Compilatori
Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 4 marzo 2025

Indice

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

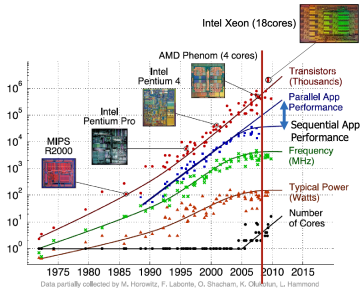
- Il compilatore **traduce un programma sorgente in linguaggio macchina**
- L'ISA agisce da "interfaccia" tra HW e SW (fornisce a SW il set di istruzioni, e specifica a HW che cosa fanno)

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 (e conseguenti problemi di dissipazione del calore prodotto) → nasce l'idea di cambiare completamente il paradigma di sviluppo di un processore: dal singolo core sempre più potente passo a **più core "isopotenti"** sullo stesso chip



Notiamo come dal 2005 circa si raggiunge un plateau in termini di consumo, di frequenza e di performance di programmi *sequenziali*, e aumenta la performance di programmi che **sfruttano la parallelizzazione** → i programmi devono essere "consapevoli" che il processore è multicore! Qui capiamo l'impatto dello sviluppo tecnologico sull'evoluzione dei compilatori

Per evitare un cambio nella formazione dei programmatori, le aziende sperano nei compilatori autoparallelizzanti - non sarà mai possibile e nel mentre sono stati introdotti molti paradigmi di pr. parallela

Il compilatore mantiene un ruolo fondamentale: oltre a rendere meno "traumatico" il passaggio alla programmazione parallela, 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 alcune 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:

$$\text{Performance} = \frac{1}{\text{Execution Time}} \qquad \text{Execution Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Frequency}}$$

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`

1.2.1 Esempi di ottimizzazione

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

- AS (Algebraic Semplification): 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 (dobbiamo ovviamente tenere conto delle implementazioni hw delle operazioni): esempi a slide 30
  esempio sofisticato: for con operazioni su array, sostituito da operazioni su puntatori (aritmetica dei pt.) → il risultato si vede nel codice assembly (riporta listings)
- CSE (Common Subexpression Elimination): elimino i calcoli ridondanti di una stessa espressione riutilizzata in più istruzioni (statement)
- 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
- 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) (slide 38 esempio di sequenza di ottimizzazioni applicate in sequenza) (stiamo sempre ipotizzando che i valori a fine esempi siano poi **utilizzati**, 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

```
while (i<100) {
  *p = x/y + i;
  i = i + 1;
}
```

diventa

```
t = x + y;
while (i < 100) {
  *p = t + i;
  i = i + 1;
}
```

recupera questione su load e store, questione su CPI medio per una load e quando si riduce/ arriva a 1 (quando si trova in cache, di solito ordine delle decine)
chiedi a dani

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
- sono in generale target delle ottimizzazioni, essendo centrali per il parallelismo

1.3 Anatomia di un compilatore

Un compilatore svolge almeno due compiti: analisi del sorgente e sintesi di un programma in linguaggio macchina; lo fa operando su una rappresentazione intermedia (IR) che si interpone tra frontend e backend, e tra source code e target code

il blocco di middle-end agisce sul codice intermedio, e in vari passaggi lo trasforma e lo ottimizza (diverso a seconda del compilatore)

caso llvm: clang (frontend) → opt (middleend) → llc (backend)

l'ottimizzatore 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 ottimizzazione)

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`: esecuzione dei passi in un ordine 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

slide 49-51

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:

- facilita di generazione (effetti sul frontend)
- facilita e costo di manipolazione
- livello di astrazione e di dettaglio esposto: effetti sul frontend e sul backend (diverse IR da un lato e dall'altro, a seconda del livello di astraz. e di dettaglio necessari)

2.2 Tipi di IR

- AST (abstract syntax tree)
- DAG (grafi diretti aciclici)
- 3-address code (3AC): assomiglia molto all'assembly (3 indirizzi in quanto registro destinazione e max2 operandi)
- SSA (Static Single Assignment): evoluzione di 3ac con ulteriori proprietà di control flow (?)
- CFG (control flow graphs)
- CG (call graph)
- PDG (program dependence graphs)

call graph: rappresenta "come" vengono chiamate le funzioni (a partire dal main)
ottimizzazioni 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)
pdg: rappresentazione fondamentale per lavorare sul parallelismo, multithreading eccetera

2.3 categorie di IR

- grafiche (o strutturali)
  - orientate ai grafi
  - molto usate nella source-to-source translation: uso tipico - ottimizzazioni che non hanno bisogno della struttura sofisticata di un middle-end (es. openMP: sono di fatto delle annotazioni sul codice, dunque lo scopo principale di questo programming model è fornire uno strumento semplice al programmatore (es. outlining: prendo es un loop e lo impacchetto in una funzione che poi dovra essere eseguita dai thread per la parallelizzazione) - dunque in questo caso non sto ottimizzando nel senso proprio del termine, ma sto trasformando il codice e lo sto rendendo eseguibile in maniera parallela - mi bastsa un IR di questa categoria)
  - solitamente voluminose (basate su grafi) - tradeoff con il fatto che sono usate prima del middle-end (senza doverlo coinvolgere)
  - es. ast, dag
- lineari
  - pseudocodice per macchine astratte
  - livello di astrazione vario
  - strutture dati semplici e compatte
  - facile da riarrangiare (evidentemente il piu 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)

maniera piu semplice in quanto piu vicina al livello di astrazione "umano" di ragionamento sul programma, ma non il livello corretto per ottimizzare ne 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, con il nodo radice che rappresenta il **programma**, il quale a sua volta contiene un blocco di istruzioni dal quale discendono tanti figli quante le sue istruzioni

esempi immagini

pro: molto comodo per interpreti (basta usare 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 (digressione)

contrazione di ast che evita la duplicazione di espressioni → rappresentazione piu compatta

limite: il riuso e possibile solamente dimostrando che il suo valore non cambia nel programma → essendo assegnamenti e chiamate frequentissimi, evidentemente 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

Esempio

codice generato da questo dag corretto? corretta, mentre per altro esempio no

2.4.4 3ac

evidentemente adatto alle ottimizzazioni: tutte le istruzioni del programma vengono spezzettate in istruzioni di forma semplice simile all'assembly

in questo caso istr di tipo `x = y op z` (1 operatore, massimo 3 operandi)

esempio:

`x - 2 * y` → `t1 =`

pro: espressioni complesse spezzettate, forma compatta e simile a assembly introdotti registri temporanei intermedi, virtuali e illimitati (non mi preoccupo dei problemi architetturali di quanti registri fisici ho a disposizione, sono a livello decisamente piu alto - le eventuali op. di spill (aggiungere load o store in mncanza di registri fisici) non le gestisco qui)

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 → uso l'indice dell'espressione nell'array come "nome" del registro destinazione → risparmio spazio, ma diventa piu 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**

→ ottimizzazione che una IR di tipo 3ac non puo applicare immediatamente (devo prima analizzare il resto del codice) → 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

2.5 scelta della IR

dipende ovviamente dal livello di dettaglio necessario per ogni specifico compito - in un compilatore coesistono piu IR

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

2.5.1 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 → 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 puo 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 piu grandi, ma le piu piccole di questo tipo sono i BB)

un arco connette due nodi s.s.se il secondo puo 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 falltrough)

un cfg **normalizzato** ha i bb **massimali** (non possono essere resi piu 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

2.5.2 dependency graph

soprattutto nell'erad ei nulticore questa rappr assume sempre piu 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 e stato scritto - nella fase di decode identifico i registri usati nell'operazione, e appunto rw ancora non contiene il risultato aggiornato, che sara pronto appena tra 2 cicli

questo e 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 affinche il dato sia disponibile → sposto 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 sposto 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)

2.5.3 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, eccetera)

esempio: for i = 0, i < n, i++ A[i]=init()

in questo caso è evidente come ogni iterazione resta indipendente dalle altre (uso il solo indice del loop)

sufficiente aggiungere qualcosa come A[i-1] = A[i] per generare dipendenze tra le iterazioni

di fatto mi sta dicendo che questo loop non è parallelizzabile in nessun modo, e deve per forza essere eseguito in sequenza

evidentemente le casistiche reali sono molto più complesse → esistono vari modi per rappresentare lo spazio delle iterazioni di un loop e le dipendenze che ne derivano

modello usato all'oggi: 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

2.5.4 call graph

rappresetnazione 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