12 High level optimisation 3 - Loop optimisation

What are loops?

It is quite clear in many high level programs what is meant by the term loop. The syntactic structures of while and for loops delimit repetitions in the code. In the case of functional languages this is more complicated, since it is necessary to detect tail auto-recursion in functions. At the level of three address code or triples this structure is lost, but any method which can detect loops in such a low level representation may also detect loops written using gotos and labels in the high level code. It is important to define the concept of a loop and a method for detecting it.

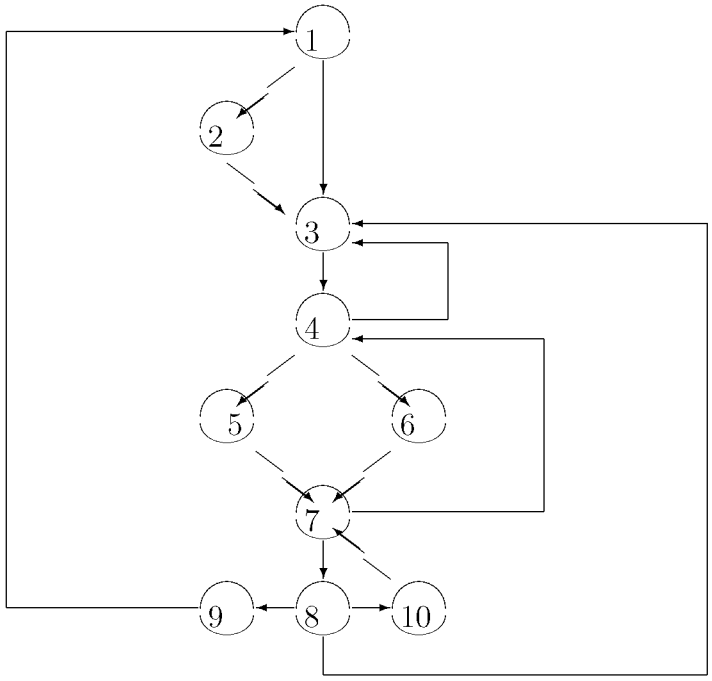
Loop definition

Put as simply as possible a loop is a collection of basic blocks in a flow graph with a single entry point and with all the blocks forming a strongly connected graph. This is much more general than the sort of thing allowed in a conventional high level program.

The requirement for strongly connected blocks ensures that whatever the current position of the program within the loop it could (but may not) revisit or visit any other part of the loop.

Loop detection

Following Aho and Ullman, we define loops as sets of blocks in a flowgraph where the heads of links dominate their tails. This assumes a property called dominance, which says that one block dominates another where all paths to the second block from the initial block of the graph must pass through the first. A node is assumed to dominate itself. Consider:



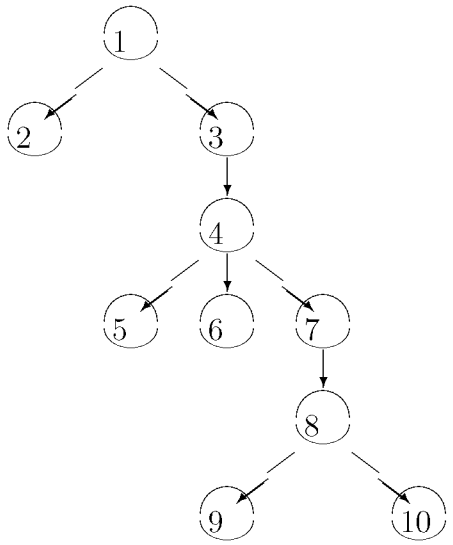
This can be translated into a dominator tree, where each parent dominates its children, grandchildren etc. and the initial node is the root of the tree. This is possible since dominance can be shown to be:

1. A reflexive partial order, so that

a	DOM	a					for all a	(reflexive)
a	DOM	b	and	b	DOM	a	implies $a = b$	(anti-symmetric)
a	DOM	b	and	b	DOM	c	implies a DOM	(transitive)

2. A linear ordering relation, so that the dominators of n appear in the same order on any path from the initial node.

The dominance tree for the flow graph above is:



From this analysis it is quite easy to find those edges in the graph whose heads dominate their tails, where for edge $a \rightarrow b$, b is the head and a is the tail. These are known as *back edges*.

Given a back edge $n \rightarrow d$, the natural loop containing this edge is found by finding those nodes that can reach n without going through d. In the example we can start with the fact that $10 \rightarrow 7$ is a back edge and find the loop consists of 7, 8 and 10.

Reducible flow graphs

For a special class of restricted flow graphs, known as reducible flow graphs, many pleasant properties hold. These include the fact that loops are unambiguously defined, dominators are easily found by simple algorithms and some data flow analysis is easily achieved.

In essence these graphs are guaranteed by high level language programming which does not use `gotos`, but restricts itself to `if-then-else` branches and `while/for/until` loops,

with `break` and `continue`. This corresponds to a restriction that *no forward jump into a loop is ever permitted.*

A definition of a reducible flow graph is:
G is reducible iff the edges can be partitioned into two disjoint sets, forward and back edges, where;

1. Forward edges form an acyclic graph where all nodes can be reached from the initial node;

2. Back edges are all edges whose heads dominate their tails.

Reducible flow graphs allow all loops to be identified by finding the natural loops of back edges. Furthermore all loops have a single header, which is essential for optimisation by code motion and induction variable elimination. In practice non-reducible loops are rare and can often be isolated from the rest of the code and so they ar ignored here.

Optimisations on loops

Having found the loops in the programs, we are now able to set about optimising them. We will consider a number of optimisations, namely:

code motion for loop invariants;

induction variable elimination;

loop unrolling and loop jamming.

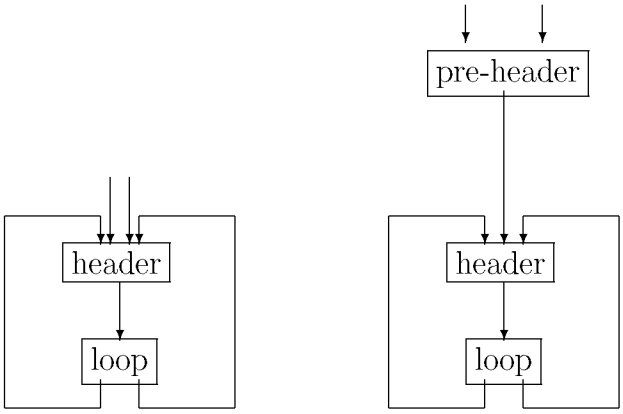
Code motion

We saw earlier that it may be possible to move out of the body of a loop the evaluation of expressions which remain constant over iterations of the loop body. Since these may not be expressible in the language of the program, substantial gains may be made. To maximise the benefits of such reordering, it is usual to begin with the most deeply nested loops and move code from them to their enclosing loop and then reapply the analysis to see what can be moved to the next enclosing level etc. This results in code propagating outwards as far as it remains invariant.

Loop invariant computations are detected by finding statements of the form

$$A \;:=\; B \text{ op } C$$

such that B and C are both defined by their values outside the loop and no route for defining them exists passing through the loop. This depends on a technique called *ud-chaining*, which will be considered under data flow analysis later in the course. Having identified this computation it can be moved to a new block immediately before the start of the loop. All previous incoming edges, except those back from the loop itself, to the header of the loop will now enter this block, which will have one outgoing edge to the loop header.



By repeated application of this approach, further statements may be identified, such as

$$D \quad := \quad A + C$$

It is important to be cautious about such motion, however, unless

1. The block containing the code moved dominates all exit blocks of the loop. Failure could:
   (a) increase the running time of the program if the other exit routes were used more frequently than that from which the code was moved;
   (b) cause incorrect results from running the program by moving a conditionally executed assignment to an unconditionally executed pre-header. This can be relaxed in a number of ways with further analysis, but we do not consider these here.
2. No other assignment is made within the loop to the subject of this assignment, A.
3. The assigned variable, A, must not be used elsewhere within the loop where its value derives from another route than the statement moved.

Example before optimisation

```
(1)     B1:   I := 1
(2)           J := 1
(3)           K := 1

(4)     B2:   if I>N goto B5
(5):    B3:   if J>N goto B6
(6)     B4:   T1 := 0[SP]
(7)           T2 := T1 - 4
(8)           T3 := 4 * I
(9)           T4 := T2[T3]
(10)          T5 := 4[SP]
(11)          T6 := T5 -4
(12)          T7 := 4 * J
(13)          T8 := T6[T7]
(14)          if T4 <= T8 goto B6

(15)    B5:   T9 := 8[SP]
(16)          T10 := T9 - 4
(17)          T11 := 4 * K
(18)          T12 := 4[SP]
(19)          T13 := T12 - 4
(20)          T14 := 4 * J
(21)          T15 := T13[T14]
(22)          T10[T11] := T15
(23)          J := J + 1
(24)          goto B7

(25)    B6:   T16 := 8[SP]
(26)          T17 := T16 -4
(27)          T18 := 4 * K
(28)          T19 := 0[SP]
(29)          T20 := T19 - 4
(30)          T21 := 4 * I
(31)          T22 := T20[T21]
(32)          T17[T18] := T22
(33)          I := I + 1

(34)    B7:   K := K + 1
(35)          T23 := 2 * N
(36)          if K <= T23 goto B2
```

Example after code motion

```
(1)     B1:  I := 1
(2)          J := 1
(3)          K := 1

(6)     B8:  T1 := 0[SP]
(10)         T5 := 4[SP]
(15)         T9 := 8[SP]
(18)         T12 := 4[SP]
(25)         T16 := 8[SP]
(28)         T19 := 0[SP]
(35)         T23 := 2 * N
(7)          T2 := T1 - 4
(11)         T6 := T5 -4
(16)         T10 := T9 - 4
(19)         T13 := T12 - 4
(26)         T17 := T16 -4
(29)         T20 := T19 - 4

(4)     B2:  if I>N goto B5

(5):    B3:  if J>N goto B6

(8)     B4:  T3 := 4 * I
(9)          T4 := T2[T3]
(12)         T7 := 4 * J
(13)         T8 := T6[T7]
(14)         if T4 <= T8 goto B6
 (17)   B5:  T11 := 4 * K
(20)         T14 := 4 * J
(21)         T15 := T13[T14]
(22)         T10[T11] := T15
(23)         J := J + 1
(24)         goto B7

(27)    B6:  T18 := 4 * K
(30)         T21 := 4 * I
(31)         T22 := T20[T21]
(32)         T17[T18] := T22
(33)         I := I + 1

(34)    B7:  K := K + 1
(36)         if K <= T23 goto B2
```

Induction variable elimination

Induction variables are those which advance arithmetically at the top of the loop (i.e. in the header block and before they are used). This can be established in the same analysis as invariant identification. The task is then to replace each use with an addition to the value of the previous iteration. The commonest situation is probably where a loop control variable is used to control indexing into a number of arrays (or different places in the same array). Even

simple indexing usually requires some adjustment of the offset by a constant factor for each offset.

The method for handling induction variables is:

1. Find all basic induction variables, i.e. those of the form

   $I := I \pm C$

   where C is a constant or does not change within the loop

   This uses the invariance information as before.

2. Find additional induction variables, belonging to the family of a basic induction variable. Thus induction variables, A, of the family of variable B will have the form

   $A := C_1 B + C_2$,

   which we refer to as $SFA(B)$

   where C1 and C2 are constant or invariant. The following five forms can be identified:

   $A := B * C$, $A := C * B$, $A := B/C$, $A := B \pm C$, $A := C \pm B$

   where C is a constant or invariant and B is an induction variable (basic or otherwise). If B is not basic, some additional restrictions apply to it, i.e.

   (a) that there be no intervening assignment to the basic induction variable to whose family it belongs between Bs assignment and its use in assigning to A;

   (b) that no definition of B made outside the loop reaches A.

3. For each basic induction variable B in turn, consider each A in its its family:

   (a) create a new name $SFA(B)$ (if 2 As have the same linear function $FA(B)$ give them the same name)

   (b) Replace the assignment to A with this name, $A := SFA(B)$

   (c) Assign the value of $FA(B)$ to $SFA(B)$ at the end of the preheader.

$$\begin{aligned} \text{if} FA(B) \text{is} C_1 B + C_2 \text{add,} & \\ SFA(B) &:= C_1 * B \\ SFA(B) &:= SFA(B) + C_2 \end{aligned}$$

   (d) Immediately after every $B := B + D$, where D is invariant, add

$$SFA(B) := SFA(B) + C_1 * D$$

   where $FA(B) = C_1 B + C_2$.

   If D is a loop -invariant name create a new, loop-invariant temporary for $C1 * D$.

4. For each basic induction variable, B, only used to compute values in its family, look for each test of the form

   `if` B `relop` X `goto` Y

   and choose the simplest A in Bs family and substitute

$$\begin{array}{lll} R & := & C_1 * X \\ R & := & R + C_2 \\ \text{if} & SFA & \text{relop } R \text{ goto } Y \end{array}$$

   `if` X `relop` B can be handled in the same way.

   Now delete all assignments to B in the loop, as none are needed.

5. Finally for each induction variable, A, introduced in (3) find if there can be no assignment to $SFA(B)$ between the introduced statement $A := SFA(B)$ and any use of A, replace all uses of A by the name $SFA(B)$ and delete the assignment to A.

This depends heavily on data flow analysis, but can result in dramatic improvements in code quality.

Example after induction variable elimination

```
(1)   B1:  I := 1
(2)        J := 1
(3)        K := 1

(6)   B8:  T1 := 0[SP]
(10)       T5 := 4[SP]
(15)       T9 := 8[SP]
(35)       T23 := 2 * N
(7)        T2 := T1 - 4
(11)       T6 := T5 -4
(16)       T10 := T9 - 4
           S4*I := 4 * I
           S4*J := 4 * J
           S4*K := 4 * K
           R1 := 4 * N
           R2 := 4 * T23

(4)   B2:  if S4*I>R1 goto B5

(5):  B3:  if S4*J>R1 goto B6

(9)   B4:  T4 := T2[S4*I]
(13)       T8 := T6[S4*J]
(14)       if T4 <= T8 goto B6

(21)  B5:  T15 := T6[S4*J]
(22)       T10[S4*K] := T15
(23)       S4*J := S4*J + 1
(24)       goto B7

(31)  B6:  T22 := T2[S4*I]
(32)       T10[S4*K] := T22
           S4*I := S4*I + 4

(34)  B7:  S4*K := S4*K + 4
(36)       if S4*K <= R2 goto B2
```

Loop unrolling

Another simple optimisation is to identify loops where each iteration is independent of its predecessor and generate a sequential series of repetitions of the code. This may be done for more than simple counting loops, as long as the initial values in each iteration can be set at the point of starting the loop. A combination of invariant analysis and induction variable analysis can help in this.

Loop jamming

A further extension is to identify loops which have the same induction variables and combine them into one loop. Thus two loops traversing identical sized arrays can be merged. It

is important to remember that some of those loops which appear to be capable of such combination may not be safe to combine, as the second may be dependent on the completion of the first.