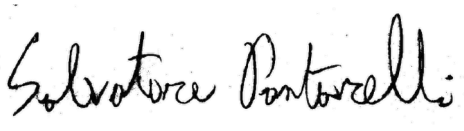
SAPIENZA
UNIVERSITÀ DI ROMA

Sviluppo e valutazione delle prestazioni di una cuckoo hash table lock-free per il tracciamento delle connessioni di rete

Facoltà di Ingegneria dell'informazione, Informatica e Statistica
Dipartimento di Informatica
Corso di Laurea in Informatica

Candidato
Luca Masi
Matricola 1969412

Responsabile di tirocinio
Prof. Salvatore Pontarelli



A.A. 2021/2022

Relazione non ancora discussa

Sviluppo e valutazione delle prestazioni di una cuckoo hash table lock-free per il tracciamento delle connessioni di rete.

Relazione di tirocinio. Sapienza – Università di Roma

Email dell'autore: luca.masi48@gmail.com

Ringraziamenti

Ringrazio tutta la mia famiglia per avermi permesso di arrivare fino qui, in particolare mia mamma e mio papà che mi hanno sostenuto economicamente e hanno creduto in me, ringrazio mia Nonna per avermi accolto e sopportato in questi ultimi due anni. Ringrazio Francesco e Andrea diventati miei amici durante questo percorso, che mi hanno supportato e aiutato a raggiungere uno dei miei obiettivi, infine ringrazio il professore S. Pontarelli per essere stato sempre disponibile e gentile, e per avermi indirizzato più volte sulla giusta strada.

Introduzione

Il lavoro svolto durante il periodo di tirocinio riguarda lo sviluppo e la valutazioni di un'applicazione scritta in linguaggio di programmazione C++ per il tracciamento delle connessioni di rete attive su di un server, in cui viene simulata (in locale) la creazioni di pacchetti di rete e in seguito inseriti e ricercati nella cuckoo hash table.

Con l'aumento prestazionale delle schede di rete e della rete stessa, le macchine in particolar modo i server sono ormai in grado di ricevere grandi quanti di dati, difatti le più moderne schede di rete (Network Interface Controller) superare i 100Gb di dati ricevuti al secondo.

Nasce cosi la necessità di velocizzare il tracciamento delle connessioni attive per aumentare le performance e il throughput delle applicazioni di rete. Per connessioni attive si intende un flusso bidirezionale di pacchetti tra client e server. Ogni qualvolta un server riceve un pacchetto dati, deve effettuare una ricerca in memoria in base all'identificativo del pacchetto, e controllare se è già presente una connessione verso quel client, nel caso positivo deve recuperare lo stato della connessione, altrimenti deve creare una nuova connessione. Più veloce è questa ricerca, più l'applicazione che il server sta esponendo alla rete, riesce ad operare ad alte velocità.

Il tracciamento delle connessioni riveste quindi un ruolo fondamentale nelle moderne applicazioni di rete come ad esempio per i giochi online, per i firewall, per la traduzione degli indirizzi di rete "NAT", o per il routing.

Le hash table sono un ottima struttura dati che permette di mappare tali connessioni ovvero associare una data "chiave" al suo relativo "valore". La chiave è appunto un identificativo univoco, in questo caso è l'identificativo del pacchetto dati, che viene dato in input ad una funzione di hash.

Le funzioni di hash prendono in input una sequenza di bit di lunghezza arbitraria, ovvero la chiave, e producono in output una stringa detta "hash" di lunghezza predefinita, tale stringa è in realtà una sequenza di bit interpretabile come numero (intero) che viene usato per indicizzare la tabella ovvero memorizzare in quella posizione la coppia "chiave-valore". Ogni qualvolta si deve ricerca una chiave nella hash table, occorre prima calcolare l'hash e poi controllare se in quella posizione è presente la chiave ricercata.

La particolarità di queste hash table, è che il costo medio di ricerca di ogni elemento è indipendente dal numero di elementi, ciò rende la ricerca istantanea o quasi.

Siccome però non esiste una funzione di hash perfetta possono verificarsi delle collisioni. Le collisioni si verificano quando due o più chiavi generano lo stesso hash e quindi lo

stesso indice, per risolvere tale problematica esistono diversi approcci: in questo progetto è stato usato quello basato sulla cuckoo hash table.

La cuckoo hash table è un particolare tipo di hash table che si basa sulla tecnica del open hash (indirizzamento aperto) per risolvere le collisioni, ovvero viene sondata l'hash table in posizioni alternative finché non viene trovato uno slot vuoto in cui mettere la coppia "chiave-valore". In particolare l'hash table si comporta come un tipo di uccello: il cuculo che quando è pulcino spinge fuori dal nido le altre uova, nello stesso modo, nella cuckoo quando si verifica una collisione la nuova chiave spinge fuori la vecchia chiave dalla tabella, la vecchia chiave viene poi riposizionata tramite un'altra funzione di hash in un'altra posizione. L'idea, che è stata poi sviluppata, è quella di usare due tabelle che insieme formano la hash table, e per ogni tabella usare una funzione di hash diversa, così che quando si verifica una collisione, la chiave che viene "cacciata" viene inserita nell'altra tabella. Tale hash table risulta essere un ottima struttura nel nostro caso perché essendo a indirizzamento aperto, il tempo di ricerca di una chiave è costante nel peggiore dei casi. Per la ricerca di una chiave occorre verificare se è presente in ognuna delle tabelle, calcolando l'hash con la rispettiva funzione di hash.

L'obiettivo principale di questo lavoro, è stato quello di sviluppare una versione *lock-free* della cuckoo hash table. Per *lock-free* si intende un applicazione multithread priva di blocchi come mutex e semafori in modo da evitare che un thread blocchi l'esecuzione di altri per un tempo indefinito, perché non riesce a rilasciare un lock acquisito. In questa versione della cuckoo è possibile effettuare contemporaneamente più ricerca e inserimenti mandando in esecuzione più thread in parallelo. Per fare questo, vengono usate delle istruzioni atomiche.

Le istruzioni atomiche essendo eseguite in un solo colpo di clock, evitano il principale problema delle applicazioni multithread, ovvero incoerenze sui dati causati dall'esecuzione di più flussi/thread su una struttura dati condivisa.

In particolare vengono usate le istruzioni:

- *___atomic_compare_exchange_16*
- *___atomic_exchange_16*

che si basano sull'istruzione CMPXCHG16B messa a disposizione da Intel sui processori a 64bit; tali istruzioni implementano operazioni di compare-and-exchange e exchange con word a 16 byte. Ed è proprio grazie a queste istruzioni che è stato possibile concludere tale lavoro perché, il dato da memorizzare nella hash table, occupa appunto uno spazio totale di 128bit ovvero 16 byte.

Capitolo 1

Reti

In questo capitolo viene descritto il motivo per cui è necessario tenere traccia dei pacchetti ricevuti, di come vengono identificati i pacchetti e delle informazioni legate ad una connessione.

Viene discusso anche in breve delle strutture dati usate nell'applicazione per identificare i pacchetti e per lo stato della connessione.

1.1 Pacchetto dati

Un pacchetto è un'unità di dati utilizzata nelle comunicazioni in una rete ed è formattato in una specifica struttura composta da due elementi generali: il payload che contiene i dati che siamo interessati ad inviare e l'header che porta con sé le informazioni di controllo. L'header contiene tutte le informazioni utili per identificare il pacchetto.

Per l'identificazione sono di particolare interesse i campi appartenenti a due dei cinque livelli che compongono un pacchetto: quello di trasporto e rete.

Il livello di trasporto contiene informazioni sulla porta di partenza e su quella di arrivo del pacchetto; esse sono numerate con un valore compreso nell'intervallo da 0 a 65535 ciò implica che il campo porta è quindi rappresentato da 16 bit.

Il livello di rete contiene invece l'indirizzo IP sorgente e destinazione. Un indirizzo IP è un etichetta che ha lo scopo di identificare un'interfaccia nella rete. Sono in uso due versioni del protocollo IP: IPv4 e IPv6. Nell'applicazione viene usata la prima versione, che fa uso di 32 bit per rappresentare un indirizzo.

Classificare un pacchetto significa identificare, in base ai campi sopra descritti, la sua appartenenza ad un determinato flusso di dati ovvero a quale connessione.

1.2 Tracciamento connessioni

Una connessione è un flusso bidirezionale di dati fra due host, collegati fisicamente alla rete, ogni connessione può essere identificata univocamente per mezzo di una chiave.

La chiave è la 4-tupla formata da:

1. Indirizzo sorgente
2. Porta sorgente
3. Indirizzo destinazione
4. Porta destinazione

Tali campi vengono estratti dall'header del pacchetto ricevuto.

Il tracciamento delle connessioni attive è quindi fondamentale per consentire ai due host connessi di scambiarsi dati a vicenda. Ad esempio nell'architettura client-server, il client invia e riceve pacchetti da uno o più server, viceversa ogni server invia e ricevi pacchetti da più client, però sia il client che il server devono conoscere lo stato della connessione, perché ci sono alcune informazioni fondamentali per iniziare la connessione, per scambiarsi dati e per concludere la connessione. Per fare questo quindi ogni host deve tenere tracciare delle connessioni attivi, in base all'identificativo del pacchetto, e ottenere info sullo stato di quella specifica connessione.

1.3 Stato della connessione

Nelle connessioni TCP, ogni connessione attiva ha uno stato ovvero in base alla fase in cui si trova la connessione, il pacchetto trasporta diversi tipi di dati che occorre salvare in memoria:

1. Handshake: ovvero la fase in cui si instaura la connessione. I pacchetti che si scambiano i due host contengono i flag SYN (1bit) e ACK (1bit), il numero di sequenza del pacchetto (32bit) e il numero di riscontro detto ACKn (32bit). Durante questa fase il client invia al server un messaggio di "sincronizzazione" contenente il flag SYN posto a 1, ACK posto a 0 e numero di sequenza casuale, il server risponde con un messaggio di "riconoscimento della sincronizzazione", contenente SYN=1, ACK=1 e numero di sequenza= casuale e numero di riscontro= numero di sequenza del client +1, il client poi risponde inviando un messaggio di "riconoscimento" che contiene ACK=numero di sequenza del server +1, qui termina la fase di handshake e inizia la fase di scambio dei dati.
2. Fase di trasmissione: i flag che interessano questa fase sono: PSH (push) e ACK, il client ha PSH e ACK posti a 1, il server ha solo il flag ACK posto a 1.

3. Fase di chiusura: in questa fase i pacchetti contengono il flag FIN (finish) posto a 1 più relativi numero di sequenza e riscontro finali.

Occorre quindi estrarre dal pacchetto e memorizzare in memoria tutti questi campi per tenere sotto controllo la connessione e il corretto scambio di dati.

1.3.1 Strutture dati

In base a quanto detto, per realizzare il programma che simula la ricezione di tali pacchetti è stata creata un apposita struttura dati denominata packetID, per identificare il pacchetto ovvero la chiave.

```
typedef struct {
        uint16_t src_port;
        uint16_t dst_port;
        uint32_t src_addr;
        uint32_t dst_addr;
} packetID;
```

Mentre per il valore è stato usato un semplice intero e non un apposita struttura dati contenete le info relative allo stato della connessione, questo perché altrimenti la struttura da memorizzare nella hash table supererebbe la dimensione massima di 128bit, non permettendo l'uso delle istruzioni atomiche. Per ovviare a tale problema si potrebbe usare come valore un apposito puntatore che punta alla struttura dati dello stato della connessione, che però dovrebbe essere completamente riallocata ogni qualvolta si aggiorna, per questo motivo è stato scelto, per semplicità, di usare un intero.

```
typedef struct{
        bool SYN;
        bool ACK;
        bool PSH;
        bool FIN;
        uint32_t nSEQ;
        uint32_t ACKn;
} connectionState;
```

Capitolo 2

Hash table

Le hash table sono un ottima struttura dati per effettuare ricerche in tempo costante a prescindere dal numero di elementi che popolano la tabella, questo perché si cerca di accedere agli elementi della tabella in modo diretto, trasformando la chiave in indirizzi della tabella per mezzo di operazioni aritmetiche.

2.1 Cos'è un Hash Table

L'hash table è uno struttura dati, banalmente si può dire che è una tabella ovvero un array, usata per mettere in corrispondenza una data *chiave* con un dato *valore*.

Ogni cella della tabella è chiamata bucket e deve contenere una coppia chiave-valore, la chiave è un identificatore univoco, il valore è il dato da associare alla chiave.

La chiave viene usata per indicizzare il valore all'interno della tabella, ovvero per trovare l'indirizzo della tabella dove inserire la coppia chiave-valore. Tramite la chiave è possibile ottenere l'accesso diretto all'elemento senza dover scorrere l'intera tabella. Questa è la principale caratteristica degli algoritmi di ricerca basati su hashing ovvero consentono di effettuare una ricerca in tempo costante.

La struttura entryTable usata nel programma è appunto una struttura dati formata dalla coppia chiave-valore che viene inserita in una cella della hash table.

```
typedef struct {
        packetID key;
        int value;
} entryTable;
```

2.2 Funzioni di hashing

Il primo passo per realizzare un hash table è quello di determinare la funzione di hash. La funzione di hash è un particolare tipo di funzione matematica che prende in input una sequenza di bit di lunghezza arbitraria, e fornisce in output un "hash" ovvero una sequenza di bit di lunghezza predefinita. L'hash viene poi interpretato come un intero e tramite l'operazione di modulo:

$$hash \ \% \ n$$

dove n è la size (dimensione) della hash table viene trasformato in un numero compreso tra 0 e n-1. Il valore così ottenuto, è l'indirizzo del bucket nell'array dove andrà memorizzata la coppia "chiave-valore".

Esistono molteplici funzioni di hash, però nessuna è totalmente iniettiva, cioè che ad una chiave corrisponde uno ed un solo hash. I principali fattori di valutazione di un hash function sono due:

- Latenza: ovvero quando tempo impiega a fornire in output l'hash
- Distribuzione: si riferisce alla capacità della funzione di distribuire equamente le chiavi nella tabella riducendo al minimo le collisioni.

Per questo progetto è stata usata la funzione di hashing: CRC a 32 bit, una funzione con una buona distribuzione, ma con bassissima latenza, questo perché l'obiettivo principale è quello di velocizzare il tracciamento delle connessioni, quindi occorre che la funzione di hash non faccia perdere tempo durante il calcolo del hash.

2.2.1 CRC32

CRC acronimo di Cyclic Redundancy Check (controllo di ridondanza ciclico) è un algoritmo per il calcolo del checksum ovvero una sequenza di bit usata per rilevare la presenza di errori in un dato. Tale algoritmo può essere usato come funzione di hash perché genera in output una sequenza di bit predefinita che dipende dai dati in input.

Il nome deriva dal fatto che i dati d'uscita sono ottenuti elaborando i dati di ingressi un po' per volta, accumulati in un registro e poi elaborati di nuovo insieme ad altri dati d'input.

Il calcolo del *CRC* è basato sull'aritmetica modulare, in cui i numeri "si avvolgono su loro stessi" ogni volta che raggiungono i multipli di un determinato numero detto modulo.

$$^{1}\text{CRC(M)} = (\text{M} << \text{w}) \bmod \text{P}$$

Questo è il modo in cui viene calcolato il *CRC* ovvero è il resto della divisione tra il messaggio e un polinomio detto polinomio generatore. La scelta del polinomio P da

utilizzare è fondamentale per produrre un algoritmo *CRC* con buone proprietà di rilevamento degli errori.

I processori Intel mettono a disposizione una propria istruzione per il calcolo del *CRC*:

- *unsigned int _mm_crc32_u32(unsigned int crc, unsigned char data)*

questa è l'istruzione usata nel programma per il calcolo del *CRC* a 32 bit che è stato usato come hash.

L'istruzione parte con un valore iniziale nel primo operando (operando di destinazione), accumula il valore del CRC32 nel secondo operando (operando sorgente) e salva il risultato nel operando di destinazione. L'operando di destinazione deve essere un registro o locazione di memoria a 32bit. Cambiando il valore iniziale con cui l'istruzione parte, si ottiene un diverso *CRC* in output.

Esistono anche i *CRC* a 64 bit ma sono usati raramente perché quelli a 32 bit sono già "abbastanza buoni", perché riescono a rilevare errori per messaggi di qualsiasi lunghezza.

2.3 Gestione delle Collisioni

Come già detto non esiste una funzione di hash perfetta, per questo motivo possono verificarsi delle collisioni.

Le collisioni si hanno quando due chiave distinte, generano lo stesso valore di "hash", ciò vuol dire che si deve memorizzare nello stesso bucket due chiavi distinte, ma essendo il bucket a chiave unica risulta impossibile.

Le collisioni non dipendono solamente da quanto è "buona" una funzione di hash ma sono fortemente legate anche al cosiddetto fattore di carico (*load factor*):

$$load\ factor = chiavi\ da\ inserire/\ dimensione\ della\ tabella$$

tale valore indica quanta probabilità ha un nuovo elemento di collidere con uno già presente nella tabella. È bene dunque mantenere il *load factor* il più basso possibile (di solito un valore di 0.75 è quello ottimale) per ridurre al minimo il numero di collisioni. Ciò può essere fatto, ad esempio, ridimensionando l'array ogni volta che si supera il *load factor* desiderato.

Esistono diverse tecniche per la gestione delle collisioni:

- Open Hash (indirizzamento aperto): tutti gli elementi sono memorizzati nella tabella hash; ovvero ogni cella della tabella contiene un elemento. Quando si verifica una collisione, viene sondata l'hash table in posizioni alternative finché non viene trovato uno slot vuoto in cui mettere l'elemento.

Nell'indirizzamento aperto, la tabella hash può "riempirsi" al punto tale che non possono essere effettuati altri inserimenti. La ricerca di un elemento è però effettuata in tempo costante.

- Hash con concatenazione: per ogni cella della tabella hash si fa corrispondere invece di un elemento, una lista di elementi. In questo modo un elemento che collide viene aggiunto alla lista corrispondente all'indice ottenuto.

  E' possibile quindi aggiungere quanti elementi si vogliono a discapito però delle prestazioni in termini di ricerca perché occorrerà scorrere tutta la lista finché non viene trovato l'elemento desiderato.

In questo progetto viene usata la cuckoo hash table per gestire le collisioni, che si basa sull'indirizzamento aperto.

Capitolo 3

Cuckoo hash table

[2]La cuckoo hash table è un tipo di hash table basato sull'indirizzamento aperto, cioè quando si verifica una collisione, la chiave che l'ha generata viene inserita, se possibile, in una posizione alternativa.

In particolare quando si verifica una collisione, la cuckoo si comporta come il cuculo, da qui il nome.

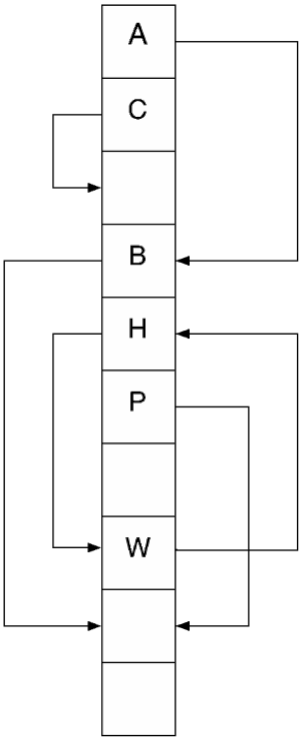
Il cuculo è un uccello che quando è pulcino spinge fuori dal nido le altre uova così che la madre acquisita (non è la vera madre) non riesce a rendersi conto che il pulcino non appartiene alla sua specie, nello stesso modo quando si verifica una collisione nella cuckoo, la chiave che occupa uno slot viene rimpiazzata con la nuova chiave, la vecchia chiave viene poi posizionata in un altro indirizzo, calcolandolo con un altra funzione di hash.

In figura un esempio di cuckoo hash a tabella singola.

Le frecce mostrano la posizione alternativa di ogni chiave.

Se vorremmo inserire una nuova chiave nell'ubicazione di A, questo causerebbe un riposizionamento della vecchia chiave che occupa A, spostandola nella sua posizione alternativa, attualmente occupata da B, e spostando di conseguenza B nella sua posizione alternativa che è attualmente libera.

L'inserimento di un nuovo elemento nella posizione di H non andrebbe a buon fine: poiché H fa parte di un ciclo (insieme a W), il nuovo elemento verrebbe espulso di nuovo.



Ciò che rende tale hash table appropriata al nostro obiettivo è che la ricerca di una chiave è diretta ovvero avviene in tempo O(1).

3.1 Funzionamento e implementazione

La cuckoo hash table implementata in questo progetto usa due tabelle distinte, ogni tabella ha una propria funzioni di hash e ogni bucket è in grado di contenere più coppie chiave-valore.

Il funzionamento è simile alla cuckoo a tabella singola, solo che in questo caso quando la chiave da inserire trova lo slot pieno, sostituisce la vecchia chiave e la vecchia chiave viene inserita nell'altra tabella usando la funzione di hash ad essa associata; anche in questo caso se lo slot nell'altra tabella è occupata si ripete il procedimento altrimenti si inserisce la chiave.

[3]Utilizzando un bucket con 1 solo slot cioè in grado di contenere solo una coppia chiave-valore, l'hash table può raggiungere un *load factor* del 50%. Il *load factor* è il rapporto tra il numero di chiave inserite e la dimensione della hash table. Con 1 solo slot per bucket il procedimento di riposizionamento delle chiavi può essere ripetuto una sola volta, cioè al massimo la chiave può essere riposizionata una sola volta nella seconda tabella.

Nel nostro caso, vengono usati 4 slot per bucket cioè, un bucket può contenere un massimo di 4 coppie chiave-valore, il *load factor* sale al 90% questo perché il procedimento di riposizionamento può essere ripetuto più volte, scegliendo a caso una fra le quattro chiavi da riposizionare nell'altra tabella. Tale procedimento fallisce quando dopo un certo numero di riposizionamenti la chiave non viene inserita in nessuno slot, in questo caso la chiave viene posizionata in un apposita locazione di memoria associata alla hash table, chiamata "*vittima*".

La "*vittima*" è destinata appunto a contenere la chiave non inserita nella hash table, tale chiave appartiene lo stesso alla hash table ed è quindi ricercabile. Quando la *vittima* è occupata non è possibile inserire più ulteriori chiavi.

Per la ricerca di una chiave occorre ricercarla in ognuna delle tabelle usando le rispettive hash function e anche nella *vittima*.

La *struct* definita nel progetto per la *cuckoo hash table* contiene:

- Numero di "elementi" presenti nella tabella
- Numero totali di "elementi" che può contenere
- Puntatore alla prima tabella
- Puntatore alla seconda tabella

- Due valori distinti, uno per ogni tabella, da passare alla hash function usati come valore iniziali nel calcolo del CRC32
- Size delle singole tabelle
- Numero di slot ovvero quante coppie chiave-valore si possono memorizzare in un bucket
- Numero massimo di volte che si può ripetere il procedimento di riposizionamento
- Locazione di memoria destinata a contenere la vittima

```
typedef struct{
        uint32_t SIZE;
        int num_entry;
        uint32_t sizeSingleTable;
        int max_move;
        entryTable **T1;
        entryTable **T2;
        uint32_t hash_T1;
        uint32_t hash_T2;
        int slot;
        int TOTentry;
        entryTable victim;
} cuckooTable;
```

Le tabelle sono degli array bidimensionali perché ogni bucket contiene più slot, tali array vengono allocati dinamicamente al momento della creazione della cuckoo:

```
entryTable** ckh_alloc_table ( uint32_t sizeTable, int numslot)
{
        entryTable nop ={.key={0},.value=0};
        entryTable** T = ( entryTable**) CALLOC ( entryTable, sizeTable);
        for( int i = 0; i < sizeTable; i++){
                T[i] = ( entryTable*) CALLOC ( entryTable, numslot);
                for( int j = 0; j < numslot; j++){
                        T[i][j] = nop;
                }
        }
        return T;
}
```

3.2 Inserimento

```
int ckh_insert(cuckooTable *D, packetID *key, int value)
{
        uint32_t h;
        entryTable tmp, x;
        int countMove =0;
        if ( ckh_update( D, key, value)== 0)      //controllo se la key è già presente e in caso
                return 0;                         //aggiorno

        if ( key_empty( &D→victim.key)== 0){  //se la vittima è piena
                D→num_entry++;                    // aumento il numero di elementi presenti
                x.key =*key;
                x.value =value;
                while ( countMove < D→max_move)
                {
                        h =hash ( D→sizeSingleTable, &x.key, D→ hash_T1);   //calcolo l'indice
                        for ( int i=0; i<D→slot; i++)
                        {
                                if ( key_empty(&D→T1[h][i].key)== 0){  //se c'è uno slot vuoto
                                        D→ T1[h][i] = x;              // inserisco l'elemento
                                        return 0;
                                }
                        }
                        int j =rand()%D→ slot;
                        tmp =D→ T1[h][j];                    //scambio le chiavi
                        D→T1[h][j] =x;
                        x =tmp;
                        h =hash ( D→ sizeSingleTable, &x.key, D→ hash_T2);
                        for ( int i=0; i<D→ slot; i++)
                        {
                                if ( key_empty(&D→T2[h][i].key)== 0){ //provo ad inserire la key
                                        D→ T2[h][i]= x;               //nella seconda tabella
                                        return 0;
                                }
                        }
                        tmp =D→ T2[h][j];
                        D→ T2[h][j] =x;               //scambio chiavi
                        x =tmp;
                        countMove++;
                }
                D→ victim =x;        //se fallisce, inserisco nella vittima
                return 0;
        }
        return -1;
}
```

Per inserire un “elemento” nella cuckoo, prima occorre verificare se la chiave è già presente, in caso positivo occorre solo aggiornare il valore associato alla chiave altrimenti inizia la procedura di inserimento:

si verifica che la vittima sia vuota, in caso positivo si prova a posizionare l’elemento nella prima tabella, se c’è uno slot vuoto la chiave viene inserita altrimenti se tutti gli slot sono pieni, si estrae uno slot casuale e si scambia l’elemento nuovo con quello vecchio, l’elemento vecchio viene riposizionato nella seconda tabella, anche qui se tutti gli slot sono pieni viene estratto un nuovo slot, si scambiano le chiavi e la chiave che risiedeva nella seconda tabella viene “provata” ad inserire nella prima tabella. Tutto il procedimento di riposizionamento si ripete per un numero di volte prestabilito (max_move =200) dopodiché se la chiave non è stata inserita viene memorizzata nella vittima.

Quando la vittima è piena non è possibile inserire più chiavi.

3.3 Lettura

```
int ckh_getVal ( cuckooTable *D, packetID *key)
{
	uint32_t h;
	h=hash(D→sizeSingleTable, key, D→hash_T1);		//hash table 1
	for(int i=0; i<D→slot; i++)
	{
		if (key_cmp(&D→T1[h][i].key, key) == 0 && key_empty(&D→T1[h][i].key)==0)
		{
			return D→T1[h][i].value;
		}
	}

	h=hash(D→sizeSingleTable, key, D→hash_T2);		//hash table 2
	for(int i=0; i<D→slot; i++)
	{
		if (key_cmp(&D→T2[h][i].key, key) == 0 && key_empty(&D→T2[h][i].key)==0)
		{
			return D→T2[h][i].value;
		}
	}

	if(key_cmp(&D→victim.key, key)==0 && key_empty(&D→victim.key)==0)
	{
		return D→victim.value;
	}
	return -1;
}
```

La lettura consiste nel ottenere il "valore" associato a una determinata chiave.

Data una chiave in input, occorre verificare se è presente in ognuna delle tabelle, calcolando i rispettivi hash e controllando nei rispettivi indici per ogni slot, inoltre occorre controllare anche nella vittima; ovviamente se la chiave viene trovata, non c'è bisogno di andare avanti nella ricerca, basta far ritornare il valore. Le due funzioni usate nella lettura:

- `int key_cmp ( packetID* a, packetID* b);`
- `int key_empty ( packetID* a);`

servono rispettivamente per comparare due chiave, e per controllare se uno slot è vuoto.

3.4 Delete

```
int ckh_delete ( cuckooTable *D, packetID *key)
{
	uint32_t h;
	entryTable nop={.key={.src_port=0,.dst_port=0,.src_addr=0,.dst_addr=0}, .value=0};

	h=hash(D→sizeSingleTable, key, D→hash_T1);		//hash table 1
	for(int i=0; i<D→slot; i++)
	{
		if (key_cmp(&D→T1[h][i].key, key) == 0 && key_empty(&D→T1[h][i].key)==0)
		{
			D→T1[h][i] = nop;
		}
	}

	h=hash(D→sizeSingleTable, key, D→hash_T2);		//hash table 2
	for(int i=0; i<D→slot; i++)
	{
		if (key_cmp(&D→T2[h][i].key, key) == 0 && key_empty(&D→T2[h][i].key)==0)
		{
			D→T2[h][i] = nop;
		}
	}

	if(key_cmp(&D→victim.key, key)==0 && key_empty(&D→victim.key)==0)
	{
		D→victim = nop;
	}

	return -1;
}
```

L'eliminazione di una chiave dalla hash table è molto simile alla ricerca, infatti occorre prima trovare la chiave da eliminare, e poi azzerare (nop è appunto una entry tutta a zero) lo slot.

3.5 Update

```
int ckh_update ( cuckooTable *D, packetID *key, int value)
{
	uint32_t h;

	h=hash(D→sizeSingleTable, key, D→hash_T1);		//hash table 1
	for(int i=0; i<D→slot; i++)
	{
		if (key_cmp(&D→T1[h][i].key, key) == 0 && key_empty(&D→T1[h][i].key)==0)
		{
			D→T1[h][i].value = value;
		}
	}

	h=hash(D→sizeSingleTable, key, D→hash_T2);		//hash table 2
	for(int i=0; i<D→slot; i++)
	{
		if (key_cmp(&D→T2[h][i].key, key) == 0 && key_empty(&D→T2[h][i].key)==0)
		{
			D→T2[h][i].value = value;
		}
	}

	if(key_cmp(&D→victim.key, key)==0 && key_empty(&D→victim.key)==0)
	{
		D→victim.value = value;
	}

	return -1;
}
```

Anche l'update come l'eliminazione è simile alla lettura; prima occorre trovare la chiave, poi aggiornare il "valore" ad essa associata.

Capitolo 4

Cuckoo Hash table lock-free

Per velocizzare il tracciamento delle connessioni, oltre che ad usare questa particolare hash table, il programma è in grado di effettuare contemporaneamente più ricerca; ciò è possibile perché ogni macchina è dotata di più core sui quali eseguire più thread, quindi è possibile "lanciare" più thread che leggono dalla cuckoo. Per far sì che ogni thread, sia quello di scrittura che quello in lettura, non si intralcino a vicenda causando un drastico calo delle performance, è stata implementata una versione *lock-free* della cuckoo hash table.

4.1 Multithreading e Lock-free

Il multithreading indica il supporto hardware da parte di un processore di eseguire più thread in parallelo. Un thread è una suddivisione di un processo in due o più flussi o sottoprocessi che vengono eseguiti concorrentemente. Questa tecnica si distingue da quella alla base dei sistemi multiprocessore per il fatto che i singoli thread condividono lo stesso spazio d'indirizzamento, la stessa cache e lo stesso translation lookaside buffer (TLB).

Il multithreading migliora le prestazioni dei programmi solamente quando questi sono stati sviluppati suddividendo il carico di lavoro su più thread che possono essere eseguiti in parallelo.

Principali problematiche del multithreading:

- Più thread condividono le stesse risorse come la memoria e quindi possono interferire a vicenda rallentandosi o peggio causando incoerenze sui dati.
  Sono note due problematiche in ambito della sincronizzazione sui dati:
  - problema del Produttore - Consumatore
  - problema Lettori – Scrittori
- Le prestazioni dei singoli thread non migliorano, ma anzi possono degradare all'aumento dei thread, questo perché la CPU non è in grado di eseguire in parallelo i thread, e quindi vanno in concorrenza.
  Quando più thread sono in concorrenza, alternano la loro esecuzione su un unità di elaborazione causando continui context switch (scambio fra lo stato del processo

attualmente in esecuzione e quello che deve essere eseguito) che possono rallentare l'esecuzione complessiva del programma.
- Il supporto hardware del multithreading richiede che i programmi siano adattati per gestire questa nuova possibilità.

L'ideale per ottenere le massime prestazioni dal multithreading è eseguire sempre un numero di thread inferiore o uguale, a quello supportato da un sistema di elaborazione, cosi che nessun thread vada in concorrenza con altri, e ogni thread potrà essere eseguito in parallelo. Inoltre occorre fare attenzione ai problemi legati alla sincronizzazione, nel nostro particolare caso, al problema Lettori - Scrittori.

Esistono diverse tecniche per ovviare a tale problema:

- Lock
- Semafori
- Monitor
- Istruzioni atomiche

quella a noi di interesse, che viene usata in questo progetto, riguarda le istruzioni atomiche. Questo perché le istruzioni atomiche garantiscono la scrittura di un programma *lock-free*.

Un programma si dice *lock-free o non-blocking* se il fallimento o la sospensione di un thread non può causare il fallimento o la sospensione di un altro thread. Un algoritmo *lock-fre*e è quindi un algoritmo privo di "blocchi" e privo di attesa, che garantisce l'avanzamento dell'intero programma a prescindere dall'andamento degli altri thread.

L'approccio tradizionale alla programmazione multithread consiste nell'utilizzare i blocchi per sincronizzare l'accesso alle risorse condivise. Le primitive di sincronizzazione come mutex e semafori sono però tutti meccanismi mediante i quali, se un thread tenta di acquisire un blocco che è già detenuto da un altro thread, il thread si bloccherà finché il blocco non sarà libero, ciò non solo causa un calo delle prestazioni, perché mentre il thread è bloccato non può fare nulla, ma possono verificarsi anche condizioni di errore come il deadlock (situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa).

Gli algoritmi *lock-free* fanno al nostro caso perché durante il tracciamento delle connessioni, un server può effettuare contemporaneamente scritture e letture dalla hash table, quindi occorre che i thread di scrittura e lettura vengano eseguiti a prescindere

dalle operazioni fatte sulla hash table, senza che nessun thread si blocchi o aspetti altri thread, portando di conseguenza ad un notevole aumento prestazionale del programma in termini di tempo, sia in scrittura che in lettura.

Con poche eccezioni, gli algoritmi non bloccanti utilizzano primitive atomiche di lettura-modifica-scrittura che l'hardware deve fornire, la più notevole è il compare and swap (CAS), istruzione atomica usata in questo progetto.

4.1.1 Istruzioni Atomiche

In generale, un'operazione si dice atomica se è indivisibile, ovvero se nessun'altra operazione può cominciare prima che la prima sia finita, e quindi non può esserci interleaving.

Un istruzione è atomica quando viene eseguita tutta insieme, in un solo colpo di clock della CPU, quindi non può essere stoppata e nessun altra operazione può essere eseguita nel mentre. Grazie a tali istruzioni ci si può non preoccupare dell'accesso alle sezioni critiche in un programma multithread perché quella data istruzione sarà eseguita per forza da un thread alla volta e quindi verranno evitate possibili incoerenze sui dati dovuti all'ordine di esecuzione dei thread.

In questo progetto, in particolare, sono state usate due istruzioni atomiche:

- *___atomic_compare_exchange_16*
- *___atomic_exchange_16*

tali istruzioni sono messe a disposizione dal compilatore tramite l'opzione -mcx16, la quale abilita il compilatore a generare l'istruzione CMPXCHG16B, istruzione di Intel sui processori a 64bit che implementa operazioni di compare-and-exchange con word a 16 byte.

[4]

- *bool ___atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak, int success_memorder, int failure_memorder)*;

questa è la funzione che implementa un compare and exchange atomico, sfruttando l'istruzione atomica: *___atomic_compare_exchange_16.* Tale funzione compara il contenuto di "ptr" con "expected", se sono uguali compie un operazione di lettura-modifica-scrittura che va a scrive "desired" in "ptr". Restituisce "true" se il compare è andato a buon fine, "false" altrimenti.

- *void ___atomic_exchange (type *ptr, type *val, type *ret, int memorder)*

questa è la funzione che implementa uno swap atomico fra due variabili, sfruttando l'istruzione: *___atomic_compare_exchange_16.* Tale funzione salva il contenuto di "val" in "ptr" e il valore originale di "ptr" viene copiato in "ret".

4.2 Atomic Insert

```
int ckh_atomic_insert(cuckooTable *D, packetID *key, int value)
{
        uint32_t h;
        int countMove=0;
        entryTable nop={.key=0, .value=0};
        entryTable entry={.key=*key, .value=value};
        if ( ckh_update( D, key, value)==0)
                return 0;

        if(__atomic_compare_exchange (&D→victim, &nop, &entry, 0, 0, 0)==true){
                __atomic_add_fetch(&D→num_entry, 1, 0);
                while(countMove < D→max_move)
                {
                        h=hash(D→sizeSingleTable, &D→victim.key, D→hash_T1);
                        for(int i=0; i<D→slot; i++)
                        {
                                if(__atomic_compare_exchange (&D→T1[h][i], &nop, &D→victim,
                                                                        0, 0, 0)==true){
                                        D→victim=nop;
                                        return 0;
                                }
                        }
                        int j=rand()%D→slot;
                        __atomic_exchange(&D→T1[h][j], &D→victim, &D→victim, 0);
                        h=hash(D→sizeSingleTable, &D→victim.key, D→hash_T2);
                        for(int i=0; i<D→slot; i++)
                        {
                                if(__atomic_compare_exchange (&D→T2[h][i], &nop, &D→victim,
                                                                        0, 0, 0)==true){
                                        D→victim=nop;
                                        return 0;
                                }
                        }
                        __atomic_exchange(&D→T2[h][j], &D→victim, &D→victim, 0);
                        countMove++;
                }
                return 0;
        }
        return -1;
}
```

L'inserimento nella cuckoo deve essere atomico perché nel caso di inserimenti multipli, un thread potrebbe trovare uno slot in cui inserire la propria chiave, ma a causa di una sua interruzione, un altro thread potrebbe fare lo stesso ed inserire la propria chiave nel

medesimo slot, quando il thread sospeso riprende l'esecuzione andrebbe a sostituire una chiave nella cuckoo che non potrà più essere ricercare.

L'inserimento atomico implementato, funziona nello stesso modo dell'inserimento normale, ma ci sono principalmente tre differenze:

1. L'elemento da inserire "entry" viene subito messo nella "vittima" se vuota, tramite un operazione di compare and exchange che compara la vittima con la entry nop (nop è una entry tutta a zero), in caso negativo l'inserimento fallisce perché la vittima è già occupata.
2. Per trovare lo slot libero in cui inserire l'elemento viene usata la funzione di compare and exchange, che compara uno slot della cuckoo con la entry nop, in pratica controlla se lo slot è vuoto, in caso positivo mette nello slot l'elemento entry formato da chiave più valore.
3. Nel caso in cui non ci sono slot libero, gli elementi devono essere scambiati, e siccome il nuovo elemento è nella vittima, basta scambiare il contenuto della vittima con uno slot estratto a caso.

4.3 MultiReader non-blocking

```
indexTable ckh_multiRead_nblck(cuckooTable *D, packetID *key)
{
        indexTable start={.i=0, .j=0, .table=0, .value=0};
        indexTable end={.i=1, .j=1, .table=1, .value=1};

        while(start.i!= end.i && start.j!= end.j && start.table!= end.table && start.value!
                                                                        =end.value{
                start=ckh_getVal(D, key);
                end=ckh_getVal(D, key);
        }
        return start;
}
```

Anche la lettura come l'inserimento deve essere atomica, perché può succedere che mentre un thread sta leggendo un valore, nel frattempo un altro thread faccia un inserimento giusto in quello slot oppure modifichi il valore associato a quella chiave, di conseguenza il thread che sta leggendo, ottiene un valore errato. Il problema è che non esiste un istruzione in grado di comparare solo le chiavi e scambiare anche il valore ad esse associate. Quindi non è possibile avvalersi di nessuna istruzione. Ciò che è stato fatto in questo progetto per leggere il valore associato ad una data chiave è una doppia lettura. La doppia lettura sfrutta la funzione: *ckh_getVal(cuckooTable *D, packetID *key)*, discussa nel capitolo precedente per effettuare una lettura, in una sua versione aggiornata, che oltre a restituire il valore associato ad una chiave, restituisce anche:

- il bucket in cui risiede
- lo slot che occupa
- la tabella (o vittima).

La struttura dati *indexTable* consente appunto di ottenere tutte queste info.

```
typedef struct{
        int bucket;
        int slot;
        int table;
        int value;
} indexTable;
```

tramite queste informazioni è possibile capire se una data chiave è stata spostata, aggiornata o eliminata da altri thread, perché rileggendo due volte, e poi confrontando tali informazioni, se le informazioni risultano le stesse in entrambe le lettura allora questo vuol dire che nessun altro thread nel frattempo ha operato su quella chiave, in quello slot. Questo è proprio quello che viene fatto nel codice, tramite il ciclo *while* che fa ripetere le due letture finché le informazioni non combaciano; una volta usciti dal *while* si è quindi sicuri che il valore letto è proprio quello associato alla chiave.

4.4 Atomic Delete e Atomic Update

```
int atomic_ckh_delete(cuckooTable *D, packetID *key)
{
        bool compare=false;
        entryTable nop={.key=0, .value=0};

        while(compare==false){
                indexTable val_index=ckh_multiRead_nblck(D, key);
                entryTable tmp={.key=*key, .value=val_index.value};

                if(val_index.table==0)
                        compare=__atomic_compare_exchange (&D→T1[val_index.i][val_index.j],
                                                                &tmp, &nop, 0, 0, 0);
                else if(val_index.table==1)
                        compare=__atomic_compare_exchange (&D→T2[val_index.i][val_index.j],
                                                                &tmp, &nop, 0, 0, 0);
                else if(val_index.table==2)
                        compare=__atomic_compare_exchange (&D→victim, &tmp, &nop, 0, 0, 0);
                else
                        return -1;
        }
        __atomic_sub_fetch(&D->num_entry, 1, 0);
        return 0;
}
```

```
int atomic_ckh_update(cuckooTable *D, packetID* key, int value)
{
        bool compare=false;
        entryTable newVal={.key=*key, .value=value};

        while(compare==false){
                indexTable val_index=ckh_multiRead_nblck(D, key);
                entryTable tmp={.key=*key, .value=val_index.value};

                if(val_index.table==0)
                        compare=__atomic_compare_exchange (&D→T1[val_index.i][val_index.j],
                                                                &tmp, &newVal, 0, 0, 0);
                else if(val_index.table==1)
                        compare=__atomic_compare_exchange (&D→T2[val_index.i][val_index.j],
                                                                &tmp, &newVal, 0, 0, 0);
                else if(val_index.table==2)
                        compare=__atomic_compare_exchange (&D→victim, &tmp, &newVal, 0, 0, 0);
                else
                        return -1;
        }
        return 0;
}
```

l'update e il delete sono due funzioni molto simili, con la solo differenze che in una si aggiorna lo slot con un entry formata da: chiave in input + valore, nell'altro caso si aggiorna lo slot con una nop ovvero si azzero lo slot.
Sia update che delete devono essere atomici perché in caso di esecuzione di più thread potrebbe succedere che si modifichi o elimini una chiave che non è quella richiesta, perché altri thread potrebbero averla spostata.
Anche in questo caso, come nella lettura, non c'è un istruzione che consente di confrontare solo la chiave e poi cambiare tutta la entry (compreso il valore); quindi per realizzare queste due funzioni viene implementato un algoritmo che:
prima legge il giusto valore associato alla chiave tramite la funzione di lettura: *indexTable ckh_multiRead_nblck(cuckooTable *D, packetID *key);* poi crea una entry temporanea "tmp" formata da: chiave in input + valore ottenuto dalla lettura;
in base poi alle informazioni relative alla posizione della chiave, ottenute sempre tramite la lettura, si va a comparare la entry "tmp" con l'attuale valore nella tabella alla posizione ottenuta. Se il compare va a buon fine, lo slot viene, in caso di update, aggiornato con la entry "newVal", in caso di delete, viene azzerato con la entry "nop".

Usando il compare and exchange atomico si ha la certezza di effettuare tali operazioni sulla giusta chiave.

Capitolo 5

Testing

In questo capitolo vengono analizzate le prestazioni della cuckoo hash table implementata durante il tirocinio, spiegati i test effettuati su di essa e di come viene simulata la ricezione di pacchetti di reti.

5.1 Setup

Per simulare la ricezione di pacchetti di rete viene allocato dinamicamente un array della struttura dati che segue:

```
typedef struct{
        packetID key;
        int id;
        int ckh;
}packet_main;
```

quindi da linea di comando si può indicare quanti pacchetti di rete creare; si può accedere a tali pacchetti tramite puntatore. “key” è appunto l’identificativo del pacchetto,”id” invece è il valore ad esso associato, “ckh” è un campo che viene posto ad uno quando la key viene inserita nella cuckoo (è servita prevalentemente per controlla se le key venissero realmente inserite).

*int generate_packet(packet_main *pack, int numEntry)* è la funzione che genera pacchetti casuali, in sostanza assegna valore casuali ma sempre diversi per la 4-tupla che identifica il pacchetto, e un valore crescente al campo “id”.

Sempre da input è possibile passare la size, cioè la dimensione della cuckoo, che tramite le funzioni: *void init_ckhTable(cuckooTable *Table,……), entryTable** *ckh_alloc_table(uint32_t sizeTable, int numslot)* inizializza e alloca in memoria la cuckoo hash table.

5.2 Test

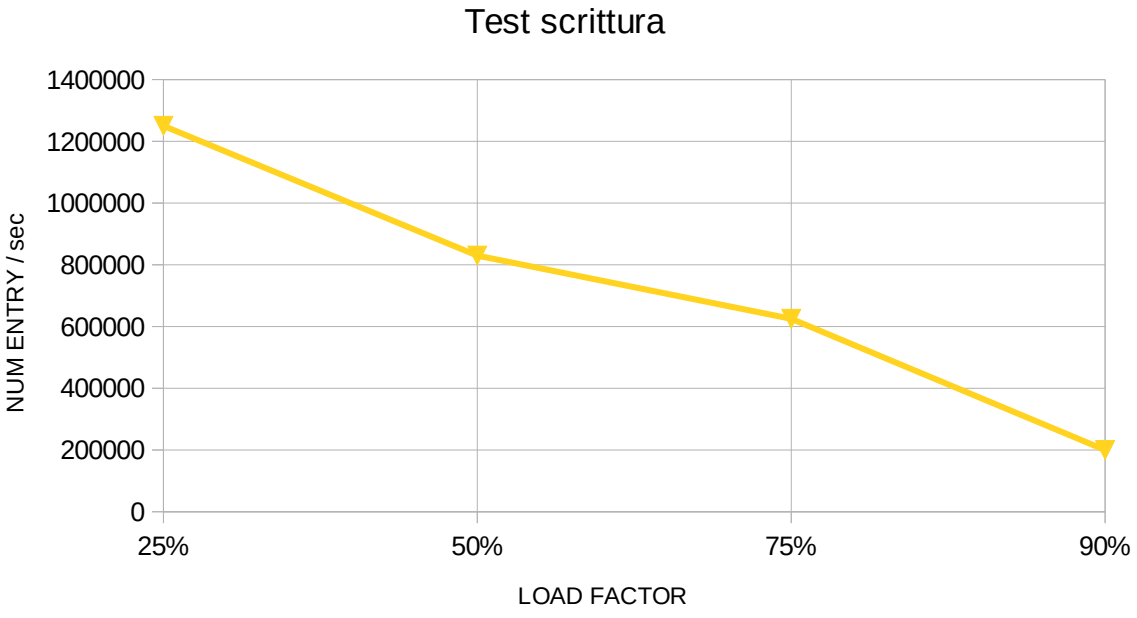
I test effettuati sulla cuckoo table sono principalmente 3:

- test in scrittura: viene testata la velocità di inserimento nella cuckoo quando il load factor è al 25%, 50% e al 75%.
- test in lettura: viene testata la velocità in lettura sulla cuckoo hash table quando nessun altro thread sta facendo modifiche su di essa, quindi la tabella viene prima riempita fino al suo massimo (all'incirca 90%) e poi viene effettuata la lettura.
- test in scrittura e lettura: viene testata la velocità di lettura mentre altri thread stanno apportando modiche alla cuckoo (inserimenti, delete o update). In particolare vi è un test che riempie la cuckoo fino all 80% circa, e poi inizia a leggere ma nel mentre vengono effettuati insert e delete di continuo.

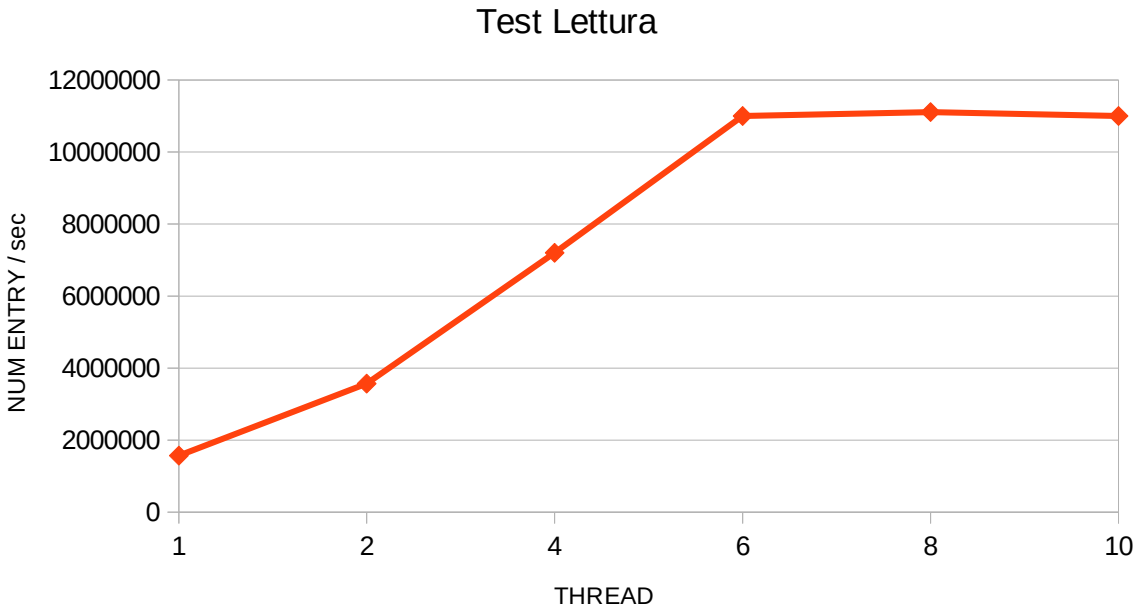
Tali test mirano a mettere sotto stress la struttura dati creata, e a capire se una cuckoo hash table *lock-free* è realmente più veloce, sia in scrittura che in lettura, rispetto ad una cuckoo hash table che usa meccanismi di sincronizzazione come i mutex.

5.3 Valutazione delle prestazioni

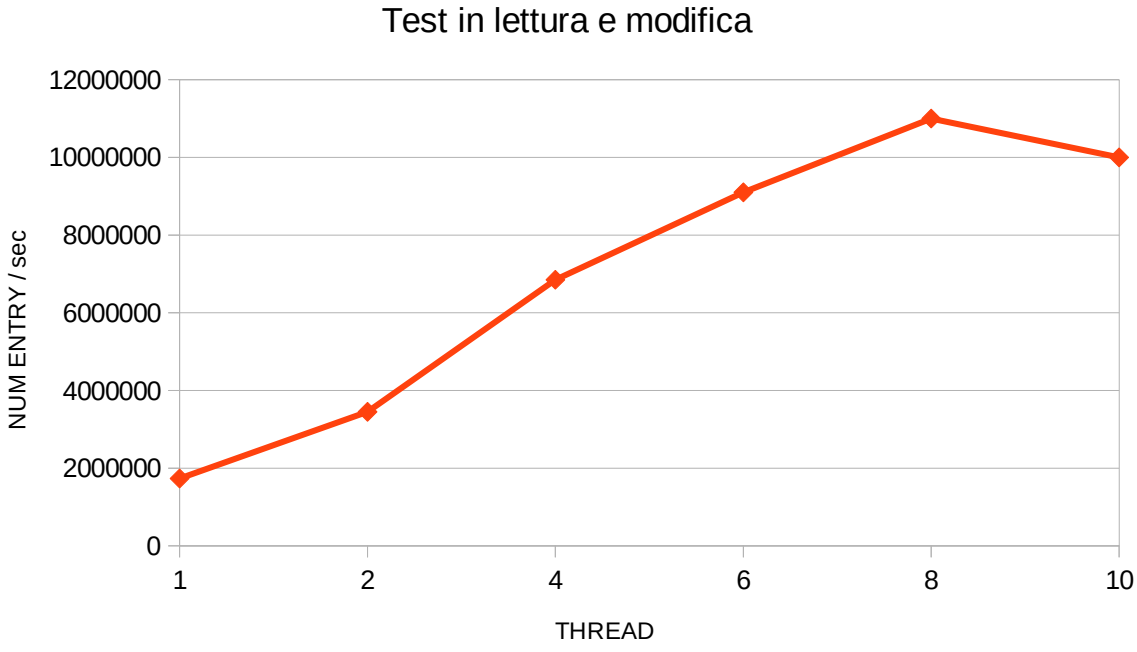
Il capitolo mostra i risultati ottenuti nei vari test dall'infrastruttura software sviluppata.



In figura è riportato il grafico che rappresenta il numero di pacchetti inseriti al secondo, man mano che la cuckoo si riempie. Come si può ben notare le prestazioni calano man mano che il load factor raggiunge il 90%.



Il grafico rappresenta il numero di query (letture) al secondo, all'aumentare del numero di thread. Dal grafico si evince come all'aumentare del numero dei thread (quelli per la ricerca di una chiave), il numero di query effettuate al secondo cresce linearmente, fino al raggiungimento degli 6 thread, questo perché la macchina sulla quale è stato effettuato il test, possiede 4 core e 2 thread per core, per un totale massimo di 8 thread. Quindi al raggiungimento dei 6 thread, essendo anche il processo principale in esecuzione, tutte le unità di elaborazione risultano occupate, per questo i thread per essere eseguiti vanno in concorrenza con gli altri, non portando nessun miglioramento in termini di prestazioni.



Il grafico rappresenta il numero di query (letture) al secondo, all'aumentare del numero di thread. A differenza dell'altro test però ci sono altri 2 thread in esecuzione durante la lettura, un thread delete, che elimina le chiavi dalla cuckoo e un thread insert che inserisce chiave; in questo modo il thread in lettura dovrebbe fare più fatica perché la cuckoo viene continuamente modificata dagli altri due thread. Però da come si vede nel grafico, le prestazioni rimangano pressoché le stesse del test solo in lettura, questo vuol dire che anche apportando modifiche alla cuckoo, i thread in lettura continuano normalmente a fare lookup senza essere stoppati. Rispetto all'altro grafico le prestazioni calano solo al raggiungimento degli 8 thread, questo perché ci sono altri due thread (delete thread e insert thread) in esecuzione e quindi il calo prestazionale arriva prima.

Capitolo 6

Conclusioni

Il lavoro effettuato durante il tirocinio ha portato allo sviluppo di una cuckoo hash table *lock-free*.

Durante la prima parte del tirocinio, sono state studiate varie hash table, giungendo alla conclusione che la cuckoo fosse quella più adatta per questo progetto perché consente di effettuare ricerche in maniera diretta cioè in tempo costante, consentendo un più veloce tracciamento delle connessioni; poi sono state valutate diverse funzioni di hashing, finché si è deciso di usare il CRC32 di Intel, questo non solo perché il calcolo del *CRC* è semplice e veloce è quindi non ci sono perdite di tempo nel calcolo dell'hash, ma anche perché, in particolare, il CRC32 di Intel viene accelerato dal hardware stesso, risultando ancora più veloce.

Una volta poi implementata la struttura della cuckoo hash table e gli algoritmi di inserimento, lookup, delete e modifica, si è proceduto allo sviluppo di una versione *lock-free* in grado di soddisfare contemporaneamente più richieste/operazioni senza che nessun thread si mettesse in attesa.

Lo sviluppo della versione *lock-free* ha portato allo studio di una particolare istruzione messa a disposizione dai processori Intel a 64 bit, ovvero: CMPXCHG16B, istruzione senza la quale non sarebbe stato possibile effettuare il compare and exchange fra word a 128 bit (dati con un sizeof di 16 byte) e concludere quindi tale lavoro, dato che le entry da memorizzare nella hash table occupano appunto 16 byte.

In particolare sono state usate due funzioni basate su questa istruzione che sono:

- *___atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak, int success_memorder, int failure_memorder)*;
- *void ___atomic_exchange (type *ptr, type *val, type *ret, int memorder)*

Infine il sistema sviluppato è stato testato per valutarne le prestazioni in termini di numero di letture al secondo, al variare del numero di thread concorrenti, che richiedono operazioni di lettura. I dati raccolti dimostrano, che il sistema fornisce un throughput lineare rispetto al numero di thread, anche nel caso in cui ci siano continue richieste di update.

Riferimenti

[1] https://github-com.translate.goog/komrad36/CRC?_x_tr_sl=auto&_x_tr_tl=it-IT&_x_tr_hl=it

[2] https://girondi.net/pub/conntrack21.pdf

[3] Kirsch, Adam, Michael Mitzenmacher, and Udi Wieder. "More robust hashing: Cuckoo hashing with a stash." *SIAM Journal on Computing* 39.4 (2010): 1543-1561

[4] https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html