

Compilatori

Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 21 maggio 2025

Indice

1	Introduzione	4
1.1	Motivazione	4
1.1.1	La funzione dei compilatori	4
1.1.2	L'evoluzione dei compilatori	4
1.1.3	Eterogeneità architetturale	4
1.2	Ottimizzazione	4
1.2.1	Esempi di ottimizzazione	5
1.2.2	Ottimizzazioni sui loop	6
1.3	Anatomia di un compilatore	6
1.3.1	Flag di ottimizzazione	7
1.3.2	Uso di IR	7
1.3.3	Ingredienti dell'ottimizzazione	7
2	Rappresentazione intermedia	7
2.1	Proprietà di una IR	7
2.2	Tipi di IR	7
2.3	Categorie di IR	8
2.4	Esempi di rappresentazione	8
2.4.1	Sintassi concreta (testo)	8
2.4.2	AST (Abstract Syntax Tree)	8
2.4.3	DAG (Directed Acyclic Graph)	9
2.4.4	3AC (3-Address Code)	9
2.4.5	SSA (Static Single Assignment)	10
2.4.6	CFG (Control Flow Graph)	10
2.4.7	DG (Dependency Graph)	11
2.4.8	DDG (Data Dependency Graph)	11
2.4.9	CG (Call Graph)	12
3	Ottimizzazione locale e Local Value Numbering	12
3.1	Scope dell'ottimizzazione	12
3.2	Dead Code Elimination	12
3.2.1	Algoritmo per la DCE	12
3.3	Local Value numbering	13
4	Data Flow Analysis	14
4.1	Cos'è la DFA	14
4.1.1	Rappresentazione del programma statica o dinamica	15
4.1.2	Effetti di istruzioni e BB	15
4.2	Reaching Definitions	16
4.2.1	Schema DFA	16
4.2.2	Effetti di uno statement	17
4.2.3	Effetti di un BB	17
4.2.4	Effetti degli archi aciclici	17
4.2.5	Effetti degli archi ciclici e condizioni iniziali	18

4.2.6	Algoritmo iterativo	18
4.2.7	Algoritmo Worklist	19
4.3	Liveness Analysis	19
4.3.1	Live Variable Analysis	19
4.3.2	Forward e Backward analysis	19
4.3.3	Funzione di trasferimento	19
4.3.4	Algoritmo iterativo	20
4.4	Framework per DFA	21
4.5	Available Expressions	21
4.6	Very Busy Expressions (Assignment 2)	22
4.7	Dominator Analysis (Assignment 2)	22
4.8	Constant Propagation (Assignment 2)	22
5	Loops e UD-DU chains	23
5.1	Cos'è un loop	23
5.1.1	Definizioni formali	23
5.1.2	Loop naturali	24
5.2	Identificare i loop naturali	24
5.2.1	Trovare i dominatori	24
5.2.2	Trovare i back edges	24
5.2.3	Trovare il loop naturale associato	25
5.3	Preheader	25
5.4	Use-def e Def-use chains	25
5.4.1	Dove viene definita o usata una variabile	25
6	Static Single Assignment (SSA)	26
6.1	La funzione Φ	27
6.2	SSA triviale	27
6.3	SSA minimale	27
6.4	Dove inserire le funzioni Φ	28
6.5	Proprietà di dominanza della forma SSA	28
6.6	Dominanza e Dominance Frontier	28
6.7	Utilizzo della Dominance Frontier per calcolare la forma SSA	29
6.7.1	Utilizzo della DF per posizionare le funzioni $\Phi()$	29
6.7.2	Rinominare le variabili	30
7	Ottimizzazioni sulla memoria	31
7.1	Memorie, tecnologie RAM	31
7.1.1	Tecnologia DRAM	31
7.1.2	DRAM performance factors	31
7.1.3	Memory wall	31
7.2	Cache	32
7.2.1	Gerarchie di memoria	32
7.2.2	Principi di località	32
7.2.3	Cache misses	33
7.2.4	Tipologie di cache	33
7.3	Memory Hierarchy optimizations	33
7.4	Reuse and locality	34
7.5	Optimizing cache performance	34
7.5.1	Il ruolo del compilatore	34
7.6	Tipi di oggetti da considerare	34
7.6.1	Scalari	34
7.6.2	Strutture dati e puntatori	35
7.6.3	Array	35
7.7	Iteration space	35
7.8	Cache behavior of array accesses	36
7.8.1	Approcci per variabili di cui non conosco il valore staticamente	36
7.9	Esempi di trasformazioni per i loop	36
7.9.1	Loop interchange	36

7.9.2	Loop tiling/blocking	37
7.10	Loop fusion	38
7.11	Software prefetching	38
7.11.1	Coping with memory latency	38
7.11.2	Tollerare la latenza con il prefetching	38
7.11.3	Algoritmo di prefetching del compilatore	39

1 Introduzione

1.1 Motivazione

Ricordiamo il ruolo del compilatore tra le tecnologie informatiche, quello dell'ISA e del linguaggio assembly, i passaggi gestiti dal compilatore, dall'assembler, eccetera

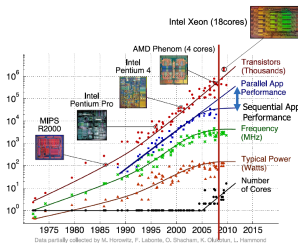
- Il compilatore **traduce un programma sorgente in linguaggio macchina**
- L'ISA agisce da "interfaccia" tra HW e SW (fornisce a SW il set di istruzioni, e specifica a HW che cosa fanno)

1.1.1 La funzione dei compilatori

- Funzione principale e più nota: trasformare il codice **da un linguaggio ad un altro** (es. C → Assembly RISC-V) (ricordiamo che è solo il primo passo di un'intera toolchain di programmi per creare eseguibili)
- Gestendo la traduzione a linguaggio macchina al posto dei programmatori, l'altra funzione importante è l'**ottimizzazione** del codice, che permette la **produzione di eseguibili di stesse funzionalità**, ma diversi a livello di **dimensioni** (es. per sistemi embedded e high-performance), **consumo energetico**, **velocità di esecuzione**, ma anche in termini di determinate **caratteristiche architetturali** utilizzate (es. proc. multicore)

1.1.2 L'evoluzione dei compilatori

Le rivoluzioni in termini di "classe" di dispositivi e di dimensioni dei transistor sono molto frequenti (Bell, Moore), e nei primi 2000 si arriva ai **limiti fisici della miniaturizzazione e della frequenza operativa** dei processori (problemi di dissipazione del calore) → idea di cambiare il paradigma di sviluppo di un processore: dal singolo core sempre più potente passo a **più core "isopotenti"** sullo stesso chip



~ 2005: plateau di consumo, frequenza e performance di programmi *sequenziali*, aumento di performance di p. che **sfruttano la parallelizzazione** → i programmi devono essere "consapevoli" che il processore è multicore!

Il compilatore mantiene un ruolo fondamentale: oltre a rendere meno "traumatico" il passaggio alla programmazione parallela, (non sono ancora auto-parallelizzanti) si interfaccia con i nuovi paradigmi di programmazione parallela offerti ai programmatori: il programmatore sfrutta interfacce semplici e astratte, mentre il compilatore traduce i

costrutti in codice parallelo eseguibile (es. OpenMP)

1.1.3 Eterogeneità architetturale

La programmazione parallela e il parallelismo architetturale sono oggi paradigmi consolidati, e i processori general purpose (seppur multicore e ottimizzati) non sono sufficienti per attività specializzate come la grafica → nascono componenti **acceleratori** di vario tipo: GPU, GPGPU, FPGA, TPU, NPU... Questo complica ulteriormente la scrittura del software, e dunque impone altre evoluzioni nei compilatori e nelle ottimizzazioni.

1.2 Ottimizzazione

Ricordiamo le metriche usate:

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

$$\text{Execution Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Frequency}}$$

Le ottimizzazioni possono avvenire dal punto di vista **HW (parametri architetturali)** e da quello **SW (p. di programma)**. Il compilatore può agire anche ad es. a livello di cache, aiutando a ridurre i miss e dunque i CPI delle istruzioni **load** e **store** (sappiamo che il costo di accesso aumenta di ordini di grandezza)

1.2.1 Esempi di ottimizzazione

Distinguiamo le ottimizzazioni che avvengono a compile time o a runtime (statiche o dinamiche)

- **AS (Algebraic Simplification)**: ottimizzazione a runtime

```
-(-i); → i;  
b or true; → true; //cortocircuito logico
```

- **CF (Constant Folding)**: valutare ed espandere espressioni costanti a compile time

```
c = 1+3; → c = 4;  
(100<0) → false
```

- **SR (Strength Reduction)**: sostituisco op. costose con altre più semplici: classico es. MUL rimpiazzate da ADD/SHIFT (esecuzione in 1 ciclo invece di multic.):

```
y = x*2;  
y = x * 17;
```

→

```
y = x+x;  
y = (x<<4) + x;
```

es. sofisticato: **for** con operazioni su array, sostituito da operazioni su puntatori (aritmetica dei pt.) → il risultato si vede nel codice assembly

```
for (i=0; i<100; i++)  
  a[i] = i*100;
```

→

```
t = 0;  
for (; t<10000; t += 100) {  
  *a = t;  
  a = a + 4;  
}
```

```
li s0, 0 // i = 0  
li s1, 100  
LOOP:  
bge s0, s1, EXIT  
slli s2, s0, 2  
add s2, s2, a0  
mul s3, s0, 100  
sw s3, 0(s2)  
addi s0, s0, 1  
jal zero, LOOP  
EXIT:
```

→

```
li s0, 0 // t = 0  
li s1, 10000  
LOOP:  
bge s0, s1, EXIT  
sw s0, 0(a0)  
addi a0, a0, 4  
jal zero, LOOP  
EXIT:
```

- **CSE (Common Subexpression Elimination)**: elimino i calcoli ridondanti di una stessa espressione riutilizzata in più istruzioni (statement)

```
y = b * c + 4  
z = b * c - 1
```

→

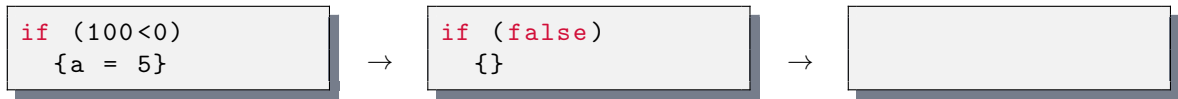
```
x = b * c  
y = x + 4  
z = x - 1
```

- **DCE (Dead Code Elimination)**: elimino tutte le istruzioni che producono codice mai letto (e dunque utilizzato), es. variabili assegnate e mai lette, codice irraggiungibile → uno dei passi eseguiti più di frequente durante l'ottimizzazione del codice da parte del compilatore, per rimuovere anche tutto il dead code generato dagli altri passi di ottimizzazione

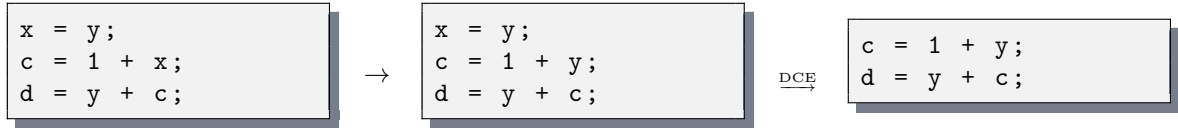
```
b = 3  
c = 1 + 3  
d = 3 + c
```

→

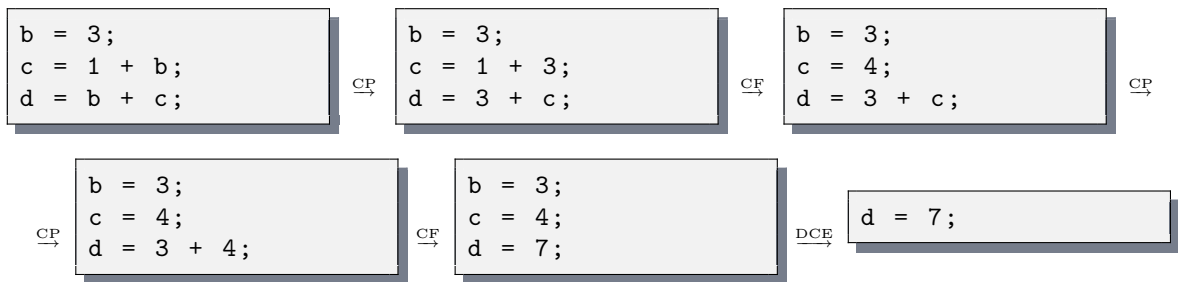
```
c = 1 + 3  
d = 3 + c
```



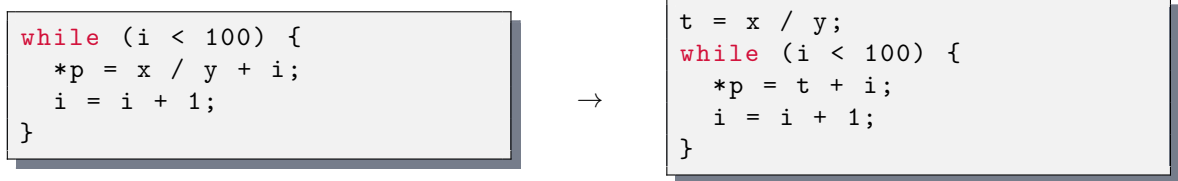
- **Copy Propagation:** per uno statement $x = y$, sostituisco gli usi futuri di x con y se non sono cambiati nel frattempo (propedeutico alla DCE)



- **CP (Constant Propagation):** sostituisco usi futuri di una variabile con assegnato valore costante con la costante stessa (se la variabile non cambia) (sempre ipotesi che i valori a fine es. siano poi usati, e non dead code)



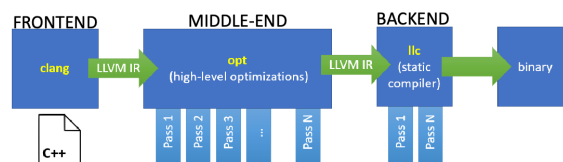
- **LICM (Loop Invariant Code Motion):** si occupa di muovere fuori dai loop tutto il codice **loop invariant**; evita i calcoli ridondanti



1.2.2 Ottimizzazioni sui loop

- grande impatto sulla performance dell'intero programma (per ovvie ragioni)
- spesso sono ottimizzazioni propedeutiche a quelle machine-specific (effettuate nel backend): register allocation, instruction level parallelism, data parallelism, data-cache locality

1.3 Anatomia di un compilatore



- almeno due compiti: **analisi del sorgente** e **sintesi di un programma in linguaggio macchina**, operando su una IR che si interpone tra frontend e backend, e tra source code e target code
- Il blocco di middle-end agisce su IR, e in vari passaggi lo trasforma e ottimizza (\neq a seconda del compilatore)
- caso llvm: clang (frontend) \rightarrow opt (middleend) \rightarrow llc (backend)

- **opt** si basa su una serie di **passi di ottimizzazione (o di analisi)**: un passo di analisi scorre l'IR e lo analizza (non lo trasforma, ma produce informazioni utili); un passo di ottimizzazione sfrutta informazioni conosciute per trasformare l'IR (applica le ottimizzazioni)
- alcune ottimizzazioni non possono essere effettuate o finalizzate senza conoscere l'architettura target (es. sulle cache), e dunque vengono eseguite dal backend

1.3.1 Flag di ottimizzazione

sono flag che passo al compilatore (al pass manager) per influenzare **ordine e numero dei passi di ottimizzazione**

- **-g**: solo debugging, nessuna ottimizzazione
- **-O0**: nessuna ottimizzazione
- **-O1**: solo ott. semplici
- **-O2**: ott. più aggressive
- **-O3**: ordine dei passi che sfrutta compromessi tra velocità e spazio occupato
- **-OS**: ottimizza per dimensione del compilato

1.3.2 Uso di IR

un backend che fa uso di IR permette di disaccoppiare con facilità frontend e backend, lavorare su ottimizzazioni machine-independent, semplificare il supporto per molti linguaggi, eccetera

Per supportare un nuovo linguaggio o una nuova architettura, basta scrivere un nuovo front/backend - il middle-end può rimanere lo stesso!

1.3.3 Ingredienti dell'ottimizzazione

- **formulare un problema di ottimizzazione** con molti casi di applicazione, sufficientemente efficiente e impattante su parti significative
- **rappresentazione** che astrae dettagli rilevanti → **analisi** di applicabilità → **trasformazione del codice** → **testing** → ☺

2 Rappresentazione intermedia

Ricordiamo: middle end come sequenza di passi, di analisi o di trasformazione → per analizzare e trasformare il codice occorre una rapp. intermedia (IR) **espressiva** che **mantenga le informazioni importanti da un passo all'altro**

2.1 Proprietà di una IR

scegliamo IR diverse a seconda del loro uso, in generale alcune caratteristiche sono sempre richieste:

- facilità di **generazione** (effetti sul frontend)
- facilità e costo di **manipolazione**
- livello di astrazione e di **dettaglio esposto**: effetti su frontend e backend (\neq IR da un lato e dall'altro, a seconda di astraz. e dettaglio necessari)

2.2 Tipi di IR

- AST (abstract syntax tree)
- DAG (grafi diretti aciclici)
- 3AC (3-address code): simile all'assembly (3 indirizzi: registro destinazione e max 2 operandi)
- SSA (Static Single Assignment): evoluzione di 3ac con ulteriori proprietà di control flow
- CFG (control flow graphs): rappresenta "come" vengono chiamate le funzioni (a partire dal main)
- CG (call graph)
- PDG (program dependence graphs): fondamentale per lavorare sul parallelismo, multithreading...

Le ott. inter-procedurali devono per forza basarsi su IR di tipo CG (es. per decidere quando fare **inlining** - espandere il codice della funzione invece di chiamarla - evidente tradeoff tra dimensione del codice e overhead dovuto alla chiamata di funzione)

2.3 Categorie di IR

- grafiche (o strutturali)
 - orientate ai grafi
 - molto usate nella source-to-source translation, tipicam. per ott. che non hanno bisogno della struttura sofisticata di un middle-end
es. openMP: di fatto annotazioni sul codice, come strumento semplice per la parall. (es. **outlining**: prendo es un loop e lo impacchetto in una funzione che poi dovrà essere eseguita dai thread per la parallelizzazione) - non sto ottimizzando nel senso proprio del termine, ma sto trasformando il codice e lo sto rendendo eseguibile in maniera parallela
 - solitamente voluminose (basate su grafi) - tradeoff con il fatto che non coinvolgono il middle-end
 - es. AST, DAG
- lineari
 - pseudocodice per macchine astratte
 - livello di astrazione vario
 - strutture dati semplici e compatte
 - facile da riarrangiare (evidentemente il più comodo per eseguire le ottimizzazioni)
 - es. 3AC
- ibride (sfruttano combinazioni delle prime due) (es. CFG)

2.4 Esempi di rappresentazione

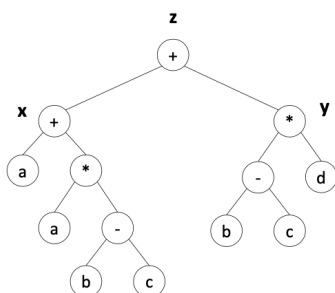
2.4.1 Sintassi concreta (testo)

Più semplice in quanto più vicina al livello di astrazione "umano" di ragionamento sul programma, ma non il livello corretto per ottimizzare né comprendere correttamente la semantica del programma

```
let value = 8;  
let result = 1;  
for (let i = value; i>0; i = i - 1) {  
  result = result * i;  
}  
console.log(result);
```

2.4.2 AST (Abstract Syntax Tree)

Albero i cui nodi rappresentano diverse parti del programma: il nodo radice rappresenta il **programma**, il quale a sua volta contiene un blocco di istruzioni dal quale discendono tanti figli quante le sue istruzioni



```
x = a + a * (b - c)
y = (b - c) * d
z = x + y
```

PRO: molto comodo per interpreti (basta usare una fz. ricorsiva per processare l'albero)

CONTRO: un nodo è un oggetto troppo generico → analizzare un ast per l'ottimizzazione impone ogni volta di ragionare sulla differenza semantica tra i nodi (complica molto)

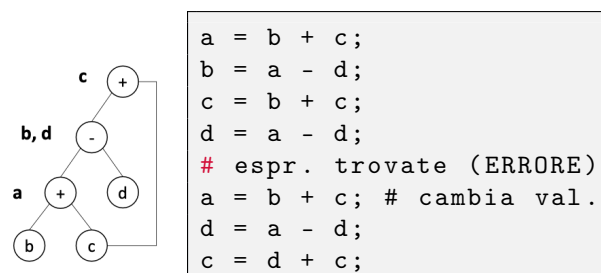
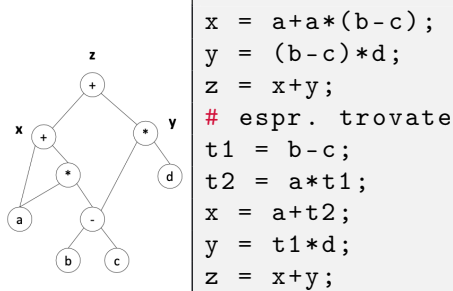
2.4.3 DAG (Directed Acyclic Graph)

Contrazione di ast che evita la duplicazione di espressioni → **rappresentazione più compatta**

Limite: il riuso è possibile solamente dimostrando che il suo **valore non cambia** nel programma

essendo assegnamenti e chiamate frequentissimi, il fatto che il dag non abbia nozione di come le espr. cambino valore nel tempo non lo rende un buon candidato per le ottimizzazioni

Esempi



2.4.4 3AC (3-Address Code)

Evidentemente adatto: tutte le istr. del programma vengono spezzettate in istr. di forma semplice simile all'assembly, di tipo $x = y \text{ op } z$ (1 operatore, massimo 3 operandi)

$x = 2 * y$ → $t1 = 2 * y$
 $t2 = x - t1$

assignments	$x = y \text{ op } z$
	$x = \text{op } y$
	$x = y[i]$
	$x = y$
branches	goto L
conditional branches	if x relop y goto L
procedure calls	param x
	param y
	call p
address and pointer assignments	$x = \&y$
	$*y = z$

PRO:

- espressioni complesse spezzettate
- forma compatta e simil-assembly
- registri temporanei **intermedi, virtuali e illimitati** (tralascio problemi architetturali - n. di r. fisici a disposizione e eventuali op. di spill, cioè aggiungere load o store in mancanza di r. fisici)

Varianti di 3AC

A seconda dei vincoli che ho per l'implementazione pratica:

- **quadruple:** id istruzione, opcode, i 3 registri → semplice struttura record, facile da analizzare e riordinare ma i nomi espliciti prendono più spazio
- **triple:** id istruzione, opcode, 2 operandi → uso l'indice dell'espressione nell'array come "nome" del registro destinazione → risparmio spazio, ma diventa più complesso da analizzare (nomi impliciti) e riordinare

Quadruple

x - 2 * y				
(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

Triple

x - 2 * y				
(1)	load	y		
(2)	loadi	2		
(3)	mult	(1)	(2)	
(4)	load	x		
(5)	sub	(4)	(3)	

Inapplicabilità diretta della Constant Propagation con forma 3AC

La CP è applicabile solo se la variabile **se non cambia nel frattempo** → una IR di tipo 3AC non può applicarla immediatamente (devo prima analizzare il resto del codice)

2.4.5 SSA (Static Single Assignment)

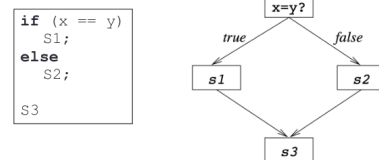
- Evoluzione di 3AC che impone che la **definizione (assegnamento)** delle variabili avvenga **solo una volta** (def. multiple sono tradotte in multiple versioni della var)
- **PRO**: ogni definizione ha associata direttamente una **lista di tutti i suoi usi** - semplifica enormemente le ottimizzazioni di tipo CP e non solo

Quasi sempre uno dei passi di ottimizzazione prevede il passaggio a forma SSA

La scelta della IR dipende ovviamente dal livello di dettaglio necessario per ogni specifico compito
 → **in un compilatore coesistono più IR** (anche per questo esistono forme ibride)

2.4.6 CFG (Control Flow Graph)

- modella il trasferimento (flusso) del controllo in un programma tra **blocchi** di istruzioni
- permette di aggiungere informazioni sui **salti** al di sopra di una IR lineare
- i suoi nodi sono Basic Block
- gli archi rappresentano il flusso di controllo del programma (loop, condizioni, ecc.)



- un BB è una seq. di istruzioni in forma 3AC
 - singolo *entry point*: solo la prima istruzione può essere raggiunta dall'esterno
 - singolo *exit point*: se eseguo la prima istr. **devo eseguire tutte le altre** - garantisco che venga **eseguito interamente**
 - Le chiamiamo sezioni single-entry, single-exit (possono essere sezioni anche più grandi, ma le più piccole di questo tipo sono i BB)
- un arco connette due nodi $B_i \rightarrow B_j \iff b_j$ può eseguire dopo B_i in qualche percorso del ctrl flow del programma
 - prima istr. di B_j è target dell'istr. di salto al termine di B_i
 - $\forall B_i$ non ha un istr. di salto come ultima istr. (nodo *fallthrough*) e B_j è suo unico successore
- un CFG **normalizzato** ha i BB **massimali**
 - non possono essere resi più grandi senza violare condizioni
 - unisco i BB fallthrough che non hanno label all'inizio
 - posso avere CFG non norm. dopo qualche generico passo di ottimizzazione (non le facciamo accadere "spontaneamente")

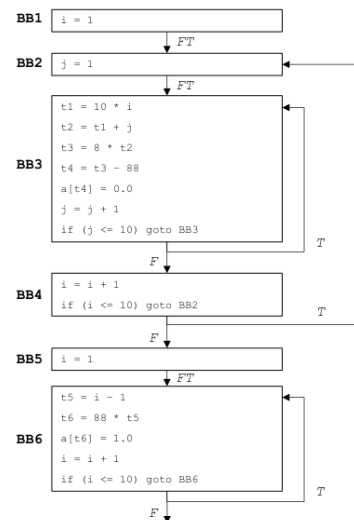


Fig. 2.1: Esempio di CFG

Algoritmo per la costruzione del CFG

1. identificare il **leader** di ogni BB:
 - la prima istruzione
 - il target di un salto
 - ogni istruzione dopo un salto
2. il BB **comincia** con il leader e **termina** con l'istruzione immediatamente precedente un nuovo leader (o l'ultima istruzione)
3. **connettere** i BB tramite archi di 3 tipi:
 - **fallthrough** (o fallthru): esiste solo un percorso che collega i due blocchi
 - **true**: il secondo blocco è raggiungibile dal primo se un condizionale è **true**
 - **false**: il secondo blocco è raggiungibile dal primo se un condizionale è **false**

2.4.7 DG (Dependency Graph)

I nodi di un DG sono istruzioni; un arco connette due nodi di cui **uno usa il valore definito dall'altro**. Sono indispensabili per l'*instruction scheduling* e per mantenere il CPI della pipeline (ricordiamo quella RISC-V):

↷ IF Instruction Fetch
↓ ID Instruction Decode
↓ EXE Execute
↓ MEM Memory Access
↶ WB Write Back

Instruction / Cycle Count	1	2	3	4	5	6
add x2, x3, x4		F	D	E	M	W
sub x5, x2, x6			F	D	E	M

Fig. 2.2: Esempio di data hazard

Esempio: risultato di una add

Se in fase di decode provo a leggere il registro usato in una **add** immediatamente precedente, questo ancora non contiene il risultato aggiornato (pronto appena tra 2 cicli) → **data hazard**, gestito solitamente dalla **forwarding unit** che bypassa MEM e WB e inoltra direttamente il dato

Soluzione generica (inefficiente)

Per quanto i controlli vengano svolti dalla fw. unit, in generale l'unico modo per evitare questo tipo di hazard è distanziare le istruzioni tra loro affinché il dato sia disponibile → inserisco nop (cicli di stallo), ma vado a "rompere" l'IPC pari a 1 della pipeline sempre piena (< performance)

Soluzione migliore

scheduling del programma, spostando istruzioni che non dipendono da quei registri al posto di aggiungere nop → uno dei compiti principali di un backend, che per evitare di cercare le istruzioni libere "manualmente" sfrutta la IR di tipo DG che fornisce esattamente le informazioni sulle dipendenze tra istruzioni

2.4.8 DDG (Data Dependency Graph)

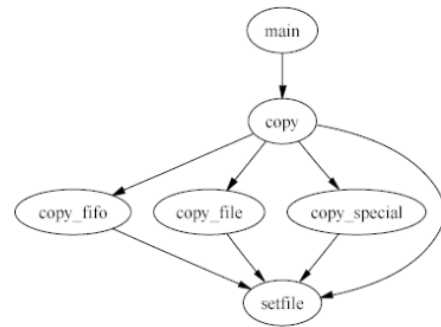
- specifico per multicore e parallelismo, usato per dare una rappresentazione tra le dipendenze dei **dati** - tipicamente i loop, a patto che non ci siano dipendenze di dato tra le varie iterazioni.
- tipicamente uso il **polyhedral model** → rappresento lo spazio delle it. come un poliedro (a seconda del numero di loop innestati), che permette di capire se esiste qualche permutazione dei loop (direzione di attraversamento dello spazio delle iterazioni; ovvero ad esempio scambiare l'ordine dei loop) **non soggetta a dipendenze**

2.4.9 CG (Call Graph)

Rappresentazione gerarchica a grafo usata per ragionare sull'insieme delle potenziali chiamate tra funzioni della translation unit del sorgente

Nota

Il compilatore ha visibilità solo fino a livello dei singoli moduli: posso estendere le ottimizzazioni al massimo fino ai legami tra funzioni dello stesso modulo, quelle più ampie si spostano a framework di ott. che agiscono es. a livello di linker

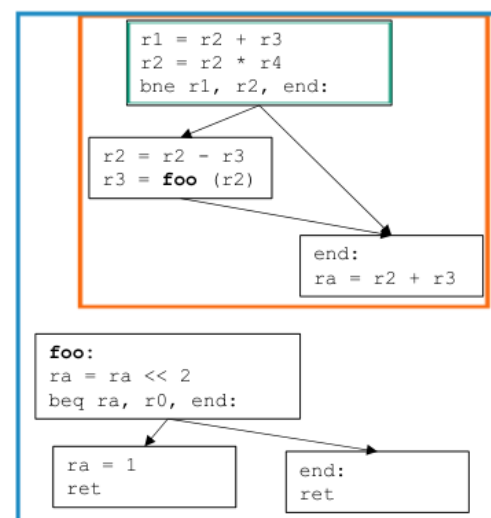


3 Ottimizzazione locale e Local Value Numbering

3.1 Scope dell'ottimizzazione

Lo scope viene influenzato da come viene gestito il flusso di controllo in un programma

- ott. **locale**: entro un singolo BB, non si preoccupa del flusso
- ott. **globale**: lavora a livello dell'intero CFG
- ott. **interprocedurale**: lavora a livello del call graph, e quindi sui CFG di più funzioni



3.2 Dead Code Elimination

Cominciamo ragionando su ott. locali come la DCE (sono *dead code* le istr. che definiscono una variabile mai utilizzata)

Sicuramente va tolta la def. di c, ma poiché **print non definisce nulla**, stando alla def. è dead code! → Estendiamo la definizione dicendo che lo sono le istr **prive di side effects** che definiscono ...

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
    print d;  
}
```

3.2.1 Algoritmo per la DCE

- ∀ istruzione in BB
 - aggiungi operandi ad un metadato array **used**
- ∀ istruzione in BB
 - se non ha destinazione e non ha side effects rimuovila
 - altrimenti se la destinazione non corrisponde a nessuno degli elem di **used** rimuovila

```

used = {};

for instr in BB:
    used += instr.args;

for instr in BB:
    if instr.dest &&
        instr.dest not in used:
    delete instr

```

A questo punto rendiamolo iterativo, per farlo eseguire fino a "convergenza" (elimino tutto il *d-c*)

```

while prog changed:
{
    ... #alg
}

```

```

main {
    int a = 100;
    int a = 42;
    print a;
}

```

→ Consideriamo questo esempio in cui si ridefinisce una variabile: il nostro algoritmo corrente non elimina nulla perché non gestisce le **dead stores**

→ per estenderlo dobbiamo poter **rilevare gli assegnamenti multipli** di una variabile, e quindi anche **preoccuparci dell'ordine** delle istruzioni! (più complicato senza forma SSA)

- dopo ogni istr. teniamo traccia delle variabili definite, ma non usate
- se troviamo un altro assegnamento alla stessa variabile prima della fine del blocco sappiamo che quello precedente può essere eliminato

```

last_def = {}; /* lista di variabili definite ma non usate (ptr alla piu
    ' recente per le sole variabili mai usate) */

for instr in BB:
    last_def -= instr.args; /*rimuovo ogni argomento (operando) dell'
        istruzione corrente: se presente, e' un uso della variabile */

    if instr.dest in last_def:
        delete last_def[instr.dest]; /*se a questo punto la destinazione
            dell'istr. corrente e' presente in last_def posso sovrascrivere la
            definizione precedente*/
    last_def[instr.dest]=instr;

```

- nota: per situazioni tipo $x=2$; $x=x+3$; si vuole evitare che la seconda istr. venga eliminata perché non ho realizzato che x è usata (non è *d-c*) → per questo controllo prima gli usi e poi le definizioni
- anche questo algoritmo va ripetuto fino a convergenza

3.3 Local Value numbering

Tecnica utilizzata per considerare il concetto di ordine delle definizioni in assenza di proprietà di tipo SSA

osserviamo 3 pattern che forniscono opportunità di eliminazione di codice ridondante:

- dead code elimination: 1 variabile e più valori
- copy propagation: 1 valore e più variabili

- common subexpression elimination: 1 valore (in forma di espressione) e più variabili

sono tutti modelli di computazione che si focalizzano sulle **variabili** → focalizzandoci sui **valori** possiamo eliminare tutte le forme di ridondanza

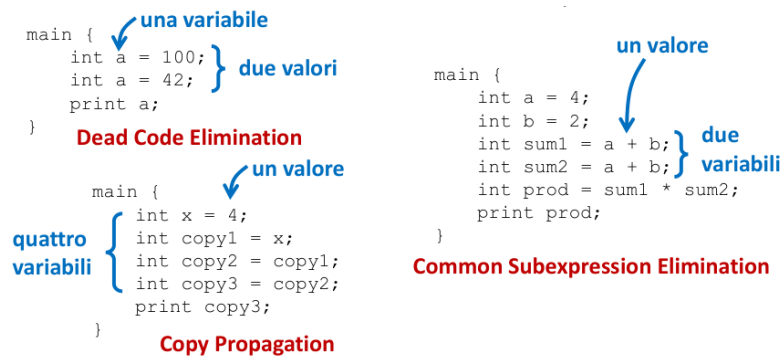
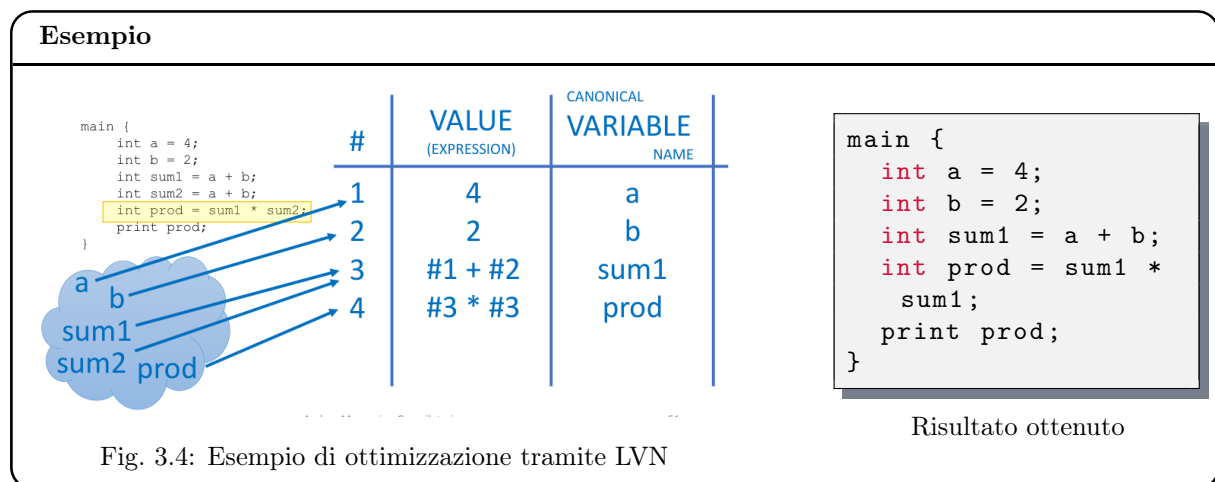


Fig. 3.3: Esempi di codice legati ai pattern citati

- costruisco un metadato in forma di tabella che riscrive le espr. (istruzioni) in funzione dei valori già osservati → evitando di riassegnare lo stesso valore a più variabili si evita la ridondanza
- analizzo le istruzioni in termini del value, e in caso questo coincida con entry della tabella già presenti punto direttamente a quella



```
# ...
int sum1 = a + b;
int sum2 = b + a;
```

Semplice variante del programma non riconosciuta dall'algoritmo (non conosce la pr. commutativa della somma) → vado a **canonica-**
→ **lizzare** l'algoritmo, ovvero imporre un ordine numerico tra le entry (i valori) e usarle sempre in ordine crescente per le op. commutative (di fatto svolto da tutti i compilatori)

4 Data Flow Analysis

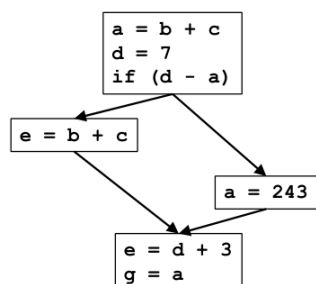
4.1 Cos'è la DFA

La possiamo vedere come una **metodologia** o come un **framework** di analisi, applicabile a vari problemi di ottimizzazione (CP, CSE, DCE, ...)

- analisi **locale**: si focalizza sull'effetto di ogni istruzione → posso comporre gli effetti di tutte le istr. per derivare informazione dall'inizio del BB ad ogni istruzione
- analisi **globale** (DFA): simile, ma molto più complessa → analizza l'effetto di ogni BB, e poi ha una metodologia per comporre l'effetto dei BB ai **confini** degli stessi per derivare informazione

La DFA è **sensibile al flusso di controllo in una funzione** e prevede un'analisi **intraprocedurale** (singola funzione, singolo CFG)

Esempio



La DFA ci permette di derivare informazioni da un CFG, ad esempio \forall variabile x in ogni punto del grafo:

- qual è il suo valore?
- quale "definizione" la definisce?
- la definizione è ancora "valida" (*live*)?

Queste risposte normalmente le otteniamo grazie alla forma SSA delle istruzioni e alla struttura di LLVM, ma qui stiamo ancora "costruendo" la forma SSA

4.1.1 Rappresentazione del programma statica o dinamica

- **statica**: programma finito, un pezzo di codice \rightarrow molto facile da analizzare
- **dinamica**: può avere infiniti percorsi di esecuzione, rappresenta una possibile esecuzione reale!
 - es. loop che si basa sull'analisi di un input

condizioni che un compilatore **non può analizzare**, soprattutto non in maniera statica e *finita* in termini di possibili istanze \rightarrow spesso si passano al comp. informazioni di "profiling" misurate durante ripetute esecuzioni del programma pre-ottimizzato

\rightarrow Con la DFA siamo in grado di dire qualcosa **per ogni punto del programma**, combinando informazioni relative a **tutte le possibili istanze** dello stesso p.to

4.1.2 Effetti di istruzioni e BB

Effetti di un'istruzione: $(a = b + c;)$

- **uses**: delle variabili (b, c)
- **kills**: una precedente definizione (a)
- **defines**: una variabile (a)

Combinando gli effetti delle singole istr. si definiscono gli **effetti di un BB**:

- **uso localmente esposto** (*locally exposed use*): in un BB è un uso di una var. che non è preceduto nel BB da una definizione della stessa variabile
- ogni definizione di una var. nel BB **kill**a tutte le definizioni della stessa variabile che **raggiungono** il BB
- **definizione localmente disponibile** (*locally available definition*): ultima definizione di una variabile nel BB

Esempio

```

t1 = r1+r2      1
r2 = t1         2
t2 = r2+r1      3
r1 = t2         4
t3 = r1*r1      5
r2 = t3         6
if r2>100 goto L1 7

```

- usi loc. esposti: 1 (r1,r2), 3 (r2), ...
- kill: 2 (r2), ...
- definizioni loc. disponibili: 6 (r2), 5 (t3), ...

4.2 Reaching Definitions

Primo esempio di problema inquadrabile e risolubile mediante DFA

- ogni istruzione di assegnamento è una definizione
- una definizione d raggiunge (reaches) un punto itp se esiste un percorso da d a p tale per cui d non è uccisa (sovrascritta) lungo quel percorso

Definizione del problema:

- determinare **per ogni punto** del programma se **ogni definizione** nel programma **raggiunge** quel punto

→ usiamo un **bit vector** per ogni istruzione, con lunghezza del vettore pari al numero di definizioni
 ⇒ diventa una sorta di matrice con righe le istruzioni, colonne le definizioni

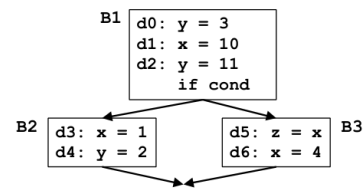
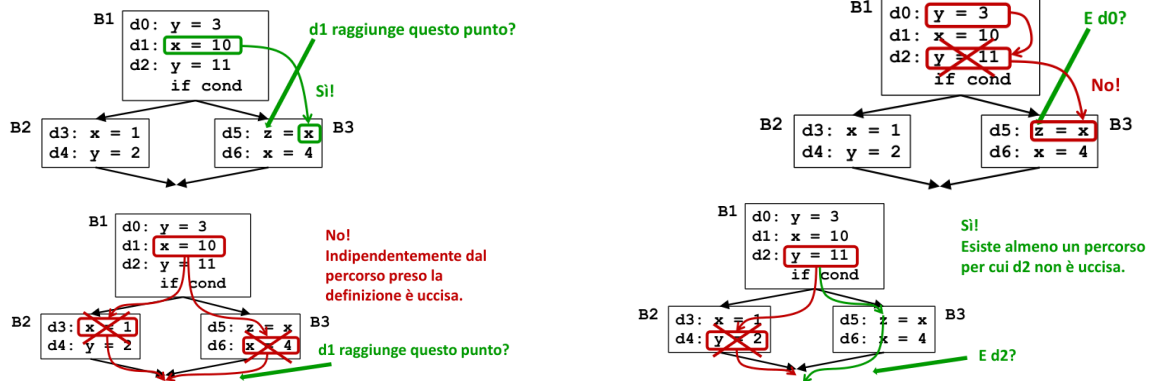


Fig. 4.5: Esempio

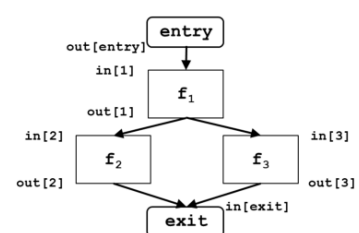
Esempio



4.2.1 Schema DFA

Consideriamo un flow graph:

- aggiungiamo un **BB entry** e uno **exit** → garantisco *single-entry* e *single-exit* points
- per stabilire l'effetto del codice in **ciascun BB** uso delle **funzioni di trasferimento** f_B , che correlano $out[B]$, $in[B]$ per un dato BB B
- stabilisco l'effetto **del flusso di controllo** in base alla vicinanza dei blocchi: correla $out[B_i]$, $in[B_j]$ di blocchi adiacenti
- infine dobbiamo solo risolvere le equazioni



Nota: stabiliamo anche una cosiddetta **boundary condition**, ovvero qual è l'informazione che riceve il primo BB dall'*entry block* → per le Reaching Definitions stabiliamo $out[entry] = \emptyset$

4.2.2 Effetti di uno statement

- la fz. di trasferimento di uno statement astrae l'esecuzione rispetto al problema di interesse
- per uno statement s ($d: x = y + z$):
 - $Gen[s]$: definizioni **generate** ($Gen[s] = \{d\}$)
 - definizioni **propagate**: $in[s] - Kill[s]$ dove $Kill[s]$ sono le altre definizioni di x nel resto del programma
 - funzione di trasferimento di uno statement s :

$$out[s] = f_s(in[s]) = Gen[s] \cup (in[s] - Kill[s])$$

- in altre parole, Gen è l'insieme di proprietà **generate** dal blocco, $Kill$ l'insieme di quelle **eliminate** nel blocco e in l'insieme di quelle **ereditate**

Stiamo lavorando su bit vector → una f_s riempie $out[s]$ a partire da $in[s]$ e applicando qualche tipo di calcolo

4.2.3 Effetti di un BB

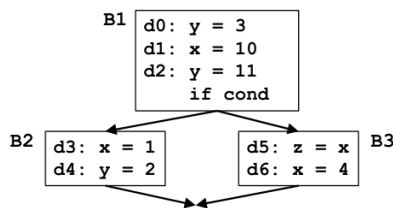
- la fz. di trasferimento di un BB compone linearmente le fz. dei suoi *statements*

$$\begin{aligned} out[B] &= f_B(in[B]) = f_{d_n} \dots f_{d_0} \\ &= Gen[B] \cup (in[B] - Kill[B]) \end{aligned}$$

- $Gen[B]$: definizioni localmente disponibili (**a fine BB**)
- $Kill[B]$: definizioni uccise da B **in tutto il programma**

- una f_B associa **incoming reaching definitions** → **outgoing reaching definitions**

Esempio



	Gen	Kill
B ₁	1, 2	0, 2, 3, 4, 6
B ₂	3, 4	0, 1, 2, 6
B ₃	5, 6	1, 3

NB: in $Kill[B1]$ d0 e d2 si uccidono a vicenda → l'approccio bit-vector **non ha nozione di ordine di esecuzione**

4.2.4 Effetti degli archi aciclici

In caso di predecessori multipli, devo decidere con che criterio unire l'informazione:

- nodo di unione (**join**): nodo con **predecessori** multipli
- operatore di **meet** (\wedge): $in[B] = out[p_1] \cup \dots \cup out[p_n]$ con p_1, \dots, p_n tutti predecessori di B

Il meet operator è **specifico del problema**, non dell'analisi mediante DFA! → Per il problema delle reaching definitions, usiamo $\wedge = \cup$ in quanto la condizione deve essere verificata per *almeno un percorso*

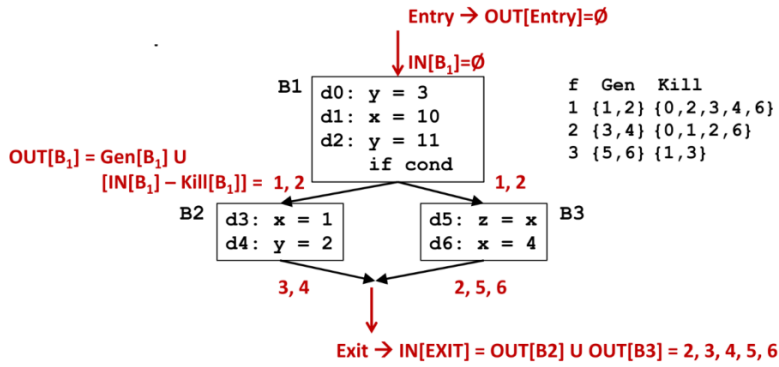


Fig. 4.6: Esempio (continua)

4.2.5 Effetti degli archi ciclici e condizioni iniziali

Gli archi ciclici (*backedges*) possono cambiare le loro *out* o non averle ancora calcolate durante l'esecuzione dell'algoritmo:

- itero fino a convergenza
- definisco delle **condizioni iniziali** per inizializzare ogni BB (similmente alla boundary condition): stabilisco $out[B] = \emptyset$ (specifico per Reaching Defs.)

4.2.6 Algoritmo iterativo

```

input: control flow graph CFG = (N, E, Entry, Exit)
// Boundary condition
out[Entry] = ∅

// Initialization for iterative algorithm
for each basic block B other than Entry
    out[B] = ∅

// Iterate
while (changes to any out[] occur) {
    in[B] = ∪ (out[p]), for all predecessors p of B
    out[B] = fB(in[B]) // out[B] = Gen[B] ∪ (in[B] - Kill[B])
}

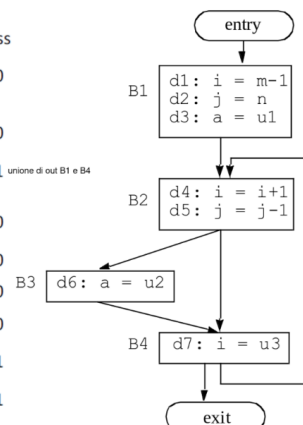
```

Esempio di esecuzione

La rappresentazione del bit-vector segue l'ordine di numerazione delle definizioni:
 $IN[B1] = \langle 000 \ 00 \ 0 \ 0 \rangle$ con d1 primo bit, ...

Nota: dopo la seconda iterazione $out[B2]$ non cambia più

	First Pass	Second Pass
IN[B1]	000 00 0 0	000 00 0 0
OUT[B1]	111 00 0 0	111 00 0 0
IN[B2]	111 00 0 0	111 01 1 1 <small>unione di out B1 e B4</small>
OUT[B2]	001 11 0 0	001 11 1 0
IN[B3]	001 11 0 0	001 11 1 0
OUT[B3]	000 11 1 0	000 11 1 0
IN[B4]	001 11 1 0	001 11 1 0
OUT[B4]	001 01 1 1	001 01 1 1
IN[exit]	001 01 1 1	001 01 1 1



4.2.7 Algoritmo Worklist

La worklist contiene le variabili che devono ancora essere processate. Quando la worklist è vuota l'algoritmo termina.

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
out[Entry] =  $\emptyset$  // can set out[Entry] to special def
                  // if reaching then undefined use
for all nodes i
    out[i] =  $\emptyset$  // can optimize by out[i] = gen[i]
ChangedNodes = N

// Iterate
while ChangedNodes  $\neq \emptyset$  {
    remove i from ChangedNodes
    in[i] =  $\cup$  (out[p]), for all predecessors p of i
    oldout = out[i]
    out[i] =  $f_i$ (in[i]) // out[B] = Gen[B]  $\cup$  (in[B] - Kill[B])
    if (oldout  $\neq$  out[i]){
        for all successors s of i
            add s to ChangedNodes
    }
}
```

4.3 Liveness Analysis

4.3.1 Live Variable Analysis

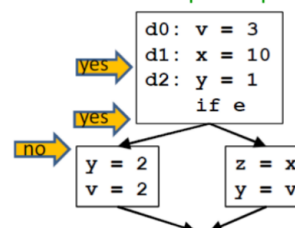
Definizione: Una variabile v è **viva** (*live*) in un punto p del programma se il valore di v è usato lungo qualche percorso del FG a partire da p (altrimenti è **morta**)

for i = 0 to n
... i ...
for i = 0 to n
... i ...

Posso riusare lo stesso registro se i non è viva qui

Motivi dell'analisi: (oltre alla DCE) es. *register allocation*: i registri reali sono limitati - la *ra* è l'operazione che cerca di evitare le spill in cache e in memoria, cercando i casi in cui risulta possibile **riutilizzare** un certo registro \rightarrow dipende evidentemente dalla liveness di una variabile

\bar{v} è viva in questo punto?



Definizione del problema DFA

- devo stabilire \forall BB quali sono le variabili **vive** in ciascuno di essi
- bit-vector di lunghezza pari al numero di variabili

4.3.2 Forward e Backward analysis

es. *Reaching Definitions*: che informazione sto cercando per capire se una definizione raggiunge o meno un punto p del programma? Devo analizzare il "**passato**" - gli statement tra la definizione e p per cercare eventuali kill (analisi da *entry* a p)

\rightarrow Nel contesto della *Liveness Analysis* invece, devo analizzare il "**futuro**" - cerco gli usi della variabile da *exit* a p

4.3.3 Funzione di trasferimento

Per la formulazione più tipica della *LA* (ce ne sono diverse, a seconda delle fonti) usiamo

- l'insieme delle variabili vive che può **generare** un BB ($Use[B]$)
- l'insieme delle variabili **definite** nel BB ($Def[B]$)
- le variabili vive in ingresso che il bb può propagare ($Out[B] - Def[B]$)

→ **funzione di trasferimento** per il blocco B :

$$In[B] = Use[B] \cup (Out[B]Def[B])$$

Flow Graph

- $In[B] = f(Out[B])$ (*backward analysis*, contrario rispetto a *Reaching Definitions*)
- **join node**: nodo con **successori** multipli
- **meet operator** (\wedge): $out[B] = in[s_1] \cup \dots \cup in[s_n]$ con s_1, \dots, s_n tutti predecessori di B (ricorda: $\cup \implies \exists$ almeno un percorso)

4.3.4 Algoritmo iterativo

- **boundary condition**: \emptyset
- **starting conditions**: \emptyset

La scelta delle condizioni iniziali deriva principalmente dalla scelta del meet operator - se per qualsiasi motivo scegliessimo come meet operator \cap , usare \emptyset impedirebbe qualsiasi propagazione dell'informazione

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
in[Exit] =  $\emptyset$ 

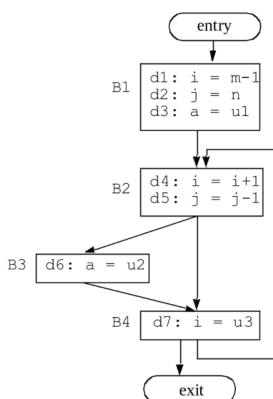
// Initialization for iterative algorithm
for each basic block B other than Exit
    in[B] =  $\emptyset$ 

// Iterate
while (changes to any in[] occur) {
    for each basic block B other than Exit {
        out[B] =  $\cup$  (in[s]), for all successors s of B
        in[B] =  $f_B(out[B])$  //  $in[B] = Use[B] \cup (Out[B] - Def[B])$ 
    }
}
```

A convergenza si arriva **indipendentemente dall'ordine** in cui calcolo le funzioni dei BB (cambia al massimo il numero di iterazioni per arrivarci)

Esempio

	First Pass	Second Pass
OUT[entry]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
IN[B1]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
OUT[B1]	{i,j,u2,u3}	{i,j,u2,u3}
IN[B2]	{i,j,u2,u3}	{i,j,u2,u3}
OUT[B2]	{u2,u3}	{j,u2,u3}
IN[B3]	{u2,u3}	{j,u2,u3}
OUT[B3]	{u3}	{j,u2,u3}
IN[B4]	{u3}	{j,u2,u3}
OUT[B4]	{}	{i,j,u2,u3}



4.4 Framework per DFA

Possiamo usare una tabella di questo tipo per descrivere problemi nell'ambito della DFA:

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: out[b] = $f_b(\text{in}[b])$ in[b] = $\wedge \text{out}[\text{pred}(b)]$	backward: in[b] = $f_b(\text{out}[b])$ out[b] = $\wedge \text{in}[\text{succ}(b)]$
Transfer function	$f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b)$	$f_b(x) = \text{Use}_b \cup (x - \text{Def}_b)$
Meet Operation (\wedge)	\cup	\cup
Boundary Condition	out[entry] = \emptyset	in[exit] = \emptyset
Initial interior points	out[b] = \emptyset	in[b] = \emptyset

4.5 Available Expressions

Altro problema modellabile tramite DFA, utile ad es. per la Common Subexpression Elimination

```
if (...) {
  x = m + n;
} else {
  y = m + n;
}
z = m + n;
```

```
if (...) {
  x = m + n;
} else {
  ...
}
z = m + n;
```

$m + n$ è ridondante perché già calcolato Cosa cambia se non viene calcolato nel ramo **else**?

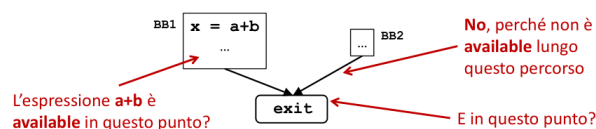
Il concetto di available expr. è necessario come maniera rigorosa di ragionare sulla **ridondanza**

Definizione del problema:

- **dominio:** tutte le espressioni del programma
 - consideriamo solo espr. binarie del tipo $x \oplus y$
- un'espressione $x \oplus y$ è **available** in un punto p del programma se **ogni** percorso che parte da *Entry* e arriva a p valuta l'espressione
- **funzione di trasferimento** di un BB:

$$f_B(x) = \text{Gen}[B] \cup (x - \text{Kill}[B])$$

- un blocco **genera** l'espressione $x \oplus y$ se la valuta e **non ridefinisce in seguito** x o y
- un blocco **uccide** l'espr. $x \oplus y$ se assegna (o potrebbe assegnare) un valore a x o y e non ricalcola successivamente l'espressione
- **direzione di analisi: forward**
 - nell'analisi delle *Available Expressions* eliminiamo un'espr. perché è stata calcolata **in passato**
 - (nell'analisi delle *Live Variables* eliminiamo una var. perché non verrà usata **in futuro**)
- equazioni *Out*: $\text{Out}[B] = f_b(\text{In}[B])$
- equazioni *In*: $\text{In}[B] = \wedge_{p \in \text{pred}(B)} (\text{Out}[B])$
- **meet operator** (\wedge): \cap ("ogni percorso che parte da Entry [...]")



- **boundary conditions:** $Out[Entry] = \emptyset$
- **initial conditions:**
 - ~~$Out[B_i] = \emptyset$~~ → il meet op. è \cap
 - $Out[B_i] = \mathcal{U}$

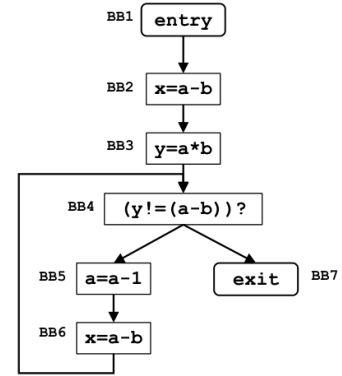
	Available Expressions
Domain	Sets of Expressions
Direction	Forward: $out[b] = f_b(in[b])$ $in[b] = \wedge out[pred(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation (\wedge)	\cap
Boundary Condition	$out[entry] = \emptyset$
Initial interior points	$out[b] = \mathcal{U}$

Fig. 4.7: Tabella completa

Esempio

Risolto su fogli, dominio $\{a=b, a*b, a-1\}$

Essendo la DFA per natura **statica**, considera sempre il caso **peggiore** di esecuzione.
Il risultato dell'analisi sarebbe diverso se conoscessi l'istanza specifica - sapendo ad es. che BB4 sarà falso potrei fare delle assunzioni che, staticamente, non posso fare → ritorno conservativamente al caso peggiore che il loop esegua almeno una volta



4.6 Very Busy Expressions (Assignment 2)

- Un'espressione è **very busy** in un punto p se, indipendentemente dal percorso preso da p , l'espressione viene usata prima che uno dei suoi operandi venga definito.
- Un'espressione $a+b$ è **very busy** in un punto p se $a+b$ è valutata in tutti i percorsi da p a *Exit* e non c'è una definizione di a o b lungo tali percorsi

Dove

- $Gen[b]$: espressioni valutate all'interno del Basic Block
- $Kill[b]$: un'espressione viene uccisa quando un suo operando viene ridefinito all'interno di b

	Very Busy Expressions
Domain	Expressions
Direction	Backward $in[b] = f_b(out[b])$ $out[b] = \wedge in[succ[b]]$
Transfer function	$f_b = Gen_b \cup (x - Kill_b)$
Meet Operation (\wedge)	\cap
Boundary Condition	$in[exit] = \emptyset$
Initial interior points	$in[b] = \mathcal{U}$

4.7 Dominator Analysis (Assignment 2)

- In un CFG diciamo che un nodo X **domina** un altro nodo Y se il nodo X appare in ogni percorso del grafo che porta dal blocco *Entry* al blocco Y
- annotiamo ogni basic block B_i con un insieme $Dom[B_i]$:
 - $B_i \in Dom[B_j] \iff B_i$ domina B_j
 - $B_i \in Dom[B_i]$: per definizione un nodo domina se stesso

	Dominator Analysis
Domain	Basic Blocks
Direction	Forward $out[b] = f_b(in[b])$ $in[b] = \wedge out[pred[b]]$
Transfer function	$f_b = Dom[b] = b \cup x$
Meet Operation (\wedge)	\cap
Boundary Condition	$out[entry] = entry$
Initial interior points	$out[b] = \mathcal{U}$

4.8 Constant Propagation (Assignment 2)

- L'obiettivo della *constant propagation* è quello di determinare in quali punti del programma le variabili hanno un valore costante.

- L'informazione da calcolare per ogni nodo del CFG è un insieme di coppie del tipo **<variabile, valore costante>**.

	Constant Propagation
Domain	pairs (v, c)
Direction	Forward $out[b] = f_b(in[b])$ $in[b] = \wedge out(pred[b])$
Transfer function	$f_b = Gen[b] \cup (x - Kill[b])$
Meet Operation (\wedge)	\cap
Boundary Condition	$out[entry] = \emptyset$
Initial interior points	$out[b] = \mathcal{U}$

- $Gen[b]$ rappresenta l'insieme di nuovi assegnamenti (all'interno di b) con valore costante. I casi di interesse sono:
 - $x = c$ con c costante: $Gen[b] = Gen[b] \cup (x, c)$
 - $x = c1 \oplus c2$ con $c1, c2$ costanti: $Gen[b] = Gen[b] \cup (x, c1 \oplus c2)$ (calcolando il valore dell'espressione)
 - $x = c \oplus y$ o $x = y \oplus c$ con c costante:
 1. controlliamo che y sia presente nell'insieme in input: $\exists (y, v_y) \in In[b]$
 2. se è vero, $Gen[b] = Gen[b] \cup (x, e)$ con $e = c \oplus v_y$ (o $v_y \oplus c$)
 - $x = y \oplus z$, valutando y e z allo stesso modo del caso precedente:

$$\exists (y, v_y), (z, v_z) \in In[b] \implies Gen[b] = Gen[b] \cup (x, e) \text{ con } e = v_y \oplus v_z$$

Nel caso di definizioni multiple di x , riteniamo valida solamente l'ultima all'interno di b

- $Kill[b]$: ogni definizione $x = \text{expr}$ uccide le coppie $(x, c) \in \mathcal{D}$ (dominio)

5 Loops e UD-DU chains

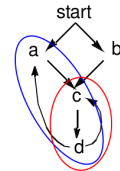
5.1 Cos'è un loop

Sapendo che un programma spende la maggior parte del tempo nei loop, l'obiettivo è definire un loop **in termini di teoria dei grafi** (CFG), ovvero **indipendentemente dalla loro sintassi e dal tipo** (**for**, **while**, **goto**, costrutti stile assembly con salti condizionati e non...)

Elementi chiave per riconoscere un loop:

- gli archi devono formare almeno un ciclo
- (fondamentale) **singolo entry point** (tutti gli archi, se multipli, devono entrare nello stesso punto)

Generalmente, non tutti i *cicli* sono un "loop" da un p.to di vista dell'ottimizzazione: $c \rightarrow d$ è un loop, $a \rightarrow c \rightarrow d$ no



5.1.1 Definizioni formali

Dominator Un nodo d domina un nodo n in un grafo ($d \text{ dom } n$) se ogni percorso da $ENTRY$ a n passa per d

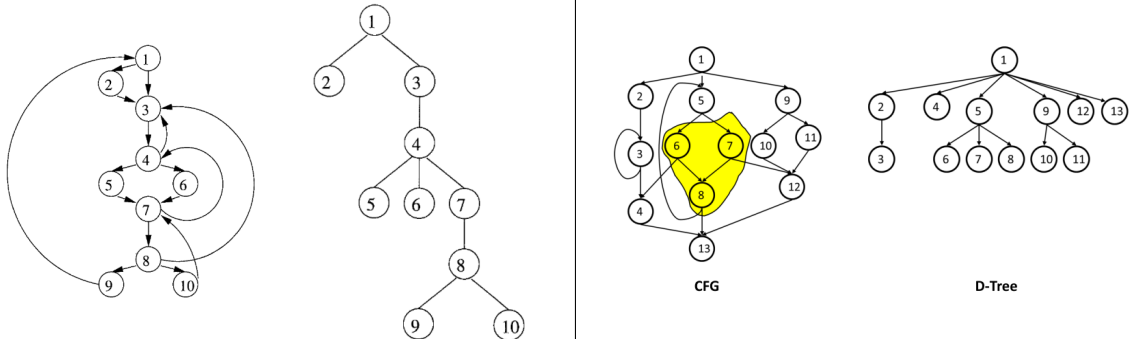
Immediate dominator L'ultimo dominator di n su qualsiasi percorso da $ENTRY$ a n m domina **immediatamente** (strettamente) n ($m \text{ sdom } n$) $\iff m \text{ dom } n \wedge m \neq n$

Dominator Tree Modo per rappresentare la proprietà di dominanza **in forma di albero**

- $a \rightarrow b$ nel dominator tree $\iff a \text{ sdom } b$
- **non compaiono** le relazioni di "**auto-dominazione**" (ogni nodo domina sempre se stesso, ma nell'albero non serve rappresentarlo)

- *ENTRY* è la **radice**
- ogni nodo *d* domina solo i **suoi discendenti** nell'albero

Esempi di DT



5.1.2 Loop naturali

Nonostante le diverse forme trovate nei sorgenti, dal punto di vista dell'analisi ci interessa solo che abbiano proprietà che facilitino l'ottimizzazione, ovvero

- **singolo entry point**: header \rightarrow l'header **domina tutti i nodi nel loop**
- back edge, ossia arco la cui testa domina la propria coda (tail \rightarrow head): un back edge **deve far parte di almeno un loop**

5.2 Identificare i loop naturali

1. trovare le relazioni di **dominanza**
2. identificare i **back edges**
3. trovare il **loop naturale associato** al back edge

5.2.1 Trovare i dominatori

Lo impostiamo in termini di DFA (vedi secondo assignment)

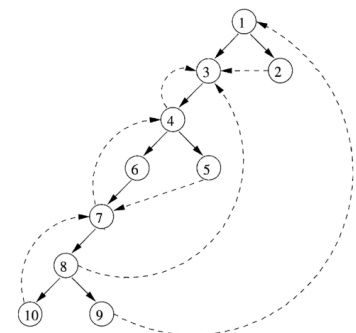
5.2.2 Trovare i back edges

Usiamo un algoritmo basato su *depth-first traversal*

- inizio alla radice e visito **ricorsivamente** i figli di ogni nodo **in qualsiasi ordine**
- importante la "velocità di discesa": prima scendo ed esploro **in profondità**, a prescindere dall'ordine

Il percorso della visita definisce un **depth-first spanning tree (DFST)**:

- archi **solidi**: struttura del DT
- archi **tratteggiati**: altri archi del CFG



Categorizzazione degli archi

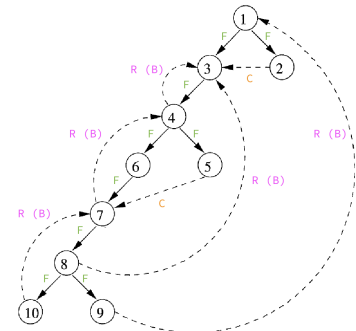
- **advancing** (A) edges: da antenato a discendente, ovvero gli archi detti *proper* - gli archi solidi sono tutti A
- **retreating** (R) e.: da discendente a antenato (non necessariamente proper \rightarrow da un nodo a se stesso) - solo archi tratteggiati
- **cross** (C) e.: archi tali per cui nessuno dei due nodi è antenato dell'altro

Se disegniamo il DFST in modo che i figli siano aggiunti da sx a dx nell'ordine di visita, allora i cross edges vanno sempre da dx a sx

Algoritmo

- esegui una depth-first search
- \forall retreating edge $t \rightarrow h$ controlla se $h \text{ dom } t$

Vado a riconoscere i *back edges* come casi specifici di *retreating edges* \rightarrow la maggior parte dei programmi hanno control flow **riducibili**, ovvero tutti i *retreating edges* sono anche back



5.2.3 Trovare il loop naturale associato

Il loop naturale di un back edge è il **più piccolo** insieme di nodi che **include head e tail** del back edge e **non ha predecessori fuori da questo insieme**

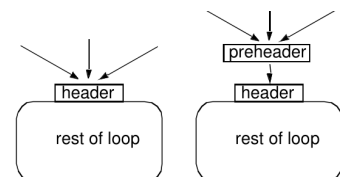
Algoritmo

- eliminare h dal CFG
 - trovare i nodi che raggiungono t : questi (più h) formano il **loop naturale** $t \rightarrow h$
- \rightarrow genericamente rappresento i loop in maniera "header-centrica"

5.3 Preheader

Spesso (propedeutico a varie ott.) è necessario garantire che, quando arrivo all'header, ci arrivo tramite arco *fallthrough*: es. LICM (caso di code hoisting), che sposta un'istr. in un punto comune al control flow di tutte le iterazioni del loop

\rightarrow tipicamente inserisco un blocco *preheader* appena prima del loop, apposta per inserire le istr. "hoistate"



Per manipolare i loop in LLVM esistono primitive per recuperare proprio questi blocchi fondamentali (poi per navigare nel resto dei BB usiamo gli iteratori, sia seguendo l'ordine del CFG sia non)

5.4 Use-def e Def-use chains

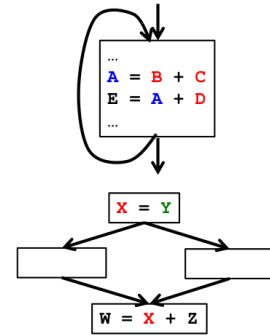
La capacità di ricollegare in maniera efficiente la definizione di una variabile a tutti i suoi usi (per esempio per propagare un risultato) è indispensabile all'ottimizzazione \rightarrow per questo vengono previsti questi riferimenti nell'IR LLVM

5.4.1 Dove viene definita o usata una variabile

Le definizioni di DU e UD chain **precedono quella di SSA** \rightarrow lo scope lessicale del programma in cui si potevano trovare gli usi della variabile era molto più ampio

Esempi

- LICM: se le var. usate nell'espressione che definisce A sono **definite all'interno del loop**, non posso spostare la definizione fuori
- CP: per un dato uso di X, trovo le sue *reaching definitions* (es. $X = Y$) e, se non sono ridefinite (appunto "reaching") vado a sostituirne gli usi



A questo tipo di ottimizzazioni beneficia molto poter scorrere agevolmente le relazioni DU delle variabili → vogliamo un'IR che lo preveda e dunque permetta una **forma di analisi "sparsa"** (ignoro tutte le istruzioni non relative alla var. correntemente analizzata)

In generale le catene possono essere onerose (per N defs e M uses, $O(NM)$ spazio e tempo) → Possiamo semplificare ulteriormente se riusciamo a **ridefinire completamente anche il nome della variabile ad ogni sua ridefinizione** (ci avviciniamo a SSA) (permette catene sensibilmente più corte)

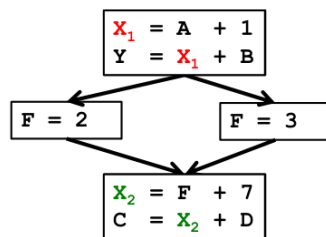


Fig. 5.8: Un altro motivo per rinominare le ridefinizioni è che occorrenze della stessa variabile potrebbero essere scorrelate e dunque ottimizzabili separatamente

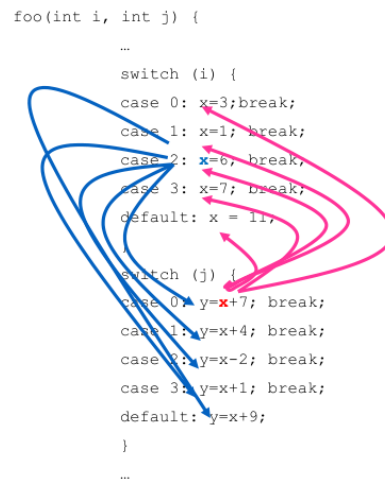
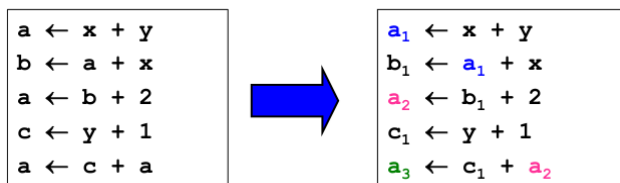


Fig. 5.9: Esempio di catena onerosa. Ulteriore soluzione è limitare ogni variabile a una definizione

6 Static Single Assignment (SSA)

Forma di IR dove ad ogni variabile viene **assegnato un valore solo una volta** → Facile dentro ad un BB, ma cosa succede nei punti di **join** di un CFG? → **usiamo una notazione fittizia: la Φ function**

Local value numbering



All'interno di un BB posso fare *local value numbering*, visitando ogni istruzione in ordine:

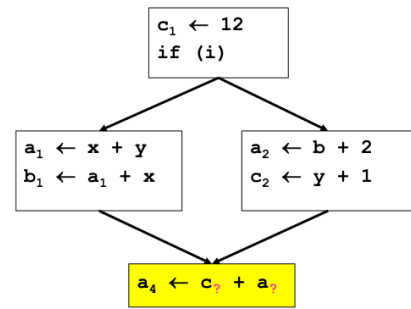
- LHS: assegna ad una nuova versione della variabile
- RHS: usa la versione più recente della stessa

```

c ← 12
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
a ← c + a

```

→

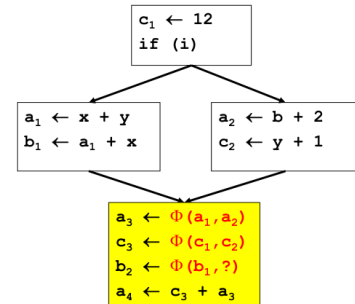


6.1 La funzione Φ

- Φ fonde multiple definizioni derivanti da multipli percorsi in una singola definizione
- per un BB con p predecessori ci sono p argomenti nella funzione Φ :

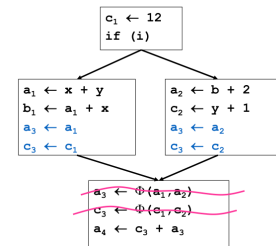
```
xnew ←  $\Phi(x_1, x_2, \dots, x_p)$ 
```

- tipicamente **non ci interessa quale x_i usare** → se rilevante, usiamo le definizioni derivanti dall'arco di interesse



Come si implementa Φ

Di fatto, a tempo di esecuzione ci sarà sempre un percorso **effettivamente eseguito** → per questo consideriamo Φ come notazione fittizia



6.2 SSA triviale

- ogni assegnamento genera una nuova versione della variabile
- aggiungiamo una fz. Φ ad ogni p.to di join \forall le variabili *live*

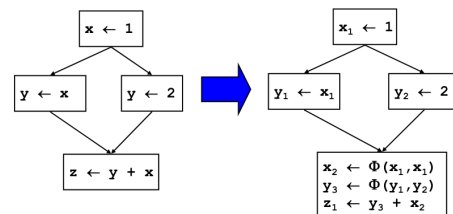
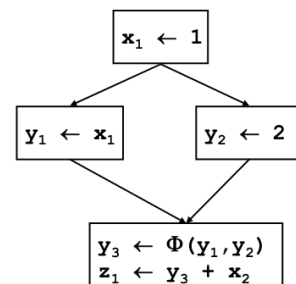


Fig. 6.10: Molte di queste funzioni Φ non sono necessarie

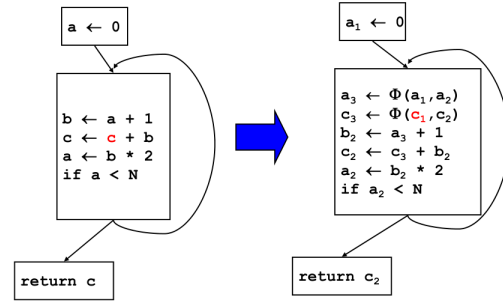
6.3 SSA minimale

- ogni assegnamento genera una nuova versione della variabile
- aggiungiamo una fz. Φ ad ogni p.to di join \forall le variabili *live* con definizioni multiple



Esempio con loop

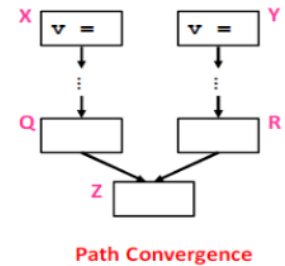
c_1 altro non è che la rappresentazione del valore iniziale che assume c a tempo del primo assegnamento (prima esecuzione del loop), mentre c_2 il valore della variabile all'iterazione precedente



6.4 Dove inserire le funzioni Φ

Inserisco una Φ per una variabile a nel blocco $Z \iff$:

- a è stata definita più di una volta (es. in X e Y , $X \neq Y$)
- \exists due percorsi da X a Z e da Y a Z t.c.
 - $Pxz \cap Pyz = \{Z\}$ (**Z unico BB comune** tra i percorsi)
 - $Z \notin Pxz \vee Z \notin Pyz$, dove $Pxz = Pxz \rightarrow Z, Pyz = Pyz \rightarrow Z$ (almeno un percorso **raggiunge Z per la prima volta**)



Note

- **ENTRY** contiene una definizione implicita di tutte le variabili
- $v = \Phi(\dots)$ è una definizione di v

6.5 Proprietà di dominanza della forma SSA

Nella forma SSA le **definizioni dominano gli usi**:

- se x_i è usato in $x \leftarrow \Phi(\dots, x_i, \dots) \implies \text{BB}(x_i)$ **domina** il predecessore i -esimo di $\text{BB}(\Phi)$
- se x è usato in $y \leftarrow \dots x \dots \implies \text{BB}(x)$ **domina** $\text{BB}(y)$

SI può usare questa proprietà per derivare un algoritmo efficiente per convertire la IR in forma SSA

6.6 Dominanza e Dominance Frontier

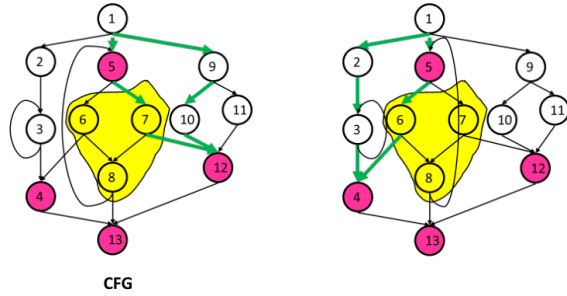
Risulta interessante considerare i BB "appena prima" o "appena dopo" i blocchi dominati o che ci dominano \rightarrow i **dominators** e **postdominators** ci dicono quale BB **deve** essere eseguito **prima**, o **dopo**, un certo BB x

Dominance Frontier di un nodo x : $DF(x) = \{w | x \text{ dom } \text{pred}(w) \wedge (x \text{ sdom } w)\}$

\rightarrow ovvero, l'insieme di tutti i BB che sono successori immediati dei blocchi dominati da x , ma che non sono strettamente dominati da x

Esempio

Per il CFG di esempio, $DF(5) = \{4, 5, 12, 13\}$: osservando il DT (pag. 24), notiamo che 4, 12, 13 sono successori di 6, 7, 8 (prima parte della condizione), e non sono dominati da 5 (dunque neppure strettamente, seconda parte). Anche 5 è successore di 8, e poiché un nodo non domina strettamente se stesso, lo includiamo nella DF



6.7 Utilizzo della Dominance Frontier per calcolare la forma SSA

I passaggi da seguire sono

1. posizionare tutte le funzioni Φ (se c'è una definizione di a nel blocco x , i nodi nella $DF(x)$ **necessitano di una funzione Φ** per a)
2. rinominare tutte le variabili

6.7.1 Utilizzo della DF per posizionare le funzioni $\Phi()$

- identifico tutti i siti dove vengono definite le variabili
- \forall variabile v :
 - \forall sito di definizione n (**non siamo ancora in forma SSA, la stiamo costruendo**):
 - * \forall nodo in $DF(n)$
 - se non c'è una $\Phi()$ nel nodo, mettiamone una
 - se questo nodo non definiva la variabile in precedenza, aggiungiamo questo nodo alla lista dei siti di definizione

Algoritmo

```
foreach node n {
  foreach variable v defined in n {
    orig[n]  $\cup$  = {v}
    defsites[v]  $\cup$  = {n}
  }
}
foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n] {
      if y  $\notin$  PHI[v] {
        insert "v  $\leftarrow \Phi(v, v, \dots)$ " at top of y
        PHI[v]  $\cup$  = {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
    }
  }
}
```

Questo algoritmo calcola iterativamente la DF, inserendo il numero minimo necessario di funzioni $\Phi()$

6.7.2 Rinominare le variabili

Algoritmo

- scorrere il DT, rinominando le variabili come le si incontra
- rimpiazzare gli usi con la più recente definizione rinominata
- in presenza di branch e join: usare la definizione più vicina tale per cui la definizione si trova **prima** dell'uso nel DT

Algoritmo per implementazione tramite stack

```
// inizializza stacks e contatori
foreach variable v {
    stacks[v] =  $\emptyset$ 
    counters[v] = 0
}

// procedura per la generazione del prossimo nome
define GenName(variable v) {
    i = counters[v]
    replace v by  $v_i$ 
    push i onto stacks[v]
    counters[v]++
}

// procedura per il renaming visitando in pre-order il DT
define Rename (node X) {
    foreach  $\Phi$  p in X { // scorro i phi
        GenName(LHS(p)) // nuova def.
    }

    foreach statement a in X { // scorro il resto degli statement
        foreach variable v in RHS(a) {
            i = top(stacks[v])
            replace v by  $v_i$  // sostituisco gli usi con la def. piu' recente
        }
        foreach variable v in LHS(a): GenName(v) // nuova def.
    }

    foreach Y in succ(X) { // sostituisco gli usi nelle phi dei successori
        j = "position in Y's  $\Phi$  corresponding to X"
        foreach  $\Phi$  p in Y {
             $v_k$  = LHS(p) // k fittizio, utile solo per recuperare stacks[v]
            i = top(stacks[v])
            replace RHS(p)[j] by  $v_i$ 
        }
    }

    foreach Y in succ(X) { // visito ricorsivamente i successori
        Rename(Y)
    }

    foreach  $\Phi$  or statement a in X { // pop delle var. definite in X
        foreach  $v_i$  in LHS(a): pop(stacks[v])
    }
}
```

7 Ottimizzazioni sulla memoria

7.1 Memorie, tecnologie RAM

Le ottimizzazioni sulla memoria sono solitamente **architecture dependent** (si effettuano nel backend), ma alcune si possono già fare nel middle end, conoscendo le **problematiche generali di efficienza** della DRAM (memoria principale)

7.1.1 Tecnologia DRAM

- Dati immagazzinati in condensatori, in cui c'è o non c'è carica elettrica; va effettuata periodicamente una operazione di *refresh*
- bit in DRAM organizzati come **array rettangolari**
- accesso effettuato su righe: insieme di condensatori lungo un certo numero di *words*
- **accessi costosi**: cerco di minimizzarli, e in particolare massimizzare quelli nella riga attuale
- negli anni si sono studiate varie possibili ottimizzazioni, in particolare sulle modalità di accesso
 - **burst mode**: latenza ridotta per accessi a parole successive nella stessa riga
 - **DDR** (Double Data Rate) RAM: sfrutta i fronti di salita e discesa del clock del circuito elettronico come momenti di trasferimento dati
 - **QDR** (Quad DR): separa input e output di DDR

→ sono tutti trucchi per migliorare i tempi delle operazioni su HW che di base è "lento"

DRAM generations

Negli anni la tecnologia DRAM si è dimostrata estremamente **cost effective** (costo/unità di memoria), mentre la **latenza** di accesso (sia a righe nuove che quelle già caricate) decresce molto più lentamente → scarsa in termini di performance

7.1.2 DRAM performance factors

Vediamo altre tecniche per migliorare le prestazioni:

- **row buffer**: mantengo un buffer delle righe accedute di frequente
- **DRAM sincrona**: sfrutta accessi in burst per migliorare la banda (quantità di mem acceduta o trasferita in un periodo di tempo)
- **DRAM banking**:
 - *master*: qualsiasi device (CPU, periferiche, ecc.) che può essere utilizzatore della memoria
 - se tanti master devono accedervi (già è lenta), la DRAM diventa bottleneck per le prestazioni
 - il concetto di banking sfrutta quello di parallelismo: ho più "banchi" di DRAM che operano **indipendentemente** tra loro
 - abbasso il **tempo medio** di accesso: i master accederanno a risorse che non necessariamente si trovano sullo stesso banco

7.1.3 Memory wall

Uno dei "muri" tecnologici che hanno portato alla rivoluzione dei primi 2000 sul passaggio da single a multicore CPU

- Cresce moltissimo il divario tra CPU e DRAM performance (**55% aumento annuo vs 7%**)
- $CPI_{ideale} \leq 1$, mentre **load** e **store** (uniche a interagire con DRAM) hanno $CPI_{l/s} \simeq 100$
- naturalmente \nexists programmi che non accedono alla memoria, e negli anni 80 non ci si poneva questi problemi (\nexists cache, il costo di op. su DRAM era equiv. a quello di op. su CPU)

→ cambio di approccio: si introducono le cache

7.2 Cache

- Usano tecnologia SRAM (Static R.), basata su transistor che decide se far passare o meno corrente
 - si trovano sulla motherboard **molto vicino alla CPU**: anche questo aumenta la velocità
- **t. di accesso** molto ridotti (ordine dei *ns*, freq. di GHz - come la CPU) e **densità** molto più alta
- ovviamente, a discapito del costo → per questo usiamo le **gerarchie di memoria**

7.2.1 Gerarchie di memoria

- **block (line)**: unità di copia, solitam. multiple *word*
- in base alla presenza o meno del dato nell'attuale livello di cache, ottengo una **hit** o una **miss** - quando *misso* devo fermare la load/store unit e copiare il dato dalla cache inferiore (e così via)
 - hit ratio: $\frac{hits}{accesses}$
 - miss penalty: tempo necessario per gestire la miss
 - miss ratio: $1 - hit\ ratio$

Velocità delle cache

- L1: 1-2 *cicli per accesso* (=CPU), ma molto piccola
- L2 ~ 5-10 (nonostante stessa tecnologia): in parte per dimensioni maggiori, ma soprattutto perché devo **spezzare il critical path**
- **c. path**: path fisico più lungo nel circuito - il clock rate deve adattarsi (CPU monolitiche devono garantire l'esecuzione di tutti gli stadi di un'istr. in 1 ciclo), dunque per $cp > \Rightarrow$ freq. $<$
- scelta di design basilare: spezzare il critical path verso L2 con dei registri intermedi → aumenta il *CPaccess* ma non penalizza il clock r.

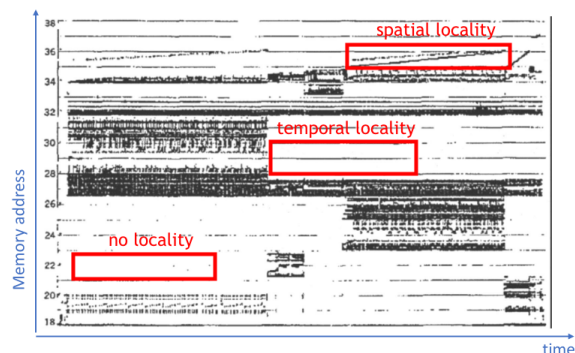
7.2.2 Principi di località

Si basano sulla conoscenza generale di memoria e cache (HW), e sui tipici pattern di accesso ai dati (a liv. SW)

- **l. temporale**: el. acceduti recentemente sono probabilmente necessari di nuovo in breve (es. istr. in loop)
- **l. spaziale**: probabilità di accedere in successione a el. contigui (es. accesso ad array)

Visualizzazione grafica delle località

- no locality: segmenti in cui non ho accessi o non ho pattern di accesso
- temporal locality: righe dritte orizzontali
- spatial locality: righe diagonali



7.2.3 Cache misses

- **cold cache (compulsory)**: non ci sono dati nella cella della cache (tipicamente appena eseguo il programma)
- **conflict**: quando due indirizzi in DRAM sono mappati alla stessa linea di cache
- **capacity**: ho finito lo spazio disponibile in cache

7.2.4 Tipologie di cache

Direct mapped

- linee mappate in base agli ultimi bit dell'indirizzo in DRAM
- *valid bit* garantisce la validità di una linea
- schema di indirizzamento e rete logica semplice (area, costo, velocità <) → **hit time basso**
- uso non ottimale dello spazio: frequenti *conflict* (1 sola possibile locazione in cache \forall linea)

Fully associative

- nessun indirizzamento fisso, la nuova entry occupa la prossima linea libera
- richiede la ricerca **in tutte le linee** contemporan. → **costoso** (1 comparatore \forall entry)

N-way set associative

- mantiene il concetto di bit di indice, per indicare non un indirizzo unico ma un **set** di indirizzi, e poi tanti comparatori quanti gli elementi del set

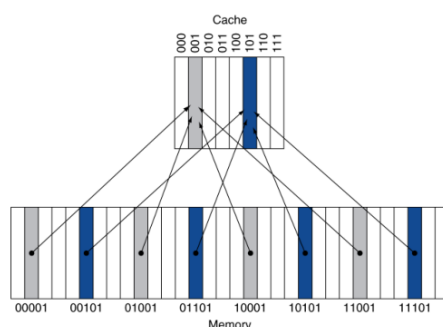


Fig. 7.11: Cache direct mapped

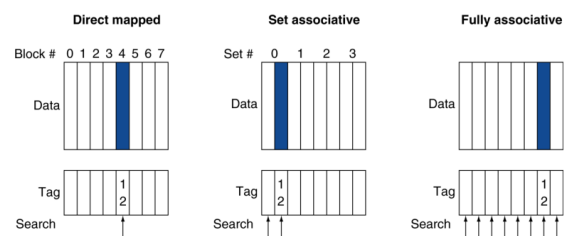


Fig. 7.12: Le tre tipologie di cache

Per sfruttare la spatial l. le linee di cache devono essere più grandi di una word (in seguito a una miss carico molti dati adiacenti e riduco miss rate), ma per cache di dim. fissa blocchi molto grandi aumentano la competizione e inquinano la cache

7.3 Memory Hierarchy optimizations

Queste ottimizzazioni vogliono

- **ridurre la latenza di accesso** alla memoria e diminuire CPI per load e store
- **massimizzare la banda** di memoria: poter accedere a più dati nella stessa unità di tempo (stesso concetto della pipeline: non riduco il tempo di exec. di una singola istruzione, ma a regime aumento il throughput)
- **gestire overhead**, i costi aggiuntivi dovuti all'ottimizzazione: maschero il tempo "perso" sovrapponendo altre operazioni

7.4 Reuse and locality

Consideriamo come accediamo ai dati:

- **data reuse**: sfrutta la località temporale, dunque il riuso di degli stessi dati o molto vicini tra loro - posso assicurare che il dato acceduto rimanga sempre in cache
- **data locality**: (es. accesso ad array) il compilatore si deve assicurare che questi dati vengano acceduti in questo modo, e poi garantire la loro presenza

Le miss dipendono dai pattern di accesso, dunque anche dal comportamento dell'algoritmo e delle ottimizzazioni del compilatore

7.5 Optimizing cache performance

Se una computazione ha del riuso, posso ottimizzarlo tramite

- layout e posizionamento ottimale dei dati
- trasformazioni sul codice (*reordering* delle operazioni)

Cosa migliorare

- l. spaziale
- l. temporale

Cosa minimizzare

- conflitti (es. rimpiazzamenti sbagliati)

7.5.1 Il ruolo del compilatore

Con il compilatore possiamo ottimizzare

- **tempo** (*quando accedo a un dato?*) → riordino delle operazioni
 - cosa rende difficile sapere **quando** accedo ad un oggetto?
 - come possiamo predire un momento di accesso **migliore**?
 - come possiamo essere sicuri che il riordino sia **safe**?
- **spazio** (*dove esiste il dato nell'address space?*) → modifica del layout dei dati
 - cosa sappiamo della **locazione** del dato? (dipende dal tipo)
 - come possiamo stabilire un layout **migliore** e quanto possiamo **alterare quello attuale**?

7.6 Tipi di oggetti da considerare

- scalari: tipi di dato semplici, non aggregati
- strutture dati e puntatori (più complessi)
- array

7.6.1 Scalari

- var. **locali** (scope e tempo di vita locale alla funzione in cui sono definite)
- var. **globali** (scope e tempo di vita globali pari alla vita del programma)
- **argomenti** delle funzioni

```
int x;
double y;
foo(int a){
    int i;
    ...
    x = a*i;
    ...
}
```

Tipicamente sono gestiti sempre tramite registri → non dobbiamo gestire gli accessi alla memoria (salto la fase di MEM, e la load-store unit non esegue nulla per questa istruzione)

Unico caso problematico: finiscono i registri utilizzabili → devo fare uno spill sulla memoria, in particolare sullo stack frame (ovvero faccio una store, e poi una load, esattamente come per i save registers (modificati da una chiamata a funzione, che alla fine dell'esecuzione deve ripristinarli)

→ **ottimizzazioni di backend** (calling convention, register file, ISA sono HW specific)

7.6.2 Strutture dati e puntatori

- struct: di base un'aggregazione di bytes → elem. **contigui in memoria** → in generale un oggetto complesso potrebbe non essere grande un multiplo di word o cache line, dunque **non posso sfruttare la spatial locality**
- posso **trasformare** la struttura del dato, ad esempio **estraendo dati di tipi primitivi** (nel caso di array di tipi composti) e costruendo array separati (devo conoscere la struttura della str. dati)

```
struct {
    int count;
    double vel;
    double inertia;
    struct node *neighb[N];
} node;
```

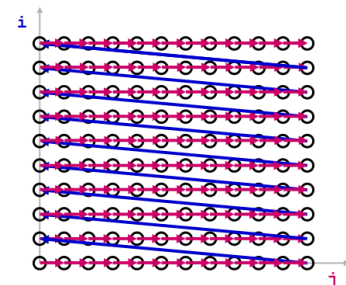
7.6.3 Array

- posso ottimizzarli sfruttando principi di l. spaziale
- solitamente accesso tramite loop nests (array multidim., op. matriciali, ...)
- staticamente, possiamo analizzare l'andamento della posizione relativa dell'elemento acceduto, ovvero l'evoluzione dell'**iteration space**

```
double A[N][N], B[N][N];
...
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```

7.7 Iteration space

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```



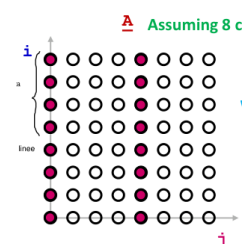
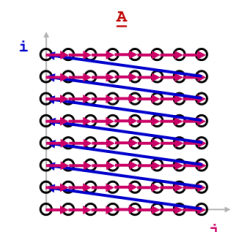
- ogni posizione rappresenta un'iterazione
- **iteration s. ≠ data s.**

Dall'esempio: cache miss per accessi ad A

Immaginiamo una cache di 4 parole per riga e 8 righe, vuota in partenza

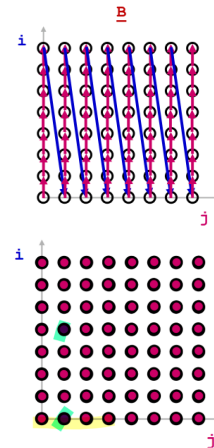
- A[0][0]: *cold cache m.*
- A[0][1]-A[0][3]: *hit* avendo caricato l'intera riga
- si ripete fino a A[0][15]
- a partire da A[1][0]: *conflict m.* (i bit di indexing si sovrappongono)

NB: non cambia il pattern di miss (1 ogni 4)



Dall'esempio: cache miss per accessi ad B

- $B[0][0] - B[3][0]$ *cold cache*, seguono *conflict* (suppongo stesso indexing per 1^a e 5^a riga)
- $B[0][1]$: **conflict** → la linea attualmente caricata è quella con $B[3][1]$
- situazione di **miss sistematica**, rilevabile e da correggere
- possiamo distinguere accessi **cache friendly** e non anche dalla discordanza tra *iteration* e *data s.*



7.8 Cache behavior of array accesses

Per ottimizzare gli accessi ci chiediamo

- **quando avvengono** le miss? → *locality analysis*
- possiamo cambiare **ordine delle iterazioni** (o **data layout**) per migliorare il comportamento degli accessi?
 - valuto il **costo** delle alternative, per stimare il beneficio ottenuto → giustificato solo se il riuso **copre il costo aggiuntivo** (es.: precalcolo di una matrice trasposta)
- ottengo sempre **risultati corretti**? → verifico gli effetti "collaterali" (*dependence analysis*)
 - dipendenza: **vincolo nell'ordine di esecuzione delle istr.**, dunque nell'iteration space
 - caso di loop-carried dependency semplice: $A[i] = A[i+1]$
 - framework di analisi solitamente basati sul *polyhedral model*

Sono ottimizzazioni di tipo "best effort" basate su euristiche → ottimizzano l'esec. del caso **medio**, e nel caso peggiore non ottimizzano nulla ma **non peggiorano l'esecuzione**

7.8.1 Approcci per variabili di cui non conosco il valore staticamente

- **conservativo**: non so determinare il valore, dunque (pessimisticamente) non faccio nulla
- genero entrambe le versioni del codice, e **stabilisco a runtime quale eseguire** (aggiungo un condizionale che esegue il pr. ottimizzato solo per qualche N abbastanza grande da giustificare il costo aggiuntivo)

7.9 Esempi di trasformazioni per i loop

7.9.1 Loop interchange

```

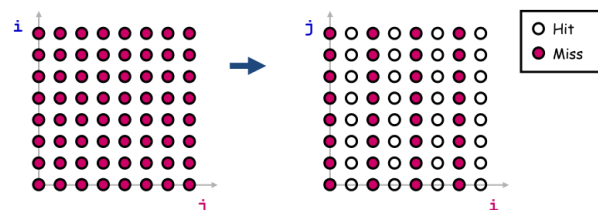
for i = 0 to N-1
  for j = 0 to N-1
    A[j][i] = i*j;
    
```

↔

```

for j = 0 to N-1
  for i = 0 to N-1
    A[j][i] = i*j;
    
```

Scambio (o permuta) l'ordine dei loop: in assenza di dipendenze, non altero in nessun modo il risultato finale e **miglioro la performance** (magari non accedendo nello stesso ordine di prima, ma se indipendente non importa)



Obiettivi della permutazione

- locality optimization (specialmente quella spaziale)
- riarrangiare i loop nest per ottenere la giusta **granularità** del parallelismo:
 - verso l'interno: parallelismo *fine-grain* → frammento il lavoro da svolgere in blocchi più piccoli, e mantengo più stabile la q.tà di lavoro tra thread e proc.

Caso estremo: task talmente piccoli che l'overhead (costo per creare thread, assegnare lavoro, proc. sync) supera il tempo di esecuzione (tipicamente il costo di creazione di un thread ha costo fisso → **dipende da HW**)

- esterno: *coarse-grain* parallelism

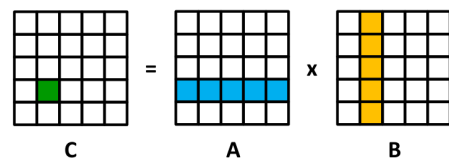
Caso estremo: pochi task rispetto ai proc. disponibili: se un proc. si ritrova un blocco "lento" (es. molte miss), gli altri dovranno aspettarlo una volta finito il loro lavoro

7.9.2 Loop tiling/blocking

In generale, il tiling viene usato gerarchicamente a "tutti" i livelli di memoria (registri, caches, buffer SW-managed, memorie piccole, ...) per gestire limiti di spazio

Esempio: Matrix multiplication

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      C[i][j] += A[i][k] * B[k][j]
```



Consideriamo il caso in cui non posso ospitare l'intera struttura in cache

- 3 matrici A, B, C $N \times N$, con $c_{ij} = \sum_{k=1}^N a_{ik} b_{kj}$, $i, j = 1, \dots, N$
- 3 loop innestati, di cui i primi due iterano sugli el. di C, e il terzo scorre sulla riga e colonna corrispondenti
- supponiamo che $N > \dim(\text{line}_{\text{cache}})$ → linee multiple per una riga intera
- ogni riga di A viene usata \forall colonna di B (e viceversa, ma iterare su colonne è meno *cache friendly*) → conviene tenerla in cache

Nel loop più interno:

- C[i][j] acceduto 1 volta, nessun riuso
- A[i][k] comporta accesso all'intera riga (che non sta in line singola)
- B[k][j] comporta accesso all'intera colonna (non sta in cache e ha bassa *spatial l.*)

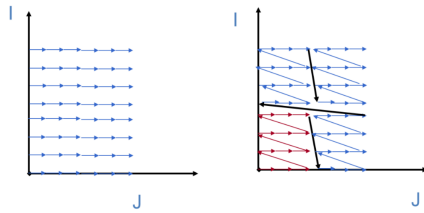
Miss totali - $2N^3 + N^2$:

- C: N^2 : ogni el. viene scritto N volte in sequenza (ma rimane in cache)
- A: N^3 : riga letta N volte in sequenza (ma non resta in cache)
- B: N^3 : col. letta N volte in sequenza (non resta e accedo in *column-major* order)

→ cerco una dim. ottimale di blocco che permette di **avvicinare temporalmente il riuso dei dati**

- cambiare l'ordine delle iterazioni e iterare "a blocchi" permette di sfruttare al massimo il riutilizzo dei dati (altrimenti siamo sicuri che gli usi sono talmente lontani da causare *conflict*)

→ aggiungo dei loop nest, che permettono di iterare blocco per blocco invece che a dimensione intera



```
for (jj=0; jj<N; jj+=B)
  for (kk=0; kk<N; kk+=B)
    for i = 0 to N-1
      for (j = jj; j < jj+B; j++)
        for (k = kk; k < kk+B; k++)
          c[i][j] += a[i][k]*b[k][j];
```

7.10 Loop fusion

Obiettivo: "fondere" tra loro loop separati del programma che accedono alle stesse strutture, per **garantire il riutilizzo dei dati caricati in cache** (annullo la distanza tra gli usi degli stessi dati)

Quando non conviene ottimizzare? Quando sono sicuro che la somma delle dimensioni dei dati (le matrici in questo caso) è **minore della dim. della cache** (i dati saranno ancora disponibili)

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    d[i][j] = a[i][j]+c[i][j];

↓

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];
    d[i][j] = a[i][j]+c[i][j];
```

7.11 Software prefetching

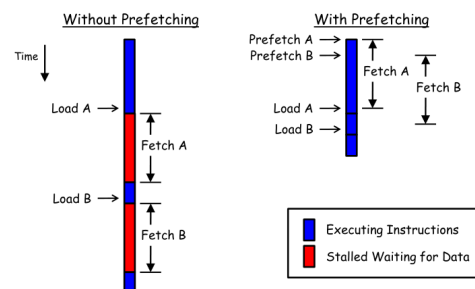
7.11.1 Coping with memory latency

- **ridurre la l.:** *locality optimizations* (sez. 7.5) - riordino iterazioni e layout per migliorare il riutilizzo
- **tollerare la l.:** *SW prefetching* - carico i dati in cache prima dell'uso per evitare stalli della load-store u. (comportamento **non lazy**)

7.11.2 Tollerare la latenza con il prefetching

I sistemi HW moderni permettono di disaccoppiare e **sovrapporre la computazione con gli accessi in memoria** → nascondo le latenze "infilando" il prefetching sotto ad altre operazioni

- Posso eseguire il pref. sia come ottimizzazione che direttamente nel sorgente
- un'istr. di prefetching agisce come una load non bloccante



Problemi pratici nel calcolo del prefetching

Nella teoria sono "certo" di quando ho bisogno di un dato, mentre in pratica **le risorse sono condivise** (possibili conflitti di accesso tra processi)

- pref. **troppo in anticipo** ⇒ la mem. potrebbe essere sovrascritta
- pref. **"tardi"** ⇒ inutile

7.11.3 Algoritmo di prefetching del compilatore

Tipicamente lavoro su **loop** (sono strutture ad esec. regolare), e svolgo

- **analisi** (cosa prefetchare) → *analisi di località*
- **scheduling** (quando/come eseguirlo) → *loop splitting* e *SW pipelining*

Loop splitting

Decompongo i loop per isolare istanze di *cache miss* (meno costoso di inserire **if**)

Locality Type	Predicate	Loop Transformation
None	True	None
Temporal	$i \div l = 0$	Peel loop l
Spatial	$(i \bmod l) = 0$	Unroll loop l by l

Applico ricorsivam. per loop **innestati**, sopprimo per loop **troppo grandi** (evito *code explosion*)

Software pipelining

Data la regolarità dei loop, posso esprimere la **distanza di pre-fetch** come un n^o di loro it.:

- l : mem. latency (tempo di lettura)
- s : shortest path attraverso il loop body

⇒ **it. necessarie per coprire il pref.** $It = \lceil \frac{l}{s} \rceil$

A questo punto suddivido il loop in 3 parti:

- **prolog**: solo prefetch, It volte (es. $It = 5$)
- **steady state** (s. a regime): pref. e accedo ai dati pronti
- **epilog**: finisco di usare gli ultimi dati prefetchati

Sono ottimizzaz. che si basano sempre su euristiche **best effort** → ovviamente **non posso eseguirle su feature e sistemi ritenuti critici**

```
for (i = 0; i < 100; i++) {  
    a[i] = 0;  
}
```

↓

```
/* Prolog */  
for (i = 0; i < 5; i++) {  
    prefetch(&a[i]);  
}  
/* Steady State */  
for (i = 0; i < 95; i++) {  
    prefetch(&a[i+5]);  
    a[i] = 0;  
}  
/* Epilog */  
for (i = 95; i < 100; i++)  
{  
    a[i] = 0;  
}
```