SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE
INFORMATICA E STATISTICA

Tesi di Laurea Magistrale in

Ingegneria Informatica

Progettazione e implementazione di un meccanismo di rollback parziale per memorie software transazionali

Relatore
Prof. Francesco Quaglia

Candidata
Alice Porfirio

Correlatori
Dott. Pierangelo Di Sanzo
Dott. Alessandro Pellegrini

Anno Accademico 2010/2011

Indice

1 Introduzione **5**

2 Descrizione delle tecnologie usate **8**
2.1 Architetture hardware e software 8
2.1.1 Concorrenza: benefici e problematiche 9
2.1.2 Architettura IA-32 10
2.1.3 Architettura Intel64 13
2.1.4 Il formato ELF 14
2.2 Parser e instrumentatore 16
2.2.1 Fase di parsing 17
2.2.2 Fase di instrumentazione 19
2.2.3 Modifiche alla fase di instrumentazione 21
2.2.4 Modifica dell'ELF 25
2.2.5 Funzionamento del modulo di monitoring 25
2.3 Transactional Locking II 26
2.3.1 Cenni sul concetto di transazione 26
2.3.2 Memorie software transazionali 28
2.3.3 STM basate su lock 29
2.3.4 Transactional Locking II 29
2.4 Lavori correlati 33
2.4.1 Instrumentazione 33
2.4.2 Rollback parziale 34

3 Scelte progettuali **36**
3.1 Analisi delle strutture usate 36
3.1.1 Struttura per il log delle scritture locali 37
3.2 Analisi degli algoritmi 42
3.2.1 Procedura di estensione dello snapshot 43
3.2.2 Protocollo di rollback parziale 46

4 Fase implementativa **50**
4.1 Gestione e reperimento delle informazioni necessarie 51
4.2 Estensione dello snapshot 52
4.3 Instrumentazione delle scritture locali 54
4.3.1 Individuazione e instrumentazione delle sezioni transazionali 55
4.3.2 Individuazione e instrumentazione delle funzioni . . 56
4.4 Costruzione del log delle scritture locali 58
4.4.1 Creazione delle strutture per il log 59
4.4.2 Funzione per la costruzione del log 61
4.4.3 Logica aggiuntiva 63
4.4.4 Cenni sulla fase linking 63
4.5 Protocollo di rollback parziale 64

5 Conclusioni **67**

Bibliografia **69**

Elenco delle figure

2.1 Esempio di deadlock 10
2.2 Formato delle istruzioni IA-32 11
2.3 Modalità di indirizzamento 12
2.4 Il prefisso REX 13
2.5 Struttura di un file ELF 14
2.6 Creazione di un file eseguibile 17
2.7 Struttura insn_info 19
2.8 Strutture Log e AVPair 31

3.1 Struttura dati per il log delle scritture locali 39
3.2 Gestione delle collisioni 41
3.3 Procedura di riallocazione 42
3.4 Procedura di estensione dello snapshot a) riuscita b) fallita . 45
3.5 Rollback Parziale: Ripristino delle variabili locali 48
3.6 Rollback Parziale: Ripristino di read-set e write-set 48
3.7 Rollback Parziale: Ripresa dell'esecuzione 49

Capitolo 1

Introduzione

Un sistema è detto concorrente quando in esso possono esservi in esecuzione più processi o thread contemporaneamente. Attualmente la concorrenza è una delle principali problematiche nell'ambito informatico. Vi è ormai un'ampia diffusione dei multi-core: grazie ad essi otteniamo l'esecuzione parallela di più thread su differenti core e una maggiore velocità nello smaltimento del carico di lavoro. Tuttavia, thread eseguiti su core diversi possono trovarsi ad accedere in maniera concorrente alla stessa area di memoria e ciò può portare a situazioni di incoerenza.

Le memorie software transazionali sono tra i più innovativi meccanismi di gestione della computazione concorrente: intere porzioni di codice vengono viste e eseguite come blocchi atomici e vengono effettuati controlli di letture e scritture su zone di memoria condivisa. Come per le transazioni nell'ambito dei database, anche in questo caso al termine di un'esecuzione possiamo avere un commit (in caso di esecuzione corretta) o un rollback (in presenza di incoerenze).

L'obiettivo di questo lavoro è la progettazione e l'implementazione di un meccanismo che permetta il rollback parziale di transazioni nell'ambito delle memorie software transazionali. Per rollback parziale dopo l'abort di una transazione si intenda la ripresa dell'esecuzione del codice non dall'inizio di essa, bensì dal punto in cui è avvenuta l'incoerenza. Questo permetterebbe di salvare la parte del lavoro eseguita prima dell'occorrenza del problema, avendo potenziali effetti positivi sui costi in termini di tempo.

Per sviluppare questo tipo di meccanismo sono stati focalizzati i seguenti punti:

- **Salvataggio delle operazioni di scrittura su variabili locali:** sarà necessario tenere traccia del lavoro svolto dalla transazione, poiché, in caso di rollback, si dovranno ripristinare le variabili ai loro ultimi valori consistenti.

- **Individuazione del punto che ha portato all'incoerenza:** le operazioni che generano criticità sono le letture di variabili condivise. Di questo tipo di operazioni viene tenuta traccia, mantenendo alcune informazioni in proposito: di fondamentale importanza è il timestamp associato ad ognuna di queste istruzioni.

- **Protocollo di rollback parziale:** ottenute e memorizzate tutte le informazioni necessarie, si passerà alla fase di esecuzione vera e propria. In essa dovranno essere ripristinati gli ultimi valori coerenti e si dovrà far ripartire l'esecuzione dall'istruzione che ha generato problemi in precedenza.

In questa relazione presenterò innanzitutto le architetture e le tecnologie già esistenti su cui si è lavorato e le modifiche che vi sono state apportate per lo scopo. Forniràò quindi, la descrizione di alcuni aspetti architetturali, sia a livello hardware che a livello software, di particolare rilevanza nell'ambito del progetto.

Successivamente, presenterò un instrumentatore supportato da un parser del codice macchina che permette la modifica dell'applicativo in maniera trasparente all'utente. La possibilità di aggiungere moduli logici all'esecuzione, mi ha permesso di analizzare le variabile che vengono modificate nell'ambito di una porzione transazionale.

In seguito, descriverò nei dettagli l'algoritmo per memorie software transazionali su cui si è scelto di lavorare: si tratta del Transactional Locking II (TL2), un algoritmo innovativo basato sulla combinazione di lock a tempo di commit e una nuova tecnica di validazione che usa timestamp globali.

Andrò poi ad analizzare alcune scelte progettuali prese in corso d'opera. L'attenzione si concentrerà sulle strutture dati ideate ed sviluppate per il salvataggio e la gestione dei log delle scritture locali e sul protocollo implementato nell'ultimo step per effettuare la fase del rollback parziale vero e proprio.

Seguirà quindi la descrizione dettagliata dell'implementazione, in tutte le sue fasi: dai cambiamenti apportati all'instrumentatore per permettere la scansione delle sole sezioni transazionali e la loro estensione con la logica necessaria per mappare le scritture locali, all'ampliamento delle funzionalità di TL2, tramite la creazione di nuovi moduli e la modifica di quelli esistenti; infine verrà descritto nei dettagli l'algoritmo di rollback parziale.

In conclusione, dopo un breve riepilogo del lavoro svolto, verranno elencati vantaggi e svantaggi ottenibili tramite il meccanismo implementato in questo progetto.

Capitolo 2

Descrizione delle tecnologie usate

Illustrerò di seguito i dettagli degli strumenti usati per sviluppare questo progetto. Inizierò con il fornire alcune informazioni sul tipo di architettura con cui si è lavorato, in termini sia hardware che software, concentrandomi soprattutto su alcuni aspetti di interesse nell'ambito del progetto. Di seguito vi sarà la descrizione dettagliata delle tecnologie specifiche usate, quali parser e instrumentatore da un lato e i meccanismi del Transactional Locking II per memorie software transazionali dall'altro.

2.1 Architetture hardware e software

Il progetto è stato sviluppato su una macchina **i386** a 64 bit, con quattro processori AMD Opteron 6128 da 2 GHz e 64 Gb di Ram. Ogni processore ha 8 core, per un totale di 32. Tuttavia il lavoro è stato pensato per adattarsi sia a processori a 32 bit sia ad un numero di core variabile, che possa fornire maggiore i minore concorrenza. Il parser usato lavora su architetture con **Instruction Set Intel-compliant** per fornire la più ampia diffusione possibile: si tratta di un instruction set a formato variabile che segue il paradigma **CISC** (Complex Instruction Set Computer). Esso permette una maggiore espressività grazie alla possibilità di eseguire anche operazioni complesse direttamente in memoria.

Per quanto riguarda il lato software, si è lavorato su un sistema operativo **GNU/Linux** con conseguente interazione con il formato di file **ELF** (*Executable and Linkable Format*): nella fase di instrumentazione del codice applicativo infatti, si sono rese necessarie modifiche direttamente nella struttura dei file esaminati.

2.1.1 Concorrenza: benefici e problematiche

Forniamo un breve riepilogo di alcuni concetti fondamentali sulla concorrenza, una delle problematiche alla base di questo lavoro.

Un sistema è detto *concorrente* se è possibile avere in esso un insieme di processi o thread in esecuzione contemporaneamente. Ci troviamo in ambiente concorrenziale nei casi sia di parallelismo reale (come accade nelle architetture multi-processore, in cui vi sono effettivamente più processori che lavorano nello stesso momento), sia di parallelismo solo virtuale (si tratta delle architetture in cui vengono parallelizzati flussi di istruzioni tramite politiche di scheduling). Il parallelismo è una qualità ormai diffusa e ricercata nei sistemi di elaborazione, poiché grazie ad esso è possibile aumentare il throughput e avere quindi una velocità di esecuzione maggiore.

Tuttavia, al contrario di quanto accade nei sistemi *sequenziali* (in cui i processi vengono eseguiti uno alla volta), i sistemi concorrenti possono portare ad interazioni tra computazioni parallele ed esse possono generare varie tipologie di problematiche.

Più processi che lavorano in parallelo si trovano spesso ad accedere ad aree di memoria condivisa: ciò può portare al cosiddetto problema delle *corse critiche*, situazione in cui il risultato di una computazione eseguita da più processi in parallelo, dipende dall'ordine di esecuzione delle. Questo implica la necessità di un controllo degli accessi a tali aree di memoria per evitare il crearsi di stati incoerenti e risultati impredicibili.

Entriamo così nell'ambito della *mutua esclusione*, che comprende un insieme di meccanismi per evitare che due o più processi che si contendono una risorsa condivisa, possano accedervi contemporaneamente. Tra le tecniche più conosciute e usate vi sono *mutex*, *semafori* e *spinlock*, variabili con le quali è possibile riservare e rilasciare una risorsa condivisa, tramite il loro blocco e sblocco. Per riservare risorse è inoltre possibile l'uso di *messaggi* tra processi di sistemi paralleli; mentre per bloccare l'esecuzione di processi concorrenti vi sono primitive come wait (mette in stato di attesa un processo) e *signal* (risveglia un processo in attesa) o tecniche come le *barriere*, che fermano l'esecuzione dei processi fin quando tutti non sono giunti ad un certo punto della computazione e sono quindi pronti a passare allo stadio successivo. Di tutte le azioni che riguardano blocco e sblocco di risorse e processi deve tuttavia essere garantita l'atomicità: ovvero ognuna di esse deve essere eseguita come fosse un'unica istruzione.

Si ricordi, però, come la stessa mutua esclusione possa generare problemi se non è implementata in maniera efficiente: la cattiva gestione di blocchi e sblocchi di processi e risorse può portare a situazioni di *starvation* o, nelle ipotesi peggiori, di *deadlock*. Con starvation si intenda la situazione in cui vi sono processi sfavoriti dalle politiche di mutua esclusione: questi processi potrebbero potenzialmente restare bloccati in uno stato di attesa perenne mentre gli altri lavorano sulle risorse condivise. Il problema dei deadlock è più spinoso, in quanto comporta lo stallo dell'intero sistema: esso si verifica quando vi sono processi bloccati in attesa di risorse che sono in quel momento riservate da processi a loro volta in attesa di risorse riservate dai primi processi. Nessuno verrà sbloccato e il sistema smetterà inevitabilmente di funzionare.

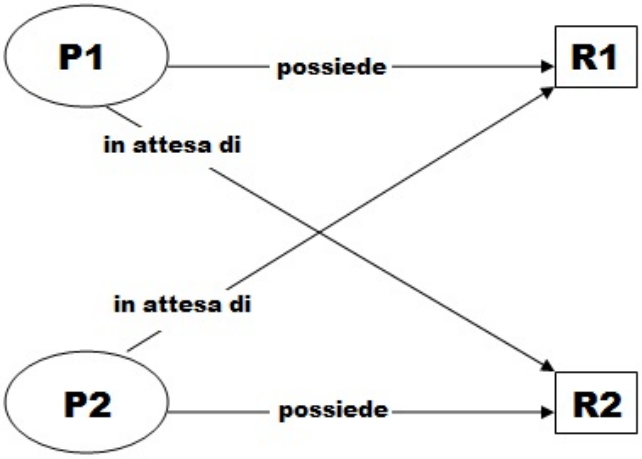

Figura 2.1: Esempio di deadlock

2.1.2 Architettura IA-32

L'architettura che andiamo a descrivere è quella usata nei microprocessori prodotti dalla Intel e dalla AMD: è un Instruction Set che presenta istruzioni a formato variabile con un'alta espressività semantica.

L'architettura Intel a 32 bit (IA-32) [11] prevede, di base, 8 registri a 32 bit detti general-purpose: sono, nell'ordine di codice numerico, eax, ecx, edx,

ebx, esp, ebp, esi ed edi. In particolare, il registro esp (stack pointer) contiene il riferimento in memoria alla cima dello stack, mentre nel registro ebp (base pointer) è salvato il riferimento in memoria alla prima locazione di memoria dello stack corrente. Sono poi previsti sei segment-registers da 16 bit: CS, DS, SS, ES, FS, GS. Non sono tuttavia usati nei sistemi operativi Unix. Infine sono presenti i registri EFLAGS (che raccoglie vari indicatori che descrivono lo stato del programma) e EIP (l'instruction pointer che punta all'istruzione successiva).

Vediamo ora qualche dettaglio sul formato delle istruzioni dell'IA-32 e i campi che possono essere presenti.

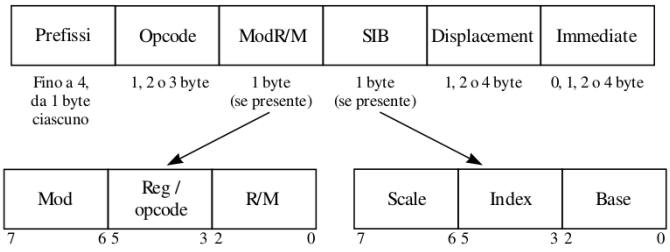

Figura 2.2: Formato delle istruzioni IA-32

I **prefissi** sono divisi in quattro gruppi: per ogni gruppo può essere presente al più un solo prefisso. Il gruppo 1 comprende i prefissi che permettono di ripetere listruzione immediatamente successiva, finchè una determinata condizione non risulti verificata. Il gruppo 2 contiene i prefissi che indicano l'override di segmento, utile nel caso di accesso a dati. Ai gruppi 3 e 4 appartengono quei prefissi che specificano se il dato immediato (gruppo 3) o lindirizzo in memoria (gruppo 4) hanno dimensioni differenti rispetto a quanto indicato dallopcode.

Il campo **opcode** è anchesso di formato variabile. Il primo byte identifica una classe di istruzioni: se lopcode è composto da un solo byte, allora identifica univocamente unistruzione; se, invece, è composto da più byte, il primo di essi corrisponde ad una famiglia di istruzioni che condividono la semantica o i campi utilizzati o la rappresentazione dei dati adottata.

All'opcode primario può far seguito un campo di 3 bit (il reg/opcode), allinterno del byte **ModR/M**. Le istruzioni che devono riferire un operando in memoria necessitano di un byte che indica la forma di indirizza-

mento usata. Questo byte prende è detto ModR/M ed è composto da tre sottocampi:

- Il campo **mod**, che viene combinato con il campo r/m per formare 32 possibili valori: 8 registri e 24 modalità di indirizzamento;
- Il campo **reg/opcode**, che indica il numero di un registro, oppure i tre bit aggiuntivi dellopcode precedentemente citati;
- Il campo **r/m**, che può specificare un registro come operando o può essere combinato con il campo mod per indicare la modalità di indirizzamento.

Per alcune codifiche di ModR/M vi è l'aggiunta di un ulteriore byte pensato per lindirizzamento: è il **SIB**. Anch'esso è formato da tre sottocampi:

- Il campo **scale**, che specifica il fattore di scala e il cui valore può essere pari a 1, 2, 4 o 8;
- Il campo **index**, che indica il numero del registro indice;
- Il campo **base**, che indica il registro di base.

Vi sono modalità di indirizzamento che necessitano di uno spiazzamento (il **displacement**): viene indicato subito dopo il byte ModR/M (o il byte SIB, se presente) e può avere una lunghezza pari a 1, 2 o 4 byte, dipendente dall'istruzione che lo utilizza.

$$\left\{\begin{array}{c}\text{CS:}\\ \text{DS:}\\ \text{SS:}\\ \text{ES:}\\ \text{FS:}\\ \text{GS:}\end{array}\right\}\left[\left\{\begin{array}{c}\text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{ESP}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{array}\right\}\right]+\left[\left\{\begin{array}{c}\text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{array}\right\}*\left\{\begin{array}{c}1\\ 2\\ 4\\ 8\end{array}\right\}\right]+[displacement]$$

Figura 2.3: Modalità di indirizzamento

2.1.3 Architettura Intel64

Mi concentrerò ora sulle architetture a 64 bit, che nella famiglia Intel vengono indicate come Intel64. Esse costituiscono un sovrainsieme dellarchitettura IA-32: con questo di intende che un processore x86-64 ha la possibilità di eseguire sia codice a 32 bit, sia codice a 64 bit., indistintamente.

L'architettura x86-64 presenta un formato di istruzione praticamente identico al corrispettivo a 32 bit. La differenza più evidente la troviamo nella presenza di un nuovo prefisso, denominato REX, posto tra i 4 prefissi preesistenti e lopcode principale.
Il prefisso **REX**, ha i primi 4 bit preimpostati a 0100, mentre i restanti quat-

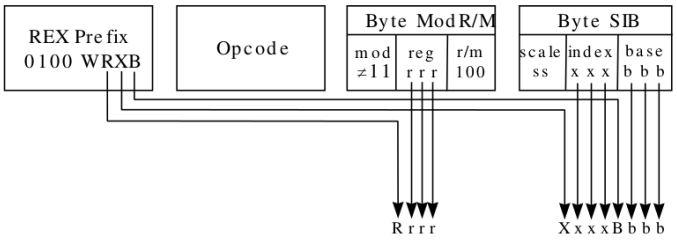

Figura 2.4: Il prefisso REX

tro bit permettono di estendere larchitettura a 32 bit. In particolare, hanno i seguenti significati:

- **REX.W**, settato a 1, indica che la dimensione degli operandi coinvolti nellistruzione è di 64 bit;
- **REX.R** è unestensione del campo reg del byte ModR/M;
- **REX.X**, invece, è unestensione del campo index del byte SIB;
- **REX.B**, infine, può essere unestensione del campo r/m del byte ModR/M, del campo base del byte SIB oppure del campo reg dellopcode.

In un'istruzione possono essere presenti diversi byte REX consecutivi, tuttavia soltanto l'ultimo immediatamente precedente all'opcode principale verrà considerato e letto: questo viene specificato dalla definizione dell'architettura.

Per quanto riguarda gli altri campi del formato delle istruzioni, essi in genere restano identici a quelli relativi all'architettura a 32 bit. Questo vale anche per le dimensione dei suddetti campi: anch'esse vengono mantenute a 32 bit.

2.1.4 Il formato ELF

Il formato ELF (Executable and Linkable Format) un formato di file che permette di rappresentare codice rilocabile (generato da assemblatori o linker), un eseguibile (un programma pronto per l'esecuzione) o librerie condivise [20].

Un file ELF è formato da varie sezioni che saranno presenti o meno a seconda di ciò che il file vorrà rappresentare:

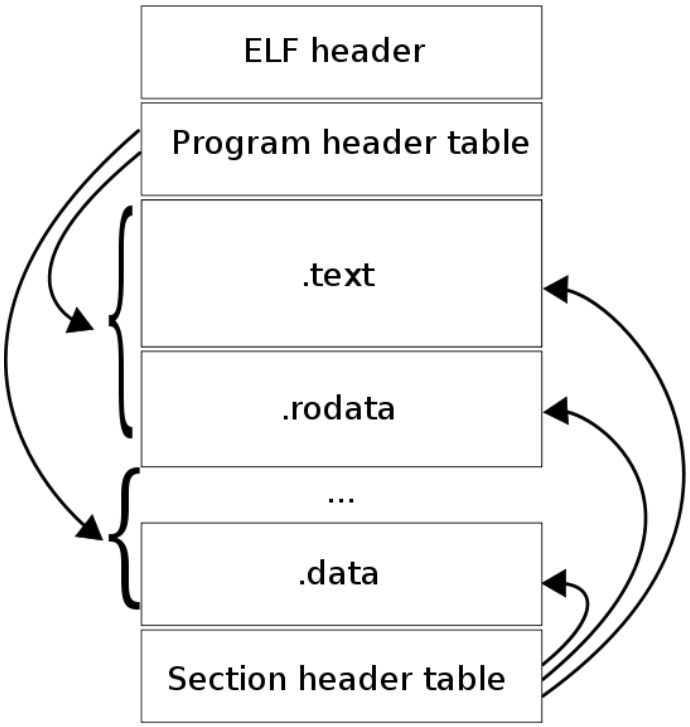

Figura 2.5: Struttura di un file ELF

All'inizio, è sempre presente in tutte le tipologie di file ELF, un header generale (**ELF Header**) che descrive la struttura del file: esso fornisce informazioni riguardanti il tipo di codice contenuto nel file, indica dove possono essere trovate le tabelle degli header del programma e degli header di sezioni, tramite degli offset dallinizio del file.

Gli **header del programma** servono per indicare al sistema come creare l'immagine del processo, come organizzare i segmenti e i privilegi da assegnare. Gli **header delle sezioni** forniscono informazioni sulle sezioni presenti nel file come dimensione, nome e attributi; inoltre indicano la tabella di rilocazione associata alla singola sezione. Per gli oggetti eseguibili sono necessari gli header di programma, mentre i rilocabili devono presentare gli header di sezione.

Vediamo qualche dettaglio in più sulla struttura dei file rilocabili, che saranno quelli presi in considerazione e modificati dall'instrumentatore.

Come detto, in questo caso sono presenti sicuramente gli header delle sezioni: essi sono organizzati in una tabella in cui ciascun elemento identifica una sezione, specificandone il suo nome, gli attributi, la dimensione e la posizione nel file tramite offset dallinizio di esso. Le sezioni che hanno al loro interno del codice che riferisce delle variabili vengono associate ad una tabella di rilocazione, posta all'interno di unaltra sezione: essa fornisce al linker la posizione nel codice dei riferimenti alle variabili e alle chiamate alle funzioni. Il linker potrà così effettuare la sostituzione di indirizzo, se necessaria. Le due sezioni sono collegate tra loro tramite il campo *sh_info* della tabella di rilocazione.

Le **tabelle di rilocazione** sono formate da una serie di elementi formati dai seguenti campi:

- **r_offset** che indica la posizione su cui operare lazione di rilocazione.
- **r_info** che fornisce l'indice allinterno della tabella dei simboli.
- **r_addend** che definisce una costante da sommare allindirizzo che verrà sostituito nelloperazione di rilocazione. E' un campo opzionale.

Tramite il campo *r_info* è possibile estrarre un indice relativo alla tabella dei simboli specificata da *sh_link*, all'interno dellheader della sezione.

La **tabella dei simboli** contiene le informazioni relative a tutte le variabili e a tutte le funzioni presenti nel file, in qualsiasi sezione esse si trovino. I campi di interesse nel nostro caso, poichè usati in seguito dall'instrumentatore, sono: *st_name* (che contiene lo spiazzamento allinterno della tabella delle stringhe ed in essa troveremo il nome della variabile o funzione riferita) e *st_value* (che corrisponde al riferimento del simbolo nel codice, inteso come offset dall'inizio di esso).

La tabella delle stringhe è composta da sequenze di caratteri ognuna delle quali termina con il carattere *NULL* . Queste stringhe vengono usate per rappresentare i nomi di simboli e sezioni vengono riferite tramite indici (come nel caso di st_name sopraccitato) allinterno della tabella. Se un riferimento punta alla stringa di indice 0, l'oggetto in questione non ha nome o ha nome nullo.

2.2 Parser e instrumentatore

Dopo aver fornito un riepilogo su alcuni aspetti delle architetture su cui si è lavorato e sulle strutture con cui si è interagito, scendiamo nei dettagli di due strumenti basilari in questo progetto: parser e instrumentatore.

Il loro lavoro in coppia fornisce la possibilità di modificare il codice applicativo (nella forma dell'appena illustrato formato ELF) in maniera totalmente trasparente rispetto all'utente: una volta fornito in input il file di interesse, avremo in output un oggetto a cui è stata aggiunta logica applicativa, senza che l'utente dell'applicazione abbia dovuto scrivere alcuna riga di codice.

Sarà l'instrumentatore, infatti, ad avere il compito di aggiungere la suddetta logica richiesta in alcuni punti specifici del codice, selezionati a seconda delle necessità del lavoro che si vuole svolgere. Per trovare questi punti, l'instrumentatore si serve del parser che è in grado di disassemblare l'intero codice nelle sue istruzioni assembly, facendo riferimento all'Instruction Set Intel-Compliant precedentemente descritto.

L'instrumentazione è una tecnica con un grande potenziale, ma deve tenere in considerazione le tre seguenti principali problematiche [15]:

- si lavora a livello di codice macchina e vengono inseriti nel file ELF stream di byte corrispondenti ad istruzioni;

- per ottenere la trasparenza nei confronti dell'utente è necessario preservare la coerenza dei riferimenti interni al codice;
- è necessario individuare le posizioni in cui inserire il codice di instrumentazione, in modo da poter poi interpretare il codice originale del programma.

Il lavoro dell'instrumentatore va quindi a posizionarsi tra quello del compilatore e quello del linker: sarà da quest'ultimo che otterremo infine il codice eseguibile.

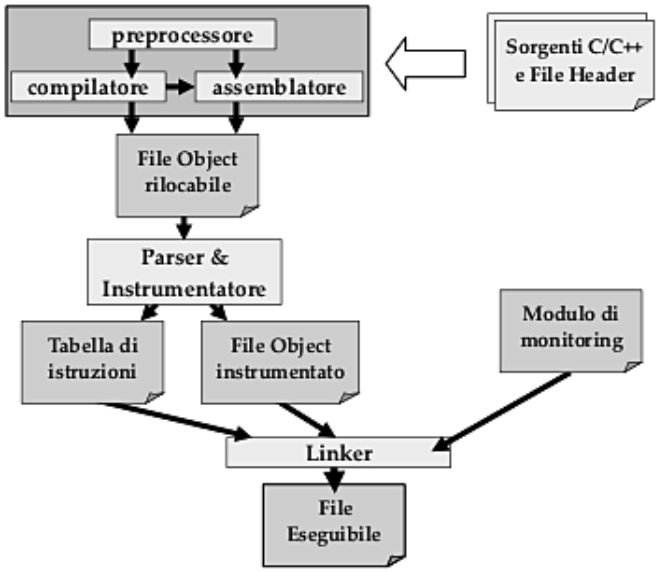

Figura 2.6: Creazione di un file eseguibile

Nei prossimi paragrafi sarà fornita sia la spiegazione del funzionamento di questi due strumenti, sia la descrizione delle modifiche che vi sono state apportate per adattare il loro lavoro al caso di studio.

2.2.1 Fase di parsing

Lavorando sull'Istruction Set Intel-Compliant, il parser ha a che fare con istruzioni la cui lunghezza non è conosciuta a priori. Esso deve quindi essere in grado di riconoscere ogni istruzione definita dalla suddetta architettura: per fare ciò, esso sfrutta una tabella in cui sono raggruppate, in ordine di opcode, le famiglie di istruzioni.

Ogni entry della tabella corrisponde ad una **struct insn** che fornisce tutte le informazioni utili sull'istruzione identificata. In particolare, vengono indicati: mnemonico dellistruzione (ovvero il suo nome in codice assembly), metodo di indirizzamento di ciascun operando dellistruzione (registro, indirizzo in memoria, offset...), tipo di operando (informazioni sulla dimensione standard e la tipologia di registri in cui può essere trovato), la funzione del parser (atta ad analizzare la classe a cui l'istruzione appartiene) e alcuni flags. Tali flags si riveleranno utili nell'ambito del progetto, fornendo ulteriori informazioni sulla semantica dell'istruzione: ad esempio se opera in memoria (I_MEMWR per le scritture e I_MEMRD per le letture), se è un'istruzione di salto (I_JUMP o, caso di particolare interesse per questo progetto, se opera nello stack.
L'accesso alla tabella è immediato dopo la lettura del primo byte dell'istruzione, poiché le entry sono ordinate per opcode crescente.

E' la funzione **x86_disassemble_instruction** che, richiamata in un ciclo, è in grado di esplorare il codice byte a byte e interpretare l'istruzione che si trova davanti. Il primo step è l'individuazione di quanti e quali siano i prefissi presenti (compresi i prefissi REX, nel caso di codice a 64 bit). Di seguito, si andrà a leggere il primo byte di opcode, da cui, sfruttando la tabella, si potranno ricavare le informazioni sopra-elencate. Tra le altre, vi è la funzione usata per il parsing della famiglia a cui l'istruzione appartiene: tra le altre operazioni che essa svolge, vi è il settaggio di alcuni dei flags che definiranno la semantica dell'istruzione analizzata.

Dopo aver controllato la presenza dei byte ModR/M e SIB e dell'eventuale spiazzamento, si passa alla fase di analisi degli operandi: per ognuno di essi viene invocata una funzione che estrae informazioni relative al formato e alla specie dell'operando.

Di particolare interesse per questo progetto è la funzione **format_addr_m** : il suo compito è quello di gestire gli operandi che rappresentano locazioni in memoria. Le istruzioni di interesse nel nostro caso sono quelle che effettuano write nello stack, ovvero scritture su variabili locali: ho perciò definito un nuovo flag, **I_STACK**, che viene settato in format_addr_m quando si è in presenza di un'istruzione che opera nello stack. Quando l'operando analizzato si trova tra base e stack pointer, il nuovo flag viene aggiunto ai precedenti, con la seguente istruzione:

```
state->instrument->flags = I\_{}STACK \|{} state->instrument->flags;
```

Questa nuova informazione ottenuta sarà cruciale nella fase instrumentativa, poiché ci permetterà di discriminare in modo veloce le istruzioni su cui bisognerà operare.

Al termine di queste operazioni, *x86_disassemble_instruction* avrà riempito tutti i campi di una nuova struttura, la **struct insn_info**: passata inizialmente tra i parametri di input, sarà essa che fornirà tutte le informazioni trovate che si riveleranno necessarie nella successiva fase instrumentativa. Forniamo la struttura completa nella figura 2.5.

```
struct insn_info {

        unsigned char insn[15];  // I byte dell'istruzione
        unsigned char opcode[2]; // L'opcode dell'istruzione
        char mnemonic[8];        // Il nome dell'istruzione
        unsigned long initial;   // Posizione iniziale nel testo
        unsigned long insn_size; // Lunghezza dell'istruzione
        unsigned long addr;      // Indirizzo puntato dall'istruzione
        unsigned long span;      // Quanto in memoria verrà riscritto/letto

        bool has_index_register; // L'indirizzamento sfrutta un indice?
        unsigned char ireg;      // Quale registro contiene l'indice?
        char ireg_mnem[8];       // Mnemonico del registro di indice
        bool has_base_register;  // L'indirizzamento sfrutta una base?
        unsigned char breg;      // Quale registro contiene la base?
        char breg_mnem[8];       // Mnemonico del registro
        bool has_scale;          // L'indirizzamento utilizza una scala
        unsigned long scale;     // La scala
        unsigned long disp_offset; // Lo spiazzamento dall'inizio del testo
        int32_t jump_dest;       // Dove punta la jmp

        bool uses_rip;

        unsigned long flags;     // Flags contenenti informazioni utili

};
```

Figura 2.7: Struttura insn_info

2.2.2 Fase di instrumentazione

La fase instrumentativa si può suddividire in tre fasi: il calcolo dell'incremento della taglia del file (incremento necessario, in vista dell'aggiunta di ulteriore logica e quindi nuove istruzioni), la decodifica vera e propria (dove si inseriscono effettivamente le chiamate al nuovo modulo assembler) e, infine, la correzione dei salti.

A queste tre fasi corrispondono altrettante funzioni, richiamate ciclicamente dall'instrumentatore, che scorre le varie sezioni del file ELF in input e lavora su quelle contententi codice eseguibile. Le tre funzioni sono le seguenti:

- **compute_size_increment:** come detto, il suo scopo è il calcolo dell'incremento necessario per la taglia del file. Si scorre, quindi, una prima volta il codice usando il parser: un puntatore segna la posizione corrente che viene incrementata direttamente dalla funzione x86_disassemble_instruction e il ciclo terminerà quando si sarà raggiunta la fine della sezione. Si è alla ricerca sia di quelle istruzioni che dovranno essere in seguito instrumentate, sia di istruzioni di salto, che al termine del processo dovranno essere corrette. Tali istruzioni vengono riconosciute controllando il campo flags della struttura insn_info costruita dal parser: esso ci fornisce la semantica dell'istruzione e quindi ci permette di individuare rapidamente ciò che stiamo cercando. La funzione alla fine restituirà l'incremento totale necessario che verrà poi usato dall'instrumentatore per calcolare la taglia della nuova sezione su cui verrà scritto il codice applicativo modificato.

- **decode_code_section:** rappresenta il cuore dell'instrumentatore. Ancora una volta il codice viene decodificato tramite il parser e vi sono controlli istruzione per istruzione sui flags della struttura *insn_info*. In questa fase, vi è l'inserimento delle nuove call (una chiamata al nuovo modulo assembly con la logica aggiuntiva detto monitor per ogni istruzione da instrumentare), il salvataggio dei riferimenti che andranno corretti direttamente nella struttura del file (riferimenti che vanno dall'indirizzo di destinazione di un salto, al punto in cui va innestata la chiamata al monitor) e la copia di ogni istruzione, con relative aggiunte, nella nuova sezione.

- **correct_jumps:** l'ultima fase si occupa della correzione di tutte le istruzioni di salto trovate nel codice. Dopo l'inserimento delle varie chiamate a monitor, la maggior parte di tali istruzioni devono subire modifiche sia nel loro indirizzo di destinazione sia nella dimensione del salto. Alcune istruzioni andranno perciò totalmente sostituite con delle nuove che tengano conto dei cambaimenti effettuati. Ancora una volta viene invocato il parser in un ciclo, ancora una volta la tipologia di istruzione viene riconosciuta tramite i flags. Per ogni salto viene chiamata una funzione che calcola il nuovo indirizzo di destinazione sfruttando le informazioni sugli shift salvate nella fase pre-

cendente. Infine, l'ultima funzione richiamata, build_correct_jump, effettuerà la correzione vera e propria dell'istruzione.

Terminate le tre fasi, l'instrumentatore andrà a sostituire la vecchia sezione con la nuova appena creata e ad aggiornare la struttura dell'ELF.

2.2.3 Modifiche alla fase di instrumentazione

Si vedrà ora come il processo di instrumentazione di base appena descritto sia stato dovuto modificare per essere adeguato alle esigenze del progetto.

La prima considerazione da fare è la seguente: l'instrumentatore così impostato instrumenta indistintamente tutto il codice eseguibile passatogli in input, dall'inizio alla fine di ogni sezione da modificare. Necessaria in questo lavoro, invece, è l'instrumentazione delle sole porzioni di codice dette transazionali. Quindi, si è innanzitutto dovuto trovare un modo per demarcare nel codice le suddette porzioni, un modo che fosse poi riconoscibile e sfruttabile dall'instrumentatore.

Gli strumenti usati per questo scopo sono state delle *label*: mentre delle generiche label definite in C all'interno codice applicativo non sembravano sopravvivere nel nuovo file oggetto (non comparivano nella tabella dei simboli e perciò non erano utilizzabili), la loro definizione tramite un modulo in inline-assembly garantiva questa sopravvivenza.
Sono state scritte, perciò, le seguenti *define*:

```
#define STRINGIFY(x) #x
#define TOSTRING(x) STRINGIFY(x)

#define MARK_TX_BEGIN()   __asm__ __volatile__
                          ("begin_"__FILE__TOSTRING(__LINE__) ":");

#define MARK_TX_END()     __asm__ __volatile__
                          ("end_"__FILE__TOSTRING(__LINE__) ":");
```

Le prime due si sono rese necessarie per il processo di stringificazione della macro __LINE__ che corrisponderebbe ad un intero. Le ultime due sono le definizioni vere e proprie delle label cercate: esse sono pensate per marcare l'inizio (begin_) e la fine (end_) di una porzione transazionale. Sono inoltre utilizzabili più volte anche nell'ambito dello stesso file: l'uso delle macro **__FILE__** (che concatena il nome del file corrente) e **__LINE__** (che concatena il numero di riga in cui la macro è richiamata) garantiscono l'unicità della label così creata e quindi la riusabilità e la trasparenza della define.

Le suddette define andranno inserite all'interno del codice del transactional locking II, l'algoritmo per memorie software transazionali che andrò ad illustrare negli ultimi paragrafi di questo capitolo. Richiamandole all'interno del codice applicativo otterremo, ad esempio:

```
begin_main.c_50:

end_main.c_62:

begin_main.c_68:

end_main.c_77:
```

Una volta trovato il modo per segnalare le porzioni di interesse, lo step successivo è stato permettere all'instrumentatore di trovare queste nuove label, salvarne gli indirizzi corrispondenti e usarli per riconoscere il codice su cui poi andare a lavorare. Per questi scopi sono state implementate le seguenti due funzioni, che agiscono direttamente sulla struttura e le tabelle dell'ELF:

- **count_symbol:** passando in input una stringa che rappresenta il nome (o anche solo parte del nome) del simbolo e il file ELF in cui cercare, questa funzione restituisce il numero di occorrenze di tale simbolo nel codice. La funzione, dopo aver recuperato la tabella delle stringhe del file, cicla su di essa alla ricerca di simboli i cui nomi contengano la stringa in input e ne conta le occorrenze. Usandola nell'instrumentatore sulle stringhe begin e end otteniamo così, sia il numero di simboli di cui cercare il riferimento, sia il numero di porzioni transazionali presenti nel file.
- **find_symbol:** in questo caso si ha come input non soltanto la stringa con il nome del simbolo e il file, ma anche un intero che rappresenta il numero di occorrenze di quel nome di cui ancora bisogna trovare il riferimento. Questo contatore ci permetterà di ignorare le label già prese in considerazione. Scopo di questa funzione è trovare il riferimento di un dato simbolo, fornito come offset dall'inizio della sezione in cui tale simbolo compare. Ancora una volta si andrà a recuperare la tabella delle stringhe dell'ELF e, ciclando su essa, verrà cercato il nome del simbolo: di esso interessa la posizione che ha nella tabella, inteso come spiazzamento dall'inizio di essa. Ottenuto questo valore, si dovrà estrarre e scorrere un'altra tabella presente

nell'ELF: si tratta della tabella dei simboli, che contiene le informazioni da noi cercate. Ogni entry di essa, infatti, corrisponde ad una struttura composta da vari campi: due sono i campi di interesse per il nostro scopo. Il primo è *st_name*, che sarà confrontato con il valore ottenuto dalla ricerca sulla tabella delle stringhe: in caso di esito positivo si sarà trovato la struttura associata al simbolo cercato e si potrà estrarre da essa il secondo valore, ovvero *st_value*. Esso è il riferimento che serve: rappresenta lo spiazzamento del simbolo dall'inizio del codice e quindi, in questo ambito, l'inizio o la fine della porzione transazionale che andremo ad instrumentare.

A questo punto, si ha il numero di porzioni transazionali e il modo per trovarne l'inizio e la fine nel codice. Prima di passare alla vera fase di instrumentazione selettiva, sono necessarie ancora alcune informazioni sul codice su cui si lavorerà: nello specifico, è di interesse quante e quali funzioni (se presenti) sono richiamate all'interno delle porzioni transazionali, poiché anch'esse dovranno essere instrumentate.

Anche in questo caso, è stata necessaria l'implementazione di due nuove funzioni che permettessero il conteggio e l'individuazione delle chiamate all'interno del codice transazionale:

- **find_function:** richiamata per ogni porzione transazionale, non si limita a calcolare il numero di chiamate a funzione presenti, ma ne salva il nome in una lista di strutture appositamente creata. Questa lista mantiene i nomi delle funzioni che in seguito andranno instrumentate per la sezione dell'ELF correntemente analizzata. Il codice viene letto tramite un ciclo che chiama la seconda funzione che descriveremo di seguito. E' da notare che *find_function* è ricorsiva: infatti, per ogni funzione trovata, richiamerà se stessa sulla nuova porzione di codice che inizia dalla suddetta chiamata. Questo è necessario poiché sono di interesse anche le funzioni chiamate all'interno di funzioni instrumentate.

- **search_call:** richiamandola dalla funzione precedentemente descritta, sono passati in input non soltanto il file e la sezione corrente, ma anche inizio e fine della porzione che stiamo analizzando. Ancora una volta si farà uso della funzione di parsing che permette di scorrere tutte le istruzioni assembly del codice. In questo caso si è alla ricerca delle istruzioni call (individuabili tramite il campo flags e il campo mnemonic della struttura insn_info fornita dal parser) che

compaiono tra l'indirizzo di inizio e quello di fine. Avendo l'indirizzo di tali istruzioni, è possibile facilmente ottenere il nome della funzione chiamata, usando la *get_symbol_name_by_reloc_position*. La stringa contenente il nome verrà restituita a *find_function*, che, come detto in precedenza, provvederà al suo salvataggio nella lista.

Ora abbiamo l'elenco dei nomi delle funzioni da instrumentare: per trovarne le posizioni nel codice, è possibile usare la stessa *find_symbol* sfruttata in precedenza per trovare gli offset delle label. Questo è possibile perchè anche i nomi delle funzioni sono visti come simboli in un file ELF.

Ottenuti così anche questi ultimi riferimenti, si può avviare l'instrumentazione selettiva. Per ogni sezione del file da instrumentare, si hanno così due cicli: nel primo vengono tenute in considerazione le porzioni transazionali identificate dalle label precedentemente descritte. Per ogni porzione si cercano prima le funzioni in essa chiamate (*find_function*) e poi si richiamano le prime due fasi dell'instrumentazione base: il calcolo dell'incremento della taglia della sezione e l'aggiunta delle chiamate al monitor. Nel secondo ciclo si passa all'instrumentazione delle funzioni trovate in precedenza: per ogni stringa presente nella lista creata, si cerca il suo offset e anche in questo caso si richiamano le prime due fasi instrumentative.

Si noti che le funzioni che gestiscono le prime due fasi dell'instrumentazione hanno adesso bisogno di avere in input gli offset di inizio e di fine porzione: solo le istruzioni comprese in questi due estremi verranno prese in considerazione nel lavoro da loro svolto.

Ho scelto di spostare la terza fase dell'instrumentazione, ovvero la correzione dei salti, dopo la fine dei due cicli: questo mi ha permesso di effettuarla in un unico step, senza doverla richiamare ad ogni passo iterativo. Di contro, questa scelta ha portato alla necessità di mantenere i riferimenti ai salti sempre aggiornati: ogni step instrumentativo infatti, può modificare questi e tutti gli altri riferimenti che vengono mantenuti per il successivo lavoro di modifica dell'ELF. Per questo, al termine di ogni chiamata a *decode_code_section* è stata aggiunta una fase di aggiornamento di tutti i riferimenti a indirizzi che vengono salvati nell'instrumentazione.

Terminata la correzione dei salti, la nuova sezione è pronta e, se non ve ne sono altre da instrumentare, è possibile passare alla fase finale, la modifica dell'ELF, al termine della quale si avrà il nuovo file instrumentato:

ricordiamo che tutto il lavoro appena illustrato viene svolto in maniera totalmente trasparente rispetto all'utente.

2.2.4 Modifica dell'ELF

Come detto, la modifica della sezione contenente il codice rende necessario modificare anche alcuni riferimenti: in particolare quelli che puntano alle locazioni in cui il linker, quando andrà a generare leseguibile finale, dovrà svolgere le operazioni di rilocazione . Questa operazione di shift viene effettuata in particolare a tutte le entry di rilocazione relative a simboli e funzioni in posizioni successive al codice instrumentato .

Le tabelle di rilocazione verranno aggiornate sfruttando le informazioni sulle modifiche ai riferimenti che sono state collezionate durante tutta la fase instrumentativa e salvate in diversi file.

Al termine di queste operazioni, avverrà l'aggancio, al monitor, il modulo in assembly che conterrà la nuova logica. Nel nostro caso, come vedremo più avanti, il monitor avrà il compito di segnalare le variabili locali che subiscono scritture e di salvarne l'indirizzo, la taglia e il nuovo valore.

2.2.5 Funzionamento del modulo di monitoring

Vediamo brevemente come il modulo assembly aggiunto al codice applicativo riesca a tenere traccia delle istruzioni di interesse, estraendone le informazioni di cui necessitiamo (nel nostro caso indirizzo e taglia relativi alle scritture sullo stack).

Ogni volta che un'istruzione di scrittura nello stack sta per essere eseguita, si effettua la chiamata al nuovo modulo: esso è scritto in codice assembly per mantenere minimo l'overhead del tracciamento degli accessi.
La funzione di monitoring, innanzitutto, salva nello stack una fotografia dello stato del processore al momento della chiamata: vengono salvati i valori dei registri al momento della chiamata della funzione, in ordine di codice numerico, e l'EFLAGS. Il registro ebp viene modificato in modo che punti al valore originale di eax.

Viene quindi recuperato nello stack il valore di ritorno della funzione chiamante: è il valore che costituirà la chiave per laccesso alla tabella degli indirizzi, tabella strutturata come in forma di hash, e quindi con un tempo di

accesso molto efficiente. Dalla riga nella tabella viene letto il campo flags che indica il tipo di scrittura stia avvenendo (tipo mov o tipo stringa).

Se abbiamo un'istruzione di tipo stringa (movs o stos), la procedura è la seguente. Il registro edx contiene lo spiazzamento in byte allinterno della tabella per accedere alla riga associata allistruzione e sommando 4 byte si può ottenere accesso al campo size.
Viene recuperato dallo stack il valore originale del registro edi che contiene lindirizzo di destinazione iniziale della scrittura: con un parametro all'interno del registro EFLAGS, si determina se la scrittura procede in avanti o allindietro, partendo da quell'indirizzo. Se necessario, a questo punto viene sottratta la taglia della scrittura allindirizzo di base , calcolando così un nuovo indirizzo di base.

Nel caso di istruzioni di tipo mov, vengono effettuati dei controlli per verificare se l'indirizzo utilizza i campi indice e base. Se è presente un indice, del valore del registro viene calcolato il complemento a due e moltiplicato per la dimensione del registro: questo sarà utilizzato come spiazzamento all'interno della finestra dello stack. A questo punto il valore della scala viene recuperato dalla tabella del monitor e moltiplicato per il valore di indice appena calcolato.
Se è presente una base, è recuperato dallo stack il contenuto del registro originale e sommato allindirizzo in fase di calcolo. Infine, viene caricata la dimensione della scrittura nel registro esi.

Ora, indipendentemente dal tipo di istruzione, il registro esi conterrà la taglia della scrittura e il registro edi il suo indirizzo iniziale.

2.3 Transactional Locking II

Concentriamoci ora sullo strumento alla base di questo progetto: le memorie software transazionali [10] e, in particolare, il transactional locking II, ovvero l'algoritmo con cui si è scelto di lavorare.

2.3.1 Cenni sul concetto di transazione

Riassumiamo brevemente alcuni concetti base sulle transazioni, strumenti base ampiamente usati nell'ambito informatico, in particolare dei Database Management Systems.

Una transazione è una porzione di codice iniziato da un'istruzione di **begin** e terminato da un'istruzione di **end**: alla fine di essa, vi è un controllo che deve validare o meno il lavoro svolto. In caso positivo, si avrà un **commit**, le modifiche diverranno permanenti e si procederà con l'esecuzione delle operazioni successive. In caso negativo, si avrà un **abort** e il conseguente **rollback**, ovvero la ri-esecuzione della transazione dall'inizio.

Una transazione deve garantire alcune proprietà per prevenire gli abort. Queste proprietà sono identificate dall'acronimo **ACID** e sono le seguenti:

- **Atomicità:** una transazione è vista come un'unica azione indivisibile e pertanto va eseguita in tutte le sue parti o per nulla;
- **Coerenza:** all'inizio e al termine della transazione, il sistema deve trovarsi in uno stato coerente;
- **Isolamento:** l'esecuzione di una transazione non deve interferire in alcun modo con l'esecuzione delle altre;
- **Durevolezza:** a commit effettuato, i cambiamenti apportati dalla transazione devono diventare permanenti.

La violazione di una di tali proprietà (atomicità e isolamento in particolare) può portare a stati inconsistenti. Tra le anomalie più note riscontrabili, vi sono le seguenti:

- **Perdita di aggiornamenti:** se due transazioni scrivono consecutivamente su una variabile condivisa, il primo aggiornamento andrà perso.
- **Lettura non ripetibile:** come nel caso precedente, l'aggiornamento di variabili condivise potrebbe portare problemi: due letture consecutive della stessa variabile potrebbero restituire due valori differenti.
- **Aggiornamento fantasma:** una transazione potrebbe aggiornare variabili condivise precedentemente lette da un'altra transazione, senza che quest'ultima ne venga a conoscenza: essa quindi lavorerebbe su valori obsoleti.

Ovviamente in tutte queste situazioni la coerenza risulta minata e il risultato finale non è più valido.

2.3.2 Memorie software transazionali

Le memorie software transazionali (STM) sono un meccanismo di gestione della concorrenza che fornisce un'alternativa valida e innovativa rispetto ai classici algoritmi di locking. Concetto alla base di suddetto meccanismo è la transazione, concetto appena descritto. Nel caso delle STM, in una porzione transazionale si avranno operazioni di lettura e scrittura su zone di memoria condivisa e da ciò scaturirà il problema della concorrenza.

Contrariamente a quanto accade nel caso dei classici meccanismi di locking, nelle STM, un thread completa il lavoro sulla porzione transazionale senza tener conto, in un primo momento di quel che svolgono gli altri thread paralleli: ogni operazione di lettura e scrittura che compie, viene tuttavia registrata in un log. Al termine di una transazione, un thread che ha letto (e non un thread che ha solo scritto) valori condivisi, verificherà la consistenza del lavoro svolto: se la validazione otterrà esito positivo, si avrà un commit, in caso contrario, si procederà con un abort e quindi il conseguente rollback (ovvero la ri-esecuzione completa della transazione).

Questo approccio permette di evitare ai thread lunghe attese per gli accessi alle risorse; inoltre essi possono accedere contemporaneamente a parti diverse di una stessa struttura condivisa, situazione impensabile con un locking classico, applicato sull'intera struttura. Inoltre, sempre confrontandolo con gli approcci lock-based, le STM semplificano notevolmente la programmazione multithreading, evitando molti problemi che i lock spesso portano. La programmazione lock-based, infatti, richiede un'attenzione particolare a numerose operazioni che sono apparentemente scorrelate tra loro, ma che in realtà possono generare inconsistenze; richiede l'implementazione di politiche per evitare occorrenze di deadlock e starvation, compito mai banale; infine, si possono creare violazioni nell'ordine delle priorità dei thread, poiché anche un thread a priorità elevata potrebbe essere costretto ad attendere il rilascio di una risorsa da parte di un thread a bassa priorità. I suddetti problemi sono evitati o comunque limitati in maniera significativa dalla programmazione basata su memorie transazionali, che concettualmente risulta molto più semplice.

D'altra parte, le STM richiedono un costo in termini di spazio per mantenere i log di letture e scritture, e in termini di tempo per la fase di validazione. Inoltre è da notare che l'uso di memorie transazionali preclude lo svolgimento di operazioni su cui non sia possibile effettuare un rollback:

solo le operazioni reversibili sono ammesse. In ogni caso, i costi e gli svantaggi sono sostenibili e giustificati dai benefici precedentemente elencati.

Le problematica maggiori delle STM che si cerca di affrontare riguardano scalabilità e dinamicità: risulta, cioè, essenziale che il numero di locazioni di memoria accessibili da una transazione sia il meno vincolato possibile e che possa variare dinamicamente senza che ciò porti problemi nell'esecuzione del codice.

2.3.3 STM basate su lock

Lo step successivo, su cui la ricerca sta vertendo al momento, è l'idea di combinare memorie transazionali e meccanismi di locking: l'unico svantaggio resterebbe la gestione dei deadlock, che potrebbero essere evitati tramite timeout.

L'idea di base è di lasciare che le transazioni si muovano anche in stati potenzialmente inconsistenti, acquisiscano lock per le locazioni su cui devono scrivere e mantengano informazioni sulle scritture in una lista non condivisa con gli altri thread. Questo approccio permetterebbe di evitare l'uso di liste condivise tra i thread, necessarie in caso di memorie transazionali non bloccanti.

2.3.4 Transactional Locking II

Dopo aver fornito un quadro generale sull'attuale stato dell'arte delle memorie software transazionali, scendiamo nel dettaglio dell'algoritmo usato in questo progetto.

Il transactional locking II (TL2) è un algoritmo per memorie software transazionali che sfrutta un meccanismo di two-phase locking a tempo di commit e una nuova tecnica di validazione basata version-clock globali [7]. I miglioramenti che TL2 fornisce rispetto agli algoritmi precedenti sono riassumibili in tre punti:

- TL2 si adatta a qualsiasi tipo di gestione della memoria, compresi i sistemi che usano *malloc* e *free*: questo tipo di gestione (che poi èquella usata in linguaggi come C e C++) può creare problemi perchè una struttura potrebbe subire una free ed essere quindi eliminata dalla memoria condivisa, mentre vi sono ancora thread che accedono ad essa;

- grazie alla nuova tecnica di validazione vengono evitati periodi di esecuzione in stati non sicuri: il codice infatti opera solo in stati consistenti;
- dai test effettuati, TL2 ha performance migliori rispetto a tutti gli altri algoritmi sviluppati per memorie software transazionali, indipendentemente dalla loro tipologia.

Per ogni sistema transazionale abbiamo un **version-clock globale** (GVC), condiviso tra i vari thread. Questo contatore globale viene incrementato da tutte le transazioni che effettuano scritture e letto da tutte le transazioni che effettuano letture.

Ad locazione di memoria condivisa viene associato un **write-lock** così strutturato: esso è un semplice spinlock che usa un solo bit per indicare se è preso o no e il resto dei bit per segnare il numero di versione ad esso associato (che viene incrementato ogni volta che il lock viene rilasciato). Per associare lock a strutture dati complesse esistono vari schemi di mapping: i più usati, nonché quelli considerati in TL2 sono il *per object* (PO, in cui un lock viene assegnato ad un intero oggetto, che può essere una parte della suddetta struttura) e il *per stripe* (PS, in cui una porzione di memoria condivisa viene partizionata e ogni locazione transazionale è associata ad una partizione). La seconda tipologia di mapping è preferibile in quanto implementabile senza la necessità di modifiche alle strutture.

Ad ogni thread è associata la corrispondente struttura **Thread** che ne mantiene numerose informazioni utili durante l'esecuzione e per le statistiche finali. Tra i vari attributi, di fondamentale importanza sono le due strutture, chiamate **read-set** e **write-set**. Esse sono usate per tenere traccia, rispettivamente, delle letture e delle scritture gestite transazionalmente (quindi su variabili condivise) da ogni thread. Sono rappresentate tramite liste le cui entry hanno il doppio puntatore, sia all'elemento che precede, sia a quello che segue: sono quindi consultabili in entrambe le direzioni.
La struttura che implementa tale lista è detta **Log**. Le informazioni in essa contenute sono puntatori ad alcune entry, in particolare:

- **put:** punta alla prossima entry su cui scrivere;
- **tail:** rappresenta l'ultima entry scritta;
- **end:** è l'ultima entry disponibile del *Log*. Quando si arriva ad essa è necessario estendere la struttura.

Ogni entry del Log è rappresentata dalla struttura **AVPair**. Essa mantiene varie informazioni di interessa sull'operazione di lettura/scrittura tracciata. Di particolare interesse in questa trattazione sono:

- **Next/Prev:** puntatore all'entry successiva/precedente;
- **Addr:** l'indirizzo da cui si andrà a leggere o su cui si andrà a scrivere;
- **LockFor:** il lock associato ad Addr;
- **rdv:** il valore del GVC associato all'operazione, la cui gestione vedremo nel dettaglio a breve.

```
typedef struct _Log {
    AVPair* List;
    AVPair* put;
    AVPair* tail;
    AVPair* end;
    long ovf;
#ifndef TL2_OPTIM_HASHLOG
    BitMap BloomFilter;
#endif
} Log;
```

```
typedef struct _AVPair {
    struct _AVPair* Next;
    struct _AVPair* Prev;
    volatile intptr_t* Addr;
    intptr_t Valu;
    volatile vwLock* LockFor;
    vwLock rdv;
#ifndef TL2_EAGER
    byte Held;
#endif
    struct _Thread* Owner;
    long Ordinal;

} AVPair;
```

Figura 2.8: Strutture Log e AVPair

Vediamo ora come gli algoritmi eseguiti sia dalle transazioni che effettuano scritture sia quelle che si limitano alle letture.

Transazioni che effettuano scritture:

1. il primo passo è il caricamento e il salvataggio dell'attuale valore del GVC. La variabile locale in cui questo valore viene salvato è chiamata read-version (rv);
2. si inizia ad eseguire il codice transazionale: per ogni lettura globale effettuata viene aggiornato il read-set del thread con l'indirizzo appena letto e per ogni scrittura globale viene aggiunta una coppia <indirizzo, valore> al write-set. Quando si ha una lettura transazionale, viene effettuato preventivamente un controllo per vedere se l'indirizzo in questione compare nel write-set: in caso positivo, viene caricato

direttamente il valore da lì estratto. Bisognerà poi assicurarsi che il numero di versione del lock associato alla locazione di memoria sia minore del read-version locale. In caso contrario, la locazione è stata modificata da altri dopo l'inizio dell'esecuzione e la transazione deve abortire;

3. lo step successivo è l'acquisizione dei lock, prestando attenzione all'ordine di richiesta, per evitare deadlock: tutti i lock necessari devono essere acquisiti, altrimenti la transazione fallisce;

4. ad acquisizione avvenuta, si andrà ad incrementare il valore del GVC e il nuovo valore verrà salvato nella variabile locale write-version (wv);

5. sarà poi necessaria la validazione del read-set: essa consiste nel controllare che per ogni locazione il numero di versione del lock ad essa associato sia minore del read-version e che nessun altro thread abbia bloccato suddetta locazione. Se la validazione fallisce, la transazione deve abortire;

6. infine, per ogni elemento del write-set viene scritto il nuovo valore nell'indirizzo associato e rilasciato il relativo lock. Al termine, si effettua il commit.

Transazioni che effettuano solo letture: l'esecuzione di queste transazioni risulta essere molto efficiente usando TL2.

1. Anche in questo caso, viene innanzitutto salvato il valore del GVC nella variabile locale read-version;

2. si lancia, quindi, l'esecuzione del codice transazionale: per ogni operazione di lettura si controlla a posteriori che il lock associato alla locazione letta non sia bloccato e che il suo numero di versione sia minore del read-version. Se l'esito è positivo, si effettua il commit; in caso contrario la transazione deve abortire. Come si può vedere, questo tipo si transazioni non necessitano la costruzione di un read-set.

In conclusione, si può dire che TL2 fornisce un'implementazione per le STM facile da gestire e con buone performance (i risultati sperimentali mostrano risultati dieci volte migliori rispetto all'uso di classici meccanismi a lock globali). Le buone prospettive future che questo algoritmo propone mi hanno spinto ad usarlo come base per questo progetto.

2.4 Lavori correlati

Di seguito, presenterò brevemente alcuni lavori già svolti negli ambiti toccati da questo progetto. In particolare, prima descriverò alcuni lavori sull'instrumentazione in generale, nelle varie forme che sono state ideate, e dove l'approccio instrumentativo scelto per questo progetto sia già stato usato. Infine, si vedrà come il concetto di rollback parziale sia già stato affrontato sia in altri ambiti, sia proprio, di recente, come studio specifico.

2.4.1 Instrumentazione

Uno dei primi lavori nel campo dell'instrumentazione è Dynamo, presentato in [4]. Si trattava di un sistema per l'ottimizzazione dinamica dei software: l'obiettivo era il miglioramento delle performance di un flusso di istruzioni, garantendo la trasparenza nei confronti dell'utente. Dynamo ha ottenuto buoni risultati anche nel caso di input che derivassero dall'esecuzione di codice binario compilato staticamente. Il fatto che non fossero richiesti né azioni aggiuntive dell'utente, né hardware o software dedicati, ne garantivano la trasparenza. Il fatto che operazioni di Dynamo si svolgessero a tempo di esecuzione, però, portava alla necessità di prestare molta attenzione anche al costo di tali operazioni: non si doveva fare attenzione solo a migliorare il codice applicativo, ma anche quello di Dynamo stesso.

Sempre nell'ambito dell'instrumentazione dinamica, in [24] è stata descritta la EDDI (Efficient Debugging using Dynamic Instrumentation) una tecnica di instrumentazione volta al debugging. Grazie ad essa, si poteva accelerare il debug di software, anche quando l'hardware di supporto era poco adeguato. L'idea di base era di abbassare i costi e la frequenza dei controlli delle anomalie a tempo di esecuzione e l'utilizzo di breakpoint.

Gli svantaggi di queste tecniche rispetto a quella usata in questo lavoro sono dovute proprio alla loro caratteristica di modifica dinamica e all'uso di servizi a livello kernel: in questi ambiti, le performance del software stesso non possono essere sacrificate e bisogna prestarvi attenzione, aggiungendo complessità e riducendo la portabilità.

I lavori che più si avvicinano alla tecnica instrumentativa qui usata sono quelli descritti in [18, 19]: in essi vengono presentati software per operazioni trasparenti di log/restore nell'ambito della simulazione ottimistica basata sull'interoperabilità HighLevel-Architecture (HLA). Qui è usata

un'instrumentazione basata sulla protezione delle pagine, volta a rilevare aggiornamenti in memoria e tracciare in appositi log le letture delle pagine. Rispetto all'instrumentazione usata in questo lavoro, i costi risultano però più alti: proteggere la pagina implica sfruttare dei segmentation fault indotti e ciò richiede un passaggio a modo kernel che ha un costo molto notevole e non accettabile quando le alte prestazioni sono di primaria importanza, come accade nei software scritti per le STM.

In [16] è stato usato lo stesso approccio instrumentativo di questo progetto: l'instrumentatore è sfruttato nell'ambito della gestione della memoria. Al contrario dell'instrumentazione selettiva che occorre per il rollback parziale, lì tutto il codice applicativo viene instrumentato e le istruzioni di interesse sono quelle che operano in memoria e non quelle che operano nello stack come accade qui. Questo dimostra la versatilità dello strumento scelto e usato in questo lavoro.

2.4.2 Rollback parziale

L'idea del ripristino di un certo stato in una computazione non è certamente nuova. Nell'informatica è una questione spesso affrontata, con tutte le problematiche del caso.

Un ambito che si avvicina a quello studiato in questo progetto è quello del Parallel Discrete Event Simulation (PDES).
Essa è una tecnica per l'esecuzione di vari tipi di simulazione dove i modelli sono descritti come oggetti, chiamati Processi Logici [9]. Le interazioni tra essi sono espresse tramite la generazione di eventi. La problematica principale da affrontare è la sincronizzazione: tra gli approcci più promettenti trovati per risolvere la questione vi è il protocollo Time Warp [17]. Con protocolli di questo genere vi è la possibilità di effettuare computazioni parallele senza il pericolo di politiche bloccanti, ed è garantita la consistenza causale proprio tramite politiche di rollback e recovery, con cui vengono salvati gli stati corretti ed eventualmente ripristinati.
Rollback e recovery vengono supportati usando due approcci, ovvero la reverse computation [6] da una parte e alcune utilità di log/restore [8] dall'altra. La prima si basa sull'annullamento degli effetti delle operazioni svolte per ripristinare uno stato corretto: in questo caso, invece di un semplice restore di uno stato corretto, si cerca di invertire le operazioni che sono state eseguite. Essa porta al vantaggio di evitare (almeno potenzialmente) operazioni di log intermedie; tuttavia il tempo di ripristino è

proporzionale alle operazioni che vanno annullate, e potrebbe avere un costo più elevato del restore classico che viene eseguito potenzialmente in *O(1)*. Infine, bisogna notare che non tutte le operazioni sono invertibili: in alcuni casi non è possibile eseguire la reverse computation e bisogna affidarsi al secondo approccio, ovvero il log/restore.
Esso si occupa del salvataggio di uno snapshot dello stato di ogni Processo Logico, in modo che, in presenza di eventuali incoerenze, l'ultimo stato corretto possa essere ripristinato: come vedremo in seguito, questo approccio è analogo a quello usato nel presente meccanismo di rollback parziale.
I due meccanismi hanno numerose analogie, seppure si muovono in ambiti diversi: il protocollo di rollback parziale che ho progettato deve affrontare problematiche simili a quelle che si incontrano in PDES, ma è pensato per le memorie transazionali, in particolare quelle che fanno uso di TL2.

Tuttavia, esistono anche studi sul rollback parziale anche nell'ambito delle memorie transazionali, usciti nell'ultimo anno. In [22] è presentato un algoritmo che cerca di predire gli accessi concorrenti e salvare dei checkpoint prima delle operazioni indiziate: se la predizione è corretta, verrà effettuato il rollback parziale da quel punto in poi. Tuttavia, c'è ancora una percentuale di errore e, nel caso la predizione sia sbagliata, dovrà essere effettuato il rollback totale, al contrario di quanto accade nel presente lavoro.

Un altro lavoro simile è presentato in [2, 3]: la memoria condivisa viene vista come un insieme di oggetti che possono essere di tipi base o struct definite dall'utente. Ad ognuno di questi oggetti è associata una probabilità che possano generare conflitti: vengono creati dei nuovi checkpoint solo di quegli oggetti che hanno valori elevati per la suddetta probabilità, altrimenti vengono mantenuti i vecchi.
Anche in questo caso, quindi, c'è una probabilità di perdita del lavoro già svolto, se i valori associati agli oggetti non sono abbastanza precisi.

Capitolo 3

Scelte progettuali

Dopo aver fornito un quadro dettagliato degli strumenti che ho usato alla base di questo progetto, scendiamo nello specifico di questo lavoro e andiamo ad analizzare le principali scelte progettuali che sono state fatte.

Inizialmente fornirò la descrizione della struttura dati che è alla base dell'efficienza del protocollo di rollback parziale: si tratta della struttura che manterrà il log per il tracciamento delle scritture locali. E' fondamentale avere un accesso veloce a tali informazioni e che siano mantenute solo quelle che effettivamente potrebbero rivelarsi necessarie.

In seguito, descriverò i dettagli progettuali dei due algoritmi implementati: l'estensione dello snapshot, che si colloca tra i primi passi da eseguire nell'ambito del progetto, e il protocollo di rollback parziale, scopo finale del lavoro.

3.1 Analisi delle strutture usate

La maggior parte delle strutture dati che si andranno ad usare, sono già presenti in TL2: si tratta delle liste con cui vengono implementati i log per tracciare letture e scritture transazionali. Sono i read-set e write-set, descritti nel secondo capitolo, costituiti dalle strutture AVPair, che mantengono svariate informazioni sulle suddette informazioni. Per reperire un nodo appartenenti ad essi, è necessario scorrerle linearmente (entrambi i sensi sono permessi dai doppi puntatori): nel peggiore dei casi vanno letti per intero.

Ciò che di nuovo occorre nell'ambito di questo progetto è invece una strut-

tura per mantenere il log delle scritture locali: come si vedrà, è stata abbandonata la struttura sopra descritta, poiché non garantiva un'efficienza sufficienza per lo scopo. Si è scelto un approccio diverso, che andremo ad illustrare di seguito.

3.1.1 Struttura per il log delle scritture locali

Di fondamentale importanza per l'efficienza del protocollo di rollback parziale è la struttura dati che servirà ad implementare il log delle scritture su variabili locali.

In TL2 esistono liste che tengono traccia delle letture e delle scritture transazionali: sono il read-set e il write-set che ogni thread mantiene nella sua struttura. Vengono aggiornate ad ogni operazione avvenuta e sono necessarie per gestire il rollback classico. Essendo liste con doppio puntatore è possibile scorrerle facilmente in entrambi i sensi.

In prima istanza si è pensato di usare una struttura simile anche per il log delle scritture locali: una lista a doppio puntatore, mantenuta all'interno della struttura del thread e aggiornata tramite un'apposita funzione. Ogni nuovo nodo del read-set sarebbe stato fornito di un puntatore all'entry corrente di tale log. Richiamato il rollback parziale, sarebbe stato così possibile scorrere all'indietro tale lista fino all'entry puntata dalla lettura transazionale problematica, per ripristinare tutti i valori precedenti.

Tuttavia, si è notato come tale soluzione peccasse molto in termini di efficienza: una struttura di questo genere avrebbe costretto al salvataggio e al conseguente rispristino di tutte le scritture locali, anche quelle non necessarie. Se, infatti, sulla stessa variabile locale, una transazione va a scrivere più volte in un breve lasso di tempo (prendiamo come esempio un semplice ciclo while che incrementa ad ogni iterazione un intero x), quello che è di interesse per il protocollo di rollback parziale è solo il valore più antico, tralasciando tutti gli aggiornamenti successivi. In effetti, i valori che devono essere ripristinati sono gli ultimi scritti prima della lettura transazionale: perciò gli aggiornamenti successivi potrebbero essere totalmente ignorati. Si vedrà come ciò in realtà non sia del tutto possibile: non potendo stabilire a priori quale sarà il punto in cui la transazione diverrà incoerente, risulta anche impossibile sapere con certezza da quale momento si potrebbero ignorare le scritture locali.

Va perciò cercato un compromesso che riesca a salvaguardare in parte l'efficienza: se tracciare la totalità delle scritture locali comporterebbe un costo notevole sia in termini di spazio (il numero di entry del log potrebbe diventare elevato) sia in termini di tempo (ripristinare nella fase di rollback tutte le operazioni svolte, risulterebbe un compito lungo e inutile), ma non tracciarle completamente sarebbe impossibile, è necessario selezionare un insieme di scritture da mantenere. Tale insieme conterrà ovviamente tutte le informazioni strettamente necessarie e, in aggiunta, parte di quelle di cui non conosciamo a priori l'interesse.

Per effettuare questa scrematura dell'insieme delle scritture locali, si consideri la computazione come fosse divisa in blocchi: ogni blocco di istruzioni è iniziato da una lettura transazionale. Sarà ogni entry del read-set del thread a scandire la computazione e sarà perciò, la struttura di tale entry a mantenere un piccolo log relativo alle scritture avvenute nel suo blocco di competenza. Non si avrà più, perciò, una grande struttura globale al thread e i relativi puntatori in corrispondenza delle letture transazionali, ma tante strutture di dimensioni minori e più gestibili.

Quali miglioramenti porta questo approccio? Se nell'ambito di una struttura globale non è possibile stabilire a priori quali scritture vadano sicuramente tracciate e quali no, considerando suddette sottostrutture la selezione è invece possibile. Questo perchè il punto da cui ripartirà l'esecuzione avvenuto un rollback parziale, non si troverà mai all'interno di uno dei suddetti blocchi: sarà sempre all'inizio di esso, ovvero in corrispondenza della lettura transazionale di riferimento. Se, perciò, all'interno del blocco corrispondente si avranno più scritture sulla stessa variabile locale (come nel caso del ciclo while precedentemente citato), soltanto la prima sarà di interesse per il rollback, ovvero quella in cui verrà mantenuto il valore della variabile precedente al punto di rottura.

Tale ragionamento, ovviamente, è valido solo nell'ambito di un blocco: non è possibile stabilire cosa vada tracciato nei blocchi successivi, facendo riferimento a ciò che è stato salvato in precedenza, non avendo informazioni preventive sull'occorrenza del problema. In ogni caso, si ottiene così una scrematura parziale dei dati da salvare che alleggerisce l'esecuzione. Quando, nel rollback parziale, si andranno a ripristinare i vecchi valori delle variabili locali, il read-set sarà scandito all'indietro e si potrà saltare in corrispondenza di ogni inizio di blocco senza curarsi di ciò che è avvenuto in mezzo.

Deciso dove collocare tali informazioni, il problema successivo da affrontare è la scelta della tipologia di struttura: quello che occorre è una struttura che permetta un accesso sufficientemente rapido alle entry e che non sia eccessivamente costoso da scandire per intero. L'accesso rapido servirà in fase di scrittura sul log: volendo salvare solo aggiornamenti di variabili nuove, sarà necessario che ogni variabile abbia una e una sola entry nel log appartenente ad un blocco. Per questo motivo, verrà scelta una chiave che permetta una rapida individuazione dell'entry corrispondente: se tale entry risulterà libera, l'aggiornamento verrà tracciato; in caso contrario,verrà ignorato.

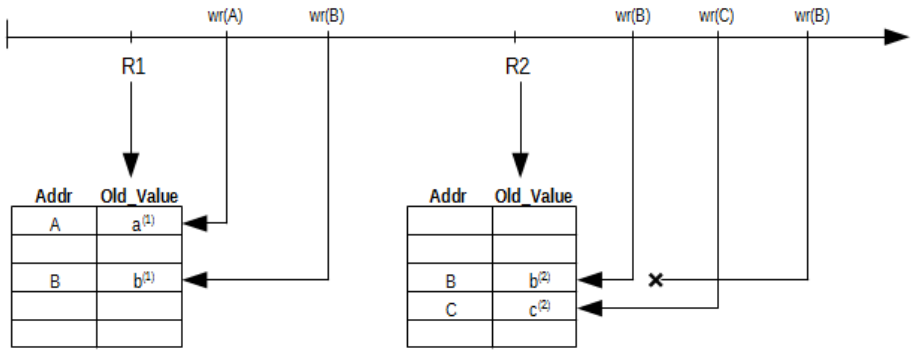

Figura 3.1: Struttura dati per il log delle scritture locali

La tipologia di struttura dati scelta è una tabella, di taglia prefissata. La tipologia di struttura scelta è una tabella, di taglia prefissata. La taglia scelta sarà una potenza di 2, così da poter calcolare facilmente l'indice partendo dall'indirizzo: basterà tenere in considerazione solo gli ultimi n bit, con n = log_2 *(numero di entry)*.

Le entry di tale tabella saranno costituite da *struct* definite per lo scopo, che manterranno le informazioni necessarie (l'indirizzo scritto, il vecchio valore, la taglia del valore, lo stato dell'entry, ovvero se è libera o meno). Si cercherà di far sì che anche ogni singola struttura che costituirà un'entry della tabella, abbia dimensione pari ad una potenza di 2, per mantenere l'allineamento.

L'indice di ogni entry sarà calcolato partendo dall'indirizzo della variabile aggiornata, tramite il mascheramento dei bit più alti: essendo tutti indirizzi interni allo stack, saranno ravvicinati tra loro e i bit più bassi costituiranno una discriminante sufficiente. Quanti bit sia più conveniente mascherare resta una problematica aperta. Il mascheramento di troppi bit, potrebbe portare ad un elevato numero di collisioni (la cui occorrenza è comunque prevista e la cui gestione verrà illustrata di seguito). Tuttavia mascherarne troppo pochi farebbe crescere le dimensioni di una tabella che non dovrà essere necessariamente molto grande.

Nel presente progetto si è scelto di settare il numero di entry della tabella a 32: di conseguenza i bit dell'indirizzo da prendere in considerazione nel calcolo dell'indice saranno 5 (poiché 32 = 2^5)
è sembrata una scelta consona alle esigenze: analizzando i benchmark di TL2 [14] si è notato che il numero di variabili locali usate non è particolarmente alto e ho ritenuto 32 entry un numero sufficiente per gestire le normali esecuzioni, senza il rischio di dover ricorrere a nuove allocazioni troppo frequentemente.

Anche scegliendo un numero di bit da mascherare che risulti efficiente, non è possibile evitare completamente il verificarsi di collisioni: non è, cioè, certo che a due indirizzi diversi corrispondano sempre due indici diversi. Tuttavia, le scritture corrispondenti dovranno essere salvate entrambe. Si è scelto di non far uso di liste di trabocco che porterebbero all'esplosione delle dimensioni della struttura e renderebbero poco efficiente il suo scorrimento in fase rollback. L'approccio scelto è quello della scansione lineare: selezionando tramite l'indice precedentemente descritto l'entry da riempire, si avranno sicuramente molti vuoti all'interno della struttura, ed essi possono essere usati nella gestione delle collisioni. Se andando a leggere l'entry ottenuta tramite un dato indice, essa risulterà occupata i casi possibili saranno due:

1. la variabile corrispondente è effettivamente stata già scritta, perciò l'aggiornamento verrà ignorato;

2. siamo in presenza di una collisione. Nel secondo caso, si passerà alla successiva entry della tabella: se risulterà libera, le informazioni verranno scritte, in caso contrario, si tornerà alle due possibilità sopra descritte. Si procederà in questo modo fino a trovare un'entry libera o fino a che non ricadremo nella prima possibilità.

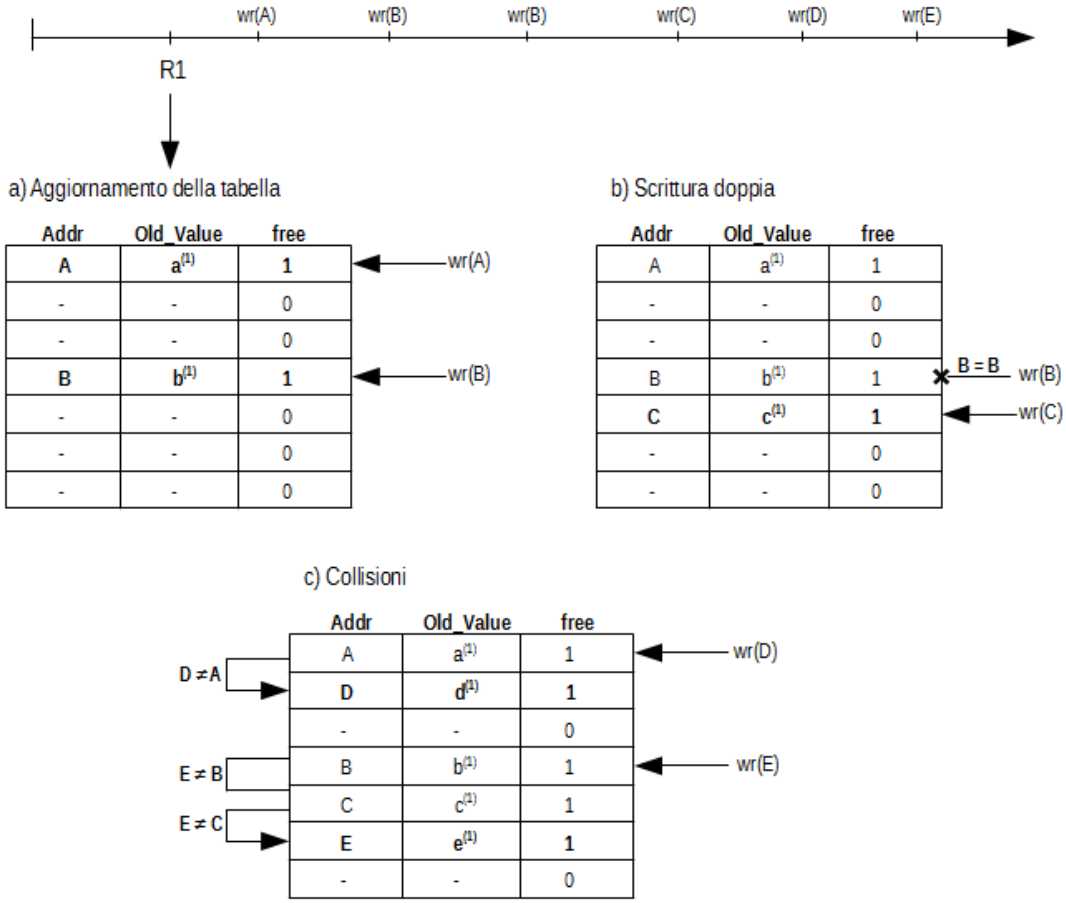

Figura 3.2: Gestione delle collisioni

Infine, poiché si è scelto di assegnare una taglia prefissata alla struttura e tale taglia non sarà eccessivamente grande, è necessario prevedere anche un meccanismo di estensione. Questo è possibile farlo semplicemente riallocando la struttura, copiandone il contenuto in una più grande, quando le entry libere terminano. L'operazione è di per sé, abbastanza costosa: tuttavia, non verrà effettuata frequentemente, per cui il suo costo risulta ammortizzato.

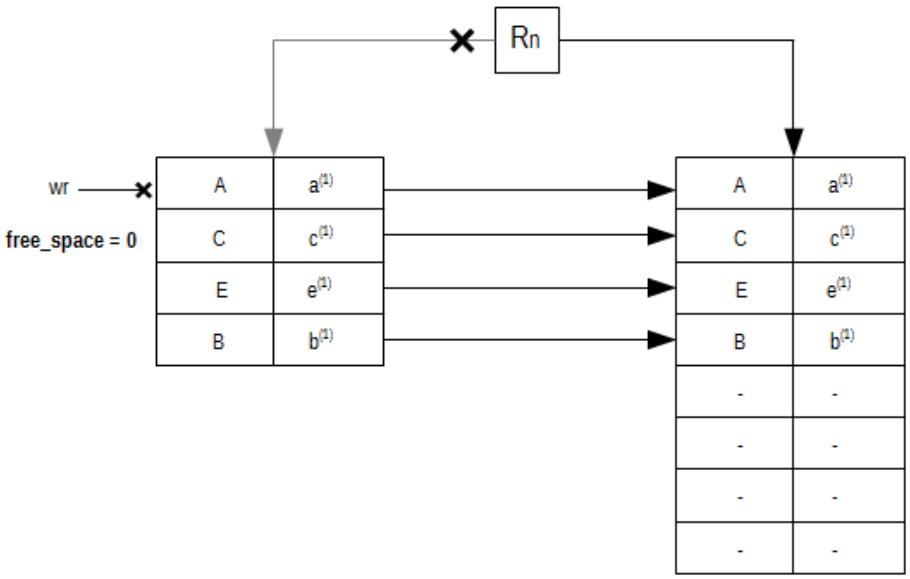

Figura 3.3: Procedura di riallocazione

3.2 Analisi degli algoritmi

Presenterò ora i due principali algoritmi ideati, progettati e infine implementati nel corso del progetto.

Il primo, l'algoritmo di estensione dello snapshot, è una procedura che verrà effettuata in corrispondenza di ogni lettura sospetta, ovvero quando il read-version della transazione risulta minore del numero di versione del lock da acquisire per la suddetta lettura. Grazie a questa procedura si potrà, nella migliore delle ipotesi, risparmiare al thread un aborto. In caso negativo invece, si otterrà la prima informazione necessaria per il rollback parziale, ovvero il punto da cui inizia la fase incoerente.

Il secondo algoritmo è il cuore nonché la finalità del progetto: si tratta della procedura di rollback parziale. Essa sarà lo step finale e ne discuterò l'efficienza.

3.2.1 Procedura di estensione dello snapshot

Il primo algoritmo che è stato necessario ideare nell'ambito di questo progetto ha lo scopo di trovare il punto esatto in cui l'incoerenza è avvenuta.

Prima di ogni lettura globale, TL2 effettua alcuni controlli per verificare che read-version e lock-version relativi al thread corrente siano ancora validi: nel caso positivo, si avrà il tracciamento di tale lettura. Come detto in precedenza, ogni thread mantiene un proprio read-set: esso è implementato tramite una lista con doppio puntatore e viene aggiornato ad ogni lettura globale valida effettuata.

Il caso di interesse in questo lavoro è ovviamente quello negativo: cosa accade se dai controlli preventivi risulta che la transazione in questione ha letto un GVC ormai antecedente alla numero di versione del lock che dovrà richiedere, ovvero, se dovrà andare a leggere il valore di una variabile modificata? Nel tradizionale algoritmo di transactional locking II, questo porterebbe ad un aborto della transazione e ad una sua nuova esecuzione: esattamente ciò che si vorrebbe evitare.

La considerazione da fare perciò, è la seguente: se la lettura attuale risulta non valida, deve esserci stato un momento precedente ad essa in cui il read-version della transazione è diventato obsoleto. Questo momento si troverà in corrispondenza di una lettura globale, perciò gli scenari possibili sono i seguenti due:

- Da una lettura precedente a quella corrente, il read-version smette di essere valido, diventando minore del numero di versione del lock corrispondente: questo porta all'invalidazione di tutte le operazioni avvenute dopo quel momento.
- Il read-version smette di essere valido solo in corrispondenza della lettura corrente (non ancora avvenuta): ciò implica che le operazioni svolte in precedenza sono ancora consistenti e non è necessario annullarle.

Il secondo caso è ovviamente quello preferibile e di più facile gestione: se le operazioni già svolte risultano ancora valide, non resta che estendere lo snapshot. Si tratterà di andare a rileggere il GVC corrente e quindi aggiornare il read-version del thread: in questo modo, auspicabilmente, anche la

lettura corrente diventerà valida e si potrà procedere nella normale esecuzione della transazione, risparmiando un costoso e inutile rollback.

Il primo caso è senz'altro il più complesso, nonché quello più di interesse nell'ambito di questo progetto: se vi sono operazioni precedenti a quella corrente che non risultano più valide, non è possibile estendere lo snapshot. Questo perchè l'estensione porterebbe alla validazione di operazioni inconsistenti e alla violazione delle proprietà di atomicità e isolamento delle transazioni. Sarebbe quindi necessario un rollback: nel tradizionale TL2 si avrebbe un aborto con conseguente rollback totale; quello che invece è lo scopo del progetto è procedere ad un rollback solo parziale, partendo dal punto in cui il read-version ha smesso di essere valido. Tale punto viene fornito proprio dalla procedura qui descritta.

L'algoritmo per l'estensione dello snapshot si riduce ad un semplice ciclo in cui si va a scorrere il read-set del thread corrente, dall'inizio fino ad un certo punto: tale punto può essere l'ultima entry scritta, se la procedura avrà successo; o una qualsiasi entry della lista, nel caso in cui verrà trovata una lettura non valida precedente a quella corrente. Ad ogni iterazione del ciclo, verrà confronterà il numero di versione del lock corrispondente alla lettura con l'attuale read-version.

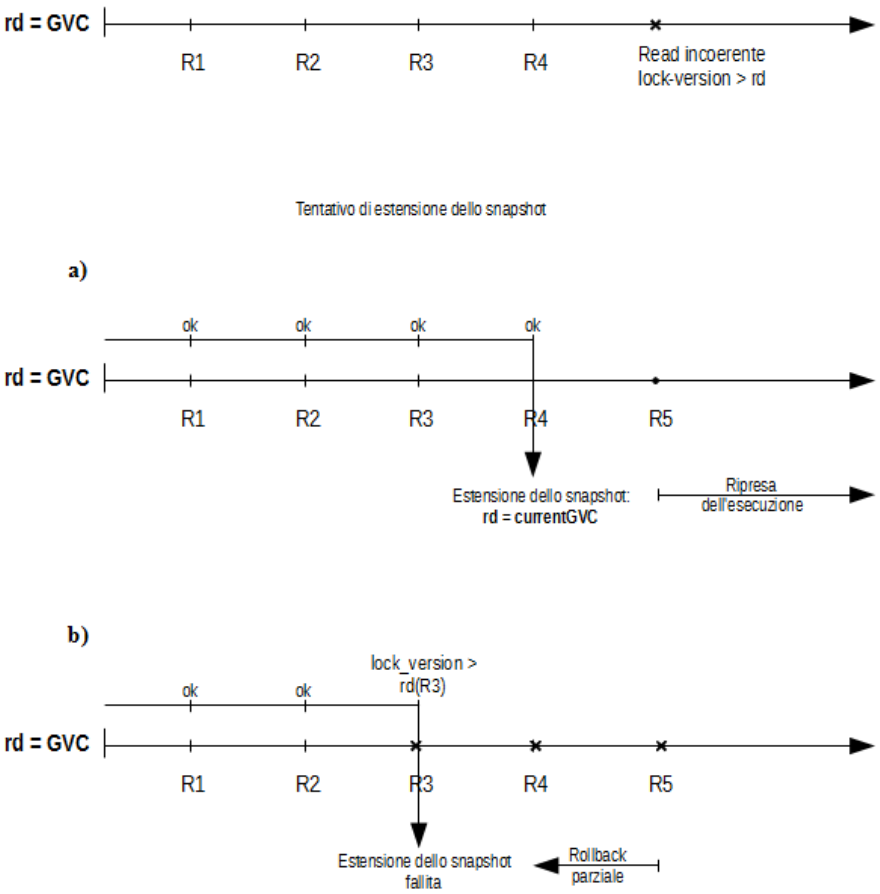

Figura 3.4: Procedura di estensione dello snapshot a) riuscita b) fallita

Tale procedura, nel caso peggiore, ha costo *O(n)*, dove n corrisponde al numero di entry presenti nel read-set. Bisogna notare, però, che il caso peggiore corrisponde spesso, alla soluzione globalmente migliore: se arriviamo a scorrere la lista fino all'ultima entry, vuol dire che, con buona probabilità, potremo effettuare l'estensione dello snapshot, risparmiando così il costo di un rollback. Potremmo, tuttavia, trovarci anche nel caso in cui sia proprio l'ultima entry a non essere valida: questo caso, seppure possibile, risulta, da alcuni test effettuati, poco probabile. Si è notato infatti che, qualora si trovino letture precedenti in corrispondenza delle quali il

read-version si invalida, esse si trovano in buona percentuale posizionate tra un terzo e la metà del lavoro già svolto.

Il costo di tale operazione, perciò, pur dovendo essere sostenuto ad ogni lettura globale che non abbia superato i controlli preventivi, non va a gravare eccessivamente sull'efficienza dell'algoritmo.

3.2.2 Protocollo di rollback parziale

Ultimo e più importante step è infine l'algoritmo che gestisce il vero rollback parziale. Si ricordi che questa procedura verrà invocata nel caso in cui l'estensione dello snapshot fallisse: nel qual caso l'algoritmo precedente restituirebbe il riferimento alla lettura transazionale che costituirebbe il punto di incoerenza.
Questa fondamentale informazione sarà usata nel protocollo descritto di seguito come punto di arrivo: andranno ripristinati tutti i valori subito precedenti ad esso.

Il protocollo di rollback parziale progettato è costituito da quattro parti fondamentali:

1. Ripristino delle variabili locali: in cui verranno annullati gli aggiornamenti a livello locale avvenuti dopo che la transazione è divenuta incoerente.

2. Ripristino delle informazioni sul write-set: in cui verrà riportato il write-set ad uno stato consistente.

3. Ripristino delle informazioni sul read-set: in cui verrà riportato il read-set all'entry precedente a quella che ha generato il problema.

4. Ripresa dell'esecuzione: la transazione ripartire dall'ultimo stato consistente.

La prima fase è la più lunga e costosa: le informazioni di interesse sono salvate all'interno dei nodi del read-set del thread, perciò bisognerà scorrere parte di questo log per reperirle. Si inizia dall'ultima entry scritta, ovvero dall'ultima lettura transazionale effettuata e procede all'indietro. Di ogni nodo del read-set è di interesse la struttura dati descritta nel paragrafo 3.1.1: essa contiene tutte le informazioni sulle scritture locali che vanno ripristinate nell'ambito del blocco preso in considerazione. Una volta estratta la tabella, essa andrà letta per intero, scorrendo le entry alla ricerca di

quelle che contengono informazioni: per ogni scrittura locale che è stata salvata si reperirà così l'indirizzo e il vecchio valore, che verrà ripristinato.
Questa procedura andrà iterata per tutti i nodi del read-set fino al raggiungimento del nodo che rappresenta il punto di rottura: i due cicli annidati rappresentano un costo ed è perciò di fondamentale importanza mantenere basse le dimensioni iniziali delle tabelle.

La seconda fase sarà molto più rapida: avendo preventivamente salvato i puntatori del write-set di interesse, sarà sufficiente ripristinare quei valori e le entry successive verranno semplicemente sovrascritte. Anche i puntatori al write-set verranno mantenuti in ogni nodo del read-set: in particolare, in corrispondenza di ogni lettura transazionale, andranno mantenute le informazioni sui puntatore alla corrente entry da riempire (da cui si potrà risalire all'ultima scritta) e a quella che rappresenta l'attuale fine della lista (che può variare nel tempo a causa di rilocazioni).

Per quanto riguarda il ripristino del read-set, vi saranno ancora meno operazioni da compiere: l'entry relativa alla lettura che ha causato l'incoerenza infatti, è esattamente il nodo che sarà il prossimo da riempire nella nuova computazione. Basterà settare il puntatore del read-set ad esso e quella e le successive entry verranno poi sovrascritte.

Ripristinate tutte le informazioni sull'ultimo stato consistente della transazione, il rollback parziale potrà dirsi concluso e non resterà che far ripartire la computazione da lì: ancora una volta sarà il nodo del read-set a contenere ciò che occorre, ovvero il buffer dove sono stati salvati i valori dei registri di processore prima della lettura transazionale che ha generato il problema. Avendo queste informazioni sarà sufficiente una chiamata a siglongjmp e l'esecuzione verrà riportata a quel punto.
Il costo di tale procedura è un *O(m*k)*, dove **m** è il numero di nodi del read-set che dovremo andare a leggere per ripristinare lo stato e **k** è il numero di entry medio di ogni log per le scritture locali. Si tratta di un costo abbastanza elevato, ma confrontato con i benefici portati dall'evitare numerosi rollback totali, sarebbe sostenibile. Inoltre, trovare una dimensione efficiente per la tabella che racchiude il log delle scritture locali (e quindi un numero di bit da mascherare nel calcolo dell'indice), potrebbe rivelarsi una chiave per potenziare le performance.

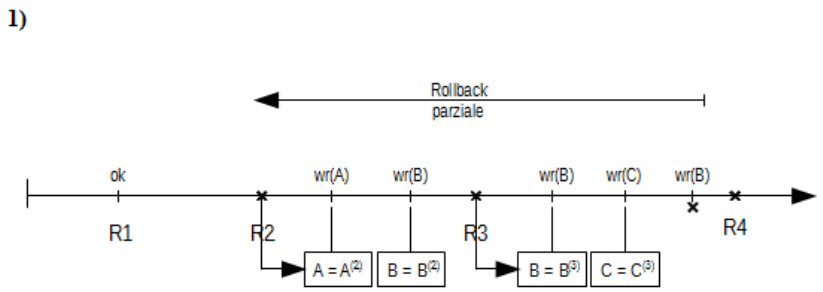

Figura 3.5: Rollback Parziale: Ripristino delle variabili locali

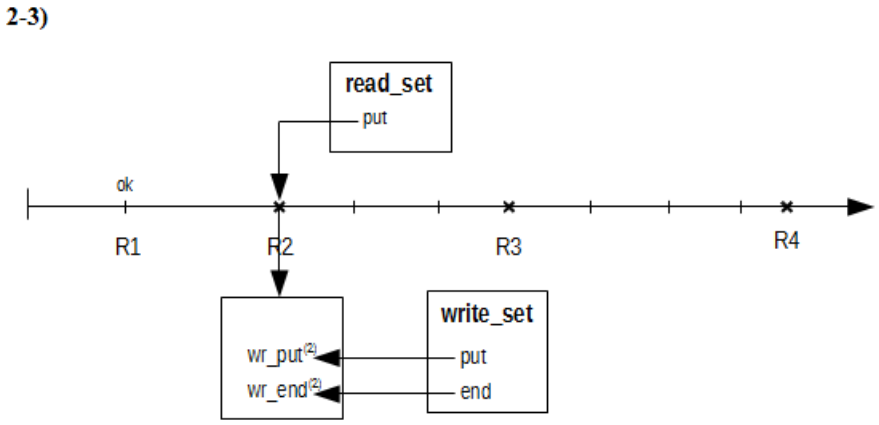

Figura 3.6: Rollback Parziale: Ripristino di read-set e write-set

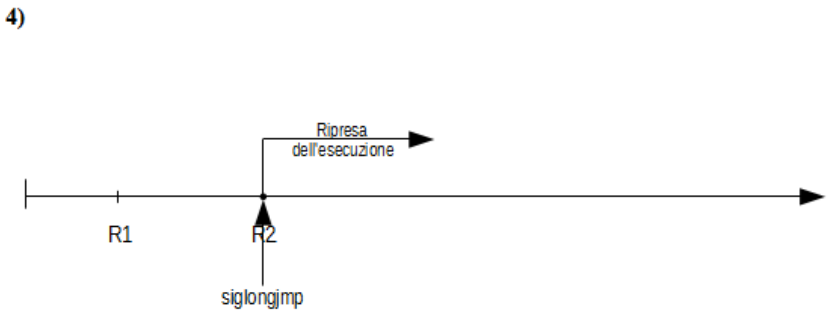

Figura 3.7: Rollback Parziale: Ripresa dell'esecuzione

Capitolo 4

Fase implementativa

Passiamo infine alla descrizione completa dell'implementazione del progetto, in tutte le sue parti.

Inizierò fornendo i dettagli sulle prime modifiche che è risultato necessario apportare a TL2 per individuare l'operazione che ha generato l'incoerenza e per il mantenimento delle informazioni necessarie alla successiva fase di rollback. Sarà la funzione che controlla le letture transazionali ad avere una parte fondamentale: sono infatti, queste le operazioni che possono generare stati di incoerenza. Ad ogni loro verificarsi, perciò, andranno effettuati nuovi controlli oltre a quelli già presenti.

Come visto in precedenza, la struttura di TL2 da sola non è sufficiente a collezionare tutte le informazioni necessarie per il rollback parziale: dovendo ripristinare un determinato stato in un punto non noto a priori di un'esecuzione, si dovrà tenere traccia di tutte le scritture locali effettuate. Per fare ciò, andrò ad usare parser e instrumentatore, descritti nel secondo capitolo della presente relazione, in tutti i loro aspetti, comprese le modifiche apportate loro in funzione dell'uso che se ne farà.

Queste le operazioni di scrittura locale devono essere, però gestite da TL2: ogni thread deve mantenerne le relative informazioni, come già fa per le letture e le scritture transazionali. Descriverò quindi come le suddette informazioni vengono collezionate e passate a TL2 sfruttando il modulo che l'instrumentatore aggancia al codice applicativo e una nuova funzione definita direttamente all'interno di TL2.

Ottenute tutte le informazioni necessarie e implementati i meccanismi e le strutture per gestirle, non resterà che passare alla fase finale: il roll-

back parziale. Del protocollo precedentemente descritto, fornirò i dettagli implementativi.

4.1 Gestione e reperimento delle informazioni necessarie

In prima istanza, è necessario individuare quali informazioni saranno necessarie in seguito per ripristinare uno stato coerente e salvarle di volta in volta. Per fare ciò occorre estendere alcune strutture dati già presenti in TL2 con nuovi attributi.

Quello che andremo maggiormente a considerare sarà il read-set di ogni thread poiché le letture transazionali sono le operazioni critiche. Sarà, perciò AVPair, entry di tale lista, la struttura designata a mantenere il maggior numero di informazioni: dovrà contenere un quadro completo dello stato in cui l'esecuzione si trova quando essa viene eseguita.

Durante l'esecuzione di una transazione, TL2 mantiene informazioni sulle operazioni svolte: accanto al suddetto read-set, troviamo anche il corrispettivo write-set, che si occupa di tracciare le scritture transazionali, salvando indirizzo e valore scritto. Anche questa lista, come abbiamo visto in precedenza, è gestita tramite le struttre Log e AVPair: nella struttura Thread abbiamo un puntatore al rispettivo Log di write-set. La struttura Log ci fornisce informazioni sulla prossima entry da riempire (*put*), sull'ultima scritta (*tail*) e sull'ultima entry disponibile della lista (*end*). Mentre nel caso di rollback totale sarebbe sufficiente inizializzare di nuovo il write-set, volendo effettuare un rollback parziale dobbiamo sapere in corrispondenza di quale entry andrà re-impostato il puntatore presente in Thread.
Ogni AVPair del read-set dovrà quindi mantenere i correnti puntatori al write-set: in particolare saranno necessari i puntatori a put e a end. Il puntatore a tail corrisponde al predecessore di put, ed è perciò reperibile. Il puntatore a end è necessario poiché le liste possono venire estese in caso di necessità, quindi l'ultima entry può cambiare nel corso dell'esecuzione. Legata all'ultima entry abbiamo un'altra informazione che la struttura Log ci fornisce e che è necessario mantenere: si tratta dell'*overflow*, ovvero di quanto viene incrementata la lista in caso di estensione.

Infine, quando si andrà ad eseguire il rollback parziale, l'ultima, ma non meno importante azione da effettuare, dopo aver reso di nuovo coerenti strutture dati e valori delle variabili, sarà il ripristino dello stato del processore (e quindi di tutti i suoi registri): sarà questa operazione che ci permetterà effettivamente di far ripartire l'esecuzione sa un dato punto.
Anche questa informazione viene salvata in corrispondenza di ogni lettura transazionale, nel rispettivo AVPair: esiste la struttura **sigjmp_buf** fatta

per mantenere i valori dei registri di processori. Essa viene riempita tramite la funzione sigsetjmp, presente nell'header setjmp.h.

L'informazione sullo stato del processore viene salvata direttamente nella funzione *TxLoad* che gestisce le letture transazionali: la chiamata a sigsetjump, viene effettuata dopo il controllo sull'indirizzo richiesto (per vedere che non sia presente nel write-set del thread) e prima dell'effettivo tentativo di lettura. In caso di problemi perciò, si avranno già a disposizione le informazioni necessarie.

Vengono mostrate di seguito le modifiche effettuate alla struttura AVPair:

```
typedef struct _AVPair {

        ...
        sigjmp_buf*       Status_CPU;
        struct _AVPair* wr_put;
        struct _AVPair* wr_end;
        long wr_ovf;
  ...

} AVPair;
```

Queste modifiche non sono ancora sufficienti, come si vedrà a breve: sarà necessario anche un puntatore alla nuova struttura che rappresenterà il log delle scritture locali, le cui informazioni andremo a collezionare tramite l'uso dell'instrumentatore. Approfondirò la questione più avanti.

4.2 Estensione dello snapshot

Fondamentale step per la gestione di un rollback parziale è l'individuazione dell'operazione che ha portato ad uno stato incoerente: sarà infatti quello il punto da cui si cercherà di far ripartire l'esecuzione, assumendo che il lavoro svolto prima di quel momento sia coerente.
Poichè, come detto in precedenza, sono le letture transazionali le operazioni incriminate, il controllo che descriverò di seguito verrà effettuato ad ogni loro occorrenza e anche in questo caso, sarà con il read-set di ogni thread che andremo principalmente ad interagire.
L'idea di base è di scorrere il suddetto read-set per verificare che le letture effettuate in precedenza siano ancora tutte coerenti: in caso positivo andremo ad aggiornare il read-version del thread ed estenderemo così lo snapshot.

Durante ogni operazione di lettura transazionale, dopo aver controllato

che l'indirizzo richiesto non sia presente nel write-set del thread (in qual caso potremmo andare a leggere direttamente il valore locale della variabile modificata dal thread stesso), e aver salvato le informazioni descritte nel paragrafo precedente, andiamo a verificare che il numero di versione del lock relativo alla locazione di memoria di interesse, sia minore del read-version corrente del thread. In caso contrario, la transazione dovrà abortire poiché altrimenti si andrebbe a lavorare con un valore modificato dopo l'inizio della transazione, compromettendone l'atomicità.
Si procede, invece, con un tentativo di estensione dello snapshot per verificare se questa sia l'unica operazione che genererebbe problemi o se lo stato di incoerenza è iniziato precedentemente.

Questo controllo viene effettuato dalla funzione **snapshot_extension**, richiamata all'interno di TxLoad, che in TL2 gestisce le letture a variabili condivise. Suddetta funzione, va innanzitutto a leggere il read-version corrente e ad estrarre il read-set del thread. Partendo dal primo AVPair del read-set inizia a controllare la coerenza di ogni lettura effettuata: va a leggere il numero di versione del lock associato all'indirizzo letto (il *LockFor*) e lo confronta con il read-version. Quest'ultimo deve risultare maggiore del LockFor dell'AVPair analizzato: se ogni entry del read-set soddisfa questa condizione, possiamo aggiornare il read-version del thread andando a leggere il GVC corrente. L'estensione dello snapshot avrà avuto successo e l'abort potrebbe non rivelarsi più necessario.

Se invece, scorrendo il read-set, si trovasse una lettura non più coerente, si dovrà andare a restituire l'AVPair corrispondente: esso rappresenterebbe il punto in cui ripristinare lo stato dell'esecuzione per farla poi da lì ripartire. E' perciò l'informazione che si stava cercando, ovvero il riferimento all'operazione che ha dato inizio allo stato di incoerenza.

Di seguito, il codice della funzione appena illustrata:

```
AVPair* snapshot_extension(Thread* Self) {

        vwLock currentGVC = GVRead(Self);

        MEMBARSTLD();

        vwLock rv= Self->rv;
        Log* const rd = &Self->rdSet;
        AVPair* const end = rd->put;
        AVPair* e;

        for (e = rd->List; e != end; e = e->Next) {

                vwLock v = LDLOCK(e->LockFor);
```

```
                while (v & LOCKBIT) {
                        v = LDLOCK(e->LockFor);
                }

                if (v > rv)
                        return e;
        }

        return NULL;
}
```

Dopo la chiamata a snapshot_extension, in caso di successo l'esecuzione procederà normalmente: si cercherà di acquisire il lock necessario e, in caso di successo, la lettura verrà salvata.

4.3 Instrumentazione delle scritture locali

Vediamo ora come reperire le ultime informazioni necessarie per un rollback parziale: quelle riguardanti le operazioni di scrittura su variabili locali.

Quando l'esecuzione verrà riportata all'ultimo stato coerente, le letture e le scritture transazionali non creeranno problemi e verranno gestite direttamente da TL2: sarà necessario solo mantenere i puntatori alle entry corrette delle strutture corrispondenti, come abbiamo visto in precedenza. Le scritture locali, invece, non vengono tracciate automaticamente da TL2; tuttavia, esse possono influire sul risultato finale: se le variabili modificate non venissero ripristinate ai loro valori precedenti al punto da cui l'esecuzione viene fatta ripartire, si giungerebbe al termine della transazione con un risultato non corretto. E' necessario pertanto, salvare per ogni scrittura l'indirizzo di memoria coinvolto e il nuovo valore.

Per fare ciò, userò l'instrumentatore e il parser descritti nel capitolo 2. Avendo già fornito ampia descrizione del funzionamento di questi due strumenti e delle modifiche ad essi apportate per adattarli allo scopo di questo progetto.
Mi concentrerò ora sulla logica algoritmica, ripercorrendo i passi necessari per arrivare al tracciamento delle scritture locali.

Il codice applicativo, nel formato di file ELF, viene passato all'instrumentatore che si occupa autonomamente di aggiungere la logica necessaria:

tutte le operazioni descritte di seguito sono effettuate prima della fase di *linking*.

4.3.1 Individuazione e instrumentazione delle sezioni transazionali

Primo passo fondamentale è l'individuazione delle porzioni transazionali, poiché non è di interesse instrumentare tutto il codice. Esse sono identificate da due label, nelle seguenti forme:

- *begin_nomefile_numeroriga* all'inizio di ogni porzione;
- *end_nomefile_numeroriga* al termine.

Ogni porzione, pertanto, è identificata univocamente.

Viene richiamata innanzitutto la funzione count_symbol sulle due stringhe begin e end: dopo aver controllato che le due label hanno lo stesso numero di occorrenze (condizione necessaria poiché una sezione transazionale deve iniziare e terminare, non può restare aperta), si può salvare l'intero ottenuto in una variabile contatore. Questo sarà il numero di porzioni da instrumentare.

Lanciamo ora il doppio ciclo che si occupa di effettuare l'instrumentazione: il ciclo più esterno scorre le sezioni del file ELF per trovare quella che contiene codice eseguibile; il più interno, scorre la sezione individuata alla ricerca delle porzioni transazionali, finchè il contatore precedentemente settato non si esaurisce.
Ogni porzione viene trovata tramite la funzione find_symbol, passandole anche in questo caso le stringhe begin e end, più il contatore che indicherà quale occorrenza andare a ricercare nella tabella delle stringhe. Con questo step si ottengono gli indirizzi di inizio e fine della porzione da instrumentare, sotto forma di offset dall'inizio della sezione.

Tali indirizzi verranno passati alle prime due funzioni che costituiscono la logica dell'instrumentatore, già descritte nel capitolo 2. Riepiloghiamole brevemente:

- ***compute_size_increment:*** calcola l'incremento di taglia della sezione dell'ELF, necessario per l'inserimento delle nuove istruzioni. Tenendo conto del valore da essa ritornato, si andrà allocare la nuova sezione.

- ***decode_code_section:*** crea la nuova sezione, scrivendo in essa il codice, istruzione per istruzione (usando il parser) inserendo, dove necessario, le call che poi andranno a richiamare il modulo di logica aggiuntiva. In questa funzione viene anche tenuta traccia di tutte le informazioni di shift necessarie nelle fasi finali dell'instrumentazione.

Come detto, tali funzioni lavorano usando il parser e andando a leggere ogni istruzione assembler del codice, dall'inizio alla fine: è necessario, perciò, discriminare quali sono le istruzioni di interesse nel nostro caso. Le altre verranno semplicemente copiate nella nuova sezione, senza modifica alcuna. La prima discriminante è data dagli indirizzi di inizio e fine porzione che vengono passati alle due funzioni: di ogni istruzione si conoscono la posizione nel codice, pertanto è possibile andare a considerare solo quelle comprese tra questi due estremi.
Non è ancora sufficiente. Ciò che cerchiamo sono le istruzioni che effettuano scritture su variabili locali, ovvero scritture nello stack corrente. Per individuarle, ho usato i flags presenti nella struttura insn_info legata ad ogni istruzione: i flags di interesse sono quelli che indicano un'operazione di scrittura (**I_MEMWR**) e che segnalano azioni nello stack (il flag **I_STACK** definito in precedenza nell'ambito del parser, come descritto nel capitolo 2).

Per ogni istruzioni estratta dal parser, viene effettuato pertanto il seguente controllo:

```
if      (IS_MEMFLAG(insn) && IS_MEMWR(insn)
        && curr_pos >= beg && curr_pos <= end)
```

Se soddisfatto, l'istruzione corrispondente andrà instrumentata.

Al termine di questa fase, si avrà la nuova sezione, che vedrà instrumentata la porzione transazionale presa in esame. Tale sezione sovrascriverà la precedente nel file ELF e si passerà ad una nuova iterazione del ciclo interno, finchè il contatore non si esaurirà.
Avremo allora esaurito le porzioni transazionali comprese tra le label begin e end, ma come vedremo di seguito, il lavoro sul codice applicativo non è ancora terminato.

4.3.2 Individuazione e instrumentazione delle funzioni

E' possibile (e in genere, probabile) che nelle porzioni transazionali prese in esame vi siano delle chiamate a funzione. In questo caso, la sola instrumentazione della porzione compresa tra begin ed end non sarà

più sufficiente, perché anche le operazioni svolte nelle funzioni richiamate avranno importanza nel corso dell'esecuzione e nel calcolo del risultato finale. Tali chiamate andranno perciò individuate e il loro codice andrà a sua volta instrumentato.

L'individuazione di eventuali call all'interno delle porzioni transazionali avviene all'interno del ciclo instrumentativo precedentemente descritto. Prima di chiamare computeI_sizeI_increment, viene usata la funzione findI_function descritta nel capitolo 2. Essa richiama un'altra funzione che lavora con il parser, ovvero search_call, che cerca le istruzioni di chiamata a funzione, all'interno dei limiti della porzioni presa in esame. Per individuare questo tipo di istruzioni ho fatto uso ancora una volta dei flags: in questo caso, il flag di interesse è **I_CALLRET**. Per evitare di considerare anche le istruzioni di ritorno, leggiamo anche il mnemonico dell'istruzione. La condizione completa da verificare, pertanto, è la seguente:

```
if      (IS_CALLRET(insn) && strstr(insn.mnemonic, "call")!= NULL
        && curr_pos >= beg && curr_pos <= end)
```

Quello che search_call restituirà sarà il nome della funzione, estratto tramite get_symbol_name_by_reloc_position, una funzione messa a disposizione nel pacchetto dell'instrumentatore, che interagisce direttamente con le tabelle dell'ELF.

Tutti i nomi delle funzioni trovate in questo step vengono salvate in una lista di stringhe, comune a tutte le porzioni transazionali: se una funzione è già presente non verrà riaggiunta (evitiamo così la possibilità di instrumentazioni doppie).
Da notare che find_function è ricorsiva: ogni volta che trova una chiamata a funzione, richiama se stessa settando come indirizzo di partenza quello dell'inizio della funzione. In questo modo vengono individuate anche chiamate annidate.

Al termine del primo ciclo instrumentativo, quindi, si avrà anche la lista dei nomi delle funzioni da instrumentare: non resta che effettuare un secondo ciclo, questa volta sulla suddetta lista, e ripetere gli stessi passi che si sono seguiti per l'instrumentazione delle porzioni comprese tra le label.

L'unica differenza con la situazione precedente è che, in questo caso, si ha soltanto l'offset che rappresenta l'inizio della funzione (ricordiamo che tale riferimento viene trovato sempre usando find_symbol, poiché anche il nome di una funzione è un simbolo, in un file ELF): non si conosce a priori il punto in cui la definizione della funzione termina. Per ovviare a questo problema, ho passato sia a compute_size_increment che a decode_code_section, un valore impossibile per la end (ad esempio -1). Quello

che è necessario in questo caso, è un ulteriore controllo sull'istruzione attualmente analizzata tramite parser, per riconoscere quella che sarà l'istruzione di chiusura della funzione: essa corrisponderà alla prima istruzione di tipo ret (il flag corrispondente è ancora I_CALLRET) che si trova dopo aver passato la posizione corrispondente a begin. Dopo aver effettuato questo controllo:

```
if      (end == -1 && IS_CALLRET(insn) && curr_pos >= beg
        && strstr(insn.mnemonic, "ret")!= NULL)
```

è possibile procedere con l'esecuzione, sapendo che le istruzioni successive non sono da instrumentare.

Anche in questo caso, al termine di decode_code_section, la nuova sezione instrumentata è pronta e va a sostituire la vecchia: il ciclo, come detto, termina quando si arriva alla fine della lista di funzioni.

Conclusosi il ciclo più esterno che ricerca le sezioni con codice eseguibile, si ha il nuovo codice in cui sono presenti le chiamate tramite cui si andrà ad agganciare il modulo assembly. Terminate le rilocazioni descritte nel secondo capitolo, si è pronti a tracciare anche le scritture locali.

4.4 Costruzione del log delle scritture locali

Ora che il codice applicativo è stato instrumentato e le scritture locali sono state individuate, è necessario collegare l'instrumentatore e TL2, in modo che i valori trovati vengano passato all'algoritmo per memorie transazionali, che si occuperà del salvataggio di essi in un'apposita struttura dati.

Questo collegamento è costituito dal monitor, modulo assembler che il codice instrumentato richiama prima delle istruzioni di scrittura nello stack: sarà esso a estrapolare indirizzo e taglia della scrittura.
Per passare queste informazioni al livello superiore, il monitor chiamerà una funzione che andremo a definire direttamente in TL2: accettandole come parametri, la funzione estrarrà anche il valore scritto e salverà il tutto nella struttura dati creata per lo scopo, struttura che abbiano analizzato nel capitolo 3.

Nei prossimi paragrafi, mostrerò prima l'implementazione della struttura dati che rappresenta il log delle scritture locali; poi mi concentrerò sul-

la funzione definita in TL2 per la gestione di suddetta struttura; infine, fornirò qualche dettaglio sul funzionamento del monitor.

4.4.1 Creazione delle strutture per il log

Come è stato illustrato nel capitolo 3, era di fondamentale importanza trovare una struttura dati che si adattasse allo scopo: non era solo sufficiente salvare i dati relativi alle scrittura locali, ma era necessaria anche una parziale scrematura dei suddetti dati. Questo per evitare un numero troppo elevato di aggiornamenti non necessari: una variabile che subisce due aggiornamenti ravvicinati, nella fase di rollback parziale, dovrà essere ripristinata all'ultimo valore prima dell'incoerenza, ignorando le scritture successive.

Nella fase progettuale è stato spiegato come, per ottenere questa scrematura, tale struttura dati debba essere associata ad ogni singola lettura transazionale: andremo quindi ad aggiungerla alla struttura AVPair, che costituisce il singolo nodo del read-set del thread.
L'implementazione scelta è un array, di taglia prefissata: tuttavia è prevista la possibilità di ri-allocazione, nel caso la dimensione si rivelasse insufficiente. A tal proposito, viene mantenuto anche il numero di entry attualmente libere, che permetterà di controllare la disponibilità di spazio nella struttura prima di effettuarvi una scrittura.

```
typedef struct _AVPair {

          ...
        Local_Write* Local_Log;
        int free_space;
        int dim_log;
        ...

} AVPair;
```

- **Local_Log:** è l'array, che verrà allocato alla creazione dell'AVPair, che conterrà le informazioni sulle scritture locali;
- **free_space:** rappresenta il numero di entry attualmente libere nell'array;
- **dim_log:** rappresenta l'attuale dimensione dell'array.

Ogni entry dell'array sarà costituita dalla seguente struttura:

```
typedef struct _Local_Write{

        volatile intptr_t* Addr;
        long long Value;
        size_t Size;
        short int free;
        char dummy[6];

} Local_Write;
```

I quattro campi in essa contenuti, hanno le seguenti funzioni:

- **free**: indica se lo slot corrispondente è libero, ovvero se in Addr non è mantenuto alcun indirizzo già scritto. Questo ci permette di evitare sovrascritture degli slot, in caso di collisioni nel calcolo dell'indice;
- **Addr**: è l'indirizzo di memoria su cui è avvenuta la scrittura locale;
- **Value**: è il valore che la variabile aveva prima della scrittura locale;
- **Size**: corrisponde alla taglia del valore salvato in Value;
- **dummy:** costituiscono byte di padding per allineare le strutture e raggiungere una dimensione potenza di 2.

La loro taglia è stata scelta appunto per avere una dimensione totale che sia potenza di 2 (in questo caso sarà pari a 32 byte): in questo modo sarà semplice mantenere anche la dimensione dell'intero array pari ad una potenza di 2.

Legata alla struttura dati che rappresenta il log, è stata implementata anche una funzione che gestisce la riallocazione della tabella in caso di necessità: si tratta di un'operazione che ha un certo costo, comportando l'allocazione di un nuovo array, di dimensioni doppie rispetto al precedente (per mantenere le dimensioni pari ad una potenza di 2). Tale operazione deve essere seguita dalla copia delle entry della vecchia struttura nella nuova. Dopo di ciò, verranno aggiornati i contatori presenti in AVPair: free_space verrà riportato al massimo (ora abbiamo di nuovo a disposizione un numero di entry pari alle dimensioni della vecchia struttura) e dim_log verrà raddoppiato. Quest'ultima informazione servirà per eventuali future riallocazioni.

```
void realloc_log(AVPair* t){

        int dim = t->dim_log;
        int incr = 2*dim;
        Local_Write* realloc = (Local_Write*)malloc(sizeof(Local_Write)*incr);
```

```
        int i = 0;
        Local_Write* aux = t->Local_Log;

        while (i < t->dim_log){
                memcpy(&(realloc[i]), &(aux[i]), sizeof(Local_Write));
                i++;
        }

        free(t->Local_Log);
        t->Local_Log = realloc;
        t->free_space = t->dim_log;
        t->dim_log = incr;

}
```

4.4.2 Funzione per la costruzione del log

Implementata la struttura dati per tenere traccia delle scritture locali, vediamo ora come andare a riempirla.

Quando l'instrumentatore intercetterà un'istruzione di interesse e ne estrarrà i valori cercati, questi valori andranno passati a TL2 perchè li salvi nella struttura descritta precedentemente. La fase di salvataggio è effettuata dalla funzione SaveLocalWrite a cui vengono passati come parametri l'indirizzo che si è andati a modificare e la taglia del valore scritto.
Ho scelto di rendere il più semplice possibile l'interfacciamento tra il modulo assembler e TL2: per questo alla funzione vengono passati solo i suddetti due parametri. Le altre informazioni che andrò ad usare, verranno estratte in altro modo, per non appesantire il codice del modulo.

Prima fondamentale informazione è l'individuazione del thread in cui la scrittura è avvenuta. Per fare ciò, viene usata una funzione già presente in TL2: si tratta di **pthread_getspecific**, che, ricevendo in input la *global_key_self* (che mantiene l'indicatore al thread corrente), restituisce la struttura dati corrispondente.

Dalla struttura del thread corrente va estratto il read-set: come detto, ogni entry di questo insieme manterrà il log delle scritture locali relative al suo blocco di istruzioni. Del read-set è di interesse l'ultima entry riempita: essa corrisponde al puntatore tail; nel caso in cui esso fosse nullo (ovvero, non è ancora avvenuta alcuna lettura transazionale), verrà considerato il puntatore List che indica la prima entry del read-set. Dell'AVPair che ora si ha a disposizione, viene letto il *free_space*: se esso è pari a 0, si procederà all'estensione e conseguente ri-allocazione dell'array. Altrimenti, verrà

semplicemente estratto il Local_Log dell'AVPair.

Per individuare l'entry, dell'indirizzo passato in input vengono mascherati gli n bit più alti: si ottiene così un indice quasi univoco e si può andare a leggere l'entry del log. In questa implementazione, poichè la struttura dati ha 32 entry, verranno mascherati tutti i bit dell'indirizzo, tranne gli ultimi 5.
A questo punto, viene controllato il campo free della struttura e sono possibili i seguenti tre scenari:

- il campo *free* è uguale a 0: lo slot è libero, l'indirizzo corrispondente non è ancora stato scritto nell'ambito di questo blocco. La entry viene aggiornata con le informazioni sull'indirizzo, la taglia e il valore che viene letto direttamente dalla memoria con una memcpy;
- il campo *free* è uguale ad 1 e l'indirizzo in input è uguale a quello già salvato nell'entry: in questo caso ci troviamo ad un aggiornamento successivo al primo della stessa variabile. L'aggiornamento viene scartato;
- il campo *free* è uguale ad 1, ma l'indirizzo in input non corrisponde a quello salvato: siamo in presenza di una collisione. Si passa all'entry successiva, viene rieffettuato il controllo su free: il procedimento prosegue fino a che uno dei primi due punti non sia verificato.

Al termine della ricerca dello slot libero, se l'indirizzo risulterà nuova, vi sarà la scrittura dei campi della struttura. Il campo free verrà settato ad 1 e il contatore delle entry libere verrà decrementato.

Forniamo di seguito il codice completo della funzione:

```
void SaveLocalWrite (volatile intptr_t* addr, unsigned int size){

        Thread* Self = (Thread*)pthread_getspecific(global_key_self);
        Log* rd = &Self->rdSet;
        AVPair* tail;

        if (rd->tail != NULL)
                tail = rd->tail;
        else
                tail = rd->List;

        if (tail->free_space == 0)
                realloc_log(tail);

        unsigned int index = (long)addr & 0x1FF;

        Local_Write* aux = tail->Local_Log;
```

```
        if (aux[index].free != 0){

                while (aux[index].free != 0){

                        if (addr == aux[index].Addr)
                                return;

                        index++;
                if (index ==  tail->dim_log)
                        index = 0;
                }
        }

        aux[index].free = 1;
        aux[index].Addr = addr;
        aux[index].Size = size;
        memcpy(&(aux[index].Value), addr, size);
        tail->free_space--;
}
```

4.4.3 Logica aggiuntiva

Abbiamo visto nel capitolo 2 come il modulo assembly che è stato aggiunto codice instrumentato, tracci le scritture nello stack e di esse, riesca a recuperare l'indirizzo di base e la taglia. Si ricordi che al termine del lavoro del monitor, si ha l'indirizzo nel registro edi e la taglia nel registro esi.

Passare tali valori (che sono esattamente quelli cercati) a TL2, a questo punto risulta semplice. Basterà caricare i registri sopra indicati, passarli come parametri (i valori sono già posizionati nei registri usati in genere per il passaggio di parametri) e effettuare una call alla funzione descritta nel paragrafo precedente.

Di seguito riporto il breve modulo aggiunto al codice assembly per la chiamata a SaveLocalWrite:

```
push    %rdi
push    %rsi
callq   SaveLocalWrite
pop     %rsi
pop     %rd
```

4.4.4 Cenni sulla fase linking

Terminate le modifiche su entrambi i fronti, è necessario collegare i due strumenti, instrumentatore e TL2, perché vengano compilati nel modo corretto.

Quello che va passato all'instrumentatore perché vi effettui il suo lavoro in maniera trasparente all'utente, non è l'eseguibile, bensì i file Object rilocabili. Essi sono i file *.o* che si ottengono dalla compilazione dell'applicazione di prova, prima che venga eseguito su di esso il linking con le librerie di TL2.

L'instrumentatore andrà ad instrumentare tali file in fase di compilazione e genererà un nuovo insieme di *.o*, questa volta instrumentati. Su di essi verrà effettuato il linking con il modulo assembly e, infine, con le librerie TL2.

Quello che si avrà al termine di questi passaggi, sarà finalmente il file eseguibile, instrumentato e pronto per essere testato.

4.5 Protocollo di rollback parziale

Tutta l'implementazione appena descritta è funzionale all'algoritmo che segue: si tratta della vera gestione del rollback parziale.

Esso verrà richiamato subito dopo il fallimento di un tentativo di estensione dello snapshot: quest'ultimo avrà restituito un AVPair non nullo, corrispondente alla lettura transazionale dalla quale la transazione ha smesso di essere consistente. L'AVPair suddetto e la struttura del thread corrente vengono passate alla funzione che si occupa di effettuare i passi (descritti nel capitolo 3) del rollback.

Dalla struttura del thread viene estratto il read-set: ciò che occorre di esso è la coda (tail) ovvero il puntatore all'ultima entry scritta; questo perché il ripristino che verrà effettuato, scandirà il read-set all'indietro.

Partendo quindi dalla tail, si leggono tutti gli AVPair fino ad arrivare a quello passato in input, ovvero al nodo problematico: per ogni AVPair bisogna ripristinare i valori delle variabili locali scritte tra quella lettura e la successiva. Si va quindi a leggere il *Local_Log*, ovvero l'array che rappresenta la tabella in cui sono salvate le informazioni reperite preventivamente con l'instrumentatore.

Questa fase è costituita da due cicli *while* annidati: il più esterno scorrerà il read-set, il più interno leggerà ogni struttura dell'array. Se una struttura

avrà il campo free pari ad 1, essa conterrà informazioni utili: nel qual caso, verranno letti indirizzo, valore e taglia della scrittura locale e la variabile verrà ripristinata al valore precedente. Dopo questa operazione il campo free verrà settato a 0, così che quando l'esecuzione riprenderà, quella entry potrà eventualmente essere sovrascritta.

Lo step successivo è abbastanza immediato: per il ripristino del write-set, sarà sufficiente leggere i valori dei puntatori salvati nell'AVPair passato in input e ripristinare put, tail, end e ovf di conseguenza.

Per quanto riguarda il read-set, terminate le precedenti operazioni, si potrà settare il suo puntatore put proprio all'AVPair della lettura da cui è iniziato il problema: essa e tutte le successive verranno poi sovrascritte.

Con il ripristino del read-set, si chiude la fase di rollback: i valori di tutte le variabili e le liste di interesse sono stati riportati all'ultimo stato safe: ora si può far ripartire l'esecuzione da quel punto. Per fare ciò viene effettuata una chiamata a *siglonfjmp*, passandogli in input il sigjmp_buf salvato nell'AVPair: esso contiene i valori dei registri della CPU, prima che la lettura venisse effettuata.

```
void partial_rollback(Thread* Self, AVPair* t){

        Log* rd = &Self->rdSet;
        AVPair* aux_av = rd->tail;
        Local_Write* aux_lw = aux_av->Local_Log;
        int i = 0;

        while (aux_av != t->Prev){

         while (i < aux_av->dim_log){

          if (aux_lw[i].free == 1){
                memcpy(aux_lw[i].Addr,&(aux_lw[i].Value),aux_lw[i].Size);


                 aux_lw[i].free = 0;
                 i++;
                }

                aux_av = aux_av->Prev;
                aux_lw = aux_av->Local_Log;
                i = 0;
        }

        Self->wrSet.put = t->wr_put;
        Self->wrSet.tail = t->wr_put->Prev;
        Self->wrSet.end = t->wr_end;
        Self->wrSet.ovf = t->wr_ovf;

        Self->rdSet.put = t;
```

```
        Self->rdSet.tail = t->Prev;
        SIGLONGJMP(*(t->Status_CPU), 1);
}
```

Capitolo 5

Conclusioni

In questo lavoro ho presentato il progetto e una possibile implementazione di un meccanismo di rollback parziale per memorie software transazionali. Le STM sono un meccanismo di gestione della concorrenza che fornisce un'alternativa valida e innovativa rispetto ai classici algoritmi di locking. Rappresentano attualmente una buona strada su cui lavorare per sviluppi futuri.

In prima istanza ho descritto le tecnologie hardware e software usate per tale progettazione. L'algoritmo per memorie software transazionali scelto è stato il Transactional Locking II, meccanismo innovativo basato su un sistema di locking a tempo di commit e una tecnica di validazione che usa timestamp globali. Ho deciso di usare TL2 poiché esso fornisce un'implementazione per le STM di semplice gestione, con ottime performance e buone prospettive per futuri sviluppi.

La seconda tecnologia principalmente sfruttata in questo progetto è stata un instrumentatore basato su un meccanismo di parsing di codice macchina: l'instrumentazione, ovvero l'aggiunta di logica al codice applicativo, è una tecnica che offre notevoli possibilità future. La sua forza è la trasparenza: l'utente dell'applicazione non avrà sentore del lavoro che l'instrumentatore svolge e potrà beneficiare dei moduli aggiunti senza dover scrivere codice a sua volta. Nell'ambito di questo progetto, il lavoro di instrumentazione è servito per il tracciamento delle scritture su variabili locali, operazioni non gestite da TL2,ma le cui informazioni risultavano indispensabili per il ripristino di uno stato coerente. Ho presentato le modifiche apportate a parser e instrumentatore, per adattare il loro lavoro allo scopo del progetto.

Successivamente, mi sono concentrata sulle scelte progettuali fatte: in particolare, ho fornito la descrizione della struttura dati ideata per gestire il log delle scritture locali, discutendone l'efficienza e i possibili sviluppi futuri per migliorarla; di seguito, sono stati presentati i due principali algoritmi progettati, ovvero quello per l'estensione dello snapshot e il protocollo di rollback parziale vero e proprio.

Infine, ho presentato i dettagli implementativi del progetto: ho descritto i passi necessari per raggiungere l'obiettivo, le modifiche apportate agli algoritmi di TL2, le nuove funzioni e strutture implementate. Dei vari step, ho fornito le parti salienti del codice, scritto principalmente in ANSI-C e un piccolo modulo in assembly.

In conclusione, l'obiettivo di questo progetto è stato raggiunto: è possibile pensare ad un meccanismo di rollback parziale che permetta di ridurre il numero di abort nelle transazioni, riducendo così , teoricamente, il costo in tempo in un'esecuzione. Invece di dover rilanciare dall'inizio una porzione di codice, si può ripartire dal punto in cui l'incoerenza si è generata, salvando il lavoro fatto in precedenza. Da alcuni test preliminari sui benchmark di TL2, risulta che la quantità di lavoro salvato è circa un terzo: infatti in genere il punto di incoerenza si colloca tra un terzo e la metà dell'esecuzione.

Per il futuro, si potranno effettuare test più approfonditi sulle performance e miglioramenti per la struttura dati per la gestione delle scrittura locali: sarebbe di interesse uno studio approfondito sul numero di bit da mascherare nell'indirizzo, per ottenere l'indice della tabella. Da questo numero dipende l'efficienza della struttura poiché si lega alle dimensioni e al numero di entry presenti: se fosse troppo elevato tali dimensioni crescerebbero eccessivamente, incidendo sul costo in termini di spazio.

Bibliografia

[1] S. Agarwal, R. Garg, M. S. Gupta,J. E. Moreira, *Adaptive incremental checkpointing for massively parallel systems*, 2004.

[2] S. Agarwal, M. Gupta, S. Rudrapatna, Kallikote, *Automatic Checkpointing and Partial Rollback in Software Transaction Memory*, 2011.

[3] S. Agarwal, M. Gupta, S. Rudrapatna, Kallikote, *Clustered Checkpointing and Partial Rollbacks for Reducing Conflict Costs in STMs*, 2010.

[4] V. Bala, E. Duesterwald, S. Banerjia, *DYNAMO: a transparent dynamic optimization system*, 2000.

[5] H. Bauer, C. Sporrer, *Reducing rollback overhead in time warp based distributed simulation with optimized incremental state saving*, 1993.

[6] C. D. Carothers, K. S. Perumalla, and R. Fujimoto, *Efficient optimistic parallel simulations using reverse computation*, 1999.

[7] D. Dice, O. Shalev, N. Shavit, *Transactional Locking II*, 2006.

[8] J. Fleischmann, P. A. Wilsey. *Comparative analysis of periodic state saving techniques in Time Warp simulators*, 1995.

[9] R. M. Fujimoto, *Parallel discrete event simulation*, 1989.

[10] M. Herlihy, J. E. B. Moss, *Transactional memory: architectural support for lock-free data structures*, 1993.

[11] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2*, 2011.

[12] D. R. Jefferson, *Virtual Time*, 1985.

[13] V. J. Marathe, M. L. Scott, *A Qualitative Survey of Modern Software Transactional Memory Systems*, 2004.

[14] C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, *STAMP: Stanford Transactional Applications for Multi-Processing*, 2008.

[15] A. Pellegrini *Formato degli Eseguibili e Strumenti Avanzati di Compilazione*, 2011.

[16] A. Pellegrini, R. Vitali, F. Quaglia, *Di-Dymelor: Logging only Dirty Chuncks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects*, pp. 2-4, 2009.

[17] R. Rönngren, M. Liljenstam, R. Ayani, J. Montagnat, *Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation*, 1996.

[18] A. Santoro, F. Quaglia, *A Version of MASM Portable Across Different UNIX Systems and Different Hardware Architectures*, 2005.

[19] A. Santoro, F. Quaglia, *Transparent State Management for Optimistic Synchronization in the High Level Architecture*, 2005.

[20] Z. Shao, *Executable and Likable Format (ELF)*, 2006.

[21] J. S. Steinman, *Incremental state saving in SPEEDES using C++*, 1993.

[22] M. M. Waliullah, *Efficient partial roll-backing mechanism for transactional memory systems*, 2011.

[23] D. West, K. Panesar, *Automatic Incremental State Saving*, 1996.

[24] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, W. Wong, *How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation*, 2008.