Architettura degli Elaboratori

Anno Accademico 2014-2015- Seconda prova di verifica intermedia

(Risultati e calendario degli orali saranno pubblicati sulla pagina web del corso appena disponibili)

Si consideri il seguente frammento di codice, che lavora sui vettori di interi A, B, C e D, ciascuno da N=8K posizioni:

```
for(i=0; i<N; i++)  {
   a[i]=a[i]+b[i]*c[i];
   b[i]=b[i]+c[i]+cost;
   j=b[i] mod 128;
   d[j]=d[j]+cost;
}
```

Si supponga che il risultato della compilazione venga eseguito su un processore D-RISC pipeline con EU parallela (EU slave che esegue moltiplicazione o divisione in 4*t*). La gerarchia di memoria comprende una cache di primo livello con linee (blocchi) da σ=64 parole e la memoria centrale interallacciata (*m* moduli).

Si chiede di

a) determinare il working set del programma
b) valutare il numero di fault di cache generati dalla esecuzione del programma e le eventuali ottimizzazioni
c) ottimizzare il codice ottenuto applicando le regole di compilazione standard
d) discuterne le prestazioni.

Facoltativo:

e) discutere cosa accade in caso di utilizzo di un processore D-RISC superscalare a due vie.

Traccia di soluzione

Working set

	Località	**Riuso**
Codice	Si	Si
A, B, C	Si	No
D	No	Si

Numero di Fault

Il numero di fault generati dal programma sarà dunque la somma di

- *k* fault per le istruzioni del programma (verosimilmente *k=1* visto che σ=64 e #istruzioni=16 (vedi codice))
- 3N/σ fault (complessivamente) per A, B e C
- 128/σ = 2 fault per D

Prefetching

In questo caso l'utilizzo della tecnica del prefetching permette la riduzione del numero di fault per A, B e C ai soli 3 fault iniziali. Occorrerebbe verificare che il prefetch delle tre linee di cache necessarie impiega meno del tempo impiegato per eseguire le σ-1 iterazioni che lavorano in cache prima dell'accesso alla nuova linea di cache per A, B, e C. Per fare questo occorre stimare il tempo speso nella singola iterazione e il tempo necessario al trattamento di un fault. Quest'ultima cosa richiede di fare delle assunzioni sul tipo di memoria e sulla relative struttura di interconnessione M-C.

Compilazione

Il codice compilato con le regole standard è il seguente:

```
1.  loop: LOAD RB, Ri, Rbi      // load c[i]
2.        LOAD RC, Ri, Rci      // load b[i]
3.        MUL Rbi, Rci, R1      // b[i]*c[i]
4.        LOAD RA, Ri, Rai      // load a[i]
5.        ADD Rai, R1, R2       // a[i]+b[i]*c[i]
6.        STORE RA, Ri, R2      // store a[i]
7.        ADD Rbi, Rci, R3      // b[i] + c[i]
8.        ADD R3, Rconst, R4    // b[i]+c[i]+cost
9.        STORE RB, Ri, R4      // store b[i]
10.       AND R4, #127, Rj      // j = b[i] mod 128
11.       LOAD RD, Rj, Rdj      // load d[j]
12.       ADD Rdj, Rconst, R5   // d[j] + cost
13.       STORE RD, Rj, R5      // store d[j]
14.       INC Ri                // inc i
15.       IF< Ri, RN, loop      // loop
16.       END
```

Abbiamo dipendenze 3→5 (EU-EU), 5→6 (IU-EU), 8→9 (IU-EU), 10→11 (IU-EU), 12→13 (IU-EU), 14→15 (IU-EU).

La simulazione:

	0	1	2	3	4	5	6	7	8	9
IM	LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci	MUL Rbi, Rci, R1	LOAD RA, Ri, Rai	ADD Rai, R1, R2	STORE RA, Ri, R2				
IU		**LOAD RB, Ri, Rbi**	**LOAD RC, Ri, Rci**	**MUL Rbi, Rci, R1**	**LOAD RA, Ri, Rai**	**ADD Rai, R1, R2**	**STORE RA, Ri, R2**			
DM			LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci		LOAD RA, Ri, Rai				
EUM				LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci	MUL Rbi, Rci, R1	LOAD RA, Ri, Rai	ADD Rai, R1, R2		
EU*/i							MUL Rbi, Rci, R1	MUL Rbi, Rci, R1	MUL Rbi, Rci, R1	MUL Rbi, Rci, R1

10	11	12	13	14	15	16	17	18	19
	ADD Rbi, Rci, R3	ADD R3, Rconst, R4	STORE RB, Ri, R4		AND R4, #127, Rj	LOAD RD, Rj, Rdj		ADD Rdj, Rconst, R5	STORE RD, Rj, R5
	STORE RA, Ri, R2	**ADD Rbi, Rci, R3**	**ADD R3, Rconst, R4**	**STORE RB, Ri, R4**	**STORE RB, Ri, R4**	**AND R4, #127, Rj**	**LOAD RD, Rj, Rdj**	**LOAD RD, Rj, Rdj**	**ADD Rdj, Rconst, R5**
		STORE RA, Ri, R2				STORE RB, Ri, R4			LOAD RD, Rj, Rdj
ADD Rai, R1, R2			ADD Rbi, Rci, R3	ADD R3, Rconst, R4			AND R4, #127, Rj		

20	21	22	23	24	25	26	27	28	29
		INC Ri	IF< Ri, RN, loop		end	LOAD RB, Ri, Rbi			
STORE RD, Rj, R5		**STORE RD, Rj, R5**	**INC Ri**	**IF< Ri, RN, loop**	**IF< Ri, RN, loop**	~~**end**~~	LOAD RB, Ri, Rbi		
			STORE RD, Rj, R5					LOAD RB, Ri, Rbi	
LOAD RD, Rj, Rdj	ADD Rdj, Rconst, R5			INC Ri					LOAD RB, Ri, Rbi

ci permette di concludere che la singola iterazione costa 26*t* e di conseguenza il tempo di servizio T vale 26*t*/15, poco meno di 2*t*.

Il tempo di servizio si sarebbe potuto calcolare anche utilizzando il modello analitico. Le dipendenze IU-EU sono tutte di distanza 1 (k=1) e hanno, rispettivamente, n=2,1,1,2,1 da cui d_1=5/15 e N_q = 7/5. La dipendenza EU-EU è di distanza 2 e abbiamo d_2 = 1/15 e L_{pipe}=4. Per i salti abbiamo λ=1/15. Con

questi valori otteniamo

$$T = (1+\lambda)\, t + \Delta_1 + \Delta_2 = 16t / 15 + 7t/15 + 3t /15 = 26t/15$$

L'analisi data flow del codice del corpo del ciclo ci permette di capire che alcune delle istruzioni possono essere spostate all'interno del codice assembler:

- il calcolo di b[i]+c[i]+cost non dipende dal risultato della moltiplicazione, dunque queste istruzioni possono essere utilizzate per allontanare le istruzioni della prima dipendenza (3→5).
- Anche il calcolo di b[i] mod 128 può essere anticipato in un punto qualunque del codice successivo alla produzione del nuovo valore per b[i]
- Le store possono essere eseguite come ultime istruzioni prima della nuova iterazione. In particolare, la STORE RD, Rj, R5 può essere utilizzata come istruzione da inserire nel delay slot della IF< che conclude il ciclo, visto che non dipende da Ri.

Dunque il codice può essere riscritto come segue:

1. loop: LOAD RB, Ri, Rbi
2. LOAD RC, Ri, Rci
3. MUL Rbi, Rci, R1
4. ADD Rbi, Rci, R3
5. ADD R3, Rconst, R4
6. LOAD RA, Ri, Rai
7. ADD Rai, R1, R2
8. AND R4, #127, Rj
9. STORE RB, Ri, R4
10. STORE RA, Ri, R2
11. INC Ri
12. LOAD RD, Rj, Rdj
13. ADD Rdj, Rconst, R5
14. IF< Ri, RN, loop, delayed
15. STORE RD, Rj, R5
16. END

Con la simulazione osserviamo che

	0	1	2	3	4	5	6	7	8	9
IM	LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci	MUL Rbi, Rci, R1	ADD Rbi, Rci, R3	ADD R3, Rconst, R4	LOAD RA, Ri, Rai	ADD Rai, R1, R2	AND R4, #127, Rj	STORE RB, Ri, R4	STORE RA, Ri, R2
IU		LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci	MUL Rbi, Rci, R1	ADD Rbi, Rci, R3	ADD R3, Rconst, R4	LOAD RA, Ri, Rai	ADD Rai, R1, R2	AND R4, #127, Rj	STORE RB, Ri, R4
DM			LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci				LOAD RA, Ri, Rai		
EUM				LOAD RB, Ri, Rbi	LOAD RC, Ri, Rci	MUL Rbi, Rci, R1	ADD Rbi, Rci, R3	ADD R3, Rconst, R4	LOAD RA, Ri, Rai	ADD Rai, R1, R2
EU*/i							MUL Rbi, Rci, R1	MUL Rbi, Rci, R1	MUL Rbi, Rci, R1	MUL Rbi, Rci, R1

10	11	12	13	14	15	16	17	18	19
	INC Ri	LOAD RD, Rj, Rdj	ADD Rdj, Rconst, R5	IF< Ri, RN, loop, del	STORE RD, Rj, R5		LOAD RB, Ri, Rbi		
STORE RA, Ri, R2	STORE RA, Ri, R2	INC Ri	LOAD RD, Rj, Rdj	ADD Rdj, Rconst, R5	IF< Ri, RN, loop, del	STORE RD, Rj, R5	STORE RD, Rj, R5	LOAD RB, Ri, Rbi	
STORE RB, Ri, R4		STORE RA, Ri, R2	INC Ri	LOAD RD, Rj, Rdj				STORE RD, Rj, R5	LOAD RB, Ri, Rbi
ADD Rai, R1, R2	AND R4, #127, Rj				LOAD RD, Rj, Rdj	ADD Rdj, Rconst, R5			

Arriviamo ad un tempo di servizio pari a 17*t* /15, vicino a *t*, in effetti. Sulla IU abbiamo due uniche bolle da 1 posizione.