Architettura degli elaboratori—Seconda prova di verifica intermedia

A.A. 2017-18—18 dicembre 2017

Riportare in alto a destra di ciascun foglio consegnato Nome, Cognome, Corso (A o B) e numero di matricola.
I risultati saranno comunicati via web appena disponibili.

Si consideri un'architettura D-RISC Pipeline, con EU parallela (EU master per le aritmetico logiche corte ed EU slave da 4 stadi per la moltiplicazione e divisione fra interi). Il sistema è dotato di cache di primo livello associative su insiemi a 4 vie con linee (blocchi) da σ=8 parole e di una cache di secondo livello (on chip) ancora associativa su insiemi, capace di trasferire una parola verso la cache di primo livello ogni 2τ.

Si consideri il seguente frammento di pseudocodice:

```
int x[N], a[N], b[N], z[N];
…
for(int i=0; i<N; i++) {
  x[i] = (a[i]*b[i])+c;
  if(b[i]<N)                    // in ¾ dei casi b[i] risulta minore di N
     z[b[i]] = x[i]+a[i];
}
```

Assumiamo che N=1024, che tutti i vettori ed il codice risiedano in cache di secondo livello.

Si richiede:

- di discutere il *working set* dell'applicazione;
- l'individuazione di tutte le cause di degrado delle prestazioni;
- di valutare il tempo di completamento ideale e reale del programma (possono essere escluse dal computo le istruzioni di inizializzazione);
- l'individuazione di possibili ottimizzazioni;
- la valutazione del guadagno di prestazioni ottenuto.

Successivamente, si consideri la possibilità di utilizzare un processore modificato che permette l'esecuzione *out-of-order* delle istruzioni sulla EU master e se ne discuta l'impatto sulle prestazioni.

Bozza di soluzione

Valutazione del working set

Il programma accede i vettori a, b e x in modalità sequenziale (località senza riuso), mentre il vettore z viene acceduto in maniera random. Dunque il working set del programma sarà costituito dalle linee relative al codice, da una linea per ciascuno dei vettori a, b e x, e dall'intero vettore z.

Numero di fault

Il numero di fault minimo (fisiologici) è dato dal numero di fault per il codice e da N/σ fault per ciascuno dei vettori a, b e x e per il vettore z. Il vettore x è scritto interamente, quindi possiamo trascura il tempo di trasferimento dai livelli superiori della gerarchia di memoria in caso di fault. Il vettore z è anch'esso in sola scrittura ma, non essendo scritto interamente, il fault va trattato come un fault di lettura.

Codice D-RISC

Il programma può essere compilato in D-RISC come segue:

```
1.          CLEAR Ri
2.  loop:   LOAD RbaseA, Ri, Rai              // legge a[i]
3.          LOAD RbaseB, Ri, Rbi              // legge b[i]
4.          MUL Rai, Rbi, Rtemp               // calcola a[i]*b[i]
5.          ADD Rc, Rtemp, Rxi                // calcola a[i]*b[i] + c
6.          STORE Rbasex, Ri, Rxi             // memorizza x[i]
7.          IF>= Rbi, RN, cont                // controlla se b[i] < N
8.          ADD Rxi, Rai, Rtemp2              // se non lo è calcola a[i]+x[i]
9.          STORE Rbasez, Rbi, Rtemp2, non_deallocare      // e lo memorizza in z[b[i]]
10. cont:   INC Ri                            // i++
11.         IF< Ri, RN, loop                  // se i<N esegui un'altra iterazione
12.         END
```

Cause di degrado delle prestazioni

Abbiamo:

- una dipendenza EU-EU fra la MUL e la ADD successiva (4→5 e 8→9)
- una dipendenza IU-EU fra le ADD e le STORE successive (5→6 e 8→9)
- una dipendenza EU-IE fra la INC e la IF< (10→11)
- una dipendenza IU-EU fra la seconda LOAD e la IF>= (3→7)

Inoltre, abbiamo bolle da salto

- nel 25% delle iterazioni per via della IF>= (7)
- in tutte le iterazioni, tranne l'ultima, per via della IF< (11)

Tempo di completamento

Consideriamo l'esecuzione di una iterazione nei due casi (b[i]<N o no):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
IM	LD	LD	MUL	ADD	ST							IF>	ADD	ST	INC		IF<		END	LD			
IU		LD	LD	MUL	ADD		ST					ST	IF>	ADD	ST	ST	INC	IF<	IF<	END	LD		
DM			LD	LD									ST				ST					LD	
Eum				LD	LD	MUL	ADD				ADD				ADD			INC					LD
EU*							MUL	MUL	MUL	MUL													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18				
IM	LD	LD	MUL	ADD	ST							IF>	ADD	INC	IF<		END	LD					
IU		LD	LD	MUL	ADD		ST					ST	IF<	~~ADD~~	INC	IF<	IF<	~~END~~	LD				
DM			LD	LD									ST							LD			
Eum				LD	LD	MUL	ADD				ADD					INC					LD		
EU*							MUL	MUL	MUL	MUL													

Abbiamo 19t per iterazione con **then** eseguito e 17t per i rimanenti casi dunque per ciascuna delle iterazioni spendiamo ((3/4)*19+(1/4)*17)t) 18.5 t

Vengono eseguite N iterazioni e dunque il tempo di completamento al netto dell'inizializzazione sarà dell'ordine dei $1024*18.5*2*\tau$.

A questo dobbiamo aggiungere il tempo di trattamento dei fault, che sono in totale

2 (codice) + N/σ (vettore a)+ N/σ (vettore b)+ al più N/σ (vettore z[j]) = 2 * 3*128 = 386

(non abbiamo considerato i fault per x che viene acceduto in solo scrittura ed interamente) e che richiedono ciascuno un tempo pari a

$2(\tau+T_r) + 2\tau\ \sigma + \tau$

Per il trasferimento della linea dalla cache di secondo livello. Essendo la cache sullo stesso chip della CPU il tempo di trasferimento è 0 e dunque il tempo per un trattamento del fault si riduce a

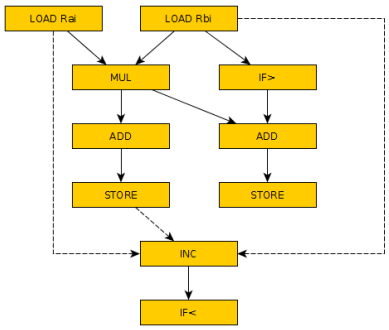
$2\tau + 16\tau + \tau = 19\tau$

Il tempo di completamento risulta quindi

$Tc = TcID + Tfault = 1024*18.5*2*\tau + 386 * 19\tau$

Ottimizzazioni

Il grafo data flow è il seguente:



La prima store (quella che memorizza x[i]) può essere chiaramente posticipata per alleviare il problema della dipendenza con il calcolo del valore da memorizzare. La seconda store (su x[b[i]]) invece

deve essere eseguita esclusivamente nel ramo **then** e dunque non può essere positicipata. La INC può essere anticipata purché mantenuta successiva alle load e purché la store x[i] avvenga con il registro base anticipato. Questo permette di ridurre l'effetto della dipendenza INC→IF< e, a seconda della posizione di arrivo della INC, di mitigare l'effetto di un'altra dipendenza. Noi scegliamo di interporla fra la prima MUL e la ADD successiva.

```
        CLEAR Ri
loop:   LOAD RbaseA, Ri, Rai
        LOAD RbaseB, Ri, Rbi
        MUL Rai, Rbi, Rtemp
        INC Ri
        ADD Rc, Rtemp, Rxi
        IF>= Rbi, RN, cont
        ADD Rxi, Rai, Rtemp2
        STORE Rbasez, Rbi, Rtemp2
cont:   IF< Ri, RN, loop1, delayed
        STORE Rbasex-1, Ri, Rxi
        END
```

La simulazione mostra che in questo caso l'esecuzione con il **then** preso impiega 14t e quella senza 11t.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IM	LD	LD	MUL	INC	ADD	IF>		ADD	ST			IF<		ST	LD		
IU		LD	LD	MUL	INC	ADD		IF>	ADD			ST	ST	IF<	ST	LD	
DM			LD	LD										ST		ST	LD
Eum				LD	LD	MUL	INC	ADD			ADD	ADD					
EU*							MUL	MUL	MUL	MUL							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IM	LD	LD	MUL	INC	ADD	IF>		ADD	IF<	ST		LD					
IU		LD	LD	MUL	INC	ADD		IF>	ADD	IF<	ST	ST	LD				
DM			LD	LD									ST	LD			
Eum				LD	LD	MUL	INC	ADD			ADD				LD		
EU*							MUL	MUL	MUL	MUL							

Dunque avremo un tempo medio di completamento ideale per iterazione pari a ((3/4)14+(1/4)11)t = 13.25t.

EU out-of-order

La presenza di una unità EUmaster out-of-order non porta alcun vantaggio nel codice ottimizzato. La secodna ADD non può comunque essere eseguita prima di quella che produce Rxi perché legge Rxi, nel caso ramo then preso, mentre nel caso di ramo then non proso non vi sono code di aritmetico logiche sulla EUmaster.

Diversa sarebbe stata la situazione se non avessimo invertito la prima ADD e la INC, ovvero nel codice

```
        CLEAR Ri
loop:   LOAD RbaseA, Ri, Rai
```

```
LOAD RbaseB, Ri, Rbi
MUL Rai, Rbi, Rtemp
ADD Rc, Rtemp, Rxi
INC Ri
IF>= Rbi, RN, cont
ADD Rxi, Rai, Rtemp2
...
```

In questo caso, la INC verrebbe eseguita sulla EU master prima del completamento della ADD (in attesa del risultato della MUL.