Dall'ISA ai circuiti

Le prestazioni di un calcolatore

Patterson: 1.6

Introduzione

- Non ci basta sapere che in teoria la soluzione ad un problema esiste: spesso noi vogliamo la soluzione nel minor tempo possibile.
  - A nessuno infatti interessa quanto velocemente viene calcolato un risultato sbagliato
- Le prestazioni sono un fattore critico per la creazione di software (o piattaforma HW-SW) di successo
  - PS5 o XBOX?
    FLOPS, giochi disponibili ed esclusivi, retrocompatibilità, costo acquisto ed abbonamento…
- Prestazione è quindi un termine ambiguo
  - In un sistema embedded è importante considerare anche lo spazio necessario a memorizzare il programma
- In ogni caso le prestazioni ottenibili dipendono in massima parte da quanto conosciamo l'hardware e dagli strumenti che utilizziamo.

Esempio: aeroplano con le prestazioni migliori

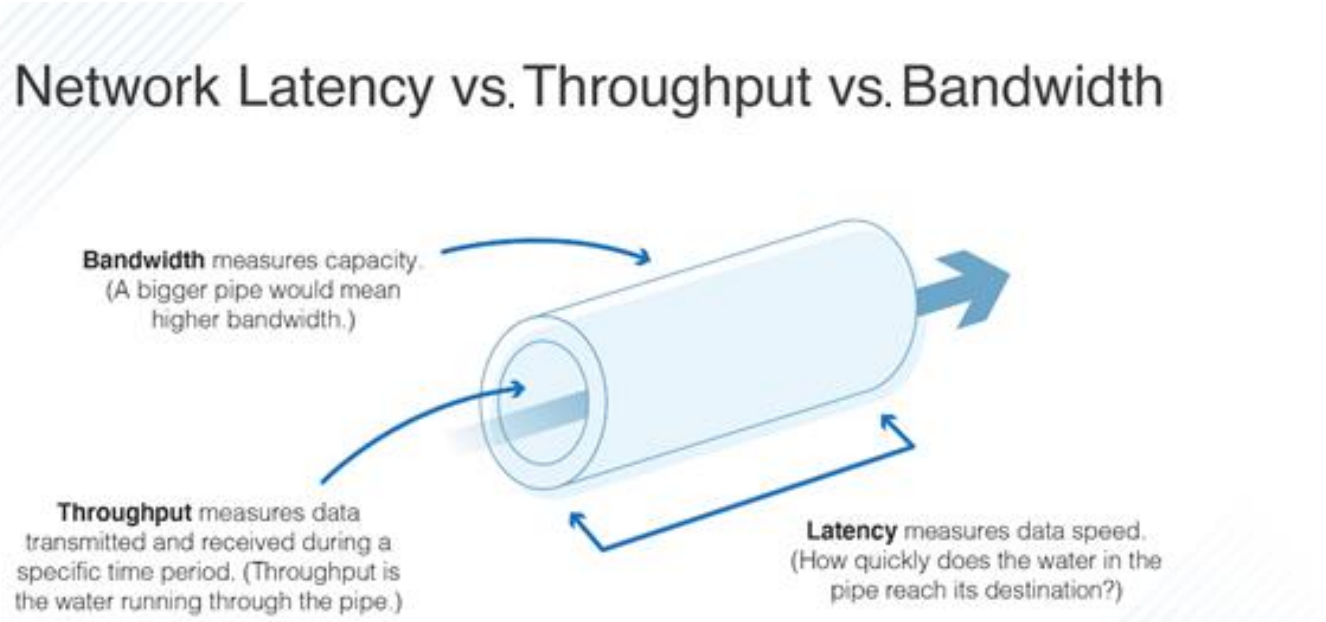
- Dipende tutto da cosa intendiamo per prestazioni

Tipo di aeroplano	Capacità di trasporto (numero di passeggeri)	Autonomia di volo (km)	Velocità di crociera (km/h)	Portata (passeggeri × km/h)
Boeing 777	375	7400	980	367 500
Boeing 747	470	6640	980	460 600
BAC/Sud Concorde	132	6400	2160	285 120
Douglas DC-8-50	146	13 950	870	127 020

Figura 1.14 Capacità, autonomia di volo e velocità di alcuni aeroplani commerciali. L'ultima colonna mostra l'efficienza con la quale l'aeroplano trasporta i passeggeri, cioè la capacità moltiplicata per la velocità di crociera (non vengono considerati i tempi di decollo e atterraggio).

Quindi?

- Tempo di esecuzione: quanti secondi sono necessari al programma A per essere eseguito sul calcolatore B
  - Altresì detto tempo di risposta, wall-clock time…
  - Non è solo una questione HW!
    - Compilatore, e.g. gcc vs icc vs nvcc vs clang…
    - Librerie….
- Throughput: numero di task eseguiti nell'unità di tempo
  - Importante in un centro di calcolo / fornitore di servizi



Esercizio

Throughput e tempo di esecuzione

Le seguenti modifiche apportate a un calcolatore aumentano il throughput, diminuiscono il tempo di esecuzione o entrambi?

1. Sostituire il processore del calcolatore con una versione più veloce.
2. Aggiungere processori addizionali a un sistema che utilizza più processori per task separati, per esempio per la ricerca sul web.

Throughput e tempo di esecuzione

Le seguenti modifiche apportate a un calcolatore aumentano il throughput, diminuiscono il tempo di esecuzione o entrambi?

1. Sostituire il processore del calcolatore con una versione più veloce.
2. Aggiungere processori addizionali a un sistema che utilizza più processori per task separati, per esempio per la ricerca sul web.

Diminuire il tempo di esecuzione di un sistema significa quasi sempre aumentare il throughput. Quindi nel caso 1 migliorano sia il tempo di esecuzione sia il throughput. Nel caso 2 nessun task viene eseguito più velocemente e di conseguenza aumenta soltanto il throughput.

Tuttavia, nel secondo caso, se le richieste di utilizzo del calcolatore fossero vicine al suo throughput, il sistema potrebbe doverle accodare. In questo caso, migliorare il throughput potrebbe migliorare anche il tempo di esecuzione, poiché ridurrebbe il tempo di attesa dei task in coda. Quindi, in molti calcolatori reali, modificare il tempo di esecuzione spesso influenza il throughput e viceversa.

Definizioni

- Le prestazioni sono in unità di "oggetti-per-secondo"
  - "grande" è meglio
- Se siamo soprattutto interessati al tempo di risposta
  - $P_X = \frac{1}{E_X}$
  - P_X = prestazioni della macchina X
  - E_X = tempo di esecuzione della macchina X
- Speed up rispetto a una macchina di riferimento
  - $S_{XY} = \frac{P_X}{P_Y} = \frac{E_Y}{E_X}$
  - Lo speed up di X rispetto a Y dice "X è S_{XY} volte più veloce di Y"
  - In percentuali, $s_{xy}=(S_{XY}-1)*100$ dice "X è s_{xy}% più veloce di Y"
- Esempio:
  - E_X = 1sec, E_Y = 1.2sec $\Rightarrow$ $S_{XY} = 1.2 \Rightarrow$ X è 1.2 volte più veloce di Y
    $s_{xy} = 20\% \Rightarrow$ X è il 20% più veloce di Y

Tempo Totale e Tempo di CPU

- T_J = Tempo Totale
  (o Tempo Assoluto,di Risposta, Trascorso, o Elapsed Time)
  - tempo necessario per completare un lavoro; comprende una serie di tempi che sono "allungati" dalle interferenze di altri programmi che generano attività come:
    - accesso ai dischi
    - accesso alla memoria
    - attività di I/O
    - sovraccarico del Sistema Operativo
- T_{CPU} = Tempo di CPU (Tempo di Esecuzione del programma)
  - Tempo durante il quale un processore ha lavorato su un singolo programma
    - senza tempo speso in I/O
    - senza tempo per eseguire altri programmi
  - Può essere suddiviso in
    - T_U = tempo di CPU relativo all'utente
    - T_K = tempo di CPU relativo al Sistema Operativo (K=Kernel)

Tempo di CPU misurato in UNIX/LINUX

- Dato il programma `myprogram`

```
time <myprogram>

81.3sec   10.6sec   2:19min   66%
```

T_U

T_K

T_J

$$\frac{T_{CPU}}{T_J} \% = \frac{(T_U+T_K)}{T_J} * 100$$

Per il 34 % la macchina è usata
- per I/O
- per altri programmi
- per altra attività di sistema

$T_U >> T_{CPU}$ se uso più thread

```
[dadagostino@hpcocapie11 vect]$ time ./MatVector

ROW:101 COL: 102
Execution time is 4.159 seconds
GigaFlops = 4.905222
Sum of result = 195853.999899

real    0m4.192s
user    0m4.163s
sys     0m0.012s
```

```
[dadagostino@hpcocapie11 openmp]$ time ./mandelbrot_par
Area 1.512079, error 0.001512, executed in 0.624283 sec.

real    0m0.650s
user    0m16.911s
sys     0m0.391s
```

by 32 THREADS, (i.e. ≈0 .65 *32)

https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1

Equazione delle prestazioni

$$T_{CPU} = C_{CPU} \cdot T_C = \frac{C_{CPU}}{fc} = \boxed{\frac{N_{CPU} \cdot \overline{CPI}}{fc}}$$

- T_C è il periodo di clock
- f_C è la frequenza di clock == 1/ T_C
- C_{CPU} sono i cicli di clock
- N_{CPU} è il Numero di istruzioni eseguite DINAMICAMENTE
- $\overline{CPI}$ è il numero di Cicli Per Istruzione (medio)

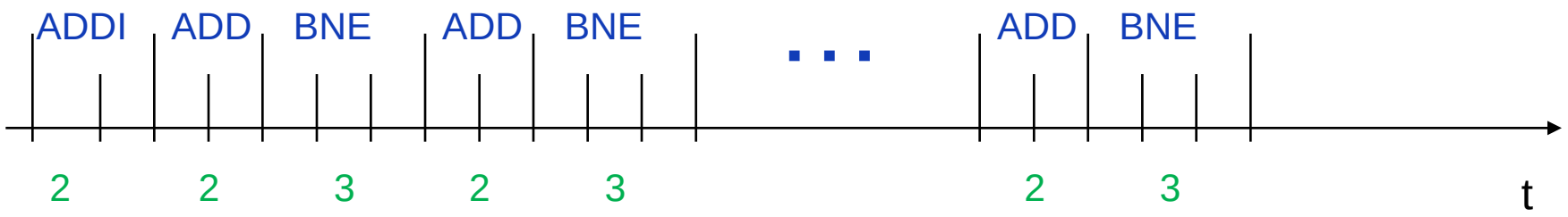
Esempio:

bne → 2 cicli
add → 1 ciclo
2 istruzioni, 3 cicli → /CPI=3/2=1.5

Esempio 1

Assumiamo t0=0, t1=0 :

```
        addi t1, t1, 100
loop:   add  t0, t0, 1
        bne  t0, t1, loop
```



C_{CPU}=502, N_{CPU}=201

CPI= 502/201 ≈ 2.5

- **Per ottenere maggiori prestazioni si può:**
  - **ridurre i cicli di clock per un programma**
  - **aumentare la frequenza di clock**
- **Se aumento la frequenza potrebbe però verificarsi che il sottosistema di memoria 'non tiene il passo del processore', quindi il CPI potrebbe aumentare…**

Esempio 2

- $T_{CPU}(A)$ = 10s $f_C(A)$ = 400 MHz ← MACCHINA A
- $T_{CPU}(B)$ = 6s $f_C(B)$ = ? ← MACCHINA B

Supponiamo che in B si riesca a ridurre il tempo ma con il risultato di avere

$$C_{CPU}(B) = 1.2 * C_{CPU}(A)$$

- Quanto deve essere la frequenza di clock della macchina B affinché il tempo di esecuzione del programma sia 6s?

- Soluzione:

$$C_{CPU}(A) = (f_C * T_{CPU})|_A = 4 * 10^9$$

$$f_C(B) = (C_{CPU} / T_{CPU})|_B = (1.2 * 4 * 10^9 / 6) = 800 \text{ MHz}$$

- Per ottenere una diminuzione del 40% del tempo di esecuzione è necessario raddoppiare la frequenza di clock

Esempio 3

- $f_C(A)$ = 1 GHz =1000 MHz $f_C(B)$ = 500 MHz
- $\overline{CPI(A)}$ = 2.5 $\overline{CPI(B)}$ = 1.2
- Quale calcolatore è più veloce?
  (il set di istruzioni rimane lo stesso)

- $S_{AB} = \frac{T(B)}{T(A)} = \frac{N(B)*CPI(B)*f_C(A)}{N(A)*CPI(A)*f_C(B)} = \frac{1.2*1000}{2.5*500}$
- set istruzioni uguale → $N_{CPU}(A) = N_{CPU}(B)$
- S_{AB}= 0.9 → B è 1.1 volte più veloce di *A*

Componenti base del tempo di esecuzione

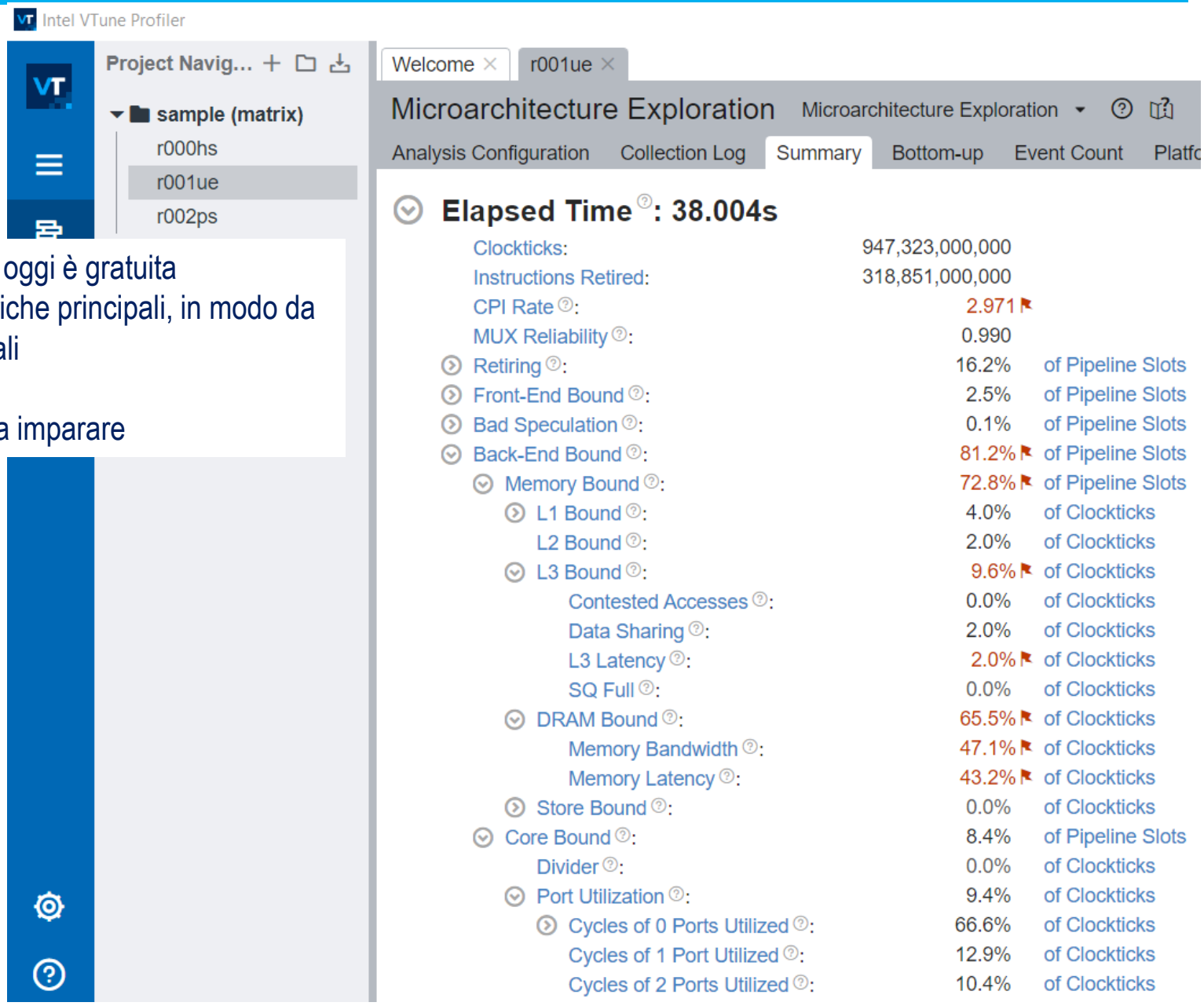
- Il tempo di esecuzione si misura (in secondi, ore…)
- Il numero di istruzioni tramite tool sw come un profiler
  NOTA: può cambiare in funzione dell'input (if..switch..for)
- Il CPI cambia anch'esso a seconda dell'input ma soprattutto del programma.
  - In realtà parleremo di IPC quando vedremo effettivamente come funziona il processore, in quanto CPI può essere < 1.
- NON PRENDETE un sottoinsieme! Le tre misure vanno considerate insieme.

Componente delle prestazioni	Unità di misura
Tempo di esecuzione della CPU per un dato programma	Secondi per programma
Numero di istruzioni	Istruzioni eseguite per singolo programma
Cicli di clock per istruzione (CPI)	Numero medio di cicli di clock per istruzione
Durata del ciclo di clock	Secondi per ciclo di clock

Esempio di profiler

Intel® VTune™ Profiler
Find and Fix Performance Bottlenecks Quickly and Realize All the Value of Your Hardware

- La maggior parte di questi tool oggi è gratuita
- Nel corso introduciamo le metriche principali, in modo da saper leggere i risultati principali
- I dettagli tra 5 anni
  - Per ora c'è molto altro da imparare



MIPS - Million Instructions per Second

- Sembra una buona idea misurare le prestazioni di un'architettura dividendo il numero di istruzioni per il tempo di esecuzione (equivale a frequenza / (CPI $*10^6$))
  - Più è performante un processore più alto avrà il MIPS
- MA

1. Due architetture diverse potrebbero ottenere lo stesso risultato con un numero differente di istruzioni
   - Se eseguo più istruzioni ma più veloci ho un valore più alto a parità di tempo di esecuzione
2. Il MIPS continua ad essere legato al programma (e all'input)

Misura	Calcolatore A	Calcolatore B
Numero di istruzioni	10 miliardi	8 miliardi
Frequenza di clock	4 GHz	4 GHz
CPI	1,0	1,1

a. Quale calcolatore ha il più alto numero di MIPS? A, 4k vs 3636
b. Quale calcolatore è più veloce? B, 2.2 vs 2.5

E quindi?

Potete affermare quale **processore** (non sistema) sia più veloce nel caso di

- Confronto banale: processore del 1995 vs 2022
- Due CPU della stessa (micro)architettura

Altrimenti?

- Benchmark

Non c'è benchmark migliore della propria applicazione determinato quale sia l'input più comune.

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610

https://netlib.org/utk/people/JackDongarra/faq-linpack.html

Giusto pochi anni fa…

ACM Turing Award Honors Jack Dongarra for Pioneering Concepts and Methods Which Resulted in World-Changing Computations

Dongarra's Algorithms and Software Fueled the Growth of High-Performance Computing and Had Significant Impacts in Many Areas of Computational Science from AI to Computer Graphics

ACM named Jack J. Dongarra recipient of the 2021 ACM A.M. Turing Award for pioneering contributions to numerical algorithms and libraries that enabled high performance computational software to keep pace with exponential hardware improvements for over four decades. Dongarra is a University Distinguished Professor of Computer Science in the Electrical Engineering and Computer Science Department at the University of Tennessee. He also holds appointments with Oak Ridge National Laboratory and the University of Manchester.

The ACM A.M. Turing Award, often referred to as the "Nobel Prize of Computing," carries a $1 million prize, with financial support provided by Google, Inc. It is named for Alan M. Turing, the British mathematician who articulated the mathematical foundation and limits of computing.

Dongarra has led the world of high-performance computing through his contributions to efficient numerical algorithms for linear algebra operations, parallel computing programming mechanisms, and performance evaluation tools. For nearly forty years, Moore's Law produced exponential growth in hardware performance. During that same time, while most software failed to keep pace with these hardware advances, high performance numerical software did—in large part due to Dongarra's algorithms, optimization techniques, and production-quality software implementations.

2021 ACM A.M. Turing Award Laureate

https://awards.acm.org/about/2021-turing

Patterson, David and Hennessy, John L (2017^); Berners-Lee, Tim (2016); Dijkstra, Edsger Wybe (1972); Thompson, Kenneth Lane and Ritchie, Dennis M. (1983^); Wirth, Niklaus E (1984)

Tipi di Benchmark

Vantaggi		Svantaggi
• rappresentativo	**WORKLOAD (carico effettivo)** insieme di programmi abitualmente usati su un dato calcolatore	• molto specifici • non portabili • difficili da eseguire, o da misurare • difficile identificare cause
• portabilità • ampiamente usati • catturano i miglioramenti	**BENCHMARK SUITE** Insieme di programmi campione che si spera abbiano un comportamento simile al proprio workload (e.g. SPEC2000, TPC)	• meno rappresentativo
• facili da eseguire, usati nelle prime fasi di progetto	**SMALL-KERNEL** programma “giocattolo” usato per le fasi di testing di prototipi (e.g. Lawrence Livermore Loops, Linpack)	• facili da fuorviare
• identificare prestazioni di picco e colli di bottiglia	**MICRO-BENCHMARKS** sequenze di istruzioni ritenute significative (e.g. REPS MOVE)	• il “picco” può essere distante dalle prestazioni delle reali applicazioni

Prestazioni

Componente hardware o software	Che cosa influenza?	Come?
Algoritmo	Numero di istruzioni, eventualmente il CPI	L'algoritmo determina il numero di istruzioni del programma sorgente e quindi il numero di istruzioni in linguaggio macchina che vengono eseguite dal processore. L'algoritmo può anche influenzare il CPI, favorendo l'utilizzo di istruzioni più o meno veloci. Per esempio, se l'algoritmo utilizza più operazioni di divisione, tenderà ad avere un CPI più elevato.
Linguaggio di programma-zione	Numero di istruzioni, CPI	Il linguaggio di programmazione influenza certamente il numero di istruzioni, dal momento che i costrutti del linguaggio ad alto livello sono tradotti in istruzioni in linguaggio macchina e queste determinano il numero di istruzioni eseguite. Il linguaggio ad alto livello può anche influenzare il CPI a seconda delle sue caratteristiche; per esempio un linguaggio con un esteso supporto per i dati astratti (per esempio Java) richiederà chiamate indirette a funzione, che sono caratterizzate da un CPI più alto.
Compilatore	Numero di istruzioni, CPI	L'efficienza del compilatore influenza sia il numero di istruzioni sia il numero medio di cicli per istruzione, dal momento che il compilatore traduce le istruzioni dal linguaggio sorgente ad alto livello nelle istruzioni in linguaggio macchina. Il ruolo del compilatore può essere molto complesso e può influenzare il CPI in maniera complicata.
Architettura dell'insieme delle istruzioni	Numero delle istruzioni, frequenza di clock, CPI	L'architettura dell'insieme di istruzioni influenza tutti e tre i fattori delle prestazioni della CPU, dato che influenza le istruzioni richieste da una data funzione, il costo in numero di cicli di ogni istruzione e la frequenza del processore.

Fondamentale

Corsi di Algoritmi e Strutture dati e seguenti.

Non avrai altro linguaggio per le prestazioni all'infuori del C++

Non esiste solo il g++

Ne parliamo qui.

Prestazioni - Riassunto

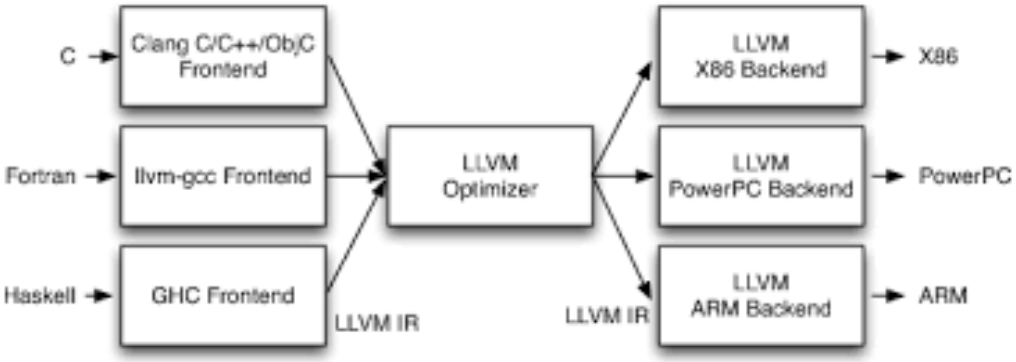
- Buone prestazioni hardware = efficacia dell'intero sistema
- Difficoltà a misurare le prestazioni
  - complessità dei moderni sistemi software
  - peculiarità hardware non sempre ideali in qualsiasi situazione
- Non basta il manuale del Set di Istruzioni e un insieme significativo di applicazioni
  - le metriche possono variare a seconda del tipo di applicazione
- Le prestazioni sono un parametro decisionale per l'acquisto di un calcolatore
  - i produttori e i venditori hanno interesse a mettere in luce gli aspetti più convenienti per loro
- Quali elementi di un calcolatore influiscono sulla metrica di interesse
  - CPU, Memoria, I/O
- Nostro obiettivo: capire le implicazioni su costi e prestazioni di determinate scelte architetturali

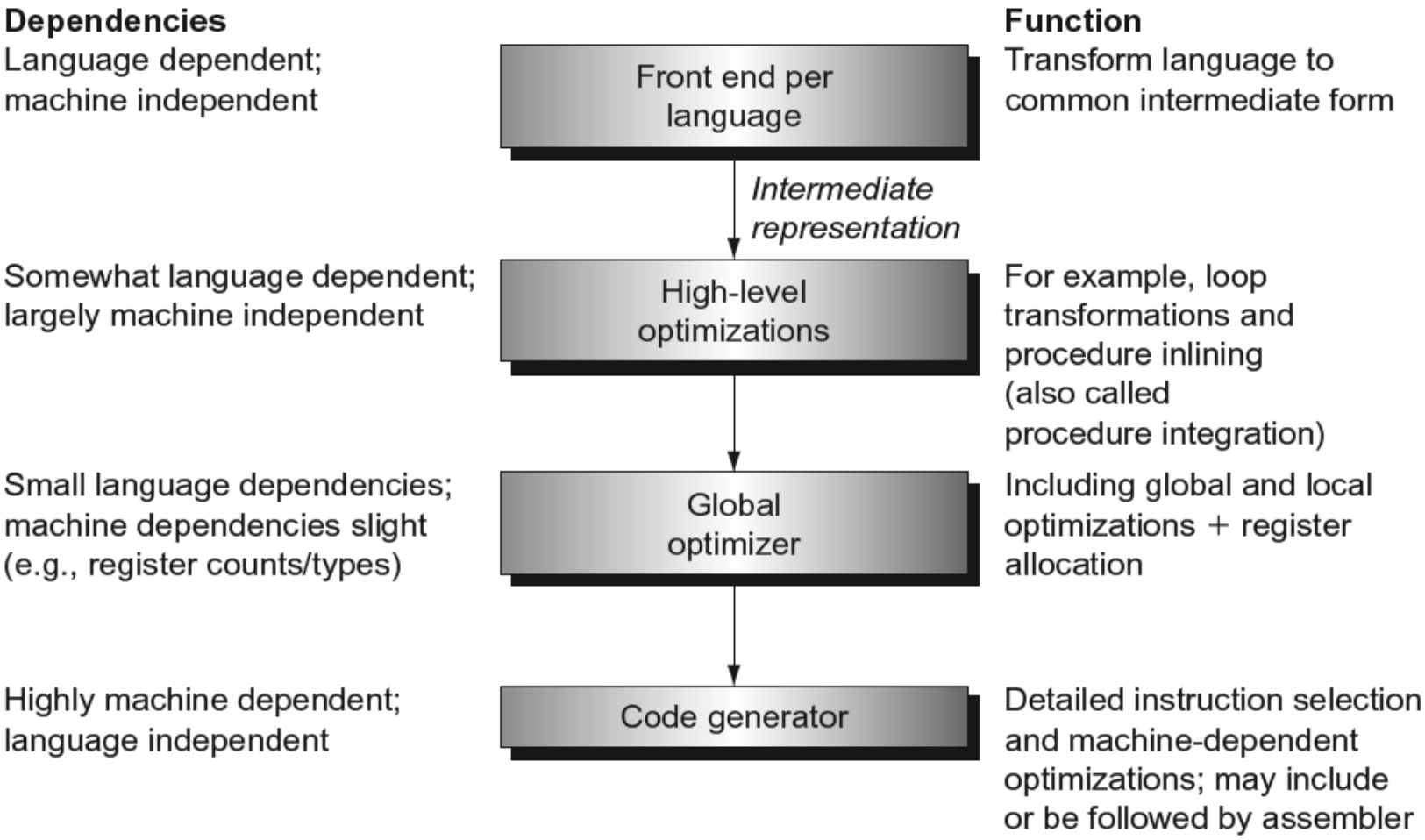
Il compilatore, colui che determina N_{CPU}

Il paragrafo 2.15 va scaricato dal sito

Anatomia di un compilatore

- NOTA. La figura mostra la struttura logica.
- Fino a poco tempo fa però ogni compilatore era un sw monolitico
- LLVM ha cambiato le cose



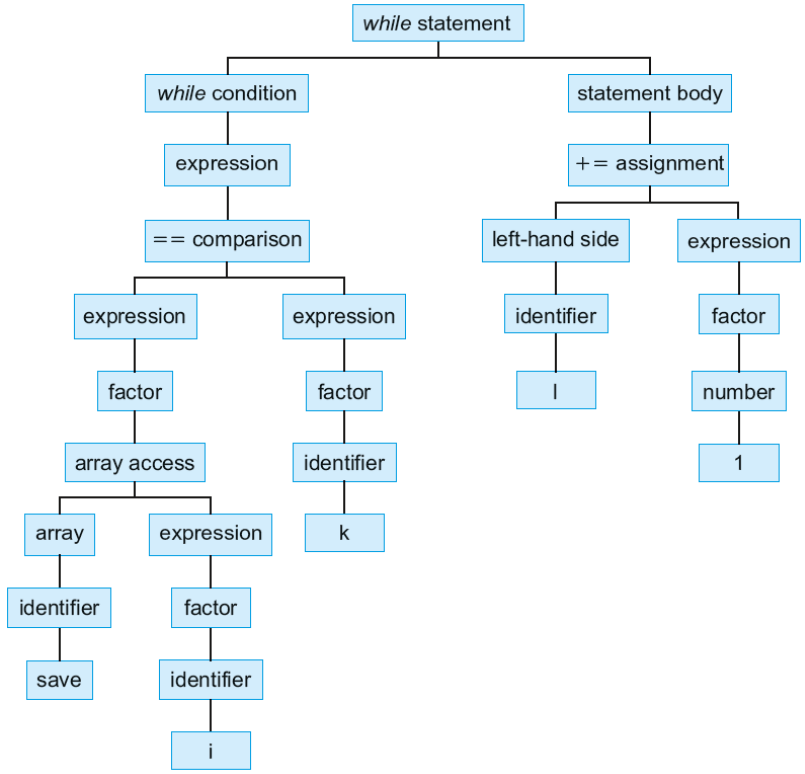


Front-end

Dal codice produce una rappresentazione intermedia language independent utilizzando un abstract sintax tree

- Legge carattere per carattere e crea una sequenza di token
- Se la sintassi è corretta crea il sintax tree
- Analizza la semantica
  - Variabili dichiarate
  - Type cheking
  - Crea una symbol table
- Genera la IR (rappresentazione intermedia)
  - Molto simile all'assembler del Risc-V (vedi LLVM Ref)
  - Un altro esempio di IR è il bytecode di Java
    vedi Linguaggi e programmazione OO dell'anno prossimo
  - Numero INFINITO di registri

```
while (save[i] == k)
  i += 1;
```



Ottimizzazioni High-Level

- **Procedure inlining**: sostituisce tutte le chiamate a funzione con il corpo della funzione.
  - Evito salti, salvataggi dello stato, creazione dei record di attivazione mantenendo tutte le proprietà di modularità, riuso… che avete appreso (sostituisce lui, non voi)
- **Loop unrolling**: in generale riduce l'overhead, essenziale per vettorizzazione

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

```
# s0 = i, s1 = sum
      addi  s1, zero, 0      # sum = 0
      addi  s0, zero, 0      # i = 0
      addi  t0, zero, 10     # t0 = 10
for:  bge   s0, t0, done     # i >= 10?
      add   s1, s1, s0       # sum = sum + i
      addi  s0, s0, 1        # i = i + 1
      j     for              # repeat loop
done:
```

- Swap di loop annidati per ottimizzare l'accesso in memoria tramite cache

```
for (loop=0; loop<10; loop++) {
  for (i=0; i<N; i++) {
    ... = ... x[i] ...
  }
}
```

```
for (i=0; i<N; i++) {
  for (loop=0; loop<10; loop++) {
    ... = ... x[i] ...
  }
}
```

R Subsequent use of x[i] are spaced too far for large N

Memory accesses are reduced to 1/10

Ottimizzazioni locali e globali

- La differenza è nel fatto di considerare uno o più BASIC BLOCK
  - Sono le sequenze di istruzioni senza salti, in cui le istruzioni sono eseguite tutte, una alla volta ed in sequenza
- Solo a livello globale considero l'effettiva allocazione dei registri

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

```
while (save[i] == k)
  i += 1;
```

FIGURE e2.15.3 The *while* loop example is shown using a typical intermediate representation. In practice, the names `save`, `i`, and `k` would be replaced by some sort of address, such as a reference to either the local stack pointer or a global pointer, and an offset, similar to the way `save[i]` is accessed. Note that the format of the RISC-V instructions is different from the rest of the chapter, because they represent intermediate representations here using `rXX` notation for virtual registers.

Ottimizzazioni locali

- Elimino le sottoespressioni comuni - es x[i] calcolato una sola volta
- Semplifico le operazioni dove possibile - es mul con slli
- Identifico e propago costanti e copie di valori
- Elimino codice inutile o mai eseguito - es in if(true) non ha senso else
- Sposto codice – es se loop invariant … e altre ottimizzazioni

```
// x[i] + 4
la r100,x
ld r101,i
mul r102,r101,8
add r103,r100,r102
ld r104, 0(r103)
//
addi r105, r104,4
la r106,x
ld r107,i
mul r108,r107,8
add r109,r106,r107
sd r105,0(r109)
```

```
// x[i] + 4
la r100,x
ld r101,i
slli r102,r101,3
add r103,r100,r102
ld r104, 0(r103)
// value of x[i] is in r104
addi r105, r104,4
sd r105, 0(r103)
```

x[i] = x[i] + 4

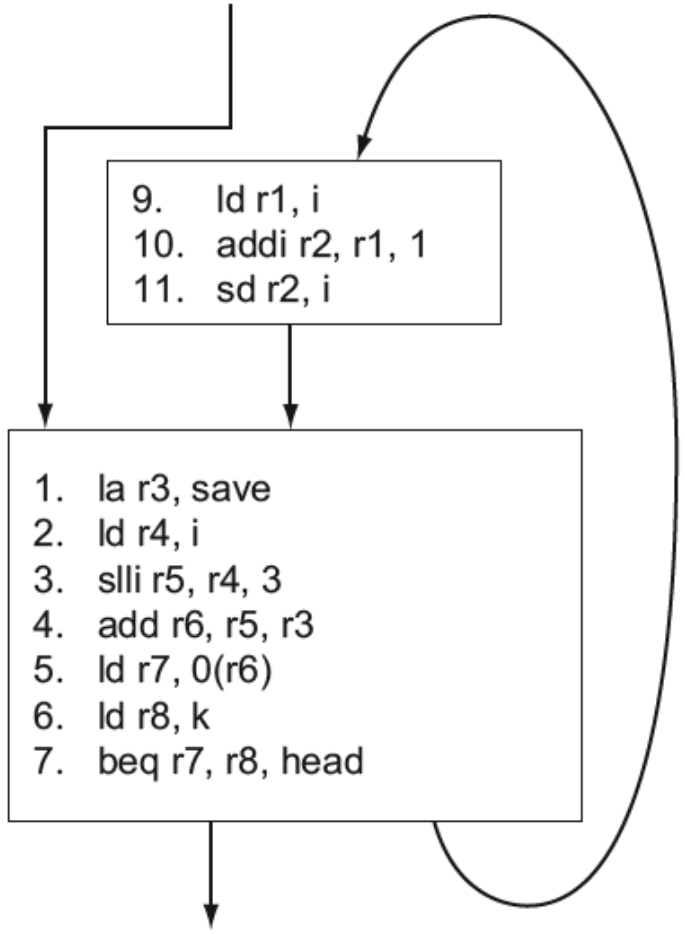
Esempio completo

```
while (save[i] == k)
  i += 1;
```

```
loop:
   # comments are written like this--source code often included
   # while (save[i] == k)
   la   r100, save          # r100 = &save[0]
   ld   r101, i
   li   r102, 8
   mul  r103, r101, r102
   add  r104, r103, r100
   ld   r105, 0(r104)       # r105 = save[i]
   ld   r106, k
   bne  r105, r106, exit
   # i += 1
   ld   r106, i
   addi r107, r106, i       # increment
   sd   r107, i
   j    loop                # next iteration
exit:
```

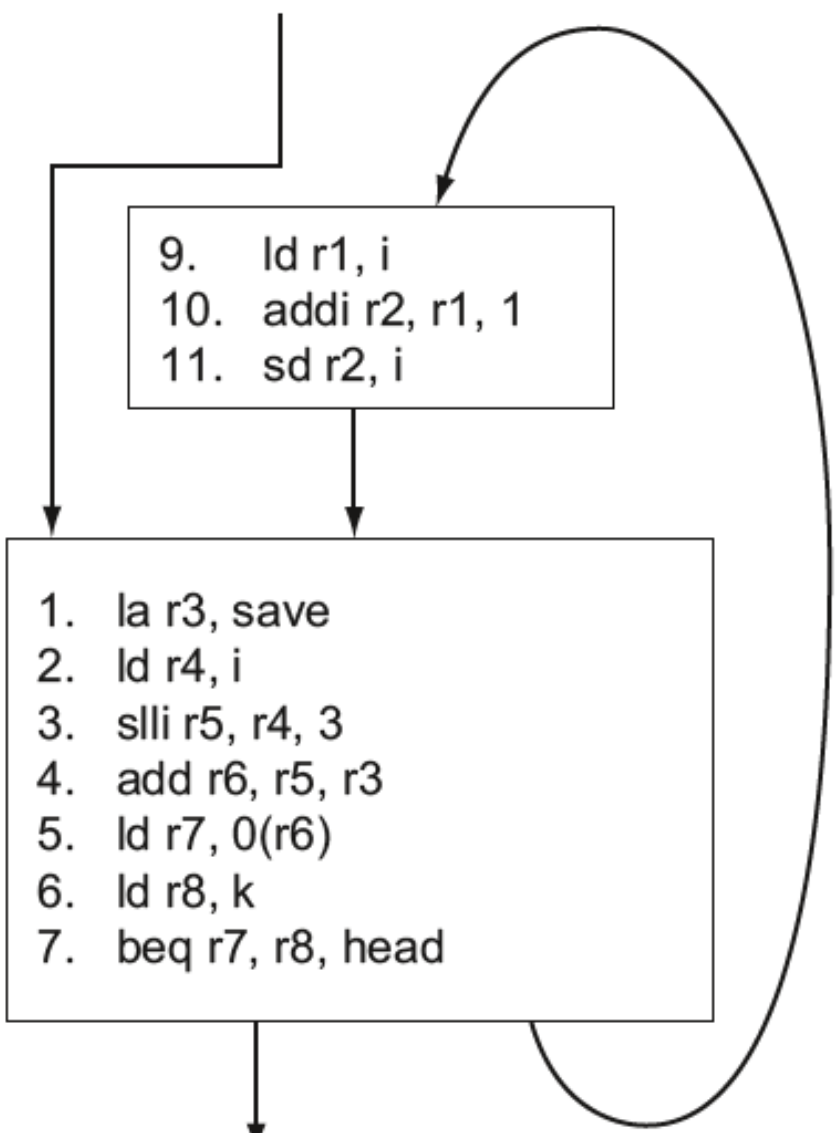
3

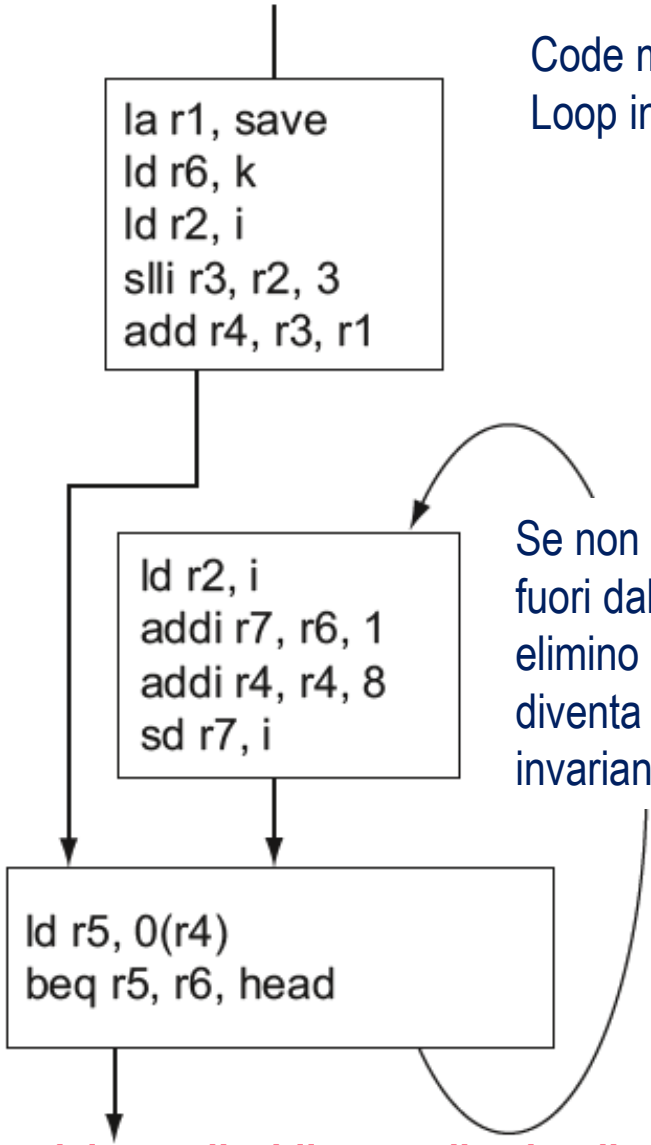
7



Esempio completo

```
while (save[i] == k)
  i += 1;
```





Code motion
Loop invariant

Se non uso i fuori dal while elimino ld e sd: diventa loop invariant

Ma se non uso i al di fuori del while potrei eliminare del tutto il while… codice inutile

Infine

- Poter allocare i registri in modo da minimizzare ld e sd è una delle ottimizzazioni più importanti.
- Divido il codice in regioni, sezioni di codice dove una variabile può rimanere in un registro
- Processo iterativo basato sulla colorazione dei nodi di un grafo in cui i nodi che hanno blocchi in comune sono connessi da un arco
- Se richiesta eseguo l'interprocedural analysis (-IPO in icc), ovvero analizzo come vengono chiamate le funzioni.
- NOTA: il compilatore NON esegue ottimizzazioni se non è certo al 100% che queste non modificano il comportamento del programma
- Dovreste parlare con il compilatore…

Riassumendo

Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	03
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	01
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	01
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	01
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	02
Code motion	Remove code from a loop that computes the same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	01
Pipeline scheduling	Reorder instructions to improve pipeline performance	01
Branch offset optimization	Choose the shortest branch displacement that reaches target	01

Dall'ISA alla machine organization

L'appendice A, in inglese, va scaricata dal sito

Computer Architecture =

Instruction Set Architecture (ISA):
Come l'Hardware viene visto dal Software

+

Machine Organization:
Implementazione fisica in termini di blocchi funzionali e logici