Loop Motion

Per eseguire questo assignment abbiamo rivisto la teoria della loop motion vista in classe e abbiamo provato ad applicarla al codice di esempio dato.

Algoritmo visto a lezione

Dato un insieme di nodi in un loop

- Calcolare le **reaching definitions**
- Trovare le istruzioni **loop-invariant**
- Calcolare i **dominatori** (dominance tree)
- Trovare le uscite del loop (i successori fuori dal loop)
- Le istruzioni candidate alla code motion:
    - Sono **loop invariant**
    - Si trovano in blocchi che **dominano tutte le uscite** del loop
    - **Assegnano un valore a variabili non assegnate altrove** nel loop
    - Si trovano in blocchi che **dominano tutti i blocchi nel loop** che usano la variabile a cui si sta assegnando un valore
- Eseguire una ricerca depth-first dei blocchi
    - **Spostare l'istruzione candidata nel preheader** se tutte le istruzioni invarianti da cui questa dipende sono state spostate

Quindi l'obiettivo principale è trovare le istruzioni che abbiano i requisiti di code motion:

- sono Loop invariant → facciamo uno studio sugli operandi e sulle loro reachin definition
- Si trovano in blocchi che dominano tutte le uscite del loop → calcoliamo il dominance tree
- Assegnano un valore a variabili non assegnate altrove nel loop → questo è rispettato di base dalla IR che utilizziamo, dato che la forma SSA non permette il riassegnamento delle variabili
- Si trovano in blocchi che dominano tutti i blocchi nel loop che usano la variabile a cui si sta assegnando un valore → utilizziamo il dominance tree

Analisi del codice dato

```
LOOP:
  z = z + 1;
  y = c + 3;
  q = c + 7; l
  if (z < 5) {
    a = a + 2;
    h = c + 3;
;   } else {
    a = a - 1;
    h = c + 4;
    if (z >= 10) {
;     goto EXIT;
;   }
; }
  m = y + 7;
  n = h + 2;
  y = c + 7;
  r = q + 5;
  goto LOOP;
; EXIT:
  printf("%d,%d,%d,%d,%d,%d,%d,%d\n", a, h, m, n, q, r, y, z);
; }
```

BB3

BB8

BB11

BB15

BB20

Analizzando il codice di esempio dato possiamo notare che solo due istruzioni rispettano i requisiti della code motion:

- y = c + 3
  - loop invariant perchè:
    - c è un parametro della funzione
    - 3 è una costante
  - domina tutte le uscite del loop, perchè per ogni uscita del loop bisogna passare dal blocco 3
  - La variabile che definisce non viene riassegnata prima del suo uso: (m= y + 7) viene riassegnata dopo in y = c + 7, ma quest'ultima è una definizione inutile perchè al prossimo ciclo del loop y viene ridefinita da y = c + 3
  - domina il blocco del suo unico uso
- q = c + 7
  - loop invariant perchè:
    - c è un parametro della funzione
    - 7 è una costante
  - domina tutte le uscite del loop, perchè per ogni uscita del loop bisogna passare dal blocco 3
  - La variabile che definisce non viene riassegnata prima nel loop
  - domina il blocco del suo unico uso (r = q + 5)

Soluzione

L'obiettivo del nostro algoritmo di ottimizzazione è identificare le istruzioni all'interno di un loop che non dipendono dalle iterazioni del loop stesso e spostarle al di fuori del loop, riducendo così il numero di volte che vengono eseguite.

L'algoritmo è implementato come una classe chiamata "**LoopInvariantCodeMotion**" che eredita dalla classe "**LoopPass**" del framework LLVM.
La classe LoopInvariantCodeMotion rappresenta un passo di ottimizzazione che può essere eseguito sui loop all'interno di una funzione e contiene la dipendenza richiesta per l'analisi con il DominanceTree.

Algoritmo

Abbiamo implementato l'algoritmo nel metodo onRunLoop e creato funzioni di supporto per dividerne la complessità.
Passi dell'algoritmo:

1. Verifica se il loop è nella forma normalizzata, ovvero ha un preheader (blocco di ingresso al loop) e i blocchi interni del loop sono collegati correttamente
2. Per ogni blocco all'interno del loop:
    a. Per ogni istruzione all'interno del blocco:
        i. Verifica se l'istruzione è un'operazione binaria (BinaryOperator).
        ii. Verifica se l'istruzione è **loop-invariant**, ovvero non dipende dalle variabili del loop. Questo viene fatto attraverso la funzione isLoopInvariant.
        iii. Se l'istruzione è loop-invariant e domina tutte le uscite del loop, verifica se domina anche tutti i blocchi che utilizzano la stessa variabile. Questo viene fatto attraverso le funzioni checkDominatesAllExits e checkDominatesAllUsers.
        iv. Se l'istruzione soddisfa i requisiti di code motion, viene aggiunta alla lista delle istruzioni da spostare al di fuori del loop.
3. Sposta le istruzioni selezionate al di fuori del loop, prima del terminatore del preheader del loop.

L'algoritmo utilizza diverse funzioni di supporto:

- isLoopInvariant → controlla se i due operatori dell'istruzione sono indipendenti dal loop con la funzione isLoopInvariantOperand, dove:
    - Controlla se l'operando è una costante o un argomento della funzione → loop-invariant
    - Controlla se la reaching definition è una phi → loop-variant
    - Altrimenti se la reaching definition è un'altra istruzione, controllo:
        - se è all'interno del loop → loop-invariant
        - altrimenti controlla se quell'istruzione è loop invariant o meno (in modo ricorsivo)
- checkDominatesAllExits → controlla se il blocco dove è definita l'istruzione domina tutti i blocchi predecessori di tutte le uscite del loop, grazie al DominatorTree
- checkDominatesAllUsers → controlla se il blocco dove è definita l'istruzione domina tutti i blocchi in cui viene usata la variabile definita dall'istruzione.
    - usesVariable → controlla se l'istruzione usa quella variabile o meno

Output del programma

```
LOOPPASS INIZIATO...
FORMA NORMALIZZATA
PREHEADER
  br label %3

Basic Block: %3
  %.05 = phi i32 [ 0, %2 ], [ %19, %15 ]
  %.04 = phi i32 [ 0, %2 ], [ %17, %15 ]
  %.03 = phi i32 [ 0, %2 ], [ %16, %15 ]
  %.01 = phi i32 [ 9, %2 ], [ %.1, %15 ]
  %.0 = phi i32 [ %1, %2 ], [ %4, %15 ]
  %4 = add nsw i32 %.0, 1       ###Vediamo dentro###
        Reaching definitions:   %.0 = phi i32 [ %1, %2 ], [ %4, %15 ]
                Non è loop invariant, perchè è una phi node
        **ISTRUZIONE LOOP VARIANT**

  %5 = add nsw i32 %0, 3        ###Vediamo dentro###
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 3
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT E DOMINANTE SU TUTTE LE USCITE E TUTTI I BLOCCHI
CHE USANO LA VARIABILE**

  %6 = add nsw i32 %0, 7        ###Vediamo dentro###
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 7
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT E DOMINANTE SU TUTTE LE USCITE E TUTTI I BLOCCHI
CHE USANO LA VARIABILE**

  %7 = icmp slt i32 %4, 5
  br i1 %7, label %8, label %11


Basic Block: %11
  %12 = sub nsw i32 %.01, 1     ###Vediamo dentro###
        Reaching definitions:   %.01 = phi i32 [ 9, %2 ], [ %.1, %15 ]
                Non è loop invariant, perchè è una phi node
        **ISTRUZIONE LOOP VARIANT**

  %13 = add nsw i32 %0, 4       ###Vediamo dentro###
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 4
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT NON DOMINA TUTTE LE USCITE DEL LOOP**

  %14 = icmp sge i32 %4, 10
```

```
  br i1 %14, label %20, label %15


Basic Block: %8
  %9 = add nsw i32 %.01, 2      ###Vediamo dentro###
        Reaching definitions:   %.01 = phi i32 [ 9, %2 ], [ %.1, %15 ]
                Non è loop invariant, perchè è una phi node
        **ISTRUZIONE LOOP VARIANT**

  %10 = add nsw i32 %0, 3       ###Vediamo dentro###
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 3
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT NON DOMINA TUTTE LE USCITE DEL LOOP**

  br label %15


Basic Block: %15
  %.02 = phi i32 [ %10, %8 ], [ %13, %11 ]
  %.1 = phi i32 [ %9, %8 ], [ %12, %11 ]
  %16 = add nsw i32 %5, 7       ###Vediamo dentro###
        Reaching definitions:   %5 = add nsw i32 %0, 3
                L'operando è un'istruzione, blocco: %3
                L'istruzione è dentro al loop,
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 3
                E' loop invariant, perchè è un'argomento o costante
                L'istruzione padre è loop-invariant, quindi anche l'operando non
dipende dal loop
        Reaching definitions: i32 7
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT NON DOMINA TUTTE LE USCITE DEL LOOP**

  %17 = add nsw i32 %.02, 2     ###Vediamo dentro###
        Reaching definitions:   %.02 = phi i32 [ %10, %8 ], [ %13, %11 ]
                Non è loop invariant, perchè è una phi node
        **ISTRUZIONE LOOP VARIANT**

  %18 = add nsw i32 %0, 7       ###Vediamo dentro###
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 7
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT NON DOMINA TUTTE LE USCITE DEL LOOP**

  %19 = add nsw i32 %6, 5       ###Vediamo dentro###
        Reaching definitions:   %6 = add nsw i32 %0, 7
                L'operando è un'istruzione, blocco: %3
                L'istruzione è dentro al loop,
```

```
        Reaching definitions: i32 %0
                E' loop invariant, perchè è un'argomento o costante
        Reaching definitions: i32 7
                E' loop invariant, perchè è un'argomento o costante
                L'istruzione padre è loop-invariant, quindi anche l'operando non
dipende dal loop
        Reaching definitions: i32 5
                E' loop invariant, perchè è un'argomento o costante
        **ISTRUZIONE LOOP INVARIANT NON DOMINA TUTTE LE USCITE DEL LOOP**

  br label %3




ISTRUZIONE CHE RISPETTANO I REQUISITI DI CODE MOTION DAL LOOP
  %5 = add nsw i32 %0, 3
  %6 = add nsw i32 %0, 7
```