

RELAZIONE ASSIGNMENT SULLA LOOP INVARIANT CODE MOTION

L'assignment che ci è stato assegnato riguardava l'implementazione di analisi e della conseguente ottimizzazione relativa alla Loop Invariant Code Motion, ovvero quell'ottimizzazione che permette di spostare all'esterno del loop tutte le istruzioni che sono indipendenti dalle varie iterazioni del ciclo; in questa maniera, riusciamo ad ottimizzare l'intero loop, visto che a seguito del passo di ottimizzazione ci sarà un numero minore di istruzioni da eseguire per ogni iterazione del loop.


Per riuscire ad implementare nella maniera corretta la Loop Invariant Code Motion abbiamo visto che esiste un algoritmo preciso che consente di fare ciò, e questa immagine lo riassume in maniera chiara e concisa.

Algoritmo per la Code Motion

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

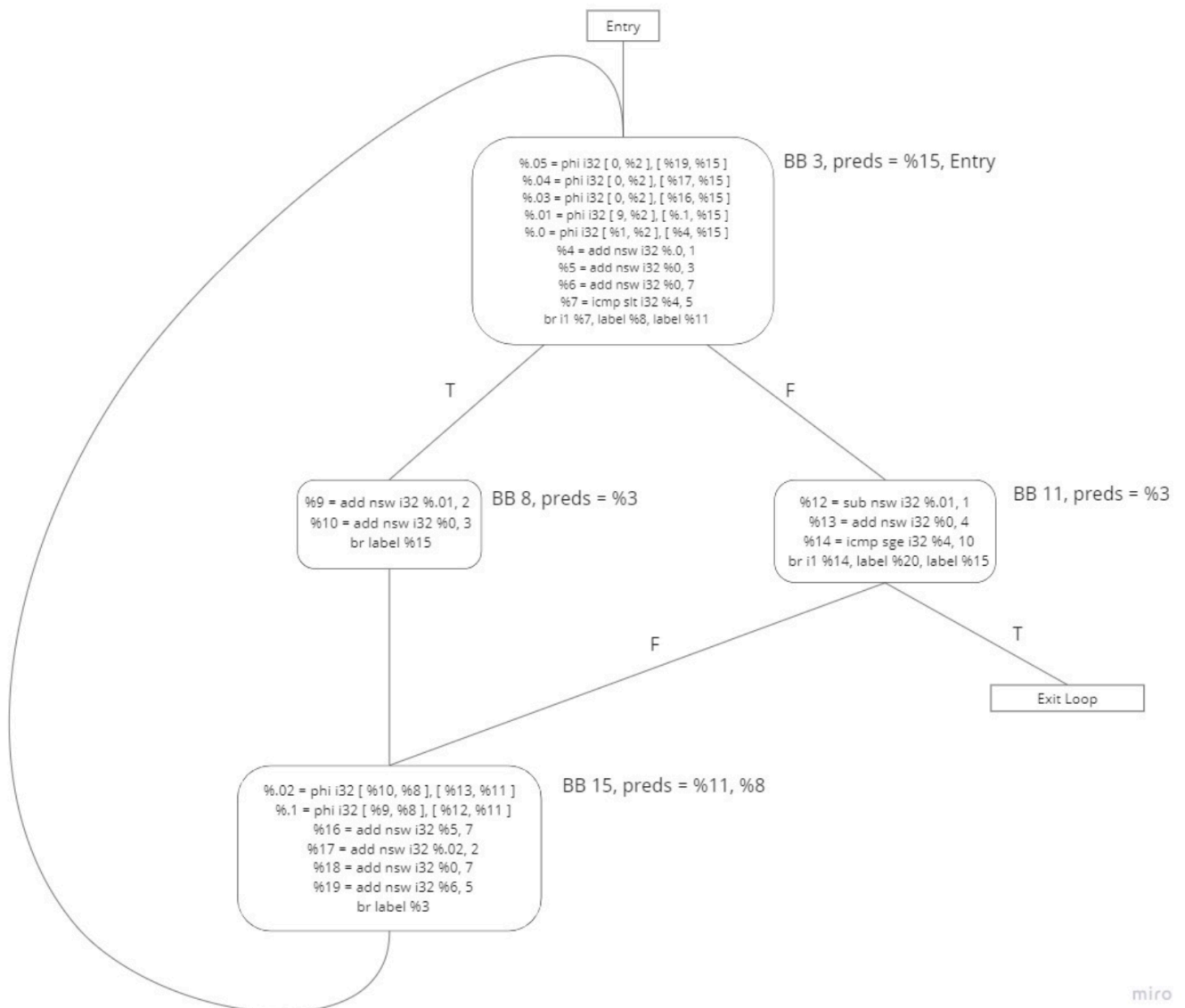
Oppure la variabile definita dall'istruzione è *dead* all'uscita del loop



I vari passi che sono segnati all'interno di questa slide rappresentano una situazione generica, sulla quale non vengono fatte ulteriori supposizioni.

Noi, in realtà, dobbiamo andare ad implementare questo algoritmo utilizzando la logica di lavoro di LLVM, che lavora con una rappresentazione intermedia

di tipo SSA, di conseguenza, alcune delle condizioni che sono indicate all'interno dell'algoritmo sono già rispettate grazie all'uso appunto della forma SSA: per esempio, l'IR di LLVM ci garantisce già che una istruzione assegnata in un punto del loop non lo sia anche in un altro punto dello stesso loop (questo grazie alla forma SSA, un solo assegnamento per ogni variabile), e quindi tale condizione non è da verificare.



L'immagine sopra rappresenta il control flow graph del loop che avevamo come esempio su cui implementare la Loop Invariant Code Motion; i vari rami che sono presenti all'interno del loop rappresentano i vari controlli e salti condizionati che ci sono anche all'interno del loop stesso e che possono nel caso alterare il flusso del loop.

La strategia che abbiamo adottato per riuscire ad implementare la Loop Invariant Code Motion e' stata quella di andare a definire una funzione per ogni singola condizione da verificare per assicurarsi che una istruzione fosse effettivamente movable e che quindi si potesse spostare al di fuori del loop, all'interno del blocco di preheader. Per riuscire ad effettuare questa ottimizzazione, pero', erano necessari alcuni strumenti e strutture dati ottenibili solamente mediante l'esecuzione precedente di alcuni passi di analisi: infatti, e' stato necessario all'interno del programma, andare a specificare che il nostro passo di ottimizzazione richiedesse l'esecuzione sia del passo di analisi che raccogliesse informazioni sul loop in questione (per disporre di tutte le informazioni per riuscire ad effettuare i controlli prima di eseguire la Loop Invariant Code Motion), sia del

passo di analisi che si occupa di costruire il Dominance Tree (questa struttura dati, infatti, risulta essere fondamentale per riuscire a determinare le istruzioni movable).

```
virtual void getAnalysisUsage(AnalysisUsage &AU) const override {  
    AU.setPreservesAll(); //indica di preservare i risultati di questo passo di analisi  
    AU.addRequired<DominatorTreeWrapperPass>();  
    AU.addRequired<LoopInfoWrapperPass>();  
}
```

Queste qui sono le istruzioni che abbiamo specificato all'interno del passo per indicare di eseguire il passo corrente solamente dopo aver eseguito quelli specificati (la prima istruzione indica solo di salvare tutti i possibili risultati che il passo attuale restituirà alla fine).

La vera e propria implementazione del passo di ottimizzazione inizia con una verifica necessaria effettuata sul loop, per andare a controllare che questo sia un loop naturale, mediante la chiamata del metodo `isLoopSimplyForm()` della classe `Loop` di LLVM. Una volta verificato che effettivamente il loop è un loop naturale, allora possiamo procedere a salvare, mediante un puntatore, il riferimento al blocco di preheader, che ci servirà quando andremo a spostare le istruzioni Loop Invariant e movable al di fuori dei basic block che compongono il corpo del loop.

Adesso, per riuscire ad effettuare i controlli sulle varie istruzioni, andiamo ad iterare sui vari basic block che compongono il loop e poi, per ogni singolo basic block individuato, andiamo a scorrere le istruzioni che lo compongono.

```
for (Loop::block_iterator BI = L->block_begin(); BI != L->block_end(); ++BI) {  
    actualBB = *BI;  
    for (auto &instr : *actualBB) { //scorro le istruzioni all'interno del singolo basic block  
        //se l'istruzione individuata è loop invariant, allora procedo ad analizzarla  
        if (isLoopInvariant(&instr, L)) {  
            outs() << "La seguente istruzione è Loop Invariant: ";  
            instr.print(outs());  
            outs() << "\n";  
            //verifico se la definizione individuata domina tutti gli usi all'interno del loop  
            if (!dominateUses(&instr, L, DT)) {  
                continue; //l'istruzione non è movable  
            }  
            outs() << "La seguente istruzione domina tutti i suoi usi: ";  
            instr.print(outs());  
            outs() << "\n";  
            if (!dominateExits(&instr, L, DT, &BBvector)) {  
                continue;  
            }  
            outs() << "La seguente istruzione domina tutte le uscite del loop: ";  
            instr.print(outs());  
            outs() << "\n";  
  
            outs() << "\nSono pronto a portare fuori dal loop l'istruzione!\n";  
            LLVMContext& C = instr.getContext();  
            MDNode* N = MDNode::get(C, MDString::get(C, "hoistable")); //creo il meta-data  
            instr.setMetadata("HOIST", N);  
            Movablevector.push_back(&instr);  
        }  
    }  
}
```

Questa immagine rappresenta la sequenza delle varie chiamate che vengono effettuate su ogni singola istruzione all'interno del loop: la prima funzione che viene invocata procede a verificare se tale istruzione è

loop invariant, controllando tutti i possibili casi.

Se tale istruzione e' Loop Invariant, allora si procede richiamando le altre funzioni che vanno a verificare prima se tale istruzione domina tutti i suoi usi all'interno del loop, e poi la funzione che procede a verificare se tale istruzione domina tutte le possibili uscite del loop.

Una delle condizioni che sono necessarie per poter considerare una istruzione di definizione di una variabile come movable, ovvero il fatto che tale variabile sia definita solo una volta all'interno del loop, risulta essere gia' verificata dalla forma SSA adottata da LLVM: essendo infatti possibile effettuare un solo assegnamento per ogni 'versione' della stessa variabile, non e' possibile procedere a ridefinire tale 'versione', ma potremo solamente andare a definirne una nuova.

La seconda funzione che viene richiamata all'interno del programma e' quella che procede a verificare se l'istruzione che e' verificata essere Loop Invariant domina anche tutti i suoi usi all'interno del loop; se cio' accade, allora si procede a verificare l'ultima condizione, ovvero se tale istruzione domina tutte le possibili uscite dal loop.

Se tutte le condizioni risultano verificate, allora quello che si fa e' andare a marcare tale istruzione come movable, andando ad aggiungere un metadato all'istruzione movable, prima di procedere allo spostamento dell'istruzione stessa all'interno del blocco di preheader del loop.

Nella seguente slide potete vedere una immagine del programma prima del passo di ottimizzazione e dopo il passo di ottimizzazione

Il codice che si vede a sinistra e' quello che si ottiene dopo l'applicazione del passo di

```
@.str = private constant [25 x i8] c"%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\0A\00", align 1

define void @foo(i32 %0, i32 %1) {
    %3 = add nsw i32 %0, 3, !HOIST !0
    %4 = add nsw i32 %0, 7, !HOIST !0
    br label %5

5:                                ; preds = %15, %2
    %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 ], [ %6, %15 ]
    %6 = add nsw i32 %0, 1
    %7 = icmp slt i32 %6, 5
    br i1 %7, label %8, label %11

8:                                ; preds = %5
    %9 = add nsw i32 %01, 2
    %10 = add nsw i32 %0, 3
    br label %15

11:                               ; preds = %5
    %12 = sub nsw i32 %01, 1
    %13 = add nsw i32 %0, 4
    %14 = icmp sge i32 %6, 10
    br i1 %14, label %20, label %15

15:                               ; preds = %11, %8
    %02 = phi i32 [ %10, %8 ], [ %13, %11 ]
    %1 = phi i32 [ %0, %8 ], [ %12, %11 ]
    %16 = add nsw i32 %3, 7
    %17 = add nsw i32 %02, 2
    %18 = add nsw i32 %0, 7
    %19 = add nsw i32 %4, 5
    br label %5

20:                               ; preds = %11
    %21 = call i32@ (i8*, ...) @printf(i8* getelementptr inbounds ([25 x i8], [25 x i8]* @.str, 164 0, 164 0), i32 %12,
    ret void
}

declare i32 @printf(i8*, ...)

define i32 @main() {
    call void @foo(i32 0, i32 4)
    call void @foo(i32 0, i32 12)
    ret i32 0
}

!0 = !{"hoistable"}
```

```
30:    foo(0, 4);
31:    foo(0, 12);
32:    return 0;
33: }
34
35: @.str = private constant [25 x i8] c"%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\0A\00", align 1
36
37: define void @foo(i32 %0, i32 %1) {
38:     br label %3
39
40: 3:                                ; preds = %15, %2
41:     %05 = phi i32 [ 0, %2 ], [ %19, %15 ]
42:     %04 = phi i32 [ 0, %2 ], [ %17, %15 ]
43:     %03 = phi i32 [ 0, %2 ], [ %16, %15 ]
44:     %01 = phi i32 [ 9, %2 ], [ %1, %15 ]
45:     %0 = phi i32 [ %1, %2 ], [ %6, %15 ]
46:     %4 = add nsw i32 %0, 1
47:     %5 = add nsw i32 %0, 3
48:     %6 = add nsw i32 %0, 7
49:     %7 = icmp slt i32 %4, 5
50:     br i1 %7, label %8, label %11
51
52: 8:                                ; preds = %3
53:     %9 = add nsw i32 %01, 2
54:     %10 = add nsw i32 %0, 3
55:     br label %15
56
57: 11:                               ; preds = %3
58:     %12 = sub nsw i32 %01, 1
59:     %13 = add nsw i32 %0, 4
60:     %14 = icmp sge i32 %4, 10
61:     br i1 %14, label %20, label %15
62
63: 15:                               ; preds = %11, %8
64:     %02 = phi i32 [ %10, %8 ], [ %13, %11 ]
65:     %1 = phi i32 [ %0, %8 ], [ %12, %11 ]
66:     %16 = add nsw i32 %5, 7
67:     %17 = add nsw i32 %02, 2
68:     %18 = add nsw i32 %0, 7
69:     %19 = add nsw i32 %6, 5
70:     br label %3
71
72: 20:                               ; preds = %11
73:     %21 = call i32@ (i8*, ...) @printf(i8* getelementptr inbounds ([25 x i8], [25 x i8]* @.str, 164 0, 164 0), i32 %12,
74:     ret void
75: }
76
77: declare i32 @printf(i8*, ...)
78
79: define i32 @main() {
80:     call void @foo(i32 0, i32 4)
81:     call void @foo(i32 0, i32 12)
82:     ret i32 0
}
```

ottimizzazione sul codice presente sul lato destro.