Special Reductions in PFC

Ervan Darnell

Supercomputer Software Newsletter #13

Department of Computer Science

P.O. Box 1892

Rice University

Houston, Texas 77251

October 16, 1986

Introduction

Many Fortran statements that perform a repeated operation on the same scalar in a loop can be reduced using special functions in Fortran 8x, e.g. SUM. This is profitable when the underlying hardware can compute SUM directly. It is safe when the multiplications and additions involved can be assumed to be associative. PFC reduces most statements of the form:

```
DO ...
     T=T+expr      into      T=T+SUM(expr)
ENDDO
 -or-
DO ...
      T=T*expr      into      T=T+PROD(expr)
ENDDO
```

These special reductions produce vector code for statements that could not otherwise be vectorized because of dependences. The above statement has true, anti, and output dependences which form a cycle. Techniques such as scalar expansion may remove some of these dependencies but are not sufficient to break the recurrence.

Dependence analysis is used to break the program into PI blocks so that possible reduction statements can be found. For instance:

```
DO I=1,upb
     T=T+A(I)
     S=S+A(I)
     T=T+B(I)
ENDDO
```

gets split into two PI blocks. One is:

```
     T=T+A(I)
     T=T+B(I)
```

and the other is:

```
     S=S+A(I)
```

Everything of relevance to SUM reduction is thereby collected into one place. If something that appears to be SUM reducible is used twice on each iteration of the loop it might not be reducible. Example:

```
DO I=1,upb
     T = T + B(I)
     S = S + T + 1
ENDDO
```

The simplest case is where the assignment to reduce is in a PI block by itself. Then it is possible to directly analyze the expression to see if reduction is possible. It is also possible to reduce some PI blocks of more than one statement (see below).

Single Statement Sum Reduction

The following is a prototype DO loop where reduction might occur. The `I` could be any loop control variable and `T` any scalar variable.

```
DO I=1,upb
      .
      .
      T=xT + y
      .
      .
ENDDO
```

T his can be reduced when:

1) T occurs nowhere else in the loop

2) x & y are expressions constant w.r.t. T

3) x is constant w.r.t. I

4) y is constant or is directly indexed, e.g. 6, C, A(I), or B(I)*A(I)

5) The assignment is executed on every iteration.

It will reduce to:

```
T=x**upb*T+SUM(y(1:upb)*x**(upb-SEQ(1,upb,1)))
```

where y(I) is the i'th value of the expression represented by y.

Condition (1) can be checked as a matter of dependence analysis and PI block construction. Conditions (2) & (3) can be checked by examining all parts of the expression for I & T. Condition (4) can be checked by some tests that represent a special case mechanism while checking for (2) & (3). Condition (5) can be checked simply by looking for a guard on the statement. This is easy since GOTOs and IFs have already been converted to statement guards via IF-Conversion.

When x=1, a common case, this becomes the far more managable assignment:

```
T=T+SUM(y(1:upb)).
```

If y is constant w.r.t. I and x is constant then the assignment can be algebraicly reduced and a SUM isn't necessary. Example,

```
DO i=1,upb
   T=T+2                 becomes     T=T+upb*2
ENDDO
```

-and-

```
DO i=1,upb
   T=2*T+y               becomes     T=2**upb*T+(2**upb-1)/(2-1)*y
ENDDO
```

If the assignment is guarded, i.e. not executed on every loop iteration, it is still possible to SUM reduce where x=1. The guard clause that would otherwise determine when to execute the statement becomes a mask clause. Example,

```
DO I=1,upb
     IF (A(I).GT.0)
         T=T+A(I)        becomes     T=T+SUM(A(1:upb),A(1:upb).GT.0)
     ELSE                            T=T+SUM(-A(1:upb),.NOT. A(1:upb).GT.0)
         T=T-A(I)
     ENDIF
ENDDO
```

It is not possible to have an algebraic reduction for a guarded statement. The intuitive reason is that for a guarded statement something is changing on every iteration and an algebraic reduction assumes that the same action occurs on each iteration.

When x $\neq$ 1 and the statement is guarded it is not possible to do a reduction. This is because the general SUM reduction formula used above depends on the value from the previous iteration being fed in to the next iteration. When the statement is guarded this isn't true.

When y is constant w.r.t. I:

```
        Guarded    Not Guarded
X=1   |   SUM       Algebraic
X<>1 |Not Possible    SUM
```

Single Statement PROD Reduction

```
DO I=1,upb
   .
   .
   T=x T
   .
```

.

```
ENDDO
```

This can be reduced when:

1) T occurs nowhere else in the loop

2) x is constant w.r.t. T

3) x is constant or is directly indexable

4) The assignment is executed on every iteration

it will reduce to:

```
T=T*PROD(x(1:upb))
```

When x is constant w.r.t. I it is algebraicly reducible. Example:

```
DO I=1,upb
   T=T*2                          becomes          T=T*2**upb
ENDDO
```

When the assignment is guarded the guard becomes the PROD mask clause. This always works.

Table for Single Statement Reduction

x	y	Guarded	Not Guarded
Constant	Constant w.r.t. I	Not Possible	Algebraic
1	anything	Simple SUM	Simple SUM
x<>1	anything	Not Possible	SUM
Constant w.r.t I	doesn't exist	PROD	Algebraic
Not Constant w.r.t I	doesn't exist	PROD	PROD

A Simple SUM is a SUM reduction where T is not multiplied by an exponential. Not Possible means that no reasonable reduction expression exists. Where case 1 intersects with case 2 or 3 the better reduction occurs.

Multiple Reductions

When there are several assignments and for each one x=1, i.e. they are of the form:

```
T=T+y
T=T+y'
.
.
.
```

then they can be independently reduced regardless of the guards on any particular assignment. Notice that this reassociates additions. Example:

```
DO I=1,upb
   T=T+A(I)         becomes              T=T+SUM(A(1:upb))
   T=T+B(I)                              T=T+SUM(A(1:upb))
ENDDO
```

When x $\neq$ 1 and the are no guards for any of the assignments then they can be forward substituted to produce a reducible statement. Example:

```
DO I=1,upb
   T=2*T+A(I)
   T=3*T+B(I)
ENDDO
```

becomes

```
DO I=1,upb
   T=6*T+3*A(I)+B(I)
ENDDO
```

becomes

```
T=6**upb*T+SUM((3*A(I)+B(I))*6**(upb-SEQ(1,upb,1)))
```

For PROD reductions it is always possible to reduce each one independently. To understand this, look at:

```
DO I=1,upb
   T=T*A(I)
   T=T*B(I)
ENDDO
```

It doesn't matter what A(I) is, multiplying T by B(I) has the same relative effect. Assuming associativity, it doesn't matter what order the multiplications are made in. Contrast this with:

```
DO I=1,upb
   T=2*T+A(I)
   T=3*T+B(I)
ENDDO
```

where the affect of T=3*T+B(I) is influenced by the value of A(I). Multiplications and additions are not inter-associative!

The other restrictions for the single statement case still apply. For instance,

```
DO I=1,upb
   S=S+B(I)+T
   T=T+A(I)+S
ENDDO
```

is not reducible because T occurs somewhere else in the loop besides the assignments it appears in.

Table for Multiple Statement Reductions

x	y	Guarded	Not Guarded
Constant	Constant w.r.t. I	Not Possible	Algebraic
1	anything	Simple SUM	Simple SUM
x<>1	anything	Not Possible	SUM
Constant w.r.t I	doesn't exist	PROD	Algebraic
Not Constant w.r.t I	doesn't exist	PROD	PROD

This is the same as the table for single statement reductions, but the reasons are different.

Summary

The combination of dependence analysis and pattern matching allows special case reduction functions to be recognized that wouldn't otherwise be caught by general purpose mechanisms. Possible future improvements include generalizing the indexing cases for the y part of assignment that allow reduction and recognizing other reduction functions besides SUM and PROD (currently MIN & MAX are recognized in limited circumstances).