

Compilatori

Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 10 marzo 2025

Indice

1	Introduzione (25 feb)	2
1.1	Motivazione	2
1.1.1	La funzione dei compilatori	2
1.1.2	L'evoluzione dei compilatori	2
1.1.3	Eterogeneità architetturale	2
1.2	Ottimizzazione	2
1.2.1	Esempi di ottimizzazione	3
1.2.2	Ottimizzazioni sui loop	4
1.3	Anatomia di un compilatore	4
1.3.1	Flag di ottimizzazione	5
1.3.2	Uso di IR	5
1.3.3	Ingredienti dell'ottimizzazione	5
2	Rappresentazione intermedia (4 mar)	5
2.1	Proprietà di una IR	5
2.2	Tipi di IR	5
2.3	Categorie di IR	6
2.4	Esempi di rappresentazione	6
2.4.1	Sintassi concreta (testo)	6
2.4.2	AST	6
2.4.3	DAG	6
2.4.4	3AC	7
2.4.5	cfg	7
2.4.6	dependency graph	8
2.4.7	data dependency graph ddg	9
2.4.8	call graph	9

1 Introduzione (25 feb)

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

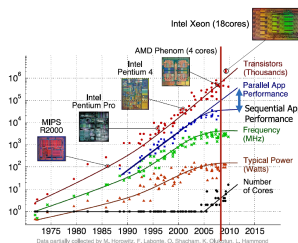
- Il compilatore **traduce un programma sorgente in linguaggio macchina**
- L'ISA agisce da "interfaccia" tra HW e SW (fornisce a SW il set di istruzioni, e specifica a HW che cosa fanno)

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)** 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
```

```
if (100<0)  
{a = 5} → if (false)  
{}
```

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

<pre>x = y; c = 1 + x; d = y + c;</pre>	→	<pre>x = y; c = 1 + y; d = y + c;</pre>	\xrightarrow{DCE}	<pre>c = 1 + y; d = y + c;</pre>
---	---	---	---------------------	----------------------------------

- **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)

<pre>b = 3; c = 1 + b; d = b + c;</pre>	\xrightarrow{CP}	<pre>b = 3; c = 1 + 3; d = 3 + c;</pre>	\xrightarrow{CF}	<pre>b = 3; c = 4; d = 3 + c;</pre>	\xrightarrow{CP}	<pre>b = 3; c = 4; d = 3 + 4;</pre>	\xrightarrow{CF}	<pre>b = 3; c = 4; d = 7;</pre>	\xrightarrow{DCE}	<pre>d = 7;</pre>
---	--------------------	---	--------------------	-------------------------------------	--------------------	-------------------------------------	--------------------	---------------------------------	---------------------	-------------------

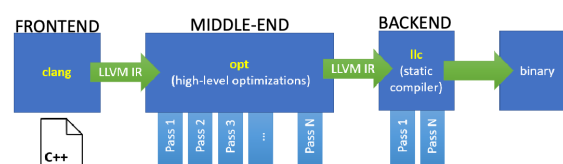
- **LICM (Loop Invariant Code Motion):** si occupa di muovere fuori dai loop tutto il codice **loop invariant**; evita i calcoli ridondanti

<pre>while (i<100) { *p = x/y + i; i = i + 1; }</pre>	→	<pre>t = x + y; while (i < 100) { *p = t + i; i = i + 1; }</pre>
--	---	---

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) → opt (middleend) → 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 (4 mar)

Ricordiamo: middle end come sequenza di passi, di analisi o di trasformazione → per analizzare e trasformare il codice occorre una rappr. 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

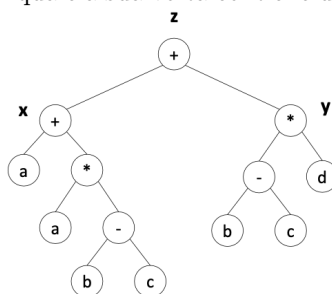
2.4.1 Sintassi concreta (testo)

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

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)

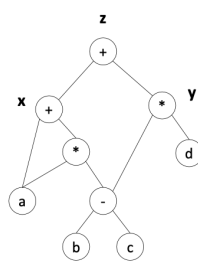
2.4.3 DAG

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

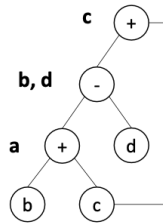
Esempi



```

x = a+a*(b-c);
y = (b-c)*d;
z = x+y;
# espr. trovate
t1 = b-c;
t2 = a*t1;
x = a+t2;
y = t1*d;
z = x+y;

```



```

a = b + c;
b = a - d;
c = b + c;
d = a - d;
# espr. trovate (ERRORE)
a = b + c; # cambia val.
d = a - d;
c = d + c;

```

2.4.4 3AC

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)

esempio:

$x - 2 * y \rightarrow t1 =$

PRO:

- espressioni complesse spezzettate
- forma compatta e simil-assembly
- introdotti registri temporanei intermedi, virtuali e illimitati (non mi preoccupo di problemi architetturali - quanti r. fisici ho a disposizione) (non gestisco eventuali op. di spill - aggiungere load o store in mancanza di r. fisici)

rec

varianti di 3ac

varianti 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 \rightarrow uso l'indice dell'espressione nell'array come "nome" del registro destinazione \rightarrow risparmio spazio, ma diventa più complesso da analizzare (nomi impliciti) e soprattutto da riordinare

ricordiamo la constant propagation: sostituisco usi futuri di una variabile con valore costante con la costante stessa **se non cambia nel frattempo**

\rightarrow ottimizzazione che una IR di tipo 3ac non può applicare immediatamente (devo prima analizzare il resto del codice) \rightarrow SSA come evoluzione della 3ac, in quanto impone che la **definizione (assegnamento)** delle variabili avvenga **solo una volta** (definizioni 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

\rightarrow **in un compilatore coesistono più IR**

Anche per questo ci sono le forme ibride - es. cfg con 3ac

2.4.5 cfg

vediamo un esempio con costrutti condizionali (non solo sequenze lineari)

immagine slide 2-28

un cfg permette di aggiungere informazioni sui **salti** al di sopra di una IR lineare \rightarrow modello il flusso di controllo del programma come grafo composto da blocchi (BB)

ogni bb contiene sequenze lineari di istruzioni

gli archi rappresentano il flusso di controllo del programma
formalmente: un BB è una seq. di istruzioni in forma 3AC

- solo la prima istruzione può essere raggiunta dall'esterno (garantito un singolo entry point)
- 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 s.s.se il secondo può eseguire dopo il primo in qualche percorso del ctrl flow del programma (prima istr del secondo è target dell'istr di salto al termine del primo, oppure il secondo è l'unico successore del primo che non ha un istr di salto come ultima istr - il secondo lo chiamo nodo fallthrough)

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, evidentemente) (situazioni di cfg non normalizzato possono avvenire dopo qualche generico passo di ottimizzazione, evidentemente non le facciamo accadere noi "spontaneamente")

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

esempi ed esercizi fino slide 52

2.4.6 dependency graph

soprattutto nell'era dei multicore questa rappr assume sempre più importanza

almeno due tipi: (recupera questa cosa)

ricordiamo pipeline riscv e data hazard: il fatto che un registro voglia leggere da un registro usato in un'op precedente (e non ancora salvato? recupera)

if id exe mem wb le fasi di pipelining

esempio con una mul: ricordiamo che se la mul successiva prova a usare il registro usato nell'istr prima (dipendenza), il risultato ancora non è stato scritto - nella fase di decode identifico i registri usati nell'operazione, e appunto rw ancora non contiene il risultato aggiornato, che sarà pronto appena tra 2 cicli

questo è un data hazard, nel caso comune si gestisce con una **forwarding unit** che bypassa le fasi successive e inoltra direttamente il risultato appena ottenuto

per quanto la fw unit sia un pezzo di hw dedicato che fa questi controlli autonomamente, in generale l'unico modo per evitare questo tipo di hazard è distanziare abbastanza le istruzioni tra loro affinché il dato sia disponibile → spostare l'istruzione di mul inserendo nop (cicli di stallo) (evidentemente non buono - vado a "rompere" l'IPC paria a 1 della pipeline sempre piena, evidentemente diminuisce la performance)

soluzione migliore: scheduling del programma, ovvero spostare istruzioni che non hanno bisogno di quei registri per riempire quello spazio altrimenti occupato necessariamente da nop

questo è uno dei compiti principali di un backend - **in che ordine genero le istruzioni per massimizzare l'efficienza del programma**

come faccio a fare questa cosa? manualmente: vado a guardare le istruzioni e cerco a mano le dipendenze; il backend sfrutta la IR di tipo dg che fornisce esattamente le informazioni sulle dipendenze tra istruzioni

qui si capisce cosa si intende per instruction level parallelism: dunque, quando è possibile sfruttare tutte le parti di architettura per "parallelizzare il codice" anche in caso di single thread (ottimizzazione senza reale parallelismo)

2.4.7 data dependency graph ddg

Specifico per multicore e parallelismo, usato per dare una rappresentazione tra le dipendenze dei **dati** - tipicamente i loop

per esempio, (loop innestati lavorano tipicamente su str dati complesse e multidimensionali (matrici, immagini, ecc.))

```
for (i = 0, i < n, i++)  
  A[i]=init();
```

Ogni it. resta indipendente dalle altre (uso il solo indice del loop) - basta aggiungere ad es. `A[i-1] = A[i]` per generare dipendenze tra le iterazioni → il loop non è più parallelizzabile, e deve per forza essere eseguito in sequenza

Data la complessità delle casistiche reali, esistono vari modi per rappresentare lo spazio delle iterazioni di un loop e le dipendenze che ne derivano:

all'oggi usiamo 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

2.4.8 call graph

rappresenazione grafica a grafo usata per ragionare sulle relazione tra le funzioni della translation unit del file (insieme delle potenziali chiamate tra funzioni)

rappr. gerarchica, utile soprattutto a livello di analisi **interprocedurale** (la maggior parte delle ottimizzazioni avvengono a livello *intraprocedurale*) (recupera questo concetto) (lavora sui file che abbiamo chiamato translation units, ovvero quelli da cui poi genero i file oggetto)

→ evidente come il compilatore abbia visibilita solo fino a livello dei singoli moduli: posso estendere le ottimizzazioni al massimo fino ai legami tra funzioni dello stesso modulo - le ottimizzazioni piu ampie si spostano a framework di ottimizzazione che agiscono a livello di linker per esempio