Loop Fusion in PFC

Ervan Darnell
Supercomputer Software Newsletter #17
Department of Computer Science
P.O. Box 1892
Rice University
Houston, Texas 77251

October 6, 1987

1 Introduction

Loop fusion is an optimization that puts the bodies of two loops together into a single loop (Fig. 1). The rest of this section discusses the motivation for fusion. Section two discusses how to test for legal fusion, in general. Section three discusses how the general method of section two is implemented in PFC. Section four discusses some of the details and difficulties of the PFC implementation.

Loop fusion can lower memory traffic in either the scalar or vector case. This occurs because values that would have to be referenced in memory twice, once for each of two loops, only need be referenced once after loop fusion because the value can be kept in a register. In the case where a value is stored in the first loop and loaded in the second loop it might be possible to eliminate the memory reference completely (assuming it is dead afterward). The disadvantage of loop fusion relative to memory traffic is that register usage is extended and extra spill code might be needed after fusion that was not before.

Figure 1 Simple Fusion

DO I=...
    A(I)=I
ENDDO
DO I=...
    B(I)=I
ENDDO
	$\Rightarrow$	DO I=...
		    A(I)=I
		    B(I)=I
		ENDDO

This consideration is not of principal importance in the PFC environment because the source to source transformation that PFC makes does not clearly have any effect on register usage.

The primary motivation for fusion in PFC is the elimination of the extra loop overhead. In the case of a scalar or purely vector machine this may not be significant, but in a language that expresses asynchronous parallelism or task parallelism (e.g. `@DO` in VM/EPEX Fortran) the gain is significant because fewer tasks are being created (half as many, in the simple case).

Assuming the loops in question have been normalized (i.e. the lower bound and step are both one), different upper bounds must be handled. There are two ways to do this. The first is to split the larger loop into two loops, one of which has the same bounds as the smaller loop, then fuse. This still leaves two loops but the advantages of saving memory traffic are realized. If the maximum bound cannot be determined this can still be done by splitting both loops then fusing and looping up to the minimum of the two upper bounds (calculated at run time) followed by the two loops (for a total of three from the original two) both looping from the minimum to their respective upper bound (only one of which will execute, which would require an extra `IF` in Fortran).

The second approach is to completely fuse the loops but surround the code of the smaller loop with an `IF` to prevent its execution when the index is too large (Fig. 2). If the smaller upper bound cannot be determined the body of both loops must be properly surrounded. In the scalar or vector case this is likely a loss since the extra time to execute the `IF` is more than the gain from the memory traffic saved. However, in the parallel case, this is much less of a problem and likely offset by the gain of fewer task creations and less synchronization. It is necessary to make an evaluation of possible gain and loss since two loops with wildly different upper bounds would not profit from loop fusion.

Currently, PFC does not fuse loops with different upper bounds. When it does the second method will be used.

Figure 2 Mismatched bounds fusion

DO I=1,10
    A(I)=I
ENDDO
DO I=1,12
    B(I)=I
ENDDO
	$\Rightarrow$	DO I=1,12
		    IF (I.LE.10)
		        A(I)=I
		    ENDIF
		    B(I)=I
		ENDDO

```
DO I=1,10
        A(I)=...
ENDDO
DO I=1,12
        ...=A(I-1)
ENDDO
```

Figure 3 Fusable with forward dependences

δ

```
DO I=1,10
        A(I)=...
ENDDO
DO I=1,12
        ...=A(I+1)
ENDDO
```

Figure 4 Not fusable because of backward dependences

$\bar{\delta}$

2 Fusion Dependency Test

Fusion can occur between two loops whenever there is no backward dependence between them relative to the level (of the loop body, not header) at which fusion is being considered. In other words, if, after fusing the loops, there is a backward dependence that did not previously exist then the loops cannot be fused. Any dependences at other levels will still exist after fusion and fusion will create no dependences at other levels. Some loop independent forward dependences may become loop carried forward dependences, but this will not change the semantics of the code (Fig. 3) nor prevent fusion.

Prior to fusion all data clearly flows from the first loop to the second loop (ignoring loop carried dependences induced by outer loops which are not relevant). Therefore, if the fused loops have a backward dependence, then data flows backward which would be incorrect. No backward dependences is a necessary condition for fusion. To see that it is a sufficient condition is a little more difficult because forward dependences might become loop dependent. However, this does not affect the number of forward edges nor the precise data flow [Wol82]. [1] The loops in Fig. 4 cannot be fused because the loop independent forward edge from `A(I)` to `A(I+1)` would become a loop carried backward edge. Before being fused, `A(2)` is set then used, but after fusion, `A(2)` is used then set. This would change the semantics of the code.

[1] Michael Wolfe. *Optimizing Supercompilers for Supercomputers* . PhD thesis, University of Illinois at Urbana-Champaign, October 1982. Chap. 4.

3 Dependences in PFC

So that the proper fusion decision can be made later (in code generation) dependence analysis must be extended to build dependences at the fusion levels (fusion edges). These dependences are what would exist if the loops were fused. However, to simply enter these dependences in the dependence graph would be wrong since that would make them appear as if the dependency existed before fusion. To handle this, dependences at fusion levels are always entered as loop independent forward dependences. This makes them appear redundant with other dependence edges already detected (Fig. 5).

An extra field, the fusion level, is then set to indicate the actual level of the dependency. The fact that the fusion level of any given dependency edge is not loop independent means the edge is a fusion edge.

If the fusion edge is already forward, the type (e.g. true or anti) of the entered edge is the same. However, if the fusion edge is backward then the type of the entered forward edge is the opposite. True dependences must be entered as anti-dependences and vice versa. This is necessary because the forward loop independent edge which is created must be correct, and reversing the direction of an edge reverses its type.

For example, if there is a backward true fusion edge, it goes from a definition back to a prior use and is loop carried (all fusion edges are). The loop independent edge which is entered in the dependence graph is forward and therefore runs from a use to a definition. It must be an anti-dependence in this case. The original fusion edge represents dependence if the loops were fused and actual loop independent edge which is created represents dependence when the loops are not fused (as is the case at dependence analysis time).

Output dependences are still output dependences. To indicate that the direction and the type have changed the sense field in the dependence graph is negated (Fig. 6).

4 Prior status of fusion in PFC

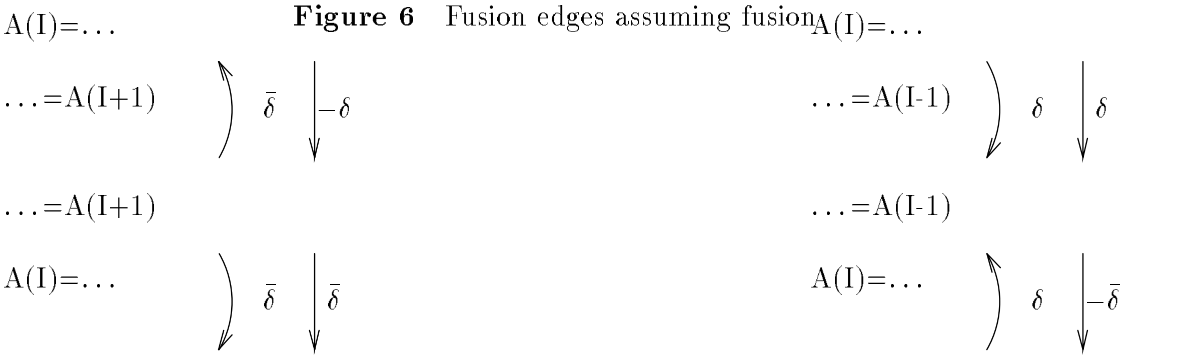
Prior versions of PFC made no attempt to find fusion edges; it was therefore necessary to *prevent* fusion from occurring. This was done by using the same upper bound variable for any loop at the same level (PFC assigns the upper bound, whatever it is, to a special variable). Statements in a loop then depend on the upper bound variable. The upper bound variable for a second loop has not only true dependences to statements in its loop, but also has anti-dependences to statements in the previous loop (Fig. 7). This prevents fusion.

In the figure, `A(I)` depends upon `IUPB1 = N` and `B(I)` depends upon `IUPB1 = M`. However, given that the upper bound variables have the same name, `B(I) = I` depends upon `IUPB1 = N` (since `B` implicitly references `IUPB1`) and `IUPB1 = M` has an anti-dependence on `A` (since `A` implicitly references `IUPB1`). This anti-depedence prevents fusion because `B(I)=I` must follow `A(I)=I` with an intervening serial statement.

Figure 5 Conversion of fusion edges to loop independent dependences

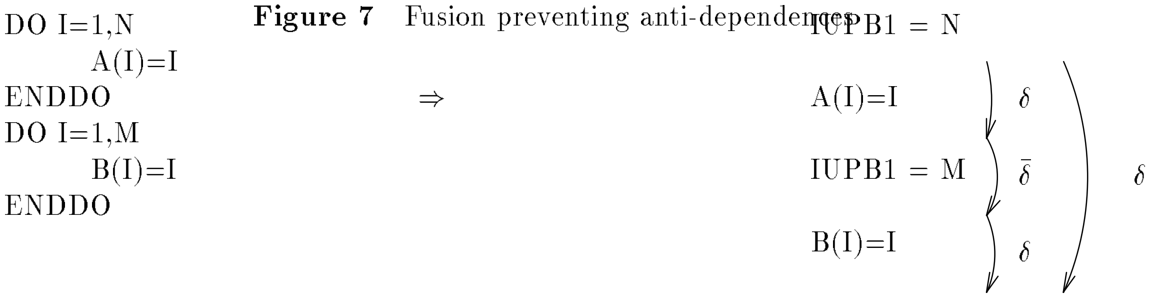
```
Let F = the fusion edge
Let R = the real edge entered in the dependence graph
R.fusion_level = F.level
R.level = loop independent

If F is backward
      R.sense = - F.sense
      case F.type
            output : R.type = output
            true : R.type = anti
            anti : R.type = true
      end
else
      R.sense = F.sense
endif
```

Figure 6 Fusion edges assuming fusion



Arcs represent the desired fusion edges. Lines represent the dependency edges as entered in the dependency graph. A dash indicates negated sense.

Figure 7 Fusion preventing anti-dependences



It is necessary to change the way upper bound variables are named so that they are all uniquely named and the fusion preventing anti-dependence does not exist. In the figure `IUPB1 = M` would become `IUPB2 = M`, thereby eliminating the anti-dependence since `A` still has an implicit reference to `IUPB1` and `B` has an implicit reference to `IUPB2`. This creates several implementation problems: upper bound variable name generation, location of upper bound variable names, and altering of levels on loops shifting.

The first is that, previously, upper bound variable names were all generated in PLMAIN once and referenced as needed. That worked because the maximum nesting depth was known, but the maximum number of `DO` loops is not known. It is now necessary to generate upper bound variables on demand, as `DO` loops are found or constructed (from `GOTO`'s). This is not a major problem.

The second problem is that the upper bound variable associated with a statement was found by indexing the upper bound table with the level of the statement. Since there was one upper bound variable per level this was simple enough. With the new upper bound variable names this global name finding scheme will no longer work. It is necessary to proceed from the statement itself to the proper upper bound variable. To handle this, DDSTMT now contains a pointer for each statement to the assignment statement of its most closely nested controlling upper bound. That statement contains a pointer to its controlling upper bound, etc. So, from any statement there is a chain of upper bound assignments, and the upper bound for any particular level can be found in that chain.

The third, and hardest, problem is that loop switching and similar transformations work by changing the level of the upper bound variables. When, as before, each upper bound variable corresponds to exactly one level and everything at that level uses the same upper bound variable, it easy enough to switch loops by changing the level of the globally available upper bound variables. With the new numbering scheme this will not work. It is necessary to scan up the chain of upper bounds and change the level of the upper bound variables found in that chain. This implies the upper bound variables in the chain are **not** necessarily in level order. If the upper bound variable for a particular level is desired it must be searched for, not indexed directly. This is done in UTFUPB (a new procedure). One of the strange implications of this is that given an arbitrary upper bound assignment, its controlling upper bound list might not contain all of its controlling upper bounds. But, if the search starts from an original statement (as opposed to a generated statement) this problem does not exist. The alternative approach of changing the links instead of just the levels would be worse because it would still be necessary to change to levels as well as do the additional work of sorting out the links. It would then be possible to know that the upper bound list is then in sorted level order; but, this is of no real value.

The handling of lower bound variables, index variables, and step variables is unchanged. These variables do not produce any dependences relevant to fusion.

5 Summary

PFC now extends dependence analysis to find the dependences that would exist between statements if their containing loops were fused. This implies extending the analysis past the common nesting level to all relevant levels. The absence of a backward dependence between two loops at the level in question allows those two loops to be fused thereby saving on task creation and synchronization overhead.