Sistemi di Calcolo (A.A. 2015-2016)
Corso di Laurea in Ingegneria Informatica e Automatica
Sapienza Università di Roma

Esempio di prova di esame per esonerati – Durata 1h 30'

Esercizio 1 (gerarchie di memoria)

Si consideri un sistema di calcolo basato su processore Intel Core i7 con la seguente configurazione di cache: L1=32 KB dati per core, L2=256 KB per core ed L3=6 MB condiviso da tutti i core. Ogni linea di cache è di 64 byte e ogni pagina di memoria virtuale è di 4 KB.

Si consideri il seguente frammento di programma eseguito sequenzialmente su un singolo core della CPU dove `v` è di tipo `short*`:

```
for (i=0; i<1024*1024; i++) x += v[32*i];
```

Rispondere alle seguenti domande, assumendo che `sizeof(short)==2` e che le variabili `i`, `x` e `v` siano tenute in registri della CPU.

1. Mostrare come calcolare i cache miss e i cache hit generati dal programma su ciascun livello della cache. Che numeri si ottengono?
2. Quante pagine distinte di memoria vengono accedute dal programma?

Inserire le risposte nel file `es1.txt`.

Esercizio 2 (allocazione dinamica della memoria)

Si consideri il seguente frammento di programma C:

```
typedef struct {
    char x;
    int y;
} S;
p1=malloc(1000*sizeof(S));
p2=malloc(2000*sizeof(S));
p3=malloc(4000*sizeof(S));
free(p1);
free(p3);
p4=malloc(6000*sizeof(S));
```

Si assuma che l'allocatore parta da un heap inizialmente di dimensione 4 KB. L'allocatore cercherà di minimizzare la dimensione dell'heap tentando di usare lo spazio libero con gli indirizzi più bassi. Se non si riesce a soddisfare una richiesta di allocazione, l'heap verrà espanso del minimo indispensabile. Rispondere alle seguenti domande:

1. Come è partizionato l'heap in blocchi liberi/in uso dopo ogni `malloc`/`free`?
2. Si genera frammentazione durante l'allocazione? Se sì, di che tipo?
3. Quanto è grande l'heap alla fine?

Inserire le risposte nel file `es2.txt`.

Esercizio 3 (analisi del costo dei programmi)

1. Si considerino i seguenti due frammenti di codice x86:

Versione 1	Versione 2
`movl (%eax), %ecx` `addl $4, %ecx`	`leal (%eax), %ecx` `addl $4, %ecx`

Ci aspettiamo prestazioni diverse per i due frammenti? Se sì, di che fattore possono differire nel caso peggiore su un sistema di calcolo reale?

2. Si considerino i seguenti due frammenti di codice x86:

Versione 1	Versione 2
`addl %ecx, %ecx`	`imull $2, %ecx`

Quale versione ci aspettiamo che sia più veloce? Quale tecnica di ottimizzazione è stata applicata per ottenere una versione dall'altra?

3. Si considerino i seguenti due frammenti di programma C:

Versione 1	Versione 2

`for (i=0; i<100; i++) {`
`  s += a[i];`
`  s += b[i];`
`  s += c[i];`
`}`
	`for (i=0; i<100; i++) {`
	`  s1 += a[i];`
	`  s2 += b[i];`
	`  s3 += c[i];`
	`}`
	`s += s1+s2+s3;`

Quale versione ci aspettiamo che sia più veloce? Perché?

Inserire le risposte nel file `es3.txt`.

Esercizio 4 (ottimizzazione di programmi)

Si crei nel file `sort-opt.c` una versione ottimizzata del seguente modulo `sort.c`:

```
#include "sort.h"

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void sort(int* v, int n) {
    int k, i;
    for (k=0; k<n; k++)
        for (i=1; i<n; i++)
            if (cmp(v+i-1, v+i) > 0) swap(v+i-1, v+i);
}
```

Compilare due versioni del programma, usando `gcc` **a 32 bit** con livello di ottimizzazione 1 e lo stesso modulo `main.c`:

1. Non ottimizzata: eseguibile `sort`.
2. Ottimizzata: eseguibile `sort-opt`.

Ai fini dell'ottimizzazione:

1. Usare `gprof` per identificare le porzioni più onerose computazionalmente. Per evitare confusione, chiamare gli eseguibili usati per la profilazione `sort-pg` e `sort-opt-pg`.
2. Esaminare il modulo assembly `sort.s` generato a partire da `sort.c` con `gcc –S -O1` per capire quali ottimizzazioni siano già state effettuate dal compilatore.

Rispondere alle seguenti domande:

1. Descrivere le ottimizzazioni applicate e dire perché si ritiene che siano efficaci.
2. Riportare il tempo di esecuzione di `sort` e di `sort-opt` usando il comando `time`.
3. Riportare i flat profile delle due versioni usando `gprof`.
4. Di quante volte è più veloce l'**eseguibile** `sort-opt` rispetto a `sort`?
5. Usando i profili `gprof` delle due versioni del programma e la legge di Amdahl, calcolare di quante volte è più veloce la **funzione** `sort` ottimizzata rispetto a quella originale. Per tempo speso in `sort` si intende quello speso nella funzione stessa (self) e nelle funzioni da essa chiamate (children).

Inserire le risposte nel file `es4.txt`. Alla fine del compito, **non cancellare** `gmon.out` e gli altri eseguibili creati.