to loop unwinding, however, the trip count of the loop to be unrolled can be anything. Both CF77 and SCC automatically unroll loops. However, you can manually unroll loops before compiling them. The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000002	0.000001	1.5751
64	0.000021	0.000003	6.9180
65	0.000021	0.000004	4.8643
500	0.000159	0.000023	6.8492
1000	0.000318	0.000045	7.0217

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000001	0.000001	1.2250
64	0.000011	0.000002	6.7074
65	0.000011	0.000002	6.4638
500	0.000084	0.000010	8.0208
1000	0.000167	0.000020	8.2597

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000007	0.000006	1.2386
64	0.000070	0.000018	3.7875
65	0.000070	0.000024	2.9517
500	0.000527	0.000145	3.6356
1000	0.001062	0.000291	3.6477

CRAY-2 timings

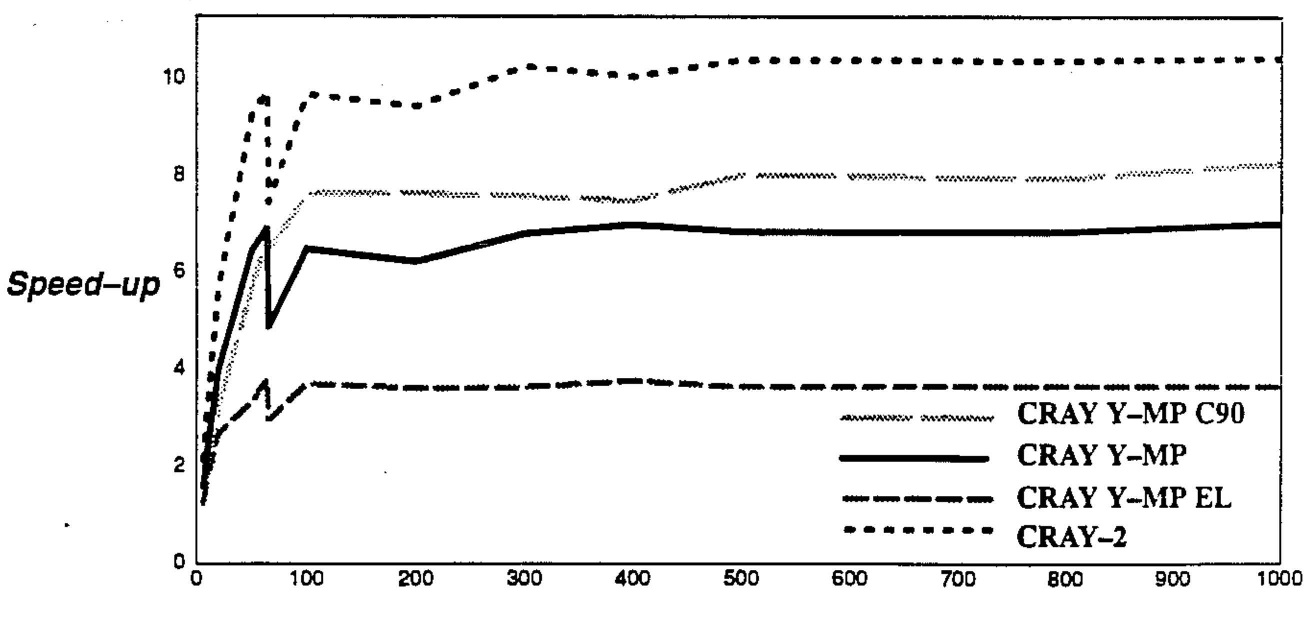
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000006	0.000003	2.1223
64	0.000056	0.000006	9.7348
65	0.000057	0.000008	7.4421
500	0.000437	0.000042	10.3964
1000	0.000874	0.000084	10.4371

Manual Loop Unrolling *(in Fortran)*

```
      REAL FUNCTION ORIGINAL(N)
      REAL    A(1000,6), B(1000,6), C(1000,6)

      DO 10 I=1,N
         DO 10 J=1,6
10          A(I,J) = B(I,J) * C(I,J)
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION MODIFIED(N)
      REAL    A(1000,6), B(1000,6), C(1000,6)

       DO 10 I=1,N
         A(I,1) = B(I,1) * C(I,1)
         A(I,2) = B(I,2) * C(I,2)
         A(I,3) = B(I,3) * C(I,3)
         A(I,4) = B(I,4) * C(I,4)
         A(I,5) = B(I,5) * C(I,5)
         A(I,6) = B(I,6) * C(I,6)
10     CONTINUE
```

Short vector loop (VL=6).

Inner loop unrolled; full vectorization. (VL=N).



Iterations

The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000002	0.000001	1.4530
64	0.000020	0.000003	6.2964
65	0.000020	0.000004	4.8136
500	0.000156	0.000022	7.0767
1000	0.000312	0.000046	6.7446

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000001	0.000001	1.4385
64	0.000014	0.000002	8.4725
65	0.000014	0.000002	8.6675
500	0.000109	0.000010	10.5254
1000	0.000217	0.000021	10.5046

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000008	0.000008	1.0349
64	0.000081	0.000020	4.0656
65	0.000081	0.000026	3.1724
500	0.000623	0.000144	4.3396
1000	0.001244	0.000292	4.2641

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
6	0.000005	0.000003	2.1653
64	0.000056	0.000006	9.8136
65	0.000057	0.000008	7.5314
500	0.000437	0.000042	10.3949
1000	0.000874	0.000084	10.4425

Manual Loop Unrolling *(in C)*

```
float a[6][1000], b[6][1000], c[6][1000];
float original(n)
int n;
{
int i, j;
    for (j=0;j<n;j++)
        for (i=0;i<6;i++)
            a[i][j] = b[i][j] * c[i][j];
}
```

Short vector loop (VL=6).

```
/*************************************/
float modified(n)
int n;
{
int i, j;
    for (i=0;i<n;i++) {
        a[0][i] = b[0][i] * c[0][i];
        a[1][i] = b[1][i] * c[1][i];
        a[2][i] = b[2][i] * c[2][i];
        a[3][i] = b[3][i] * c[3][i];
        a[4][i] = b[4][i] * c[4][i];
        a[5][i] = b[5][i] * c[5][i];
    }
}
```

Inner loop unrolled, full vectorization. (VL=n).

Speed-up
0 2 4 6 8 10
0 100 200 300 400 500 600 700 800 900 1000
Iterations

CRAY Y-MP C90
CRAY Y-MP
CRAY Y-MP EL
CRAY-2

Segmenting a Non–vectorizing Loop
6.5

When a true dependency conflict is encountered within a critical loop, and the dependency cannot be removed, performance may still be improved by minimizing the impact of the dependency. Many loops that contain code with a dependency conflict also contain code that is unrelated to that dependency and is, in itself, vectorizable. One way to minimize the effects of dependency is to isolate the dependent code into a separate loop, creating two loops: one that vectorizes, and one that does not.

Fortran and C versions of this technique follow, along with comparative timing comparisons.

Segmenting a Non–vectorizing Loop

Looping construct that does not vectorize due to:

embedded I/O
data dependency
etc.

can be segmented to take advantage of vectorization.

```
DO I=1,N
 vectorizable work

 scalar work

 END DO
```

```
for (i=0;i<N;i++) {
 vectorizable work

 scalar work

}
```

```
DO I=1,N
 vector loop
END DO

DO I=1,N
 scalar loop
END DO
```

```
for (i=0;i<N;i++) {
 vector loop
}

for (i=0;i<N;i++) {
 scalar loop
}
```

The following output was generated from the Fortran program shown on the facing page.

```
                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAYY-MP               5       0.000006    0.000004    1.7633
timings               50       0.000071    0.000008    9.0381
                      65       0.000051    0.000012    4.3163
                     500       0.000351    0.000069    5.0667
                    1000       0.000702    0.000138    5.0912

                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAYYMP-C90            5       0.000004    0.000003    1.4892
timings               50       0.000047    0.000005    8.9322
                      65       0.000038    0.000008    4.9252
                     500       0.000272    0.000036    7.5948
                    1000       0.000544    0.000070    7.7403

                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAYYMP-EL             5       0.000029    0.000016    1.8958
timings               50       0.000345    0.000041    8.4838
                      65       0.000227    0.000059    3.8638
                     500       0.001545    0.000372    4.1559
                    1000       0.003092    0.000752    4.1133

                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAY-2                 5       0.000003    0.000004    0.9286
timings               50       0.000022    0.000008    2.6437
                      65       0.000029    0.000014    2.1810
                     500       0.000218    0.000064    3.3923
                    1000       0.000436    0.000127    3.4401
```

Segmenting a Non–vectorizing Loop *(in Fortran)*

```
      REAL FUNCTION ORIGINAL(N)
      REAL    A(1000), B(1000), C(1000)
* SINGLE NON-VECTORIZING LOOP
      DO 10 I=2,N
         A(I) = A(I-1) * C(I)
         B(I) = A(I) * C(I-1)**2
         C(I-1) = SQRT(A(I)) - B(I)
10    CONTINUE
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION MODIFIED(N)
      REAL    A(1000), B(1000), C(1000)
* SINGLE LOOP SEGMENTED INTO SEPARATE LOOPS
      DO 10 I=2,N
10       A(I) = A(I-1) * C(I)
      DO 20 I=2,N
         B(I) = A(I) * C(I-1)**2
20       C(I-1) = SQRT(A(I)) - B(I)
```

Scalar loop because of recurrence on `A`**.**

Separate loops: one scalar and one vector.

The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000006	0.000004	1.5172
50	0.000072	0.000008	8.5264
65	0.000054	0.000013	4.1325
500	0.000360	0.000075	4.8271
1000	0.000724	0.000151	4.8031

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000003	1.3047
50	0.000047	0.000005	9.1813
65	0.000040	0.000008	5.0400
500	0.000282	0.000038	7.5121
1000	0.000564	0.000074	7.5750

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000029	0.000017	1.7411
50	0.000350	0.000043	8.0664
65	0.000246	0.000064	3.8385
500	0.001605	0.000403	3.9864
1000	0.003220	0.000798	4.0377

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000004	0.7512
50	0.000020	0.000010	2.1350
65	0.000027	0.000016	1.7221
500	0.000198	0.000071	2.8028
1000	0.000394	0.000140	2.8230

Segmenting a Non–vectorizing Loop *(in C)*

```
#include <math.h>
float a[1000], b[1000], c[1000];
float original(n)
int n;
{
int i;
    for (i=1;i<n;i++) {
        a[i] = a[i-1] * c[i];
        b[i] = a[i] * c[i-1] * c[i-1];
        c[i-1] = sqrt(a[i]) - b[i];
     }
}
/***********************************/
float modified(n)
int n;
{
int i;
    for (i=1;i<n;i++)
        a[i] = a[i-1] * c[i];
    for (i=1;i<n;i++)
        b[i] = a[i] * c[i-1] * c[i-1];
        c[i-1] = sqrt(a[i]) - b[i];
}
```

Scalar loop because of recurrence on `a`**.**

Separate loops: one scalar and one vector.

Speed–up: 0, 2, 4, 6, 8

Iterations: 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000

CRAY Y–MP C90
CRAY Y–MP
CRAY Y–MP EL
CRAY–2

Jamming Vectorizing Loops
6.6

Programmers often encounter two or more independent, vectorizing loops with the same iteration count. By combining two of these loops into one vector loop, the overhead of loop control code for the loop can be eliminated. Combining three or more loops eliminates even more overhead. The process of combining vectorizing loops is called loop jamming.

Fortran and C versions of this technique follow, along with comparative timing comparisons.

Jamming Vectorizing Loops

Adjacent looping constructs can be jammed together to:

promote better loop optimizations
decrease loop overhead

```
DO I=1,N
 vector loop
END DO
DO I=1,N
 vector loop
END DO
```

```
for (i=0;i&lt;N;i++) {
 vector loop
}
for (i=0;i&lt;N;i++) {
 vector loop
}
```

```
DO I=1,N
 combined vector
 loop
END DO
```

```
for (i=0;i&lt;N;i++) {
 combined vector
 loop
}
```

The following output was generated from the Fortran program shown on the facing page.

CRAYY-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000008	0.000003	2.8821
64	0.000020	0.000007	2.8162
65	0.000027	0.000009	2.9271
500	0.000151	0.000051	2.9550
1000	0.000300	0.000101	2.9662

CRAYYMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000005	0.000002	2.9830
64	0.000014	0.000005	2.9515
65	0.000020	0.000007	2.9953
500	0.000015	0.000005	3.1394
1000	0.000031	0.000010	3.1673

CRAYYMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000033	0.000011	3.1176
64	0.000095	0.000034	2.7951
65	0.000124	0.000043	2.8829
500	0.000722	0.000254	2.8402
1000	0.001442	0.000506	2.8485

CRAY-2 timings

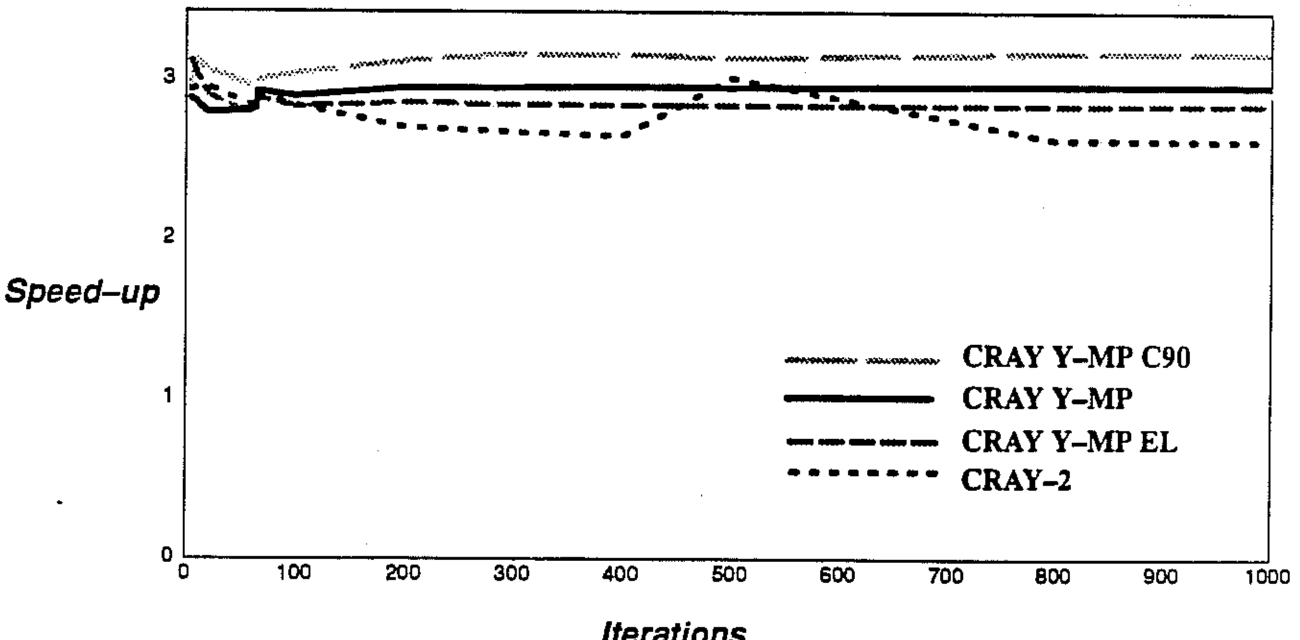
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000002	2.9359
64	0.000014	0.000005	2.8309
65	0.000018	0.000006	2.9046
500	0.000081	0.000027	3.0147
1000	0.000133	0.000051	2.6245

Jamming Vectorized Loops (*in Fortran*)

```
      REAL FUNCTION ORIGINAL(N)
      REAL    A(1000), B(1000), C(1000), D(1000) E(1000), F(1000)
       DO 10 I=1,N
10        A(I) = E(I) + SQRT(F(I))
       DO 20 I=1,N
20        B(I) = E(I) * SQRT(F(I))
       DO 30 I=1,N
30        C(I) = E(I) - SQRT(F(I))
       DO 40 I=1,N
40        D(I) = E(I) / SQRT(F(I))
       END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
       REAL FUNCTION MODIFIED(N)
       REAL    A(1000), B(1000), C(1000), D(1000), E(1000), F(1000)
        DO 10 I=1,N
           A(I) = E(I) + SQRT(F(I))
           B(I) = E(I) * SQRT(F(I))
           C(I) = E(I) - SQRT(F(I))
           D(I) = E(I) / SQRT(F(I))
10      CONTINUE
        END
```

Separate vectorizing loops.

Loops jammed into single vectorizing loop.



The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000009	0.000003	2.9611
64	0.000020	0.000007	2.7693
65	0.000027	0.000009	2.9022
500	0.000151	0.000051	2.9692
1000	0.000300	0.000101	2.9819

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000001	3.0394
64	0.000018	0.000006	3.0347
65	0.000023	0.000008	2.9816
500	0.000146	0.000046	3.1906
1000	0.000285	0.000089	3.2107

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000036	0.000011	3.3857
64	0.000095	0.000035	2.7613
65	0.000124	0.000042	2.9117
500	0.000720	0.000248	2.9001
1000	0.001439	0.000493	2.9203

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000002	3.0220
64	0.000015	0.000005	2.7190
65	0.000019	0.000007	2.8323
500	0.000078	0.000029	2.6919
1000	0.000140	0.000057	2.4717

Jamming Vectorized Loops (*in C*)

```
#include <math.h>
float a[1000], b[1000], c[1000], d[1000], e[1000], f[1000];
float original(n)
int n;{
int i;
    for (i=0;i<n;i++)
        a[i] = e[i] + sqrt(f[i]);
    for (i=0;i<n;i++)
        b[i] = e[i] - sqrt(f[i]);
    for(i=0;i<n;i++)
        c[i] = e[i] * sqrt(f[i]);
    for(i=0;i<n;i++)
        d[i] = e[i] / sqrt(f[i]);
}
/*****************************************/
float modified(n)
int n;{
int i;
    for (i=0;i<n;i++) {
        a[i] = e[i] + sqrt(f[i]);
        b[i] = e[i] - sqrt(f[i]);
        c[i] = e[i] * sqrt9f[i]);
        d[i] = e[i] / sqrt9f[i]);
    }
}
```

Separate vectorizing loops.

Loops jammed into single vectorizing loop.

Speed-up

3

2

1

0

0 100 200 300 400 500 600 700 800 900 1000

Iterations

CRAY Y-MP C90

CRAY Y-MP

CRAY Y-MP EL

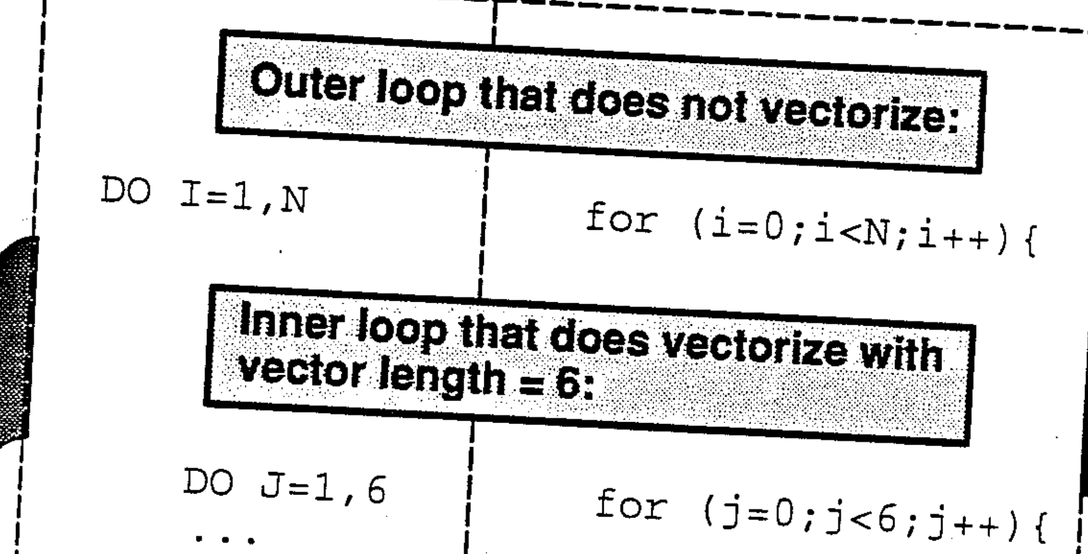
CRAY-2

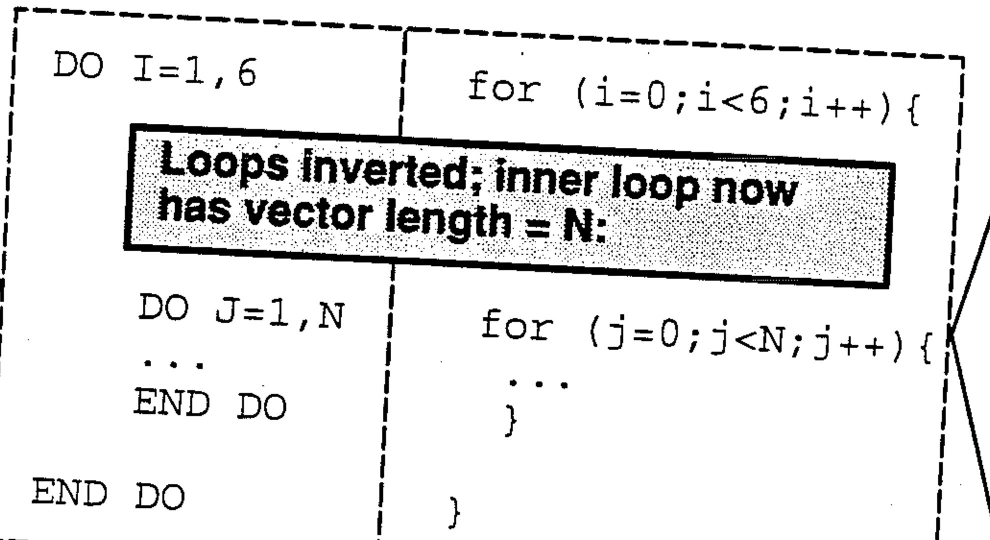
Loop Inversion to Force Maximum Work into Innermost Loop
6.7

If the outer loop of a nested loop structure has a larger iteration count than the inner, vectorizing loop, performance will be improved by inverting the loops (that is, putting the outer loop inside of the inner loop). This will increase the vector length of the computation and therefore increase the speed of execution. However, inverting loops changes the order in which operations take place. Therefore, this technique should not be used if loop inversion changes the results of the loop, causes a dependency conflict to emerge, or introduces memory bank conflicts.

Fortran and C versions of this technique follow, along with comparative timing comparisons.

Inverting Loops to Maximize Vector Length





The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000001	0.000006	0.2576
64	0.000016	0.000006	2.6282
65	0.000016	0.000009	1.8148
500	0.000123	0.000039	3.1503
1000	0.000244	0.000076	3.2152

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000001	0.000004	0.3235
64	0.000011	0.000004	2.9740
65	0.000011	0.000004	3.0226
500	0.000084	0.000016	5.1328
1000	0.000167	0.000030	5.5414

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000024	0.2778
64	0.000075	0.000031	2.3923
65	0.000076	0.000047	1.5920
500	0.000573	0.000223	2.5654
1000	0.001145	0.000437	2.6164

CRAY-2 timings

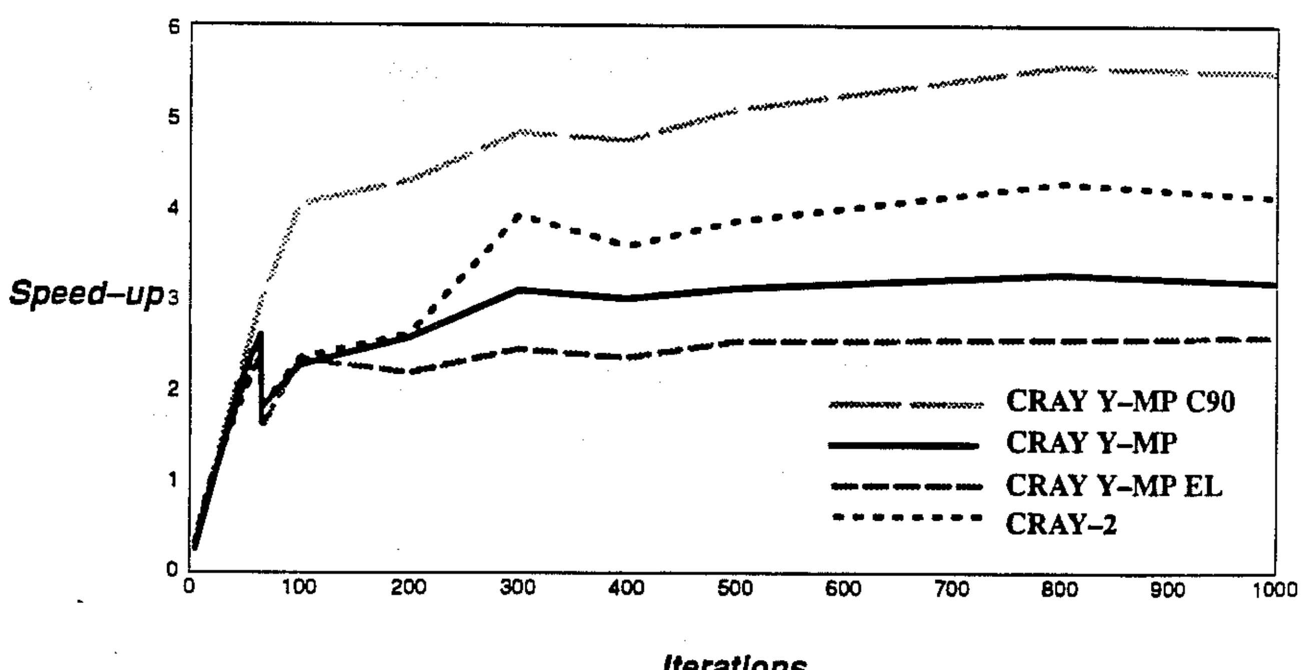
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000012	0.3310
64	0.000046	0.000019	2.3914
65	0.000046	0.000025	1.8179
500	0.000369	0.000095	3.8943
1000	0.000701	0.000168	4.1660

Loop Inversion to Force Maximum Work into Inner Loop (*in Fortran*)

```
      REAL FUNCTION ORIGINAL(N)
      REAL    A(1001,10), B(1001,10), C(1001,10)
       DO 10 I=1,N
          DO 10 J=1,10
10           A(I,J) = B(I,J) * C(I,J)
       END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION MODIFIED(N)
      REAL    A(1001,10), B(1001,10), C(1001,10)
       DO 10 J=1,10
          DO 10 I=1,N
10           A(I,J) = B(I,J) * C(I,J)
       END
```

Short vector loop.

Loops switched; longer VL on inner loop.



The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000002	0.000007	0.2603
64	0.000020	0.000007	2.7924
65	0.000020	0.000010	1.9907
500	0.000156	0.000041	3.8177
1000	0.000312	0.000077	4.0629

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000001	0.000005	0.2653
64	0.000014	0.000005	3.1034
65	0.000014	0.000005	3.1659
500	0.000109	0.000016	6.6834
1000	0.000217	0.000030	7.1759

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000031	0.2305
64	0.000082	0.000033	2.4523
65	0.000084	0.000046	1.8040
500	0.000632	0.000221	2.8632
1000	0.001263	0.000428	2.9489

CRAY-2 timings

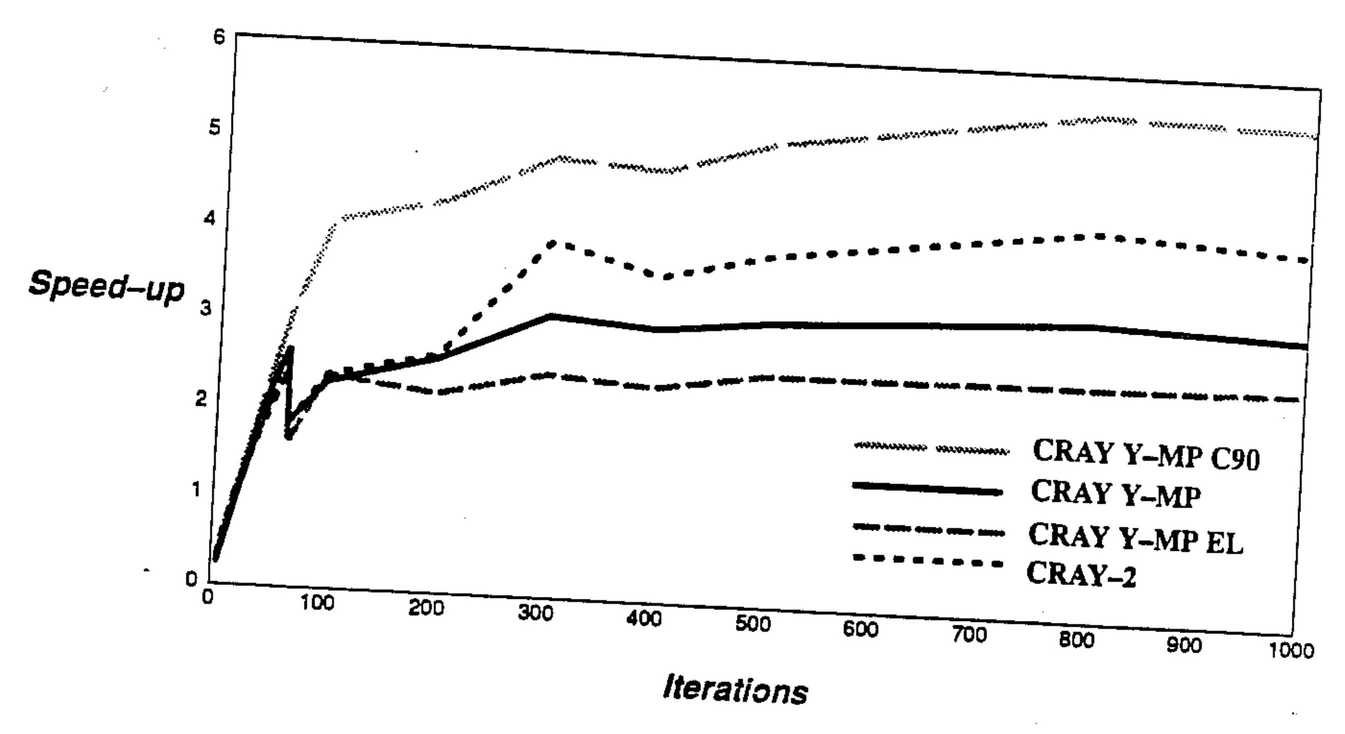
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000014	0.2674
64	0.000046	0.000022	2.0965
65	0.000047	0.000045	1.0436
500	0.000360	0.000099	3.6438
1000	0.000719	0.000175	4.1114

Loop Inversion to Force Maximum Work into Inner Loop (*in C*)

```
float a[10][1001], b[10][1001], c[10][1001];
float original(n)
int n;
{
int i, j;
/* SHORTER LOOP INSIDE */
   for (j=0;j<n;j++)
      for (i=0;i<10;i++)
         a[i][j] = b[i][j] * c[i][j];
}
/*******************************************/
float modified(n)
int n;
{
int i, j;
/* LONGER LOOP INSIDE */
   for (i=0;i<10;i++)
      for (j=0;j<n;j++)
         a[i][j] = b[i][j] * c[i][j];
}
```

Short vector loop.

Loops switched; longer VL on inner loop.

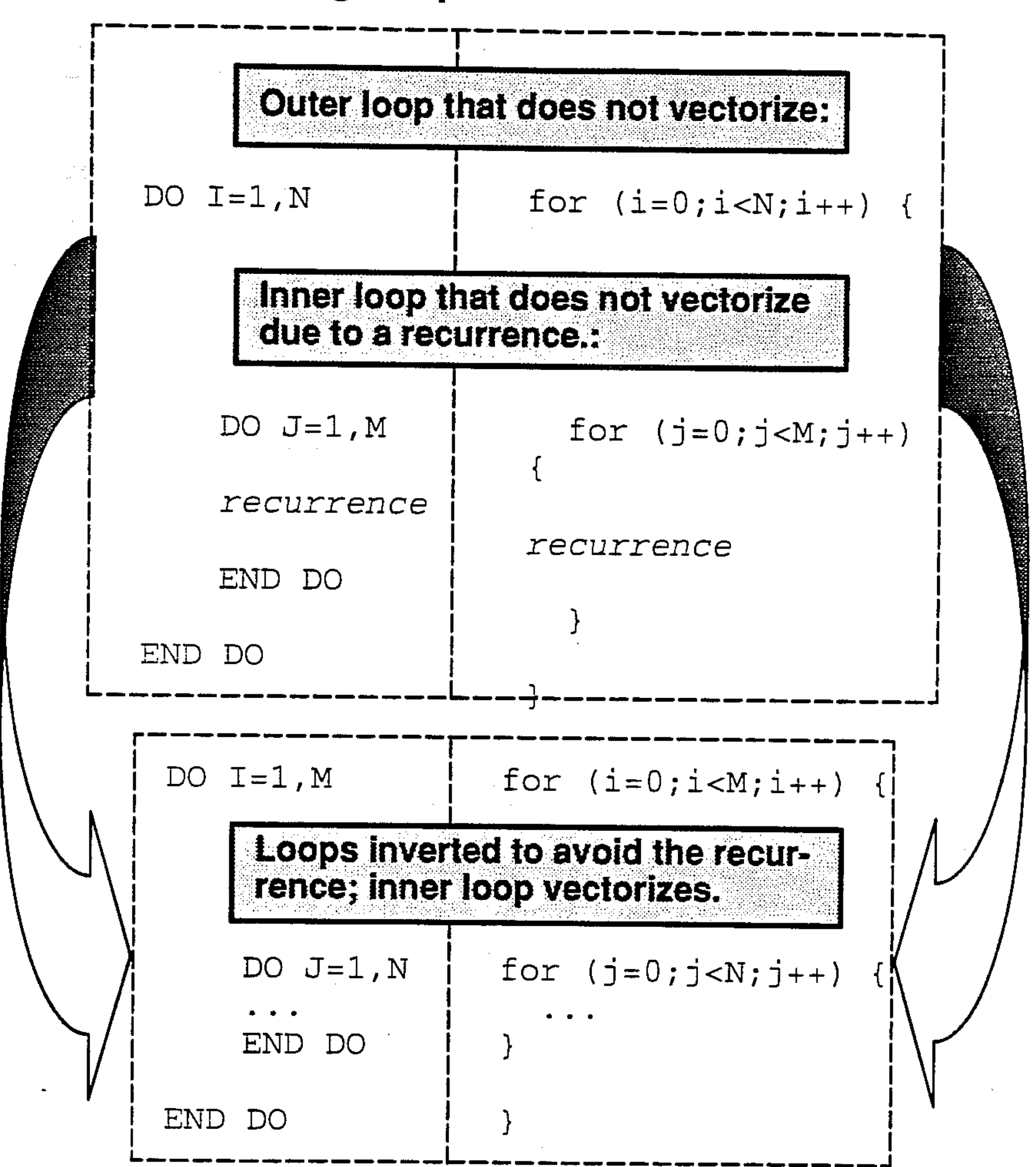


Loop Inversion to Avoid a Recurrence
6.8

Loop inversion (switching the order of nested loops) can also be used to eliminate dependency conflicts. In many cases in which there is a recurrence in one dimension of a multidimensional array, switching the order in which the array is processed (from row to column or from column to row) can eliminate the recurrence. This can also increase performance of operations that sum rows (or columns) of matrices by replacing vector reduction operations (one of the special–case vectorizing recurrences) across rows (or down columns) with vector sums down the columns (or across the rows). However, as previously mentioned, be aware of the side effects that occur when the order in which operations execute is changed.

Fortran and C versions of this technique follow, along with comparative timing comparisons. Other related techniques for avoiding a recurrence are shown in the timing comparisons of appendix A.

Inverting Loops to Avoid a Recurrence



The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000002	1.8549
64	0.000320	0.000043	7.5266
65	0.000336	0.000043	7.7735
500	0.013216	0.002108	6.2696
1000	0.052934	0.008307	6.3720

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000001	1.9122
64	0.000187	0.000022	8.4677
65	0.000198	0.000022	8.9621
500	0.007132	0.000858	8.3158
1000	0.028526	0.003338	8.5457

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000016	0.000007	2.2367
64	0.001655	0.000201	8.2155
65	0.001733	0.000209	8.3106
500	0.071543	0.010280	6.9597
1000	0.286755	0.040567	7.0686

CRAY-2 timings

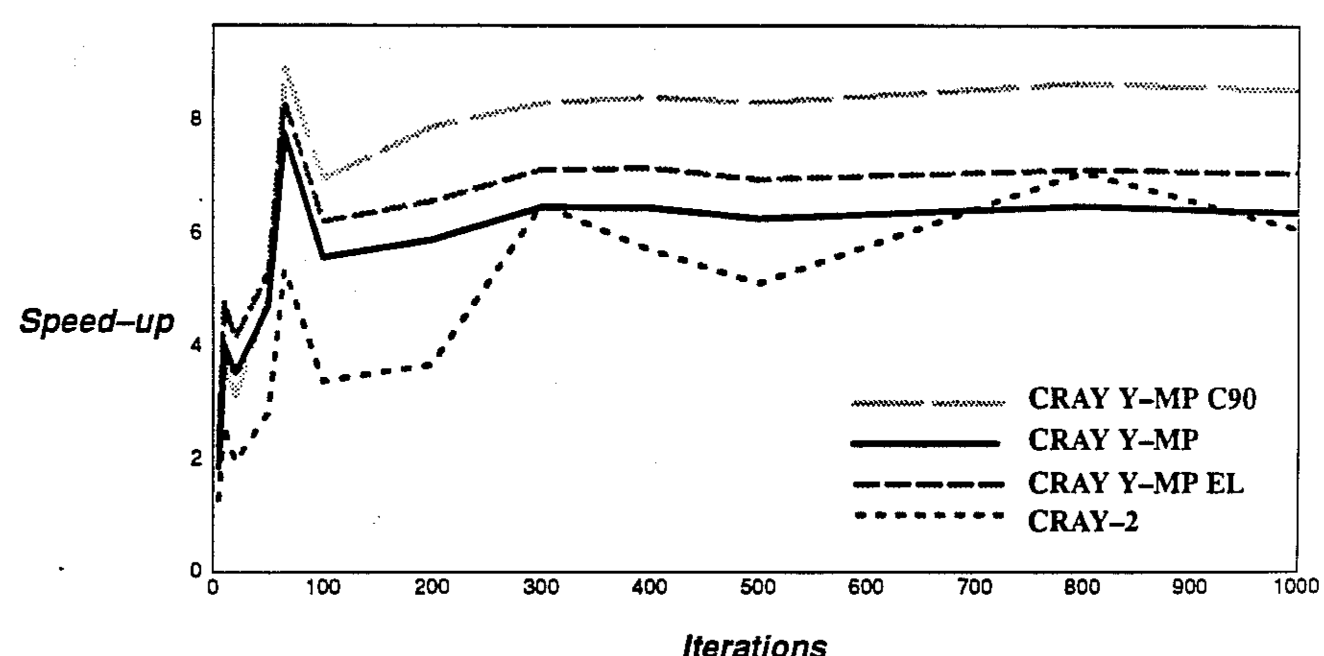
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000005	1.2865
64	0.000511	0.000097	5.2498
65	0.000538	0.000101	5.3521
500	0.018285	0.003574	5.1162
1000	0.073192	0.012038	6.0798

Loop Inversion to Avoid a Recurrence (*in Fortran*)

```
      REAL FUNCTION ORIGINAL(N)
       REAL    A(500,500)
* RECURRENCE IN INNER LOOP
       DO 10 I=2,N
         DO 10 J=2,N
10         A(I,J) = A(I,J-1) * 2.0
       END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION MODIFIED(N)
       REAL    A(500,500)
* LOOPS SWITCHED TO ELIMINATE RECURRENCE
       DO 10 J=2,N
         DO 10 I=2,N
10         A(I,J) = A(I,J-1) * 2.0
       END
```

Recurrence on inner loop, no vectorization.

Loops inverted to eliminate the recurrence.



The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000005	0.000003	1.9187
64	0.000383	0.000045	8.5637
65	0.000403	0.000045	8.8650
500	0.013522	0.002140	6.3184
1000	0.054157	0.008298	6.5264

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000002	2.0024
64	0.000237	0.000024	9.9059
65	0.000250	0.000024	10.4276
500	0.007417	0.000836	8.8685
1000	0.029673	0.003292	9.0128

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000021	0.000013	1.6330
64	0.001878	0.000305	6.1571
65	0.001973	0.000311	6.3369
500	0.072524	0.011238	6.4537
1000	0.290424	0.042560	6.8238

CRAY-2 timings

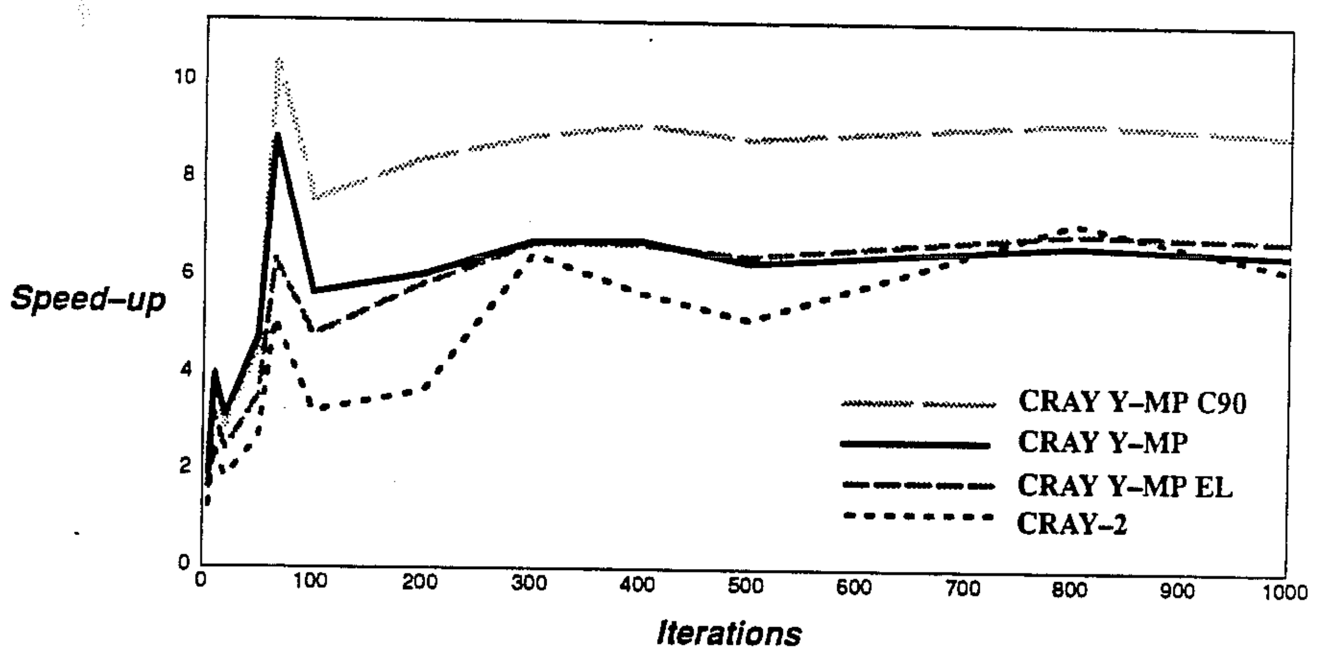
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000006	1.2663
64	0.000543	0.000110	4.9231
65	0.000572	0.000112	5.1035
500	0.018846	0.003667	5.1394
1000	0.075422	0.012113	6.2265

Loop Inversion to Avoid a Recurrence (*in C*)

```
float a[500][500];
float original(n)
int n;
{
int i, j;
     for (j=1;j<n;j++)
          for (i=1;i<n;i++)
               a[i][j] = a[i-1][j] * 2.0;
}
/*******************************************/
float modified(n)
int n;
{
int i, j;
     for (i=1;i<n;i++)
          for (j=1;j<n;j++)
               a[i][j] = a[i-1][j] * 2.0;
}
```

Recurrence on inner loop; no vectorization.

Loops inverted to eliminate the recurrence.



Linearizing Loops
6.9

In many cases, it is necessary to perform operations on all elements of multidimensional matrices. Such operations are frequently thought of in terms of nested loops. But because these matrices are stored contiguously in memory, they may be referenced as a linear space in which one of the indices is intentionally "over indexed". The net result of this is to eliminate extra loop overhead, reduce the number of short vectors, and, in some cases, reduce the complexity of the offset calculations necessary for indexing the arrays. It should be noted that this technique may introduce portability problems.

In some cases, the Fortran preprocessor is capable of performing this operation on behalf of the programmer, but it is not capable of recognizing all such cases.

Examples of this technique are on the next page. Note that the index that will be incremented is the first index for Fortran and the last index for C.

Outer loop that does not vectorize:

Inner loop that does vectorize with vector length = N:

```
DO I=1,N
    DO J=1,N

    END DO
END DO
```

```
for (i=0;i<N;i++) {
  for (j=0;j<N;j++)
{

  }
}
```

Linearized loop with vector length = N*N.

```
DO I=1,N*N

 linearized
  operations
END DO
```

```
for (i=0;i<N*N;i++)
{

 linearized
 operations
}
```

The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000001	3.2857
64	0.000136	0.000043	3.1631
65	0.000213	0.000044	4.8895
500	0.011761	0.002566	4.5831
1000	0.046914	0.010226	4.5876

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000002	0.000001	2.3514
64	0.000046	0.000011	4.2218
65	0.000047	0.000012	3.9936
500	0.003313	0.000627	5.2871
1000	0.013147	0.002476	5.3101

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000013	0.000004	3.2574
64	0.000295	0.000171	1.7278
65	0.000498	0.000178	2.7978
500	0.024380	0.010295	2.3682
1000	0.097172	0.040833	2.3797

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000002	3.4097
64	0.000372	0.000062	5.9727
65	0.000416	0.000063	6.6023
500	0.021791	0.003671	5.9357
1000	0.085770	0.014573	5.8855

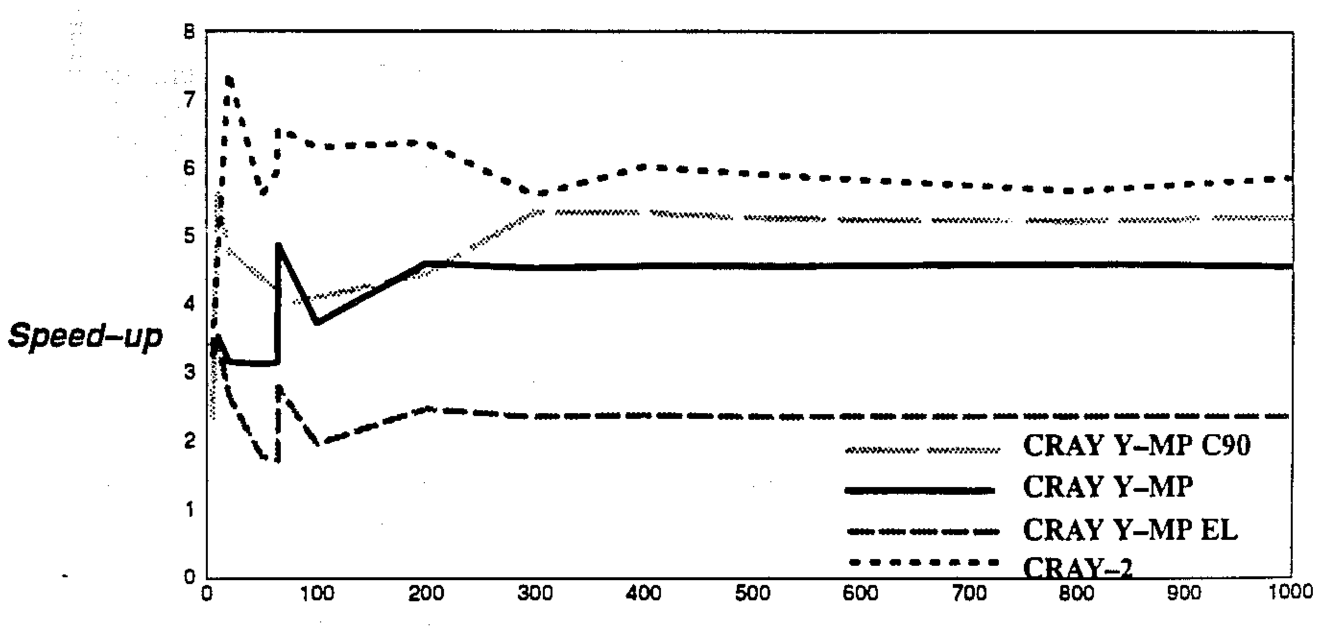
Linearizing Loops (*in Fortran*)

```
 REAL FUNCTION ORIGINAL(N)
REAL A(1000,1000),B(1000,1000),C(1000,1000)
 COMMON /BLK1/ A,B,C

  DO 10 J=1,N
     DO 10 I=1,N
10       A(I,J) = B(I,J) + C(I,J)
  END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Nested loops with inner VL = N.

```
 REAL FUNCTION MODIFIED(N)
REAL A2(1000000,1),B2(1000000,1),C2(1000000,1)
 COMMON /BLK2/ A2,B2,C2

  DO 10 I=1,N*N
10     A2(I,1) = B2(I,1) + C2(I,1)
  END
```

Linearized vector loop with VL = N*N.



The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000001	3.8491
64	0.000047	0.000043	1.0981
65	0.000071	0.000043	1.6454
500	0.002633	0.002557	1.0298
1000	0.010379	0.010243	1.0134

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000001	2.7824
64	0.000029	0.000011	2.6182
65	0.000030	0.000011	2.6146
500	0.000771	0.000616	1.2524
1000	0.002773	0.002460	1.1274

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000016	0.000005	3.3171
64	0.000202	0.000160	1.2622
65	0.000299	0.000173	1.7225
500	0.010096	0.009741	1.0365
1000	0.040811	0.039746	1.0268

CRAY-2 timings

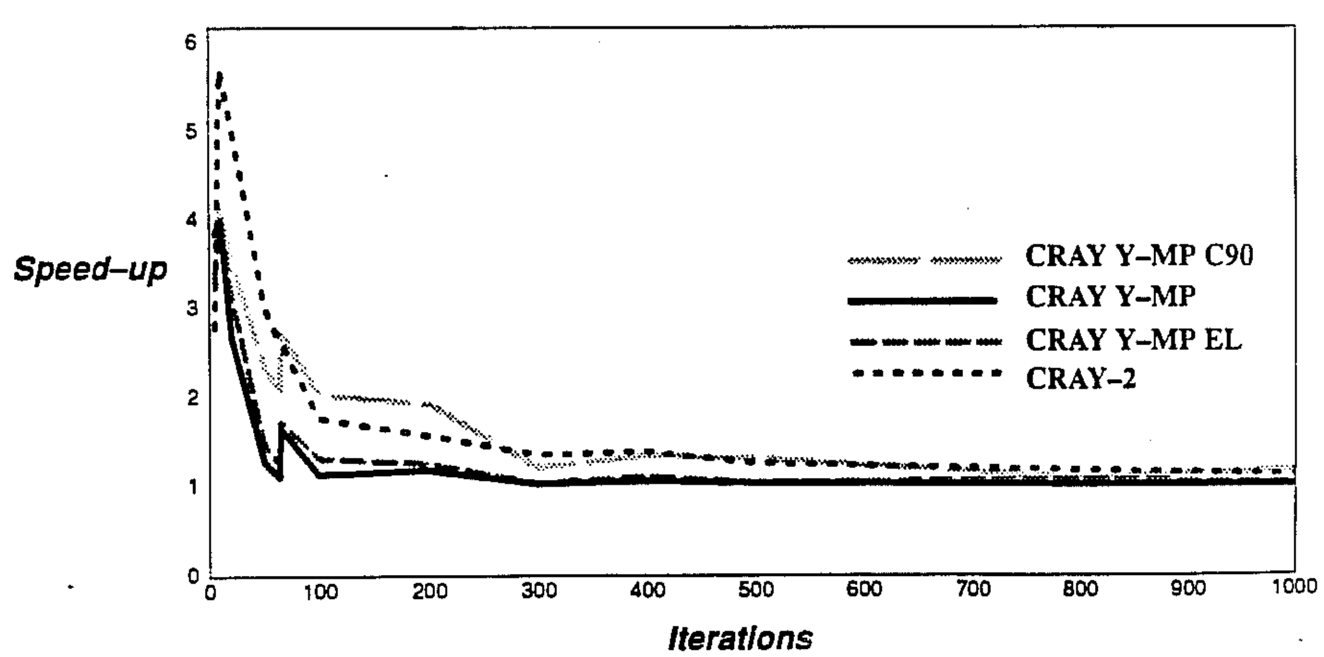
ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000007	0.000002	3.7986
64	0.000129	0.000062	2.0837
65	0.000173	0.000064	2.7278
500	0.004808	0.003668	1.3107
1000	0.017050	0.014524	1.1739

Linearizing Loops (*in C*)

```
float a[1000][1000], b[1000][1000], c[1000][1000];
float a2[1][1000000],b2[1][1000000],c2[1][1000000];
float original(n)
int n;
{
int i, j;
     for (i=0;i<n;i++)
          for (j=0;j<n;j++)
               a[i][j] = b[i][j] + c[i][j];
}
/*****************************************/
float modified(n)
int n;
{
int i, j;
     for (i=0;i<n*n;i++)
          a[0][i] = b[0][i] + c[0][i];
}
```

Nested loops with inner VL = `n`**.**

Linearized vector loop with VL = `n*n`**.**



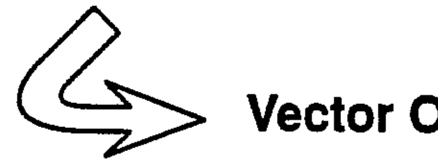
Converting Scalar Recurrences to Vector Temporaries
6.10

One way to eliminate a scalar recurrence in a loop is to store the computed scalar values in a temporary array and index them in a different order. This technique is based on the same concept as inverting loops to avoid a recurrence. However, because scalars do not have another dimension with which to work, the scalar must be transformed into an array.

This technique has the obvious drawback of using more memory because it saves all intermediate results. Moreover, this same technique can be applied to the conversion of one–dimensional arrays to two dimensions for recurrence elimination. However, in this case, the increased memory usage can be dramatic.

Fortran and C versions of this technique follow, along with comparative timing comparisons.

Scalar Recurrence



Vector Operation

```
DO I=1,N              for (i=0;i<N;i++) {

X = X + ...           x = x + ...;

  = X                   = x;

END DO                }
```

```
...one of the more difficult
optimization techniques.
```

The following output was generated from the Fortran program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000091	0.000017	5.3935
64	0.000940	0.000204	4.6125
65	0.000955	0.000207	4.6131
500	0.007218	0.001585	4.5539
1000	0.014418	0.003169	4.5495

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000069	0.000007	9.3303
64	0.000378	0.000089	4.2375
65	0.000280	0.000091	3.0866
500	0.002046	0.000694	2.9477
1000	0.004079	0.001388	2.9389

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000469	0.000083	5.6551
64	0.003127	0.001004	3.1128
65	0.002508	0.001020	2.4580
500	0.018883	0.007805	2.4192
1000	0.037840	0.015656	2.4170

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000189	0.000025	7.6781
64	0.001056	0.000295	3.5777
65	0.000772	0.000301	2.5667
500	0.005667	0.002298	2.4663
1000	0.011286	0.004629	2.4383

Converting Scalar Recurrence to Vector Temporary (*in Fortran*)

```
      REAL FUNCTION ORIGINAL(N)
       REAL     A(1000),C
       M = 100
       DO 10 J = 1,M
          S = BB
          DO 10 I = 1,M
             S = S * C
10         A(I) = A(I) + S
      END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION MODIFIED(N)
       REAL     A(1000),S(100)
       M = 100
       DO 5  I = 1,M
          S(I) = BB
5      CONTINUE
       DO 10 I = 1,N
          DO 10 J = 1,M
            S(J) = S(J) * C
10          A(I) = A(I) + S(J)
      END
```

Scalar recurrence.

Temporary vector; full vectorization.

The following output was generated from the C program shown on the facing page.

```
                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAY Y-MP                 5    0.000149    0.000017     8.9263
timings                  64    0.000760    0.000200     3.7980
                         65    0.000495    0.000203     2.4346
                        500    0.003616    0.001555     2.3251
                       1000    0.007213    0.003109     2.3201

                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAY YMP-C90              5    0.000106    0.000008    13.7141
timings                  64    0.000503    0.000093     5.4294
                         65    0.000309    0.000092     3.3470
                        500    0.002230    0.000707     3.1542
                       1000    0.004444    0.001413     3.1448

                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAY YMP-EL               5    0.000708    0.000089     7.9829
timings                  64    0.004203    0.001040     4.0413
                         65    0.003165    0.001057     2.9934
                        500    0.023374    0.008033     2.9098
                       1000    0.046640    0.016104     2.8962

                 ITERATIONS    ORIGINAL    MODIFIED    ORG/MOD
CRAY-2                    5    0.000268    0.000024    11.0714
timings                  64    0.001216    0.000296     4.1102
                         65    0.000670    0.000313     2.1423
                        500    0.004805    0.002304     2.0860
                       1000    0.009567    0.004619     2.0710
```

Converting Scalar Recurrence to Vector Temporary (*in C*)

```
float a[1000];
float original(n)
int n;
{
int i, j, m = 100;
float c = 1.00002, bb = 2.0, s;
     for (j=0;j<m;j++) {
          s = bb;
         for (i=0;i<n;i++) {
              s = s*c;
             a[i] = a[i] + s;
          }
      }
}
/***********************************************/
float modified(n)
int n;
{
int i, j, m = 100;
float s[100], c = 1.00002, bb = 2.0;
     for (i=0;i<m;i++)
          s[i] = bb;
     for (i=0;i<n;i++) {
         for (j=0;j<m;j++) {
             s[j] = s[j] * c;
             a[i] = a[i] + s[j];}}}
```

Scalar recurrence; no vectorization (original loop)

Temporary vector; full vectorization. (modified loop)

Speed-up
14
12
10
8
6
4
2
0
0 100 200 300 400 500 600 700 800 900 1000
Iterations
CRAY Y-MP C90
CRAY Y-MP
CRAY Y-MP EL
CRAY-2

Positioning Scalar Conditional Blocks
6.11

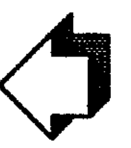
CF77 and CC execute the code within scalar conditional block structures in sequential order. In a scalar loop, if the first condition is true, then the other conditions can be ignored. If the condition most frequently true is known, placing that conditional block first can speed up the execution time of the code. The prof command can be useful in isolating the most frequently executed conditional block.

Fortran and C versions of this technique follow, along with comparative timing comparisons. In the Fortran example, note that most elements of array C have values less than the values of the elements in array B; in the C example, note that most elements of array c have values less than the values of the elements in array b.

Positioning Scalar Conditional Blocks

```
if (...)
    case that is executed
    1% of the time
else if (...)
    case that is executed
    2% of the time
else if (...)
    case that is executed
    2% of the time
else
    case that is executed
    95% of the time
```

95% of the time there are 3 if tests with a "false" result.



The most frequently executed case is tested first.

```
if (...)
    case that is executed
    95% of the time
else if (...)
    case that is executed
    2% of the time
else if (...)
    case that is executed
    2% of the time
else
    case that is executed
    1% of the time
```

The following output was generated from the Fortran program shown on the facing page.

```
                ITERATIONS     ORIGINAL     MODIFIED     ORG/MOD
                         5     0.000003     0.000002      1.3906
CRAY Y-MP               64     0.000032     0.000022      1.4149
timings                 65     0.000032     0.000023      1.4149
                       500     0.000247     0.000175      1.4122
                      1000     0.000493     0.000349      1.4132

                ITERATIONS     ORIGINAL     MODIFIED     ORG/MOD
                         5     0.000002     0.000001      1.5828
CRAY YMP-C90            64     0.000023     0.000016      1.3890
timings                 65     0.000023     0.000017      1.3965
                       500     0.000173     0.000126      1.3810
                      1000     0.000346     0.000251      1.3807

                ITERATIONS     ORIGINAL     MODIFIED     ORG/MOD
                         5     0.000012     0.000009      1.3693
CRAY YMP-EL             64     0.000137     0.000096      1.4279
timings                 65     0.000139     0.000097      1.4354
                       500     0.001058     0.000742      1.4256
                      1000     0.002112     0.001481      1.4258

                ITERATIONS     ORIGINAL     MODIFIED     ORG/MOD
                         5     0.000004     0.000003      1.2877
CRAY-2                  64     0.000047     0.000034      1.3601
timings                 65     0.000048     0.000035      1.3631
                       500     0.000364     0.000266      1.3691
                      1000     0.000728     0.000532      1.3680
```

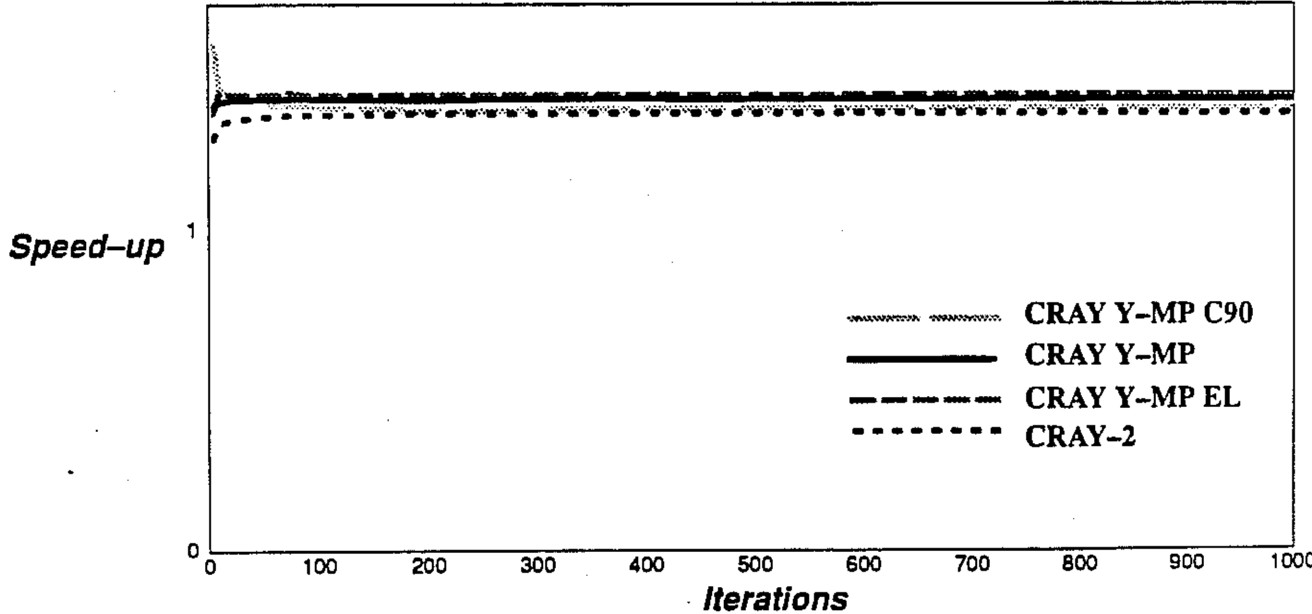
Positioning Scalar Conditional Blocks (*in Fortran*)

```
      REAL FUNCTION ORIGINAL(II)
        DO 5 I = 1,II
           TEMP = B(I)
          IF(C(I).EQ.TEMP) THEN
             D(I) = D(I-1)
          ELSEIF(C(I).GT.TEMP) THEN
             D(I) = C(I) - B(I)
          ELSEIF(C(I).LT.TEMP) THEN
             D(I) = B(I) + C(I)
            ENDIF
5       CONTINUE
        END
```

Most frequently executed code last.

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       REAL FUNCTION MODIFIED(II)
         DO 5 I = 1,II
            TEMP = B(I)
           IF(C(I).LT.TEMP) THEN
              D(I) = B(I) + C(I)
           ELSEIF(C(I).GT.TEMP) THEN
              D(I) = C(I) - B(I)
           ELSEIF(C(I).EQ.TEMP) THEN
              D(I) = D(I-1)
             ENDIF
5        CONTINUE
         END
```

Most frequently executed code first



The following output was generated from the C program shown on the facing page.

CRAY Y-MP timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000003	0.000002	1.3503
64	0.000034	0.000025	1.3676
65	0.000035	0.000026	1.3677
500	0.000267	0.000195	1.3690
1000	0.000534	0.000390	1.3691

CRAY YMP-C90 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000002	0.000002	1.3108
64	0.000024	0.000018	1.3504
65	0.000024	0.000018	1.3503
500	0.000186	0.000138	1.3487
1000	0.000371	0.000275	1.3486

CRAY YMP-EL timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000012	0.000009	1.3067
64	0.000146	0.000109	1.3432
65	0.000157	0.000108	1.4610
500	0.001123	0.000834	1.3461
1000	0.002240	0.001662	1.3482

CRAY-2 timings

ITERATIONS	ORIGINAL	MODIFIED	ORG/MOD
5	0.000004	0.000003	1.3774
64	0.000050	0.000037	1.3438
65	0.000051	0.000038	1.3437
500	0.000388	0.000289	1.3409
1000	0.000775	0.000592	1.3094

Positioning Scalar Conditional Blocks (*in C*)

```
float a[100000], b[100000], c[100000] ,d[100000];

float original(n)
int n;
{
int i;
float temp;
  for (i=0;i<n;i++) {
       temp = b[i];
       if(c[i] == temp)
         d[i] = d[i-1];
       else if(c[i] > temp)
         d[i] = c[i] - b[i];
       else if(c[i] < temp)
         d[i] = b[i] + c[i];
  }
```

Most frequently executed code last.

```
float modified(n)
int n;
{
int i;
float temp;
  for (i=0;i<n;i++) {
       temp = b[i];
       if(c[i] < temp)
         d[i] = b[i] + c[i];
       else if(c[i] > temp)
         d[i] = c[i] - b[i];
       else if(c[i] == temp)
         d[i] = d[i-1];
  }
```

Most frequently executed code first