Dipartimento di Ingegneria dell'informazione

Laurea Triennale in
INGEGNERIA DELL'INFORMAZIONE

Meccanismi scalabili di cache coherence in sistemi multiprocessore

Relatore:
Prof. Gianfranco Bilardi

Laureando:
Antonino Andrea Care'
Mat. 1195628

Anno Accademico 2022-2023

Indice

Capitolo 1

Introduzione

Nel corso degli anni l'esigenza di elaboratori con una sempre maggiore potenza di calcolo, ha concentrato l'attenzione del mondo dell'informatica sul problema della realizzazione efficiente dell'elaborazione parallela ed in particolare ha portato allo sviluppo delle architetture multiprocessore. Uno dei primi e principali passi evloutivi dei sistemi multiprocessor è stato l'introduzione delle memorie cache. Infatti fornendo ad ogni unità di elaborazione del sistema una memoria cache, quindi una propria memoria veloce e locale, il sistema ha ottenuto due importanti vantaggi: la riduzione del tempo di accesso ai dati e soprattutto è stata così garantita la riduzione dei conflitti tra i processori per accedere alla memoria principale condivisa. L'esistenza di più memorie locali comporta però l'esigenza di mantenere consistenti tra di loro tali memorie per permettere il corretto funzionamento del sistema di elaborazione, quindi l'introduzione delle memorie cache ha comportato un nuovo problema: il problema della cache coherence. Esistono varie soluzioni a questo problema, in questo studio sono state analizzate quelle implementate a livello firmware: i protocolli di cache coherence.

Questo lavoro di tesi non si concentra sull'analisi delle moderne architetture multiprocessor, o tantomeno sullo studio di tecniche per migliorare le prestazioni a livello

monoprocessore, bensì il focus principale di questo studio è quello di sondare i limiti della scalabilità orizzontale di un sistema multiprocesso. La scalabilità è la capacità di un sistema di incrementare le proprie prestazioni, in particolare nell'ambito delle architetture hardware questa si suddivide in scalabilità verticale, cioè relativa all'aumento della capacità di elaborazione del singolo processore e in scalabilità orizziontale, cioè relativa alla capacità di incrementare il numero di processori su cui è distribuito il carico di lavoro. Saranno quindi esposte le possibili opzioni architetturali per ambienti multiprocesso, le principali tecniche di cache-coherence e passo per passo analizzeremo le migliori soluzioni per un sistema con un numero di processori N che sia il maggiore possibile.

Capitolo 2

Architetture Multiprocessore

I vari modelli di sistemi multiprocessore possono presentare molteplici differenze tra di loro, in questa sezione illustreremo le principali possibili caratterizzazioni discutendo e motivando le scelte compiute per il suddetto modello di studio.

2.1 Struttura principale

La struttura principale di un architettura parallela multiprocessore è la seguente:

- Sono presenti N nodi di elaborazione chiamati anche nodi processore, costituiti da una unità di interfaccia del nodo, CPU con una eventuale memoria locale non condivisa con gli altri nodi del sistema.
- I nodi di elaborazione condividono una memoria principale (MM), ovvero ciascun processore del sistema è in grado di indirizzare ogni locazione di tale memoria condivisa.
- Esistono strutture di interconnessione che permettono l'accesso dei processori alla memoria condivisa e garantiscono la comunicazione e la trasmissione di

dati tra i vari nodi processore del sistema. Tale struttura è connessa ai vari nodi processori tramite le loro unità di interfaccia.

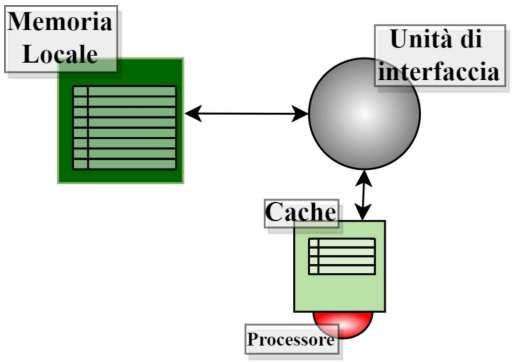


Figura 2.1: Nodo processore

2.2 Organizzazione Memoria Principale (MM)

Esistono due principali metodi per organizzare la memoria principale:

- Il primo metodo è basato su un accesso uniforme alla memoria da parte dei vari processori del sistema, da qui il nome UMA (Uniform Memory Access), ovvero il tempo di accesso ad un determinato modulo di memoria M_i non dipende da quale processore P_j richiede l'accesso, cioè il tempo di accesso del processore P_j al modulo di memoria M_i è costante per ogni i, j.

- Nel secondo metodo invece i moduli della memoria condivisa hanno distanze diverse dai vari nodi processori, in particolare l'accesso da parte di P_j al modulo di memoria M_i ha ritardo minimo quando $i = j$. Tale gruppo di architetture sono classificate come NUMA (Non Uniform Memory Access), tipicamente in questo tipo di architetture la memoria principale condivisa non è una unità di memoria assestante come nel caso UMA, bensì la memoria condivisa è costituita dall'insieme delle memorie locali dei nodi processori. In questo modo

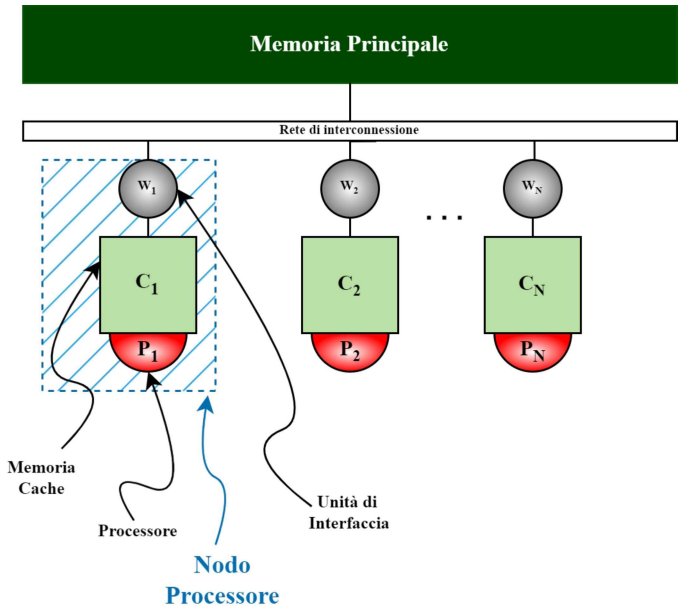


Figura 2.2: Architettura UMA

l'accesso di P_i a M_i è classificabile come accesso locale, in quanto per tale accesso non è necessario la struttura di interconnessione, a differenza del caso in cui P_i vuole accedere al modulo di memoria M_j (con $i \neq j$), questo tipo di accesso è detto remoto e ha una latenza che dipende da come sono organizzati i nodi processori.

L'architettura NUMA garantisce generalmente prestazioni migliori di UMA, infatti per quest'ultimo gruppo un solo processore alla volta può accedere alla memoria principale condivisa mentre gli altri processori devono attendere il proprio turno, la contesa tra più componenti di un'unica risorsa alla quale si può accedere uno alla volta porta alla creazione frequente di "colli di bottiglia", l'utilizzo di piccole zone di memoria esclusive e veloci adottato con sistemi NUMA permette di ridurre questo fenomeno di contesa di risorse. Inoltre l'architettura NUMA permette una semplificazione sostanziale del modello architetturale. Infatti per ora abbiamo inteso la struttura di interconnessione come una struttura che forniva due tipi di interconnessioni:

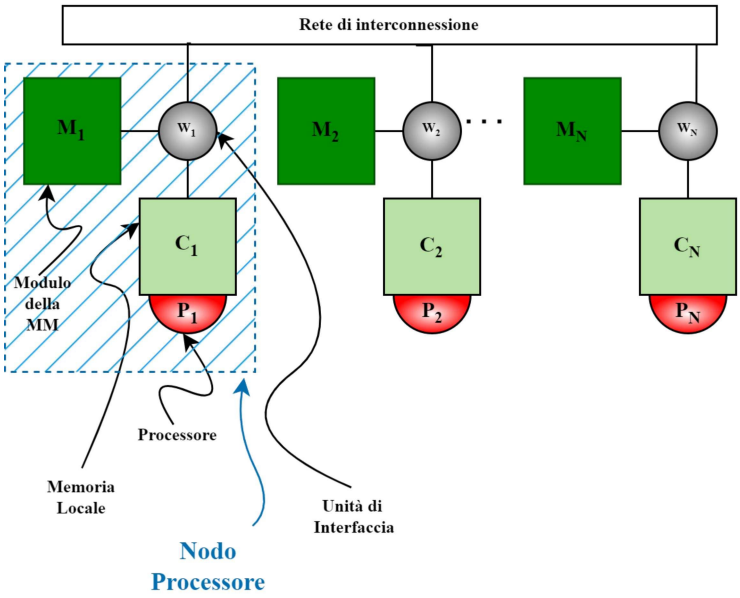


Figura 2.3: Architettura NUMA

- Struttura P-M: tra nodi processori e memoria condivisa principale per permettere ai nodi di eseguire accessi remoti alla memoria e per trasferire copie di blocchi di memoria situate nella memoria condivisa all'interno delle cache dei processori.

- Struttura P-P: tra nodi processori per lo scambio di messaggi di controllo intraprocessor

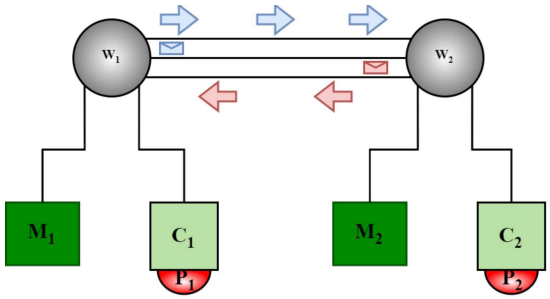


Figura 2.4: Scambio di due messaggi tra due processori in un sistema NUMA

Le architetture NUMA distribuendo i moduli della memoria principale nei vari nodi

processori del sistema permette di unificare queste due strutture in una sola riducendo il modello ad una rete di nodi processori. Per questo studio è stato quindi scelto di lavorare su una architettura NUMA, in cui in particolare la struttura di interconnessione è realizzata tramite un insieme di porte di comunicazione per ogni nodo connesse tra di loro, in particolare la connessione tra due porte di due nodi processori distinti avviene tramite canali full-duplex, ovvero l'informazione può essere trasmessa simultaneamente in entrambe le direzioni.

2.3 Organizzazione Memoria Locale

La memoria locale di ogni nodo processore è quindi composta nel seguente modo:

- Un modulo della memoria principale
- Una memoria cache in cui il processore può allocare le copie dei blocchi di memoria, generalmente tale memoria cache è organizzata in più livelli di cache, ma per semplificare il modello di studio il nostro sistema avrà una gerarchia di memoria con un solo livello di cache.

Bisogna evidenziare come l'utilizzo di memoria cache non è strutturalmente necessario per la creazione di un sistema multiprocessore, ovvero teoricamente si potrebbe creare un elaboratore multiprocessor che non fa uso di memoria cache, ma nella pratica un approccio simile è da scartare per due fondamentali motivi:

- A livello dell'elaborazione monoprocessore le memorie cache contribuiscono a diminuire il tempo di servizio per istruzione e il tempo di accesso ai dati.
- A livello complessivo del sistema le memorie cache sono fondamentali per ridurre la latenza di accesso in memoria sotto carico, ovvero a ridurre i conflitti tra i vari processori del sistema per l'accesso alla memoria condivisa.

Le memorie cache però introducono un problema noto come problema della Cache Coherence, cioè per assicurare la correttezza dell'elaborazione le informazioni contenute nelle cache dei vari nodi processori devono essere consistenti tra loro.

Capitolo 3

Cache coherence

Quando più processori contengono nella propria cache una copia di uno stesso blocco di memoria della memoria principale condivisa è necessario garantire che le copie rimangano consistenti tra loro e nei confronti del corrispondente blocco in memoria principale. In figura è stato schematizzato un esempio di inconsistenza della memoria

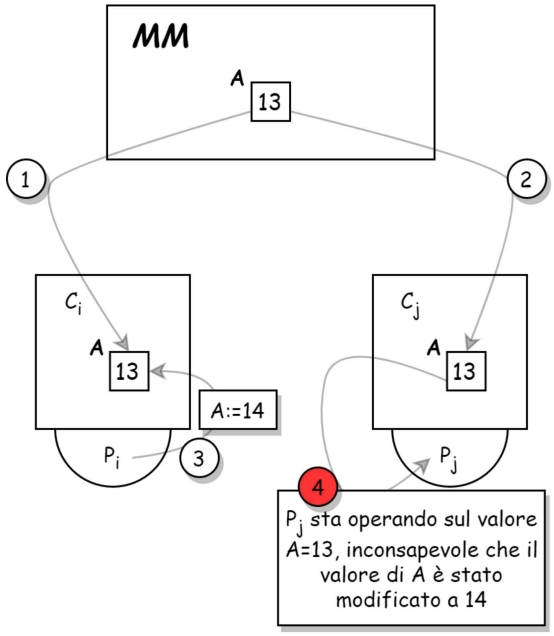


Figura 3.1: Problema della cache coherence

a livello delle cache locali. Il sistema multiprocesso raffigurato ha una memoria principale condivisa MM ed è costituito da più processori P_i, ognuno dotato di una propria cache C_i. In particolare in questo esempio una variabile A contenente un dato di tipo int con valore 13 viene trasferita da MM al nodo processore (1) e successivamente una copia della stessa variabile viene salvata anche in j (2). Fino ad ora abbiamo eseguito solo operazioni di lettura sulla variabile A e non si sono verificati problemi di coerenza. Le criticità iniziano con il passaggio (3) in cui il processore P_i esegue l'operazione A=A+1 e salva il risultato nella sua cache locale C_i, alla fine di questo passaggio siamo in una situazione in cui le copie della variabile A in C_j e nella MM non sono consistenti. In (4) il processore P_j esegue un operazione di modifica sulla sua copia della variabile A ed è in questo passaggio che a causa della mancanza di cache coherence viene meno la correttezza dell'elaborazione, infatti il processore P_j sta provando ad eseguire un operazione sulla copia in C_j della variabile A contenente il valore 13, mentre il valore aggiornato della variabile A su cui avrebbe dovuto lavorare il processore per mantenere la correttezza logica dell'elaborazione è il valore 14 salvato in C_i.

3.1 Meccanismi di cache coherence

Questi tipi di problema possono essere risolti con opportuni modelli di cache coherence, in particolare questo lavoro di tesi si concentra sulle tecniche di cache coherence di tipo automatico, ovvero implementate a livello firmware. I meccanismi utilizzati nei modelli di cache coherence automatica in ambienti multiprocesso possono essere di due tipi:

- aggiornamento, nel quale se una copia di un blocco di memoria viene modificata, allora tale modifica viene comunicata per diffusione a tutti i nodi che hanno

in memoria una copia di tale blocco permettendoli così di potere aggiornare i dati necessari.

- invalidazione, in questo meccanismo a seguito di una modifica si considera come copia valida di un dato blocco in memoria solo l'ultima copia che è stata modificata e si invalidano tutte le altre copie tramite l'invio di un segnale di invalidazione.

Se consideriamo l'esempio illustrato precedentemente (Figura 3.1) allora in base al meccanismo utilizzato nel passaggio (3) in cui P_i modifica il valore della variabile A ho due diversi comportamenti:

- con il meccanismo di aggiornamento, viene aggiornata anche la copia in C_j.
- con il meccanismo di invalidazione, viene inviato un segnale di invalidazione alla copia del blocco contenuta in C_j.

Il meccanismo di aggiornamento sembra essere l'approccio più naturale e più semplice, ma da un punto di vista del traffico di dati apportato alla rete risulta meno performante del meccanismo di invalidazione, infatti nel primo caso tutte le modifiche relative al blocco devono essere condivise con gli altri processori mentre nel secondo caso basta inviare un segnale di invalidazione con dimensioni molto piccole. In particolare modo il meccanismo di invalidazione è tanto più conveniente maggiore è il numero di processori nel sistema.

3.2 Politica di scrittura in memoria

Entrambi i tipi di meccanismo possono essere classificati a loro volta a seconda del metodo di scrittura in memoria che può essere di due tipi:

- Write-through, un blocco di dati nella memoria principale viene aggiornato appena una delle sue copie contenuta in una cache locale viene modificata. Perciò ad ogni modifica di una copia di un blocco di memoria corrisponde l'immediato aggiornamento del blocco corrispondente nella memoria principale. (La scrittura in una cache comporta la scrittura immediata in memoria principale).

- Write-back, ad una modifica locale di una copia di un blocco non segue immediatamente l'aggiornamento del blocco corrispondente nella memoria principale, in particolare di solito l'aggiornamento del blocco nella memoria principale avviene quando la copia modificata contenuta in una delle cache locali deve rimuovere la copia in questione, oppure quando la copia modificata deve soddisfare una richiesta di lettura da parte di un altro processore del sistema. (La scrittura in memoria principale non avviene per ogni scrittura in cache ma solo in caso di determinati eventi).

Il vantaggio della scrittura write-through è che semplifica la progettazione del sistema informatico, infatti la memoria principale ha sempre una copia aggiornata dei blocchi di memoria. Quindi, quando un nodo della rete del sistema multiprocesso richiede la lettura di un blocco tale operazione può essere sempre soddisfatta dalla memoria principale che può trasmettere i dati richiesti. Invece nel caso write-back, a volte i dati aggiornati si trovano nella cache di un processore e altre volte nella memoria principale. Quindi le richieste di lettura non possono essere sempre soddisfatte dalla memoria principale. Ma le architetture write-back hanno l'ovvio vantaggio di dovere compiere un numero minore di richieste di scrittura sulla memoria principale.

3.3 Protocolli di cache coherence

Analizziamo adesso come è possibile sfruttare questi meccanismi attraverso degli schemi generali di coordinamento che i vari nodi devono seguire: i protocolli di cache coherence. Un protocollo di cache coherence consiste in un insieme di stati a livello cache e a livello della memoria principale ed un insieme di messaggi di transizione che possono essere trasmessi sulla struttura di interconnessione al fine di mantenere coerenti le cache (ad esempio i messaggi di invalidazione).

3.3.1 Il caso di sistema uniprocessor

Introduciamo il funzionamento di un protocollo di cache-coherence partendo dal caso banale di un sistema uniprocessor dotato di cache, questo caso è molto utile per comprendere successivamente l'estensione al caso multiprocessor:

- sistema uniprocessor write-through: sono sufficienti due stati per mantenere la consistenza della cache e di conseguenza è sufficiente un unico bit di stato: 1 = *valid* e 0 = *invalid* (in verità il nome degli stati potrebbe essere fuorviante, infatti per questo caso sarebbe molto più evocativa la seguente associazione 1 = *blocco presente in cache* e 0 = *blocco non presente in cache*) . Inizialmente quando la cache è vuota ad ogni blocco nella memoria principale è assegnato il bit 0, quando il processore richiede una lettura di un blocco non presente in cache viene generato un fault e il blocco viene allora trasferito nella cache e il bit di stato relativo al blocco viene impostato ad 1. Le richieste di scrittura non comportano un cambiamento del bit di stato del blocco. Se un blocco presente in cache viene sostituito allora il bit di stato relativo viene impostato a 0.

- sistema uniprocessor write-back: oltre al bit di stato per indicare se un blocco è valid o invalid è necessario uno stato addizionale per indicare se il blocco è modificato o no, tale stato aggiuntivo è detto "dirty". Infatti se ad esempio un blocco in cache non dirty viene sostituito non è necessario aggiornare il relativo blocco nella memoria principale, mentre la sostituzione di un blocco dirty comporta sempre l'aggiornamento di tale blocco nella memoria principale.

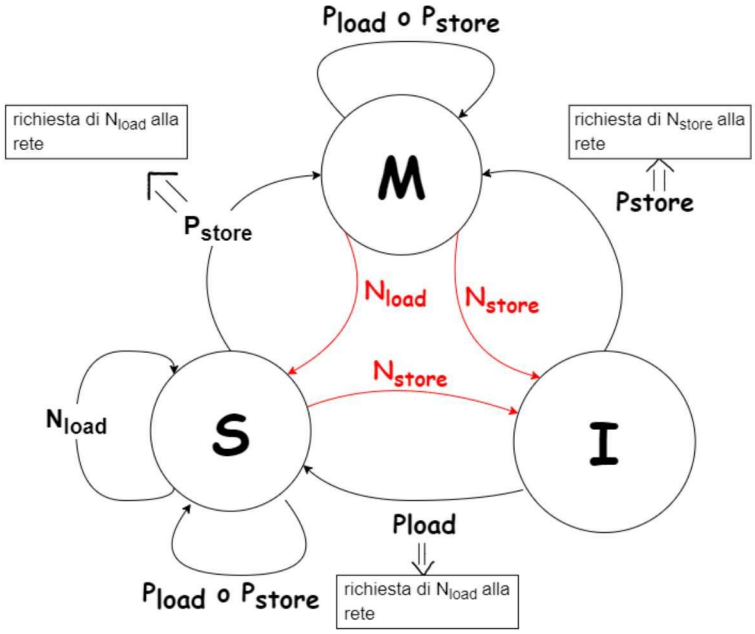
3.3.2 Il protocollo MSI

In un sistema multiprocessor invece un blocco ha uno stato in ciascuna cache, il protocollo principale di riferimento tra quelli basati su invalidazione e scrittura in memoria write-back è il protocollo MSI. Il nome del protocollo deriva dai possibili stati che un blocco contenuto in cache può assumere:

- Modified (o dirty), se una copia di un blocco è in stato dirty allora è l'unica copia valida del blocco. Inoltre le copie in stato modified sono le uniche copie ad avere il permesso di venire riscritte.

- Shared, se una copia di un blocco è in stato shared allora è una versione aggiornata del blocco ed è consistente con il blocco in memoria principale ed eventuali altre copie in altre cache del sistema in stato shared.

- Invalid, la copia è stata invalidata tramite un segnale di invalidazione. Dal punto di vista del sistema essere in stato invalid equivale a non essere presenti in cache.



Legenda:

Stati: M=Modified, S=Shared, I=Invalid

A —Pstore/load→ B : Un nodo dopo avere richiesto una P_{load} o una P_{store} è passato dallo stato A allo stato B.

A —Nstore/load→ B : E' giunta alla rete una richiesta di N_{load} o di N_{store} , ogni nodo in stato A nel sistema è passato allo stato B

: L'azione A comporta l'azione B

3.3.3 Transizioni in seguito a richieste di load e store

Il protocollo MSI oltre a specificare quali stati può assumere un dato blocco in cache descrive le varie transizioni di stato a seguito di richieste di load o di store. Quando si parla di richieste di lettura o di richiesta di scrittura è possibile distinguere il processo in due fasi, infatti quando un processore fa una richiesta di load o di store tale richiesta viene inizialmente destinata alla cache locale del nodo, chiamamo questa fase (a seconda del tipo di richiesta) P_{LOAD} o P_{STORE}, se ho un fault tale richiesta viene inoltrata da parte del nodo alla rete, chiamiamo tale fase con il termine N_{LOAD} o N_{STORE}.

- P_{LOAD} = richiesta di lettura da parte di un processore. La richiesta viene inizialmente destinata alla cache locale, se i dati richiesti non sono contenuti in cache ho un fault di lettura.

- P_{STORE} = richiesta di scrittura da parte di un processore. La richiesta viene inizialmente destinata alla cache locale, se i dati richiesti non sono contenuti in cache oppure sono contenuti in una copia senza il permesso di essere modificata ho un fault di scrittura.

- N_{LOAD} = richiesta di lettura da parte di un nodo, segue sempre da un fault di lettura in cache. Una N_{LOAD} comporta sempre il trasferimento di dati richiesti dalla cache C_j del nodo che soddisfa la richiesta alla cache C_i del nodo richiedente. Se la copia contenuta in C_j era in stato modified, allora bisogna aggiornare lo stato di tale copia a shared e aggiornare il blocco corrispondente in memoria principale.

- N_{STORE} = richiesta di scrittura da parte di un nodo, segue sempre da un fault di scrittura in cache. Una N_{STORE} comporta sempre il trasferimento di dati richiesti dalla cache C_j del nodo che soddisfa la richiesta alla cache C_i del nodo

richiedente inoltre il nodo j ottiene il permesso di scrittura per la sua copia del blocco in questione, quindi aggiorna lo stato della copia a modified. La copia contenuta in C_j deve essere invalidata, inoltre un messaggio di invalidazione deve essere propagato lungo tutte le eventuali copie del blocco in stato shared.

Vediamo adesso le possibili transizioni di stato in un protocollo MSI a seguito di una di queste richieste.

1. Richiesta di lettura da parte del processore P_i

   - CASO A: la copia salvata in C_i non è presente in cache oppure presente ma in stato invalid.

     AZIONI DA ESEGUIRE PER PERMETTERE LA LETTURA: a seguito della ricezione di fault di lettura, viene intrapresa una N_{LOAD}.

     TRANSIZIONI DI STATO: la copia salvata in C_i cambia il suo stato da invalid a shared. Se la copia salvata in C_j era in stato modified cambia il suo stato in shared.

   - CASO B: la copia salvata in C_i è in stato shared o modified

     AZIONI DA ESEGUIRE PER PERMETTERE LA LETTURA: nessuna.

     TRANSIZIONI DI STATO: nessuna.

2. Richiesta di scrittura da parte del processore P_i

   - CASO A: la copia salvata in C_i non è presente in cache oppure presente ma in stato invalid.

     AZIONI DA ESEGUIRE PER PERMETTERE LA SCRITTURA: a seguito della ricezione di fault di scrittura, viene intrapresa una N_{STORE}.

     TRANSIZIONI DI STATO: la copia salvata in C_i cambia il suo stato da invalid a modified. Se la copia salvata in C_j era in stato modified cambia

il suo stato in invalid. Se la copia salvata in C_j era in stato shared allora tutte le copie di quel blocco nel sistema vengono invalidate.

- CASO B: la copia salvata in C_i è in stato shared

  AZIONI DA ESEGUIRE PER PERMETTERE LA SCRITTURA: a seguito della ricezione di fault di scrittura (i dati sono aggiornati ma la copia non ha il permesso di venire riscritto) viene intrapresa una N_{STORE}. (La N_{STORE} comporta un trasferimento di dati non necessari, infatti viene traferita una copia uguale a quella già presente in cache. Perciò la N_{STORE} viene intrapresa unicamente per aggiornare lo stato della copia in cache da shared a modified, una possibile e frequente ottimizzazione del protocollo è quella di aggiungere alla richieste di LOAD e di STORE una richiesta di STATUS UPDATE.

  TRANSIZIONI DI STATO: la copia salvata in C_i cambia il suo stato da shared a modified. Se la copia salvata in C_j era in stato modified cambia il suo stato in invalid. Se la copia salvata in C_j era in stato shared allora tutte le copie di quel blocco nel sistema vengono invalidate.

Esistono altri protocolli di cache coherence, in particolare esistono varianti ottimizzate del protocollo MSI tra cui in particolare il protocollo MESI e MOESI che sfruttano l'utilizzo di stadi aggiuntivi per migliorare le prestazioni, ma in questo studio ci limitiamo a MSI per non complicare eccessivamente l'analisi delle varie tecniche di cache coherence.

3.4 Tecniche di cache coherence

Analizziamo ora con quali tecniche possono essere implementati i protocolli di cache coherence, queste si suddividono in due categorie principali:

- Snoopy-based, tecnica utilizzata principalmente in sistemi UMA, prevede l'utilizzo di un bus centrale comune struttura di interconnessione.
- Directory-based, tecnica utilizzata principalmente in sistemi NUMA, prevede l'utilizzo di una directory per tenere traccia delle varie copie dei blocchi di memoria.

3.4.1 Snoopy-based

Le tecniche snoopy-based sono una soluzione semplice ed efficace per risolvere il problema della cache coherence: ogni transazione, in particolare ogni richiesta di lettura o scrittura da parte di un nodo, avviene su un bus centrale a cui si interfecciano tutti i nodi. Tali segnali di trasmissione vengono fatti "riecheggiare" a tutti i nodi del sistema che quindi osservano (da qui il termine "snoop") tutte le transazioni di memoria del sistema, se tali transazioni minacciano la consistenza degli stati cache, vengono invalidate le proprie copie locali. La problematica di queste tecniche è che il bus centrale non ha una larghezza di banda sufficiente a reggere un numero elevato di processori.

3.4.2 Directory-based

Le tecniche principali di cache coherence che permettono una maggiore scalabilità del sistema si basano sul concetto di directory. Una directory consiste nella lista di tutte le locazioni delle varie copie in cache di un determinato blocco associate al loro stato di directory (il termine stato di directory è impreciso, serve per sottolineare che lo stato di una copia salvato in directory non corrisponde allo stato, della stessa copia, salvato in cache, in particolare lo stato di directory fornisce meno informazioni sulla copia e indica solo se la copia è valida o no), perciò il suo scopo è quello di fornire al sistema le seguenti informazioni:

- le informazioni necessarie a determinare la posizione all'interno della rete delle varie copie di un blocco.
- le informazioni relative alla validità delle varie copie di un blocco.

Attraverso queste informazioni il sistema può correttamente fare comunicare tra loro i vari nodi contenente le copie di un dato blocco al fine di mantenere la coerenza tra le copie. Possiamo vedere una directory come una tabella composta da tante righe quante sono i blocchi in memoria principale del sistema, in cui ogni riga, chiamata anche directory entry, contiene un insieme di record di due elementi: un puntatore ad una cache che contiene una copia e lo stato di tale copia.

3.5 Modelli di directory

Prima di procedere all'analisi dei vari schemi in cui una directory può essere organizzata introduciamo una breve notazione utile per distinguere i vari nodi durante i loro processi di comunicazione a seguito di un fault per l'accesso ad un dato blocco.

- nodo home = nodo nella cui memoria principale è allocato il blocco di cui è richiesto l'accesso ai dati.
- nodo local = nodo in cui si è verificato il fault e che quindi richiede il trasferimento del blocco nella propria cache.
- nodo dirty = eventuale nodo che contiene l'unica copia aggiornata del blocco nella sua cache (quindi il nodo dirty contiene nella sua cache una copia del blocco in stato modified).

I vari meccanismi directory-based vengono classificati in base all'organizzazione della directory e alla sua distribuzione nel sistema:

- memory based: la directory è contenuta esclusivamente nella memoria principale. Perciò tutte le informazioni di directory necessarie ad individuare le varie copie sono tutte contenute nel nodo home.

- cache based: la directory è distribuita nella memoria principale e nelle varie cache locali del sistema. In questo tipo di organizzazione si tiene traccia delle varie copie di un blocco attraverso una catena di puntatori, per questo motivo questi tipi di schemi di cache-coherence sono anche chiamati schemi di chained directories. Il nodo home contiene semplicemente un puntatore ad una copia del blocco contenuta in una delle cache del sistema che a sua volta conterrà un altro puntatore ad un altra copia di tale blocco e così via fino a giungere all'ultima copia nella cui cache risiederà il puntatore di chain termination che indica che la catena di puntatori è terminata.

3.5.1 Memory based

Le principali tecniche di cache-coherence basate su directory memory-based sono due: organizzazione della directory full-map directories (o organizzazione a vettore di bit completo) e limited directories.

Full-map directories

Nei protocolli con directory full-map per ogni entry vengono utilizzati N bit di presenza (dove N è il numero di nodi del sistema) ed un bit detto "dirty bit". Ognuno degli N bit di presenza è associato ad un nodo della rete ed indica se tale nodo ha o meno nella propria cache una copia valida del blocco. Il dirty bit serve invece per indicare se nella rete esiste una copia in stato dirty (o stato modified). Perciò se il dirty bit vale 1, allora solo il nodo dirty ha in cache una copia valida del blocco, quindi solo uno degli N bit di presenza sarà impostato ad 1 (infatti è presente una sola copia valida del blocco nel sistema). Ricordiamo che invece in ogni

cache saranno presenti due bit di stato per ogni copia, tali bit indicano lo stato della copia secondo il protocollo MSI quindi la copia può essere in stato modified, shared o invalid. Analizziamo passo per passo come uno schema full-map si comporta in caso di fault di cache in un nodo generico i: Il controllore della coerenza del nodo i determina tramite l'indirizzo del blocco se il nodo home coincide con il nodo local (il nodo i dove si è verificato il fault) oppure se il nodo home che d'ora in poi supporemmo essere il nodo j è un nodo remoto. Se $i = j$ allora la richiesta viene gestita localmente, mentre se il nodo home è remoto bisogna inoltrare la richiesta alla rete che dovrà correttamente fare arrivare la richiesta al nodo j. Tralasciamo il caso in cui il nodo local e il nodo home corrispondano e supponiamo che il nodo home sia il nodo j. Il nodo i tramite la rete deve inviare una richiesta che dovrà correttamente essere inoltrata al nodo j, infatti è il nodo j a contenere le informazioni di directory necessarie a rintracciare la versione aggiornata del blocco richiesto. Nel caso in cui la richiesta sia una richiesta di load seguono i seguenti step:

- se dirty bit=0, allora il blocco richiesto verrà inviato dalla memoria principale, quindi dal modulo di memoria M_j vengono inviati i dati al nodo i e l'i-esimo bit del vettore di presenza nella entry associata al blocco viene impostato a 1.

- se dirty bit=1, allora il nodo home contenente la directory relativa al blocco richiesto invia al nodo un messaggio contenente l'identità del nodo in stato modified contenente la versione aggiornata del blocco, a questo punto il nodo local può correttamente richiedere il trasferimento del blocco al nodo dirty, il nodo dirty a questo punto deve correttamente modificare il suo stato da modified in shared, poi in linea con la politica write-back oltre ad inviare i dati del blocco al nodo local deve inviarli alla memoria principale contenuta nel nodo home. A questo punto il controllore della coerenza nel nodo home si deve occupare di impostare il bit dirty del blocco a 0 poichè non esiste più

nella rete una copia in stato dirty ed impostare l'i-esimo bit del vettore di presenza ad 1.

Nel caso in cui la richiesta sia di store:

- se dirty bit=0, allora il nodo home ricevuta la richiesta dovrà occuparsi di trasferire il blocco ma anche di invalidare tutte le copie in stato shared nella rete, perciò intanto invia il blocco di dati al nodo local, poi invia un messaggio di richiesta di invalidazione a tutti nodi il cui relativo bit di presenza nella directory entry è relativo a 1. A questo punto il nodo home attende di ricevere un messaggio di acknowledgment che confermi l'avvenuta invalidazione da ogni nodo, poi può procedere a impostare tutti i bit nella directory entry a 0 tranne il dirty bit e l'i-esimo bit di presenza che vengono impostati a 1. Una volta che le informazioni di directory sono state correttamente aggiornate il nodo home invia il permesso di scrittura al nodo i, che quindi può cambiare lo stato della copia in cache in modified. Si ponga attenzione che la ricezione di acknowledgment permette al sistema il mantenimento della cache coherence anche se per qualche motivo uno dei messaggi di invalidazione non giunge a destinazione, infatti il permesso di scrittura viene rilasciato solo quando il nodo home prende atto attraverso i messaggi di ACK che tutte le copie sono state invalidate.

- se dirty bit=1, allora il nodo home inoltra la richiesta al nodo dirty cioè al nodo il cui bit di presenza vale 1. Il nodo dirty ricevuta la richiesta invia i dati al nodo local (il nodo local una volta ricevuti i dati e trasferiti in cache dovrà anche aggiornare lo stato della copia in modified), invalida lo stato della copia in cache e invia un ACK al nodo home che a questo punto può aggiornare le informazioni di directory.

Un dettaglio molto importante da evidenziare è che quando un nodo invia una richiesta di lettura o di scrittura il relativo processore interno al nodo viene messo in stallo fino al soddisfacimento della richiesta, perciò il lasso di tempo tra l'invio della richiesta e il suo completamento è un periodo di inutilizzazione del processore. I protocolli di full-map directory sono la soluzione più naturale per tenere traccia delle copie dei blocchi nella rete ma presentano una grave criticità legata alla quantità di memoria richiesta per le informazioni di directory, ovvero l'overhead in memoria richiesto per le informazioni di directory cresce quadraticamente rispetto al numero N di processori nella rete, causando un incompatibilità del protocollo con reti molto estese. Infatti supponiamo che la quantità di memoria principale distribuita nella rete, e quindi il numero di entry directory, cresca linearmente ad N, a sua volta anche la dimensione delle entry scala linearmente ad N (ho N bit di presenza), quindi complessivamente il carico di memoria richiesto per directory full-map è $\Theta(N^2)$.

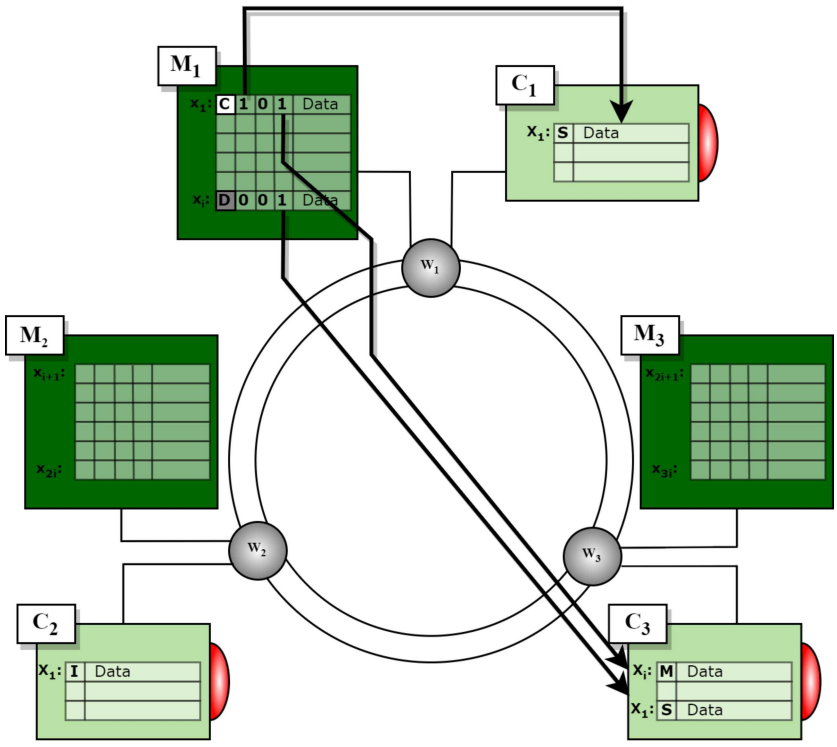


Figura 3.2: Esempio di sistema con 3 nodi con directory full-map

Limited directories

I protocolli di limited directory sfruttano gli stessi principi e meccanismi dei protocolli full-map ma apportano delle modifiche che permettono di ridurre il carico sulla memoria principale. In particolare ogni blocco può avere solo un numero limitato di copie salvate nelle varie cache della rete e se quando il numero di copie massimo è stato raggiunto arriva una richiesta di lettura del blocco da un altra cache allora prima di potere trasferire il blocco il sistema deve invalidare una delle copie in stato shared. Le informazioni di directory per un organizzazione a puntatori limitati richiede una quantità di memoria che cresce come $\Theta(NlogN)$, infatti ogni entry richiede $log_2(N)$ bit per ogni puntatore, cioè il numero di bit necessario a rappresentare l'identificativo del processore in codifica binaria. Perciò questa modifica nell'organizzazione fornisce una riduzione del carico in memoria al costo

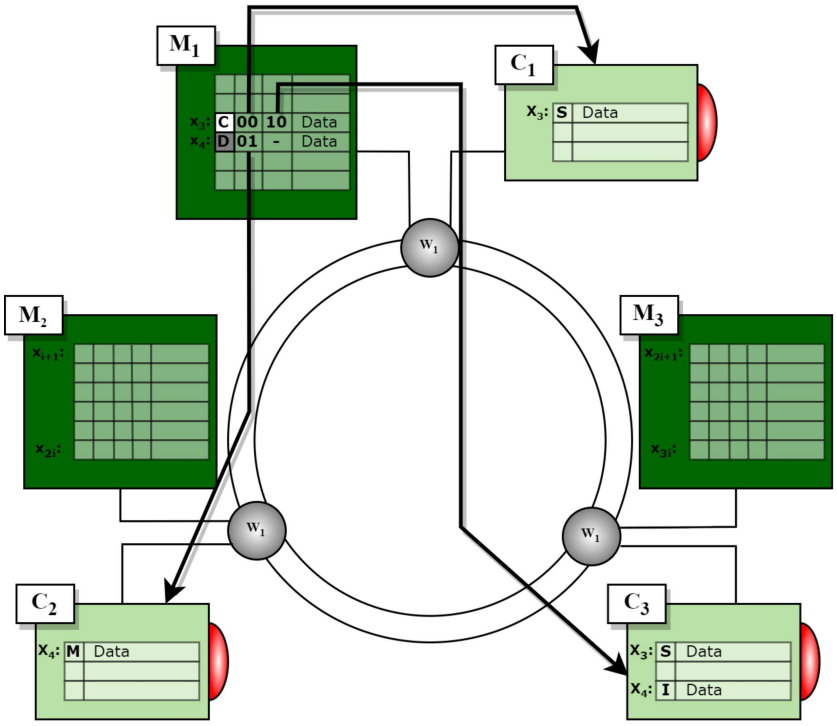


Figura 3.3: Esempio di sistema con 3 nodi con limited directory

di ridurre le prestazioni dell'elaborazione. In verità la riduzione di prestazioni può essere minima infatti in ambienti multiprocesso l'elaborazione parallela può essere organizzata in modo che normalmente solo un numero piccolo di cache mantiene una copia aggiornata dello stesso blocco, quindi l'eventualità di raggiungere la situazione di esaurimento di puntatori liberi diventa sufficientemente rara da garantire una perdita di prestazioni rispetto allo schema full-map relativamente piccola.

3.5.2 Cache based: chained directories

L'utilizzo di una directory organizzata come una single linked list offre il vantaggio di ridurre l'overhead di spazio in memoria (è necessario un puntatore di $log_2(N)$ bit per ogni entry) come nel caso delle limited directories a $\Theta(Nlog(N))$ senza però dover rinunciare alla possibilità di avere una copia valida di uno stesso blocco in ogni cache. Vediamone brevemente il funzionamento analizzando un esempio: Supponiamo di trovarci in una situazione iniziale in cui non ci sono copie condivise di un dato blocco X. In tale situazioni la catena di puntatori relativa al blocco X è costituita unicamente dal puntatore nullo di chain termination in memoria principale, ad indicare quindi che non ci sono nodi che condividono il blocco. Se a questo punto un nodo N_1 richiede la lettura di X, la memoria manderà alla cache C_1 una copia di X e le direttive per l'aggiornamento del puntatore, le informazioni di directory verranno aggiornate nel seguente modo:

- il puntatore in MM punterà a C_1.
- il puntatore in C_1 sarà nullo (CT).

A sua volta se anche un nodo N_2 richiede il blocco X, la memoria manderà alla cache C_2 una copia di X e le direttive per l'aggiornamento del puntatore, le informazioni di directory verranno aggiornate nel seguente modo:

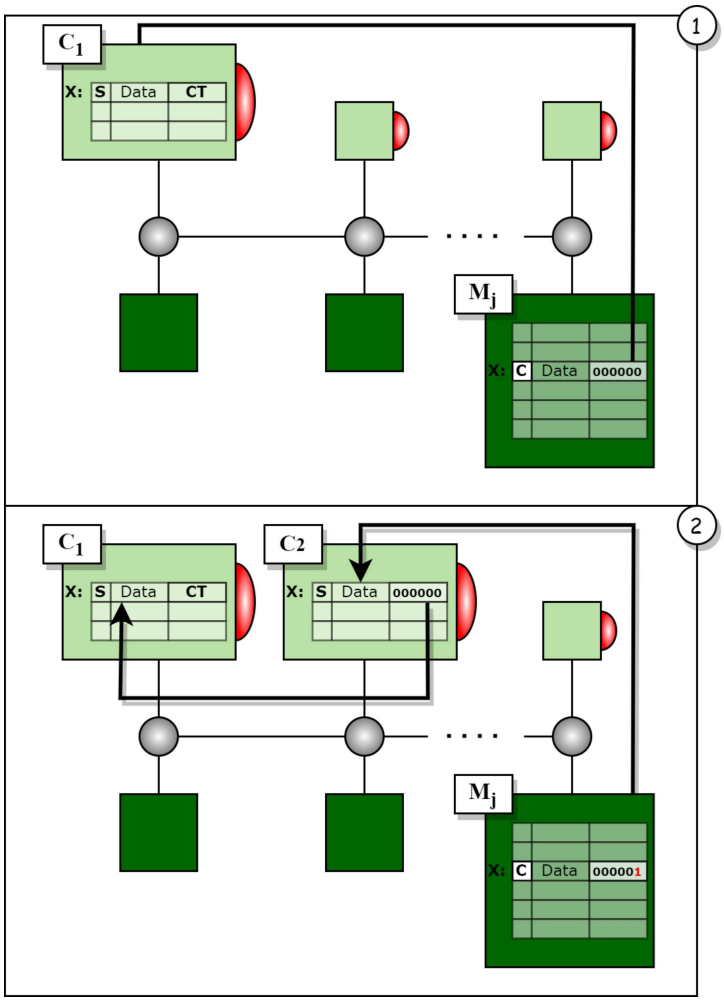


Figura 3.4: Lettura di un blocco in un sistema con directory cache-based

- il puntatore in MM punterà a C_2.
- il puntatore in C_2 punterà a C_1.

Ripetendo questo processo tutte le cache possono entrare in possesso di una copia di X a seguito di una richiesta di lettura. Se invece un nodo richiede il permesso di scrittura sul blocco X, allora è necessario inviare un messaggio di invalidazione lungo la catena di cache che detiene una copia del blocco, il processore del nodo local quindi dovrà rimanere in stallo fino a quando il nodo contenente il puntatore di chain termination non inoltra il messaggio ACK di avvenuta invalidazione. Una possibile complicazione consiste nell'eventualità che una cache debba rimpiazzare per motivi di spazio una copia di un blocco in stato shared. Infatti supponiamo che un dato blocco con locazione X è stato letto prima dal nodo N_1, poi dal nodo N_2 e così via fino al nodo N_N. Ogni nodo i-esimo N_i avrà nella sua directory un puntatore al nodo N_{i-1} (escluso il nodo N_1 che avrà il puntatore di chain termination (CT)). Supponiamo adesso che la cache C_k sia piena e che quindi per leggere un nuovo blocco non contenuto in cache deve prima liberarsi di una copia salvata di un blocco, in particolare supponiamo che si liberi della copia di locazione X, adesso la catena di puntatori che teneva traccia delle copie del blocco X è interrotta, infatti ora il puntatore in C_{k+1} punta alla cache C_k che non contiene più una copia valida del blocco, inoltre adesso espellendo la copia X da C_k sono state rimosse anche le informazioni di directory necessarie per rintracciare la copia in C_{k-1} a cui puntava C_k. In questa situazione esistono due soluzioni:

- Con l'espulsione del blocco viene inviato un messaggio lungo la catena per aggiornare il puntatore in C_{k+1} in modo che punti non più a C_k ma a C_{k-1}.
- Prima di espellere il blocco vengono invalidate tutte le copie contenente la copia di locazione X lungo il tratto di catena che partiva da C_k, quindi vengono invalidate tutte le copie a partire da quella in C_{k-1} fino alla copia in C_1.

Il secondo caso è la scelta maggiormente adottata poichè implica una minore complicazione del protocollo

3.5.3 Osservazioni conclusive sull'organizzazione della directory

Sintetizziamo i principali vantaggi e svantaggi dei vari schemi di directory proposti

- Full-map: è lo schema migliore a livello di prestazioni, ma l'elevato overhead di memoria lo rende incompatibile con sistemi con un elevato numero di nodi.
- Limited directories: Overhead di memoria ridotto, rispetto ad uno schema full-map le prestazioni sono ridotte a causa del numero limitato k di puntatori, il calo di prestazioni è maggiore tanto quanto è la probabilità che un blocco venga condiviso da un numero superiore a k di nodi.
- Chained directories: Overhead di memoria ridotto, ma , rispetto ad uno schema full- map le prestazioni sono ridotte a causa dell'eventualità di dovere espellere una copia in stato shared per motivi di spazio, il vantaggio dello schema è che tutte le cache possono mantenere allo stesso tempo una copia valida di uno stesso blocco.

Capitolo 4

Alcuni risulati notevoli e sinossi conclusiva

In questo capitolo vengono trattati alcuni problemi riguardanti i sistemi multiprocessore. Per non complicare ulteriormente l'analisi è stato scelto di lavorare su sistemi con topologia ad anello. Le reti ad anello ad N nodi vengono indicate con RNG_N e hanno una particolare importanza teorica poichè i risultati ottenuti su tali reti sono significativi anche per molti gruppi di reti che utiizzano topologie più complesse con maggiori prestazioni.

4.1 Routing di messaggi in una rete ad anello

4.1.1 Il paradigma del carosello

Concentriamo ora la nostra attenzione adesso sulla comunicazione, e quindi lo scambio di messaggi, tra processori in una rete ad anello, in particolare consideriamo un problema di routing dove sono inizialmente distribuiti K messaggi tra gli N nodi di una RNG_N. Un messaggio è un record $\langle i, j, B\rangle$ dove $i \in \{0, ..., N-1\}$ specifica il

nodo da cui parte il messaggio, $j \in \{0, ..., N-1\}$ specifica il nodo di destinazione del messaggio e B è il corpo del messaggio che che deve essere inviato. Supponiamo che ogni nodo abbia una coda di emissione per contenere i messaggi in attesa di essere spediti ed una coda di destinazione per accumulare i messaggi ricevuti. Descriveremo un algoritmo notevole basato su uno schema chiamato il paradigma del carosello. Il nome dell'algoritmo discende dalla metafora di vedere un anello di processori come un carosello con N carrozze che ruotano nello stesso verso lungo l'anello, queste carrozze possono caricare e scaricare messaggi quando giungono in corrispondenza di un nodo, ma ogni carrozza può trasportare un solo messaggio alla volta. La metafora del carosello è particolarmente adatta per rendere efficacemente come viene organizzata la comunicazione intraprocessori nel suddetto algoritmo di routing, per questo motivo per semplificare la descrizione dell'algoritmo utilizzeremo la modellazione astratta della RNG_N come un carosello. Inizialmente le carrozze sono vuote e sono collocate ciascuna in corrispondenza di un nodo. Le carrozze trasportano in parallelo i messaggi ruotando nello stesso senso lungo l'anello tramite il seguente meccanismo (i seguenti passi vengono iterati ad ogni ciclo di clock e vengono svolti in parallelo):

1. Se la carrozza è appena arrivata al nodo i ha un messaggio destinato a quel nodo allora la vettura si libera del messaggio lasciandolo nella coda di destinazione locale. Altriimenti il messaggio rimane sulla vettura

2. Se la carrozza che è appena arrivata al nodo i è vuota e la coda di emissione locale del nodo non è vuota, allora il messaggio in testa alla coda viene trasferito sulla carrozza, almeno che la destinazione del messaggio non sia lo stesso nodo i, in tal caso il messaggio viene direttamente trasferito sulla carrozza, almeno che la destinazione del messaggio non sia lo stesso nodo in tal caso

il messaggio viene direttamente traferito dalla coda di emissione alla coda di destinazione.

3. Ogni carrozza collocata su un generico nodo i lascia il nodo e si reca al nodo $i+1$

La correttezza e l'analisi della complessità sono date dalla seguente proposizione

Proposizione 4.1. *(Instradamento secondo paradigma del carosello)*
L'algoritmo di routing del carosello spedisce correttamente tutti i messaggi a destinazione in $N+K-2$ iterazioni al caso peggiore. Il tempo di instradamento è quindi $T = O(N+K)$ cicli di clock.

Dimostrazione. Il tempo di instradamento T può essere suddiviso in tempo di viaggio sulla vettura v e in tempo di attesa nella coda di emissione w. Perciò $T = v + w$. Tempo di viaggio v (misurato in cicli di clock): Un messaggio una volta caricato su una vettura giunge a destinazione attraversando un arco alla volta ad ogni iterazione. Le iterazioni necessarie per fare giungere a destinazione un messaggio sono $j-i$ se $i \leq j < N$, altrimenti sono $N-(i-j)$ se $j < i$

$$v = \begin{cases} j-i, & \text{se } i \leq j \\ N-(i-j), & \text{se } j < i \end{cases} \tag{4.1}$$

Si può riscrivere come

$$v = (j-i) \ mod \ N \tag{4.2}$$

In ogni caso ho che $v \leq N-1$
Tempo di attesa nella coda di emissione w (misurato in cicli di clock) : Consideriamo un messaggio μ nella coda di emissione locale dell'i-esimo nodo. Per ogni iterazione se il messaggio μ non lascia la coda può essere individato un messaggio μ'

responsabile di non avere permesso la dipartita del messaggio μ, tale messaggio μ' può essere o un messaggio che si trovava già sulla carrozza quando questa è giunta al nodo e con una destinazione $j \neq i$ oppure un messaggio in coda da più tempo di μ, in ogni caso un messaggio μ' non può essere responsabile due volte del mancato accesso di μ alla carrozza, quindi al caso peggiore μ rimarrà in attesa nella coda per $K-1$ iterazioni. Siccome $v \leq N-1$ e $w \leq K-1$ complessivamente si ha che

$$T = (v + w) \leq N + K - 2 \tag{4.3}$$

□

4.1.2 Contesa del permesso di scrittura

Supponiamo che in un generico ciclo di clock T due nodi inoltrino una richiesta di scrittura su uno stesso blocco di locazione X, tale richiesta di scrittura è un messaggio che ha come destinatario il nodo home contenente il modulo della MM dove si trova il blocco in questione. Vediamo come una variante del routing a carosello può essere implementata per gestire correttamente lo scambio di messaggi di store in caso di contesa di uno stesso blocco. Naturalmente solo un processore alla volta potrà ottenere il permesso di scrittura sul blocco, perciò le richieste di store verranno accumulate in una coda di destinazione del nodo home e saranno soddisfatte una alla volta. Si consideri quindi una situazione iniziale in cui sono distribuiti, tra gli N nodi del RNG_N, K richieste di store su uno stesso blocco (ogni richiesta è inoltrata da un nodo differente perciò $K \leq N$). Osserviamo preliminarmente che i messaggi non dovranno fare coda per essere caricati sul carosello, perciò il tempo di instradamento T corrisponde al tempo di viaggio v, quindi $T = V$. Se utilizzassimo l'algoritmo di routing a carosello nella sua forma classica sarebbe banale osservare che dall'equazione (4.3) discende che tutti i messaggi arriverebbero a destinazione

entro $N-1$ cicli di clock indipendentemente dalla posizione del nodo home e dalla distribuzione dei messaggi. E' però necessario modificare l'algoritmo a causa di una necessità a livello di programmazione dell'elaborazione di potere risalire al vincitore della contesa esclusivamente dall'identificativo dei nodi. In seguito viene quindi proposto una versione modificata dell'algoritmo di routing a carosello in cui le richieste di store vengono soddisfatte in ordine decrescente dell'identificativo dei nodi da cui proviene la richiesta. Ridefiniamo con maggiore precisione il problema dello scambio di messaggi di store:

Problema

- La rete è un RNG_N in cui i nodi sono distribuiti in ordine antiorario rispetto al proprio identificativo.
- Modelliamo il sistema di comunicazione intraprocessori come nel paradigma del carosello, in particolare supponiamo che il carosello ruoti in senso antiorario.
- Bisogna sviluppare un algoritmo che si occupi di instradare e fare giungere a destinazione K messaggi di store distribuiti all'interno della rete inoltrati da nodi diversi nello stesso ciclo di clock. Una richiesta di store è un record $\langle i, j, S\rangle$ dove $i \in \{0, ..., N-1\}$ specifica il nodo da cui parte il messaggio, $j \in \{0, ..., N-1\}$ specifica il nodo di destinazione del messaggio e S è il corpo del messaggio di store.

Idea algoritmo

Supponiamo che il nodo home in possesso del modulo di MM contenente il blocco sia il nodo N_j, l'idea è che bisogna fare in modo che i messaggi con il campo i maggiore di j arrivino subito a destinazione, mentre i messaggi con $i < j$ compiano un intero giro dell'anello prima di potere fermarsi nel proprio nodo di destinazione. In questo

modo i messaggi arrivano al nodo j in ordine dal numero di identificativo maggiore a quello minore.

Algoritmo

L'algoritmo ideato per risolvere il problema della contesa di scrittura su un blocco è un algoritmo di routing a carosello in cui però la struttura tipica del record tipica per un messaggio viene modificata, infatti appena la richiesta di store viene generata viene concatenato al messaggio un bit che chiamiamo "bit di giro (BG)", perciò la struttura di una richiesta di store che sta per essere inoltrata sarà la seguente $\langle i, j, S\rangle.\langle BG\rangle$ (dove il punto indica la concatenazione e BG il bit di giro). Il bit di giro può assumere valore 1 o valore 0, l'idea è quella che tale campo serva per indicare se il messaggio debba fare un giro attorno all'anello (BG=1) o altrimenti giungere direttamente a destinazione (BG=0), perciò in conseguenza a quanto precedentemente detto il valore di BG sarà il risultato booleano di $i \leq j$ Il meccanismo dell'algoritmo è molto simile a quella utilizzata nella versione canonica del routing a carosello (i seguenti passi vengono iterati ad ogni ciclo di clock e vengono svolti in parallelo):

1. Se la carrozza che è appena arrivata al nodo i ha un messaggio di store destinato a quel nodo, allora in base al valore di BG seguono le seguenti azioni:

   - BG=0, il messaggio viene lasciato nella coda di destinazione del nodo.
   - BG=1, il campo BG viene azzerato e il messaggio rimane nella carrozza.

   Altrimenti, se invece la richiesta di store non è destinata al nodo i, il messaggio rimane nella carrozza.

2. Se la vettura che è appena arrivata al nodo i è vuota e la coda di emissione locale del nodo non è vuota, allora il messaggio in testa alla coda viene trasferito sulla vettura. (Si osservi che la coda di emissione nel nostro problema

ha al massimo un messaggio di store, inoltre se la richiesta ha $i = j$ questa viene caricata comunque sulla carrozza, mentre nella versione base del routing a carosello un messaggio con $i = j$ sarebbe stato spedito direttamente nella coda di destinazione del nodo).

3. Ogni carrozza collocata su un generico nodo i lascia il nodo e si reca al nodo $i + 1$.

La correttezza e l'analisi della complessità sono date dalla seguente preposizione

Proposizione 4.2. *L'algoritmo spedisce correttamente tutte le richieste di store in ordine decrescente dell'identificativo del nodo di provenienza in* $2N - 1$ *iterazioni nel caso peggiore.*

Dimostrazione. Calcoliamo il tempo di instradamento T di un generico store, per ognuno dei seguenti casi:

- se $i < j$ allora BG verrà inizialmente impostato a 1, perciò il messaggio dovrà compiere prima un giro attorno all'anello, in N iterazioni, poi una volta fatto il giro e azzerato BG il messaggio arriverà a destinazione in $j - i$ iterazioni, quindi $T = N + (j - i)$.

- se $i > j$ allora BG verrà inizialmente impostato a 0, perciò il messaggio dovrà semplicemente giungere a destinazione percorrendo un arco alla volta, perciò $T = N - i + j$ che può essere riscritto come $T = N + (j - i)$.

- se $i = j$ allora BG verrà inizialmente impostato a 0, ma il messaggio nonostante il nodo di destinazione coincide con il nodo di emissione, verrà in ogni caso inizialmente caricato sulla carrozza, perciò per giungere a destinazione dovrà compiere un giro intero dell'anello impiegando $T = N$ iterazioni, siccome in questo caso $j - i = 0$ posso riscriverlo come $T = N + (j - i)$.

Complessivamente ho che

$$T = N + (j - i) \quad \forall i, j \tag{4.4}$$

Perciò al caso peggiore, cioè nel caso in cui $j - i$ è massimo che si verifica quando $j = N - 1 \wedge i = 0$, ho che il tempo di instradamento è $T = 2N - 1$ □

4.2 Conclusioni

4.2.1 Sommario conclusivo

In questo lavoro di tesi abbiamo sondato le varie opzioni architetturali per un sistema multiprocesso e abbiamo analizzato quali fossero le più adatte per un sistema multiprocesso con un elevato numero di nodi, riassiumiamo le scelte effettuate:

- Distribuzione della memoria principale (NUMA vs UMA) : **NUMA**
- Meccanismi di cache-coherence (aggiornamento vs invalidazione) :**invalidazione**
- Politica di aggiornamento della Main Memory (write-through vs write-back) : **write-back**
- Tecnica di cache coherence (snoopy-based vs directory-based) : **directory-based**
- Organizzazione della directory : **limited directory** oppure **chained directory**

4.2.2 Sviluppi futuri

Un naturale sviluppo di questo lavoro di tesi è confermare sperimentalmente la validità delle scelte effettuate . Quindi vedere come simulando sistemi multiprocesso

con un numero elevato di nodi processore le opzioni selezionate nel suddetto studio si rivelino maggiormente efficaci. Dal punto di vista più teorico invece un estensione di questo lavoro è approfondire le varie possibilità per la rete di interconnessioni, soprattutto da un punto di vista topologico, approfondire i protocolli degli stati in cache più complessi, in particolare i protocolli MESI e MOESI ed infine allargare il punto di vista dello studio: da reti di nodi monoprocessore a sistemi con nodi con più di un processore. In particolare quest'ultimo possibile tema di sviluppo tratta un'interessante tecnica di cache coherence che si basa su un approccio ibrido tra tecniche snoopy-based e directory-based, in cui la coerenza delle cache dei processori viene mantenuta a livello locale del nodo tramite comunicazione broadcast su un bus centralizzato, mentre la coerenza delle memorie locali di ciascun nodo viene mantenuta tramite un approccio directory-based.

Bibliografia

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. *An evaluation of directory schemes for cache coherence. In ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pag. 353–362, New York, NY, USA, 1998. ACM.

[2] J.-L. Baer and W.H. Wang. *On the inclusion properties for multi-level cache hierarchies. In ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pag. 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[3] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.*

[4] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. *Directory-based cache coherence in large-scale multiprocessors*, Massachusetts Institute of Technology, 1990.

[5] David J. Lilja. *Cache coherence in large-scale shared memory multiprocessors: Issues and comparisons*, acm computing surveys, pag.303–338, 1993.

[6] Gianfranco Bilardi. *Parallel Architectures*, Università di Padova, pag 1-19, 2020.

[7] Yan, Solihin. *Fundamentals of parallel multicore architecture.*

[8] Sorin, Daniel J.; Hill, Mark D.; Wood, David Allen (2011-01-01). *A primer on memory consistency and cache coherence.* Morgan Claypool Publishers.