Compilatori
Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 26 febbraio 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

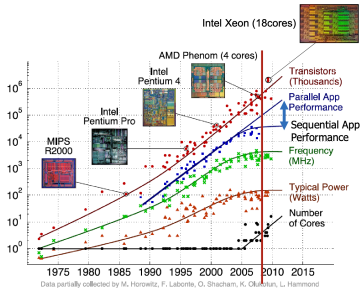
- Il compilatore **traduce un programma sorgente in linguaggio macchina**
- L'ISA agisce da "interfaccia" tra HW e SW (fornisce a SW il set di istruzioni, e specifica a HW che cosa fanno)

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