Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Dipartimento di Informatica

Laurea Specialistica in Informatica

Tesi di Laurea

Modello dei Costi delle Tecniche di Cache Coherence nelle Architetture Multiprocessor

Candidato
Silvia Lametti

Relatore
Marco Vanneschi

Controrelatore
Francesco Romani

Anno Accademico 2009 - 2010

Ad Andrea

Quelli che s'innamoran di pratica sanza scienzia
son come 'l nocchier ch'entra in navilio senza timone o bussola,
che mai ha certezza dove si vada.
(Leonardo da Vinci)

Ringraziamenti

Questa pagina è dedicata a tutte le persone che mi sono state accanto e mi hanno incoraggiata durante gli studi universitari, fino al raggiungimento di questo traguardo per me così tanto importante.

Desidero ringraziare il prof. Marco Vanneschi per la sua disponibilità e l'attenzione con cui mi ha seguita, per avermi trasmesso la passione per l'Informatica e, specialmente in questi ultimi anni, per avermi fatto conoscere il mondo della Ricerca.

Vorrei inoltre ringraziare il prof. Marco Danelutto e Daniele Buono per la magnifica esperienza che abbiamo condiviso quest'anno.

Un grosso grazie va alla mia famiglia: a mamma e papà, che se potevano mi avrebbero dato anche la luna! In questi ultimi anni mi sono accorta di quanto sono fortunata ad averli come genitori. Non posso non ringraziare la mia fantastica nonna, che non ha fatto che viziarmi da quando sono nata. Un bacione grandissimo va alla mia "sorella mancata" Ilaria, a zia Manola, a Manolo e a quell'ape di mia nipote Aurora. Naturalmente non posso dimenticarmi di Trilly che ha passato tutta l'estate con me a scrivere la tesi.

E ora passiamo alle persone con cui non ho legami di parentela, ma mi conoscono da una vita o quasi.
Chiara, che dire, la mia migliore amica da sempre, abbiamo condiviso così tante cose, peccato che non abbiamo studiato insieme a Pisa, penso che avremmo stabilito un record!

Marta, che con la sua dolcezza e sensibilità è riuscita sempre ad essere presente e pronta ad ascoltarmi nonostante la lontananza.
Vale, amica sincera e preziosa consigliera, con cui ho condiviso una colazione unica: non dimenticherò mai quella mattina, cornetto e cappuccino!
E ora tocca alle pisane d'adozione: Glo e Patu, ricordo ancora la prima volta che ci siamo viste nel corridoio della facoltà e da lì è iniziata la nostra splendida amicizia, da semplici compagne di studio a immancabili compagne di vita. Ma quante ne abbiamo combinate insieme? la doccia sotto la torre, i viaggi a Lucca mentre ci imponevamo di non ridere, Roma, le feste più strane di Pisa, le nottate sulla poltroncina al Praticelli con visite a sorpresa, i viaggi della speranza, le mille ore all'ikea, le classifiche e infine quest'anno di convivenza assolutamente fantastico nella "G&S's home and Patu's second home"!

Ringrazio tutte le persone con cui ho condiviso questi ultimi anni. Max, maestro di spritz e amico insostituibile; i ragazzi del laboratorio: Ale, Carlo, Marina, Marco, Lotta e tutti gli altri; Catia, da new entry della specialistica a sincera amica; Sabrina, se non fermava lei l'autobus la mattina ai vecchi tempi rischiavo di essere bocciata per mille ritardi e di non arrivare fin qui! Sandra, immancabile nel gruppo delle mitiche quattro ternane; e poi Marzia, Ile, Sara; e impossibile da dimenticare il Lò, Brodino e Vic, i nostri bodyguard pisani.

Infine, ringrazio con tutto il mio cuore la persona a cui ho dedicato questa tesi, il mio fornitore ufficiale di felicità, non avrei potuto desiderare miglior "compagno" di tesi.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Il problema della *Cache Coherence* nei sistemi multiprocessor è un argomento divenuto ancora più sensibile a seguito dello sviluppo dei sistemi multicore. Questo problema è stato largamente affrontato in letteratura [6, 21, 5], ponendo l'attenzione sulle prestazioni delle tecniche di cache coherence utilizzate [15, 19, 1, 8, 7] e spesso facendo uso di simulatori per confrontare le varie soluzioni [3, 13].

L'utilizzo delle memorie cache è fondamentale nelle architetture multiprocessor perché contribuiscono a diminuire il tempo di servizio per istruzione ma soprattutto perché permettono di ridurre i conflitti dei processori sulla memoria condivisa, cioè ridurre la latenza di accesso in memoria sotto carico. Nelle architetture di tipo *all-cached*, dove tutte le informazioni contenute in memoria principale sono soggette a caching, nasce il problema della cache coherence, cioè della consistenza delle informazioni presenti nelle cache dei vari nodi: se più processori trasferiscono nelle proprie cache uno stesso blocco, occorre garantire che le copie rimangano consistenti tra loro e nei confronti della copia in memoria principale.

Al fine di risolvere questo problema, sono stati studiati numerosi *protocolli di cache coherence* [20, 22, 18, 10, 18] che permettono di coordinare le azioni di un generico nodo al quale il livello hardware-firmware dei sistemi multiprocessor mette a disposizione dei meccanismi per l'invalidazione o l'aggiornamento dei blocchi di cache. Questi protocolli si basano sull'idea che, in un sistema multiprocessor, ciascun blocco di memoria ha uno stato in ciascuna cache e

questi stati cambiano in accordo ad un diagramma di transizione degli stati definito dal protocollo stesso. Si può quindi pensare allo stato di un blocco come un insieme di n stati, dove n rappresenta il numero di cache. Lo stato della cache è quindi gestito da un insieme di n macchine a stati finiti distribuite e la macchina a stati che gestisce i cambiamenti di stati è la stessa per tutti i blocchi e tutte le cache, quello che cambia è lo stato corrente del blocco nelle varie cache.

Quando il supporto della cache coherence viene delegato interamente al livello firmware si parla di *tecniche di cache coherence automatica*; queste tecniche permettono così di sviluppare programmi in maniera indipendente dalle operazioni necessarie per la gestione della coerenza e per questo vengono adottate nella maggior parte dei sistemi commerciali. In particolare, sono state studiate due possibili soluzioni: la soluzione *Snoopy-based*, con la quale si utilizza un bus come punto di centralizzazione a livello firmware e la soluzione *Directory-based*, adottata in sistemi a più alto grado di parallelismo, chiamata così perché basata sul concetto di *directory*.

Una possibile alternativa alle soluzioni commerciali, è il così detto *approccio Algorithm-Dependent*, presentato in [23]. Molti tra i sistemi multiprocessor e CPU sviluppati recentemente mettono a disposizione annotazioni o particolari istruzioni per una gestione esplicita della gerarchia di memoria, fino ai casi in cui viene permessa la disabilitazione dei meccanismi di cache coherence automatica o in altri casi in cui questi meccanismi sono completamente assenti. L'approccio algorithm-dependent permette di programmare esplicite strategie di cache coherence senza fare affidamento su specifiche soluzioni automatiche, basandosi su una progettazione del supporto per le operazioni di mutua esclusione che tenga conto del problema della cache coherence. In questo modo, nell'ambito della programmazione parallela strutturata non si ha più la necessità di utilizzare meccanismi per la cache coherence; infatti, l'unica situazione in cui si rende necessario implementare una soluzione alla cache coherence è al momento della progettazione del supporto a tempo di esecuzione.

Abbiamo quindi due alternative per la risoluzione del problema della cache coherence, che si differenziano in base al livello in cui viene affrontato il problema stesso: da una parte, le tecniche automatiche fanno affidamento sui

meccanismi forniti dal livello hardware-firmware per implementare i protocolli di cache coherence; dall'altra parte invece, l'approccio algorithm-dependent tenta di minimizzare il numero di trasferimenti di blocchi di cache e di comunicazioni interprocessor integrando la risoluzione del problema nel supporto ai meccanismi di sincronizzazione per programmi paralleli, in particolare nelle operazioni di mutua esclusione.
A questo punto è interessante effettuare un confronto sulle prestazioni di un sistema in cui si utilizzano le tecniche di cache coherence automatica rispetto ad un sistema che fornisce un supporto implementato secondo l'approccio algorithm-dependent. Per far questo, non ci è sembrato adatto effettuare un'analisi dei risultati ottenuti dall'utilizzo di uno tra i simulatori esistenti; infatti, abbiamo notato che questi non ci avrebbero fornito la possibilità di parametrizzare sufficientemente il problema. Per lo stesso motivo, non sarebbe stata utile al nostro scopo neanche un'implementazione di un supporto basato sull'approccio algorithm-dependent su una macchina reale esistente. Avevamo bisogno di un confronto che fosse più generale e parametrico possibile, non uno studio limitato a quel particolare tipo di architettura, alla particolare struttura di interconnessione o al particolare protocollo di cache coherence utilizzati. Per tutti questi motivi, la soluzione che ci è sembrata migliore è stata quella di effettuare un'analisi analitica dei due approcci.
Questo lavoro di tesi si è quindi posto come obiettivo la realizzazione di un *Modello dei Costi* che, a partire dalla valutazione delle prestazioni di un sistema multiprocessor e in particolare dalla valutazione dei tempi di accesso in memoria condivisa, permetta di valutare l'impatto che le tecniche di cache coherence studiate hanno sulle prestazioni dei programmi paralleli.
In particolare, porremo l'attenzione sui vantaggi delle strategie che affrontano il problema della cache coherence durante la progettazione del supporto di un formalismo di concorrenza a livello di processi rispetto a quelle che fanno uso di tecniche di cache coherence automatica.

1.1 Struttura della Tesi

Vediamo ora in dettaglio i capitoli in cui è suddivisa la tesi e quali sono gli argomenti affrontati in ciascuno di essi.

- **Capitolo 2 Cache Coherence** : dopo una breve introduzione alle architetture a memoria condivisa, presenteremo il problema della cache coherence e i meccanismi utilizzati per risolverlo. In particolare, verranno descritti i principali protocolli di cache coherence che fanno uso di questi meccanismi.

- **Capitolo 3 Tecniche di Cache Coherence Automatica**: in questo capitolo andiamo ad analizzare in dettaglio le due principali categorie di tecniche di cache coherence automatica: Snoopy-based e Directory-based. In particolare, vedremo come queste due tecniche implementano i protocolli presentati nel capitolo precedente.

- **Capitolo 4 Approccio Algorithm-Dependent**: dopo una breve introduzione ai meccanismi di sincronizzazione nelle architetture multiprocessor, andiamo ad analizzare come integrare la cache coherence nel supporto per la mutua esclusione, con il fine di minimizzare il numero di operazioni necessarie per mantenere la consistenza dei blocchi di cache.

- **Capitolo 5 Modello dei Costi e Analisi delle Prestazioni**: in questo capitolo vogliamo fornire un modello dei costi che permetta di valutare in modo formale le prestazioni delle soluzioni presentate nei capitoli precedenti. A partire dalla teoria delle code vedremo come valutare l'impatto che le tecniche di cache coherence hanno sulle prestazioni dei programmi paralleli e come influenzano la valutazione dei tempi di accesso in memoria condivisa.

- **Capitolo 6 Conclusioni**: la tesi si conclude ripercorrendo le soluzioni studiate per risolvere il problema della cache coherence, facendo una riflessione su vantaggi e svantaggi che il modello dei costi sviluppato ha messo in evidenza per ognuna di esse e su quale può essere una naturale continuazione della tesi.

Capitolo 2

Cache Coherence

Nelle architetture multiprocessor l'utilizzo delle memorie cache è fondamentale per due motivi principali: come nei sistemi uniprocessor, contribuiscono a diminuire il tempo di servizio per istruzione ma soprattutto permettono di ridurre i conflitti dei processori sulla memoria condivisa, cioè ridurre la latenza di accesso in memoria sotto carico.
Nelle architetture di tipo *all-cached*, dove tutte le informazioni contenute in memoria principale sono soggette a caching, nasce il problema della consistenza delle informazioni presenti nelle cache dei vari nodi, o problema della *Cache Coherence*.
In questo capitolo, dopo una breve introduzione alle architetture a memoria condivisa nella sezione 2.1, presenteremo nella sezione 2.2 il problema della cache coherence e i meccanismi utilizzati per risolverlo. In particolare, nella sezione 2.3 vengono descritti i principali protocolli di cache coherence che utilizzano questi meccanismi.

2.1 Architetture a Memoria Condivisa

I sistemi multiprocessor sono architetture parallele general-purpose (*Multiple Instruction Stream - Multiple Data Stream*) caratterizzate come segue:

- sono costituiti da n processori;
- i processori condividono la memoria principale, ed eventualmente alcuni livelli inferiori della gerarchia di memoria; ciò significa che tutti i

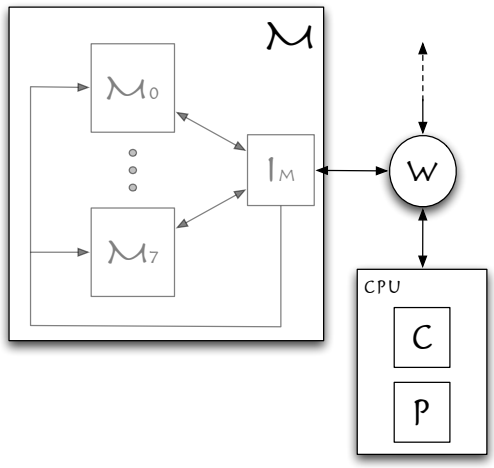


Figura 2.1: Architettura di un nodo di elaborazione

processori sono in grado di indirizzare tutte le locazioni della memoria condivisa;

- i processori sono collegati alla memoria condivisa e/o tra loro mediante opportune strutture di interconnessione, permettendo ad ogni processore di raggiungere qualunque locazione della memoria condivisa, o di scambiare valori con qualunque altro processore.

In realtà un generico processore è un vero e proprio nodo di elaborazione costituito da CPU con uno o più livelli di cache, eventuale memoria locale non condivisa con gli altri nodi, e unità di I/O locali.

La figura 2.1 mostra la generica struttura di un nodo di elaborazione: l'unità di elaborazione W funge da unità di interfaccia del nodo verso il resto del sistema essendo collegata alla struttura di interconnessione. In particolare, l'unità W si occupa di gestire le operazioni necessarie per il trattamento di un fault di cache. Inoltre, la memoria locale M è interallacciata (in figura 8 moduli di memoria) ed è a sua volta interfacciata da un'apposita unità I_m, allo scopo di rendere totalmente indipendente la realizzazione della memoria stessa.

Le figure 2.2 e 2.3 mostrano la possibile classificazione delle architetture multiprocessor in base all'organizzazione della memoria condivisa:

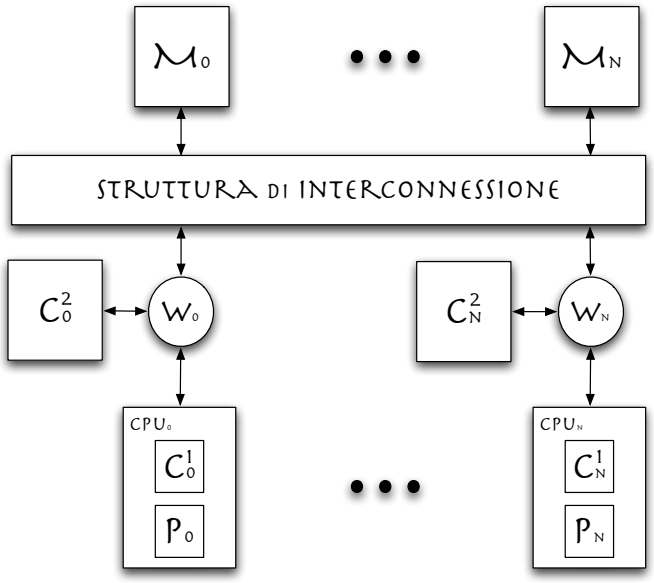


Figura 2.2: Architettura SMP

- *UMA (Uniform Memory Access) o SMP (Symmetric MultiProcessor)*, nel quale i moduli della memoria condivisa sono "equidistanti" dai nodi di elaborazione, questo vuol dire che la latenza a vuoto per un accesso da parte di un processore P_i ad un modulo di memoria M_j è costante e indipendente dai valori i e j;

- *NUMA (Non Uniform Memory Access)*, nel quale i moduli della memoria condivisa non sono tutti "equidistanti" dai nodi di elaborazione; tipicamente la memoria condivisa è costituita dall'insieme delle memorie locali e la latenza a vuoto per l'accesso locale di P_i a M_i è certamente minore di quella per l'accesso remoto di P_i a M_j, in quanto l'accesso remoto utilizza la rete di interconnessione, avente in generale una latenza dipendente dal numero dei nodi.

Una variante commerciale dell'architettura NUMA è la così detta architettura *COMA (Cache Only Memory Access)*, nella quale tutta la memoria condivisa è cache, e le informazioni transitano dinamicamente da una cache all'altra senza che esista concettualmente un supporto superiore nella gerarchia di memoria (in realtà un supporto minimo è necessario per la gestione del

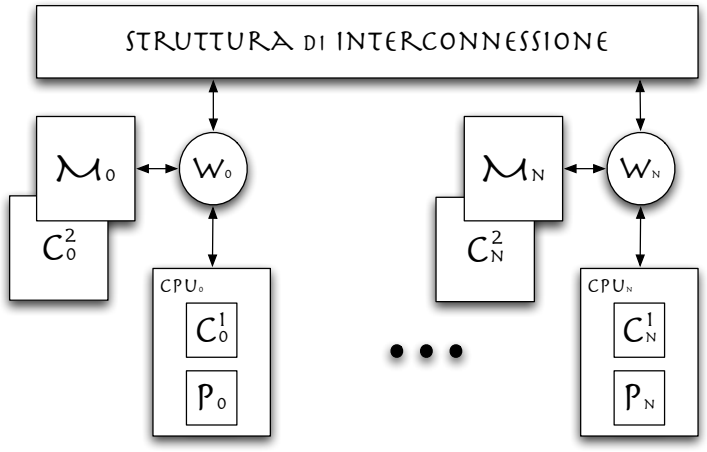


Figura 2.3: Architettura NUMA

sistema). Questo tipo di architettura nasce come conseguenza dell'architettura all-cached, la quale rende ancora più "sfumata" la distinzione tra architetture NUMA e SMP. Questo perché se le cache dei nodi svolgono efficientemente il loro compito di ridurre sostanzialmente il fattore di utilizzazione della memoria condivisa, allora l'equidistanza o meno dei nodi dai moduli di memoria non è così essenziale, in quanto in termini di prestazioni la differenza si ha solo relativamente al tempo di trasferimento dei blocchi di memoria.

Strutture di interconnessione

Concettualmente, in un'architettura multiprocessor sono necessarie due strutture di interconnessione:

- *struttura P-M*: tra nodi di elaborazione e memoria condivisa per accedere in memoria remota e per poter trasferire blocchi di cache;
- *struttura P-P*: tra nodi di elaborazione, per lo scambio di comunicazioni interprocessor.

In realtà, a causa della complessità di queste strutture e per un basso sfruttamento della struttura P-P, le due vengono implementate mediante una sola struttura. La definizione di una struttura di interconnessione si basa sul compromesso tra due esigenze contrastanti: avere una connettività ad alta banda

tra nodi e contenere l'effetto *pin count* delle unità di elaborazione ed il numero stesso di collegamenti. Le due strutture che massimizzano queste due caratteristiche sono:

- il *bus*, che minimizza il pin count e il numero di collegamenti, ma con una latenza di trasmissione lineare nel numero n di nodi;
- il *crossbar*, o interconnessione completa, che massimizza la banda di comunicazione e minimizza la latenza rendendola costante; d'altra parte il costo di n^2 collegamenti dedicati, rendendo elevato il pin count, si ripercuote negativamente anche su banda e latenza.

Le *strutture di grado limitato* come cubi, alberi, fat-tree e "butterfly" cercano il compromesso suddetto ottenendo come risultato latenze di ordine logaritmico o $\sqrt{n}$ con una banda confrontabile con quella dei crossbar.
Un'altra importante questione che riguarda le strutture di interconnessione per macchine parallele è la strategia di controllo del flusso adottata. Una tecnica alternativa al controllo del flusso "a pacchetti" è la cosiddetta strategia di controllo del flusso *wormhole*, o *virtual cut through*, in cui ogni pacchetto viene decomposto in parti più piccole, dette *flit* (ad esempio, una o poche parole); tutti i flit di uno stesso pacchetto effettuano lo stesso percorso ma, nell'ambito di tale percorso, sono considerati come elementi di uno stream in una forma di parallelismo pipeline. In altri termini, la trasmissione di ogni pacchetto è parallelizzata in pipeline. Ne consegue il tipico guadagno nel tempo di completamento di una struttura pipeline rispetto alla struttura sequenziale equivalente.

2.2 Il Problema della Cache Coherence

Il problema della cache coherence nasce dalla possibilità che diverse cache di un sistema possono mantenere una copia dello stesso blocco di memoria. In particolare studiamo come si verifica questo problema in un'architettura con un solo livello di cache.
Se più processori trasferiscono nelle proprie cache uno stesso blocco, occorre

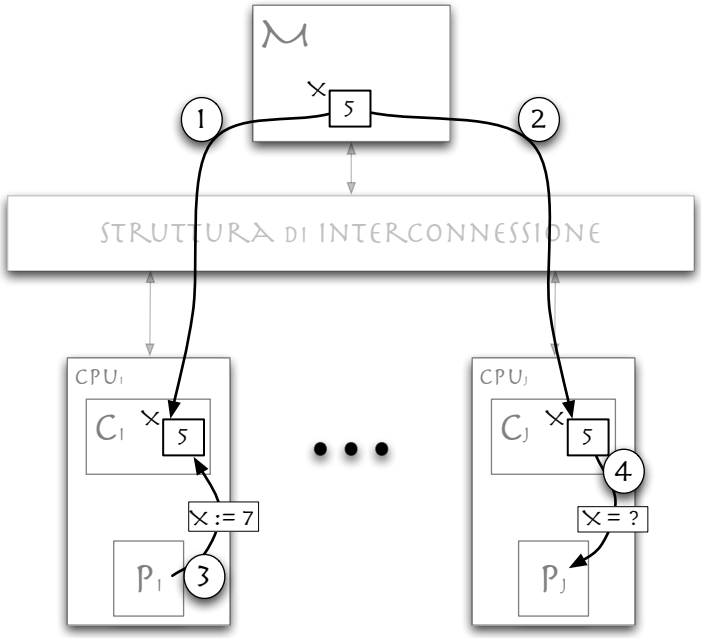


Figura 2.4: Problema della cache coherence

garantire che le copie rimangano consistenti tra loro e nei confronti della copia in memoria principale.

Come schematizzato in figura 2.4, il sistema preso in considerazione ha una memoria condivisa M, è costituito da processori P_i, ognuno dotato di cache C_i. Se più processori, come P_i e P_j, trasferiscono una stessa variabile X da M nella propria cache, non si verificano problemi di coerenza se X è usata in sola lettura. Se invece P_i modifica X, allora la copia in C_j non è consistente. Se il meccanismo di caching utilizza la tecnica di scrittura in memoria write-back, allora anche eventuali altri processore che tentano di portare X nella propria cache troveranno X inconsistente in M.

2.2.1 Invalidazione vs Aggiornamento

Nella maggior parte dei sistemi, l'architettura firmware mette a disposizione delle tecniche, dette di cache coherence *automatica*, che fanno uso di meccanismi primitivi per garantire la cache coherence. I meccanismi utilizzati da queste tecniche sono:

- *invalidazione*, con il quale si ammette che la sola copia valida di un blocco sia una di quelle, di regola l'ultima, che è stata modificata, invalidando tutte le altre copie; se la tecnica di scrittura in memoria adottata è write-through anche la copia in M è valida;

- *aggiornamento*, con il quale si fa in modo che tutti i processori possano essere in grado di usare l'informazione modificata, la quale viene comunicata per diffusione.

Se consideriamo l'esempio precedente, se P_i decide di operare su X, il primo accesso provoca il trasferimento del blocco contenente X nella propria cache. Se anche P_j ha una copia dello stesso blocco in cache, allora il primo tra i due, supponiamo P_i, che esegue una scrittura sul blocco di X:

- con il meccanismo di invalidazione, rende non valida la copia del blocco in C_j;

- con il meccanismo di aggiornamento, aggiorna la copia in C_j inviando i dati aggiornati per diffusione.

Il meccanismo di invalidazione, anche se appare più macchinoso, ha un basso overhead per le comunicazioni che avvengono tra i processori per coordinare le rispettive azioni; è infatti sufficiente inviare la segnalazione di invalidazione di un determinato blocco.
Con il meccanismo di aggiornamento, che a prima vista risulta più lineare, si va incontro ad un overhead più pesante per quanto riguarda la comunicazione di aggiornamento; infatti tutte le modifiche relative ad un blocco condiviso devono essere comunicate agli altri processori.

2.3 Protocolli di Cache Coherence

Dopo l'introduzione al problema della cache coherence e ai meccanismi che un'architettura a livello firmware può mettere a disposizione al fine di risolverlo, analizziamo ora come è possibile sfruttare questi meccanismi coordinando le azioni di un generico nodo di elaborazione attraverso i protocolli di cache

coherence.

L'utilizzo di un protocollo di cache coherence comporta che ogni blocco di cache abbia associato uno stato. Il *diagramma di transizione degli stati dei blocchi di cache* è una macchina a stati finiti che definisce come cambia lo stato dei blocchi. Mentre solo i blocchi che sono in cache hanno effettivamente l'informazione di stato associata, logicamente tutti gli altri blocchi che non sono presenti in cache possono essere visti come in uno stato speciale "non presente", o nello stato "invalido".

In un sistema uniprocessor, con cache che utilizza il metodo di scrittura in memoria write-through senza l'allocazione dei blocchi per la scrittura, solo due stati sono sufficienti per la gestione della cache coherence: valido e invalido. Inizialmente tutti i blocchi sono invalidi; quando, a seguito di una richiesta di lettura da parte del processore, viene generato un fault e il blocco viene trasferito nella cache, allora il blocco può essere marcato come valido. Le richieste di scrittura invece non comportano un cambiamento di stato del blocco, infatti se il blocco è presente in cache, questo viene aggiornato e lo stato rimane valido, se non è presente lo stato rimane sempre invalido, in quanto non viene trasferito il blocco in cache per la scrittura. Se un blocco viene rimpiazzato, allora lo stato viene modificato in invalido finché il nuovo blocco non viene trasferito in cache e marcato valido.

Con il meccanismo di scrittura in memoria write-back è necessario utilizzare uno stato addizionale per indicare il fatto che il blocco è modificato, detto anche "*dirty*".

In un sistema multiprocessor, un blocco ha uno stato in ciascuna cache e questi stati cambiano in accordo al diagramma di transizione degli stati. Si può quindi pensare allo stato di un blocco come un insieme di n stati, dove n rappresenta il numero di cache. Lo stato della cache è quindi gestito da un insieme di n macchine a stati finiti distribuite, implementate dalle unità W di ogni nodo che fungono anche da *controllori della coerenza*. La macchina a stati che gestisce i cambiamenti di stati è la stessa per tutti i blocchi e tutte le cache, quello che cambia è lo stato corrente del blocco nelle varie cache.

Il protocollo **MSI**, descritto nella sezione 2.3.1, è il protocollo di riferimento tra quelli basati su invalidazione per cache che utilizzano la tecnica di scrittura

in memoria write-back. La maggior parte dei sistemi multiprocessor moderni peró, utilizza varianti del protocollo MSI al fine di ridurre la quantità di comunicazioni generate per il mantenimento della coerenza tra i blocchi di cache.
La sezione 2.3.2 presenta il protocollo **MESI** che aggiunge all'insieme di stati del protocollo MSI lo stato *Exclusive* al fine di ridurre il traffico causato da scritture di blocchi presenti in una sola cache.
Il protocollo **MOSI**, introdotto nella sezione 2.3.3, utilizza lo stato *Owned* per ridurre il traffico causato dalla riscrittura in memoria dei blocchi mantenuti in un'unica cache.
Viene infine presentato nella sezione 2.3.4 un protocollo basato su aggiornamento, il protocollo **Dragon**.

2.3.1 Il protocollo MSI

Il protocollo MSI è il protocollo base di cache coherence utilizzato nei sistemi multiprocessor. Si tratta di un protocollo basato su invalidazione per cache che utilizzano la tecnica di rimpiazzamento dei blocchi in memoria write-back. Come in molti altri protocolli le lettere che compongono il nome del protocollo indicano l'insieme dei possibili stati che un blocco di cache può assumere. In questo caso ciascun blocco all'interno di una cache può trovarsi in uno tra i tre seguenti stati:

- ***M**odified:* spesso chiamato anche *dirty*, indica che solo questa cache possiede una copia valida del blocco, la copia in memoria principale non è aggiornata;
- ***S**hared:* il blocco è presente in uno stato non modificato nella cache e la copia in memoria principale è aggiornata; inoltre, una o più cache possono avere una copia aggiornata (in stato shared) del blocco stesso;
- ***I**nvalid:* il blocco non è valido;

Prima che un blocco in stato shared o invalid possa essere modificato, e quindi si possa cambiare il suo stato in modified, tutte le altre potenziali copie devono essere invalidate.

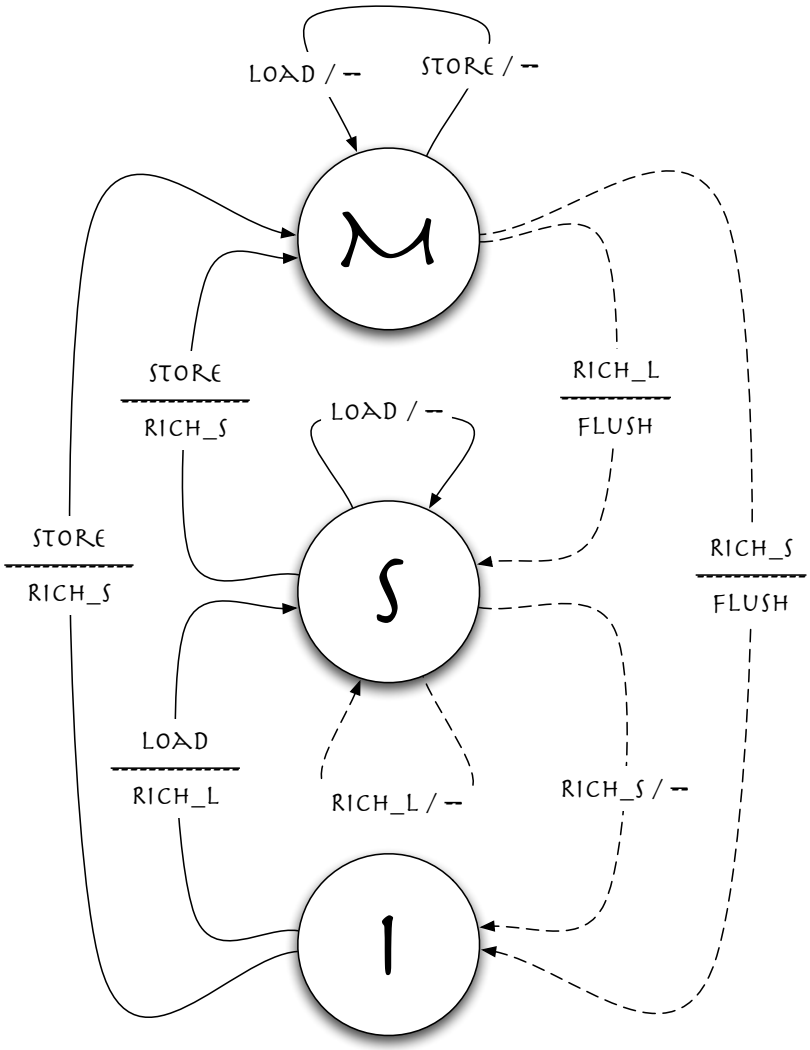


Figura 2.5: Diagramma degli stati del protocollo MSI

Il diagramma degli stati rappresentato in figura 2.5, mostra le possibili transizioni di stato che può subire un blocco di cache. Le transizioni del diagramma hanno un'etichetta della forma R/A, dove R indica una richiesta mentre A indica l'azione che il controllore della coerenza deve intraprendere a seguito della richiesta effettuata. Come vedremo, anche negli altri protocolli, ciascuna di queste transizioni è formata da una o più operazioni e, ai fini della correttezza del protocollo, è necessario che queste vengano eseguite in maniera *atomica*.
Una transizione può essere di due tipi, in base alla provenienza della richiesta:

- quelle rappresentate da una *linea continua* sono quelle generate dalle richieste di lettura (`LOAD`) e scrittura (`STORE`) da parte del processore;
- quelle rappresentate da una *linea tratteggiata* sono quelle generate dalle richieste provenienti dalle altre cache (`RICH_L` e `RICH_S`).

Le richieste provenienti dal processore possono riferirsi ad un blocco già presente nella cache o meno; in questo secondo caso un blocco attualmente presente in cache deve essere rimpiazzato dal blocco richiesto, e se il blocco presente è in stato modified, il suo contenuto deve essere riscritto in memoria (`WRITE_BACK`). Analizziamo in dettaglio tutte le possibili transizioni di stato.

Transizioni con richieste provenienti dal processore La richiesta di lettura da parte del processore di un blocco il cui stato è invalid o di un blocco non presente in cache causa l'invio di una richiesta di lettura a seguito del fault. Il blocco ottenuto viene memorizzato in cache con lo stato shared. Se il blocco di cui viene richiesta la lettura si trova in stato shared o modified non comporta azioni da intraprendere e nessuna transizione di stato.
La richiesta di scrittura da parte del processore di un blocco il cui stato è invalid genera un fault di scrittura, questo viene risolto inviando una richiesta di scrittura del blocco che, una volta ricevuto viene memorizzato in cache con lo stato modified. Se il blocco che il processore intende modificare si trova in stato shared viene, anche in questo caso, generato un fault di scrittura; in questo caso i dati ricevuti a seguito del fault vengono ignorati in quanto la

copia del blocco in cache è aggiornata, infatti una frequente ottimizzazione è quella che prevede l'introduzione di una nuova transizione in cui viene inviata una richiesta di *aggiornamento* che non comporta il trasferimento del blocco. Le successive richieste di scrittura del blocco in stato modified non comportano nessuna azione e nessuna transizione di stato.
Il rimpiazzamento di un blocco corrisponde logicamente al cambiamento dello stato in invalid; se il blocco si trovava in stato modified allora è necessario aggiornare la memoria (`WRITE_BACK`), mentre non è necessario intraprendere alcuna azione nel caso in cui lo stato del blocco fosse stato shared.

Transizioni con richieste provenienti dalle altre cache La richiesta di lettura da parte di un'altra cache di un blocco presente in cache in stato modified comporta l'esecuzione di un'operazione di `FLUSH` dei dati, cioè l'invio dei dati aggiornati da parte della cache alla cache che ne ha fatto richiesta e l'aggiornamento del blocco in memoria principale; a questo punto viene cambiato lo stato del blocco in shared. La richiesta di lettura da parte di un'altra cache di un blocco presente in cache in stato shared non comporta azioni da intraprendere e nessuna transizione di stato.
La richiesta di scrittura da parte di un'altra cache di un blocco presente in cache in stato modified causa l'esecuzione dell'operazione di `FLUSH` dei dati, come nel caso della lettura; in questo caso però il blocco deve essere invalidato.
Nel caso in cui la richiesta di scrittura sia riferita ad un blocco presente in cache in stato shared nessuna azione deve essere intrapresa; l'unico cambiamento riguarda lo stato del blocco che deve essere invalidato.

2.3.2 Il protocollo MESI

Il protocollo MESI è il protocollo di cache coherence più diffuso ed è stato presentato per la prima volta in [20] dai ricercatori dell'Università dell'Illinois; per questo motivo viene spesso riferito come il protocollo Illinois.
Analizzando il protocollo MSI, il primo fattore di inefficienza si può notare quando un processo necessita di leggere e modificare un dato: le transizioni che vengono causate sono sempre due, anche quando non ci sono altri nodi

che condividono il blocco di cache. Infatti, viene inizialmente generata una transizione che permette al nodo di ottenere il blocco a seguito della richiesta di lettura, mantenendolo in cache in stato shared. La seconda transizione è quella causata dalla richiesta di scrittura del blocco che modifica lo stato del blocco da shared a modified.
Aggiungendo lo stato ***E**xclusive* si vuole indicare che:

- il blocco in cache è l'unica (esclusiva) copia presente in tutte le cache
- e che non è stato modificato.

Questo nuovo stato pone il blocco ad un livello intermedio tra lo stato modified e lo stato shared: è *esclusivo*, quindi, diversamente dallo stato shared, una scrittura del blocco può essere eseguita e lo stato può essere cambiato in modified senza ulteriori azioni; non implica la *proprietà* (la copia in memoria è aggiornata) quindi, diversamente dallo stato modified, la cache non deve necessariamente inviare i dati a seguito della richiesta del blocco da parte di un'altra cache.

Transizioni con richieste provenienti dal processore La figura 2.6 mostra l'insieme delle transizioni causate dalle richieste di lettura e scrittura provenienti dal processore.

La prima volta che il processore richiede la lettura di un blocco, è necessario verificare se esiste o meno una copia valida in un'altra cache (indicato nella figura con (S) nell'azione di richiesta di lettura). Nel primo caso il blocco viene copiato nella cache con lo stato shared come avviene nel protocollo MSI; invece, se non esistono altre cache che hanno una copia valida del blocco, lo stato assegnato è exclusive. Le successive richieste di lettura del blocco da parte del processore mantengono invariato lo stato del blocco stesso.
La richiesta di scrittura del blocco che si trova in cache in stato exclusive comporta direttamente il cambiamento di stato in modified, senza dover causare ulteriori azioni e richieste.

Transizioni con richieste provenienti dalle altre cache La figura 2.7 mostra l'insieme delle transizioni causate dalle richieste provenienti dalle altre

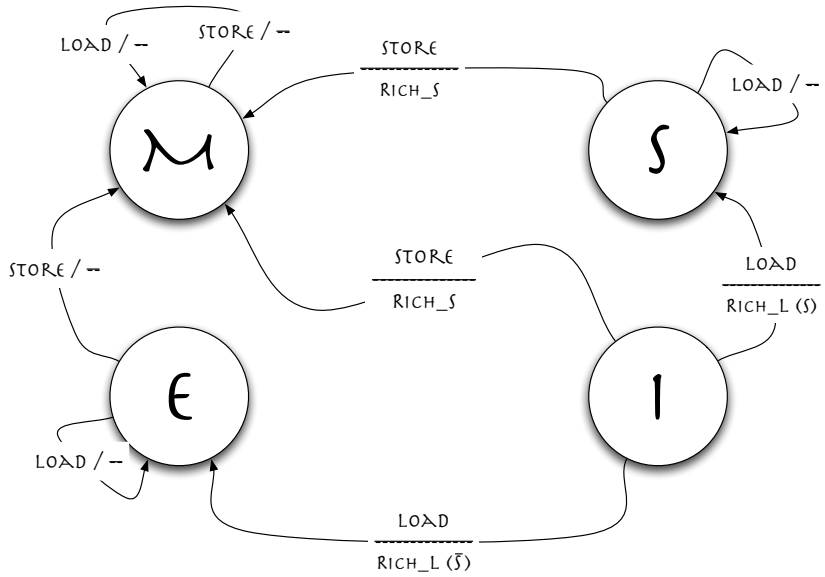


Figura 2.6: Diagramma di transizione degli stati del protocollo MESI: transizioni con richieste provenienti dal processore

cache.
La richiesta di lettura da parte di un'altra cache di un blocco in stato *esclusive* comporta il passaggio di stato del blocco a shared.
La richiesta di scrittura da parte di un'altra cache di un blocco in stato exclusive comporta, invece, l'invalidazione del blocco nella cache.
Con l'introduzione dello stato exclusive, vedremo come nei sistemi multiprocessor con cache coherence automatica di tipo *Snoopy-based*, presentata nella sezione 3.1, possa essere adottata la tecnica del *cache-to-cache sharing* per la condivisione dei blocchi tra cache; in questo modo saranno giustificate le operazioni di `FLUSH` mostrate tra parentesi nella figura 2.7 nelle transizioni in uscita dagli stati exclusive e shared.

Il protocollo Write Once

Nel protocollo MESI una scrittura può essere effettuata solo se il blocco di cache si trova in stato modified o exclusive. Se il blocco si trova in stato shared tutte le altre copie devono prima essere invalidate. L'operazione con cui viene fatta la richiesta di scrittura è conosciuta in letteratura come *Read For*

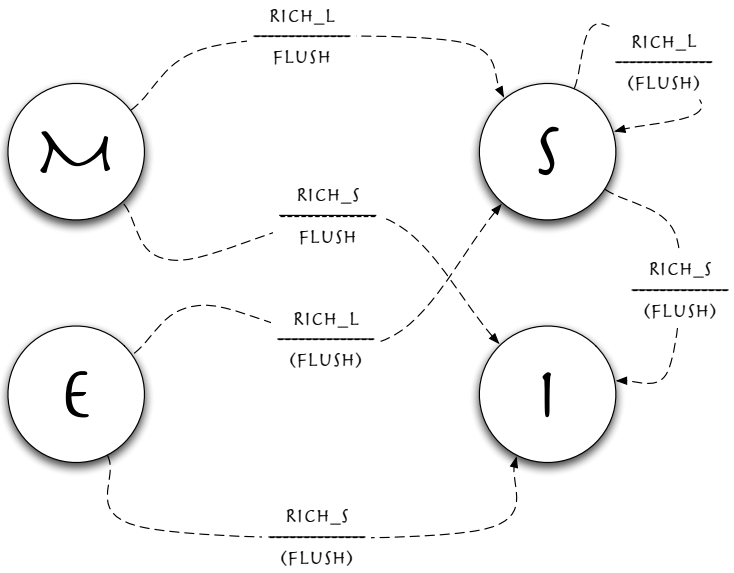


Figura 2.7: Diagramma di transizione degli stati del protocollo MESI: transizioni con richieste provenienti dalle altre cache

Ownership (RFO). Si richiede infatti di poter leggere il blocco con l'intento di poter poi scrivere nel blocco stesso e, per questo, è necessario invalidare tutte le altre copie. Nella letteratura che riguarda i protocolli di cache coherence, il protocollo textbfWrite Once [10] è stato il primo protocollo in cui la richiesta di invalidazione avviene a causa di una richiesta di scrittura (*write-invalidate*). In questo questo protocollo, ogni blocco di cache può trovarsi in uno dei seguenti stati:

- Invalid;
- Valid;
- Reserved;
- Dirty.

Questi stati hanno esattamente lo stesso significato dei quattro stati del protocollo MESI (sono semplicemente elencati in ordine inverso); si tratta però di una forma semplificata in cui non si utilizza l'operazione RFO, ma le invalidazioni avvengono scrivendo in memoria principale. In particolare, la prima

volta che il processore intende scrivere in un blocco in stato valid (shared) viene utilizzata la tecnica di write-through, che implicitamente invalida tutte le altre copie del blocco. A questo punto, il blocco si trova nella cache in stato reserved (exclusive), e le successive scritture possono essere effettuate in cache cambiando semplicemente lo stato del blocco in dirty (modified).

2.3.3 Il protocollo MOSI

Il protocollo MOSI è un'altra variante del protocollo base MSI in cui viene introdotto un nuovo stato per un blocco di cache, lo stato ***O**wned*. Un blocco di cache che si trova in stato owned mantiene la più recente e corretta copia dei dati. Questo stato è:

- simile allo stato shared, in quanto le altre cache possono mantenere la copia più recente e corretta dei dati;
- simile allo stato modified, in quanto la copia in memoria principale non è aggiornata.

Solo una cache può avere il blocco in stato owned, mentre le altre cache mantengono in blocco in stato shared. Il blocco può passare in stato modified dopo aver invalidato tutte le altre copie, oppure può passare in stato shared aggiornando la copia in memoria principale.
A seguito di richieste di lettura e/o scrittura da parte di un'altra cache di un blocco in stato owned, la cache deve effettuare un'operazione di `FLUSH` dei dati, come nel caso dello stato modified; in questo caso però la richiesta di scrittura non comporta necessariamente l'invalidazione del blocco, che può essere mantenuto in cache in stato shared.

Il protocollo MOESI

Con l'introduzione dello stato owned viene naturale pensare ad un'estensione dei due protocolli MESI e MOSI; in [22] viene infatti presentato il protocollo MOESI. La figura 2.8 permette di capire come classificare un blocco di cache, secondo il protocollo MOESI, in base alle seguenti tre caratteristiche:

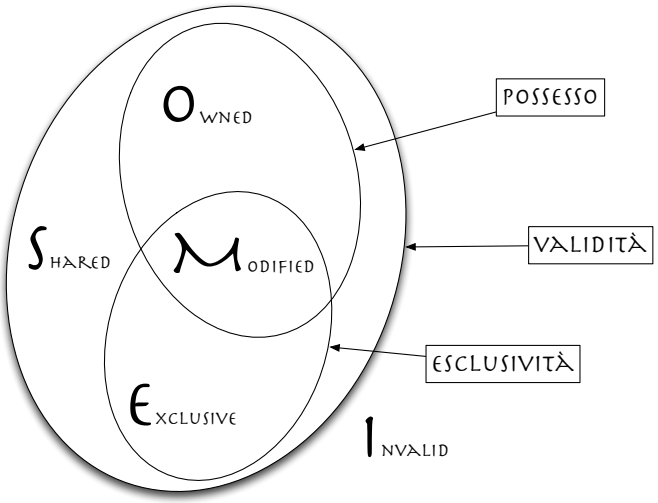


Figura 2.8: Classificazione dello stato di un blocco di cache secondo il protocollo MOESI

- validità;
- esclusività;
- possesso.

In questo modo si cerca di ridurre sia il traffico causato da scritture di blocchi presenti in una sola cache, sia quello causato dalla riscrittura in memoria dei blocchi mantenuti in un'unica cache.

2.3.4 Il protocollo Dragon

Fino ad ora abbiamo esaminato tutti protocolli basati su invalidazione; il protocollo Dragon [18], sviluppato dai ricercatori della Xerox PARC per il multiprocessor Dragon, è un protocollo basato su aggiornamento. Gli stati utilizzati dalla cache sono i seguenti quattro:

- ***E**xclusive-clean:* (o *exclusive*) ha lo stesso significato dello stato exclusive visto nei protocolli precedenti; solo una cache (questa cache) ha una copia aggiornata del blocco che è la stessa presente in memoria principale;

- ***S**hared-**c**lean:* indica che potenzialmente altre cache hanno una copia del blocco e non è assicurato che la memoria principale sia aggiornata;

- ***S**hared-**m**odified:* indica che due o più cache hanno una copia del blocco, la memoria non è aggiornata e che questa cache è responsabile dell'aggiornamento della memoria principale al momento del rimpiazzamento del blocco; solo una cache può avere il blocco in questo stato, se altre cache hanno una copia del blocco lo stato con cui lo mantengono è quello shared-clean;

- ***M**odified:* ha lo stesso significato visto nei protocolli precedenti, inoltre la cache è responsabile dell'aggiornamento delle altre cache e della memoria principale.

Trattandosi di un protocollo basato su aggiornamento non è presente lo stato che indica l'invalidità di un blocco, infatti il protocollo mantiene i blocchi nelle cache sempre aggiornati.
La figura 2.9 mostra il diagramma di transizione degli stati del protocollo Dragon. L'assenza dello stato invalid porta quindi a fare una distinzione sulle richieste provenienti dal processore, in particolare se le richieste fanno o meno riferimento ad un blocco presente o meno nella cache. Oltre alle richieste di lettura (`LOAD`) e scrittura (`STORE`) di un blocco presente in cache, consideriamo anche la generazione dei fault di cache a seguito di operazioni di lettura (`Fault_L`) e scrittura (`Fault_S`).
Le azioni che il controllore della coerenza può intraprendere a seguito delle richieste del nodo a cui è associato sono: la richiesta di lettura (`RICH_L`) e di aggiornamento (`RICH_UP`); inoltre possono essere eseguite la riscrittura in memoria del blocco (`WRITE_BACK`) e l'aggiornamento (`UPDATE`). La richiesta di aggiornamento riguarda una specifica parola del blocco ed è utilizzata per mantenere aggiornate le cache che condividono un blocco. Per far questo viene messa in evidenza l'azione `UPDATE` con cui ogni controllore della coerenza

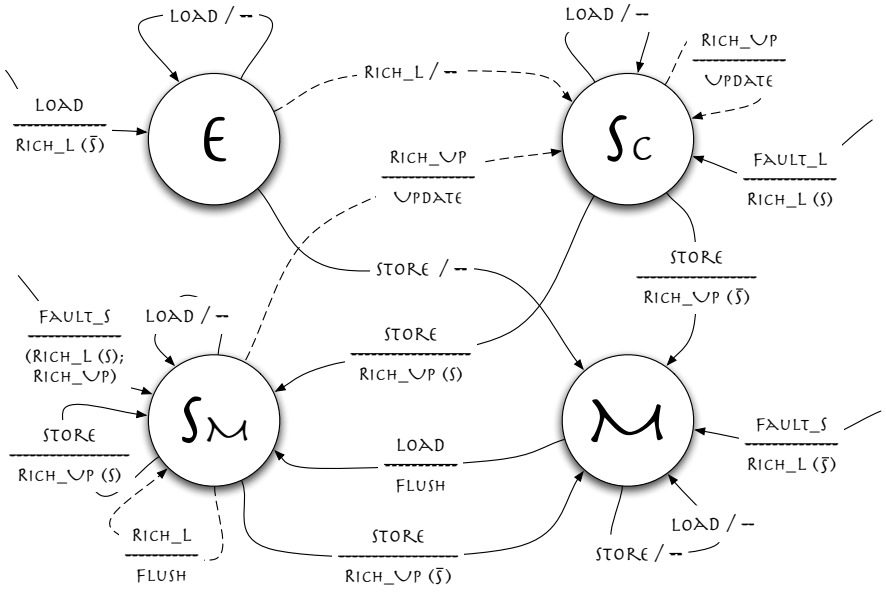


Figura 2.9: Il protocollo Dragon

mantiene aggiornata la copia locale dei blocchi aggiornati.
Analizziamo ora quali sono le azioni intraprese a seguito di un fault per una richiesta di lettura, a seguito di una scrittura (che causa o meno un fault) e di un rimpiazzamento del blocco.

Fault a seguito di una richiesta di lettura Viene generata una richiesta di lettura del blocco. In base allo stato, cioè se è presente (S) o meno ($\bar{S}$) in un'altra cache, il blocco viene memorizzato in cache in stato exclusive o shared-clean rispettivamente. Se il blocco si trova in stato modified o shared-modified in un'altra cache, allora questa è incaricata di inviare i dati aggiornati e, se diverso, lo stato è ora shared-modified. Se il blocco si trova in stato shared-clean in altre cache i dati possono essere presi dalla memoria principale. In entrambi i casi il blocco viene memorizzato nella cache in stato shared-clean. Se il blocco non è presente in nessuna altra cache i dati vengono presi dalla memoria principale e il blocco viene memorizzato in stato exclusive.

Scrittura Se il blocco si trova in stato modified, allora la scrittura non comporta alcuna azione. Se lo stato è exclusive l'unica cosa necessaria è il cambiamento di stato in modified. Nei casi in cui lo stato del blocco è shared-clean o shared-modified viene inviata la richiesta di aggiornamento e lo stato, se non lo era già, viene modificato in modified. Ogni altra cache che mantiene una copia del blocco deve di conseguenza aggiornare il proprio blocco e cambiare lo stato in shared-clean se necessario. Se nessuna altra cache ha una copia del blocco, lo stato cambia in modified.
Se invece la richiesta di scrittura genera un fault, viene inviata una richiesta di lettura del blocco e successivamente una di scrittura. Se il blocco è presente in altre cache, viene generata una richiesta di aggiornamento e il blocco è memorizzato in stato shared-modified, altrimenti lo stato scelto è modified.

Rimpiazzamento del blocco La richiesta di rimpiazzamento del blocco causa la riscrittura in memoria solo se lo stato del blocco è modified o shared-modified. Se invece il blocco è mantenuto in stato shared-clean, allora o esiste una copia in un'altra cache in stato shared-modified, o non esiste e, in questo caso, la memoria è aggiornata.

2.3.5 Proprietà di Inclusione con più Livelli di Cache

Fino ad ora abbiamo considerato per semplicità che il sistema preso in considerazione abbia una gerarchia di memoria con un solo livello di cache. La maggior parte dei sistemi moderni invece fanno spesso uso di una cache di secondo livello on-chip e di una di terzo livello off-chip. Il fatto di avere più livelli di cache sembrerebbe complicare il mantenimento della coerenza in quanto le operazioni effettuate da parte del processore sulla cache di primo livello potrebbero non essere visibili al controllore della coerenza e, ad esempio, in un sistema basato su snooping le transazioni potrebbero non essere direttamente visibili alla cache di primo livello.
Vediamo ora come è possibile estendere i meccanismi studiati nel caso di una gerarchia di memoria con due livelli di cache; l'estensione a casi con più livelli di cache è del tutto lineare. Il modo più ovvio è quello di prevedere di avere

un'unità che funge da controllore della coerenza indipendente per ogni livello della gerarchia. In questo modo però, oltre a rendere complicata la progettazione del sistema, si può facilmente riscontrare che nella maggior parte dei casi i blocchi presenti nella cache L_1 sono anche presenti nella cache L_2, rendendo di fatto inutili le comunicazioni (lo snoop, nel caso di tecniche di snooping) effettuate dalla cache L_1.
Le soluzioni usate nella pratica si basano su questa ultima osservazione. Nell'utilizzo di sistemi con gerarchia di memoria con più livelli di cache si assicura che queste preservino la *proprietà di inclusione*, la quale richiede che:

- se un blocco di memoria è presente nella cache L_1, allora deve essere presente anche nella cache L_2; in altre parole, il contenuto della cache L_1 deve essere un sottoinsieme del contenuto della cache L_2;
- se il blocco si trova in uno stato che indica che il blocco può essere modificato (ad esempio modified in MESI, owned in MOSI, shared-modified in Dragon) nella cache L_1, allora deve essere marcato con lo stesso stato anche nella cache L_2.

La prima proprietà assicura che tutte le operazioni effettuate da un altro nodo che sono rilevanti per la cache L_1 sono anche rilevanti per la cache L_2, quindi è sufficiente una sola unità che funge da controllore della coerenza collegata alla cache L_2. La secondo proprietà assicura che se una richiesta di un blocco che è presente in uno stato modificato nella cache L_1 o nella cache L_2, allora è sufficiente tener traccia di questa informazione solo per la cache L_2.

Mantenimento dell'inclusione Vediamo quali sono le problematiche da affrontare al fine di garantire la proprietà di inclusione. Si devono tenere in considerazioni tre aspetti fondamentali:

- le operazioni effettuate dal processore sulla cache L_1 causano cambiamenti di stato dei blocchi e rimpiazzamenti; questo deve avvenire in modo da mantenere l'inclusione;
- le richieste provenienti da altri nodi causano cambiamenti di stato e richieste di invio dei dati aggiornati relativamente ai blocchi presenti

nella cache L_2; queste richieste devono essere inoltrate, se necessario, alla cache L_1;

- la modifica di un blocco deve essere propagata al di fuori del nodo di elaborazione.

A prima vista sembrerebbe che l'inclusione possa essere automaticamente mantenuta facendo in modo che i fault della cache L_1 siano trattati dalla cache L_2. Il problema è che la realizzazione di questo approccio può essere complicato dall'utilizzo di alcune tecniche tipicamente implementate nelle gerarchie di cache, come: politiche di rimpiazzamento dei blocchi basate sulla storia degli accessi (ad esempio la politica di rimpiazzamento LRU), l'utilizzo di più cache allo stesso livello (ad esempio la cache di primo livello suddivisa in cache istruzioni e cache dati) o l'utilizzo di differenti dimensioni dei blocchi (σ_1 e σ_2) tra i due livelli di gerarchia. Per non rinunciare ai vantaggi ottenuti dall'uso di queste tecniche è necessario mantenere l'inclusione in modo esplicito estendendo i meccanismi usati per propagare gli eventi legati alla coerenza attraverso la gerarchia.
Ogni volta che un blocco di cache L_2 viene rimpiazzato, l'indirizzo del blocco deve essere inviato alla cache L_1 richiedendo l'invalidazione o l'invio dei dati (se modificato) dei blocchi corrispondenti (possono essere più blocchi nel caso $\sigma_2 > \sigma_1$).
Considerando le richieste proveniente dagli altri nodi, alcune delle richieste che sono rilevanti per la cache L_2 possono essere rilevanti anche per la cache L_1 e per questo devono essere propagate. Ad esempio se un blocco deve essere invalidato nella cache L_2, allora l'invalidazione deve essere propagata, se i dati sono presenti, anche alla cache L_1. Propagare tutte le richieste che sono rilevanti per la cache L_2 alla cache L_1 è la soluzione più semplice che però in molti casi risulta inutile. Per evitare questo una delle tecniche utilizzate è quella che prevede l'utilizzo un'informazione di stato (bit di inclusione) che tiene traccia dei blocchi di cache L_2 che sono presenti anche nella cache L_1. Questo permette di filtrare solo le richieste effettivamente rilevanti per la cache di primo livello.
Un'altra questione di cui è necessario tener conto riguarda le scritture effettua-

te dal processore sulla cache L_1. Le modifiche devono infatti essere comunicate anche alla cache L_2 così che possa inviare i dati aggiornati quando necessario. La soluzione più semplice è quella di utilizzare la tecnica di riscrittura write-through nella cache L_1. Si può però continuare a sfruttare i vantaggi della tecnica write-back in quanto non è strettamente necessario che i dati in L_2 siano aggiornati, basta infatti che la cache L_2 sappia che la cache L_1 mantiene i dati aggiornati. Le informazioni di stato della cache L_2 devono quindi indicare che la copia è modificata ma non aggiornata, in modo che il comportamento del nodo nei confronti del protocollo di cache coherence sia sempre quello di detentore di un blocco modificato e quando richiesto i dati possono essere inviati dalla cache L_1. Un semplice approccio è quello di settare entrambi i bit di modifica e di invalidazione nella cache L_2.
Mantenere la proprietà di inclusione permette di ottenere vantaggi a livello di prestazioni, anche in sistemi in cui un livello della gerarchia di cache è condiviso (come mostrato in [13]), in quanto si evitano, come detto anche in precedenza, di effettuare comunicazioni inutili.
Altre tecniche per il mantenimento della proprietà di inclusione delle cache sono presentate in [4].

2.4 Conclusioni

In questo capitolo abbiamo visto come il fondamentale utilizzo delle memorie cache nelle architetture multiprocessor ha portato alla nascita del problema della cache coherence.
Il problema nasce come conseguenza delle architetture all-cached e i due meccanismi che generalmente vengono messi a disposizione per farvi fronte sono l'invalidazione e l'aggiornamento.
Numerosi sono i protocolli sviluppati per descrivere in modo sempre più efficiente le azioni che devono essere intraprese dai nodi di elaborazione di un multiprocessor per risolvere il problema, attraverso i meccanismi messi a disposizione dall'architettura. Ognuno di questi protocolli prevede un diverso insieme di possibili stati da associare a ciascun blocco di cache e le diverse unità che fungono da controllori della coerenza dei nodi implementano in ma-

niera distribuita il protocollo.
In particolare tutti i protocolli presi in analisi possono essere estesi alle architetture che, come nei sistemi moderni, utilizzano una gerarchia di memorie con più livelli di cache. Inoltre ognuno di questi protocolli si semplifica nel caso di architetture, soprattutto multicore, che utilizzano la tecnica di condivisione di alcuni livelli di cache.
Nel capitolo successivo analizzeremo come questi protocolli vengono implementati attraverso le tecniche di cache coherence automatica.

Capitolo 3

Tecniche di Cache Coherence Automatica

Le tecniche di cache coherence automatica permettono di sviluppare programmi in maniera indipendente dalle operazioni necessarie per la gestione della coerenza. Infatti tutto il supporto della cache coherence viene delegato al livello firmware.
In questo capitolo andiamo ad analizzare le due principali categorie di tecniche di cache coherence automatica:

- *Snoopy-based*, presentata nella sezione 3.1, con la quale si utilizza un bus come punto di centralizzazione a livello firmware;
- *Directory-based*, che implementa i protocolli di cache coherence utilizzando strutture condivise in memoria principale, come descritto nella sezione 3.2; questa soluzione viene adottata in sistemi a più alto grado di parallelismo, facenti uso di strutture di interconnessione complesse.

3.1 Snoopy-based

Una semplice soluzione al problema della cache coherence è quella che fa uso di un punto di centralizzazione a livello firmware, in particolare l'utilizzo di un bus, detto anche *Snoopy bus* (dall'inglese *to snoop*, curiosare). Ogni dispositivo che vi è collegato può osservare ogni transazione che avviene sul bus, ad esempio ogni richiesta di scrittura o lettura da parte degli altri dispositivi.

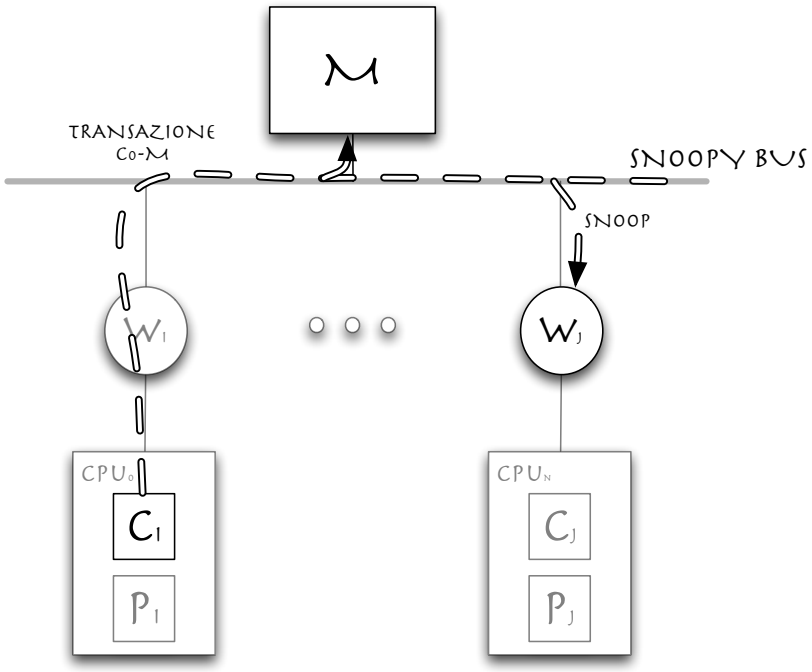


Figura 3.1: Tecnica snoopy-based

Quando un processore invia una richiesta verso la cache, l'unità W, che funge da controllore della coerenza, si occupa di esaminare lo stato della cache e intraprendere le necessarie azioni, le quali potrebbero includere la generazione di transazioni su bus per l'accesso in memoria. La coerenza è quindi mantenuta facendo in modo che ogni controllore della coerenza sia collegato al bus e osservi ogni transazione in modo da monitorare l'attività degli altri nodi, come mostrato in figura 3.1. L'unità W intraprende delle azioni solo a seguito dello *snoop* di transazioni che sono rilevanti, cioè che riguardano blocchi di memoria di cui ne esiste una copia nella propria cache. Al fine di fornire un supporto alla coerenza il bus deve quindi assicurare che:

- tutte le transazioni che avvengono tramite il bus siano visibili a tutti i controllori delle cache ad esso connesso;
- le transazioni siano visibili a tutti i controllori nello stesso ordine, quello in cui avvengono sul bus.

Per determinare se la transazione sul bus è rilevante per la propria cache le operazioni intraprese sono le stesse che avvengono a seguito delle richieste da parte del processore per verificare se un blocco è presente o meno nella cache. Le azioni intraprese possono riguardare l'invalidazione o l'aggiornamento del contenuto o dello stato del blocco di cache e/o l'invio dei dati aggiornati presenti nella cache sul bus.

3.1.1 Protocollo snooping per la cache coherence

In un sistema che utilizza la tecnica di snooping per mantenere la coerenza delle cache, ciascun controllore della coerenza riceve due insiemi di input:

- le richieste di accesso in memoria da parte del nodo;
- le informazioni di snooping circa le transazioni richieste da parte degli altri nodi.

In risposta a questo l'unità W si occupa di aggiornare lo stato del blocco di cache coinvolto dalla richiesta corrente in accordo allo stato corrente e al diagramma di transizione degli stati.
Quindi un protocollo di snooping può essere definito come un algoritmo distribuito rappresentato da un insieme di macchine a stati finiti cooperanti, definito dai seguenti componenti:

- l'insieme degli stati associati ai blocchi della cache locale;
- il diagramma di transizione degli stati, che ha come input lo stato corrente e la richiesta del processore o la transazione osservata sul bus, e produce come output lo stato successivo per il blocco;
- le azioni associato a ciascuna transizione di stato, che sono determinate in parte dall'insieme delle azioni possibili definite dal bus, dalla cache e dal processore.

Come introdotto nel capitolo 2, l'insieme delle operazioni necessarie per l'esecuzione di una transizione di stato devono essere eseguite in maniera atomica.

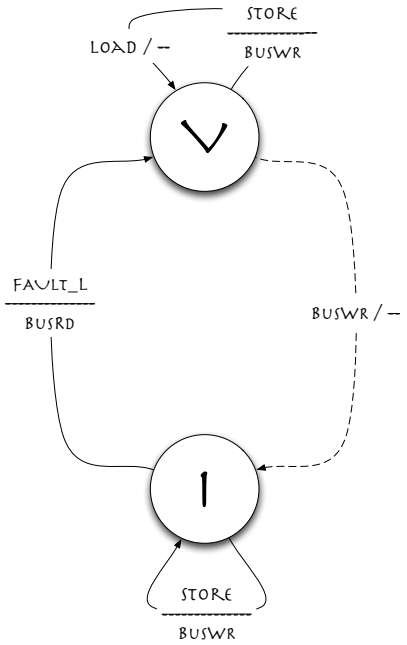


Figura 3.2: Esempio base di protocollo snoopy-based

In questo caso particolare, l'atomicità delle transazioni è garantita dal bus (*atomic bus*); infatti, solo una transizione alla volta può essere in esecuzione sul bus.
Proviamo a capire ora come interagiscono le unità W con il bus al fine di mantenere la coerenza. Partiamo dal caso più semplice di un protocollo basato su invalidazione con meccanismo di scrittura in memoria write-through, senza allocazione dei blocchi per la scrittura. In particolare, supponiamo che il bus metta a disposizione le seguenti transazioni:

- *BusRd* per la richiesta di lettura, che comprende l'indirizzo del dato richiesto;
- *BusWr* per la richiesta di scrittura, che, oltre all'indirizzo, comprende i dati da scrivere;

Come mostrato in figura 3.2, quando l'unità W osserva la richiesta di lettura da parte del processore di un dato che non è presente nella cache, allora viene generata una transazione BusRd che, una volta completata, permette di me-

morizzare il blocco in cache con lo stato valido. Ogni volta che il controllore osserva una richiesta di scrittura da parte del processore viene generata una transazione BusWr per l'aggiornamento del dato in memoria principale, senza dover effettuare cambiamenti di stato nei blocchi di cache. Quando l'unità W, a seguito dello snooping delle richieste inviate sul bus dagli altri nodi, osserva che un blocco presente nella propria cache deve essere scritto da un altro nodo, allora il blocco deve essere invalidato.

Mantenimento della coerenza nel caso write-through In un protocollo basato su write-through tutte le scritture appaiono sul bus, caratteristica che semplifica di molto il mantenimento della coerenza. Infatti, visto che può avvenire solo una transazione alla volta sul bus, in ogni esecuzione tutte le scritture riferite ad una locazione sono serializzate secondo l'ordine in cui appaiono sul bus. Dato che ciascun controllore della coerenza effettua l'invalidazione durate la transazione, le invalidazioni sono effettuate da tutti i controllori in questo stesso ordine. In questo modo il protocollo impone un ordinamento parziale delle operazioni, a partire dal quale è possibile costruire un ipotetico ordinamento totale che permette il mantenimento della coerenza delle cache. In particolare l'ordinamento può essere formalizzato come segue:

- un'operazione in memoria M_1 segue l'operazione in memoria M_2 se le operazioni sono generate dallo stesso processore nello stesso ordine;
- un'operazione di lettura segue un'operazione di scrittura W se la richiesta di lettura genera una transazione sul bus che segue quella generata per W;
- un'operazione di scrittura segue un'operazione di lettura o scrittura in memoria M se M genera una transazione sul bus e la transazione generata dalla scrittura segue quella per M;
- un'operazione di scrittura segue un'operazione di lettura se la richiesta di lettura non genera transazioni sul bus (fa riferimento ad un blocco già presente in cache) e non è separata dalla scrittura da altre transazioni sul bus.

Come abbiamo detto, la facilità con cui è possibile stabilire questo ordinamento è data dal fatto che ogni istruzione di `STORE` viene immediatamente tradotta in una transazione BusWr.
Andiamo ad analizzare ora cosa avviene nei protocolli che utilizzano tecnica di riscrittura in memoria write-back. Come abbiamo visto nella sezione 2.3, le cache di tipo write-back comportano l'introduzione dello stato modified per un blocco presente in cache e su cui vengono eseguite delle operazioni di scrittura durante l'arco di tempo che passa dal momento in cui il blocco viene caricato in cache e il momento in cui i dati vengono riscritti in memoria o il blocco viene rimpiazzato. Questo comporta il possesso esclusivo del blocco da parte della cache; esclusività che comporta che la cache può modificare il blocco senza notificare ciò agli altri nodi.
Per permettere questo il bus deve mettere a disposizione un particolare tipo di transazione chiamata *lettura esclusiva (BusRdX)*, inviata dall'unità W sul bus quando riceve una richiesta di scrittura da parte del processore:

- di un blocco che non è presente in cache;
- o di un un blocco presente in cache ma che non è nello stato modified.

Quando l'unità W osserva sul bus una transazione di questo tipo deve controllare se il blocco a cui la richiesta fa riferimento è presente in cache e, in questo caso, deve invalidare la propria copia.
Un altro tipo di transazione necessaria con l'utilizzo delle cache write-back, è la transazione *BusWB* che permette appunto di riscrivere il contenuto di un blocco in memoria a seguito del suo rimpiazzamento da parte della cache. Questa transazione viene inoltre generata dall'operazione di `FLUSH` a seguito della richiesta, da parte di un altro nodo, di un blocco presente in cache in stato modified.
Nel protocollo MESI, presentato nella sezione 2.3.2, ma anche nel protocollo Dragon (sezione 2.3.4), l'introduzione del concetto di condivisione di un blocco porta ad una distinzione nel cambiamento di stato di un blocco nel caso questo sia o meno in stato shared in un'altra cache. Nei protocolli che utilizzano lo *Snoopy Bus* è necessario che la struttura di interconnessione fornisca un meccanismo per determinare quando si verifica questa condizione. Per far questo

il bus spesso fornisce un segnale addizionale, chiamato segnale di condivisione (*shared signal*, S). Quando un controllore della cache osserva una transazione sul bus e determina se il blocco è presente in cache, in caso positivo deve inviare il segnale di condivisione che viene messo in OR con tutti i segnali provenienti dagli altri nodi. In questo modo l'unità W che aveva inviato la transazione può determinare se il blocco in questione è presente nelle altre cache in stato shared.

Mantenimento della coerenza nel caso write-back Come abbiamo visto, la transazione BusRdX assicura che la cache che sta scrivendo all'interno di un blocco ha l'unica copia valida, proprio come avviene con la BusWr nelle cache write-through. Anche se non tutte le operazioni di scrittura appaiono sul bus, sappiamo che tra due transazioni che si riferiscono al blocco di cui il processore P_i ha richiesto la copia esclusiva, solo P_i può effettuare operazioni di scrittura nel blocco stesso. Quando un altro processore P_j richiede la lettura del blocco, sappiamo che che c'è almeno una transazione sul bus (generata dall'operazione `FLUSH`) per quel blocco che separa il completamento delle operazioni di scrittura da parte di P_i dall'operazione di lettura richiesta da P_j. Questa transazione assicura quindi che le operazioni di lettura vedano tutti gli effetti delle precedenti operazioni di scrittura.

3.1.2 Condivisione tra cache (*cache-to-cache sharing*)

Un'interessante problematica che nasce con l'utilizzo delle tecniche di cache coherence automatica basate su snooping è la scelta di chi si deve occupare dell'invio di un blocco a seguito dell'osservazione di una transazione BusRd o BusRdX nel caso in cui il blocco è aggiornato sia in memoria che nella propria cache. Con l'introduzione dello stato *exclusive* nel protocollo MESI (sezione 2.3.2) infatti, se la cache mantiene un blocco in stato *shared* di cui viene fatta una richiesta di lettura o scrittura da parte di un'altra cache, allora la cache può o meno eseguire l'operazione di `FLUSH`; questa scelta dipende dal fatto che sia o meno abilitata la *condivisione tra cache*. La figura 2.7 mostra infatti che l'operazione `FLUSH` è opzionale (nell'immagine è scritta tra

parentesi). Questo perché a volte può essere conveniente, dal punto di vista del tempo di trasferimento del blocco, che sia incaricata una delle cache che possiede il blocco aggiornato di inviare i dati al nodo che ne ha fatto richiesta.

Il protocollo MESIF Quando il blocco risiede in più cache, è necessario un algoritmo di selezione per determinare quale di queste debba fornire i dati. Il protocollo MESIF è stato sviluppato da Intel e presentato in [9] per risolvere questa problematica.
Per far questo viene introdotto un nuovo stato ***F**orward* il quale indica che la cache che mantiene il blocco in questo stato è stata designata per rispondere ad ogni richiesta che fa riferimento al blocco stesso.

3.2 Directory-based

Nei sistemi a più alto grado di parallelismo vengono utilizzate strutture di interconnessioni che permettono una maggiore scalabilità rispetto a quella che si può ottenere con reti a latenza lineare, come è stato già discusso nella sezione 2.1. Questa scelta si riflette anche sulla decisione di integrare nelle architetture dei meccanismi di cache coherence automatica che scalino meglio rispetto alle soluzioni basate sullo Snoopy bus. Infatti i protocolli basati sulla tecnica di snooping necessitano che ogni nodo, in particolare ogni unità W, che funge da controllore della coerenza, possa comunicare con ogni altro nodo per poter implementare il protocollo stesso. Per indicare questa tipologia di architetture, tipicamente di tipo NUMA, che forniscono in maniera primitiva un supporto scalabile alla cache coherence, si utilizza il termine *CC-NUMA (Cache-Coherent, Non Uniform Memory Access).*
L'utilizzo di un meccanismo di cache coherence che sia scalabile si basa tipicamente sul concetto di *directory.* Poiché lo stato di un blocco nelle cache non si può più determinare implicitamente inviando una richiesta sul bus condiviso e ottenendo le informazioni relative attraverso lo snooping da parte del controllore della cache, l'idea è quella di mantenere lo stato esplicito in un posto preciso, chiamato directory. Immaginiamo che ad ogni blocco di memoria corrispondente ad un blocco di cache sia associato un record contenente

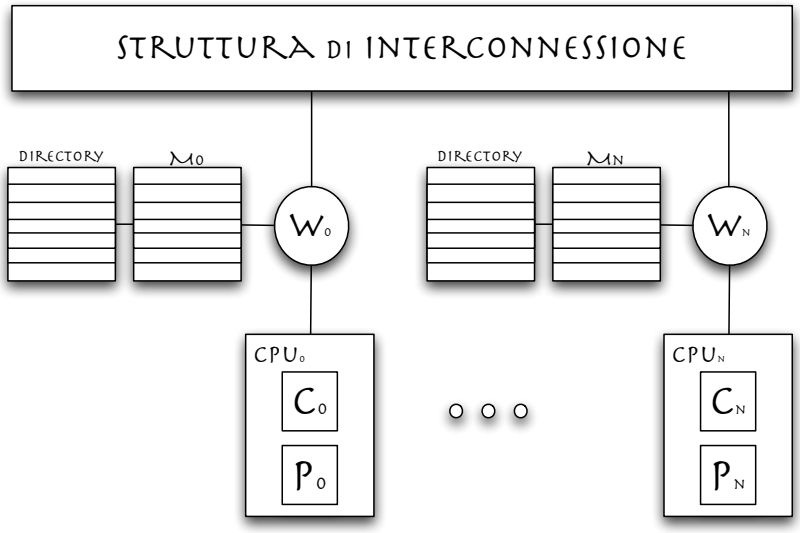


Figura 3.3: Architettura NUMA con directory

l'insieme delle cache che correntemente mantengono una copia del blocco e lo stato corrispondente. Questo record rappresenta la *directory entry* di quel blocco, come mostrato in figura 3.3. Come nelle tecniche basate sull'utilizzo dello Snoopy bus, ci possono essere più cache che possiedono una copia del blocco aggiornata, ma se il blocco è modificato allora solo una cache, quella che lo sta modificando, ne mantiene l'unica copia valida. Quando un nodo genera un fault di cache per un blocco, prima accede alla directory del blocco, poi determina, in base alle informazioni presenti nella directory entry, dove sono le eventuali copie valide del blocco e quali azioni intraprendere. Possono essere necessarie ulteriori comunicazioni da parte del nodo al fine di mantenere la coerenza del blocco. Tutti gli accessi alle directory e le comunicazioni tra nodi per il mantenimento della coerenza delle cache vengono gestiti dall'unità W, che funge da controllore della coerenza.

Per la loro natura, i meccanismi basati su directory sono indipendenti dal tipo di struttura di interconnessione utilizzata nell'architettura.

Il protocollo di cache coherence utilizzato nei sistemi directory-based può essere basato su invalidazione o aggiornamento, o una soluzione ibrida, e l'insieme degli stati è molto spesso lo stesso, normalmente quello del protocollo MESI,

discusso nella sezione 2.3.2. Dato un qualsiasi protocollo, un sistema coerente deve fornire il meccanismo che gestisce il protocollo; in particolare deve eseguire le seguenti operazioni descritte dal protocollo a seguito di un fault di cache:

1. trovare le informazioni necessarie per determinare lo stato del blocco nelle altre cache e determinare quali azioni intraprendere;

2. determinare dove sono le altre copie del blocco, se necessario (ad esempio per invalidarle);

3. comunicare con i nodi che mantengono le altre copie del blocco (ad esempio per mantenere aggiornate le informazioni di stato).

Nei protocolli basati sulle tecniche di snooping, tutte queste operazioni sono utilizzate attraverso comunicazioni broadcast e mediante l'osservazione delle transazioni che avvengono sul bus. Nei protocolli basati su directory invece, le informazioni vengono ottenute accedendo alla directory del blocco, mentre l'individuazione delle altre copie del blocco e le eventuali comunicazioni tra nodi avvengono attraverso comunicazioni interprocessor tra nodi, senza utilizzare comunicazioni di tipo broadcast.
I vari meccanismi basati su directory si distinguono in base a come avvengono le prime due operazioni del protocollo di coerenza: individuare la directory entry del blocco e determinare quali altre cache possiedono copie rilevanti del blocco. La prima operazione permette di individuare due possibili schemi che sono una valida alternativa ad una soluzione centralizzata: il più semplice viene detto *flat* ed è presentato nella sezione 3.2.1, mentre lo schema *gerarchico* viene descritto nella sezione 3.2.2. Lo schema flat può ulteriormente essere suddiviso in due approcci in base al modo in cui vengono localizzate le altre copie di un blocco; le sezioni 3.2.1 e 3.2.1 mostrano le differenze tra l'approccio basato su memoria e su cache rispettivamente. La figura 3.4 sintetizza la classificazione dei diversi schemi directory-based.

In seguito utilizzeremo la seguente notazione per distinguere i nodi di elaborazione che interagiscono tra loro; per un dato blocco:

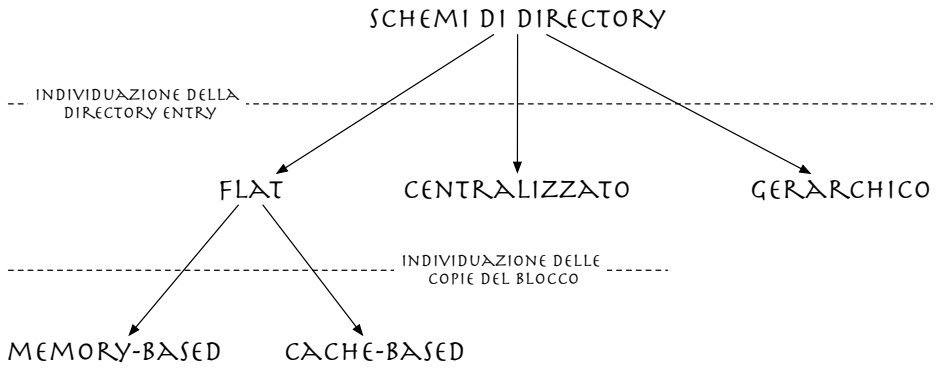


Figura 3.4: Schemi basati su directory

- *home*: è il nodo nella cui memoria principale é allocato il blocco;
- *local*: è il nodo che richiede il trasferimento del blocco di cache a seguito di un fault;
- *dirty*: è il nodo che ha una copia del blocco in cache in stato modified (dirty); notiamo che il nodo dirty può anche coincidere con il nodo home;
- *owner*: è il nodo che correntemente mantiene la copia valida del blocco e deve fornire i dati quando necessario; in un protocollo basato su directory questo nodo corrisponde al nodo home (se il blocco non è in stato modified in nessuna cache) o al nodo dirty;
- *exclusive*: è il nodo che mantiene una copia del blocco in cache in stato exclusive, la memoria può o meno essere aggiornata; il nodo dirty è quindi anche un nodo exclusive;

3.2.1 Flat

Lo schema di tipo *flat* viene chiamato così in quanto le informazioni relative alla directory di un blocco sono localizzate in un ben fissato posto, tipicamente sul nodo home che è determinato a partire dall'indirizzo del blocco. A seguito di un fault per l'accesso ai dati di un blocco, un'unica richiesta viene inviata direttamente al nodo home (se il nodo è remoto) per accedere alla directory

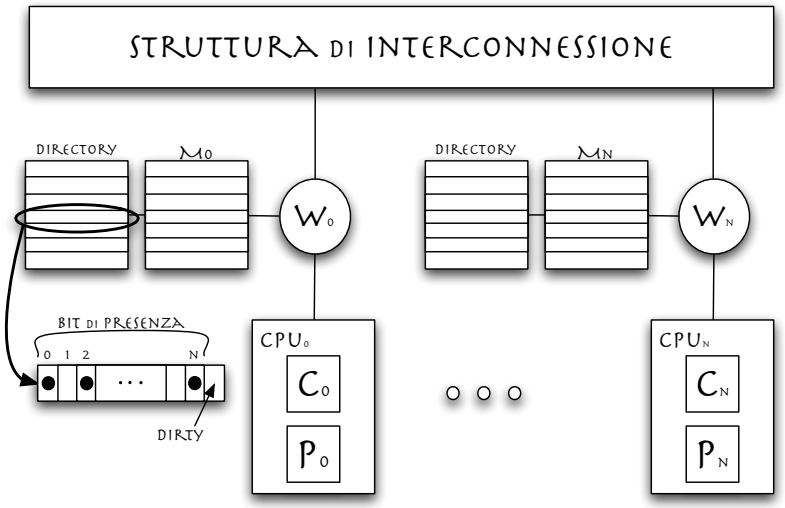


Figura 3.5: Directory memory-based

del blocco. Uno schema di tipo flat può essere diviso in due classi principali, in base a dove possono essere reperite tutte le informazioni relative ad un blocco:

- schemi *memory-based*, che memorizzano le informazioni di directory di tutte le copie nelle cache di un blocco nel nodo home del blocco stesso;
- schemi *cache-based*, nei quali le informazioni delle copie di un blocco non sono tutte contenute nel nodo home, ma sono distribuite tra le cache che contengono le varie copie; il nodo home contiene semplicemente un puntatore al una delle copie del blocco e attraverso una linked-list distribuita è possibile risalire a tutte le altre copie del blocco.

Memory-based

Consideriamo l'utilizzo del protocollo basato su directory di tipo memory-based: come rappresentato in figura 3.5, le informazioni delle directory sono mantenute insieme alla memoria principale di ogni nodo.

A seguito di un fault di cache, il nodo local deve inviare una richiesta al nodo home nel quale vengono mantenute le informazioni di directory del blocco in questione.

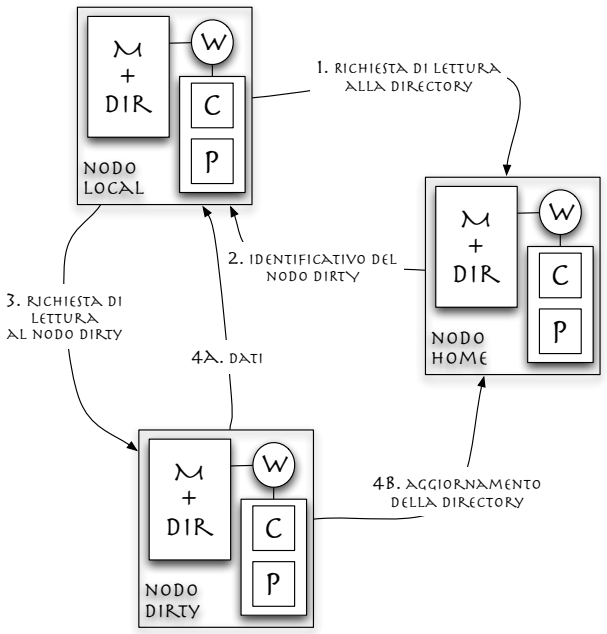


Figura 3.6: Fault di cache per la lettura di un blocco presente in un'altra cache in stato modified

Se si tratta di un fault dovuto ad un'operazione di lettura del blocco, la directory indica da quale nodo si può ottenere il dato, come mostrato in figura 3.6. Nel caso in cui il fault è avvenuto a seguito di un'operazione di scrittura, la directory permette al nodo local di identificare le copie del blocco da invalidare, come mostrato in figura 3.7. Ricordiamo inoltre che la scrittura di un blocco in stato shared è considerata come una scrittura che genera un fault di cache.

Un semplice modo di organizzare le informazioni della directory per un blocco è quello in cui si utilizza un vettore di N bit di presenza (uno per ogni nodo di elaborazione), il quale indica per ogni nodo se questo mantiene o meno nella propria cache una copia del blocco. Oltre al vettore di bit di presenza si mantiene un insieme di bit di stato del blocco; supponiamo per semplicità di utilizzare un solo bit dirty, il quale indica se il blocco è presente in stato modified in una cache. Naturalmente, se il bit dirty vale TRUE, allora solo un nodo (il nodo dirty) dovrebbe avere in cache il blocco, quindi solo uno tra i bit di presenza dovrebbe valere TRUE.

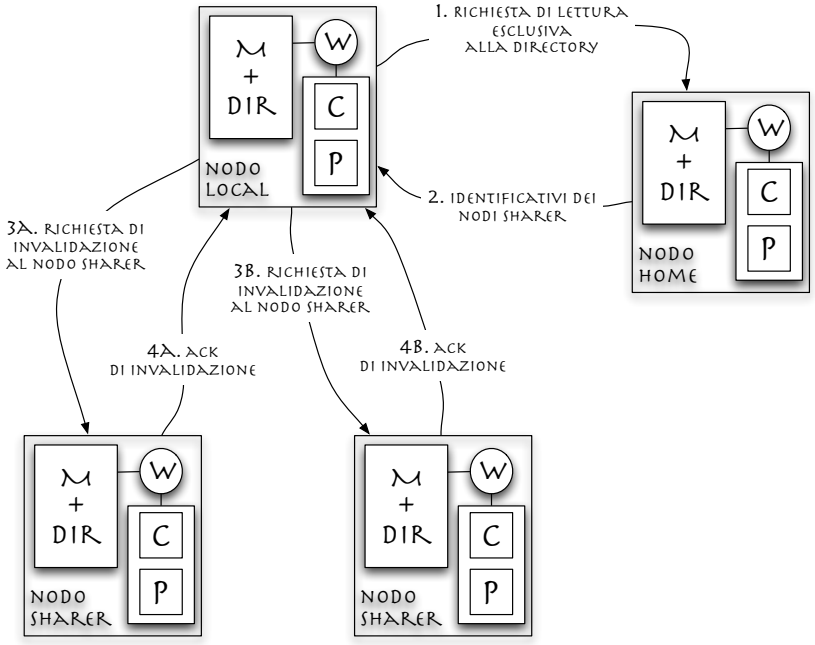


Figura 3.7: Fault di cache per la scrittura di un blocco presente in altre due cache in stato shared

Le informazioni di directory di un blocco sono quindi una semplice vista dal lato della memoria dello stato dei blocchi di cache che ne mantengono una copia; la directory non ha necessariamente bisogno di conoscere lo stato esatto di ogni cache (come richiesto ad esempio dal protocollo MESI), ma solo delle informazioni sufficienti per determinare quali azioni devono essere intraprese.
Al fine di analizzare più in dettaglio come il trattamento dei fault di cache dovrebbe interagire con l'organizzazione delle directory appena introdotta, consideriamo per semplicità il protocollo MSI in un sistema con singolo livello di cache e nodi di elaborazione uniprocessor.
Il protocollo è gestito dall'unità W di ogni nodo, che funge da controllore della coerenza. Al momento di un fault di cache nel nodo i, l'unità W_i determina, a partire dall'indirizzo del blocco di memoria se il nodo home coincide al nodo local o si tratta di un nodo remoto. Nel secondo caso viene inviata una richiesta attraverso la struttura di interconnessione al nodo home del blocco, supponiamo sia il nodo j. A questo punto l'unità W_j è incaricata del tratta-

mento del fault.
Nel caso di fault di cache a seguito di un'operazione di lettura:

- se il bit dirty vale FALSE, allora l'unità W_j legge il blocco dalla memoria principale M_j e invia i dati al nodo i e imposta l'i-esimo bit di presenza a TRUE;

- se il bit dirty vale TRUE, allora l'unità W_j invia al nodo i l'identificatore del nodo k che corrisponde al bit di presenza che vale TRUE; a questo punto l'unità W_j può inviare una richiesta di trasferimento del blocco al nodo k, il quale deve cambiare lo stato del blocco nella cache in shared e inviare il blocco sia al nodo i che al nodo j, il primo lo memorizza in cache in stato shared, mentre il nodo j memorizza il blocco in memoria modificando le informazioni della directory settando il bit dirty a FALSE e l'i-esimo bit di presenza a TRUE.

Nel caso di fault di cache a seguito di un'operazione di scrittura:

- se il bit dirty vale FALSE, allora la memoria ha una copia aggiornata del blocco; a questo punto deve essere inviata un messaggio di invalidazione a tutti i nodi j tali che il j-esimo bit di presenza vale TRUE; per far questo il nodo home, prima invia il blocco di dati al nodo i che aveva generato il fault insieme al vettore di bit di presenza, successivamente resetta i valori della directory impostando solo l'i-esimo bit di presenza e il bit dirty a TRUE (se si tratta di una richiesta di aggiornamento del blocco da parte di un nodo che ha già la versione aggiornata del blocco in cache (stato shared), viene inviato solo il vettore dei bit di presenza); a questo punto l'unità W_i può inviare le richieste di invalidazione attendendo un messaggio di *ack* dai nodi che condividevano il blocco; ora il nodo i può impostare lo stato del blocco in cache a modified;

- se il bit dirty vale TRUE, allora il blocco viene prima richiesto al nodo j il cui bit di presenza vale TRUE; in questo modo il nodo j può invalidare il proprio blocco di cache inviando i dati al nodo i che aveva generato il fault, il quale memorizza il blocco nella propria cache impostando lo

stato a modified; la directory può quindi essere aggiornata scambiando il valore del j-esimo bit di presenza con quello dell'i-esimo bit.

Nel rimpiazzamento di un blocco di cache in stato modified nel nodo i, il blocco viene riscritto in memoria (*write back*) e la directory viene aggiornata impostando l'i-esimo bit di presenza e il bit dirty a FALSE. Infine nel caso di un rimpiazzamento di un blocco di cache in stato shared, viene inviato un messaggio al nodo home con lo scopo di aggiornare la directory in modo da non dover inviare in seguito eventuali invalidazioni non necessarie.
L'organizzazione appena descritta che fa uso del vettore di bit viene anche chiamata organizzazione *full bit vector* e si tratta della scelta più immediata in un approccio di tipo *flat, memory-based*. Il principale svantaggio di questa soluzione riguarda la quantità di memoria richiesta per tenere traccia di tutte le informazioni di directory. Sono due le principali soluzioni adottate al fine di ridurre questo overhead per un dato numero di processori. La prima è quella di aumentare la dimensione dei blocchi di cache; la seconda è quella di raggruppare i processori in modo da mantenere le informazioni di directory al livello di ogni gruppo, utilizzando un protocollo a due livelli.
In realtà ci sono due soluzioni alternative più interessanti: quella che cerca di ridurre il numero di bit della *directory entry*, detta *directory width*, e quella che può ridurre il numero totale di directory entry, o *directory height*, senza avere una entry per ciascun blocco.
La prima idea viene detta *limited pointer directories* e si basa sul fatto che nella maggior parte del tempo solo un numero limitato di cache mantiene una copia del blocco. In [2] viene presentata l'idea di mantenere un numero fissato di puntatori, minore del numero di cache del sistema, che puntano ai nodi che correntemente mantengono una copia del blocco in cache.
Per diminuire il numero di directory entry, in [11] viene proposta un'organizzazione della directory stessa come cache, tenendo conto del fatto che, normalmente, solo una modesta parte dei blocchi di memoria è presente nelle cache in determinato periodo di tempo, con conseguente inutilizzo di un alto numero di directory entry.

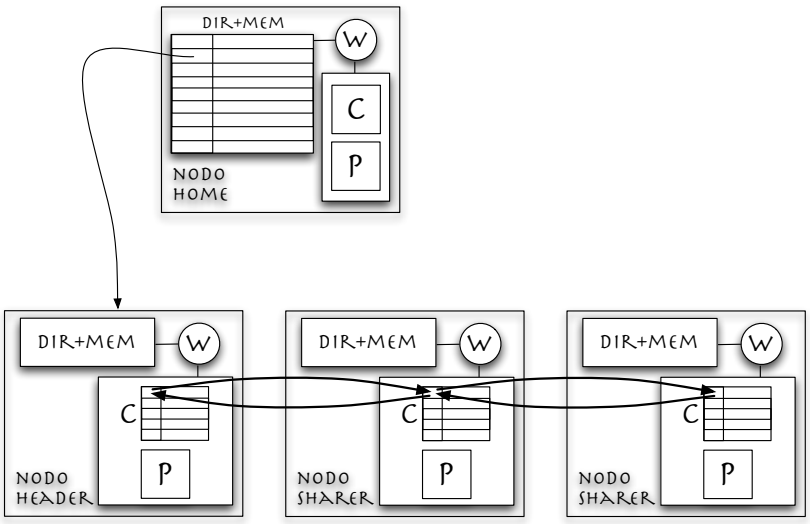


Figura 3.8: Directory cache-based

Cache-based

In un approccio cache-based è sempre utilizzato il nodo home come punto di riferimento per ottenere le informazioni di directory, la differenza rispetto all'approccio memory-based sta nel fatto che ora la directory entry non contiene le identità di tutti i nodi che mantengono una copia del blocco ma solo un puntatore al primo nodo che ha una copia in cache insieme ad un limitato numero di bit di stato. Il puntatore viene detto*head pointer* del blocco. I rimanenti nodi che mantengono una copia del blocco in cache sono collegati tra loro in una doppia *linked-list* distribuita; in particolare una cache che contiene una copia del blocco contiene inoltre un puntatore al successore ed al predecessore di questa lista, come rappresentato in figura 3.8.

Nel caso di fault di cache a seguito di un'operazione di lettura:

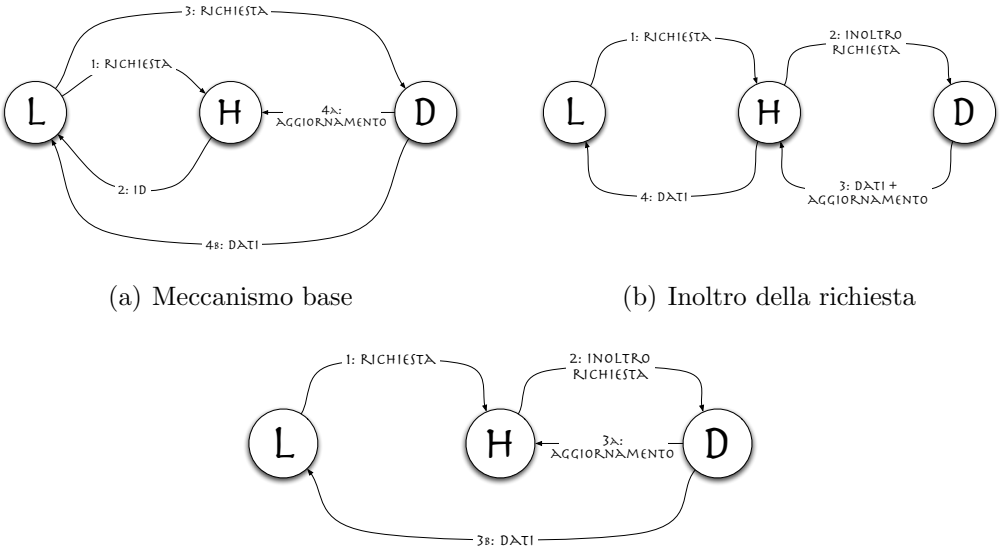
- il nodo local invia una richiesta al nodo home al fine di identificare il primo nodo che mantiene una copia del blocco;
- se il puntatore nella directory è nullo, allora non ci sono nodi che condividono il blocco e il nodo home può inviare i dati richiesti;

- se il puntatore non è nullo, il nodo local deve essere aggiunto alla lista: il nodo home invia il valore del puntatore come risposta; in questo modo il nodo local può richiedere al nodo di cui ha ricevuto l'identificatore di essere inserito in testa alla lista; i dati possono essere inviati dal nodo home se aggiornato, altrimenti dal nodo che prima si trovava in testa alla lista, il quale ha sempre una copia aggiornata del blocco in quanto nodo owner.

Nel caso di fault di cache a seguito di un'operazione di scrittura:

- il nodo local ottiene anche in questo caso dal nodo home l'identità del nodo puntato dall'header pointer;
- come prima, il nodo local deve essere inserito in testa alla lista; se il nodo è già presente nella lista in quanto mantiene una copia del blocco in cache, allora deve essere spostato in testa in quanto diviene il nodo owner;
- l'ultima cosa di cui si occupa il nodo local è quella di scorrere l'intera lista al fine di invalidare le altre copie del blocco, attendendo una conferma di avvenuta invalidazione.

Il rimpiazzamento di un blocco dalla cache richiede che il nodo elimini se stesso dalla lista dei nodi che condividono il blocco, comunicando con i nodi predecessore e successore nella lista.
Rispetto all'approccio memory-based viene risolto il problema dell'overhead di spazio utilizzato in memoria per mantenere le informazioni della directory; ora infatti ogni blocco in memoria principale ha associato il solo puntatore al nodo in testa alla lista e pochi altri bit di stato. Il numero di puntatoti al predecessore e al successore nella lista è proporzionale al numero di nodi che condividono il blocco che, come già osservato, è normalmente molto più piccolo rispetto al numero di cache del sistema. Il problema principale di questa soluzione deriva dalle operazioni di gestione della lista che possono complicare l'implementazione del protocollo. Per questo motivo è stato formalizzato uno standard per i meccanismi di basati su directory di tipo cache-based per ridurne la complessità; lo standard IEEE 1596-1992 è stato presentato in [12].



(a) Meccanismo base

(b) Inoltro della richiesta

(c) Inoltro di richiesta e risposta diretta

Figura 3.9: Riduzione delle comunicazioni inter-processor nel meccanismo memory-based

Ottimizzazione dei meccanismi flat

Una delle principali problematiche che è stata affrontata per migliorare le prestazioni dei meccanismi flat è quella che riguarda il numero di comunicazioni inter-processor generate a seguito di un fault di cache da parte di un nodo. Consideriamo cosa avviene a seguito di un fault di cache a seguito di una richiesta di lettura, nel caso in cui il nodo home è un nodo remoto e lo stato del blocco è dirty. L'insieme delle comunicazioni inter-processor necessarie per il reperimento del blocco e per mantenere aggiornate tutte le informazioni di stato necessarie al protocollo sono riassunte in figura 3.9(a). Il nodo home risponde alla richiesta del nodo local inviando l'identificativo del nodo dirty. A questo punto il nodo local può inviare la richiesta al nodo che mantiene la copia aggiornata del blocco, il quale aggiorna le informazioni di directory sul nodo home e risponde inviando i dati aggiornati.

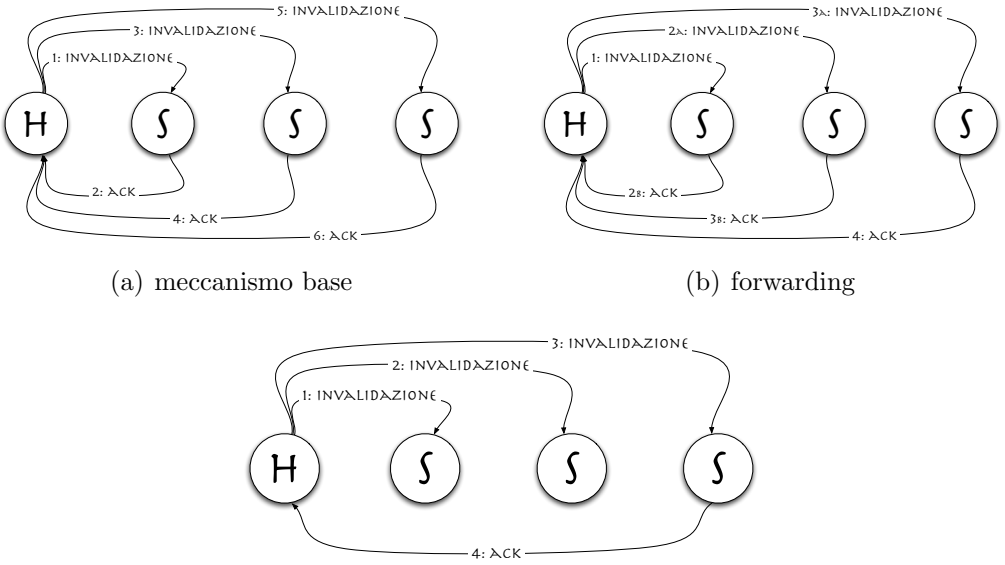
Il numero di comunicazioni inter-processor può essere ridotto attraverso l'inoltro della richiesta proveniente dal nodo local da parte del nodo home al nodo dirty. In particolare, il nodo home non risponde al nodo local ma invia la

richiesta direttamente al nodo che mantiene la copia aggiornata del blocco. A questo punto, come mostrato in figura 3.9(b), il nodo dirty può inviare i dati al nodo home che può aggiornare la directory e inviare i dati al nodo che ne aveva fatto richiesta.
Un'ulteriore ottimizzazione può essere quella rappresentata in figura 3.9(c), in cui il nodo home, al momento dell'inoltro della richiesta proveniente dal nodo local, invia al nodo dirty anche l'identificatore del nodo richiedente. In questo modo il nodo dirty può inviare direttamente i dati al nodo local e semplicemente aggiornare la directory sul nodo home.
Tecniche simili di inoltro possono essere applicate anche per ridurre il numero di comunicazioni inter-processor che sono necessarie in uno schema cache-based al momento dell'invalidazione delle cache che mantengono una copia del blocco di cui ne viene richiesta la scrittura. In una prima soluzione, mostrata nella figura 3.10(a), ogni nodo che condivide il blocco risponde alla richiesta di invalidazione proveniente dal nodo home con l'identificativo del nodo successore nella lista. Il numero di comunicazioni inter-processor è pari a due volte la lunghezza della lista.

La figura 3.10(b) mostra come un generico nodo della lista, al momento della ricezione della richiesta di invalidazione, può inoltrare la richiesta al nodo successore nella lista e inviare allo stesso tempo la conferma dell'avvenuta invalidazione al nodo home.
Il numero di comunicazioni inter-processor può effettivamente essere ridotto facendo in modo che solo l'ultimo nodo della lista invii il messaggio di conferma dell'avvenuta invalidazione del blocco al nodo home, come rappresentato in figura 3.10(c).

Correttezza dei protocolli

Come abbiamo visto, nei meccanismi basati su directory non è necessario conoscere lo stato esatto di ogni copia di un blocco, infatti è sufficiente mantenere poche informazioni nel nodo home, che sono una sorta di vista dal lato della memoria, che permettono di determinare quali azioni intraprendere al fine di garantire la coerenza. Proprio a causa del fatto che i nodi coinvolti



(a) meccanismo base

(b) forwarding

(c) forwarding con conferma finale

Figura 3.10: Riduzione delle comunicazioni inter-processor nel meccanismo cache-based

effettuano comunicazioni inter-processor per eseguire queste azioni descritte dal protocollo potrebbe verificarsi che in un particolare periodo di tempo le informazioni di directory riguardo lo stato di un blocco di cache siano incorrette. Questo avviene in quanto la modifica dello stato in una cache è stato modificato ma non è ancora avvenuto l'aggiornamento sul nodo home.
Il problema, dovuto al fatto di avere uno stato distribuito, deve essere risolto in quanto a seguito di una successiva richiesta di accesso allo stesso blocco, questa può essere trattata dal nodo home utilizzando le informazioni non ancora aggiornate, andando a compromettere la correttezza del meccanismo di coerenza adottato.
Diversi tipi di soluzioni sono state studiate per risolvere questo problema, molte di queste utilizzano degli stati addizionali nelle informazioni di directory chiamati *busy states* o *pending states*. Un blocco che si trova in stato busy nella directory indica che una precedente richiesta fatta al nodo home per questo stesso blocco è ancora in corso e non è stata completata. Al momento di una nuova richiesta per un blocco che si trova in questo stato diverse tecniche pos-

sono essere utilizzate per garantire il mantenimento della coerenza. Queste tecniche fanno spesso uso di uno tra i seguenti meccanismi:

- *Buffer sul nodo home.* Una richiesta può essere bufferizzata sul nodo home come richiesta pendente fin quando la precedente richiesta per quel blocco non è stata completata, senza tener conto del fatto che questa sia stata inoltrata al nodo dirty o che si stia usando il protocollo base senza ottimizzazioni.

- *Buffer sul nodo local.* Le richieste pendenti possono essere bufferizzate non solo sul nodo ho me ma anche sul nodo che effettua la richiesta stessa, costruendo una linked-list distribuita di richieste pendenti. Si tratta di una naturale estensione dell'approccio cache-based che fornisce già il supporto per questa soluzione. Questo meccanismo è utilizzato anche nel protocollo SCI [12].

- *Messaggio di NACK.* Una richiesta riguardante un blocco che si trova in stato busy può essere "rifiutata" da parte del nodo home, ad esempio rispondendo al nodo local con un messaggio che indica che la richiesta non può essere servita. La richiesta può quindi essere inviata di nuovo in un secondo tempo.

- *Inoltro al nodo dirty.* Se lo stato di un blocco si trova in stato pendente nella directory in quanto la richiesta precedente è stata inoltrata al nodo dirty, allora anche le successive richieste possono essere inoltrate allo stesso nodo. In questo modo è il nodo dirty che mantiene le richieste pendenti. Se lo stato del blocco nel nodo dirty cambia prima che l'inoltro di una successiva richiesta raggiunga il nodo dirty stesso (ad esempio a causa di un write-back del blocco), allora può essere utilizzato il meccanismo del messaggio di NACK per rifiutare momentaneamente la richiesta. Questo approccio è stato utilizzato nel protocollo Stantford DASH, presentato in [14].

Queste tecniche non sono però sufficienti a garantire che le transizioni di stato di un generico blocco di memoria avvengano in maniera atomica. Come abbiamo visto nella sezione 3.1.1, nelle tecniche di cache coherence automatica

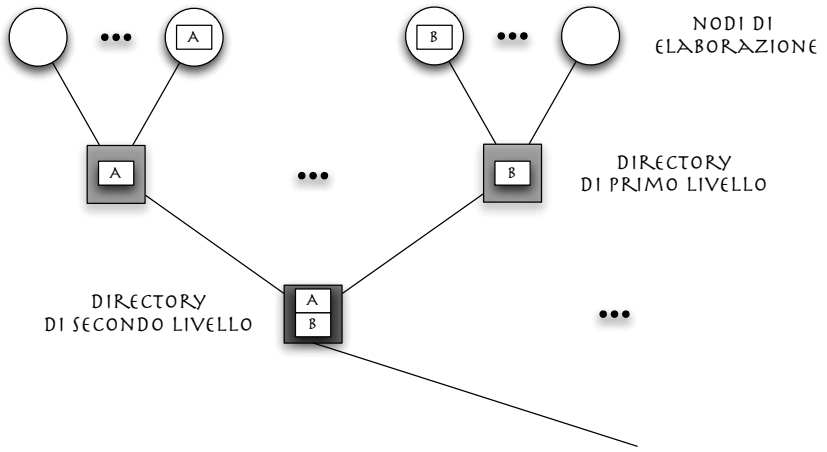


Figura 3.11: Organizzazione gerarchica delle informazioni di directory

basate su snooping, l'atomicità è fornita direttamente dal bus; nelle tecniche basate su directory non è altrettanto semplice garantire questa proprietà. È necessario infatti che, nel caso di una lettura esclusiva di un blocco, il nodo home garantisca un accesso atomico al nodo che ne ha fatto richiesta. Per far questo si possono adottare soluzioni, come quella che vedremo nella sezione 4.1.2, che permettono di bloccare temporaneamente le richieste di accesso al modulo di memoria provenienti da altri nodi, fino a quando non sono state eseguite tutte le invalidazioni necessarie.

3.2.2 Gerarchico

In uno schema di tipo gerarchico non è noto a priori dove sono memorizzate le informazioni di directory. La directory di un ciascun blocco è infatti organizzata logicamente come una struttura dati gerarchica, in particolare un albero. Ogni nodo di elaborazione, compresa la sua memoria locale rappresenta una foglia dell'albero. I nodi interni dell'albero sono semplici informazioni di directory dei blocchi organizzate secondo la logica gerarchica: un nodo tiene traccia delle copie dei blocchi mantenute da ciascuno dei propri figli. come rappresentato in figura 3.11.

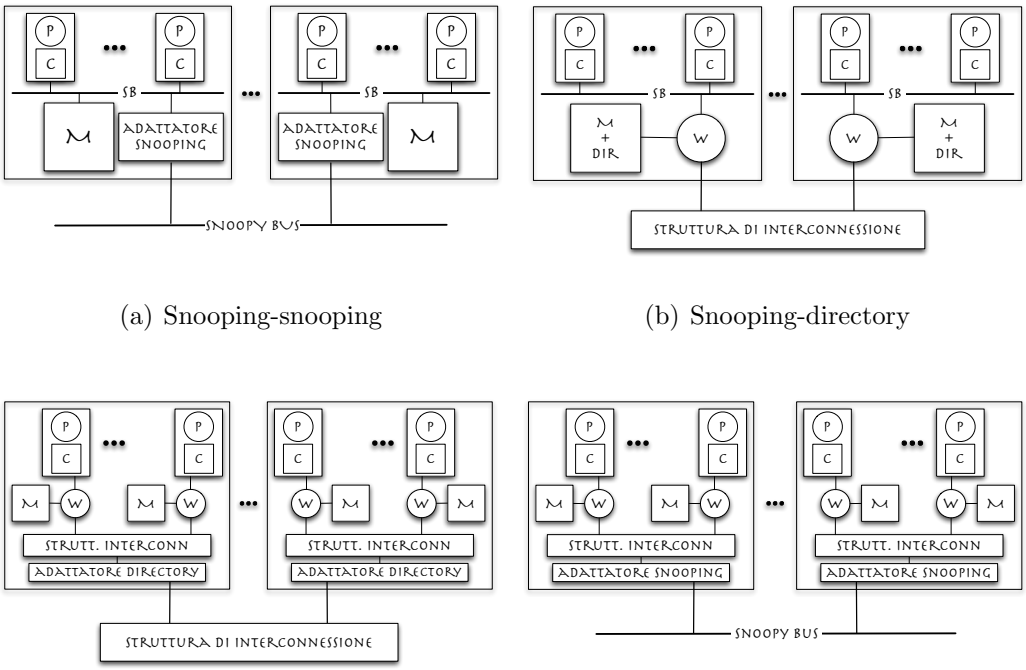
A seguito di un fault, le informazioni di directory per il blocco sono individuate attraversando i livelli della gerarchia tramite comunicazioni inter-

processor fin quando non si raggiunge un nodo directory che indica che il proprio sottoalbero mantiene il blocco. Quindi un nodo funge allo stesso tempo da foglia nel momento in cui genera un fault e da radice (nodo directory) per i blocchi di cui mantiene le informazioni di directory.
In particolare:

- un fault a seguito di una richiesta di *lettura* viene inoltrato in alto attraverso la gerarchia fin quando non si raggiunge il nodo directory indica che il suo sotto-albero mantiene una copia (dirty o no) del blocco di memoria che è stato richiesto, o fino a che la richiesta non raggiunge il primo antenato comune tra il nodo local e il nodo home del blocco, e la directory indica che il blocco non è in stato dirty al di fuori del sotto-albero. La richiesta può quindi scendere lungo la gerarchia fino al nodo appropriato per ottenere i dati, i quali vengono inviati seguendo lo stesso percorso all'indietro, aggiornando le informazioni di directory. Se il blocco era in stato dirty, una copia del blocco viene inviata allo stesso modo al nodo home.

- un fault a seguito di una richiesta di *scrittura* sale in alto lungo la gerarchia fin quando raggiungere la directory il cui sotto-albero contiene il corrente nodo owner del blocco richiesto. Il nodo owner corrisponde al nodo home o al nodo dirty relativo alla cache che ha l'ultima copia valida del blocco. La richiesta scende quindi fino al nodo owner per ottenere i dati che vengono inviati al nodo local che ne aveva fatto richiesta. Se il blocco non si trovata in stato dirty allora le invalidazioni sono propagate sempre attraverso la gerarchia fino a tutti i nodi che mantengono una copia del blocco di cache. Infine tutte le directory coinvolte durante le precedenti operazioni sono aggiornate in modo da riflettere il nuovo nodo owner e le copie invalidate.

Questo tipo di schema tende però ad essere scartato a causa delle sue prestazioni, in molti casi peggiori in confronto agli schemi di tipo flat, rendendolo meno popolare nei sistemi moderni.



(a) Snooping-snooping

(b) Snooping-directory

(c) Directory-directory

(d) Directory-snooping

Figura 3.12: Possibili combinazioni per un protocollo a due livelli

3.3 Approcci ibridi

Una soluzione alternativa è quella che prevedere l'utilizzo di un protocollo a due livelli. Ciascun nodo è esso stesso un multiprocessor. Le cache all'interno di un nodo sono mantenute coerenti tramite un protocollo di coerenza detto *protocollo interno*. La coerenza tra nodi è mantenuta da un altro protocollo detto *protocollo esterno*, che vede ciascun nodo multiprocessor come una singola cache. Per far questo viene normalmente utilizzata un'unità che funge da adattatore per il nodo nei confronti del protocollo esterno.

L'organizzazione maggiormente utilizzata è quella che prevede l'utilizzo di un protocollo basato su directory come protocollo esterno e uno basato su snooping come protocollo interno, come descritto in [17] e rappresentato in figura 3.12(b). Sono comunque utilizzate anche le altre combinazioni: snooping-snooping, directory-directory e directory-snooping, mostrate rispettivamente nelle figure 3.12(a), 3.12(c) e 3.12(d).

3.4 Conclusioni

In questo capitolo abbiamo visto come i protocolli di cache coherence presentati nel capitolo 2 possono essere implementati dal livello firmware di un'architettura multiprocessor.
Le tecniche di cache coherence automatica descritte, vengono così definite in quanto il supporto alla cache coherence è interamente delegato all'architettura, rendendo il problema trasparente ai livelli più alti.
Queste tecniche si suddividono in due principali categorie: nella soluzione snoopy-based si fa uso di un bus come punto di centralizzazione, mentre le tecniche basate su directory vengono normalmente adottate nei sistemi a più alto grado di parallelismo in quanto fanno uso di strutture condivise in memoria principale per implementare i vari protocolli.
Le due soluzioni possono essere combinate per dar luogo ad approcci ibridi.
Nel prossimo capitolo vedremo un approccio completamente opposto al problema della cache coherence, in quanto non si fa uso dei meccanismi messi a disposizione dall'architettura, bensì il problema della cache coherence viene risolto nel supporto ai meccanismi di sincronizzazione, in particolare negli algoritmi per la mutua esclusione.

Capitolo 4

Approccio Algorithm-Dependent

Una soluzione al problema della cache coherence, in alternativa alle tecniche automatiche presentate nel capitolo precedente, è il così detto *approccio Algorithm-Dependent*. Questo approccio permette di programmare esplicite strategie di cache coherence senza fare affidamento su specifiche soluzioni automatiche.
Dopo una breve introduzione ai meccanismi di sincronizzazione nelle architetture multiprocessor nella sezione 4.1, andiamo ad analizzare, nella sezione 4.2, come integrare la cache coherence nel supporto per la mutua esclusione.

4.1 Meccanismi di Sincronizzazione: Mutua Esclusione

Nelle architetture multiprocessor l'esecuzione *indivisibile* di una parte di programma, come può essere una primitiva di comunicazione, viene garantita attraverso:

- la *non interrompibilità* del processore, come avviene in un sistema uniprocessor disabilitando le interruzioni;
- l'utilizzo di operazioni di *lock* e *unlock* per la sincronizzazione.

L'algoritmo di lock più semplice che si può considerare è quello in cui un processo controlla il valore del semaforo di lock (*semlock*), implementato come

un valore booleano; se il semaforo è "libero" (*true*) allora il processore può acquisire il semaforo settandone il valore ad "occupato" (*false*). Nel caso in cui il processore, al momento della lock, trova il semaforo già occupato, allora è necessario che il processo si metta in attesa che il valore cambi. L'algoritmo di unlock dovrebbe semplicemente settare il valore del semaforo a libero. Il codice assembler illustrato nel listato 4.1 è l'implementazione delle due operazioni appena descritte, eseguite ad interruzioni disabilitate. Il linguaggio assembler utilizzato è quello descritto in [23].

Listato 4.1: Codice assembler delle operazioni di lock e unlock

```
1  lock:   LOAD    Rindsem, R0, Rsemlock, DI
2                  /*copia dello stato del semaforo
3                  nel registro Rsemlock*/
4          IF≠0    Rsemlock, lock  /*se il semaforo e'
5                                  occupato si ripete
6                                  l'operazione*/
7          STORE   Rindsem, R0, #1 /*altrimenti si occupa
8                                  il semaforo*/
9          GOTO    Rret, EI
10
11 unlock: STORE   Rindsem, R0, R0, DI /*setta a libero
12                                        il semaforo*/
13         GOTO    Rret, EI
```

Ora è necessario capire come garantire che le operazioni di lock e unlock possano essere eseguite in maniera atomica; infatti, una semplice implementazione come quella appena descritta non garantisce che la sequenza di accessi in memoria avvenga in maniera indivisibile. Per far questo si può scegliere tra due soluzioni possibili: la prima è quella che utilizza *particolari istruzioni assembler*, mentre la seconda soluzione è quella che prevede di demandare il compito di garantire l'indivisibilità al *meccanismo di arbitraggio* dei moduli di memoria comune, mantenendo un set istruzioni di tipo RISC.

4.1.1 Istruzioni assembler *read-modify-write*

Spesso i costruttori di processori forniscono direttamente istruzioni assembler per eseguire in modo indivisibile una sequenza *read-modify-write*. Ne sono esempi le istruzioni di tipo *Test&Set*, *Fetch&Op* o *Compare&Swap*. L'istru-

zione `t&s(R,L)`, dove `R` indica un registro del processore e `L` la locazione di memoria su cui operare in modo indivisibile, permette infatti di caricare il valore corrente della locazione di memoria nel registro e atomicamente scrivere la costante "1" nella locazione. In questo modo se il registro indica che il semaforo prima della scrittura era libero, allora viene occupato, altrimenti il valore in memoria rimane lo stesso. Una possibile implementazione dell'operazione lock è quella mostrata nel listato 4.2.

Listato 4.2: Codice assembler dell'operazione di lock con l'utilizzo dell'istruzione assembler *Test&Set*

```
1 lock:   T&S     Rsemlock, Rindsem, DI /*copia lo stato
2                                       del semaforo nel
3                                       registro Rsemlock
4                                       e setta lo stato
5                                       a occupato*/
6         IF≠0    Rsemlock, lock
7         GOTO    Rret, EI
```

Uno sguardo alla cache coherence

L'utilizzo di speciali istruzioni assembler che permettono di eseguire in modo indivisibile l'operazione di lock evidenziano un problema dal punto di vista della cache coherence. Infatti, nel caso in cui l'operazione di lock fallisca, il semaforo è quindi in stato "occupato" al momento della lettura, viene comunque generata l'invalidazione a causa della scrittura in memoria. Per far fronte a questa problematica, una soluzione adottata nei sistemi più moderni, è quella che utilizza due speciali istruzioni invece di una singola istruzione read-modify-write. La prima istruzione, comunemente chiamata *load-locked* o *load-linked* (LL), carica il valore della locazione di memoria specificata in un registro. Questa istruzione potrebbe essere seguita da una qualsiasi istruzione che modifica il valore nel registro, eseguendo la parte "modify" del read-modify-write. Infine, l'ultima istruzione della sequenza è la seconda istruzione speciale, chiamata *store-conditional* (SC). Con questa istruzione si scrive il valore del registro nella locazione di memoria se e solo se nessun altro processore abbia scritto nella locazione stessa (o meglio nel blocco di cache corrispondente) dal momento in cui è stata eseguita l'istruzione LL. Quindi, se l'istruzione SC ha successo, vuol

dire che il valore della locazione di memoria è stato letto, modificato e scritto in memoria in maniera atomica. Se la store-conditional rileva che è avvenuta una scrittura nel blocco di cache che contiene la variabile, allora l'istruzione fallisce, non viene eseguita alcuna scrittura in memoria e l'operazione deve essere rieseguita, come mostrato nel codice 4.3. In questo modo non vengono generate inutili comunicazioni di invalidazione nel caso in cui il semaforo fosse stato già occupato.

Listato 4.3: Codice assembler dell'operazione di lock con l'utilizzo della coppia di istruzioni assembler LL-SC

```
1 lock:   LL      Rindsem, Rsemlock, DI
2         IF≠0    Rsemlock, lock
3         SC      Rindsem, #1, Resito /*setta se possibile
4                                       il semaforo
5                                       ad occupato*/
6         IF=0    Resito, lock /*se l'esito della SC e'
7                                negativo si ripete la lock*/
8         GOTO    Rret, EI
```

4.1.2 Indivisibilità a livello hardware-firmware

Garantire l'indivisibilità delle operazioni di lock e unlock in maniera primitiva a livello hardware-firmware impone che l'accesso alle locazioni (blocchi) di memoria comune riferite da queste due operazioni sia arbitrato opportunamente. In particolare si demanda il compito di assicurare l'indivisibilità di queste operazioni al meccanismo di arbitraggio dei moduli di memoria comune. Tale meccanismo può essere pensato in modo che blocchi l'accesso solo alle specifiche locazioni di memoria sulle quali è già in corso un'operazione lock/unlock. In realtà, essendo il tempo di esecuzione di queste operazioni relativamente breve, si sceglie di bloccare completamente l'accesso al modulo (o ai moduli) di memoria comune cui appartengono le locazioni di interesse. Il set istruzioni descritto in [23] prevede meccanismi più flessibili rispetto alle soluzioni descritte nella sezione 4.1.1, sotto forma di esplicite istruzioni per l'indivisibilità (`SET_INDIV`, `RESET_INDIV`) e di opzioni da specificare nelle istruzioni `LOAD` e `STORE` di accesso alla memoria. In particolare, le istruzioni per l'indivisibilità permettono di settare il valore di un apposito bit (*indiv*)

incluso nelle informazioni all'uscita del processore per gli accessi in memoria; attraverso questo bit si può indicare l'accesso esclusivo o meno alle locazioni interessate. Il compito di arbitrare i moduli di memoria condivisa spetta alle rispettive unità di interfaccia. Il trattamento del bit di indivisibilità da parte di un modulo di memoria, prevede che, essendo iniziata una sequenza indivisibile di accesso, vengano ascoltate tutte le richieste di accesso al modulo, ma che, finché la sequenza in corso non termina, vengano bufferizzate in una coda interna tutte quelle provenienti da un nodo diverso da quello che ha iniziato la sequenza in corso. Al termine di una sequenza indivisibile di accesso, l'unità di interfaccia del modulo di memoria estrae dalla coda un'eventuale richiesta pendente, e così via. La dimensione della coda è limitata superiormente dal numero di nodi, trattandosi sempre di interazioni a domanda-risposta per accessi in memoria.
Un'altra questione legata all'implementazione della lock è che se un processore, eseguendo la lock trova il semaforo occupato, non è pensabile che ritenti continuamente la lettura dello stato del semaforo fintanto che lo trova al valore libero. Ciò comporterebbe un inutile congestionamento dei moduli di memoria comune, ritardando anche l'esecuzione dell'unlock da parte del processore che aveva occupato il semaforo. Le soluzioni tipicamente adottate sono di due tipi:

- il processore bloccato sulla lock ritenta l'esecuzione della stessa a distanza di un *intervallo di tempo* costante, tipicamente calcolato in funzione del tempo medio di esecuzione della sezione critica compresa tra le due operazioni lock/unlock;

- il processore bloccato dalla lock indica il proprio nome/identificativo in una struttura associata al semaforo e si mette in attesa di una *comunicazione di sblocco* da parte del processore che eseguirà l'operazione di unlock.

Gli algoritmi che utilizzano la prima soluzione non possono assicurare la proprietà di *fairness*, cioè l'assenza di attesa infinita, che assume peró una probabilità di verificarsi molto bassa nel caso di sezioni indivisibili di piccola durata. Vediamo le due possibili implementazioni delle operazioni di lock e unlock

utilizzando la sequenza

$$set\_indiv; S; reset\_indiv;$$

per indicare l'esecuzione della sequenza indivisibile di accessi in memoria S.

Soluzione con intervallo di tempo di attesa

Lo pseudocodice mostrato nel listato 4.4 esegue lo stesso algoritmo presentato nel listato 4.1 integrato con le istruzioni per specificare l'accesso indivisibile ai moduli di memoria durante le operazioni sul semaforo e con quelle per specificare l'intervallo di tempo di attesa nel caso di semaforo occupato.

Listato 4.4: Pseudocodice delle operazioni lock/unlock nella soluzione con intervallo di tempo di attesa

```
1  lock (semlock) {
2          ok = false;
3          while (!ok) { set_indiv;
4                  if (semlock) {
5                          semlock = false; reset_indiv;
6                          ok = true;
7                  } else { reset_indiv;
8                          <attesa per un certo numero di
                              istruzioni "NOP">
9                  }
10         }
11 }
12
13 unlock (semlock) {
14         semlock = true;
15 }
```

Soluzione con coda di attesa

In questo caso si utilizza una coda FIFO (*queue*) di nomi/identificatori di processori, la cui dimensione è limitata superiormente dal numero di processori del sistema. Come mostrato nello pseudocodice del listato 4.5, nell'algoritmo di lock un processore che trova il semaforo occupato, inserisce in coda il proprio nome/identificatore mettendosi in attesa di una comunicazione di sblocco. Durante l'esecuzione dell'unlock, un generico processore, trovando almeno un processore in attesa, invia la comunicazione al primo in coda.

Listato 4.5: Pseudocodice delle operazioni lock/unlock nella soluzione con coda di attesa

```
1  lock (semlock) {
2          set_indiv;
3          if (semlock) {
4                  semlock = false; reset_indiv;
5          } else {
6                  put(myid, queue); reset_indiv;
7                  <attesa della comunicazione di sblocco>
8          }
9  }
10
11 unlock (semlock) {
12         set_indiv;
13         if (isEmpty(queue)) {
14                 semlock = true; reset_indiv;
15         } else {
16                 id = get(queue); reset_indiv;
17                 <invio comunicazione di sblocco al
                      processore "id">
18         }
19 }
```

4.2 Mutua esclusione e Cache Coherence: l'approccio algorithm-dependent

L'approccio algorithm-dependent è basato su una esplicita progettazione del supporto per la mutua esclusione che tenga conto del problema della cache coherence. Supponiamo infatti di avere a disposizione la possibilità di specificare, a livello di istruzioni assembler, alcune opzioni che permettono di scegliere come gestire alcune operazioni che normalmente sono demandate alla cache e all'unità W, che funge da controllore della coerenza. In particolare, come avviene nella maggior parte dei sistemi moderni, possiamo specificare nelle istruzioni di `STORE` le seguenti opzioni:

- `STORE` ..., write-through, ... : per modificare la singola parola indirizzata in memoria;

- `STORE` ..., non_modificare, ... : per indicare che nessuna modifica deve essere apportata in memoria;
- `STORE` ..., riscrivi_blocco, ... : per copiare l'intero blocco di cache in memoria.

Queste opzioni permettono di implementare le operazioni di lock/unlock in modo da non aver bisogno delle tecniche di cache coherence automatica. Questo perché le operazioni in mutua esclusione racchiuse tra le due operazioni saranno eseguite mantenendo la coerenza dei blocchi di cache coinvolti. Al fine di analizzare come integrare la cache coherence negli algoritmi di lock/unlock supponiamo che:

- le operazioni di locking siano implementate come mostrato nella sezione 4.1.2 nella soluzione con intervallo di tempo di attesa;
- il semaforo di lock, implementato come booleano, sia l'unica informazione modificabile del relativo blocco di cache; in questo modo, agli effetti della cache coherence è come se il semaforo fosse contenuto in un blocco dedicato solo ad esso.

Una generica sequenza di operazioni eseguite in mutua esclusione sarà quindi racchiusa tra le due operazioni di lock/unlock:

$$lock(semlock);$$

$$<operazioni\ eseguite\ in\ mutua\ esclusione>$$

$$unlock(semlock);$$

e la sua implementazione cache coherent si caratterizza come segue:

1. l'esecuzione con successo della lock provoca la *copia* del blocco contenente il semaforo in cache;
2. l'ultimo accesso in scrittura della lock al semaforo provvede alla *riscrittura* del blocco in memoria comune, ma *non* alla sua *deallocazione* dalla

cache; questa operazione è corretta ai fini della cache coherence in quanto il semaforo, come supposto, è l'unica parola modificabile del blocco. Se un altro processore fosse stato in attesa su una sequenza di accesso indivisibile per iniziare la lock, ora può eseguirla, copiando il blocco contenente il semaforo in cache; in questo modo il semaforo viene trovato occupato e il processore rimane in attesa finché non verrà eseguita l'operazione di unlock;

3. l'esecuzione dell'unlock provoca la *riscrittura* del blocco contenente il semaforo in memoria comune e la sua *deallocazione* dalla cache. Nel caso che un altro processore fosse bloccato sulla lock, viene sbloccato e può copiare i blocchi di cache in modo consistente.

Come abbiamo detto, c'è la possibilità di effettuare gli accessi in scrittura a blocchi o su singola parola; utilizzando l'opzione write-through, nei casi in cui è garantita la correttezza dal punto di vista della cache coherence, si può ridurre la latenza nell'accesso alle memorie remote. Negli algoritmi di lock/unlock è possibile per quanto riguarda la scrittura del valore del semaforo di lock: infatti, essendo l'unica parola modificata del blocco, la scrittura può semplicemente riguardare la singola parola invece dell'intero blocco.

4.3 Conclusioni

In questo capitolo abbiamo analizzato alcune tecniche di implementazione dei meccanismi per la mutua esclusione necessari nelle architetture multiprocessor per l'accesso a strutture condivise. Si tratta di meccanismi utilizzati sia nei modelli di programmazione a memoria condivisa che in quelli a scambio di messaggi nella progettazione delle primitive di comunicazione.
Per garantire l'indivisibilità, anche nell'esecuzione delle operazioni di lock/unlock, abbiamo visto che può essere considerato un approccio che fa uso di istruzioni assembler complesse o un approccio più flessibile che utilizza un set istruzioni semplice in cui si utilizzano dei meccanismi forniti dal livello hardware-firmware.

A partire da questo secondo approccio abbiamo esteso una possibile implementazione degli algoritmi di lock/unlock in modo da integrarvi, secondo un approccio algorithm-dependent, specifiche azioni rivolte a risolvere il problema della cache coherence senza far affidamento su tecniche automatiche, come quelle presentate nel capitolo 3.
Come vedremo nel prossimo capitolo, l'approccio algorithm-dependent permette di ottenere dei vantaggi, rispetto alle soluzioni automatiche, dal punto di vista delle prestazioni, soprattutto perché si tratta di un metodo capace di ridurre il numero di trasferimento di blocchi necessari per il mantenimento della cache coherence.

Capitolo 5

Modello dei Costi e Analisi delle Prestazioni

Le tecniche per risolvere il problema della cache coherence, presentate nei capitoli precedenti, si differenziano in base al livello in cui viene affrontato il problema: da una parte, le tecniche di cache coherence automatica fanno affidamento sui meccanismi forniti dal livello hardware-firmware per implementare i protocolli presentati nel capitolo 2; dall'altra parte invece, l'approccio algorithm-dependent tenta di minimizzare il numero di trasferimenti di blocchi di cache e di comunicazioni interprocessor integrando la risoluzione del problema nel supporto ai meccanismi di sincronizzazione per programmi paralleli, in particolare nelle operazioni di mutua esclusione.
In questo capitolo vogliamo fornire un modello dei costi che permetta di valutare in modo formale le prestazioni di entrambe le soluzioni; per far questo, viene mostrato nella sezione 5.1 come è possibile valutare la latenza di accesso in memoria per un'architettura multiprocessor, attraverso lo studio di un sistema cliente-servente secondo la teoria delle code.
Nella sezione 5.2 viene preso in esame un tipico scenario che si può verificare durante l'esecuzione di un programma parallelo per poter effettuare un confronto delle due soluzioni e in modo da poter avere un'idea dell'impatto sulle prestazioni che queste tecniche hanno nei confronti della latenza di accesso in memoria.

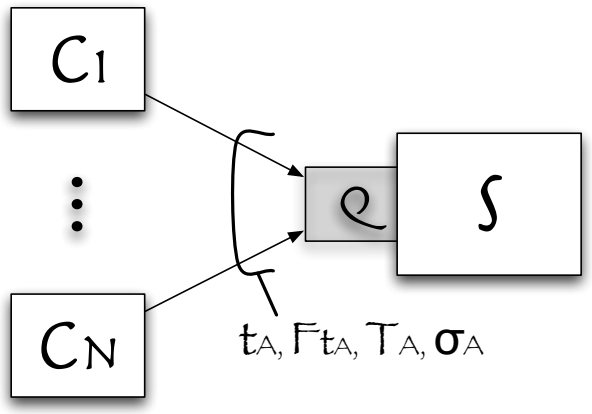


Figura 5.1: Sistema a coda

5.1 Valutazione della Latenza di Accesso in Memoria con la Teoria delle Code

Al fine di valutare le prestazioni di architetture multiprocessor, in particolare la latenza di accesso in memoria, occorre modellare il problema come un sistema *cliente-servente*.

5.1.1 Sistemi a coda

Un sistema a coda modella il comportamento di un servente S al quale uno o più clienti $C_1, ..., C_n$ si rivolgono attendendo in una fila di attesa Q di ricevere il servizio erogato, come mostrato in figura 5.1.

Il sistema è definito attraverso:

- la *disciplina del servizio*;
- la *dimensione della coda*;
- la *distribuzione probabilistica della variabile aleatoria tempo di servizio* (t_S)

  $$F_{t_S}(t) = Pr(t_S \leq t)$$

  con valor medio T_S e varianza σ_S; la *frequenza dei servizi* è indicata con $\mu = 1/T_S$;

- la *distribuzione probabilistica della variabile aleatoria tempo di interarrivo* (t_A), cioè l'intervallo di tempo tra due arrivi consecutivi alla coda

$$F_{t_A}(t) = Pr(t_A \leq t)$$

con valor medio T_A e varianza σ_A; la *frequenza degli interarrivi* è indicata con $\lambda = 1/T_A$.

Per quanto riguarda le prime due caratteristiche, una valida approssimazione, spesso utilizzata, è quella che prevede una coda con disciplina FIFO di dimensione infinita. Viene definito inoltre il *fattore di utilizzazione della coda* come:

$$\rho = \frac{T_S}{T_A} = \frac{\lambda}{\mu}$$

utilizzato per dare una misura del grado di congestione delle richieste al servente.
Conoscendo la distribuzione probabilistica della variabile aleatoria tempo di servizio e della variabile aleatoria tempo di interarrivo, le grandezze di interesse per valutare in dettaglio le prestazioni di un sistema a coda sono:

- la *lunghezza media della coda* (L_q) data dal valor medio del numero di richieste presenti nella fila di attesa;

- il *numero medio di richieste nel sistema* (N_q) che, rispetto a L_q, include anche la richiesta (o le richieste) in fase di servizio;

- il *tempo medio di attesa in coda* (W_Q) riferito al tempo di permanenza delle richieste nella fila di attesa;

- il *tempo medio di attesa nel sistema* (R_Q) riferito al tempo globale che la richiesta trascorre nel sistema a coda, inclusa la fase di servizio:

$$R_Q = W_Q + T_S$$

Queste grandezze sono in relazione tra loro secondo la *legge di Little*, come spiegato in [16], secondo cui valgono le seguenti equazioni:

$$L_q = \lambda W_Q \quad N_q = \lambda R_Q$$

Secondo la notazione introdotta da Kendall (distribuzione degli interarrivi / distribuzione dei servizi / numero di serventi), i casi notevoli di sistema a coda, per i quali è possibile esprimere analiticamente le grandezze di interesse sono i seguenti:

- coda M/M/1: la coda è infinita con disciplina FIFO, con distribuzione degli interarrivi e dei servizi di tipo esponenziale negativa;
- coda M/G/1: indica, a differenza della precedente, una qualunque distribuzione del tempo di servizio;

Un caso particolare della coda M/G/1 è la coda con distribuzione dei tempi di servizio *deterministica*, indicata con M/D/1, in cui $\sigma_S = 0$, cioè il tempo di servizio è costantemente uguale a T_S. Per questo tipo di coda si hanno i seguenti risultati per le grandezze N_Q e R_Q:

$$N_Q = \left(\frac{\rho}{1-\rho}\right)\left(1 - \frac{\rho}{2}\right)$$

$$R_Q = \left(\frac{T_S}{1-\rho}\right)\left(1 - \frac{\rho}{2}\right)$$

Sistemi a coda e computazioni cliente-servente a domanda-risposta

Analizziamo ora come applicare i risultati della teoria delle code ad un sistema in cui i clienti e il servente cooperano a domanda-risposta, come rappresentato in figura 5.2.

Il funzionamento di un generico cliente C_i e quello del servente S sono descritti nei listati 5.1 e 5.2 rispettivamente.

Listato 5.1: Comportamento del generico cliente in un sistema cliente-servente a domanda-risposta

```
1 C_i::     ...
2           while (true) {
3                   < invia x a S >;
4                   < ricevi y da S >;
5                   x = G(..., y, ...);
6           }
```

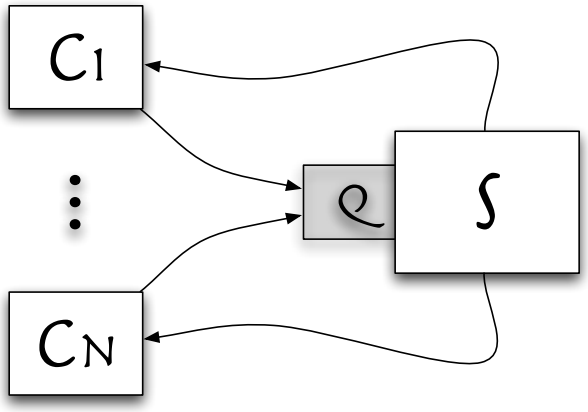


Figura 5.2: Sistema a coda con cliente-servente a domanda-risposta

Listato 5.2: Comportamento del servente in un sistema cliente-servente a domanda-risposta

```
1 S::       ...
2           while (true) {
3                   < ricevi x da un C ∈ C_1,...,C_n >;
4                   y = F(...,x,...);
5                   < invia y a C >;
6           }
```

Poiché tutti i clienti hanno un comportamento analogo, e supponendo che la funzione G abbia tempi di elaborazione sensibilmente variabili con il valore di y, si può assumere che la distribuzione del tempo di interarrivo a S sia esponenziale con media T_A.

Il comportamento del sistema è descritto dal seguente sistema di equazioni:

$$\begin{cases} T_{cl} = T_G + R_Q \\ R_Q = R_Q(\rho, T_s, L_S, \sigma_s) \\ \rho = \dfrac{T_s}{T_A} \\ \rho < 1 \\ T_A = \dfrac{T_{cl}}{n} \end{cases}$$

dove: il tempo di servizio T_{cl} rappresenta il tempo medio per eseguire un'iterazione del ciclo di C_i e T_G è il tempo medio di elaborazione della funzione G. Il vincolo $\rho < 1$ viene posto in modo che il servente non rappresenti un

"collo di bottiglia" per il sistema.
La formula di R_Q deve essere sviluppata tenendo conto della distribuzione del servente. Inoltre, nel caso di servente parallelo, le formule vanno modificate per tenere conto correttamente della differenza tra tempo di servizio (T_S) e latenza (L_S); in particolare, il tempo di risposta va valutato come:

$$R_Q = W_Q + L_S$$

5.1.2 Valutazione delle prestazioni

La teoria appena esposta permette di valutare le prestazioni di architetture multiprocessor. In particolare, siamo in grado di calcolare la latenza di accesso in memoria condivisa.
Per far questo è necessario modellare il problema con un sistema cliente-servente come segue: sono presenti N serventi distinti ognuno in media con p clienti, dove p è il numero medio di nodi che condividono una generica memoria.

- In un'architettura SMP, gli accessi si ripartiscono uniformemente nei confronti degli m macro-moduli, si può quindi stimare p come la media della distribuzione binomiale $P(k) = \binom{N}{k}(\frac{1}{m})^k(1-\frac{1}{m})^{n-k}$, ottenendo quindi $p = \dfrac{N}{m}$.
- In un'architettura NUMA, il valore di p dipende dallo specifico programma parallelo in quanto gli accessi ai macro-moduli della memoria condivisa non si ripartiscono uniformemente; di regola comunque p è sostanzialmente minore di N, in quanto i programmi paralleli sono caratterizzati da una certa località delle comunicazioni.

A partire dal sistema presentato nella sezione 5.1.1, possiamo utilizzare il parametro chiave della latenza di accesso in memoria condivisa a vuoto (o base) per rappresentare il tempo di servizio del servente. In [23] è riportata la valutazione della latenza base in un'architettura con rete di interconnessione logaritmica con controllo del flusso wormhole. Questa valutazione permette di stimare la latenza per la lettura, che si dimostra equivalente a quella per la scrittura, di un blocco di cache.

Indicando con T_p il tempo medio che il generico processore spende tra due accessi consecutivi alla memoria remota, il tempo di accesso in memoria remota in presenza di conflitti è dato dal tempo di risposta R_Q che si ricava dalla soluzione del seguente sistema di equazioni:

$$\begin{cases} T = T_p + R_q \\ R_Q = W_Q(\rho, T_s, \sigma_s) + t_{a_0} \\ \rho = \dfrac{T_s}{T_A} \\ T_s = t_{a_0} \\ T_A = \dfrac{T}{p} \\ \rho < 1 \end{cases}$$

È necessario tener conto che in una architettura NUMA è significativo distinguere tra la latenza degli accessi in memoria remota (R_{Q-rem}) e latenza degli accessi in memoria locale (R_{Q-loc}). Assumendo che, in caso di conflitto, tra accessi locali e remoti venga data priorità agli accessi locali, si può stimare:

$$R_{Q-loc} = t_{a_0-loc}(1 + \rho)$$

dove ρ è il fattore di utilizzazione del servente. Quindi, lo studio del sistema cliente-servente è comunque fondamentale sia per determinare R_{Q-rem} che R_{Q-loc}.
Per valutare W_Q, una buona approssimazione è data dalla coda *M/D/1*, in quanto il tempo di servizio di ogni macro-modulo di memoria è costante e le richieste da parte di ogni processore ad una memoria sono distribuite in maniera sostanzialmente casuale con media T_p; si ha quindi:

$$W_Q = T_s \frac{\rho}{2(1-\rho)} = ta_0 \frac{\rho}{2(1-\rho)}$$

Sostituendo, si può studiare il sistema ricavando R_Q in funzione delle varie grandezze in gioco, come:

- p numero medio di nodi che condividono una generica memoria;
- T_p tempo medio di elaborazione per nodo tra due accessi consecutivi alla memoria;

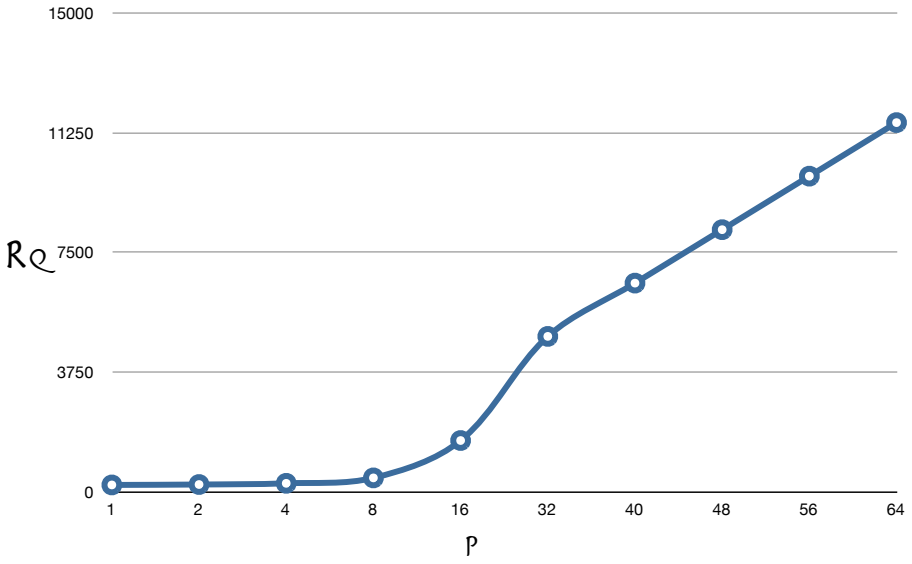


Figura 5.3: Valutazione del valore del tempo di accesso in memoria remota in presenza di conflitti al variare del numero medio di nodi che condividono una generica memoria

I grafici mostrati nelle figure 5.3 e 5.4 mostrano rispettivamente il valore di R_Q al variare di p e di T_p, in cui i valori presi in considerazione per gli altri parametri sono fissati come segue:

- nel grafico in cui varia il valore di p, il valore di T_p è fissato al valore 2000τ;
- nel grafico in viene fissato il valore di p, questo assume il valore di 4;
- il tempo di accesso alla memoria a vuoto è ricavato come descritto in [23] per un'architettura con 64 nodi, con rete di interconnessione logaritmica e controllo del flusso wormhole, in cui il ciclo di clock delle unità di memoria τ_M è pari a 10τ; si ottiene quindi un tempo di servizio del servente t_{a_0} pari a 210τ.

Nel primo caso, all'aumentare del valore di p, che ricordiamo rappresenta il numero medio di nodi che condividono un modulo di memoria, aumentano anche i conflitti sulle unità di interfaccia dei moduli di memoria e quindi il

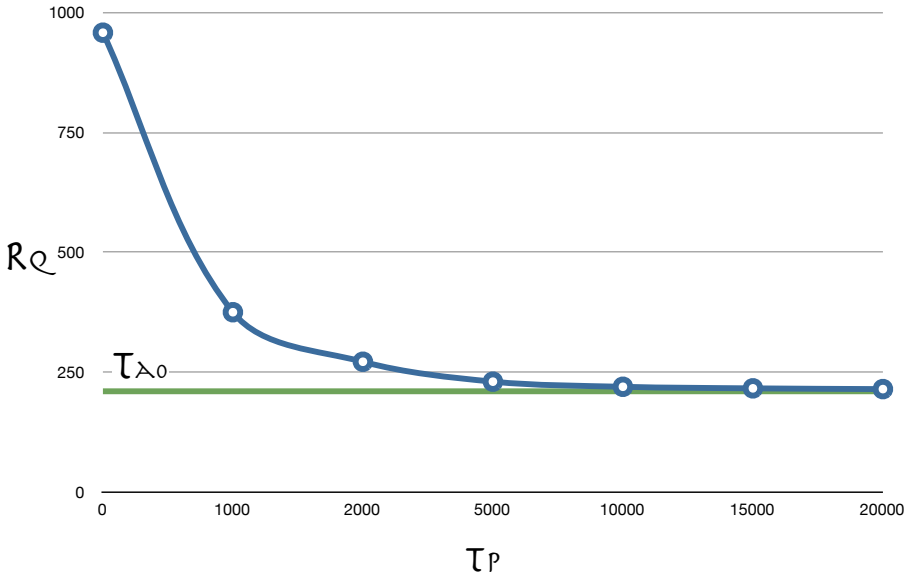


Figura 5.4: Valutazione del valore del tempo di accesso in memoria remota in presenza di conflitti al variare del tempo medio di elaborazione per nodo tra due accessi consecutivi alla memoria

valore di R_Q aumenta.
Nel secondo caso, aumentando il valore di T_p, l'intervallo di tempo che intercorre tra due accessi consecutivi alla memoria da parte di un processo aumenta, quindi il tempo di interarrivo al servente aumenta e come conseguenza l'impatto dei conflitti sulla memoria condivisa tende a diminuire ed a stabilizzarsi al valore di t_{a_0}.
La figura 5.5 mostra il grafico relativo alla famiglia di curve relative alla variazione del valore assunto da R_Q al variare di p e di T_p.

5.2 Confronto tra Tecniche Automatiche e Approccio Algorithm-Dependent

A questo punto è necessario capire come modellare le tecniche di cache coherence in modo da ottenere un confronto sulle prestazioni di un sistema che fa uso di cache coherence automatica rispetto ad un sistema che utilizza un approccio algorithm-dependent.

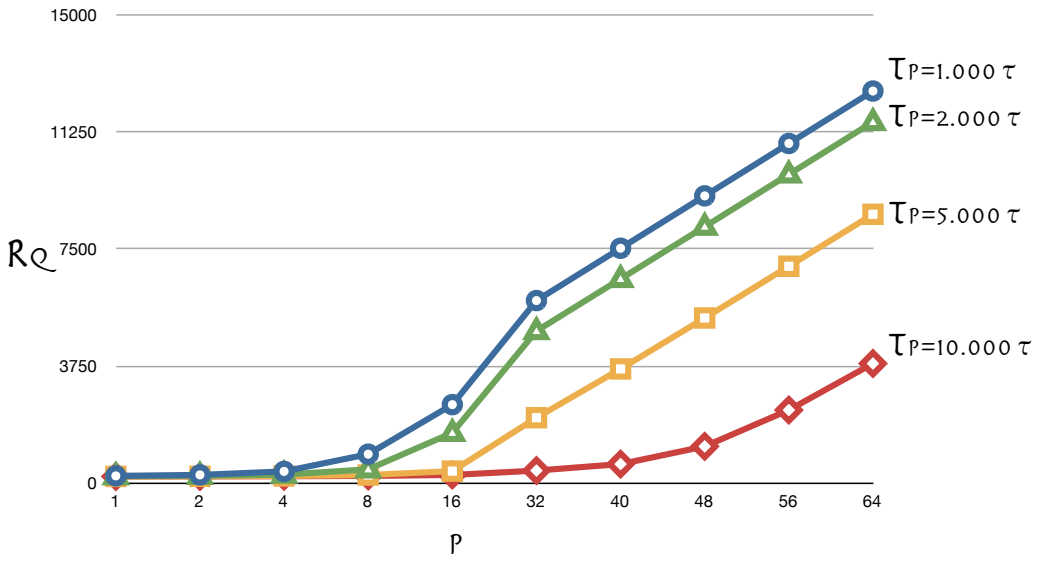


Figura 5.5: Valutazione del valore del tempo di accesso in memoria remota in presenza di conflitti al variare del numero medio di nodi che condividono una generica memoria e del tempo medio di elaborazione per nodo tra due accessi consecutivi alla memoria

Supponiamo di avere a disposizione un supporto a meccanismi di concorrenza come quello descritto nella sezione 4.1.2, in seguito utilizzeremo due parametri: x per indicare la probabilità di effettuare la lock su un semaforo già occupato e il parametro r per indicare il numero di ripetizioni del tentativo di acquisizione del semaforo di lock, che nel caso di implementazione con coda di attesa assume il valore 1; supponiamo inoltre che per il semaforo di lock sia allocato un intero blocco di cache, dove il semaforo è l'unica informazione modificabile mentre le altre sono usate in sola lettura.
Analizzeremo quindi i costi delle operazioni di mutua esclusione in due scenari diversi: il primo è quello che, in un modello di programmazione a scambio di messaggi, si presenta nell'utilizzo di un canale simmetrico; il secondo è in realtà un insieme di scenari che ci permetteranno di valutare tutte le situazioni che tipicamente ritroviamo in una computazione parallela.
In entrambi gli scenari effettueremo un confronto tra le tecniche di cache coherence automatica basate su directory, per semplicità di tipo memory-based,

e l'approccio algorithm-dependent.
Per far questo, in seguito utilizzeremo i seguenti simboli:

- $T_{read-rem}$ per indicare la latenza di accesso remoto ad un blocco di cache in lettura;
- $T_{read-rem-1}$ per indicare la latenza di accesso remoto ad una parola in lettura;
- $T_{write-rem}$ per indicare la latenza di accesso remoto ad un blocco di cache in scrittura;
- $T_{write-rem-1}$ per indicare la latenza di accesso remoto ad una parola in scrittura;
- $T_{dir-rem}$ per indicare la latenza di accesso remoto alle informazioni di directory.

I primi quattro valori possono essere approssimati al valore R_Q come detto nella sezione precedente; inoltre è possibile distinguere, nel caso di un'architettura NUMA, i casi in cui gli accessi vengono effettuati alla memoria del nodo, quindi in locale, indicando questi valori con la sigla *loc* e approssimandoli quindi al valore R_{Q-loc}.
È stato necessario distinguere il caso in cui l'accesso riguarda le informazioni di directory: in generale infatti, il tempo di accesso alla directory è maggiore rispetto a quello di accesso ad una memoria, e questo si verifica per diversi motivi.
L'implementazione della directory può avere conseguenze sul numero di accessi che una generica unità W_i deve effettuare per mantenere aggiornata la directory stessa a seguito di una richiesta da parte di un processore. In particolare le soluzioni si muovono tra i due seguenti approcci: da una parte un'implementazione "intelligente" della directory, che ad esempio permette di effettuare operazioni di tipo read-modify-write, comporta una complessità di progettazione maggiore a discapito di un tempo di accesso alto; dall'altra parte un'implementazione più semplice della directory, usata solo in lettura o scrittura, comporta un numero maggiore di accessi.

Inoltre, non è da trascurare l'influenza che il numero di nodi dell'architettura e la scelta del protocollo di cache coherence, in particolare il numero di stati, hanno sulla complessità di progettazione della directory.
Anche nel caso dell'accesso alle informazioni di directory possiamo fare una distinzione nel caso in cui la directory è presente nel nodo locale, utilizzando in questo caso il simbolo $T_{dir-loc}$. I valori $T_{dir-rem}$ e $T_{dir-loc}$ saranno quindi approssimati rispettivamente ai valori di R_{Q-dir} e $R_{Q-dir-loc}$ che, rispetto ai valori R_Q e R_{Q-loc}, tengono conto del fatto che l'accesso viene effettuato sulla directory e non al modulo di memoria.

5.2.1 Studio delle comunicazioni simmetriche

Consideriamo uno scenario in cui il processo A e il processo B siano contemporaneamente in esecuzione su due processori distinti, P_i (con cache C_i) e P_j (con cache C_j), e che entrambi provino ad eseguire l'operazione di lock sulla stessa struttura condivisa, cioè entrambi competono per trasferire nelle rispettive cache il blocco contenente il semaforo di lock. Questa situazione è la stessa che si può verificare in un ambiente a scambio di messaggi, in cui il processo mittente (A) e il processo destinatario (B) provano a fare accesso al descrittore di canale CH nello stesso istante.
Supponiamo, per semplicità di ragionamento, che il blocco contenente il semaforo di lock inizialmente non sia presente in nessuna cache. L'utilizzo di un punto di centralizzazione o di un meccanismo come quello del bit di indivisibilità impedisce che entrambi i processori riescano a copiare in cache il blocco contenente il semaforo di lock.

Studio dei protocolli directory-based

Analizziamo ora in dettaglio quali sono le comunicazioni richieste dal protocollo directory-based: a partire dagli algoritmi di lock/unlock consideriamo quali istruzioni richiedono le comunicazioni interprocessor e i trasferimenti dei dati dettati dal protocollo.

- Supponiamo che sia P_i il processo che riesce a copiare per primo il blocco in cache, in seguito ad un fault per l'accesso in lettura al blocco; la

richiesta in memoria di P_j nel frattempo sarà bloccata in qualche punto della struttura di interconnessione o in qualche interfaccia verso moduli di memoria condivisa.

```
P_i ::    lock (semlock) {
                ...
                set_indiv;
            ⟹  if (semlock) {
                ...
}
```

Le operazioni richieste dal meccanismo di cache coherence sono quelle relative al caso di trattamento di fault a seguito di una lettura nel caso in cui il bit *dirty* della directory vale *FALSE*; sappiamo inoltre che in questo caso il nodo *dirty* coincide con il nodo *home* in quanto abbiamo supposto che la copia aggiornata del blocco sia in memoria principale, in questo modo con un'unica richiesta si effettua l'accesso alla directory e alla memoria.

$$T_{dir+read-rem} \sim \begin{cases} W_i \longrightarrow W_{home} \\ W_{home} \longleftrightarrow Directory \\ W_{home} \longleftrightarrow M_{home} \\ M_{home} \longrightarrow C_i \quad \texttt{(DATI)} \end{cases}$$

- A questo punto il blocco è nella cache C_i in stato *shared* e la successiva scrittura nel blocco

```
P_i ::    lock (semlock) {
                ...
            ⟹  semlock = false;
                ...
}
```

causa la gestione di un fault dovuto a scrittura nel caso in cui il bit *dirty* vale *FALSE*.

$$T_{dir-rem} \sim \begin{cases} W_i \longrightarrow W_{home} \\ W_{home} \longleftrightarrow Directory \\ W_{home} \longrightarrow W_i \end{cases}$$

- Quando P_i termina la *lock*, P_j riesce a completare la sua richiesta al nodo *home*, in particolare l'unità W_{home} deve gestire il fault a seguito di una richiesta di lettura,

```
P_i ::    lock (semlock) {
                ...
                while (!ok) {
                        ...
                        reset_indiv;
                        ok = true;
          ⟹    }
}
P_j ::    lock (semlock) {
                ...
                set_indiv;
          ⟹    if (semlock) {
                ...
}
```

in questo caso con il bit *dirty* che vale *TRUE*: viene inviata una richiesta di trasferimento del blocco contenente il semaforo di lock al nodo i, il quale invia i dati sia al nodo *home* che al nodo j e modifica lo stato del proprio blocco a *shared*.

$$T_{dir-rem} \sim \begin{cases} W_j \longrightarrow W_{home} \\ W_{home} \longleftrightarrow Directory \\ W_{home} \longrightarrow W_j \end{cases}$$

$$T_{read-rem} \sim \begin{cases} W_j \longrightarrow W_i \\ C_i \longrightarrow C_j \quad \texttt{(DATI)} \\ C_i \longrightarrow M_{home} \quad \texttt{(DATI+aggiornamento)} \end{cases}$$

- Fin quando P_i non eseguirà la *unlock* il processo P_j è bloccato; quando P_i termina l'esecuzione delle operazioni in mutua esclusione genererà un fault a seguito della richiesta di scrittura del valore del semaforo di lock, in quanto il blocco è in stato *shared*;

```
P_i ::    unlock (semlock) {
             ⟹    semlock = true;
}
```

il bit *dirty* vale *FALSE* e, trattandosi di un aggiornamento del blocco, l'unica cosa da fare è inviare l'invalidazione al nodo j e alla ricezione dell'*ack* il blocco passa di nuovo allo stato *modified*.

$$T_{dir-rem} \sim \begin{cases} W_i \longrightarrow W_{home} \\ W_{home} \longleftrightarrow Directory \\ W_{home} \longrightarrow W_i \end{cases}$$

$$T_{inv} \sim T_{write-rem-1} \sim \begin{cases} W_i \longrightarrow W_j \texttt{(inv)} \\ W_j \longrightarrow C_j \\ W_j \longrightarrow W_i \texttt{(ack)} \end{cases}$$

- A questo punto P_j, nel richiedere la lettura del valore del semaforo di lock, genererà un fault a seguito della lettura;

```
P_j ::    lock (semlock) {
                ...
                set_indiv;
            ⟹  if (semlock) {
                ...
}
```

questo comporterà l'invio del blocco da parte del nodo i, in quanto il bit *dirty* vale *TRUE*; ora entrambi i nodi i e j hanno il blocco in stato *shared* e P_j può proseguire con l'esecuzione della lock.

$$T_{dir-rem} \sim \begin{cases} W_j \longrightarrow W_{home} \\ W_{home} \longleftrightarrow Directory \\ W_{home} \longrightarrow W_j \end{cases}$$

$$T_{read-rem} \sim \begin{cases} W_j \longrightarrow W_i \\ C_i \longrightarrow C_j \quad \texttt{(DATI)} \\ C_i \longrightarrow M_{home} \quad \texttt{(DATI+aggiornamento)} \end{cases}$$

- La modifica del valore del semaforo di lock

```
P_j ::    lock (semlock) {
                ...
            ⟹  semlock = false;
                ...
}
```

causa l'aggiornamento del blocco e quindi l'invalidazione del blocco in C_i.

$$T_{dir-rem} \sim \begin{cases} W_j \longrightarrow W_{home} \\ W_{home} \longleftrightarrow Directory \\ W_{home} \longrightarrow W_j \end{cases}$$

$$T_{inv} \sim T_{write-rem-1} \sim \begin{cases} W_j \longrightarrow W_i\texttt{(inv)} \\ W_i \longrightarrow C_i \\ W_i \longrightarrow W_j\texttt{(ack)} \end{cases}$$

- L'esecuzione dell'*unlock* da parte di P_j

```
P_j ::    unlock (semlock) {
             ⟹    semlock = true;
}
```

non causa ulteriori cambiamenti di stato del blocco in quanto si trova già in stato *modified* in C_j e il processo può quindi modificare il valore del semaforo di lock.

Possiamo quindi determinare quantitativamente i parametri del modello dei costi delle operazioni di mutua esclusioni nel caso di utilizzo di tecniche di cache coherence automatica basate su directory di tipo memory-based. Nel caso in cui il nodo dirty coincide con il nodo home abbiamo visto che è possibile effettuare un'unica richiesta; in questo caso il valore $T_{dir+read}$ sarà approssimato al valore $R_{Q-mem+dir}$, distinguendo sempre gli accessi remoti da quelli locali. Considerando la probabilità x di trovare il semaforo di lock occupato, si possono calcolare i valori di $T_{lock-dir}$ e $T_{unlock-dir}$ nei seguenti due casi:

1. architettura NUMA in cui i nodi coinvolti sono due e il processo che esegue le operazioni di lock/unlock è allocato sul nodo home:

$$\begin{aligned} T_{lock-dir} &\sim (1-x)(T_{dir+read-loc} + T_{dir-loc}) \\ &+ x(T_{dir-loc} + T_{write-rem-1} \\ &+ (1+r)(T_{dir-loc} + T_{read-rem})) \\ &\sim (1-x)(R_{Q-mem+dir-loc} + R_{Q-dir-loc}) \\ &+ x(R_{Q-dir-loc} + R_{Q-1} + (1+r)(R_{Q-dir-loc} + R_Q)) \end{aligned}$$

$$\begin{aligned} T_{unlock-dir} &\sim (1-x)(T_{dir-loc} + T_{write-rem-1}) \\ &\sim (1-x)(R_{Q-dir-loc} + R_{Q-1}) \end{aligned}$$

2. architettura NUMA in cui i nodi coinvolti sono tre, cioè la directory è allocata su un nodo che non coincide con i due coinvolti nelle operazioni di lock/unlock o architettura NUMA in cui i nodi coinvolti sono due e il processo che esegue le operazioni di lock/unlock è allocato su un nodo che non coincide con il nodo home o architettura SMP:

$$\begin{aligned} T_{lock-dir} &\sim (1-x)(T_{dir+read-rem} + T_{dir-rem}) \\ &+ x(T_{dir-rem} + T_{write-rem-1} \\ &+ (1+r)(T_{dir+read-rem})) \\ &\sim (1-x)(R_{Q-mem+dir} + R_{Q-dir}) \\ &+ x(R_{Q-dir} + R_{Q-1} + (1+r)R_{Q-mem+dir}) \end{aligned}$$

$$\begin{aligned} T_{unlock-dir} &\sim (1-x)(T_{dir-rem} + T_{write-rem-1}) \\ &\sim (1-x)(R_{Q-dir} + R_{Q-1}) \end{aligned}$$

Vediamo i valori assunti da queste grandezze nei due casi, al variare del numero di ripetizioni del tentativo di acquisizione del semaforo e al variare della probabilità di trovarlo occupato.
Il grafico mostrato in figura 5.6 mostra una famiglia di curve che rappresentano l'andamento del valore $T_{lock-dir}$ per il secondo caso considerato, al variare della probabilità x di trovarlo occupato; ad una diversa curva corrisponde un valore diverso r di tentativi di acquisizione del semaforo. Come ci si poteva aspettare all'aumentare del numero di tentativi il tempo richiesto per l'esecuzione della lock aumenta.
I grafici mostrati nelle figure 5.7(a) e 5.7(b) mostrano rispettivamente il valore assunto da $T_{lock-dir}$ e $T_{unlock-dir}$ nei due casi sopraelencati al variare di x, considerando un'architettura NUMA. Il secondo caso è chiaramente il più sfavorevole in quanto tutti gli accessi effettuati sono remoti, inoltre viene mostrato anche il caso in cui il valore assunto da p è pari a 3 invece che 2 come gli altri due casi, contribuendo ad aumentare il tempo di accesso in memoria e alla directory sotto carico nel caso in cui i nodi coinvolti sono tre.

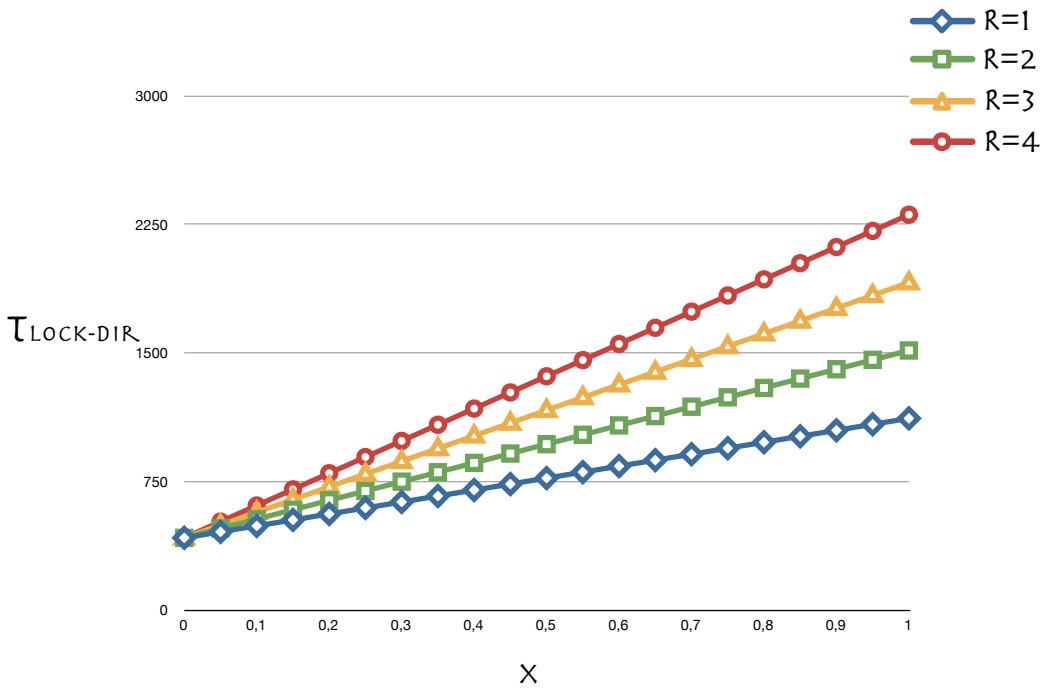
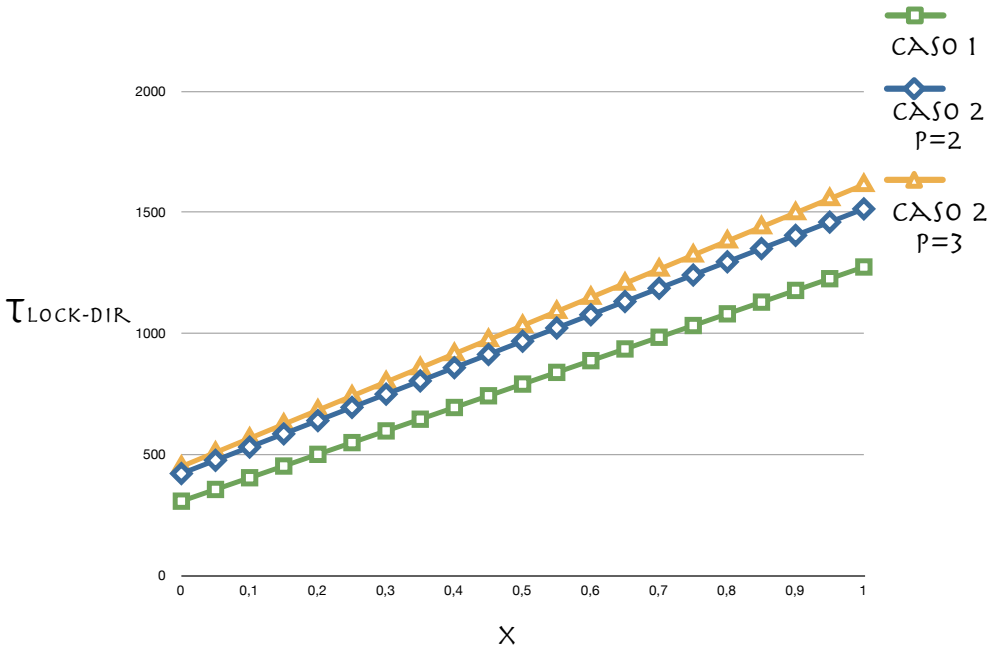
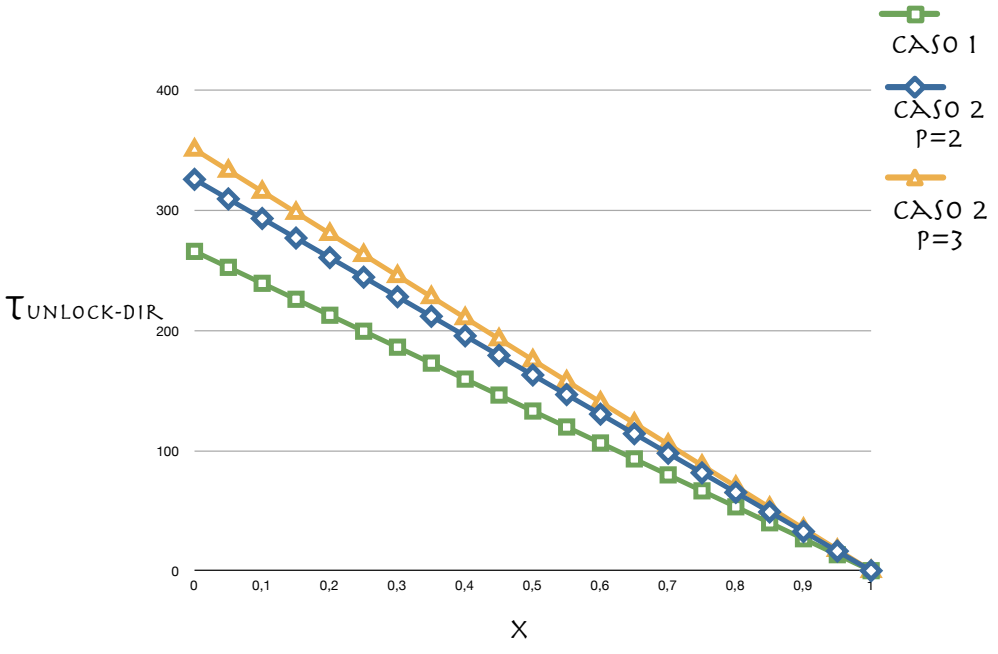


Figura 5.6: Valutazione della latenza della lock directory-based del caso 2 al variare del numero di ripetizioni del tentativo di acquisizione del semaforo e al variare della probabilità di trovarlo occupato



(a) lock



(b) unlock

Figura 5.7: Valutazione della latenza di lock e unlock directory-based al variare della probabilità di trovare il semaforo occupato

Studio dell'approccio algorithm-dependent

Vediamo ora cosa avviene nel caso di un approccio algorithm-dependent, supponendo di trovarci nella stessa situazione utilizzata per analizzare il caso precedente.

- l'esecuzione con successo della *lock* da parte di P_i provoca la copia del blocco in cache.

```
P_i ::    lock (semlock) {
                  ...
                  set_indiv;
                  if (semlock) { ⟹ T_read−rem
                  ...
}
```

- l'ultimo accesso in scrittura della *lock* al semaforo eseguita da P_i provvede alla riscrittura del blocco in memoria comune, ma non alla sua deallocazione dalla cache.

```
P_i ::    lock (semlock) {
                  ...
                  semlock = false; ⟹ T_write−rem−1
                  ...
}
```

  Se P_j fosse stato in attesa su una sequenza di accesso indivisibile per iniziare la *lock*, ora può eseguirla, copiando il blocco in cache;

```
P_j ::    lock (semlock) {
                  ...
                  set_indiv;
                  if (semlock) { ⟹ T_read−rem
                  ...
}
```

  così facendo trova il semaforo rosso e rimane in attesa attiva finché non verrà eseguita la *unlock*;

- l'esecuzione dell'*unlock* da parte di P_i provoca la riscrittura del primo blocco in memoria comune e la sua deallocazione dalla cache;

```
P_i ::    unlock (semlock) {
                  semlock = true; ⟹ T_write−rem−1
}
```

  essendo P_j bloccato sulla *lock*, viene sbloccato e ora può copiare i blocchi di cache in modo consistente.

Come nel caso precedente, possiamo valutare i parametri del modello dei costi delle operazioni di mutua esclusioni nel caso di utilizzo di un approccio di tipo algorithm-dependent; considerando la probabilità x di trovare il semaforo di lock occupato si possono calcolare i valori di $T_{lock-ad}$ e $T_{unlock-ad}$ come prima nei seguenti due casi:

1. architettura NUMA in cui i nodi coinvolti sono due e il semaforo è allocato sullo stesso nodo del processo che esegue le operazioni di lock/unlock:

$$\begin{aligned} T_{lock-ad} &\sim T_{read-loc} + T_{write-loc-1} \\ &+ \quad x(1+r)T_{read-loc} \\ &\sim R_{Q-loc} + R_{Q-loc-1} + x(1+r)R_{Q-loc} \\ T_{unlock-ad} &\sim T_{write-loc-1} \sim R_{Q-loc-1} \end{aligned}$$

2. architettura NUMA in cui i nodi coinvolti sono tre, cioè il semaforo è allocato in un nodo che non coincide con i due coinvolti nelle operazioni di lock/unlock o architettura NUMA in cui i nodi coinvolti sono due e il processo che esegue le operazioni di lock/unlock non è allocato sul nodo in cui è allocato anche il semaforo o architettura SMP:

$$\begin{aligned} T_{lock-ad} &\sim T_{read-rem} + T_{write-rem-1} \\ &+ \quad x(1+r)T_{read-rem} \\ &\sim R_Q + R_{Q-1} + x(1+r)R_Q \\ T_{unlock-ad} &\sim T_{write-rem-1} \sim R_{Q-1} \end{aligned}$$

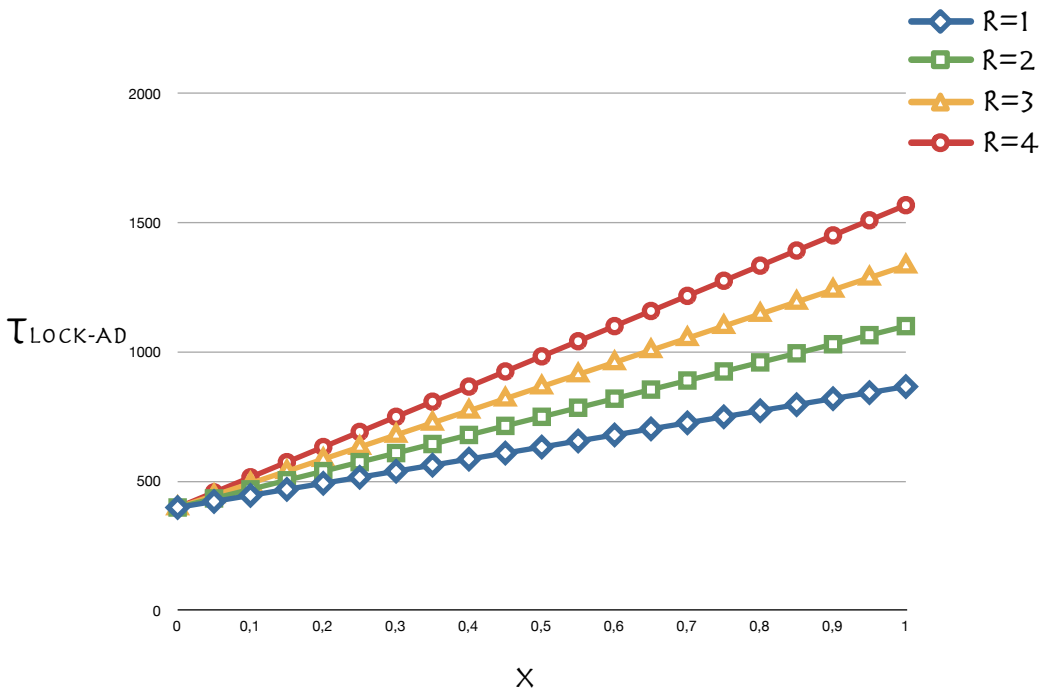
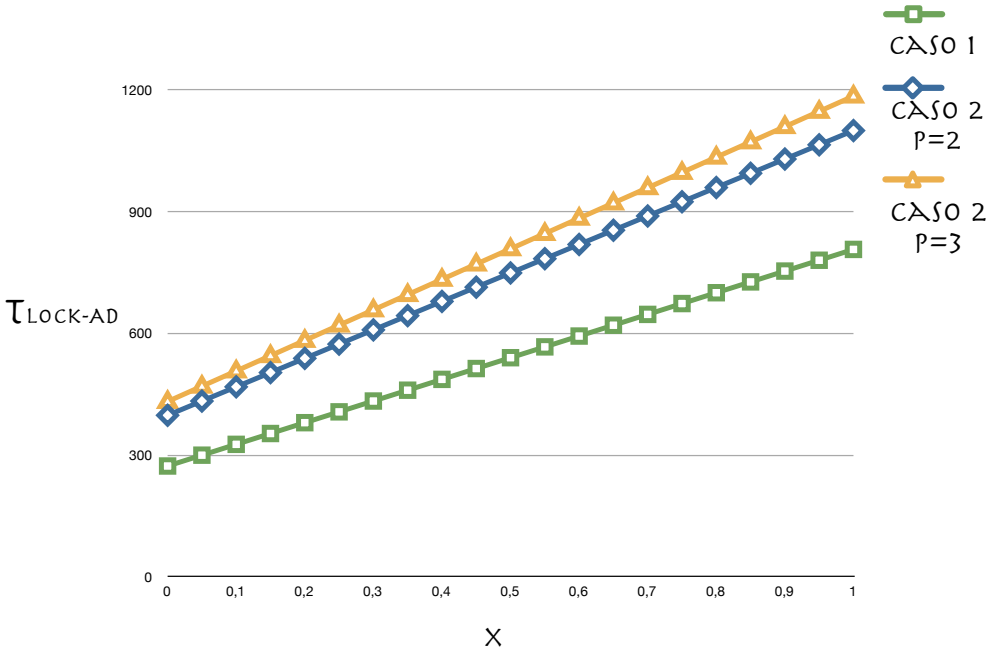
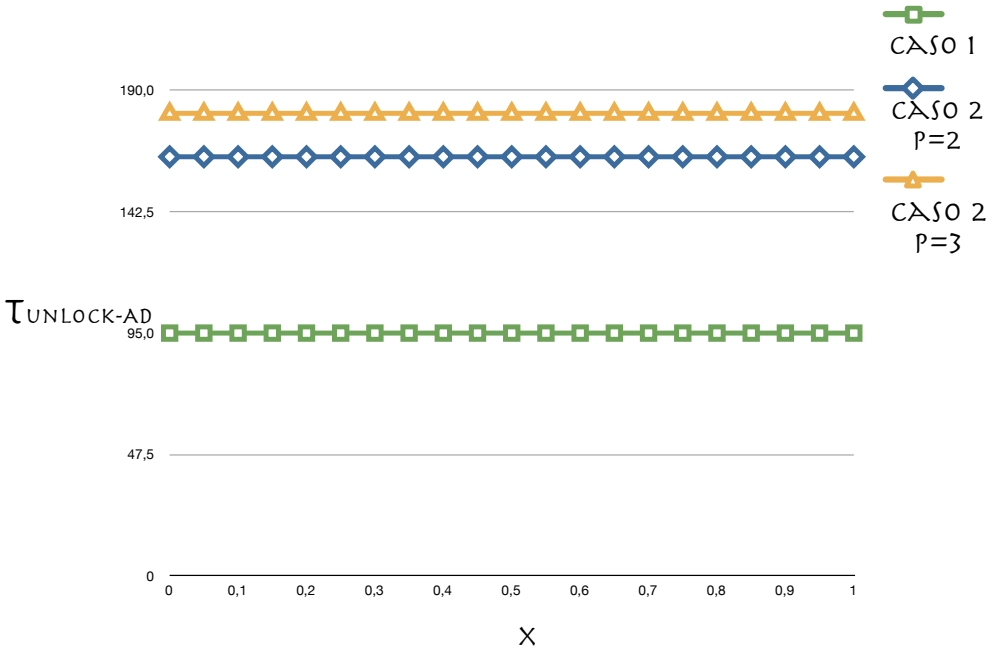


Figura 5.8: Valutazione della latenza della lock algorithm-dependent del caso 2 al variare del numero di ripetizioni del tentativo di acquisizione del semaforo e al variare della probabilità di trovarlo occupato



(a) lock



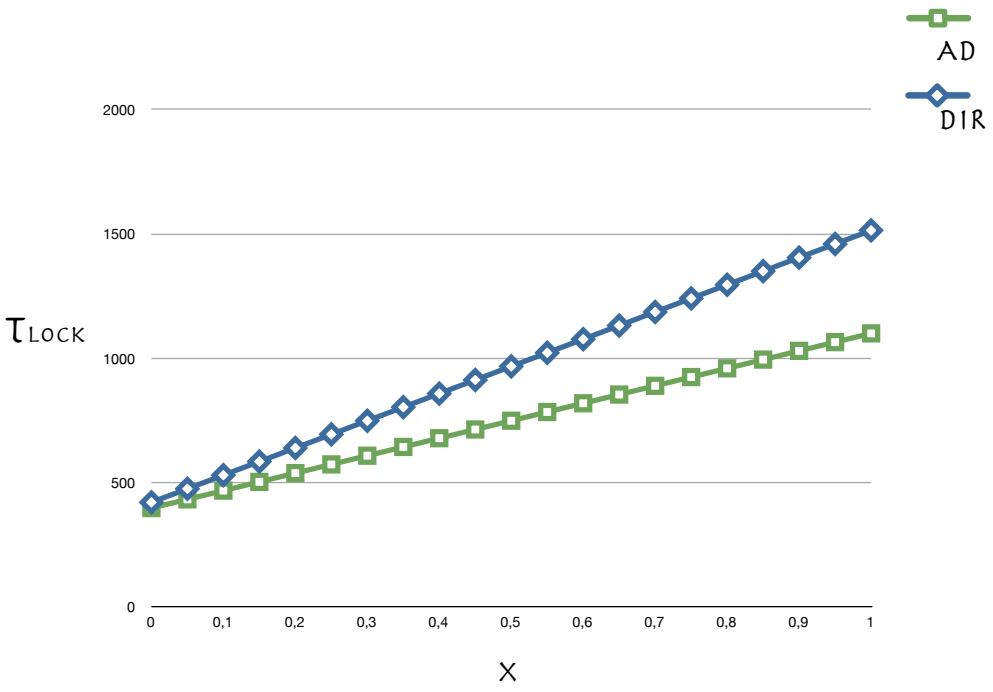
(b) unlock

Figura 5.9: Valutazione della latenza di lock e unlock algorithm-dependent al variare della probabilità di trovare il semaforo occupato

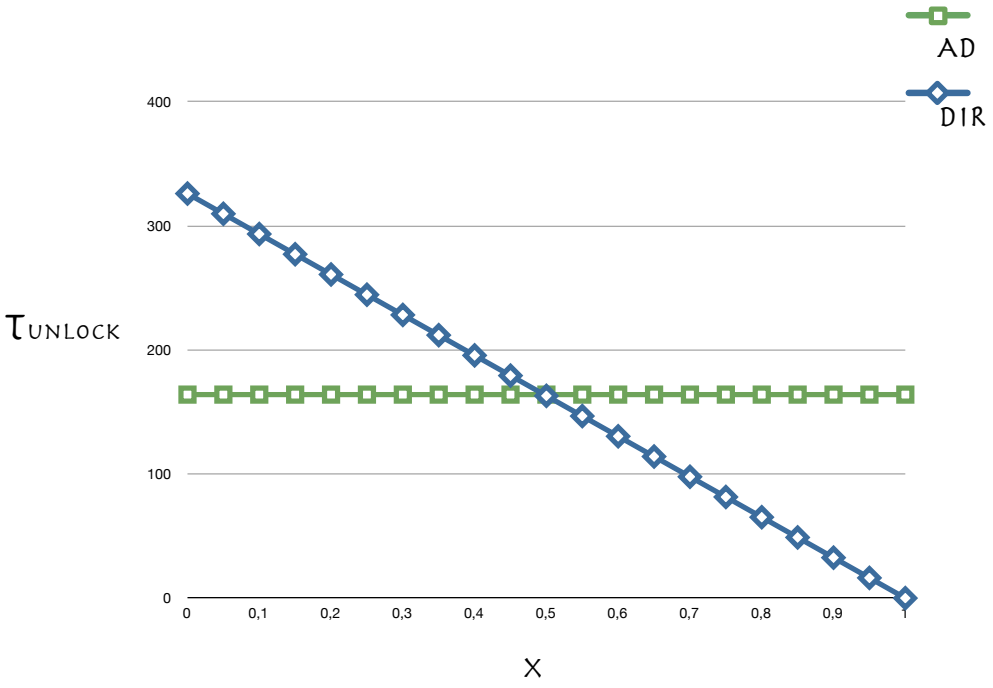
Vediamo i valori assunti da queste grandezze nei due casi, al variare del numero di ripetizioni del tentativo di acquisizione del semaforo e al variare della probabilità di trovarlo occupato.
Il grafico mostrato in figura 5.8 mostra una famiglia di curve che rappresentano l'andamento del valore $T_{lock-ad}$ per il secondo caso considerato, al variare della probabilità x di trovarlo occupato; ad una diversa curva corrisponde un valore diverso r di tentativi di acquisizione del semaforo. Come ci si poteva aspettare all'aumentare del numero di tentativi il tempo richiesto per l'esecuzione della lock aumenta.
I grafici mostrati nelle figure 5.9(a) e 5.9(b) mostrano rispettivamente il valore assunto da $T_{lock-dir}$ e $T_{unlock-dir}$ nei due casi sopraelencati al variare di x, considerando un'architettura NUMA.
Anche in questo approccio il secondo caso è chiaramente il più sfavorevole in quanto tutti gli accessi effettuati sono remoti, inoltre viene mostrato anche il caso in cui il valore assunto da p è pari a 3 invece che 2 come gli altri due casi, contribuendo ad aumentare il tempo di accesso in memoria e alla directory sotto carico nel caso in cui i nodi coinvolti sono tre.
È interessante effettuare il confronto grafico tra le tecniche di cache coherence automatica e l'approccio algorithm-dependent; i grafici mostrati nelle figure 5.10(a) e 5.10(b) mostrano rispettivamente il confronto tra i valori assunti da $T_{lock-dir}$ e $T_{lock-ad}$ e quello tra $T_{unlock-dir}$ e $T_{unlock-ad}$ al variare di x, considerando il secondo caso per tutti i valori e un'architettura NUMA in cui i nodi coinvolti sono due. Per quanto riguarda il confronto relativo all'unlock ricordiamo che, in uno scenario come quello considerato, normalmente la probabilità di trovare il semaforo occupato al momento della lock assume valori minori a 0.5.

5.2.2 Studio delle comunicazioni collettive

Al fine di fornire un modello dei costi completo, che permetta di ricoprire tutti i casi che si possono verificare in una computazione parallela, sia che essa sia implementata in un modello a memoria condivisa che in un modello a scambio di messaggi, è necessario tener conto di scenari che coinvolgono un



(a) lock



(b) unlock

Figura 5.10: Confronto tra le latenze di lock e unlock directory-based e algorithm-dependent al variare della probabilità di trovare il semaforo occupato

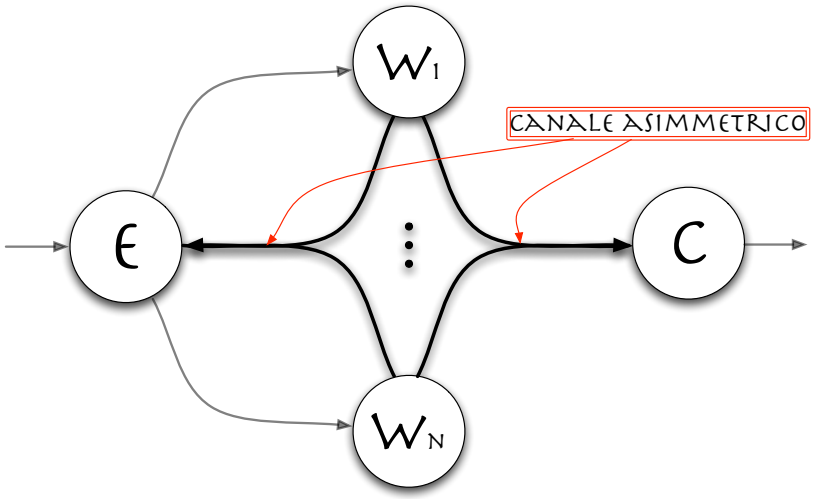


Figura 5.11: Schema di implementazione della forma di parallelismo farm

numero maggiore di processi.
In particolare, riferendoci alle forme di parallelismo più importanti e frequenti, come quelle studiate in [23], che possiamo ritrovare nella maggior parte delle applicazioni parallele, andiamo ad analizzare come si comportano i due approcci in questo diverso insieme di scenari.
Nella programmazione parallela strutturata ritroviamo due importanti forme di parallelismo: *farm* e *data-parallel*.

Farm Con il termine farm indichiamo la forma di parallelismo su *stream* consistente nel *replicare* l'intera elaborazione di un modulo, o sottosistema, su n esecutori (worker) identici, e nel realizzare opportune funzionalità di scheduling delle richieste e di raccolta dei risultati. Questa forma di parallelismo è applicabile a qualunque computazione, senza conoscerne la struttura interna, purché sia noto che tale computazione è puramente funzionale.

La figura 5.11 mostra lo schema di implementazione di un farm. Oltre agli esecutori (worker) $W_1, ..., W_n$, lo schema prevede:

- un processo *emettitore (E)* che svolge una funzionalità di distribuzione dei dati dello stream d'ingresso ai worker;

- un processo *collettore (C)* che svolge la funzionalità di raccolta dei dati elaborati dai worker, da inviare sullo stream di uscita.

L'emettitore è incaricato di effettuare lo scheduling dei valori dello stream di ingresso nei confronti dei worker allo scopo di assicurare il bilanciamento del carico dei worker stessi, condizione essenziale per ottimizzare le prestazioni. Nel caso che il tempo di calcolo di ogni dato sia (significativamente) variabile in funzione del valore del dato stesso, lo scheduling viene implementato con la strategia "on demand" tenendo traccia dei worker attualmente liberi. Per far questo è necessario che i worker segnalino la loro disponibilià all'emettitore attraverso un canale asimmetrico di cui l'emettitore ne è il ricevente. Analogamente per il collettore, viene utilizzato un canale asimmetrico in uscita dai worker su cui questi possono inviare il risultato della computazione.

Data-Parallel Nella forma di parallelismo data-parallel i dati su cui opera la computazione sono *partizionati* tra tutti i moduli esecutori (worker), e questi eseguono la stessa funzione sui dati della propria partizione.
La figura 5.12 mostra lo schema di implementazione di un data-parallel. Oltre agli esecutori (worker) $W_1, ..., W_n$, lo schema prevede:

- un processo *input (I)* che svolge la *scatter*, cioè il partizionamento dei dati d'ingresso ai worker;
- un processo *output (O)* che svolge la funzionalità di *gather* con cui a partire dai dati elaborati dai worker si ricostruisce il risultato dell'intera computazione;

La più semplice computazione data-parallel è la così detta *map* (computazioni locali), nella quale ogni worker lavora esclusivamente, sia in lettura che in scrittura, sulla propria partizione, e quindi non ci sono comunicazioni tra worker per portare avanti il calcolo. Casi più complessi sono le computazioni con *stencil* (computazioni non locali), dove per stencil si intende una configurazione di comunicazioni tra worker che sono necessarie per effettuare il calcolo stesso, in quanto ogni worker, oltre a lavorare su dati della propria partizione, necessita di dati di altre partizioni. Lo stencil può essere:

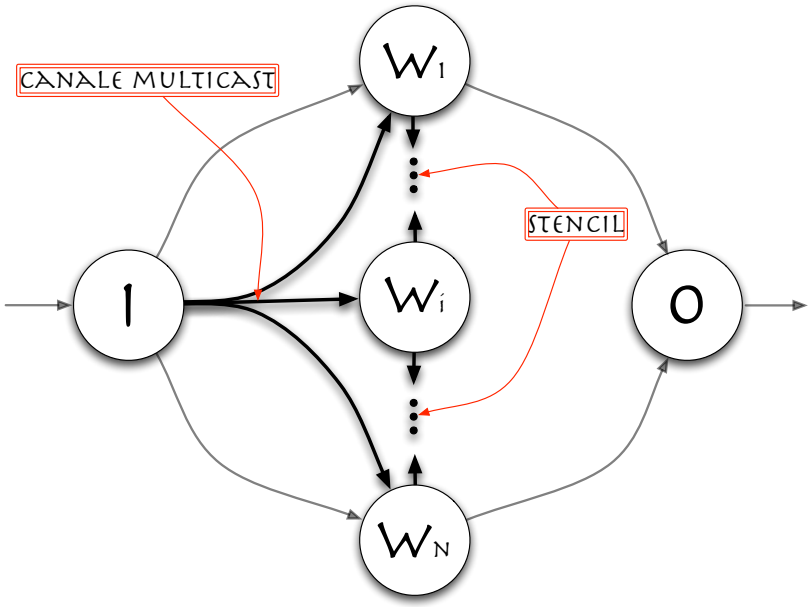


Figura 5.12: Schema di implementazione della forma di parallelismo data-parallel

- *statico* o *dinamico*, a seconda che la sua forma sia prevedibile da una analisi statica della computazione o che dipenda dal valore dei dati a tempo di esecuzione,
- nel caso statico: *fisso* per tutta la durata della computazione, oppure *variabile* durante i passi della computazione, ma (essendo statico) la variazione è tale che la forma assunta dallo stencil ad ogni passo è predicibile.

Alcune computazioni comportano inoltre la necessità di inviare uno stesso dato a tutti i worker di un data-parallel (o di un farm con stato) e per far questo il processo che funge da input può utilizzare un canale di tipo multicast di cui i worker ne sono riceventi.

Per poter analizzare l'impatto che le tecniche di cache coherence hanno sulle comunicazioni collettive come quelle appena descritte, studiamo analogamente a quanto fatto per le comunicazioni simmetriche, cosa avviene quando i processi

devono effettuare questo tipo di comunicazioni operando in mutua esclusione. Nel caso del modello di programmazione a scambio di messaggi, questa situazione si verifica quando mittente/i e destinatario/i provano accedere in mutua esclusione al descrittore del canale che implementa la comunicazione collettiva. Consideriamo un canale multicast per effettuare il confronto tra le tecniche di cache coherence automatica e l'approccio algorithm-dependent; gli stessi ragionamenti si applicano al canale asimmetrico e faremo delle considerazioni analoghe per gli stencil. Esaminiamo il caso in cui un processo I è il mittente di un canale multicast di cui un insieme di processi $W_1, ..., W_n$ ne sono destinatari. Sono due le possibili implementazioni del canale multicast dal punto di vista della mutua esclusione:

- la prima soluzione utilizza *un solo semaforo*, che supponiamo sia allocato sul nodo su cui è allocato il processo I;
- la soluzione alternativa è quella che fa uso di n *semafori* (uno per ogni W_i), ognuno dei quali allocato sul nodo su cui è allocato il corrispondente W_i.

La prima soluzione può influenzare negativamente, rispetto alla seconda, il valore di T_p ai fini della valutazione del tempo di accesso in memoria nel caso di lock implementata con intervallo di tempo di attesa, in quanto si rischia di avere sezioni critiche più lunghe e quindi un numero r di ripetizioni di acquisizione del semaforo più alto. È quindi ancora più importante minimizzare il numero di accessi in memoria durante le operazioni di lock/unlock.
Analizziamo questa prima soluzione e supponiamo, per semplicità di ragionamento, che il blocco contenente il semaforo di lock inizialmente non sia presente in nessuna cache.

Studio dei protocolli directory-based

Le operazioni richieste dal protocollo sono analoghe a quelle viste per le comunicazioni simmetriche, mostreremo in dettaglio solo i punti in cui sono richieste operazioni dovute al fatto che si tratta di comunicazioni che coinvolgono un maggior numero di processi. Ricordiamo che l'utilizzo di un punto di

centralizzazione o di un meccanismo come quello del bit di indivisibilità impedisce che entrambi i processori riescano a copiare in cache il blocco contenente il semaforo di lock.

- Supponiamo che sia I il processo che riesce a copiare per primo il blocco in cache, in seguito ad un fault per l'accesso in lettura al blocco; le eventuali altre richieste in memoria dei W_i nel frattempo saranno bloccate in qualche punto della struttura di interconnessione o in qualche interfaccia verso moduli di memoria condivisa.
  Le operazioni richieste dal meccanismo di cache coherence sono quelle relative al caso di trattamento di fault a seguito di una lettura nel caso in cui il bit *dirty* della directory vale *FALSE*; sappiamo inoltre che in questo caso il nodo *dirty* coincide con il nodo *home* in quanto abbiamo supposto che la copia aggiornata del blocco sia in memoria principale, in questo modo con un'unica richiesta si effettua l'accesso alla directory e alla memoria.

- A questo punto il blocco è nella cache C_{input} in stato *shared* e la successiva scrittura nel blocco causa la gestione di un fault dovuto a scrittura nel caso in cui il bit *dirty* vale *FALSE*.

- Quando I termina la *lock*, se uno tra i W_i era in attesa del blocco ora riesce a completare la sua richiesta al nodo *home*, in particolare l'unità W_{W_i} deve gestire il fault a seguito di una richiesta di lettura, in questo caso con il bit *dirty* che vale *TRUE*: viene inviata una richiesta di trasferimento del blocco contenente il semaforo di lock al nodo *input*, il quale invia i dati sia in memoria che al nodo i e modifica lo stato del proprio blocco a *shared*.

- Fin quando I non eseguirà la *unlock* il generico processo W_i che ha effettuato la lock è bloccato; quando I termina l'esecuzione delle operazioni in mutua esclusione genererà un fault a seguito della richiesta di scrittura del valore del semaforo di lock, in quanto il blocco è in stato *shared*;

```
I ::    unlock (semlock) {
           ⟹   semlock = true;
}
```

il bit *dirty* vale *FALSE* e, trattandosi di un aggiornamento del blocco, l'unica cosa da fare è inviare l'invalidazione a tutti i nodi i che erano bloccati sulla lock (il numero di nodi a cui inviare l'invalidazione è indicato con il parametro n_{inv}) e alla ricezione degli ack il blocco passa di nuovo allo stato *modified*.

$$T_{dir-loc} \sim \left\{ W_{input} \longleftrightarrow Directory \right.$$

$$n_{inv} * T_{inv} \sim n_{inv} * T_{write-rem-1} \sim n_{inv} * \begin{cases} W_{input} \longrightarrow W_{W_i}\texttt{(inv)} \\ W_{W_i} \longrightarrow C_{W_i} \\ W_{W_i} \longrightarrow W_{input}\texttt{(ack)} \end{cases}$$

- A questo punto il primo tra i W_i (si può assumere che appena terminata la sezione critica, il processo I non ne inizi subito un'altra) che rieseguirà la lock, nel richiedere la lettura del valore del semaforo di lock, genererà un fault a seguito della lettura; questo comporterà l'invio del blocco da parte del nodo *input*, in quanto il bit *dirty* vale *TRUE*; ora entrambi i nodi *input* e i hanno il blocco in stato *shared* e W_i può proseguire con l'esecuzione della lock.

- La modifica del valore del semaforo di lock

```
W_i ::    lock (semlock) {
                ...
             ⟹ semlock = false;
                ...
}
```

causa l'aggiornamento del blocco e quindi l'invalidazione del blocco in C_{input}. Siamo sicuri che non ci sono altri nodi che mantengono una copia del blocco in quanto stiamo supponendo che il nodo i è il primo che è

riuscito ad eseguire la lock dopo l'unlock del nodo *input*

$$T_{dir-rem} \sim \begin{cases} W_{W_i} \longrightarrow W_{input} \\ W_{input} \longleftrightarrow Directory \\ W_{input} \longrightarrow W_{W_i} \end{cases}$$

$$T_{inv} \sim T_{write-rem-1} \sim \begin{cases} W_{W_i} \longrightarrow W_{input}\texttt{(inv)} \\ W_{input} \longrightarrow C_{input} \\ W_{input} \longrightarrow W_{W_i}\texttt{(ack)} \end{cases}$$

- L'esecuzione dell'*unlock* da parte di W_i può generare un fault a seguito della richiesta di scrittura del valore del semaforo di lock, nel caso in cui il blocco è in stato *shared* a causa di tentativi di esecuzione della lock da parte degli altri processi;

```
W_i ::    unlock (semlock) {
             ⟹    semlock = true;
}
```

il bit *dirty* vale *FALSE* e, trattandosi di un aggiornamento del blocco, l'unica cosa da fare è inviare l'invalidazione a tutti i nodi che erano bloccati sulla lock e alla ricezione dell'*ack* il blocco passa di nuovo allo stato *modified*.

$$T_{dir-rem} \sim \begin{cases} W_{W_i} \longrightarrow W_{input} \\ W_{input} \longleftrightarrow Directory \\ W_{input} \longrightarrow W_{W_i} \end{cases}$$

$$n_{inv} * T_{inv} \sim n_{inv} * T_{write-rem-1} \sim n_{inv} * \begin{cases} W_{W_i} \longrightarrow W_j\texttt{(inv)} \\ W_j \longrightarrow C_j \\ W_j \longrightarrow W_{W_i}\texttt{(ack)} \end{cases}$$

Possiamo quindi determinare. come fatto per le comunicazioni simmetriche, i parametri del modello dei costi delle operazioni di mutua esclusioni nel caso di utilizzo di tecniche di cache coherence automatica basate su directory di tipo memory-based. Considerando la probabilità x di trovare il semaforo di lock occupato, si possono calcolare i valori di $T_{lock-dir}$ e $T_{unlock-dir}$ nei seguenti due casi:

1. architettura NUMA, numero di destinatari n, lock/unlock eseguite dal processo I:

$$\begin{aligned} T_{lock-dir} &\sim (1-x)(T_{dir+read-loc} + T_{dir-loc}) \\ &+ x(T_{dir-loc} + T_{write-rem-1} \\ &+ (1+r)(T_{dir-loc} + T_{read-rem})) \\ &\sim (1-x)(R_{Q-mem+dir-loc} + R_{Q-dir-loc}) \\ &+ x(R_{Q-dir-loc} + R_{Q-1} \\ &+ (1+r)(R_{Q-dir-loc} + R_Q)) \\ T_{unlock-dir} &\sim T_{dir-loc} + n_{inv}T_{write-rem-1} \\ &\sim (R_{Q-dir-loc} + n_{inv}R_{Q-1}) \end{aligned}$$

2. architettura SMP o architettura NUMA numero di destinatari n, lock/unlock eseguite dal generico processo W_i:

$$\begin{aligned} T_{lock-dir} &\sim (1-x)(T_{dir+read-rem} + T_{dir-rem}) \\ &+ x(T_{dir-rem} + T_{write-rem-1} \\ &+ (1+r)(T_{dir+read-rem})) \\ &\sim (1-x)(R_{Q-mem+dir} + R_{Q-dir}) \\ &+ x(R_{Q-dir} + R_{Q-1} + (1+r)R_{Q-mem+dir}) \\ T_{unlock-dir} &\sim T_{dir-rem} + n_{inv}T_{write-rem-1} \\ &\sim R_{Q-dir} + n_{Inv}R_{Q-1} \end{aligned}$$

Studio dell'approccio algorithm-dependent

Possiamo notare che la valutazione dei parametri del modello dei costi delle operazioni di mutua esclusioni nel caso di utilizzo dell'approccio algorithm-dependent è la stessa fatta per le comunicazioni simmetriche. Questo perché l'algoritmo è indipendente dal grado di parallelismo e il tipo di accessi effettuati non cambia. Considerando la probabilità x di trovare il semaforo di lock occupato si possono calcolare i valori di $T_{lock-ad}$ e $T_{unlock-ad}$ come prima nei seguenti due casi:

1. architettura NUMA, numero di destinatari n, lock/unlock eseguite dal processo I:

$$\begin{aligned} T_{lock-ad} &\sim T_{read-loc} + T_{write-loc-1} \\ &+ \quad x(1+r)T_{read-loc} \\ &\sim R_{Q-loc} + R_{Q-loc-1} + x(1+r)R_{Q-loc} \\ T_{unlock-ad} &\sim T_{write-loc-1} \sim R_{Q-loc-1} \end{aligned}$$

2. architettura SMP o architettura NUMA numero di destinatari n, lock/unlock eseguite dal generico processo W_i:

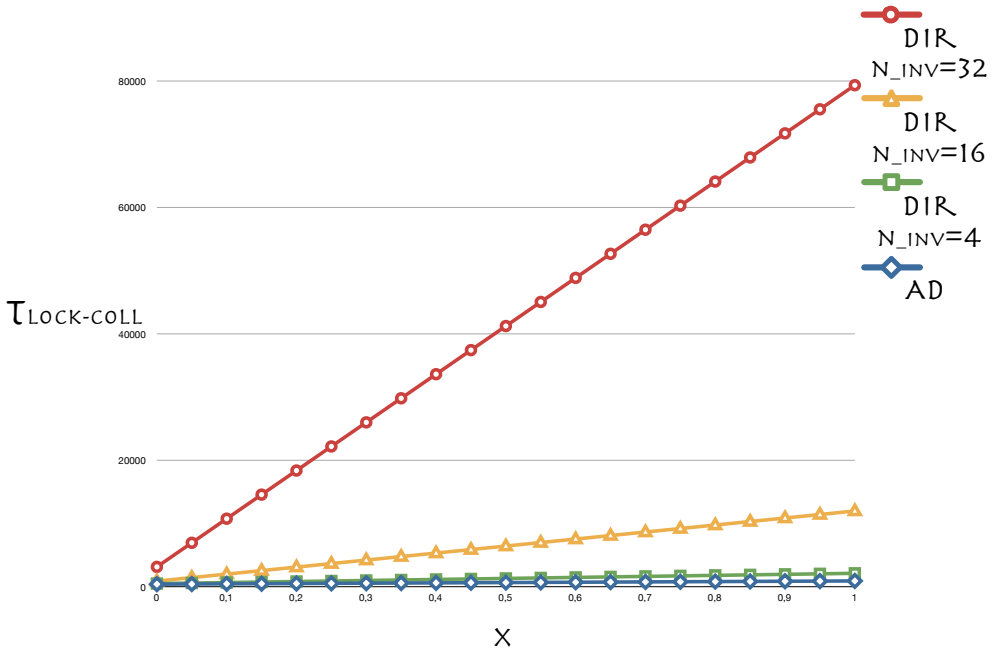
$$\begin{aligned} T_{lock-ad} &\sim T_{read-rem} + T_{write-rem-1} \\ &+ \quad x(1+r)T_{read-rem} \\ &\sim R_Q + R_{Q-1} + x(1+r)R_Q \\ T_{unlock-ad} &\sim T_{write-rem-1} \sim R_{Q-1} \end{aligned}$$

È interessante effettuare il confronto grafico tra le tecniche di cache coherence automatica e l'approccio algorithm-dependent anche in questo caso delle comunicazioni collettive; i grafici mostrati nelle figure 5.13(a) e 5.13(b) mostrano rispettivamente il confronto tra i valori assunti da $T_{lock-dir}$ e $T_{lock-ad}$ e quello tra $T_{unlock-dir}$ e $T_{unlock-ad}$ al variare di x, considerando il secondo caso per tutti i valori. In particolare ogni confronto utilizza una famiglia di curve per i valori delle tecniche automatiche al fine di mostrare l'influenza del numero di invalidazioni sulle prestazioni di queste tecniche.

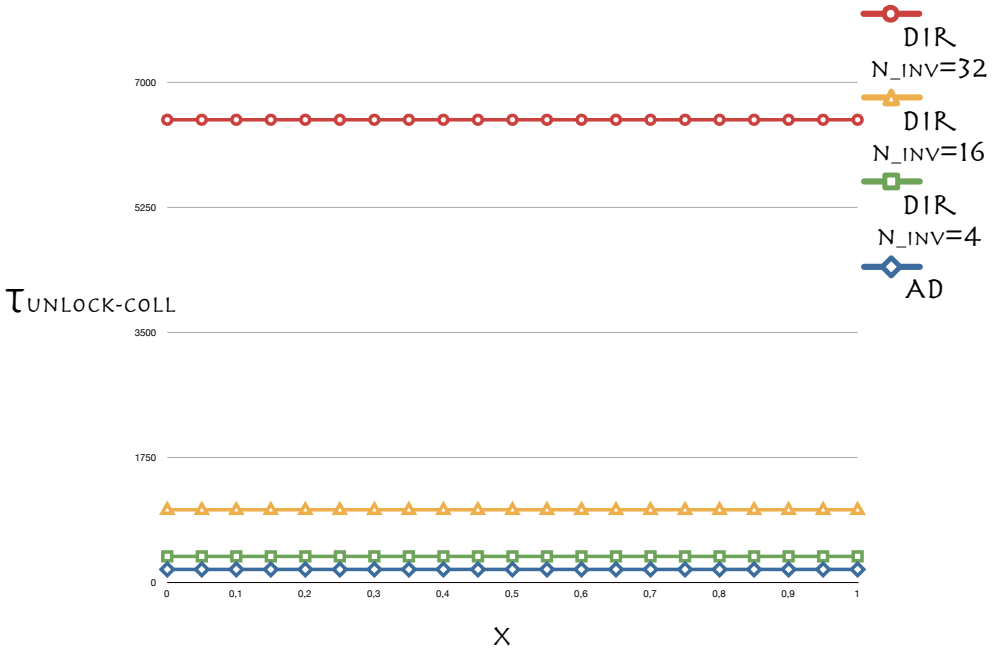
La differenza notevole delle prestazioni delle tecniche di cache coherence automatica al variare del numero di invalidazioni necessarie è chiaramente giustificata dal valore assunto da questo parametro ma non solo; vedremo un'altra motivazione nella prossima sezione in cui consideriamo anche le altre comunicazioni collettive e l'implementazione alternativa del canale con un numero di semafori pari al numero dei destinatari.

Considerazioni sulle comunicazioni collettive

Come abbiamo visto nella sezione precedente un canale multicast può essere implementato in due modi diversi relativamente al problema della mutua



(a) lock



(b) unlock

Figura 5.13: Confronto tra le latenze di lock e unlock directory-based e algorithm-dependent per le comunicazioni collettive al variare della probabilità di trovare il semaforo occupato e al variare del numero di invalidazioni richieste nel caso directory-based

esclusione; questo ragionamento può essere fatto anche per il canale asimmetrico.
Ancora più interessante è che le considerazioni fatte per il canale multicast che utilizza un solo semaforo di lock sono valide anche per il canale asimmetrico; infatti la teoria esposta è indipendente dalle operazioni della sezione critica compresa tra le operazioni di lock/unlock, che è l'unica cosa che distingue i due tipi di canale.
Per quanto riguarda la soluzione con un numero di semafori pari al numero n di destinatari nel caso del canale multicast, possiamo notare che ai fini della valutazione delle prestazioni delle operazioni di mutua esclusione, questo tipo di implementazione equivale ad una soluzione che fa uso di n canali simmetrici. Questa osservazione può essere applicata anche alle comunicazioni dettate dagli stencil definiti da una computazione data-parallel.
Vediamo ora cosa cambia tra le due soluzioni nel caso di utilizzo delle tecniche di cache coherence automatica e dell'approccio algorithm-dependent.
Nella soluzione che utilizza un solo semaforo di lock:

- con le tecniche di cache coherence automatica il valore assunto da p è
  - $p = n + 1$ nella valutazione del tempo di accesso al modulo di memoria del nodo home, che abbiamo supposto sia lo stesso su cui è allocato il processo I;
  - $p = 2 + n_{inv}$ nella valutazione del tempo di accesso al modulo di memoria dei nodi su cui sono allocati i processi W_i;
- con l'approccio algorithm-dependent il valore assunto da p è
  - $p = n + 1$ nella valutazione del tempo di accesso al modulo di memoria del nodo su cui è allocato il processo I;
  - $p = 1$ nella valutazione del tempo di accesso al modulo di memoria dei nodi su cui sono allocati i processi W_i, in quanto l'algoritmo non causa accessi a moduli di memoria diversi da quello in cui è allocato il semaforo di lock.

Nella soluzione che utilizza n semafori di lock e quindi anche nel caso degli stencil (dove n rappresenta le comunicazioni necessarie per ogni processo W_i):

- con le tecniche di cache coherence automatica il valore assunto da p è
  - $p = n+1$ nella valutazione del tempo di accesso al modulo di memoria del nodo su cui è allocato il processo I a causa delle invalidazioni da parte dei nodi su cui sono allocati i processi W_i;
  - $p = 2$ nella valutazione del tempo di accesso al modulo di memoria dei nodi home, che abbiamo supposto siano gli stessi su cui sono allocati i processi W_i;
- con l'approccio algorithm-dependent il valore assunto da p è
  - $p = 1$ nella valutazione del tempo di accesso al modulo di memoria del nodo su cui è allocato il processo I, in quanto, analogamente all'altra soluzione, l'algoritmo non causa accessi a moduli di memoria diversi da quello in cui è allocato il semaforo di lock.;
  - $p = 2$ nella valutazione del tempo di accesso al modulo di memoria dei nodi su cui sono allocati i processi W_i.

Queste osservazioni permettono di giustificare, come anticipato nella sezione precedente, l'andamento delle prestazioni delle tecniche di cache coherence automatica al variare del numero di invalidazioni necessarie: come abbiamo visto infatti questo parametro influenza anche il parametro p per il calcolo del tempo di accesso in memoria condivisa sotto carico.

5.3 Conclusioni

In questo capitolo abbiamo provato a formalizzare l'impatto che le tecniche di cache coherence hanno sulle prestazioni delle architetture multiprocessor, cercando di capire che ruolo gioca la scelta del livello di implementazione di queste tecniche.
A partire dalla teoria delle code abbiamo visto che è possibile ottenere una valutazione della prestazioni dei tempi di accesso alla memoria condivisa e,

proprio a partire da questa, è stato possibile determinare quantitativamente i parametri del modello dei costi delle operazioni di mutua esclusione sia nel caso di utilizzo di tecniche di cache coherence automatica basate su directory di tipo memory-based, sia nel caso dell'approccio algorithm-dependent.
Non solo l'approccio algorithm-dependent richiede un minor numero di accessi in memoria condivisa, ma è in grado di influenzare in maniera positiva, rispetto alle tecniche di cache coherence automatica, il parametro p che influenza il tempo di accesso in memoria condivisa, soprattutto nel caso delle comunicazioni collettive tipicamente usate nelle computazioni parallele.

Capitolo 6

Conclusioni e Sviluppi Futuri

In questa tesi abbiamo visto come il fondamentale utilizzo delle memorie cache nelle architetture multiprocessor ha portato alla nascita del problema della cache coherence. Il problema nasce come conseguenza delle architetture all-cached e i due meccanismi che generalmente il livello hardware-firmware di un sistema mette a disposizione per farvi fronte sono l'invalidazione e l'aggiornamento. Numerosi sono i protocolli sviluppati che fanno uso di questi meccanismi (MSI, MESI, Dragon ecc.); in particolare questi protocolli hanno permesso di descrivere in modo sempre più efficiente le azioni che devono essere intraprese dai nodi di elaborazione di un multiprocessor per risolvere il problema. Ognuno di questi protocolli prevede un diverso insieme di possibili stati da associare a ciascun blocco di cache e le diverse unità che fungono da controllori della coerenza dei nodi implementano in maniera distribuita il protocollo. Inoltre abbiamo visto che tutti i protocolli presi in analisi possono essere estesi anche ai sistemi moderni che fanno uso di una gerarchia di memorie con più livelli di cache.
Questi protocolli sono alla base delle tecniche di cache coherence automatica: vengono costai definite in quanto il supporto alla cache coherence è interamente delegato all'architettura, rendendo il problema trasparente ai livelli più alti. Abbiamo visto che queste tecniche si suddividono in due principali categorie: nella soluzione snoopy-based si fa uso di un bus come punto di centralizzazione, mentre le tecniche basate su directory vengono normalmente adottate nei sistemi a più alto grado di parallelismo in quanto fanno uso di strutture

condivise in memoria principale per implementare i vari protocolli. In particolare abbiamo visto diverse tecniche di tipo directory-based: la caratteristica principale che permette di distinguere le varie strategie riguarda la modalit di condivisione delle informazioni tra i nodi di elaborazione, cio se le strutture dati sono centralizzate in memoria o condivise tra i vari nodi, in maniera piatta o gerarchica. Le due soluzioni possono essere combinate per dar luogo ad approcci ibridi ed entrambe possono essere estese per trattare sistemi con più livelli di cache.

L'approccio algorithm-dependent si differenzia dalle tecniche di cache coherence automatica in quanto permettono di integrare la risoluzione del problema della cache coherence nei meccanismi per la mutua esclusione. Dopo aver analizzato alcune tecniche di implementazione dei meccanismi per la mutua esclusione necessari nelle architetture multiprocessor per l'accesso a strutture condivise, abbiamo preso in considerazione un approccio più flessibile, rispetto a quelli che fanno uso di istruzioni assembler complesse, che utilizza un set istruzioni semplice in cui si utilizzano dei meccanismi forniti dal livello hardware-firmware. A partire dall'approccio scelto abbiamo esteso una possibile implementazione degli algoritmi di lock/unlock in modo da integrarvi, secondo un approccio algorithm-dependent, specifiche azioni rivolte a risolvere il problema della cache coherence senza far affidamento su tecniche automatiche.

Per poter effettuare un confronto tra le due possibili soluzioni al problema della cache coherence abbiamo provato a sviluppare un modello dei costi che ci ha permesso di capire che ruolo gioca la scelta del livello di implementazione di queste tecniche. A partire dalla teoria delle code abbiamo visto che è possibile ottenere una valutazione della prestazioni dei tempi di accesso alla memoria condivisa e, proprio a partire da questa, è stato possibile determinare quantitativamente i parametri del modello dei costi delle operazioni di mutua esclusione sia nel caso di utilizzo di tecniche di cache coherence automatica basate su directory di tipo memory-based, sia nel caso dell'approccio algorithm-dependent. Non solo l'approccio algorithm-dependent richiede un minor numero di accessi in memoria condivisa, ma è in grado di influenzare in maniera positiva, rispetto alle tecniche di cache coherence automatica, il pa-

rametro p che influenza il tempo di accesso in memoria condivisa, soprattutto nel caso delle comunicazioni collettive tipicamente usate nelle computazioni parallele.

Una naturale continuazione del lavoro di tesi è un'analisi più estesa dell'impatto che le soluzioni viste possono avere al variare della struttura di interconnessione utilizzata dell'architettura. In particolare, adottando reti di interconnessione diverse dalle reti fat tree esaminate, le quali rendono sostanzialmente trascurabili i conflitti a livello di unità di switch, specie per valori limitati del parametro p, ci aspettiamo che sia necessario tener conto esplicitamente della congestione della rete. In questo caso può essere ancora più importante minimizzare il numero di comunicazioni interprocessor richieste ad esempio dai protocolli utilizzati nelle tecniche automatiche.

Il numero di variabili in gioco nello studio del problema della cache coherence è alto e i simulatori esistenti non ci hanno permesso di effettuare un'analisi sufficientemente parametrica del problema. Come abbiamo detto, non vogliamo uno studio del problema che sia legato ad un particolare tipo di architettura, ad una particolare struttura di interconnessione o ad un particolare protocollo di cache coherence; per questo motivo sarebbe interessante progettare e realizzare un nuovo simulatore, che sia il più generale possibile e che ci permetta di validare i risultati ottenuti dal modello dei costi sviluppato nella tesi.

Bibliografia

[1] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of hardware and software cache coherence schemes. *SIGARCH Comput. Archit. News*, 19(3):298–308, 1991.

[2] 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)*, pages 353–362, New York, NY, USA, 1998. ACM.

[3] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[4] 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*, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[5] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[6] 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.

[7] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *ISCA*

'88: Proceedings of the 15th Annual International Symposium on Computer architecture, pages 373–382, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[8] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. *SIGARCH Comput. Archit. News*, 17(3):2–15, 1989.

[9] Bin feng Qian and Li min Yan. The research of the inclusive cache used in multi-core processor. In *Electronic Packaging Technology High Density Packaging, 2008. ICEPT-HDP 2008. International Conference on*, pages 1 –4, 28-31 2008.

[10] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, New York, NY, USA, 1983. ACM.

[11] Anoop Gupta, Wolf dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

[12] Davib B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1):10–22, 1992.

[13] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–422, New York, NY, USA, 2009. ACM.

[14] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159, New York, NY, USA, 1990. ACM.

[15] David J. Lilja. Cache coherence in large-scale shared memory multiprocessors: Issues and comparisons. *ACM COMPUTING SURVEYS*, 25:303–338, 1993.

[16] John D. C. Little. A proof for the queuing formula: L= w. *Operations Research*, 9(3):383–387, 1961.

[17] Tom Lovett and Russell Clapp. STiNG: a CC-NUMA computer system for the commercial marketplace. *SIGARCH Comput. Archit. News*, 24(2):308–317, 1996.

[18] Edward M. McCreight. The Dragon Computer System: An Early Overview. Technical report, Xerox Corporation, Polo Alto Research Center, Palo Alto, Ca., 94304, December 7, 1984.

[19] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 230–242, New York, NY, USA, 1989. ACM.

[20] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM.

[21] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[22] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[23] M. Vanneschi. *Architettura degli Elaboratori.* Edizioni PLUS, Università di Pisa, 2009.