



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# 14. Assignment 4: Loop Fusion

## Compilatori – Middle end [I215-014]

*Corso di Laurea in INFORMATICA*  
(D.M.270/04) [16-215]  
Anno accademico 2024/2025

**Prof. Andrea Marongiu**  
[andrea.marongiu@unimore.it](mailto:andrea.marongiu@unimore.it)

# Copyright note

*È vietata la copia e la riproduzione dei contenuti e immagini in qualsiasi forma.*

*È inoltre vietata la redistribuzione e la pubblicazione dei contenuti e immagini non autorizzata espressamente dall'autore o dall'Università di Modena e Reggio Emilia.*

# Credits

- Barton, IBM Canada, “Revisiting Loop Fusion and its place in the loop transformation framework”
- Absar, ARM L.t.d., “Scalar Evolution Demystified”

# Struttura del passo

- Ci serve un `FUNCTION_PASS`, tramite il quale accedere alle informazioni sui loop

```
PreservedAnalyses run(Function &F,  
    FunctionAnalysisManager &AM) {  
    LoopInfo &LI = AM.getResult<LoopAnalysis>(F);  
    for (auto *L : LI)  
        ...  
}
```

```
#include "<a href='\"#\"'>llvm/Analysis/LoopInfo.h\"</a>"
```

Dentro il file header del vostro passo

# Struttura del passo

- Manipolazione dei Loop  
(esempio dal passo LoopVersioning)

```
// Build up a worklist of inner-loops to version. This is necessary as the
// act of versioning a loop creates new loops and can invalidate iterators
// across the loops.
SmallVector<Loop *, 8> Worklist;

for (Loop *TopLevelLoop : *LI)
    for (Loop *L : depth_first(TopLevelLoop))
        // We only handle inner-most loops.
        if (L->isInnermost())
            Worklist.push_back(L);
```

# Struttura del passo

- Esplorare i metodi delle classi LoopInfo e LoopInfoBase

# Loop Fusion

In order for two loops,  $L_j$  and  $L_k$  to be fused, they must satisfy the following conditions:

*1.  $L_j$  and  $L_k$  must be adjacent*

- *There cannot be any statements that execute between the end of  $L_j$  and the beginning of  $L_k$*

*2.  $L_j$  and  $L_k$  must iterate the same number of times*

*3.  $L_j$  and  $L_k$  must be control flow equivalent*

- *When  $L_j$  executes  $L_k$  also executes or when  $L_k$  executes  $L_j$  also executes*

*4. There cannot be any negative distance dependencies between  $L_j$  and  $L_k$*

- *A negative distance dependence occurs between  $L_j$  and  $L_k$ ,  $L_j$  before  $L_k$ , when at iteration  $m$  from  $L_k$  uses a value that is computed by  $L_j$  at a future iteration  $m+n$  (where  $n > 0$ ).*

# Loop Fusion

In order for two loops,  $L_j$  and  $L_k$  to be fused, they must satisfy the following conditions:

*1.  $L_j$  and  $L_k$  must be adjacent*

- *There cannot be any statements that execute between the end of  $L_j$  and the beginning of  $L_k$*

*2.  $L_j$  and  $L_k$  must iterate the same number of times*

*3.  $L_j$  and  $L_k$  must be control flow equivalent*

- *When  $L_j$  executes  $L_k$  also executes or when  $L_k$  executes  $L_j$  also executes*

*4. There cannot be any negative distance dependencies between  $L_j$  and  $L_k$*

- *A negative distance dependence occurs between  $L_j$  and  $L_k$ ,  $L_j$  before  $L_k$ , when at iteration  $m$  from  $L_k$  uses a value that is computed by  $L_j$  at a future iteration  $m+n$  (where  $n > 0$ ).*



# 1) Adiacenza dei loop

Due loop L0 e L1 sono adiacenti se non ci sono basic blocks aggiuntivi nel CFG tra l'uscita di L0 e l'entry di L1.

Se i loop sono **guarded** il successore non loop del guard branch di L0 deve essere l'entry block di L1.

Se i loop **non sono guarded** l'exit block di L0 deve essere il preheader di L1

<https://llvm.org/docs/LoopTerminology.html>

# Loop Fusion

In order for two loops,  $L_j$  and  $L_k$  to be fused, they must satisfy the following conditions:

1.  *$L_j$  and  $L_k$  must be adjacent*

- *There cannot be any statements that execute between the end of  $L_j$  and the beginning of  $L_k$*

2.  *$L_j$  and  $L_k$  must iterate the same number of times*

3.  *$L_j$  and  $L_k$  must be control flow equivalent*

- *When  $L_j$  executes  $L_k$  also executes or when  $L_k$  executes  $L_j$  also executes*

4. *There cannot be any negative distance dependencies between  $L_j$  and  $L_k$*

- *A negative distance dependence occurs between  $L_j$  and  $L_k$ ,  $L_j$  before  $L_k$ , when at iteration  $m$  from  $L_k$  uses a value that is computed by  $L_j$  at a future iteration  $m+n$  (where  $n > 0$ ).*

### 3) Control flow equivalence

- Due loop L0 e L1 si dicono control flow equivalent se quando uno esegue è garantito che esegua anche l'altro.
- Per determinare la control flow equivalenza ci servono le informazioni di dominanza e postdominanza
  - Se L0 domina L1 e L1 postdomina L0 allora i due loop sono control flow equivalenti

# Analisi di dominanza e postdominanza

```
PreservedAnalyses run(Function &F,  
FunctionAnalysisManager &AM) {  
    DominatorTree &DT =  
        AM.getResult<DominatorTreeAnalysis>(F);  
    PostDominatorTree &PDT =  
        AM.getResult<PostDominatorTreeAnalysis>(F);  
    ...  
}
```

```
#include "llvm/IR/Dominators.h"  
#include "llvm/Analysis/PostDominators.h"
```

Dentro il file header del vostro passo

# Loop Fusion

In order for two loops,  $L_j$  and  $L_k$  to be fused, they must satisfy the following conditions:

1.  *$L_j$  and  $L_k$  must be adjacent*

- *There cannot be any statements that execute between the end of  $L_j$  and the beginning of  $L_k$*

2.  *$L_j$  and  $L_k$  must iterate the same number of times*

3.  *$L_j$  and  $L_k$  must be control flow equivalent*

- *When  $L_j$  executes  $L_k$  also executes or when  $L_k$  executes  $L_j$  also executes*

4. *There cannot be any negative distance dependencies between  $L_j$  and  $L_k$*

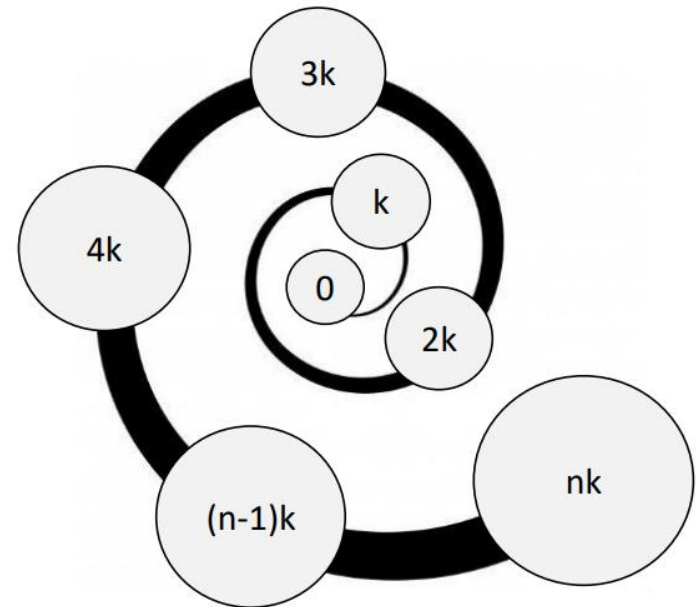
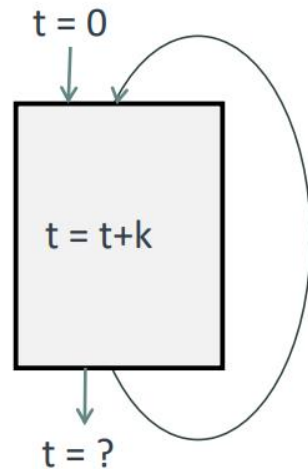
- *A negative distance dependence occurs between  $L_j$  and  $L_k$ ,  $L_j$  before  $L_k$ , when at iteration  $m$  from  $L_k$  uses a value that is computed by  $L_j$  at a future iteration  $m+n$  (where  $n > 0$ ).*

# Loop Trip Count

- La seconda condizione per la Loop Fusion richiede che si determini il *trip count* dei loop
- Il passo di analisi che, tra le altre cose, determina questo tipo di informazione è la *Scalar Evolution*
- ***Scalar Evolution***: *Change in the value of scalar variables over iterations of the loop*

# Scalar Evolution

```
void foo(int *a, int n, int k) {  
    int t = 0;  
    for (int i = 0; i < n; i++)  
        t = t + k;  
    *a = t;  
}
```



# Scalar Evolution

## EXAMPLE: Induction variable simplification

```
1. for.body:
2.  %i = phi i32 [ %inc, %for.body ], [ 0, %for.body.preheader ]
3.  %t = phi i32 [ %tk, %for.body ], [ 0, %for.body.preheader ]
4.  %tk = add nsw i32 %t, %k
5.  %inc = add nuw nsw i32 %i, 1
6.  %cmp = icmp slt i32 %inc, %n
7.  br i1 %cmp, label %for.body, label %for.cond.cleanup
...
for.cond.cleanup:
  %tk.final = phi i32 [ 0, %entry ], [ %tk, %for.body]
8.  store i32 %tk.final, i32* %a
...
```

\*\*\* IR Dump After Induction Variable Simplification \*\*\*

```
...
1'. for.cond.cleanup.loopexit:
2'.  %tk.final = mul i32 %n, %k
3'. store i32 %tk.final, i32* %a
```



# Scalar Evolution Analysis

```
PreservedAnalyses run(Function &F,  
FunctionAnalysisManager &AM) {  
    ScalarEvolution &SE =  
        AM.getResult<ScalarEvolutionAnalysis>(F);  
    ...  
}
```

```
#include "llvm/Analysis/ScalarEvolution.h"
```

Dentro il file header del vostro passo

# Loop Fusion

In order for two loops,  $L_j$  and  $L_k$  to be fused, they must satisfy the following conditions:

1.  *$L_j$  and  $L_k$  must be adjacent*

- *There cannot be any statements that execute between the end of  $L_j$  and the beginning of  $L_k$*

2.  *$L_j$  and  $L_k$  must iterate the same number of times*

3.  *$L_j$  and  $L_k$  must be control flow equivalent*

- *When  $L_j$  executes  $L_k$  also executes or when  $L_k$  executes  $L_j$  also executes*

4. *There cannot be any negative distance dependencies between  $L_j$  and  $L_k$*

- *A negative distance dependence occurs between  $L_j$  and  $L_k$ ,  $L_j$  before  $L_k$ , when at iteration  $m$  from  $L_k$  uses a value that is computed by  $L_j$  at a future iteration  $m+n$  (where  $n > 0$ ).*

# Dependence Analysis

```
for (i=0; i<N; i++) {  
    A[i] = ... ;  
}  
for (i=0; i<N; i++) {  
    B[i] = A[i+3] + ...;  
}
```


Impossibile fondere i loops:

- Nel codice originale tutti i valori di A sono calcolati nel primo loop, e quindi disponibili all'esecuzione del secondo loop
- Se fondessimo i loop non avremmo a disposizione l'elemento di A richiesto nell'iterazione corrente

# Dependence Analysis

```
PreservedAnalyses run(Function &F,  
FunctionAnalysisManager &AM) {  
    DependenceInfo &DI =  
        AM.getResult<DependenceAnalysis>(F);  
    ...  
    auto dep = DI.depends(&I0, &I1, true);  
    if (!DepResult) // not dependent  
}
```

Istruzioni con  
potenziale  
dipendenza



```
#include "llvm/Analysis/DependenceAnalysis.h"
```

Dentro il file header del vostro passo

# Trasformazione del codice

- Una volta verificate tutte le condizioni per la loop fusion passo alla trasformazione del codice
- Devo fare due cose:
  1. Modificare gli usi della induction variable nel body del loop 2 con quelli della induction variable del loop 1
    - Ricorda: in SSA sono due variabili diverse
  2. Modificare il CFG perché il body del loop 2 sia agganciato a seguito del body del loop 1 nel loop 1

# Modificare gli usi delle IV

- Modificare gli usi della IV nel LOOP 2

```
15:                                     : preds = %4
  br label %19

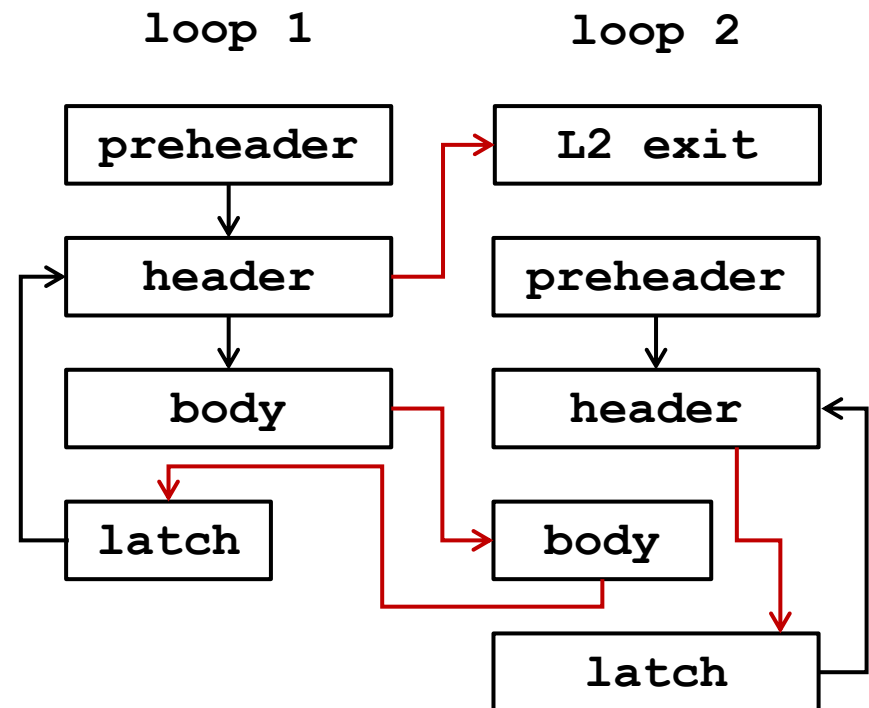
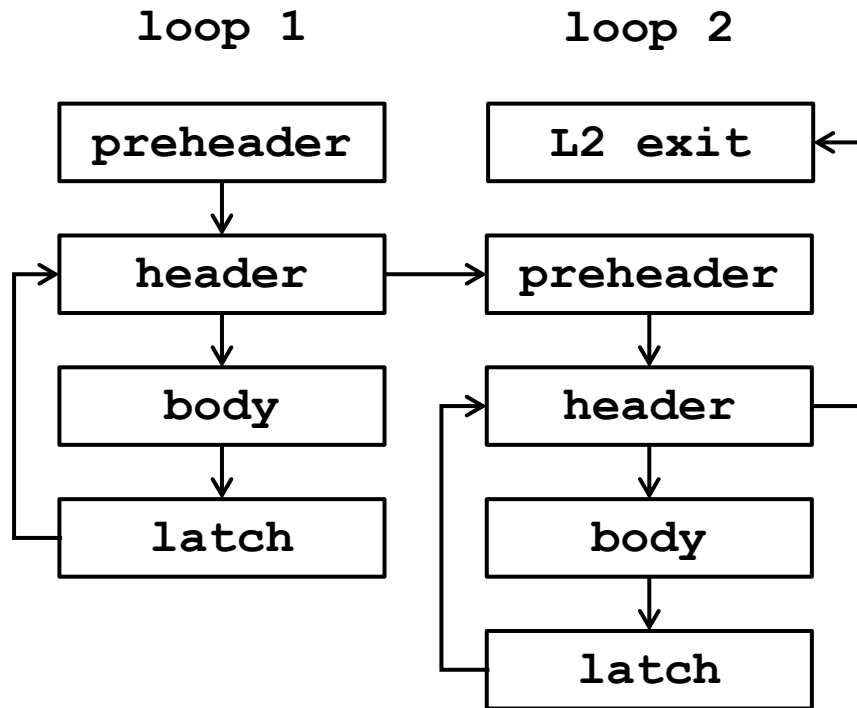
19:                                     ; preds = %31, %15
  %.1 = phi i32 [ 0, %15 ], [ %32, %31 ]
  %20 = icmp slt i32 %.1, 100
  br i1 %20, label %21, label %33

21:                                     ; preds = %19
  %22 = sext i32 %.1 to i64
  %23 = getelementptr inbounds i32, i32* %0, i64 %22
  %24 = load i32, i32* %23, align 4
  %25 = sext i32 %.1 to i64
  %26 = getelementptr inbounds i32, i32* %2, i64 %25
  %27 = load i32, i32* %26, align 4
  %28 = add nsw i32 %24, %27
  %29 = sext i32 %.1 to i64
  %30 = getelementptr inbounds i32, i32* %1, i64 %29
  store i32 %28, i32* %30, align 4
  br label %31

31:                                     ; preds = %21
  %32 = add nsw i32 %.1, 1
  br label %19, !llvm.loop !6

33:                                     ; preds = %19
  ret void
}
```

# Modificare il CFG



# Loop Fusion in LLVM

Si vedano le slides di Barton, Doefert, Finkel, Kruse

<https://llvm.org/devmtg/2018-10/slides/Barton-LoopFusion.pdf>