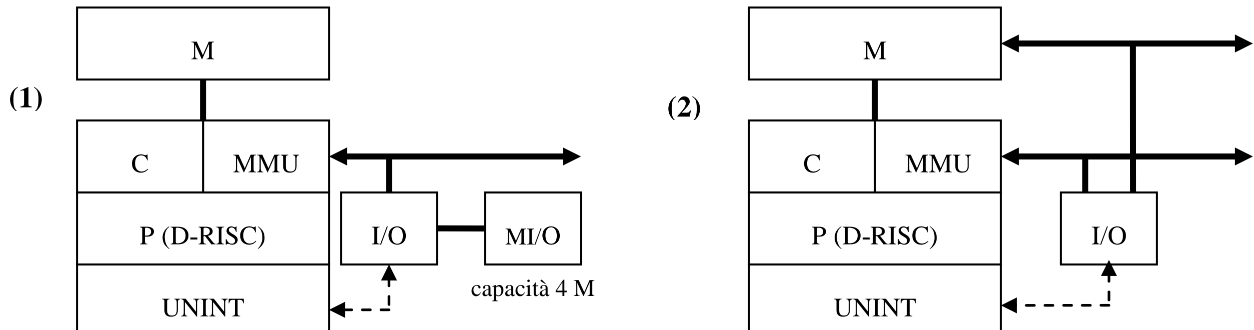
Esercitazione 4

Domanda 1

Un processo Q opera su due array di interi, A[N][N] e B [N][N], con N = 1K. Il valore di A è ottenuto da un'unità di I/O; il valore di B è il risultato di una certa elaborazione su A.

La CPU ha processore D-RISC e cache primaria. Si considerino le seguenti configurazioni, (1) e (2), dell'architettura dell'elaboratore, con particolare riguardo all'unità di I/O:



Nel caso (1) l'unità di I/O ha associata una unità di memoria MI/O contenente un componente logico memoria di capacità 4M parole, mentre nel caso (2) una simile unità non è prevista. L'unità di I/O acquisisce le parole di A dal dispositivo associato (non indicato in figura).

a) Si supponga che I/O *non* sia implementato come processo esterno e che *non* sia virtualizzato da un processo driver. Spiegare come, nei casi (1) e (2), avviene la cooperazione tra Q e l'unità di I/O affinché Q possa operare su A: cioè le azioni svolte dall'unità di I/O, dal processore, e dal processore quando è in esecuzione il processo Q. Spiegare se il tempo di completamento di Q è uguale o diverso nei casi (1) e (2) ed eventualmente in quale caso è maggiore, assumendo che tutte le unità di memoria abbiano lo stesso ciclo di clock e che tutti i collegamenti inter-chip abbiano la stessa latenza di trasmissione.

b) Si supponga che I/O *sia* implementato come processo esterno e che *non* sia virtualizzato da un processo driver. Si assuma l'esistenza di un livello con linguaggio LC. Spiegare le stesse cose del caso *a)* e, rispetto a tale caso, mettere in evidenza le differenze funzionali ed eventualmente di prestazioni.

Domanda 2

Un'applicazione è così definita:

- opera su due vettori A e B, di N = 8K interi ciascuno, e come risultato modifica B;
- riceve il valore di A e di B da una unità I/O e invia il nuovo valore di B alla stessa unità. Tale unità appartiene a un insieme di 8 unità identiche, capaci di trasferire blocchi di 512 parole tanto in ingresso quanto in uscita. Le unità di I/O sono implementate come processi esterni;
- un processo Master, associato all'applicazione, sceglie l'unità di I/O di cui sopra. All'inizio, l'applicazione attende da Master le informazioni che sono necessarie per procurarsi il valore di A e di B e per comunicare il nuovo valore di B, dopo di che attende il valore di A e di B, e prosegue nell'elaborazione.
    - *a)* Compilare l'applicazione in un processo LC.
    - *b)* Spiegare se / sotto quali condizioni l'implementazione dell'applicazione in LC comporta un overhead aggiuntivo rispetto al caso in cui le unità di I/O *non* siano implementate come processi esterni.

Domanda 3

Una unità di I/O collegata ad una CPU D-RISC è definita in maniera tale che il messaggio di interruzione alla CPU, per segnalare un certo evento *ev*, consiste in *n* parole, dove il valore di *n* è di volta in volta variabile e noto all'unità stessa.

Lo handler di *ev* provvede a copiare le *n* parole del messaggio di interruzione in una certa struttura dati B condivisa tra tutti i processi e nota ad ogni processo.

Spiegare come progettare lo handler, e scrivere lo handler in D-RISC.

Domanda 4

Una computazione a livello di processi contiene i processi A, B, C. Il processo A trasmette il valore di un array M di 1K interi a B e, solo quando è sicuro che B abbia ricevuto tale valore, trasmette a C lo stesso valore.

a) Supponendo che tutta la computazione sia descritta in LC, scrivere il processo A e le parti di B e C corrispondenti. *Usare due forme di comunicazione alternative per la cooperazione tra A e B.*

b) Supponendo che le azioni di A siano svolte da un'unità di I/O, ma che *non* siano esprimibili in LC, descrivere come deve essere implementata la stessa computazione del punto *a)*, spiegando le azioni svolte da A, B e C, con B e C processi interni.

Domanda 5

Si consideri il programma che, dati due vettori A e B di N = 8K interi ciascuno, calcola:

$$\forall i = 0 \,..\, N\text{-}1 : B[i] = f(\ B[A[i] \bmod N\ ]\ )$$

La CPU, con assembler è D-RISC e ciclo di clock τ, contiene una cache di primo livello operante su domanda, completamente associativa, di capacità 32K parole, blocchi di 16 parole, e scritture con il metodo Write-Through. Il livello superiore della gerarchia di memoria è la memoria principale, interallacciata con 4 moduli, ciclo di clock di 30τ, e collegamenti dedicati aventi latenza di trasmissione uguale a 5τ.

La funzione *f* è disponibile sotto forma di procedura D-RISC con parametri d'ingresso e di uscita in registri generali. La procedura consta di 200 istruzioni ed ogni esecuzione ne usa mediamente 100, delle quali 60 aritmetico-logiche corte, 20 aritmetico-logiche lunghe, 10 di salto incondizionato e 10 di salto condizionato.

a) Determinare il numero dei fault e l'insieme di lavoro del programma.

b) Compilare il programma e, con riferimento al caso di cui sopra, determinare il tempo di completamento e la performance in funzione di τ.

c) Determinare l'insieme di lavoro considerando anche gli oggetti del *processo* corrispondente al programma.

d) Discutere quale potrebbe essere l'impatto, sulle prestazioni del programma, dell'applicazione della tecnica del prefetching di blocchi.

Soluzione

Domanda 1

I due casi si caratterizzano per il modello di I/O, e di conseguenza per l'organizzazione fisica della memoria condivisa tra CPU e I/O:

(1) Memory Mapped I/O, con MI/O condivisa,

(2) Memory Mapped I/O e DMA, con M condivisa.

In D-RISC il modello MMI/O è primitivo, e quindi sempre disponibile, anche in presenza di DMA; anche nel caso (2), infatti, pur se non esiste fisicamente una memoria associata all'unità di I/O, MMI/O viene usato per scambiare informazioni con l'unità di I/O indirizzando locazioni fittizie.

In entrambi i casi (1) e (2), il funzionamento comprende le seguenti azioni:

1. il processo Q fa richiesta all'unità di I/O di un blocco di 1M parole. I parametri della richiesta dipendono dal formalismo disponibile (caso *a)* o *b)*); tra questi ci può essere la dimensione del blocco: per semplicità supporremo che 1M parole sia la dimensione standard del blocco;
2. il processo Q si sospende; l'invocazione della funzionalità di commutazione di contesto è esplicita o implicita a seconda del formalismo adottato;
3. l'unità I/O preleva dal dispositivo il blocco e lo scrive, parola per parola (man mano che le riceve), nella memoria condivisa: M/IO oppure M; come ricavare l'indirizzo base dipende dal formalismo;
4. I/O invia una interruzione per svegliare Q, indicando nel messaggio di I/O un riferimento a Q o al suo PCB, a seconda del formalismo. Il processo attualmente in esecuzione esegue l'handler che consiste appunto nella funzionalità di sveglia;
5. quando Q torna in esecuzione, esegue la computazione B = F (A, …) conoscendo l'indirizzo base di A in base al formalismo adottato per la descrizione.

Caso a): in qualunque linguaggio concorrente sia descritto Q, la cooperazione con l'unità di I/O deve essere espressa, per forza di cose, in assembler. Le azioni di Q si caratterizzano come segue:

azione 1: la richiesta è effettuata con una o più STORE con indirizzi logici mappati nello spazio di I/O; i parametri consistono in come riferire A e come riferire il PCB di Q. Ad esempio, adottando il metodo con indirizzi logici coincidenti, Q passa l'indirizzo logico base di A e del PCB;

azione 2: la sospensione è esplicita, chiamando la procedura per commutare contesto;

azione 5: nel caso (2) anche per l'array A (oltre che per tutti gli altri oggetti) verrà sicuramente utilizzata la gerarchia memoria principale - cache, mentre nel caso (1) dipende se l'architettura prevede il trasferimento in cache di informazioni fisicamente presenti nello spazio di I/O in quanto, se non fosse possibile sfruttare una organizzazione della memoria di I/O a larga banda, tale trasferimento potrebbe essere troppo lento agli effetti delle prestazioni della cache (come normalmente accade).

La differenza in prestazioni dipende da questa distinzione: se non è possibile usare cache, il caso (1) ha un tempo di completamente maggiore.

Caso b): Q e I/O sono descritti con il linguaggio concorrente LC. Esiste un canale di comunicazione (*ch1*) da Q ad I/O per la richiesta, ed un canale (*ch2*) da I/O a Q per la risposta (valore di A). Le azioni si caratterizzano come segue:

azione 1: la richiesta di Q è effettuata con una *send* su *ch1*; se la dimensione del blocco è implicita, l'unico parametro è il valore di *ch2* che I/O assegnerà ad una variabile channelname. Tutti gli altri parametri del caso *a)* ora sono ricavati tramite i descrittori dei canali e, in generale, dal supporto delle comunicazioni. Il supporto della *send* provvede ad inviare una segnalazione all'unità di I/O, che si trovava in attesa attiva nella *receive* su *ch1*;

azione 2: dopo aver eseguito la *send*, Q esegue una *receive* su *ch2*, che certamente provoca la sua sospensione. La commutazione di contesto è quindi effettuata implicitamente dal supporto;

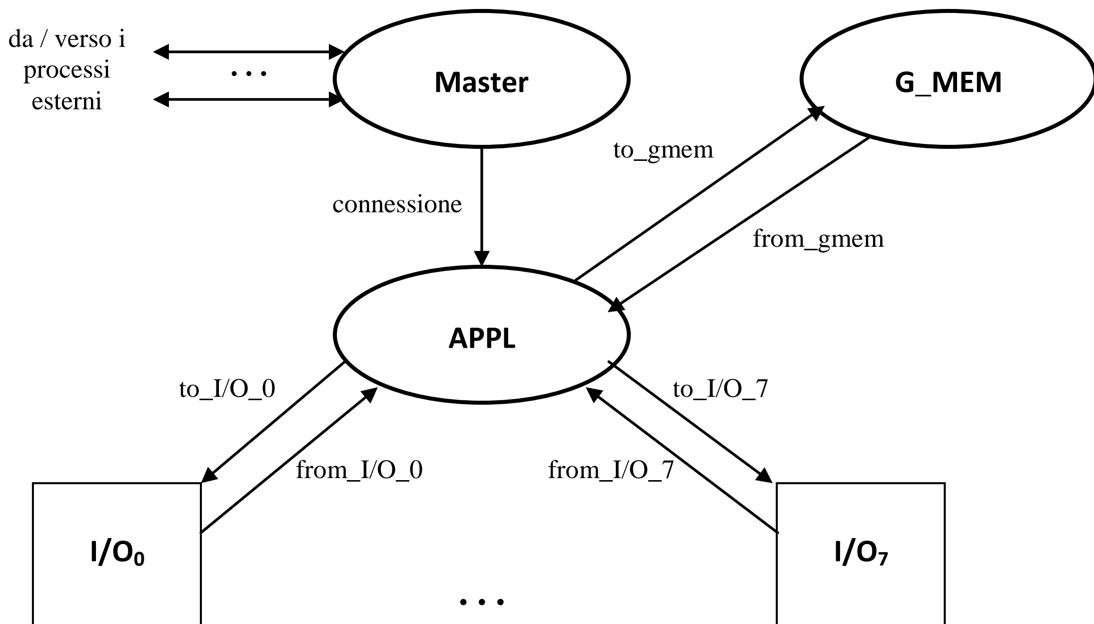
azioni 3, 4: il processo esterno I/O, dopo aver eseguito la *receive* su *ch1*, comincia a leggere i dati dal dispositivo e, contestualmente, esegue la *send* su *ch2*. Poiché trova che Q è sospeso, il supporto della *send* provvede a scrivere il blocco di dati direttamente nella variabile targa A il cui riferimento si trova nel descrittore di canale; nel descrittore di canale trova anche il riferimento al PCB di Q, che verrà inviato nel messaggio di interruzione per svegliare il partner.

Valgono le differenze tra le prestazioni delle organizzazioni (1) e (2), dovute agli accessi della CPU tanto ai descrittori di canale *ch1*, *ch2*, quanto alla variabile *A*: nel caso (2) tutti questi oggetti sono fisicamente allocati in M, e quindi soggetti al trasferimento in cache.

Il tempo di completamento è praticamente lo stesso nel caso *a)* e nel caso *b)*.

Domanda 2

a) Il processo APPL, derivante dalla compilazione dell'applicazione, fa parte del seguente grafo di processi, dove I/O_0, …, I/O_7 sono processi esterni eseguiti dalle rispettive unità di I/O:



Dal canale *connessione* APPL riceve l'indicazione circa i canali che deve usare per colloquiare con il processo esterno prescelto da Master. Usiamo due variabili *channelname*, *from_I/O* e *to_I/O* , alle quali assegnare, in seguito alla *receive* da Master, i valori dei due identificatori di canale che saranno usati per comunicare con il processo esterno. Trascurando il trattamento di eventuali eccezioni segnalate dai processi esterni (non previste dalle specifiche), il processo APPL è compilato in LC come segue:

APPL:: **int** A[N], B[N];

> **channel in** connessione (1), from_gmem (1), **var** from_I/O (32);
>
> > // è stato scelto un grado di asincronia dei canali che garantisce che i processi mittenti non si blocchino nell'esecuzione della *send* //
>
> **channel out** to_gmem, **var** to_I/O;
>
> > // i canali da /verso i processi esterni hanno tipo "blocco di 512 interi" //
>
> < definizione della funzione *block (nome_array, dimensione_blocco, indice_blocco* >;
>
> { **receive** (connessione, (from_I/O, to_I/O));
>
> **for** (i = 0; i < 16; i++)
>
> > **receive** (from_I/O, *block* (A, 512, i));
>
> **for** (i = 0; i < 16; i++)
>
> > **receive** (from_I/O, *block* (B, 512, i));
>
> *< programma per trasformare A e B in B* >;
>
> **for** (i = 0; i < 16; i++)
>
> > **send** (to_I/O, *block* (B, 512, i));
>
> **terminate** }

Entrato per la prima volta in esecuzione, con tutta probabilità APPL andrà in stato di attesa sulla *receive* da Master. Una volta tornato in esecuzione, con tutta probabilità APPL andrà in stato di attesa sulla prima *receive* da I/O_i; una volta svegliato dalla *send* eseguita da I/O_i e una volta tornato in esecuzione, APPL si

ritroverà automaticamente il valore del primo blocco nella corrispondente parte di A, senza dover effettuare una copia (grazie al supporto delle primitive nel caso di processo destinatario in attesa). Le successive *receive* porteranno APPL in attesa oppure no a seconda della velocità relativa rispetto all'elaborazione di I/O_i (che avviene con *parallelismo reale*). Quando eseguirà il ciclo di *send* verso I/O_i, APPL non andrà mai in stato di attesa a condizione che il canale to_I/O_i abbia grado di asincronia maggiore o uguale a 16, in caso contrario talvolta potrà portarsi in attesa a seconda della velocità relativa rispetto a I/O_i. Ovviamente, essendo I/O_i un processo predefinito, il grado di asincronia dei suoi canali di ingresso sarà una costante fissata una volta per tutte, il cui valore non potrà garantire il funzionamento ideale per tutte le applicazioni.

b) Nel caso che le unità di I/O *non* siano implementate come processi esterni, APPL *non* può utilizzare le primitive di LC per scambiare blocchi di dati con tali unità, bensì deve utilizzare codice espressamente previsto, nel processo stesso e in handler di interruzioni, per gestire *in modo esplicito i trasferimenti di ingresso-uscita*. A questo scopo, occorre avere conoscenza di come si comporta l'unità di I/O. In ogni caso, APPL e unità di I/O funzionano tra loro in parallelo, e interagiscono attraverso la *memoria condivisa*, implementata mediante DMA e/o Memory Mapped I/O, e, per la sincronizzazione, attraverso interruzioni da I/O a CPU e segnalazioni esplicite da CPU a I/O.

L'implementazione "non LC" può prevedere che I/O_i invii una sequenza di interruzioni, il cui secondo parametro permette di riferire una zona di memoria condivisa, in cui è presente sia il valore del blocco che una informazione per identificare APPL. Il trattamento dell'interruzione provvede a svegliare APPL (a meno che non sia già in esecuzione) ed a informarlo del modo di accedere al blocco. Non è detto che il blocco sia stato copiato direttamente nel blocco corrispondente di A, a meno che precedentemente APPL non abbia fatto conoscere ad I/O_i (con una segnalazione esplicita mediante istruzioni di STORE) l'informazione su "dove" copiarlo. In ogni caso, si ottengono prestazioni non superiori a quelle dell'implementazione in LC, nella quale la *send* di I/O_i provvede a copiare direttamente il messaggio nella variabile targa di APPL. Inoltre nell'implementazione in LC non c'è bisogno di interruzioni, e relative esecuzioni di handler, eccetto quelle di sveglia, a meno che l'unità di I/O non sia così semplice da dover demandare l'esecuzione della *send* alla CPU: anche in questo caso, le prestazioni in LC non sono inferiori, dovendo l'handler eseguire più o meno lo stesso codice del caso "non LC".

Si tenga conto che, nel caso il trasferimento dati da I/O_i avvenga mentre APPL si trova in esecuzione, nemmeno nell'implementazione "non LC" viene scritto direttamente il blocco nella parte corrispondente della variabile A; in tal caso, viene comunque effettuata una doppia copia.

Considerazioni analoghe si applicano alle comunicazioni da APPL verso I/O_i. La più efficiente implementazione "non LC" prevede che i blocchi vengano scritti direttamente nella zona della memoria di lavoro di I/O_i (Memory Mapped I/O; oppure con modalità DMA, se l'unità di I/O la prevede). Anche in questo caso, le prestazioni non sono superiori a quelle dell'implementazione in LC, in quanto la *send* di APPL avrà lo stesso comportamento: scriverà direttamente i blocchi nella variabile targa, provvedendo poi a segnalare la "sveglia"di I/O_i mediante una istruzione STORE.

In conclusione, la versione LC non introduce un overhead aggiuntivo rispetto alla versione "non LC", in quanto il supporto delle primitive di comunicazione contiene, in modo invisibile, tutte le ottimizzazioni che, eventualmente, sono presenti nel codice "non LC".

Domanda 3

In un calcolatore con CPU D-RISC le unità di I/O, dove aver ricevuto l'ack dell'interruzione, inviano alla CPU, via Bus di I/O, due parole: la prima codifica l'evento, la seconda un valore usato come parametro dallo handler. Nel nostro caso, l'unità di I/O ha memorizzato, in una struttura dati *S* nella sua memoria (MI/O), le *n* parole precedute dal valore di *n*. Il valore inviato alla CPU nella seconda parola deve permettere allo handler di indirizzare *S* mediante Memory Mapped I/O: con il metodo degli indirizzi logici coincidenti si tratta direttamente dell'indirizzo di S.

Per ogni processo, l'indirizzo logico della struttura dati *B*, in cui copiare *S*, è contenuto in un a posizione nota del proprio PCB.

Se RG[Rsecond] contiene l'indirizzo di S, il codice dello handler è:

```
        // leggi n //
                LOAD   Rsecond, 0, Rn
        // leggi indirizzo di B //
                LOAD   Rpcb, RoffsetB, RB
        // loop di copia di S in B //
                INCR   Rsecond
                CLEAR  Ri
LOOP:           LOAD   Rsecond, Ri, Rtemp
                STORE  RB, Ri, Rtemp
                INCR   Ri
                IF <   Ri, Rn, LOOP
        // ritorno da procedura //
                GOTO   Rret
```

Domanda 4

a) Per essere certo dell'avvenuta ricezione del messaggio trasmesso, A può utilizzare una canale sincrono verso B: sia *chB*. In questo caso, il codice di A sarà del tipo:

```
A:: channel out chB, chC, …
        send (chB, M);
        send (chC, M);…
```

In questo caso la send su *chC* sarà eseguita senz'altro dopo che il messaggio M è stato ricevuto dal processo che effettuerà la receive corrispondente sul canale *chB*. Alternativamente, il canale da A a B può essere asincrono, ma in questo caso occorre una nuova comunicazione per la conferma della ricezione (un messaggio senza valore ( )); possiamo quindi utilizzare un canale asincrono con una successiva receive (su un canale preferibilmente asincrono, in modo da non rallentare le operazioni di B):

```
A:: channel out chB, channel in risp(1), … ::
        send (chB, M);
        receive (risp, ( ) );
        send (chC, M);…
```

Nel primo caso, il codice di B sarà del tipo:

```
B:: channel in chB (0), …
        receive (chB, arrayM);
        …
```

mentre nel secondo caso sarà :

```
B:: channel in chB (1), …
        receive (chB, arrayM);
        send (risp, ( ) );
        …
```

Il codice di C, uguale in tutti e due i casi, sarà

```
C:: channel in chC (1), …
        receive (chC, arrayM);…
```

b) In questo caso l'interazione con l'unità di I/O A avviene senza la mediazione di LC. Le azioni relative alla comunicazione con A sono:

1. il processo B richiede la lettura di M, utilizzando MMI/O, per passare il comando di lettura ed i relativi parametri all'unità A: riferimento alla variabile M e riferimento al PCB_B, dove "riferimento" assume un significato diverso a secondo del metodo usato per risolvere il problema delle strutture condivise riferite indirettamente, ad esempio indirizzi logici coincidenti;
2. A trasferisce i dati in M (allocato in memoria principale, se si dispone di DMA, o comunque nella memoria di I/O) e genera un'interruzione il cui messaggio associato contiene il riferimento a PCB_B;
3. il trattamento di tale interruzione provoca la sveglia del processo B;
4. il processo B effettua una comunicazione al processo C per comunicare l'avvenuta lettura di M;
5. il processo C, con meccanismi analoghi a quelli appena visti per B, ordina la lettura di M all'unità A, …
6. …

Domanda 5

a) La parte codice del programma è caratterizzata sia da località che da riuso, trattandosi di un loop ripetuto *N* volte. La procedura occupa 13 blocchi, il programma principale un ulteriore blocco (vedi punto *b)*), senza fare ipotesi su come, in memoria virtuale, vengono disposte le istruzioni della procedura rispetto al programma principale; i 14 blocchi del codice provocano altrettanti fault, dopo di che sono mantenuti in cache per tutta la durata del programma, facendo quindi parte dell'insieme di lavoro. Si è supposto che anche il trasferimento dei blocchi di codice sia soggetto alla strategia su domanda.

L'array A è caratterizzato solo da località; essendo utilizzato completamente, la sua lettura provocherà N/σ fault. Agli effetti dell'insieme di lavoro, è sufficiente che un solo blocco di A alla volta sia presente in cache.

Per quanto riguarda l'array B, osserviamo come sia presente in una certa misura riuso, per quanto ad ogni passo non sia predicibile il valore dell'indice $j = A[i]$ *mod N* e quindi non sia predicibile quante volte ogni elemento di B può essere riusato (eventualmente, mai usato). Ne consegue che la strategia migliore è imporre che ogni blocco di B, una volta letto, *non venga deallocato* dalla cache, *limitando il numero di fault su B a* N/σ. Senza imporre questa strategia, nel caso più sfavorevole ogni accesso a B provocherebbe fault. D'altra parte, poiché in uno stesso blocco di B può aver luogo prima una scrittura e poi una lettura, i fault per le operazioni di *scrittura* (per le quali vale la località) devono essere gestiti *esplicitamente* con *trasferimenti dalla memoria*. Se l'unità cache è realizzata in modo da non trasferire blocchi dalla memoria in caso di scritture, la lettura in cache dei blocchi di B verrà provocata comunque inserendo istruzioni di LOAD in ordine crescente, come per A. Quindi, anche il numero di fault per B è esattamente uguale a N/σ e, per sfruttarne il riuso, *tutti i blocchi di B devono far parte dell'insieme di lavoro*. Ciò si realizza con opportune *annotazioni* ("non deallocare") nelle istruzioni di LOAD su B.

Complessivamente, il numero di fault del programma è dato da:

$$N_{fault} = N_{fault-istr} + N_{fault-A} + N_{fault-B} = 14 + 2\frac{N}{\sigma} = 1038$$

In prima approssimazione:

$$N_{fault} \sim 2N/\sigma = 1024$$

L'insieme di lavoro ha dimensione

$$WS = 14 + 1 + \frac{N}{\sigma} = 527 \ blocchi$$

In prima approssimazione:

$$WS \sim N/\sigma = 512$$

che rappresentano circa un quarto dei blocchi disponibili (2K blocchi); quindi, anche tenendo conto delle ulteriori informazioni necessarie al supporto del processo (vedi punto *c)*), l'insieme di lavoro risiede completamente in cache.

b) A tempo di compilazione sono inizializzati i registri *RA* e *RB* agli indirizzi logici base degli array, *Ri* a zero, *RN* al valore *N*, *Rf* all'indirizzo logico della procedura. I parametri della procedura sono passati attraverso i registri *Rin* e *Rout*. Come detto sopra, facciamo uso di *annotazioni*, nelle istruzioni di LOAD applicate agli elementi di B, per *non deallocare* blocchi di cache.

Il programma è compilato come segue, con ovvio significato dei nomi dei registri:

```
LOOP:  LOAD  RA, Ri, Ra
       MOD   Ra, RN, Rj
       LOAD  RB, Ri, Rx, non_deallocare
```

// questa istruzione è inserita dal compilatore per provocare comunque il trasferimento del blocco di B in cache, prima che ne venga modificato l'elemento i-esimo; il registro Rx non contiene informazione significativa; l'istruzione può essere eliminata se l'architettura firmware prevede che, anche in caso di fault in scrittura, venga sempre provocato il trasferimento del blocco in cache; il numero dei fault è lo stesso nelle due implementazioni //

LOAD RB, Rj, Rin, **non_deallocare**

// ovviamente, la prima delle due LOAD non genera fault se in precedenza il blocco è stato trasferito in cache in seguito al fault di questa seconda LOAD; quindi, le due istruzioni di LOAD fanno sì che tutti i blocchi di B siano letti in cache //

CALL Rf, Rret

STORE RB, Ri, Rout

INCR Ri

IF < Ri, RN, LOOP

END

Complessivamente le istruzioni sono

$$n_{istr} = 108$$

ripetute N volte.

Dal punto di vista delle prestazioni, l'istruzione MOD appartiene alla classe delle aritmetico-logiche lunghe, il cui rappresentante (MUL) ha tempo medio di elaborazione stimato in 50τ. L'istruzione CALL appartiene alla classe dei salti incondizionati.

Essendo $t_c = 3\tau$ il tempo di accesso della cache in assenza di fault, il tempo medio di completamento nel caso ideale è dato da:

$$T_{c\text{-}id} = N\,[\,108\,T_{ch} + 4\,T_{ex\text{-}LD} + 61\,T_{ex\text{-}INCR} + 21\,T_{ex\text{-}MOD} + 11\,T_{ex\text{-}IF} + 11\,T_{ex\text{-}GOTO}\,] =$$

$$= N\,[\,108*5\tau + 4*5\tau + 61*\tau + 21*50\tau + 11*2\tau + 11*\tau\,] = 1704\,N\,\tau$$

Le scritture con il metodo *Write-Though* non provocano degradazione di prestazioni, in quanto la distanza media tra due scritture consecutive è maggiore di $1704\,\tau$, quindi molto maggiore del tempo di servizio della memoria principale (persino nel caso in cui non fosse interallacciata).

Il tempo di trasferimento di un blocco da memoria principale (interallacciata con m = 4 moduli, e "intelligente" da servire il trasferimento di un intero blocco ricevendo un'unica richiesta dalla CPU) vale:

$$T_{trasf} = \frac{\sigma}{m}\,\tau_M + 2\,T_{tr} + m\,\tau = 134\,\tau$$

L'overhead dovuto ai fault di cache vale quindi:

$$T_{fault} = N_{fault} * Tt_{rasf} = (1876 + 17\,N)\,\tau \sim 17{,}3\,N\,\tau$$

che, grazie anche al dimensionamento dell'insieme di lavoro, risulta trascurabile rispetto al tempo di completamento ideale. L'efficienza relativa della cache vale ~ 1, e il tempo di completamento:

$$T_c \sim 1721\,N\,\tau \sim 14{,}1 * 10^6\,\tau$$

Il tempo medio di elaborazione per istruzione vale:

$$T = \frac{T_c}{N\,n_{istr}} \sim 16\,\tau$$

e la performance:

$$\wp = \frac{1}{T} = \frac{0{,}06}{\tau}\ istruzioni/_{sec}$$

c) Il processo corrispondente al programma contiene, nella sua memoria virtuale, anche tutti gli oggetti facenti parte del supporto a tempo di esecuzione del processo stesso. Alcuni di questi oggetti sono caratterizzati da un certo riuso e sono presenti in memoria principale, in quanto fanno parte dell'insieme di lavoro della gerarchia memoria virtuale – memoria principale; una volta trasferiti in cache, conviene che essi vi rimangano finchè il processo è in fase di esecuzione, quindi fanno parte anche dell'insieme di lavoro della gerarchia memoria principale – cache. Tali oggetti sono:

- PCB del processo stesso: contenuto ampiamente entro 1 pagina di memoria virtuale; assumendo la pagina di ampiezza 1K, il PCB occupa meno di 64 blocchi di cache;
- Tabella di Rilocazione: complessivamente il processo occupa uno spazio un po' superiore a *2N*; assumendo uno spazio di indirizzamento di 20 – 30 K, la tabella di rilocazione occupa 2 blocchi di cache;
- testa della Lista Pronti: 1 blocco di cache;
- codice dello scheduling a basso livello, handler di interruzioni ed eccezioni, e strutture dati relative: come ordine di grandezza circa 1 pagina, quindi circa 64 blocchi di cache;

Tutti gli altri oggetti (come PCB di altri processi, canali con il processo gestore della memoria, primitive di comunicazione, variabili targa di comunicazioni) *nel caso esaminato* sono riferiti con probabilità molto più bassa (hanno scarso riuso), e quindi non interessa forzare il loro mantenimento in cache.

Complessivamente, il numero di blocchi addizionali richiesti dal supporto del processo, da mantenere in cache per minimizzare la probabilità di fault di cache (cioè, dei blocchi addizionali appartenenti all'insieme di lavoro), è dell'ordine del centinaio.

d) Come visto, i fault di cache non penalizzano apprezzabilmente le prestazioni, per cui il prefetching non è necessario. Analizzando comunque questo aspetto, verrebbe ad annullarsi il numero di fault sulle istruzioni e su A, mentre verrebbe ridotto, ma non annullato, il numero di fault su B (senza possibilità di quantificare quest'ultimo impatto).