SAPIENZA
UNIVERSITÀ DI ROMA

Sistemi di calcolo

Capitolo 5:
Come si ottimizza un programma?

Corso di Laurea in Ingegneria Informatica e Automatica

Quali parti di un programma ottimizzare?

Cruciale identificare le parti critiche (se ci sono)

1) Studiare l'impatto che l'ottimizzazione di una parte ha sulle prestazioni del tutto

2) Identificare i colli di bottiglia (hot spot) in cui si concentra il grosso del lavoro, usando tool chiamati performance profiler.

More computing sins are committed in the name of efficiency (without necessarily achieving it) than any other single reason - including blind stupidity. (Bill Wulf) [motto del corso!]

Metriche di prestazioni

- **Latenza**: tempo richiesto per il completamento di una operazione
- **Throughput**: numero di operazioni completate nell'unità di tempo
- **Spazio**: memoria richiesta da un programma
- **Energia**: consumo energetico del sistema che esegue il programma

Scala degli eventi in un sistema di calcolo

Evento[20]	Latenza effettiva	Latenza riscalata
Ciclo di clock	0.4 nsec	1 sec
Accesso cache L1	0.9 nsec	2 sec
Accesso cache L2	2.8 nsec	7 sec
Accesso cache L3	28 nsec	1 min
Accesso RAM DDR3 DIMM	100 nsec	4 min
Accesso disco SSD	50-150 μsec	1.5 – 4 giorni
Accesso disco a rotazione	1-10 msec	1 – 9 mesi
Invio pacchetto Internet continentale/intercontinentale	65-141 msec	5 – 11 anni

Speedup

$$\text{speedup} = \frac{\text{Tempo programma non ottimizzato}}{\text{Tempo programma ottimizzato}}$$

Es. speedup 2x vuole dire che il programma ottimizzato richiede la metà del tempo di quello non ottimizzato.

Legge di Amdahl

La legge di Amdahl determina lo speedup complessivo di un programma a fronte dell'ottimizzazione di una sua parte:

$$\text{speedup} = \frac{1}{\frac{\alpha}{k} + 1 - \alpha}$$

- α = frazione del programma ottimizzata
- k = speedup ottenuto su quella frazione

Legge di Amdahl: esempi

Es 1 $\alpha = 0.5$, $K = 2x$

$$speedup = \frac{1}{\frac{0.5}{2} + 1 - 0.5} = 1.33x$$

Es 2 $\alpha = 0.9$, $K = 6x$

$$speedup = \frac{1}{\frac{0.9}{6} + 1 - 0.9} = 4x$$

Legge di Amdahl: dimostrazione

Supponiamo di dividere un programma in due parti:

A: parte da ottimizzare B: rimanente

Si ha $T_A = \alpha T$ e $T_B = (1-\alpha)T$, dove T è il tempo totale richiesto dal programma. Si immagini ora di ottimizzare A ottenendo speedup K per la versione ottimizzata A'. Si ha:

$$\text{Speedup} = \frac{T}{T'} = \frac{T}{T_{A'} + T_B} = \frac{T}{\frac{T_A}{K} + T_B} = \frac{T}{\frac{\alpha T}{K} + (1-\alpha)T}$$

($T_{A'} = T_A/K$)

Profilazione delle prestazioni

La profilazione delle prestazioni serve per identificare i colli di bottiglia, che meritano di essere ottimizzati.

Metodi di profilazione

- **Manuale**, instrumentando il codice con chiamate a funzioni di misurazione del tempo
- **Automatico**, usando dei tool (profiler) che calcolano il tempo speso nelle varie parti del programma (es. funzioni)

Profilazione manuale

Lo schema generale per profilare una porzione di codice è il seguente:

```
double start = misura_tempo();
// porzione di codice da misurare
// ...
double elapsed = misura_tempo() - start;
printf("Tempo richiesto: %f\n", elapsed);
```

strumentazione

Si noti che misura_tempo() calcola il tempo attuale rispetto a un punto di riferimento dato.

Codice di misurazione manuale del tempo

misura_tempo()

```
#include <time.h>  // clock_gettime (misura nanosecondi)
double get_real_time_msec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec*1E03 + ts.tv_nsec*1E-06;
}
```

tempo reale

misura_tempo()

```
#include <sys/resource.h>  // getrusage (misura microsecondi)
double get_user_time_msec() {
    struct rusage ru;
    getrusage(RUSAGE_SELF, &ru);
    return ru.ru_utime.tv_sec*1E03 + ru.ru_utime.tv_usec*1E-03;
}
```

tempo speso attivamente dal processo sulla CPU (tempo utente)

tempo utente

Codice di misurazione manuale del tempo

```
#include <sys/resource.h>  // getrusage (misura microsecondi)
double get_system_time_msec() {
    struct rusage ru;
    getrusage(RUSAGE_SELF, &ru);
    return ru.ru_stime.tv_sec*1E03 + ru.ru_stime.tv_usec*1E-03;
}
```

system

tempo speso attivamente sulla CPU in modalità supervisore

DEMO 5.1-profiling/E1-profile-sorting

Latenza e risoluzione funzioni misurazione tempo

Latenza: tempo richiesto dalla funzione di misurazione del tempo

Risoluzione: minimo intervallo misurabile

Entrambe devono essere << della porzione da misurare (almeno un ordine di grandezza)

Se la porzione da misurare è troppo piccola si può pensare di ripeterla più volte, misurare il tempo totale e ottenere il tempo medio dividendo per il num. di ripetizioni

attenzione agli effetti dovuti alla cache!

DEMO 5.1-profiling/E2-*, E3-*

Profilazione automatica: gprof

Misura automaticamente tempo speso nelle varie funzioni

1) compilare programma con gcc -pg ... -O mioprog

2) eseguire il programma mioprog. Questo genererà un report di profilazione chiamato gmon.out

3) visualizzare il report con gprof mioprog

DEMO 5.1-profiling/E4-gprof

Aspetti chiave delle ottimizzazioni

Ottimizzare le prestazioni: ridurre la quantità di risorse richieste dall'esecuzione (es. tempo, spazio, energia)

NB attenzione a non compromettere la correttezza!

Livello di intervento
- algoritmo: risultati più marcati, anche ordini di grandezza
- codice: ridurre istruzioni e loro costo (compilatore o programmatore)

quali vanno fatte dal programmatore?

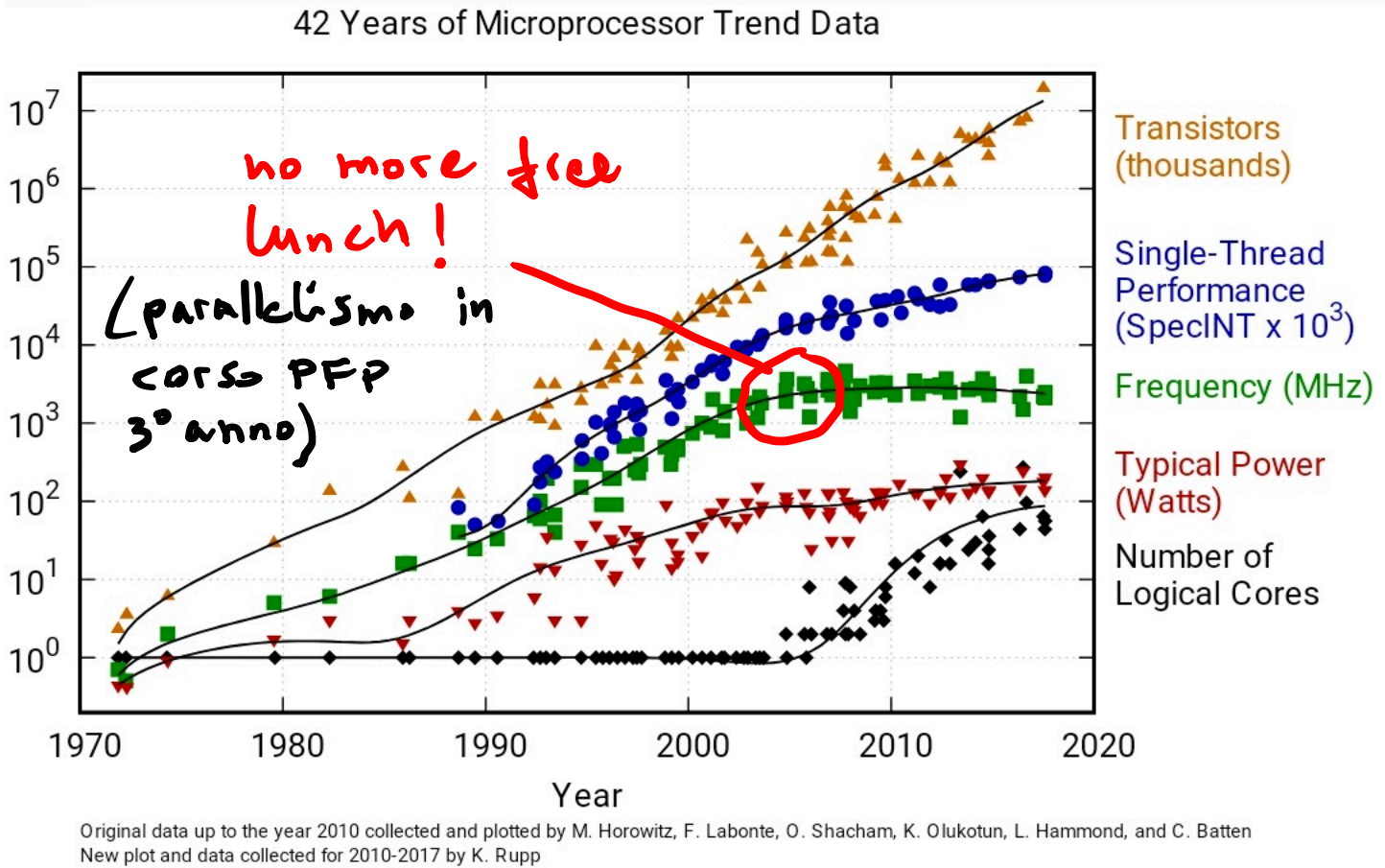
Importanza delle ottimizzazioni

Programmatori professionisti pensano innanzitutto a correttezza, manutenibilità, robustezza, portabilità, modularità, ecc. prima ancora delle prestazioni.

Perché occuparsi di prestazioni? Perché sono la "moneta" per comprare altre qualità del software che provocano riduzioni di prestazioni (es. modularità).

Con l'avvento delle architetture multi-core è diventato indispensabile che il programmatore si occupi di sfruttarle.

Legge di Moore: numero transistor x2 ogni 24 mesi



Tecniche di ottimizzazioni prestazioni

Ridurre il "work" = numero di istruzioni eseguite

Ridurre il costo delle istruzioni eseguite (tempo richiesto dalle istruzioni)

Ottimizzazioni gcc

`-O0`: default, ok per debugging

`-O1` (oppure `-O`): ottimizza, compilatore più lento di `-O0`

`-O2`: ottimizza ulteriormente, ancora più lento, non aumenta dimensione codice

`-O3`: ottimizza ancora di più, potrebbe aumentare la dimensione del codice generato

`-Os`: ottimizza per ridurre la dimensione del codice; attiva tutte le ottimizzazioni `-O2` che non aumentano la dimensione del codice, più altre

Riduzione work: constant folding

Precalcola valori costanti:

C originale: `test-cf.c`	Constant folding (manualmente)

`int f() {`
`  return 8+(14/2)*3;`
`}`
	`int f(){`
	`  return 29;`
	`}`

Incluso automaticamente in gcc -O1
Non va fatto manualmente dal programmatore

DEMO 5.2-work-opt/E1-const-fold

Riduzione work: constant propagatiom

Propaga valori costanti

C originale: `test-cp.c`	Constant propagation (manualmente)	Constant propagation + constant folding (manualmente)

`int x;`
`int f() {`
`  x = 8;`
`  return x-2;`
`}`
	`int x;`
	`int f() {`
	`  x=8;`
	`  return 8-2;`
	`}`
		`int x;`
		`int f() {`
		`  x=8;`
		`  return 6;`
		`}`

Non va fatto manualmente dal programmatore

DEMO 5.2-work-opt/E2-const-prop

Riduzione work: eliminazione sottoespressioni

Eliminazione sott-espressioni comuni

C originale: `test-cse.c`	Common subexpr. elimination (manualmente)

`int expr(int x, int y) {`
`  return (x + y)*(x + y);`
`}`
	`int expr(int x, int y) {`
	`  int z = x+y;`
	`  return z*z;`
	`}`

Non va fatto manualmente dal programmatore

DEMO 5.2-work-opt/E3-cse

Riduzione work: loop-invariant code motion (licm)

Evitare il ricalcolo di valori immutati durante i loop

C originale: `test1-licm.c`	Loop-invariant code motion (manualmente)

`int f(int x, int n) {`
`  int s = 1;`
`  for (; n>0; n--) s *= 7+x;`
`  return s;`
`}`
	`int f(int x, int n) {`
	`  int s=1; z=7+x;`
	`  for(; n>0; n--) s *= z;`
	`  return s;`
	`}`

Attenzione: potrebbe dovere essere effettuata dal programmatore

DEMO 5.2-work-opt/E4-licm, E5-licm

Riduzione work: loop unrolling

Riduzione del numero di iterazioni di un ciclo

C originale: `test-lu.c`	Loop unrolling di fattore 3 (manualmente)

`int sum(int *v, int n) {`
`  int i, s = 0;`
`  for (i=0; i<n; ++i) s += v[i];`
`  return s;`
`}`
	`int sum(int *v, int n) {`
	`  int i, s=0;`
	`  for (i=0; i+2<n; s+=3){`
	`  s+=v[i];`
	`  s+=v[i+1];`
	`  s+=v[i+2];`
	`  }`
	`  for(;i<n;i++) s+=v[i];`
	`  return s;`

Serve se n non è multiplo di 3

(NB) Alle volte viene effettuato dal compilatore.

DEMO 5.2-work-opt/E6-unroll

Riduzione work: function inlining

Rimpiazza chiamata col corpo della funzione invocata

C originale: `test-inl.c`	Function inlining (manualmente)

`int sqr(int x) { return x*x; }`
 	
`void f(int* v, int n) {`
`  int i;`
`  for (i=0; i<n; i++)`
`  v[i] = sqr(i);`
`}`
	`void f(int*x, int n){`
	`  int i;`
	`  for (i=0;i<n;i++) v[i]= i*i;`
	`}`

Eliminata in automatico dal compilatore con -O3.
Evitare di farla manualmente perché riduce modularità!

DEMO 5.2-work-opt/E7-inlining

Riduzione costo: register allocation

Tempo di accesso a registri << Tempo accesso memoria

⇒ compilatori tendono ad allocare variabili in registri

C originale: `test-ra.c`	Senza alloc. registri (`-O0`)	Con alloc. registri (`-O1`)

`void f(int n) {`
`  while (n--) g();`
`}`
	`f:  jmp  L3`
	`L4: call g`
	`L3: movl 4(%esp), %eax`
	`  leal -1(%eax), %edx`
	`  movl %edx, 4(%esp)`
	`  testl %eax, %eax`
	`  jne L4`
	`  ret`
		`f:  pushl %ebx`
		`  movl 8(%esp), %ebx`
		`  testl %ebx, %ebx`
		`  je  L3`
		`L5: call g`
		`  subl $1, %ebx`
		`  jne L5`
		`L3: popl %ebx`
		`  ret`

ciclo

ciclo

DEMO 5.3-cost-opt/E1-reg-alloc

Riduzione costo: strength reduction

L'operator strength reduction consiste nel rimpiazzare un'operazione con una equivalente ma meno costosa:

Es.
- $x * 4 \rightarrow x << 2$
- $x * 2 \rightarrow x + x$ o $x << 1$
- $x * 17 \rightarrow x * 16 + x = x << 4 + x$
- $x / 8 \rightarrow x >> 8$

NB: le operazioni di shift e somma sono molto più veloci di moltipl. e divisione

DEMO 5.3-cost-opt/E2-strength-red

Riduzione costo: strength reduction

Rimpiazzare prodotti con somme ripetute

Effettuato dal compilatore con -O1

C originale: `test-srl.c`	Strength reduction (manualmente)

`void f(int* v, int n, int k) {`
`  int i;`
`  for (i=0; i<n; i++)`
`  v[i] = i*k;`
`}`
	`void f(int* v, int n, int k) {`
	`  int i, c=0;`
	`  for (i=0; i<n; i++) {`
	`  v[i] = c;`
	`  c += k;`
	`  }`
	`}`

DEMO 5.3-cost-opt/E3-strength-red2

Riduzione costo: allineamento dati in memoria

Vincolo accessi a memoria CPU: un oggetto di dimensioni s byte deve iniziare ad un indirizzo x multiplo di s, vale a dire x mod s = 0.

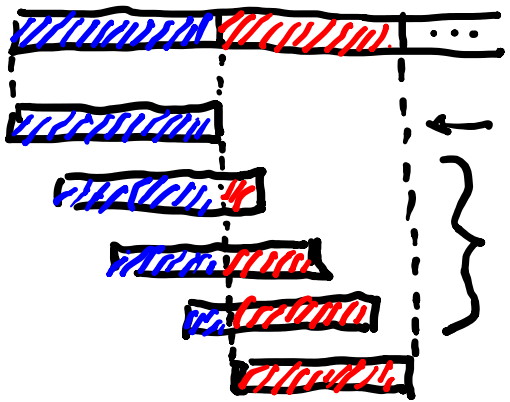
Motivazione:

modello di memoria programmatore:

modello di memoria CPU

Una CPU a 32 bit carica un intero oggetto di 4 byte alla volta se allineato a indirizzo multiplo di 4 byte

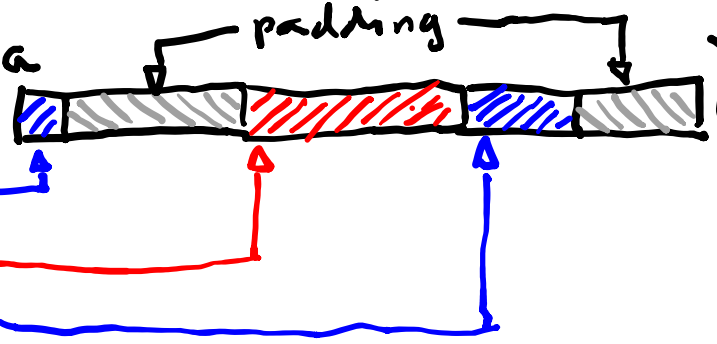
Riduzione costo: allineamento dati in memoria



← 1 accesso carica tutti e 4 byte richiesti

2 accessi a memoria per recuperare le 2 portioni del dato (word)

Es.
```
struct S {
    char x;
    int y;
    short z;
};
```



Allineamento corretto assumendo che a sia multiplo di 4 byte.

(NB) padding = spazio sprecato

Riduzione costo: allineamento dati in memoria

L'allineamento migliora le prestazioni ?

DEMO 5.3-cost-opt/E4-alignment

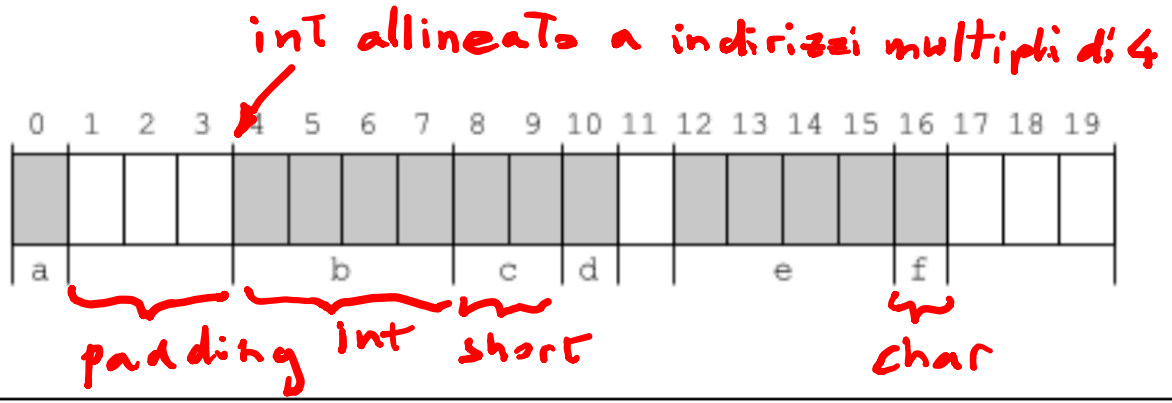
Sì, ma solo su architetture datate
Alcune CPU tuttavia generano un'eccezione
in caso di accesso non allineato

Riduzione costo: altre ottimizzazioni

Nel resto del corso vedremo altre ottimizzazioni del costo delle istruzioni legate ad aspetti hardware come i sistemi di memoria cache.

Riduzione spazio: ordinamento campi struct

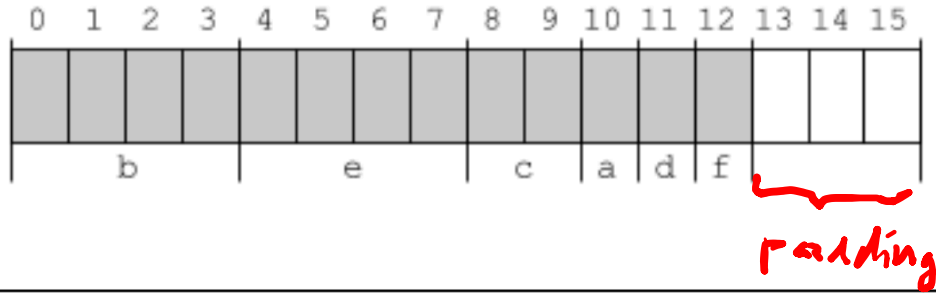
```
struct S {
     char a;
     int b;
     short c;
     char d;
     int e;
     char f;
};
```



Riduzione spazio: ordinamento campi struct

Si può ridurre lo spazio di una struct mettendo per primi i campi più grandi:

```
struct S {
    int b;
    int e;
    short c;
    char a;
    char d;
    char f;
};
```



Il padding finale serve a garantire che in un array di strutture il primo campo sia allineato correttamente.

Riduzione spazio: eliminazione codice irraggiungibile

Dead code elimination: rimozione istruzioni non raggiungibili

C originale: `test-dce.c`	Dead code elimination (manualmente)

`void f() {`
`  if (0) return 10;` ← irraggiungibile
`  return 20;`
`}`
	`void f() {`
	`  return 20;`
	`}`

Un altro modo per ridurre la grandezza (footprint) del codice è compilare con -Os

DEMO 5.4-space-opt/E1-dead-code

Lezioni imparate

1) Ottimizzare solo i colli di bottiglia
(usare un profiler per identificarli)

2) Attenzione a mantenere la correttezza!

3) Molte ottimizzazioni sono fatte direttamente dal compilatore (è utile disassemblare il codice per capire le ottimizzazioni fatte)