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?

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

Scala degli eventi in un sistema di calcolo

Evento[20]	Latenza effettiva	Latenza riscalata
Ciclo di clock	0.4 nsec	
Accesso cache L1	0.9 nsec	
Accesso cache L2	2.8 nsec	
Accesso cache L3	28 nsec	
Accesso RAM DDR3 DIMM	100 nsec	
Accesso disco SSD	50-150 μsec	
Accesso disco a rotazione	1-10 msec	
Invio pacchetto Internet continentale/intercontinentale	65-141 msec	

Speedup

Legge di Amdahl

Legge di Amdahl: esempi

Legge di Amdahl: dimostrazione

Profilazione delle prestazioni

Profilazione manuale

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

Codice di misurazione manuale del 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;
}
```

```
#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;
}
```

Codice di misurazione manuale del tempo

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

DEMO 5.1-profiling/E1-profile-sorting

Latenza e risoluzione funzioni misurazione tempo

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

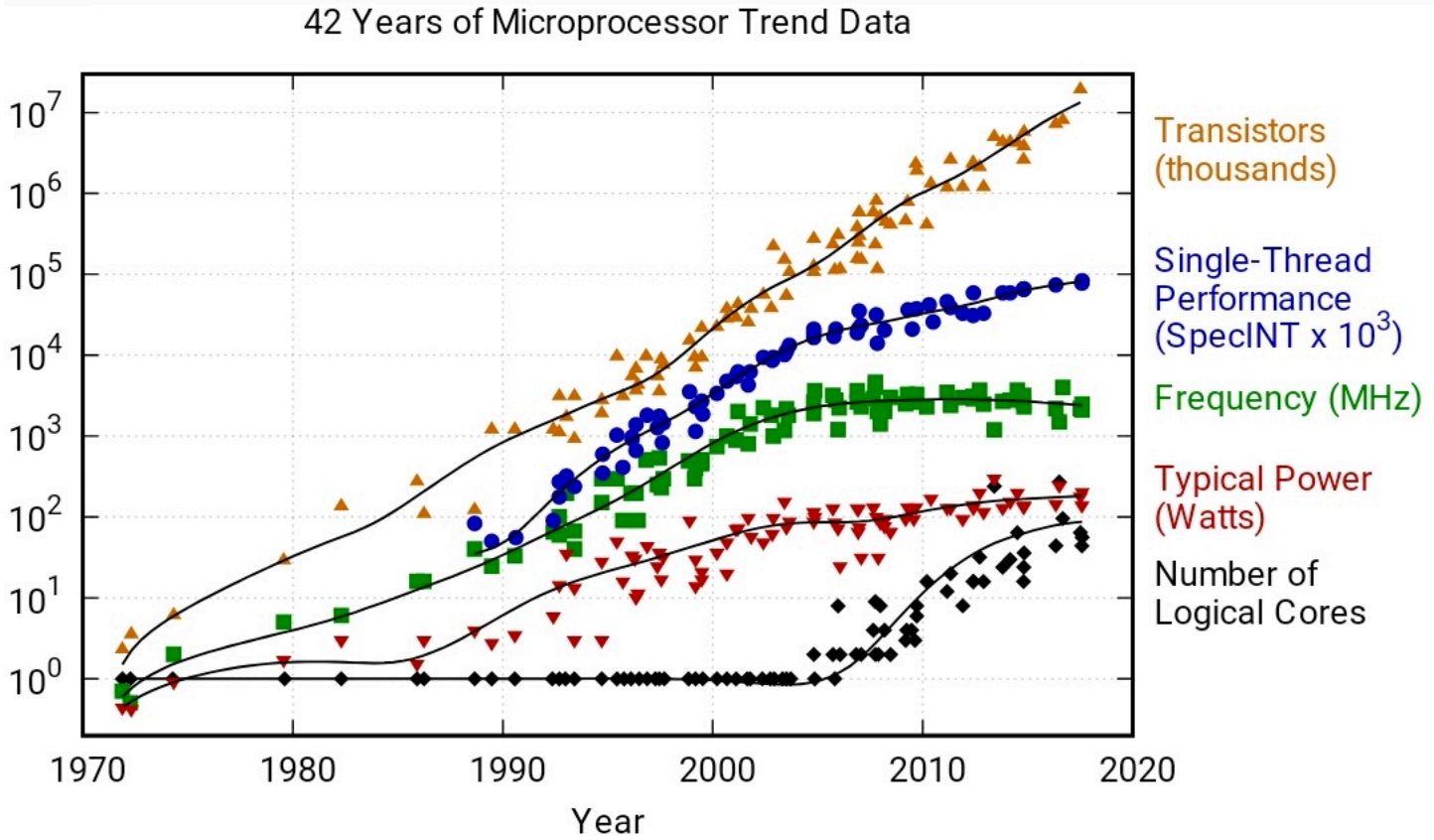
Profilazione automatica: gprof

DEMO 5.1-profiling/E4-gprof

Aspetti chiave delle ottimizzazioni

Importanza delle ottimizzazioni

Legge di Moore: numero transistor x2 ogni 24 mesi



Tecniche di ottimizzazioni prestazioni

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

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

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

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

Riduzione work: constant propagatiom

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() {`
	
	`}`
		`int x;`
		`int f() {`
		
		`}`

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

Riduzione work: eliminazione sottoespressioni

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) {`
	
	`}`

DEMO 5.2-work-opt/E3-cse

Riduzione work: loop-invariant code motion (licm)

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) {`
	
	`}`

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

Riduzione work: loop unrolling

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;`
`}`	

DEMO 5.2-work-opt/E6-unroll

Riduzione work: function inlining

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);`
`}`	

DEMO 5.2-work-opt/E7-inlining

Riduzione costo: register allocation

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`

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

Riduzione costo: strength reduction

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

Riduzione costo: strength reduction

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) {`
	
	`}`

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

Riduzione costo: allineamento dati in memoria

Riduzione costo: allineamento dati in memoria

Riduzione costo: allineamento dati in memoria

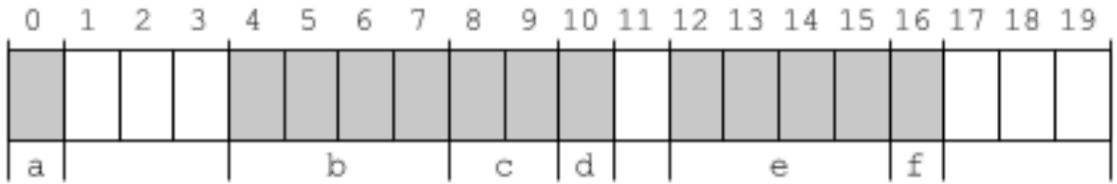
DEMO 5.3-cost-opt/E4-alignment

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.

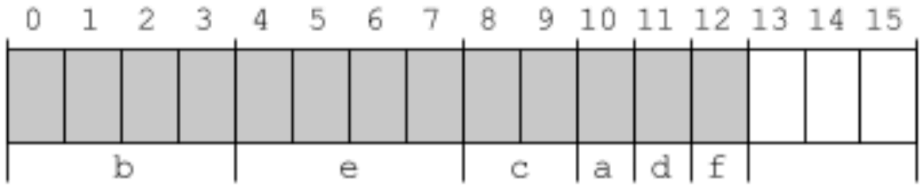
Riduzione spazio: ordinamento campi struct

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



Riduzione spazio: ordinamento campi struct

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



Riduzione spazio: eliminazione codice irraggiungibile

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

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

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

Lezioni imparate