LIMITI NELLA MISURAZIONE DELLE PRESTAZIONI

① **LEGGE DELLA GRANDEZZA**: Tempo necessario per eseguire la misurazione.

② **RISOLUZIONE DELLO STRUMENTO**: minima intervalla temporale misurabile.

> **IMPORTANTE**: la misurazione di T con uno strumento di risoluzione δ è significativa solo se la misurazione di T è maggiore di δ

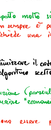
> **IMPORTANTE**: la misura deve essere ripetuta varie per la misura deve riflettere gli risultati in modo ragionevole

PROFILAZIONE AUTOMATICA

Utilizzo del Tool `gprof`:

1) Compilare il codice Target con "$-pg$"
2) Eseguo
3) Visualizzo risultati con "`gprof <nome programma>`"

CALL GRAPH



ASPETTI CHIAVE DELLE OTTIMIZZAZIONI

OBIETTIVO: riduzione uso risorse con miglioramento delle prestazioni (tempo) senza compromettere la correttezza

Possono intervenire su due livelli

ALGORITMO: scegliere l'algoritmo con costo computazionale minore che risolve il problema

Progettista o programmatore

+ Impatto molto significativo sulle prestazioni
− Non sempre è possibile
− Richiede una implementazione ad hoc

IMPLEMENTAZIONE: ottimizzare il codice che implementa l'algoritmo scelto

compilatore o programmatore

+ Soluzione (parzialmente) automatica
+ Soluzione "economica" in termini
− Possono essere platform-dependent

TECNICHE DI OTTIMIZZAZIONE DEL CODICE

Due approcci:

① **RIDUZIONE DEL "WORK"**: ridurre il numero di istruzioni da eseguire

② **RIDUZIONE DEL "COSTO"**: sostituire con più istruzioni quelle che impiegano più CPU per essere eseguite

Il compilatore applica numerose tecniche di ottimizzazione del codice attraverso le seguenti opzioni:

- `-O0` : disabilita ogni ottimizzazione
- `-O1` : ottimizza
- `-O2` : = `-O1` + ++
- `-O3` : = `-O2` + ++
- `-Os` : riduce lo spazio
- `-Og` : ottimizzare il codice con performance (dove possibile) e strumenti di debug

TECNICHE DI OTTIMIZZAZIONE

- **CONSTANT FOLDING**

Precalcolo i valori costanti (anche con OP!)

```
int f() {
    x = 8;
    return x - 2;
}
```
⇒ ① `CO` return `8-2;` ⇒ ② `CP` return `6;`

- **CONSTANT PROPAGATION**

I valori costanti vengono propagati nel codice fino a dove è possibile

```
int x;
int f() {
    x = 8;
    return x - 2;
}
```
⇒ ① `CO` return `8-2;` ⇒ ② `CP` return `6;`

- **COMMON SUBEXPRESSION ELIMINATION**

Sostituisce occorrenze uguali di una stessa sotto espressione

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

- **LOOP INVARIANT CODE MOTION**

Sposta fuori da un ciclo le espressioni che non dipendono dal ciclo stesso

```
int f(int x, int n) {
    int s = 4;
    for (; n>8; n--)
        s = 1 + x;
    return s;
}
```
⇒
```
int f(int x, int n) {
    int s = 4;
    int z = 1+x;
    for (; n>8; n--)
        s += z;
    return s;
}
```

ATTENZIONE AI CASI PARTICOLARI!

```
int f(int x, int n) {
    int s = 4;
    for (; n>8; n--)
        s = 1 + g(x);
    return s;
}

int g(int x) {
    printf("Chiamata \n");
    return x+2;
}
```
```
int f(int x, int n) {
    int s = 4;
    int z = g(x);
    for (; n>8; n--)
        s += z;
    return s;
}
```

- **LOOP UNROLLING**

Data un ciclo con un numero di iterazioni noto, riduce o elimina queste iterazioni.

```
int f() {
    int a=0;
    for (int i=0; i<60; i++)
        a++;
    return a;
}
```
⇒
```
int f() {
    int a=0;
    a++;
    a++;
    ...
    a++;
    return a;
}
```
×60

Dei casi comuni il numero di iterazioni non è noto

```
int sum (int *v, int n) {
    int i, s=0;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```
Eseguo blocchi di 4 in 4, per $n/4$ volte

Eseguo blocchi di dimensione 1 per max $k-1$ volte

```
int sum (int *v, int n) {
    int i, s=0;
    for (i=0; i<k; i++)
        s += v[i];
    for (; i<n; i+=4) {
        s += v[i];
        s += v[i+1];
        s += v[i+2];
        s += v[i+3];
    }
    for (; i<n; i++)
        s += v[i];
    return s;
}
```
k operazioni per volta

- **FUNCTION INLINING**

Dove possibile sostituisce la chiamata a funzione con il corpo della stessa

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

- **REGISTER ALLOCATION**

Massimizzare l'uso dei registri

- **OPERATOR STRENGTH REDUCTION**

Sostituisce una istruzione con altra istruzione equivalente ma meno costosa

```
x / 8        x >> 3
x * 64       x << 6
x * 1        x
x * 25       (x<<6) - x
```
```
void f(int *v, int n, int m){
    int i;
    for (i=0; i<n; i+=m)
        v[i] = i*k;
}
```
>
```
void f(int *v, int n, int m){
    int i, c=0;
    for (i=0; i<n; i+=m) {
        v[i] = c;
        c += k;
    }
}
```

- **DEAD CODE ELIMINATION**

Elimina il codice non raggiungibile

- **RIDUZIONE CAMPO DI ALLOCAZIONE**

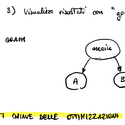
Ridimensionamento dei campi di una struct (del più grande al più piccolo) per minimizzare il padding

LA GEMA APPLICARE LE PROGRAMMATORE

- **ALLINEAMENTO IN MEMORIA**

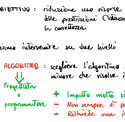
Ogni dato in dimensione 2 byte viene salvato in memoria ad indirizzo x multiplo di 2.

ESTENSIONI A 64 BIT



RAX RBX RCX, RBX, RSP, RBP, RSI, RDI

R8, R9, ..., R15



suffisso		dimensione
b	byte	1 byte
w	word	2 byte
l	int	4 byte
q	quadword	8 byte

(es.: `movq %rax, %rbx`)

PASSAGGIO PARAMETRI

num. parametro	REG
1	RDI
2	RSI
3	RDX
4	RCX
5	R8
6	R9
7	stack
⋮	⋮

CALLER - SAVED : R8, R9, R11 - R15

CALLER - SAVED : altri