Sapienza
Università di Roma

Gestione ottimizzata della delivery e del buffering dei messaggi in piattaforme multi-thread in architetture NUMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica e Automatica

Candidato
Andrea Scarselli
Matricola 1531508

Relatore
Prof. Francesco Quaglia

Correlatore
Ing. Alessandro Pellegrini

Anno Accademico 2014/2015

Tesi non ancora discussa

Gestione ottimizzata della delivery e del buffering dei messaggi in piattaforme multi-thread in architetture NUMA
Tesi di Laurea. Sapienza – Università di Roma

© 2015 Andrea Scarselli. Tutti i diritti riservati

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: scarselli.1531508@studenti.uniroma1.it

Alla mia famiglia,
che ha sempre sostenuto
ed incoraggiato
tutte le mie scelte

Indice

Elenco delle figure vii

Introduzione 1

1 Analisi delle tecnologie e strumenti 3
1.1 Architettura NUMA 3
1.2 Esecuzione speculativa 4
1.2.1 Esempio: Pipeline e Branch Prediction 5
1.3 Rollback 5
1.4 Parallel Discrete Event Simulation 6
1.5 Protocollo di sincronizzazione ottimistico 6
1.5.1 Meccanismo di controllo locale del Time Warp 7
1.5.2 Meccanismo di controllo globale del Time Warp 8

2 Descrizione del problema e soluzioni 11

3 Architettura di riferimento e Implementazione 15
3.1 API NUMA 15
3.1.1 libnuma 16
3.2 Architettura ROOT-Sim 17
3.2.1 Livello applicativo 18
3.2.2 Livello kernel 18
3.3 Implementazione 19
3.3.1 Scambio dei messaggi 19
3.3.2 Ingoing Buffer 21
3.3.3 Anatomia di un blocco di ingoing buffer 22
3.3.4 Allocazione 23
3.3.5 Deallocazione 24
3.3.6 Riallocazione 26
3.4 Sviluppi futuri 27

4 Dati Sperimentali 29
4.1 Tempi di esecuzione 30
4.2 Efficienza 30

Bibliografia 33

Elenco delle figure

1.1 Generica architettura NUMA 4

2.1 Esempio di frammentazione esterna 12

3.1 Architettura ROOT-Sim 17
3.2 Esempio di sistema PDES 19
3.3 Anatomia di un blocco di ingoing buffer 22
3.4 Allocazione, utilizzo dell'extra_buffer 23
3.5 Allocazione, caso 2 24
3.6 Allocazione, caso 3 25
3.7 Deallocazione, caso 1 25
3.8 Deallocazione, caso 2 26
3.9 Deallocazione, caso 3 e 4 26

4.1 Tempo di esecuzione e Speedup 30
4.2 Efficienza 31

Introduzione

In questo lavoro presento l'ideazione e la realizzazione in C di un allocatore e gestore di memoria finalizzato allo scambio di messaggi in un software di tipo *Parallel Discret Event Simulation* (PDES). L'allocatore è stato progettato e realizzato per essere eseguito su una macchina con architettura di tipo *Non Uniform Memory Access* (NUMA).

Un software di tipo PDES è un software parallelo, atto alla simulazione di un sistema reale, in cui viene sfruttato il paradigma della programmazione ad eventi. Nella programmazione ad eventi il flusso di esecuzione del programma è determinato dagli eventi che i thread ricevono: questi possono arrivare a causa di interazioni con l'utente, oppure, come nel nostro caso, possono arrivare da altri thread attraverso lo scambio di specifici messaggi. Essendo il flusso regolato dagli eventi è chiara l'importanza di avere una delivery rapida ed affidabile.

L'allocatore è stato ideato appositamente per essere efficiente in macchine realizzate con un'architettura di tipo NUMA: l'allocazione della memoria in macchine con questo tipo di architettura è un problema fondamentale, infatti, queste architetture necessitano di un'accurata gestione della memoria per non ricadere in situazioni che possono provocare forti latenze di esecuzione. Ho dovuto dunque tener cura di posizionare i messaggi in buffer allocati nella sezione di memoria "locale", ossia, nella memoria nella quale un determinato processo può accedere in maniera più rapida. I buffer sono preallocati e sono in grado di contenere un gran numero di messaggi; questa scelta ridurrà al minimo le richieste di allocazione (e deallocazione) di memoria al sistema operativo sottostante. Questa scelta inoltre permette ai thread di avere il payload in una posizione fissa e quindi utilizzare l'header del messaggio (molto più leggero senza payload) nei vari livelli di gestione. L'utilizzo di buffer preallocati porta con se la complicazione dovuta alla saturazione degli stessi, risolta con una riallocazione dinamica dei buffer. I messaggi sono strutturati in modo da esserre *zero-copy* e con payload variabile.

L'accesso a questa struttura dati è eseguito in sezione critica per gestire la scrittura di un gran numero di processi. Il risultato finale è un algoritmo in grado di:

- gestire l'accesso in lettura e scrittura di un elevato numero di processi;
- permettere lo scambio di messaggi arbitrariamente grandi;
- permettere lo scambio di un numero qualsiasi di messaggi;

- ottimizzato rispetto alla memoria (nell'ambito dell'architettura NUMA).

L'implementazione è stata effettuata in *ROme OpTimistic Simulator* (ROOT-Sim), un software PDES sviluppato dal gruppo di ricerca "High Performance and Dependable Computing Systems" (HPDCS) dell'università di Roma "La Sapienza".

Il resto del lavoro è organizzato come segue. Nel primo capitolo presento brevemente le tecnologie ed i paradigmi utilizzati nell'ideazione degli algoritmi e dal simulatore. Nel secondo capitolo presento il problema e discuto delle possibili soluzioni. Nel terzo capitolo presento l'architettura di riferimento, i servizi da essa offerti e presento l'implementazione degli algoritmi. Nel quarto capitolo, infine, saranno presentati dei dati sperimentali che mostreranno come variano il tempo di esecuzione e l'efficienza (relativa ai rollback effettuati, come discusso in seguito) all'aumentare del grado di parallelismo.

Capitolo 1

Analisi delle tecnologie e strumenti

1.1 Architettura NUMA

Una macchina realizzata secondo l'architettura NUMA è una macchina multicore nella quale ogni core (o un ristretto gruppo di core), può accedere in maniera privata e rapida ad una certa zona di memoria. Ogni gruppo di core (e la relativa memoria ad esso associata) è noto come nodo NUMA.

L'architettura di tipo NUMA nasce negli anni '90. In quel periodo, il divario prestazionale tra i microprocessori e le memorie cresce al punto da rendere l'accesso alla RAM un inaccettabile collo di bottiglia. I microprocessori vengono dotati di memorie cache (velocissime ma costose) sempre più grandi, ma, l'aumento delle dimensioni del software rende questa soluzione non sufficiente (a causa degli elevati costi). L'accesso alla memoria è ancor più critico in ambiente multicore in quanto un solo core alla volta può accedere al bus di memoria. Nel caso in cui più processori abbiano bisogno di accedervi bisognerà sincronizzare i processi ed attendere il proprio turno. Con l'architettura NUMA ogni processore può accedere alla propria area di memoria (chiamata *memoria privata*) con l'utilizzo di un bus privato e quindi questo problema è notevolmente ridotto. Durante l'esecuzione può capitare che un core richieda l'accesso (in scrittura o in lettura) ad una sezione di memoria attestata su un altro nodo (in questo caso si dice che il core sta effettuando un accesso in *memoria remota*); questo è possibile in quanto i nodi sono interconnessi da un bus ad alte prestazioni e viene eseguito il seguente protocollo:

1. Il core interessato ad un'indirizzo di memoria effettua una richiesta al core che vi può accedere;

2. Il core che riceve la richiesta carica in cache il contenuto presente all'indirizzo richiesto;

3. Attraverso un protocollo di message passing invia questo contenuto al core che ha effettuato la richiesta (eventualmente passando per nodi intermedi);

4. Il messaggio arriva al core che ha fatto richiesta e mantiene i dati in cache (ora è in grado di utilizzarli sia in lettura che in scrittura);

5. Un protocollo di coerenza della cache si preoccupa di propagare le eventuali modifiche effettuate da un nodo a tutti i nodi che detengono quell'area di memoria in cache.

Questo protocollo, sebbene risolva il problema dell'accesso in memoria remota, può creare parecchi problemi:

- vengono invalidate righe di cache in entrambi i nodi;
- non è facile stimare l'effetto di un accesso in memoria remota in quanto, in generale, nella fase di message passing, possono essere necessari più "hop" per raggiungere la memoria richiesta;
- il core che riceve la richiesta deve interrompere l'esecuzione per accedere in memoria e far fronte alla richiesta.

È chiaro, quindi, che l'efficienza di una macchina NUMA è fortemente caratterizzata dal numero di accessi in memoria remota.

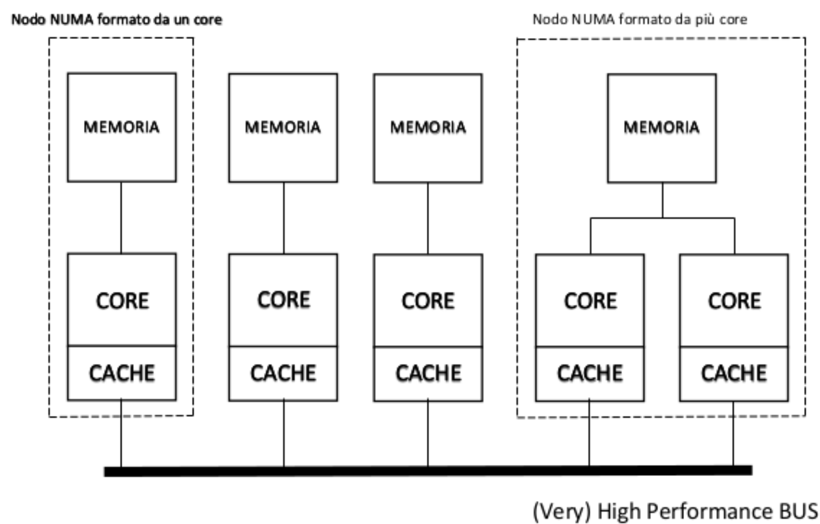

Figura 1.1. Generica architettura NUMA

1.2 Esecuzione speculativa

Con esecuzione speculativa si intende un' ottimizzazione con la quale, nel caso in cui non sia chiaro quale operazioni eseguire, si utilizzano delle tecniche di predizione per continuare l'esecuzione. Se la previsione si dovesse rilevare corretta, guadagneremmo il tempo che avremmo aspettato nell'attesa di conoscere quale operazione eseguire (in questo caso si parla di *esecuzione ritardata*); nel caso in cui, invece, la previsione si dovesse rilevare sbagliata, i risultati verebbero scartati e le modifiche effettuate verebbero ripristinate. Tenendo in considerazione che l'esecuzione delle operazioni comporta consumo di risorse bisogna essere certi che, nel medio/lungo periodo,

l'esecuzione speculativa comporti vantaggi in termini di tempo e spazio rispetto all'esecuzione ritardata.

L'esempio più diffuso di esecuzione speculativa si ha con la funzione di "branch prediction", presente in tutti i moderni microprocessori con pipeline.

1.2.1 Esempio: Pipeline e Branch Prediction

Il pipelining è una tecnica atta ad aumentare il troughput (numero di istruzioni completate per unità di tempo) dei microprocessori. Il pipeling nasce osservando che l'esecuzione di ogni istruzione può essere divisa in cinque passi (che diventeranno gli stadi della pipeline):

- Instruction Fetch (IF): Lettura dell'istruzione dalla memoria;
- Instruction Decode (ID): Decodifica dell'istruzione e prelievo degli operandi dalla memoria;
- Instruction Execution (EX): Esecuzione dell'istruzione;
- Memory Access (MEM): Accesso e aggiornamento della memoria (necessaria solo per operazioni di load/store);
- Write Back (WB): Scrittura del risultato nel registro destinatario.

Essendo queste fasi ricorrenti in tutte le istruzioni e, essendo indipendenti tra loro, si possono eseguire cinque istruzioni contemporaneamente. In questo modo, quando la pipeline sarà a regime, avremo, in ogni stadio, un'istruzione diversa. Un problema sorge quando ci troviamo a dover affrontare un'operazione di salto condizionato: non sapremo il risultato della comparazione finchè essa non sarà arrivata alla fase EX. In questo modo l'istruzione successiva non potrà essere caricata in memoria e lasceremo due stadi della pipeline inutilizzati. Un modo di procedere è quello di utilizzare il branch prediction: si utilizzano delle techiche probabilistiche per scegliere un certo branch da eseguire, ad esempio, nel caso di un ciclo, si sceglie di continuare la sua esecuzione in quanto, chiaramente, sono più le volte in cui questo accadrà che non le volte in cui si uscirà dal ciclo (una sola volta). In questo caso l'esecuzione speculativa è chiaramente un vantaggio.

1.3 Rollback

Il *rollback* è un' operazione attraverso la quale, durante l'esecuzione di un programma, viene ripristinato uno stato antecedente. Quest'operazione viene effettuata quando viene rilevato un errore (ad esempio, una scelta sbagliata nell'ambito dell'esecuzione speculativa) o si scopre che lo stato attuale non è integro. Per supportare il rollback è necessario mantenere, in una qualche struttura, gli stati passati. Ad intervalli di tempo regolari il sistema effettua una fotografia (snapshot) dello stato del sistema e la salva. La scelta della frequenza con la quale effettuare questa operazione è importante in quanto un tempo breve comporta una precisione maggiore e, di conseguenza, la possibilità di annullare il minor numero di operazioni possibili. Tuttavia, l'operazione

di cattura dello snapshot è overhead sia temporale che spaziale e quindi non può essere eseguita troppo frequentemente.

Nel caso della programmazione ad eventi possiamo definire l'*efficienza* come il rapporto tra il numero di eventi necessari per raggiungere lo stato finale (committed events) ed il numero di eventi realmente processati durante l'esecuzione. L'efficienza varia tra 0 ed 1. Nel caso di efficienza 1 si ha che nessun evento processato è stato annullato, ossia, non si è mai verificato un rollback; nel caso di efficienza uguale a 0 il sistema non riesce a raggiungere lo stato finale.

1.4 Parallel Discrete Event Simulation

Il concetto di Parallel Discrete Event Simulation (PDES) nasce come evoluzione del Discrete Event Simulation (DES). DES è un meccanismo attraverso il quale un sistema reale viene modellato matematicamente e rappresentato da uno stato S; questo stato evolve con un susseguirsi di eventi discreti. Un evento discreto è un evento impulsivo nel senso che inizia e finisce nello stesso istante di tempo (simulato). L'esecuzione di un evento causa un cambio di stato del sistema e non possono esserci cambi di stato senza il processamento di un evento.

Nel PDES la simulazione viene suddivisa in N sottoinsiemi di stati S_i. Ogni S_i contiene le informazioni su un certo numero di variabili e si ha che $S = \bigcup_{i=1...N} S_\mathrm{i}$ con $S_\mathrm{i} \cap S_\mathrm{j} = \emptyset \ \forall i \neq j$. Ogni sottoinsieme di stati viene associato ad un processo logico (Logical Process, LP) e, questi, comunicano tra loro attraverso lo scambio di messaggi. Ovviamente questi devono avere una qualche forma di sincronizzazione in quanto sono frammenti della stessa rappresentazione e devono avere informazioni coerenti. Ci sono due strade possibili:

- Metodo conservativo in cui si utilizzano i classici metodi di sincronizzazione per la mutua esclusione (semafori, lock, busy waiting su variabili di stato ecc.) ed ogni evento viene eseguito solamento quando esso è considerato "sicuro" dall'ambiente di runtime;

- Metodo ottimistico a cui è dedicato il paragrafo successivo. (Utilizzato dal ROOT-Sim).

Un software PDES può anche essere distribuito su più macchine.

1.5 Protocollo di sincronizzazione ottimistico

Un protocollo di sincronizzazione ottimistico è un protocollo nell'ambito dei sistemi paralleli e/o distrubuiti che, contrariamente a quanto accade con i metodi di sincronizzazione usuali, non prevede alcuna operazione bloccante prima dell'esecuzione di una qualsiasi sezione critica (frammento di codice il cui risultato può dipendere da fattori casuali come una determinata sequenza di scheduling); in un protocollo di questo tipo i processi implementano l'idea dell'esecuzione speculativa, infatti, eseguono l'operazione che ritengono giusta indipendentemente dal fatto che essa sia thread-safe o meno. Con un protocollo di questo tipo, infatti, i processi logici eseguono gli eventi e, nel caso in cui, in un dato istante, scoprono la presenza di un

errore, eseguono un'operazione di rollback per tornare indietro all'istante immediatamente precedente all'operazione che ha causato problemi. Possiamo dire, dunque, che questo protocollo implementa l'ottimizzazione dell'esecuzione speculativa.

Nel nostro caso viene utilizzato un protocollo noto come "Time Warp" presentato in [2].

Un sistema di tipo time warp è un sistema nel quale ogni processo ha un proprio orologio virtuale che scandisce il Local Virtual Time (LVT). L'LVT è un' informazione temporale uni-dimensionale nella quale possiamo definire una relazione di ordinamento. Possiamo inoltre definire un evento E come la coppia (x,t) data da una coordinata spaziale x (il processo che lo esegue) ed una coordinata temporale t (il tempo virtuale nel quale l'evento deve essere eseguito). Ogni azione che viene eseguita da un processo deve avvenire a causa di un evento.

I processi comunicano attraverso lo scambio di messaggi. Ogni messaggio ha un header nel quale compaiono, tra le altre cose, il processo mittente, il processo destinatario, il LVT del mittente ed il LVT nel quale il messaggio deve essere ricevuto (e processato) dal destinatario (detti timestamp).

I sistemi time warp devono rispettare le due seguenti regole semantiche:

- In ogni messaggio il timestamp di ricezione deve essere maggiore del timestamp di invio;
- Il timestamp di un messaggio che sta per essere processato da un LP deve essere maggiore o uguale di tutti i timestamp dei messaggi già processati da quel LP.

Queste due regole fondamentali impongono che i messaggi vengano inviati dal mittente con virtual time di invio crescente e vengano processati dal destinatario con timestamp crescente. Tuttavia non possiamo imporre che i vari LVT progrediscano allo stesso ritmo, infatti, una tale imposizione richiederebbe un meccanismo di sincronizzazione che faccia progredire i vari processi logici allo stesso ritmo. Questa constatazione ci porta a capire che non è detto che un processo riceva e processi i messaggi nell'ordine corretto. È infatti possibile che, in un dato VT, un processo riceva un nuovo messaggio con timestamp minore. Questo problema è risolto dal meccanismo del Time Warp che è formato da due parti: il meccanismo di controllo locale ed il meccanismo di controllo globale.

1.5.1 Meccanismo di controllo locale del Time Warp

Ogni processo ha una coda di input nella quale i messaggi sono ordinati in base al timestamp. Il meccanismo di controllo locale del processo che sta eseguendo un evento si preoccupa di far avanzare il proprio LVT con la seguente regola: il LVT di un processo non avanza mai durante l'esecuzione di un evento; esso avanza solo tra due eventi e diventa pari al timestamp del successivo evento nella coda di input. Tuttavia, essendo del tutto scorrelati gli orologi locali del mittente e del destinatario, potrebbe accadere che si riceva, in un dato istante, un messaggio con timestamp inferiore al proprio virtual time. Questo va in contrasto con le regole semantiche e l'unico modo per porre rimedio è quello di effettuare un rollback, cancellando tutti gli eventuali side-effect intermedi (che potrebbero essere stati

scorretti in quanto potenzialmente dipendenti dal messaggio arrivato in ritardo) e riprendendo l'esecuzione con lo stato relativo al LVT immediatamente antecedente al messaggio che ha causato il rollback. A questo punto, potremo processare questo messaggio nell'istante esatto e proseguire l'esecuzione. Per annullare i side-effect intermedi bisogna comunicare ai processi logici che hanno ricevuto messaggi con timestamp di invio maggiore del nuovo LVT che questi non sono più validi. Questo può essere ottenuto con l'aggiunta di una nuova coda: la coda dei messaggi inviati. In questa coda viene inserita una copia di ogni messaggio inviato; questa è uguale al messaggio originale tranne che per una differenza nell'header ove inseriamo un campo "segno" che può assumere i valori "positivo" e "negativo". Quando un messaggio viene inviato, esso ha segno positivo e viene inserita nella coda dei messaggi inviati una copia con segno negativo (chiamata antimessaggio). Quando viene eseguita una operazione di rollback al tempo t_0, tutti i messaggi che si trovano in questa coda con tempo di invio maggiore di t_0 vengono inviati. Il destinatario, alla ricezione di un antimessaggio, può trovarsi in due situazioni:

- Il messaggio positivo corrispondente non è stato ancora processato (e quindi ha $LVT < timestamp$): in questo caso può cancellare entrambi i messaggi (processo noto come annichilimento) senza problemi in quanto non ci sono stati side-effect;
- Il messaggio positivo corrispondente è stato già processato ($LVT \geq timestamp$): in questo caso, dopo l'annichilimento, sarà necessaria un'ulteriore operazione di rollback per tornare al VT del timestamp dell'antimessaggio (questa situazione può iterarsi ancora e ci può essere una "cascata di rollback").

Un processo in definitiva è composto da:

- una coordinata spaziale, unica nel sistema, che lo identifica;
- un local virtual clock;
- uno stato che contiene informazioni sul contesto di esecuzione;
- una coda di stati necessaria per il rollback;
- una coda di messaggi in arrivo, ordinata in base al timestamp;
- una coda di messaggi inviati, ordinata in base al VT di invio, necessaria per inviare gli antimessaggi.

1.5.2 Meccanismo di controllo globale del Time Warp

Oltre al Local Virtual Time (LVT), definito per ogni processo, esiste il concetto di Global Virtual Time (GVT). Il GVT è un tempo che, a differenza del LVT, non decresce mai. Il GVT può essere visto come una soglia ed è definito come il minimo tra:

- il LVT di tutti i processi;
- il LVT di invio di tutti i messaggi che non sono stati ancora processati.

Dato che un LP non invia mai eventi nel passato (a causa delle regole semantiche del Time Warp) tutti i messaggi presenti nella output queue, tutti i messaggi presenti nella input queue e tutti gli stati salvati con tempo inferiore al GVT possono essere eliminati in quanto nessun rollback potrà portare il LVT ad un tempo antecedente il GVT. Escludendo la possibilità di rollback fino ad un dato LVT i processi possono anche svolgere delle operazioni che comportano side-effect irreversibili (come, ad esempio, la stampa su schermo). Il GVT garantisce quindi che il sistema faccia progressi.

Il meccanismo di controllo globale si preoccupa di calcolare il GVT e di comunicarlo a tutti i processi logici. Oltre a questo il meccanismo di controllo globale si preoccupa di:

- Controllo del flusso;
- Individuazione di una corretta terminazione;
- Gestione degli errori run-time.

Capitolo 2

Descrizione del problema e soluzioni

Una gestione efficiente dei buffer contenenti i messaggi e della logica associata alla loro consegna richiede che vengano gestiti, allo stesso tempo, i seguenti aspetti:

- Possibilità di inviare messaggi di dimensione variabile;
- Evitare la copia e lo spostamento dei messaggi in memoria;
- Avere un accesso rapido al messaggio.

Nella ricerca di una soluzione a questi problemi bisogna tenere presente la particolare architettura nella quale stiamo lavorando. Innanzitutto facciamo un'osservazione: il mittente di un messaggio lavorerà con esso sicuramente una sola volta (quando lo crea); il destinatario, invece, può accedervi più volte: in caso di rollback, infatti, potrebbe essere necessario riaccedere ad un messaggio che era già stato processato. Data la particolare architettura nella quale ci troviamo e, data la relativa lentezza che si ha nell'accedere in un nodo NUMA remoto, risulta chiaro che bisogna inserire il messaggio nel nodo del destinatario in quanto il numero di accessi che esso vi compierà sarà maggiore o uguale di quelli che compierà il mittente.

Ogni processo logico sarà quindi dotato di un "ingoing buffer" nel quale i messaggi saranno depositati dal mittente al momento della loro creazione. Questa struttura conterrà al suo interno un buffer allocato dinamicamente e sarà prevista la possibilità di riallocarlo. Infatti, nel caso in cui si dovesse riempire, sarà possibile copiarlo in una nuova area di memoria di dimensione almeno doppia.

In ROOT-Sim ogni worker thread lavora con gli header dei messaggi ed arriva a leggere il relativo payload solo nel momento del processamento. A causa del buffer rilocabile non possiamo inserire nell'header un puntatore al payload in quanto il messaggio non si trova in una posizione stabile. Risolviamo questo problema utilizzando gli offset a partire dalla base del buffer; così facendo, il messaggio sarà sempre raggiungibile in maniera corretta e senza ulteriore overhead, anche in caso di riallocazione.

Come detto, l'ingoing buffer sarà formato da uno spazio libero allocato dinamicamente in memoria. L'accesso a questa struttura sarà gestita da un allocatore di memoria. Un allocatore di memoria efficiente deve:

1. Ridurre al minimo la frammentazione interna ed esterna;
2. Avere poco overhead;
3. Essere in grado di effettuare allocazioni e deallocazioni nel modo più rapido possibile.

Molto spesso i punti 2 e 3 sono in contrasto tra loro in quanto l'aggiunta di metadati può portare ad allocazioni più semplici. Bisogna cercare un compromesso.

Frammentazione interna

Si ha frammentazione interna quando viene allocato un blocco di dimensione maggiore del necessario. Il problema della frammentazione interna si ha quando la memoria viene partizionata in blocchi, infatti, in questo caso, un'allocazione deve prendere un intero blocco, anche nel caso in cui richieda uno spazio di dimensione molto minore! Il problema della frammentazione interna può essere alleviato ricorrendo a blocchi di dimensione variabili e soddisfacendo la richiesta di allocazione utilizzando il blocco di dimensione minore. Il problema della frammentazione interna, nella presente proposta, è risolto attraverso l'operazione di splitting che verrà discussa nel capitolo successivo.

Frammentazione esterna

Si ha frammentazione esterna quando i blocchi di memoria diventano troppo piccoli per poter soddisfare una richiesta. In questo caso può succedere che, alla richiesta di X Byte, seppur questa quantità di memoria sia disponibile, l'allocazione debba fallire in quanto gli X Byte non sono contigui. Nell'esempio, sono disponibili 9 Byte ma l'allocazione di 6 Byte non è possibile. Il problema della frammentazione

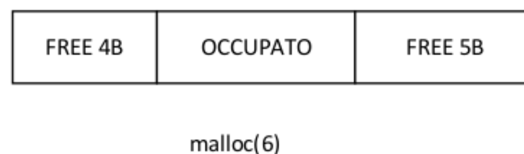

Figura 2.1. Esempio di frammentazione esterna

esterna può essere risolto attraverso la ricompattazione: periodicamente i blocchi vengono spostati per essere posizionati in memoria contigua. Dopo questa operazione abbiamo un unico grande blocco libero.

L'operazione di ricompattazione ha costo lineare. Tuttavia le memorie possono essere molto grandi e quindi un'operazione a costo O(n) risulta essere molto costosa. Per cercare di ritardare il più possibile quest'operazione sono state formulate le seguenti politiche di allocazione della memoria:

- First Fit: viene allocato il primo blocco libero in grado di accogliere la richiesta;
- Best Fit: viene allocato il blocco libero di dimensione minima in grado di accogliere la richiesta;

- Worst Firt: viene allocato il blocco libero di dimensione maggiore.

Osserviamo che, nel caso peggiore, gli algoritmi sono tutti a costo O(n) ove n è il numero di blocchi liberi. Tuttavia gli algoritmi Best Fit e Worst fit devono sempre scorrere tutta la memoria e quindi risultano essere lineari in tutti i casi (non c'è un caso migliore); questo non è vero per il First Fit. Dati sperimentali dimostrano che le politiche di tipo First Fit e Best Fit si comportano in maniera migliore rispetto al Worst Fit. Nel nostro caso, come verrà discusso nel prossimo capitolo, utilizzeremo una politica di tipo First Fit e prevederemo la possibilità di unire due blocchi liberi adiacenti (operazione di coalescing), in modo di avere blocchi liberi di dimensione maggiore (evitando la ricompattazione!).

Capitolo 3

Architettura di riferimento e Implementazione

3.1 API NUMA

Come già accennato l'efficienza di una macchina di tipo NUMA è fortemente caratterizzata da quanti accessi avvengono nella memoria remota. Infatti, accedere ad una memoria remota degrada le prestazioni in quanto:

- Viene effettuata una copia dei dati (puro overhead);
- Viene utilizzato un protocollo di message passing che, oltre ad invalidare righe di cache, non offre una garanzia sul numero di hop necessari per accedere ad un dato indirizzo in memoria remota;
- Viene utilizzato un bus comune a tutti i nodi e quindi c'è il rischio di dover attendere la disponibilità.

Le politiche NUMA sono le varie scelte che vengono effettuate per rendere l'accesso alla memoria di ogni thread efficiente (si cerca quindi di rendere l'accesso il più frequentemente possibile in memoria locale). Una API NUMA permette di decidere quale nodo (e quindi quale core e quale area di memoria) debba essere utilizzato da ogni thread ed è formata da tre parti:

- Una parte core che è formata da cinque nuove system call: *get_mempolicy()* che restituisce la politica NUMA impostata per il thread chiamante o per un certo indirizzo di memoria (dipende da un flag passato come parametro); *mbind()* che imposta la politica NUMA per un intervallo di memoria; *set_mempolicy()* che imposta la politica NUMA per il thread chiamante e, a seconda dei parametri, anche per i thread figli; *migrate_page()* che sposta tutte le pagine di un processo in un certo insieme di nodi; *move_pages()* che sposta le pagine specificate in un certo nodo;
- Una libreria: *libnuma* che può essere linkata alle applicazioni e che offre un'interfaccia più semplice ed astratta rispetto alle system call;
- Delle utility che permettono il controllo ed il profiling delle prestazioni. Ad esempio, l'utility *numastat* restituisce statistiche sull'utilizzo della memoria.

Nome	Descrizione
default	alloca nel nodo locale
bind	alloca in uno specifico insieme di nodi (se non c'è memoria disponibile in quei nodi, fallisce).
interleave	alterna l'allocazione in un insieme di nodi
preferred	prova ad allocare in un insieme di nodi, se non ci riesce, alloca in un altro nodo

Tabella 3.1. Politiche NUMA

Le possibili politiche NUMA sono descritte nella Tabella 3.1.

Le politiche possono essere per processo o per regione di memoria. Le politiche per processo sono applicate a tutte le allocazioni di memoria fatte nell'ambito del processo. Le politiche per regione di memoria vengono invece applicate a singole allocazioni. Queste ultime hanno una priorità maggiore rispetto alle prime nel senso che, se per un processo è impostata una determinata politica e, in una certa allocazione, è specificata un'altra politica, sarà utilizzata quest'ultima.

3.1.1 libnuma

Per poter utilizzare le funzioni della API NUMA bisogna includere nel programma <numa.h> ed eseguire il linking in fase di compilazione con lo switch "-lnuma". All'avvio del programma bisogna sfruttare la prima funzione messa a disposizione: *numa_available()*; questa funzione ritorna un numero negativo se il sistema non supporta nessuna politica NUMA. In questo caso l'esecuzione va interrotta in quanto il comportamento che si ha con l'utilizzo di funzioni della API NUMA in un sistema non compatibile non è definito. Con *numa_max_node()* possiamo sapere il numero di nodi del sistema.

La libreria spesso fa uso di un tipo di dato *nodemask_t* che non è nient altro che una maschera di numa_max_node() bit. Ad ogni bit corrisponde un nodo e possiamo modificare il valore con le funzioni: *nodemask_zero()* che azzera tutti i bit, *nodemask_set()* che imposta il bit passato come parametro ad 1, *nodemask_clr()* che setta il bit passato come parametro a 0. Possiamo sapere se un bit è settato ad 1 con la funzione *nodemask_isset()*.

L'allocazione di memoria ha granularità di una pagina e può essere fatta con le funzioni: *numa_alloc_onnode()* che prova ad allocare la memoria nel nodo specificato e, in caso di insuccesso, prova in un altro nodo. Tuttavia, se precedentemente fosse stata chiamata la funzione *numa_set_strict()* la *numa_alloc_onnode()*, in caso di insuccesso, dopo aver provato a liberare memoria facendo swapping senza risultati, fallisce. Con la *numa_alloc_interleaved()* l'allocazione è fatta in maniera Round Robin su tutti i nodi mentre con la *numa_alloc_interleaved_subset()* questa operazione è limitata ad un certo sottoinsieme di nodi. Con la *numa_alloc_local()* l'allocazione è fatta sul nodo in cui si trova il chiamante e, infine, con la *numa_alloc()* l'allocazione è fatta seguendo la politica scelta per il processo. Questa politica può essere cambiata con le seguenti funzioni:

- *numa_set_interleave_mask()* che abilita l'interleaving per i nodi setatti ad 1 nella maschera passata come parametro;
- *numa_set_prefered()* che setta come nodo preferito per il chiamante quello passato come parametro;
- *numa_set_membind()* che setta come unico nodo utilizzabile dal thread chiamante quello passato come parametro;
- *numa_set_localalloc()* che setta come nodo utilizzabile per l'allocazione il nodo locale al thread chiamante.

In ogni caso, con qualsiasi funzione della famiglia *numa_alloc_**, la memoria va liberata utilizzando la funzione *numa_free()*.

Per far girare un thread in uno specifico nodo possiamo utilizzare la funzione *numa_run_on_node()* o, per far girare un thread in un sottoinsieme dei nodi (passati come parametro con una nodemask_t), possiamo utilizzare la funzione *numa_run_on_node_mask()*.

3.2 Architettura ROOT-Sim

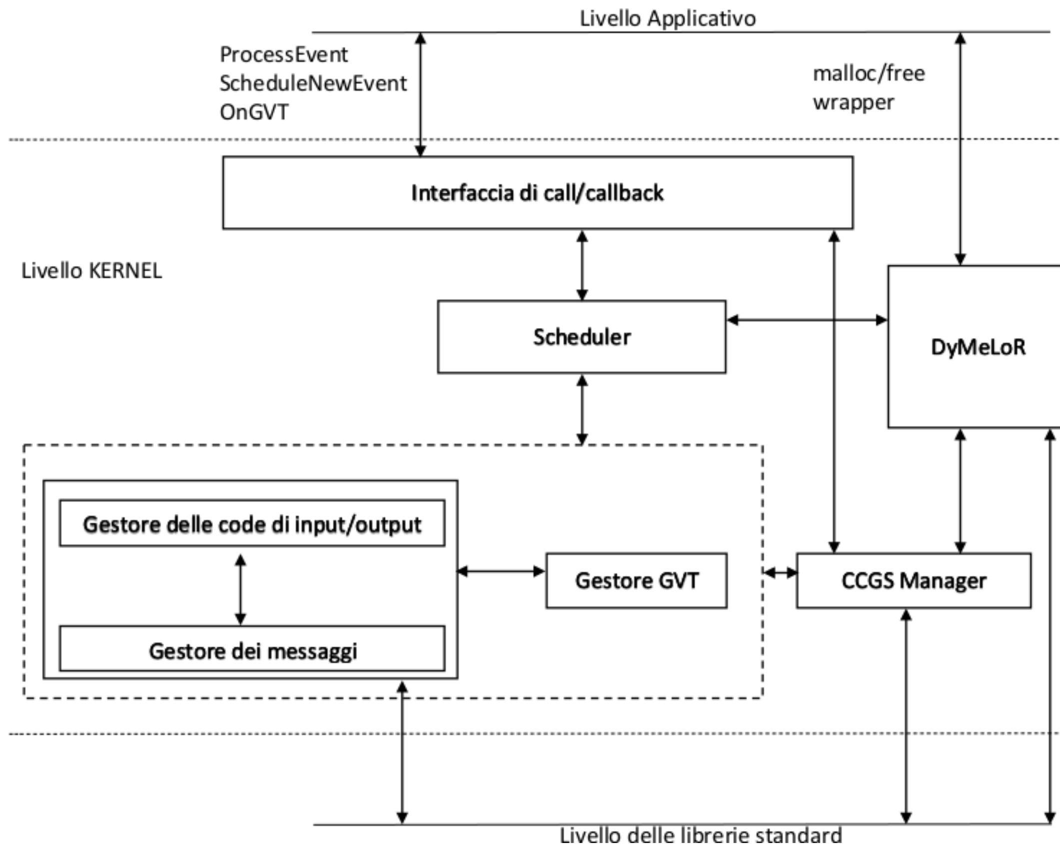

Figura 3.1. Architettura ROOT-Sim

L'architettura del ROOT-Sim è schematizzata in Figura 3.1. Sono presenti tre strati: livello applicativo, livello kernel e livello delle librerie standard. Come per ogni architettura a strati ogni livello fornisce servizi al livello superiore e ne usufruisce dal livello inferiore.

3.2.1 Livello applicativo

Il livello applicativo è il livello formato dal codice (scritto dall'utilizzatore del simulatore) che modella il fenomeno che si vuole simulare. Questo modello comunica con il livello kernel attraverso tre funzioni; due delle quali sono chiamate dal simulatore e quindi sono due funzioni di callback:

- void ProcessEvent(int me, time_type now, int event_type, void* content, int size, void* state): funzione di callback che deve essere obbligatoriamente implementata. Con questa funzione il kernel comunica all'applicazione di dover eseguire un particolare evento. Per quanto riguarda i parametri: "me" è l'identificatore del LP ricevente, "now" è il VT dell'evento, "event_type" è il tipo di evento (il significato di questo parametro dipende dall'applicazione), "content" è puntatore al payload del messaggio e "size" la sua dimensione, state è il puntatore allo stato corrente del processo logico;

- void ScheduleNewEvent(int where, time_type timestamp, int event_type, void *content, int size): è una funzione chiamata dal processo che vuole inviare un nuovo messaggio al LP indicato da "where". Questo messaggio dovrà essere eseguito al VT "timestamp". Gli altri parametri hanno significatro identico a ProcessEvent;

- bool OnGVT(unsigned int me, lp_state_type *snapshot): funzione di callback con la quale il simulatore avvisa gli LP che è stato raggiunto un nuovo GVT. Anche questa funzione deve essere obbligatoriamente implementata. Il parametro snapshot è un puntatore ad uno stato: lo stato correntemente raggiunto dall'esecuzione. Il processo logico può, in questa fase: eliminare tutti i messaggi e gli stati con timestamp minore del GVT; controllare lo stato e decidere se l'obbiettivo è stato raggiunto e quindi terminare.

3.2.2 Livello kernel

Il livello kernel è il cuore di una singola istanza del simulatore. Essendo la piattaforma parallela ci possono essere più kernel; in tal caso uno di questi assume il ruolo di master kernel e coordina tutti gli altri (slave kernel). Un esempio di sistema Parallel Discrete Event Simulation è mostrato in Figura 3.2.

Il livello kernel è formato dai seguenti sottosistemi:

- Sottosistema di gestione degli eventi: è formato dalle code di input e di ouput. In queste code i messaggi, che arrivano in ordine casuale, vengono riordinati per timestamp e processati. I messaggi vengono mantenuti finchè il loro timestamp non diventa minore del GVT; questo viene fatto per supportare l'operazione di rollback. Se viene ricevuto un messaggio con timestamp minore del LVT questo sottosistema avvia la procedura di rollback;

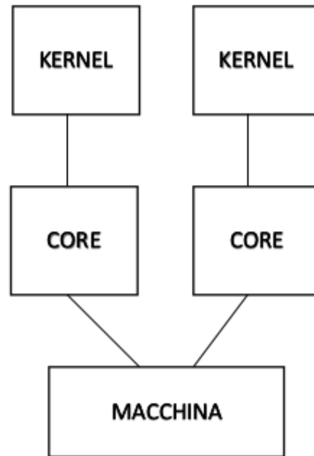

Figura 3.2. Esempio di sistema PDES

- Sottosistema per lo scheduling: in ogni core troviamo in esecuzione più processi logici. Ogni core ha il suo scheduler e lo scheduling è basato sull'interleaving. Il processo logico scelto per l'esecuzione è quello che deve processare il messaggio con VT minore (Lowest Timestamp-First, LTF);

- Sottosistema per la stima del GVT: il calcolo del GVT è basato su un algoritmo totalmente wait-free; (si veda [5]);

- Dynamic Memory Logger and Restore (DyMeLoR): è un allocatore di memoria orientato alla simulazione ad eventi discreti,che si basa su un wrapper delle funzioni malloc/free. Nel caso in cui venga chiamata una malloc o una free per il salvataggio o il ripristino di uno stato, infatti, viene utilizzato questo allocatore (in maniera del tutto trasparente). DyMeLoR incapsula lo stato in dei metadati che torneranno utili in caso di ripristino e, quando questo lavoro è compiuto, utilizza la libreria di allocazione standard.

3.3 Implementazione

3.3.1 Scambio dei messaggi

Un messaggio del simulatore è identificato dalla seguente struttura (con cui è stato definito il tipo msg_t):

Codice 3.1. struct _msg_t

```
1 typedef struct _msg_t {
2         unsigned int            sender;
3         unsigned int            receiver;
4         int                     type;
5         simtime_t               timestamp;
6         simtime_t               send_time;
7         message_kind_t          message_kind;
8         unsigned long long      mark;
```

```
9         unsigned                payload_offset;
10        int                     size;
11 } msg_t;
```

Abbiamo i seguenti campi:

- sender: identificatore del processo logico mittente;
- receiver: identificatore del processo logico ricevente;
- type: tipo del messaggio. Il significato del valore di questo campo è legato al particolare modello che sta utilizzando il simulatore;
- timestamp: tempo in cui il messaggio deve essere processato dal ricevente;
- send_time: tempo in cui il messaggio è stato inviato dal mittente (nel LVT del mittente);
- message_kind: messaggio positivo o negativo;
- mark: identificatore univoco del messaggio, utilizzato per la gestione degli antimessaggi;
- payload_offset: offset del messaggio nel buffer del processo destinatario (sarà discusso in seguito);
- size: dimensione del payload.

Lo scambio di un messaggio è un momento critico in quanto un invio lento potrebbe portare il LVT del destinatario (che intanto sta processando altri messaggi) a crescere troppo e, di conseguenza, causare un rollback appena il messaggio viene consegnato. Dunque, per ridurre le probabilità di rollback, abbiamo bisogno di un invio che sia il più rapido possibile. Per risolvere questo problema si ricorre ad un meccanismo ampiamente utilizzato nell'ambito dei device driver: si divide la routine da eseguire in due pezzi che sono la top half e la bottom half.

Top Half

La top half è la parte della funzione che viene eseguita in un contesto che è tale da richiedere un'esecuzione rapidissima. In questa fase si compiono le operazioni minime necessarie per l'esecuzione della funzione nella sua totalità. In particolare vengono effettuate le seguenti operazioni:

- Copia del messaggio nel nodo NUMA destinatario;
- Creazione dell'header (struttura msg_t) e inserimento in essa dei metadati relativi al messaggio da inviare;
- Inserimento dell'header in una coda di bottom half del destinatario.

Bottom Half

La fase di bottom half è, al contrario, una fase che può richiedere un tempo non minimale e che viene eseguita in un momento non critico dell'esecuzione: nel nostro caso la bottom half è eseguita dal thread destinatario ed è la fase nella quale gli header dei messaggi (che si trovano nella coda delle bottom half) vengono inseriti nella coda di input, ordinati in base ai propri timestamp in modo da poter permettere il processamento secondo l'ordine temporale corretto. Questa fase non è critica in quanto, l'LP destinatario non sta processando messaggi e, quindi, il LVT rimane fermo.

3.3.2 Ingoing Buffer

L'ingoing buffer è implementato da una struct _ingoing_buffer con cui è stato definito un tipo ingoing_buffer:

Codice 3.2. struct _ingoing_buffer

```
1 typedef struct _ingoing_buffer{
2         unsigned        first_free;
3         unsigned        size;
4         unsigned        extra_buffer_size_in_use;
5         spinlock_t      lock;
6         void*           extra_buffer[EXTRA_BUFFER_SIZE];
7         void*           base;
8 }ingoing_buffer;
```

Come già anticipato, per rendere il buffer rilocabile, utilizziamo per l'accesso l'offset rispetto la base. Inoltre, per far si che i messaggi possano essere consegnati anche nel caso in cui il buffer sia saturo (e quindi in attesa di riallocazione) prevediamo la pesenza di memoria ausiliaria. Le componenti dell'ingoing_buffer sono:

- first_free: offset del primo blocco attualmente libero;
- size: dimensione (espressa in Byte) del buffer;
- extra_buffer: un array di puntatori void*. Questi puntatori si riferiscono ad aree di memoria che vengono allocate dal mittente nel nodo NUMA del destinatario (con una numa_alloc_onnode()) quando vuole consegnare un messaggio ma non trova spazio disponibile nel buffer;
- extra_buffer_size_in_use: dimensione (espressa in Byte) di eventuali blocchi di memoria che stanno usando memoria ausiliaria;
- lock: il lock si rende necessario in quanto le operazioni sul buffer sono sezioni critiche e devono essere eseguite in mutua esclusione;
- base: è il puntatore al primo byte del buffer e viene usato come indice per lo spiazzamento in tutte le operazioni.

L'allocatore che useremo per la gestione di questo buffer è basato su lista dei blocchi liberi esplicita. La lista dei blocchi liberi non ha bisogno di overhead aggiuntivo.

Infatti, osserviamo che questa informazione è necessaria solo per i blocchi che non sono utilizzati e possiamo sfruttare lo spazio dedicato al payload. Il blocco, inoltre, non ha una struct che lo definisce ed è individuabile all'interno dell'array void* conoscendo le politiche con le quali esso viene gestito. L'allocatore è in grado di eseguire operazioni di splitting e coalescing.

3.3.3 Anatomia di un blocco di ingoing buffer

Distinguiamo due tipi di blocco: il blocco libero ed il blocco in uso.

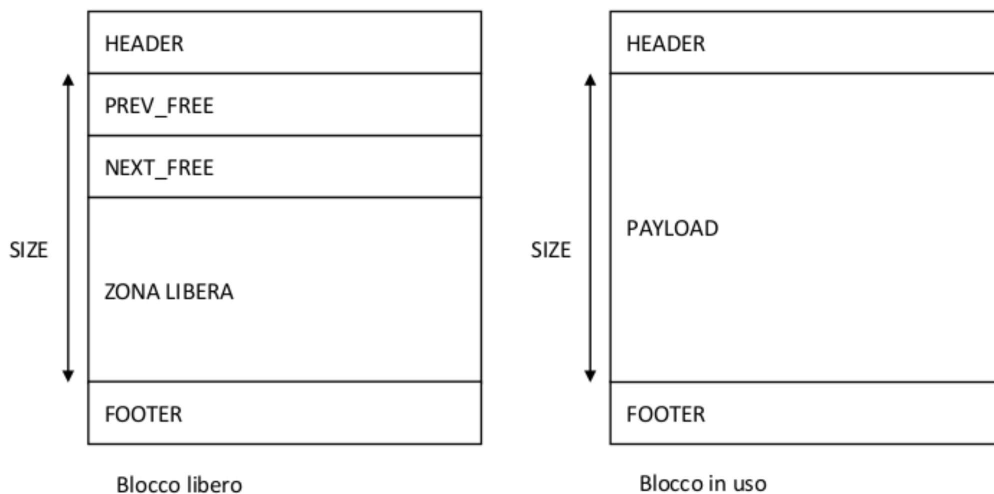

Figura 3.3. Anatomia di un blocco di ingoing buffer

Abbiamo i seguenti campi:

- Header: nell'header è scritta la dimensione. La dimensione è un unsigned soggetto ad un'operazione di bit stealing. Il Most Significant Bit (MSB) viene utilizzato come flag: MSB è impostato ad 1 se il blocco è in uso, 0 altrimenti. Inoltre, si ha che la size (come riportato nell'immagine) è al netto di header e footer, ossia è la dimensione che può essere utilizzata realmente per il payload;

- prev_free: anch'esso è un unsigned ed indica l'offset del precedente blocco libero nella free list;

- succ_free: come prev_free ma indica l'offset del successivo blocco libero nella free list;

- zona libera: spazio libero all'interno del blocco;

- payload: spazio occupato dal payload;

- footer: è un boundary tag, copia dell'header, necessario per l'operazione di coalescing effettuata durante la free del blocco.

OSSERVAZIONE: La dimensione minima di un blocco occupato deve essere la dimensione di un blocco libero, infatti, quando il blocco occupato torna ad essere libero abbiamo bisogno dello spazio per reitegrare i "puntatori" per la free list.

3.3.4 Allocazione

L'operazione di allocazione è compiuta secondo una politica First Fit: si scorre la free list fino a trovare un blocco di dimensione sufficiente. L'allocazione di memoria ha quindi un costo O(n) ove n è il numero di blocchi liberi. Per ridurre la frammentazione interna, l'allocatore supporta lo splitting, ossia, se un blocco è di dimensione sufficiente da accogliere il payload e, una volta allocato il payload, avanza spazio per un altro blocco, questo viene diviso: una parte del blocco viene allocata e l'altra parte viene utilizzata per creare un nuovo blocco libero che viene aggiunto alla free list. Ci sono quattro scenari possibili che analizziamo. Nella trattazione che segue sarà supposta una richiesta di memoria pari a X Byte.

Allocazione, caso 1

Il primo caso è quello in cui il first_free non esiste. In questo scenario la free_list è vuota e nel buffer non c'è spazio per consegnare il messaggio. In questa situazione, il mittente, con una numa_alloc_onnode() si preoccupa di allocare nel nodo NUMA del destinatario un blocco di memoria sufficiente ad accogliere il payload del messaggio ed inserisce il puntatore nel primo indice libero dell'aray "extra_buffer". Il mittente si preoccupa inoltre di aggiornare la "extra_buffer_size_in_use". Per quanto riguarda l'offset da inserire nel msg_t, questo viene calcolato come la somma della dimensione del buffer più tutte le dimensioni dei blocchi presenti nell'extra buffer (sia come spazio riservato al payload che come spazio riservato ad header e footer). In questo modo, quando l'extra_buffer sarà copiato nel nuovo buffer, gli offset saranno giusti. Nell'esempio mostrato in FIgura 3.4, ricordando che header e footer hanno dimensione pari a sizeof(unsigned), i due messaggi avranno rispettivamente $payload_offset = (5000 + sizeof(unsigned))$ e $payload_offset = (5000 + 40 + 3 * sizeof(unsigned))$.

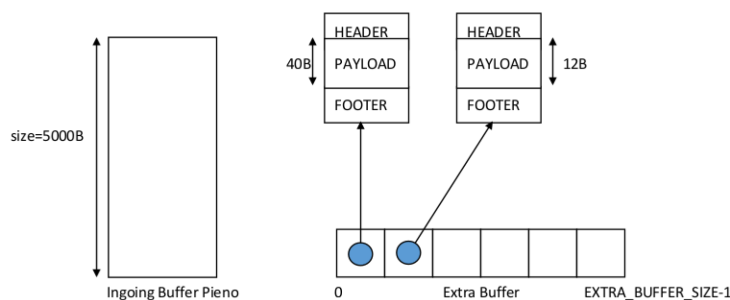

Figura 3.4. Allocazione, utilizzo dell'extra_buffer

Allocazione, caso 2

Il secondo caso è quello nel quale il blocco relativo all'offset "first_free" è di dimensione sufficiente ad accogliere il payload ed avanza spazio per un altro blocco. In questo caso effettuiamo l'operazione di splitting ed aggiungiamo il nuovo blocco in testa alla free_list. (Figura 3.5)

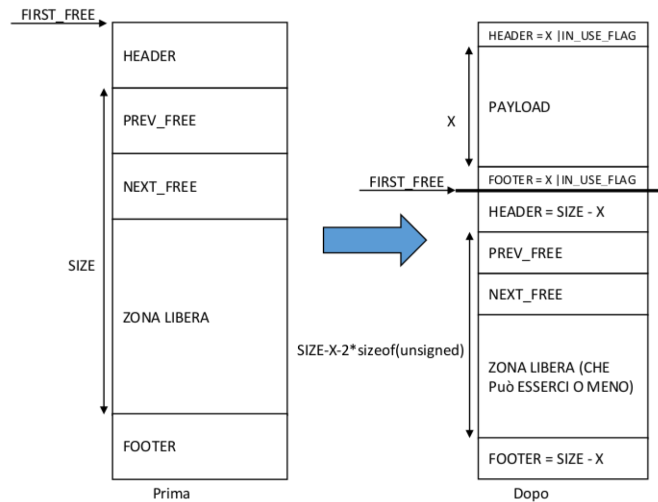

Figura 3.5. Allocazione, caso 2

Allocazione, caso 3

Il terzo caso è quello in cui il blocco identificato dal first_free può accogliere il payload ma lo spazio che avanza dopo l'allocazione è più piccolo della dimensione minima (oppure non esiste proprio). In questo caso allochiamo tutto il blocco. (Nell esempio inizialmente il first free puntava al blocco libero successivo (con indice k)). (Figura 3.6)

Allocazione, Caso 4

Nel quarto ed ultimo caso ci troviamo nella situazione in cui il first_free esiste ma non è grande a sufficienza da poter contenere il payload della dimensione richiesta. In questo casoo scorriamo la lista fino a trovare un blocco di dimensione sufficiente. Una volta trovato il blocco effettuiamo l'operazione di splitting ed adeguiamo la free list: se l'operazione riesce il nuovo blocco sarà nella free_list esattamente dove era il blocco che abbiamo occupato; se lo splitting non è possibile elimiano tutto il blocco dalla free list e adeguiamo i puntatori. Nel caso in cui il mittente scorra tutta la free list senza trovare un blocco di dimensione sufficiente ci ritroviamo nel primo caso.

3.3.5 Deallocazione

L'operazione di deallocazione è compiuta secondo una politica di tipo LIFO: un blocco usato che torna ad essere libero viene inserito in testa alla free list. La

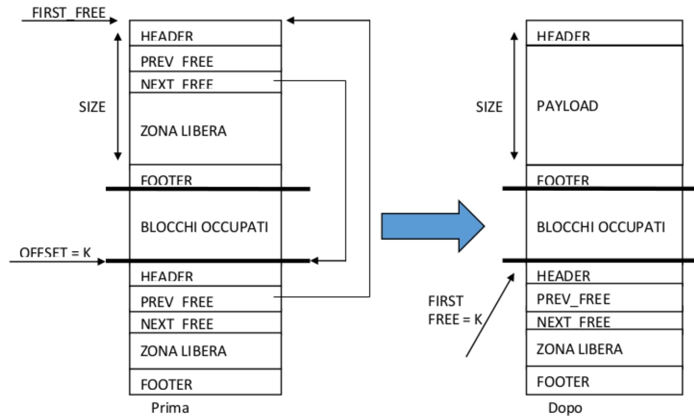

Figura 3.6. Allocazione, caso 3

deallocazione ha quindi costo O(1). Inoltre, per ridurre la frammentazione esterna la deallocazione supporta il coalescing: quando un blocco viene liberato, vengono controllati l'header del blocco adiacente in memoria successivo ed il footer del blocco precedente; nel caso in cui si tratti di blocchi liberi questi vengono fusi in un unico blocco di dimensione maggiore.

Deallocazione, caso 1

Nel primo caso abbiamo la situazione più semplice da trattare: il blocco che si è appena liberato ha solo blocchi occupati adiacenti. In questo caso le uniche operazioni da fare sono l'inserimento di questo blocco nella free list e l'aggiornamento dell'header e del footer per segnalare che il blocco non è più in uso. (Figura 3.7)

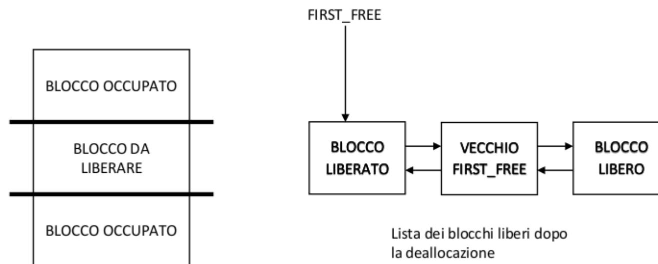

Figura 3.7. Deallocazione, caso 1

Deallocazione, caso 2

Nel secondo caso abbiamo il blocco successivo che è già presente nella free list. In questo caso le operazioni da svolgere sono: (Figura 3.8)

- eliminazione dalla free list del blocco successivo;

- fusione dei due blocchi: aggiornare l'header con la dimensione che è somma delle dimensioni dei due blocchi (più lo spazio per un header ed un footer, in quanto abbiamo due dimensioni al netto dell'overhead ed ora avremo bisogno di solo due blocchi);
- aggiunta del nuovo blocco nella free list;

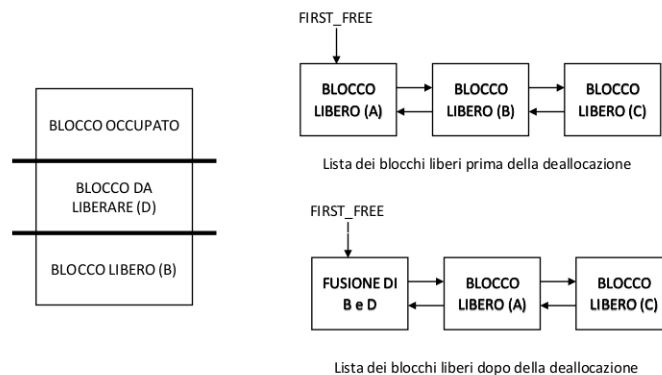

Figura 3.8. Deallocazione, caso 2

Deallocazione, caso 3

Il terzo caso è identico al precedente con l'unica differenza che, questa volta, il blocco libero è quello precedente. (Figura 3.9)

Deallocazione, caso 4

Nel quarto ed ultimo caso abbiamo che il blocco si trova tra due blocchi liberi. In questa situazione vanno eliminati entrambi i blocchi dalla free list e poi, al solito, bisogna riaggiungere il nuovo blocco (questa volta fusione di 3 blocchi) nella free list. (Figura 3.9)

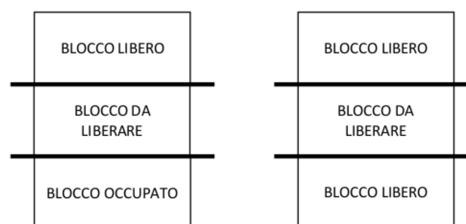

Figura 3.9. Deallocazione, caso 3 e 4

3.3.6 Riallocazione

Ogni volta che un LP si trova a dover processare un messaggio, controlla il relativo payload_offset. Se questo è maggiore o uguale della dimensione dell'ingoing_buffer,

implicitamente significa che il messaggio si trova nell'extra buffer. A questo punto il destinatario alloca un nuovo buffer di dimensione almeno pari a: vecchia dimensione + extra_buffer_in_use. Tuttavia viene fatto l'arrotondamento alla potenza di due superiore per evitare di creare un buffer che sia subito saturo nuovamente. Una volta allocato il buffer, l'LP, effettua la copia del buffer precedente e lo dealloca. Inoltre, per ogni puntatore dell'extra_buffer, si preoccupa di:

- Copiare il contenuto nel nuovo buffer (tenendo presente che gli offset e sono già corretti);
- Effettuare la numa_free().

Ovviamente questa è un'operazione molto dispendiosa di risorse e, inoltre, essendo effettuata in sezione critica, blocca l'accesso all'ingoing buffer del processo logico per un tempo non minimale. Per questo motivo questa operazione deve essere effettuata un numero limitato di volte e la dimensione iniziale del buffer va scelta tenendo presente questo problema.

3.4 Sviluppi futuri

Una possibile ottimizzazione è l'eliminazione del lock. Infatti in un progetto così ampiamente parallelo fare busy waiting su un lock non è ottimale. Tuttavia l'occupazione di un blocco, sia per utilizzarlo, sia per fonderlo con un altro blocco e la manipolazione della free list sono operazioni che producono una sezione critica. Non si può, quindi, pensare di poter risolvere questo problema senza alcun tipo di sincronizzazione.

L'algoritmo a cui sto lavorando ha bisogno del Less Sigificant Bit (LSB) del prev_free e del next_free del blocco libero. Per essere sicuro che questo bit possa essere utilizzato devo fare in modo che gli offset siano sempre pari (e che quindi abbiano l'LSB sempre uguale a 0). Questo è facilmente ottenibile in quanto la dimensione dell'ingoing buffer è sempre una potenza di due; gli header e i footer sono di dimensione sizeof(unsigned) e quindi sono anch'essi pari; basta quindi allocare blocchi di dimensioni sempre pari: se un blocco dovesse avere dimensioni dispari sprecheremo un Byte di memoria per l'allineamento.

Introduciamo all'inizio della free_list un blocco fantoccio, che non potrà mai contenere payload, che chiameremo "blocco Dummy". L'header di questo blocco sarà sempre settato IN_USE in quanto non vogliamo che sia oggetto di coalescing. Il prev_free sarà settato ad un valore che indica l'inesistenza dello stesso ed il next_free sarà il primo vero blocco libero. In fase di allocazione:

1. Partendo dal blocco dummy, cercheremo di settare atomicamente ad 1 il LSB nel next_free con un'operazione di atomic_test_and_set. Itereremo questa operazione finchè non avrà risultato positivo. Notiamo che se qualcuno dovesse prendere possesso del blocco etichettato come next, potrà cambiare il valore del blocco che noi stiamo esaminando senza alcun problema, infatti, in questa fase, non ci interessa un particolare blocco ma ci interessa solo prendere possesso del puntatore al successivo;

2. Quando questa operazione sarà riuscita, nessun altro potrà avanzare in questo blocco. Tuttavia, in un operazione di coalescing, qualche LP potrebbe cercare di ottenere questo blocco. Itereremo aspettando di riuscire a settare, anche qui atomicamente, il MSB dell'header e del footer del blocco che stiamo scandendo;

3. Settiamo il LSB del prev_free del successivo blocco libero del blocco selezionato ad 1;

4. In questo modo il blocco che voglio utilizzare è completamente isolato. Controlliamo la dimensione e, se ci interessa, lo eliminiamo dalla free list. Se non ci dovesse interessare bloccheremo il puntatore next del blocco attuale, resetteremo tutto quello che abbiamo settato nei punti 1,2,3 e itereremo proseguendo la nostra ricerca.

La fase di coalescing sarà una fase che avrà precedenza inferiore rispetto l'allocazione (bisogna evitare il deadlock, quindi una delle due operazioni deve essere sfavorita) e sarà svolta così:

1. Provo a bloccare gli header dei blocchi adiacenti a quello che sto liberando;

2. Se non ci riesco, lascio stare. Se ci riesco, provo a bloccare i link ad esso entranti nella free list. Se non ci riesco significa che qualcuno sta provando ad allocare quel blocco; sblocco tutto quello che avevo bloccato e rinuncio;

3. Se invece ci sono riuscito, il blocco sarà completamente isolato e potrò procedere.

Quando useremo l'extra_buffer compieremo una operazione atomica di CompareAndSwap sull'array di puntatori, ossia, atomicamente: se l'indice i-esimo punta a NULL, sostituisci il valore con il puntatore che ho ottenuto con la numa_alloc(); se non ci riesci, effettua questa operazione sul successivo. Implementando queste modifiche il lock sarà necessario solo quando l'LP destinatario avvierà la riallocazione del buffer. In questa fase, il destinatario si dovrà accertare che nessun processo logico stia utilizzando il suo buffer e, una volta che questa condizione è soddisfatta, potrà procedere con la riallocazione.

È importante notare che con queste modifiche potremo continuare ad utilizzare tutti gli algoritmi qui presentati.

Capitolo 4

Dati Sperimentali

La simulazione è stata svolta su una macchina dotata di CPU AMD con architettura x84_64 dotata di 24 core operanti ad una frequenza di 2.2GHz. La macchina è dotata di 32GB di RAM e 4 nodi NUMA.

Per la simulazione si è scelto di utilizzare il benchmark PCS (Personal Communication System). PCS simula l'evoluzione di un sistema di comunicazione di tipo GSM. Nel modello, ogni core gestisce un thread; ogni processo logico rappresenta una cella e mantiene una lista di record ove ogni record rappresenta una chiamata. Quando viene avviata una nuova chiamata, il processo logico interessato alloca dinamicamente un nuovo record e lo inserisce nella lista (questo record esisterà finchè la chiamata sarà terminata o sarà trasferita su un'altra cella con un hand-off). Contestualmente all'inizio della nuova comunicazione la lista dei record viene scandita per calcolare il Signal to Interference Ratio (SIR) e assegnare, alla chiamata, la potenza minima necessaria per contrastare le interferenze. Questo benchmark è stato scelto in quanto garantisce un carico costante e uniforme tra i vari processi logici. Per approfondimenti si faccia riferimento a [7].

I dati ottenuti sono riportati nella seguente tabella:

Numero di thread	**Secondi**	**Speedup**	**Efficienza**
2	154	1	0.63
4	56	2.75	0.63
8	25	6.16	0.62
12	22	7	0.53
16	15	10.26	0.53
20	14	11	0.50
24	14	11	0.46

Tabella 4.1. Dati Sperimentali

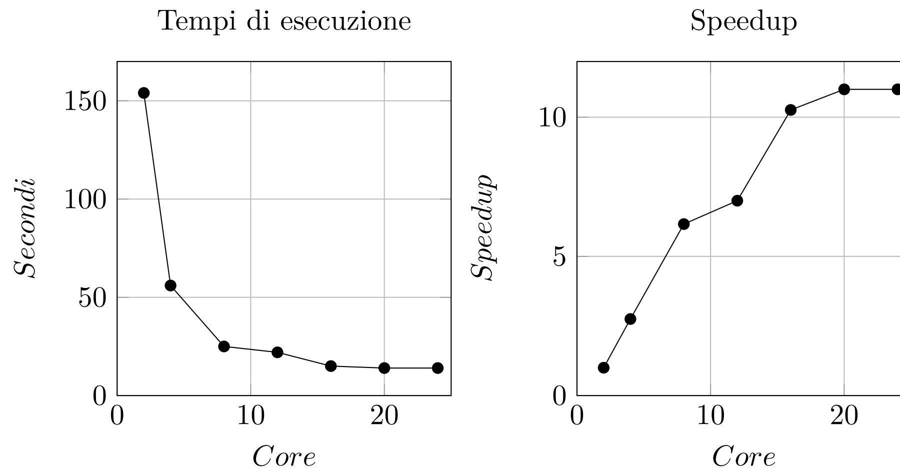

Figura 4.1. Tempo di esecuzione e Speedup

4.1 Tempi di esecuzione

Nei due grafici in Figura 4.1 sono mostrati rispettivamente il variare del tempo di esecuzione all'aumentare dei core e lo speedup. Lo speedup (che rappresenta il fattore con il quale le prestazioni incrementano) nel caso di n core è calcolato come il rapporto tra il tempo di esecuzione con 2 core ed il tempo di esecuzione con n core. Come era facile aspettarsi il tempo di esecuzione tende a diminuire con l'aumentare del grado di parallelismo. Tuttavia, vediamo che l'aumento di prestazioni si affievolisce nella seconda metà dei grafici, fino ad essere piatto nel passaggio tra 20 e 24 core. Questo avviene a causa dell'accesso in sezione critica che diventa sempre più un collo di bottiglia all'aumentare del numero dei core. Possiamo ritenerci soddisfatti in quanto osserviamo che con 20 core riusciamo ad avere uno speedup di 11x rispetto all'esecuzione con 2 core.

4.2 Efficienza

Il parametro efficienza è definito nel modo seguente:

$$efficienza = (1 - frequenza_di_rollback * lunghezza_del_rollback) * 100$$

ove:

$$frequenza_di_rollback = \frac{rollback_totali}{eventi_totali}$$

$$lunghezza_di_rollback = \frac{eventi_totali - eventi_completati}{rollback_totali}$$

Per quanto riguarda questo parametro, notiamo un andamento decrescente. Questo risultato è un risultato prevedibile ed atteso, infatti, con l'aumentare del grado di parallelismo aumenta la probabilità di rollback. Questo accade perchè, sfruttando il protocollo di sincronizzazione ottimistico, i processi avanzano in maniera indipendente: più processi abbiamo, più probabiltà abbiamo che un dato processo sia con un LVT più grande di un altro e, inviandogli un messaggio, causi un rollback. Nel

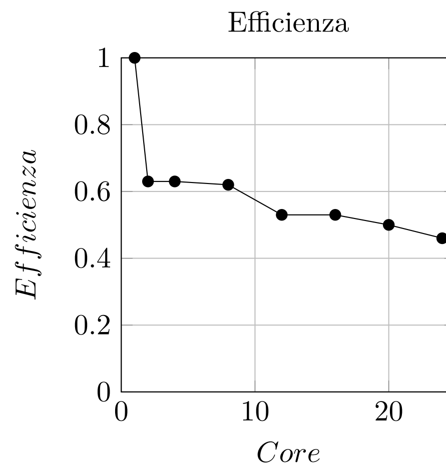

Figura 4.2. Efficienza

caso in cui abbiamo un solo processo ovviamente l'efficienza è pari ad 1 in quanto non è richiesta nessuna sincronizzazione.

Bibliografia

[1] FUJIMOTO, Richard M. *Parallel discrete event simulation.* Communications of the ACM, 1990, 33.10: 30-53.

[2] JEFFERSON, David R. *Virtual time.* ACM Transactions on Programming Languages and Systems (TOPLAS), 1985, 7.3: 404-425.

[3] KLEEN, Andi. *A numa api for linux.* Novel Inc, 2005.

[4] PELLEGRINI, Alessandro; QUAGLIA, Francesco. *The ROme OpTimistic Simulator: A Tutoria*l. In: Euro-Par 2013: Parallel Processing Workshops. Springer Berlin Heidelberg, 2014. p. 501-512.

[5] PELLEGRINI, Alessandro; QUAGLIA, Francesco. *Wait-Free Global Virtual Time Computation in Shared Memory TimeWarp Systems.* In: Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on. IEEE, 2014. p. 9-16.

[6] PELLEGRINI, Alessandro; VITALI, Roberto; QUAGLIA, Francesco. *The rome optimistic simulator: Core internals and programming model.* In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011. p. 96-98.

[7] VITALI, Roberto; PELLEGRINI, Alessandro; QUAGLIA, Francesco. *Towards symmetric multi-threaded optimistic simulation kernels. In: Principles of Advanced and Distributed Simulation (PADS)*, 2012 ACM/IEEE/SCS 26th Workshop on. IEEE, 2012. p. 211-220.

[8] WILSON, Paul R., et al. *Dynamic storage allocation: A survey and critical review.* In: Memory Management. Springer Berlin Heidelberg, 1995. p. 1-116.