SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI INGEGNERIA

Tesi di Laurea in

INGEGNERIA INFORMATICA

Tracciamento trasparente ed efficiente di scritture su memoria dinamica con granularità arbitraria in architetture per il calcolo ottimistico

Relatore
Prof. Francesco Quaglia

Candidato
Alessandro Pellegrini

Anno Accademico 2007/2008

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

Indice

Introduzione **1**

1 Analisi delle tecnologie e strumenti **3**
1.1 Architettura CISC IA-32 4
1.1.1 Formato delle istruzioni 4
1.1.2 Istruzioni che operano su stringhe 6
1.1.3 Modalità di indirizzamento 7
1.1.4 Istruzioni di salto 10
1.1.5 Esempi riassuntivi 11
1.2 Architettura CISC x86-64 12
1.2.1 Formato delle istruzioni 14
1.2.2 Modalità di indirizzamento 15
1.3 Formato ELF 15
1.3.1 Formato del file 15
1.3.2 Header delle sezioni 16
1.3.3 Tabelle di rilocazione 17
1.3.4 Tabelle dei simboli 18
1.3.5 Tabella delle stringhe 18

2 Scelte Progettuali e Implementazione **20**
2.1 Analizzatore lessicale 20
2.2 Instrumentazione 24
2.2.1 Generazione della tabella delle istruzioni 24
2.2.2 Modifica degli ELF 25
2.2.3 Modifica delle tabelle di rilocazione 26
2.2.4 Gestione dei salti 26
2.2.5 Riepilogo del processo di instrumentazione 28
2.3 Tracciamento degli aggiornamenti 30
2.4 Correzione dei salti 33

3 Integrazione con un ambiente per il calcolo ottimistico **36**
3.1 Cenni sulla simulazione parallela e distribuita 36
3.1.1 Strategia di sincronizzazione ottimistica 38
3.1.2 Salvataggio degli stati 39
3.1.3 Copy State Saving (CSS) 40
3.1.4 Sparse State Saving (SSS) 40
3.1.5 Incremental State Saving (ISS) 42
3.2 ROOT-Sim 42
3.2.1 Livello applicativo 43

3.2.2 Livello del kernel di simulazione 44
3.2.3 Livello MPI 45
3.3 DyMeLoR 46
3.4 Di-DyMeLoR 47
3.4.1 Gestore della memoria dinamica 47
3.4.2 Tracciamento delle scritture 48
3.4.3 Operazioni di salvataggio degli stati 48
3.4.4 Operazioni di ripristino degli stati 49
3.4.5 Caching dei riferimenti 50
3.5 Dati Sperimentali 51

4 Lavori Collegati **54**
4.1 Debug e Vulnerability Assessment 54
4.2 Calcolo Parallelo 55

5 Conclusioni **57**

Bibliografia **58**

Elenco delle tabelle

1.1 Prefissi di ripetizione (gruppo 1) 5
1.2 Utilizzo delle istruzioni di formato stringa 6
1.3 Indirizzamento (in memoria) a 32 bit con il byte ModR/M 8
1.4 Indirizzamento (in memoria) a 32 bit con il byte SIB 9
1.5 Esempi di salti 11
1.6 Traduzione di un'istruzione `bzero` 13
1.7 Utilizzo della tabella delle stringhe 19

2.1 Flag utilizzati dal modulo `update_tracker` 25
2.2 Flag utilizzati dal modulo `branch_corrector` 29

Elenco delle figure

1.1 Schema del formato delle istruzioni per l'IA-32 4
1.2 Metodo di indirizzamento in memoria per l'IA-32 7
1.3 Esempio di codice Assembly per IA-32 12
1.4 Codice Assembly a confronto 13
1.5 Prefisso REX 14
1.6 Struttura di un file ELF 16
1.7 Elementi della tabella di rilocazione 17
1.8 Elementi della tabella dei simboli 18
1.9 Struttura della tabella delle stringhe 18

2.1 Struttura delle righe della tabella di istruzioni 21
2.2 Automa a Stati Finiti del parser 22
2.3 Automa a Stati Finiti di `format_addr_m` 23
2.4 Elementi della tabella delle istruzioni per `update_tracker` 24
2.5 Sostituzione dell'istruzione *jcxz* 27
2.6 Elementi della tabella `branch_table` 28
2.7 Finestra dello Stack di `update_tracker` 30
2.8 `update_tracker` per le istruzioni di tipo stringa 31
2.9 `update_tracker` per le istruzioni di tipo `mov` 32
2.10 Elementi della tabella di rilocazione 33
2.11 Funzionamento della correzione a run-time dei salti 34

3.1 Architettura di un sistema PDES 37
3.2 Violazione della causalità 38
3.3 Esempio di rollback 39
3.4 Esempio di rollback con CSS 40
3.5 Esempio di rollback con SSS 41
3.6 Schema dell'architettura di ROOT-Sim 43
3.7 Architettura di DyMeLoR 46
3.8 Strutture dati di Di-DyMeLoR 48
3.9 Operazione OR-XOR sulle bitmap 50
3.10 Statistiche di base per l'applicazione di test 53

Introduzione

In questo lavoro presento l'ideazione, il progetto e l'implementazione in linguaggio C ed Assembly di un sistema di tracciamento di accessi su memoria dinamica in architetture per il calcolo ottimistico.

Questo progetto si pone l'obiettivo di sviluppare una metodologia per l'identificazione a run-time di quali aree di memoria siano soggette ad operazioni di scrittura con granularità e dimensione arbitraria, in maniera disgiunta da qualsiasi libreria. In questo modo è possibile operare anche in contesti di memoria allocata dinamicamente, tramite librerie standard (quali `malloc`). Scopo parallelo del progetto è quello di realizzare un'implementazione di questa metodologia, che permetta un'esecuzione del software applicativo soggetta ad un overhead minimo rispetto a quello che normalmente avrebbe.

Ho svolto il progetto concentrandomi sulle architetture `CISC` dei processori *Intel-compliant* a 32 ed a 64 bit, rendendolo inoltre compatibile per le versioni 3 e 4 di `gcc`, coprendo in questo modo un ampio spettro di architetture (hardware e software) presenti sul mercato. Inoltre, ho posto molta cura nel progettare il sistema di tracciamento in modo da permettere al programmatore di continuare ad utilizzare tutti gli strumenti messi a disposizione dal linguaggio `ANSI-C`.

Il tracciamento degli accessi viene realizzato con tecniche di instrumentazione statica ed in maniera completamente trasparente al programmatore, utilizzando routine di monitoraggio scritte direttamente in Assembly. Questa scelta mi ha permesso di effettuare numerose ottimizzazioni che hanno consentito di svolgere le mansioni desiderate aggiungendo un overhead molto contenuto. Ho sviluppato il sistema di instrumentazione concentrandomi sugli eseguibili del sistema operativo `GNU/Linux`, specificatamente il formato `ELF`.

Questo sistema di tracciamento è stato infine integrato all'interno di una piattaforma di simulazione di tipo ottimistico, allo scopo di consentire il salvataggio incrementale di stati (realizzato tramite la sola copia di quelle parti di memoria che sono state toccate in scrittura dal precedente salvataggio), diminuendo così il tempo necessario ad effettuare uno snapshot dello stato degli oggetti della simulazione, facendo aumentare di conseguenza le prestazioni globali dell'intera piattaforma distribuita.

Il resto di questa tesi è organizzato come segue. Nel capitolo 1 presenterò brevemente tutti gli aspetti tecnologici hardware e software relativi all'ambito in cui è stato ideato e sviluppato questo progetto. Nel capitolo 2 descriverò gli obiettivi, le problematiche insorte e le scelte progettuali ed implementative riguardanti il processo di instrumentazione statica e le relative routine di tracciamento a run-time. Nel capitolo 3 presenterò le modalità con cui questo sistema di tracciamento di accessi su memoria dinamica è stato integrato all'interno di una piattaforma di calcolo parallelo, presentando altresì alcuni dati

sperimentali, riguardanti le prestazioni del sistema. Nel capitolo 4, infine, vengono illustrati alcuni lavori collegati che trattano sia di instrumentazione, in ambiti quali il *Debugging* ed il *Vulnerability Assessment*, sia di salvataggio incrementale di stati nell'ambito del calcolo parallelo, mettendo in chiara evidenza le differenze sostanziali tra quei lavori ed il mio.

Capitolo 1

Analisi delle tecnologie e strumenti

Essendomi prefisso l'obiettivo di tracciare in maniera trasparente gli accessi su memoria dinamica, mi sono immediatamente scontrato con il problema della *portabilità*.

Volendo aggiungere un overhead minimo all'esecuzione del software applicativo, la scelta di lavorare al livello del codice macchina è stata praticamente obbligata. Da qui è sorta la necessità di individuare l'architettura che permettesse un più vasto utilizzo della metodologia presentata.

Per questo motivo ho scelto di focalizzarmi sulle architetture *Intel-compliant* a 32 e 64 bit. La difficoltà deriva dal fatto che si tratta di architetture a paradigma `CISC` (`C`omplex `I`nstruction `S`et `C`omputer), ossia di formato variabile.

Le architetture *Intel-compliant* a pipeline (ossia quelle studiate per lo sviluppo di questo progetto) traducono le macro operazioni CISC in micro operazioni (o *μops*). A differenza di architetture a paradigma `RISC` (`R`educed `I`nstruction `S`et `C`omputer) quali *MIPS*, *SPARC*, *DEC Alpha* o *ARM*, le architetture *Intel-compliant*, per consentire una maggiore *espressività semantica*, prevedono l'utilizzo di un set di istruzioni a formato variabile. La maggiore espressività risiede nel fatto che sono previste istruzioni in grado di eseguire operazioni complesse come la lettura di un dato, la sua modifica e la sua scrittura direttamente in memoria.

Ciò implica che, per l'identificazione automatizzata di tutte quelle istruzioni che effettuano scritture su memoria, è necessario utilizzare un *analizzatore lessicale* che, all'interno del flusso di byte che compone il programma, sia in grado di identificare le singole istruzioni ed i campi che le compongono.

Inoltre ho scelto di concentrarmi sul sistema operativo `GNU/Linux` e sul formato di eseguibili `ELF` per consentire la successiva integrazione del sistema di tracciamento in una piattaforma di calcolo distribuito, precedentemente esistente, di nome *ROOT-Sim*.

Per rendere più evidenti le problematiche incontrate e le ragioni delle scelte progettuali adottate, di seguito farò una breve panoramica sulle architetture e sul formato degli eseguibili coinvolti.

1.1 Architettura CISC IA-32

Con *IA-32*, oppure a volte anche con *i386* ed *x86*, si definisce l'*Instruction Set* dei microprocessori prodotti principalmente da *Intel* ed *AMD*.

Caratteristica principale di questo *Instruction Set* è la lunghezza variabile. Essa era molto utile negli anni '70 ed '80, poiché permetteva di risparmiare molta memoria, allora estremamente costosa.

L'architettura Intel a 32 bit è ben descritta in [14]. Essa prevede, di base, la presenza di 8 registri a 32 bit *general-purpose*, chiamati `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi` ed `edi`, ai quali sono associati, nell'ordine, dei codici numerici da 0 ad 8. Tra i registri *general-purpose*, `esp` (o *stack pointer*) mantiene il riferimento in memoria alla cima dello stack, mentre `ebp` (o *base pointer*) mantiene il riferimento in memoria alla finestra dello stack relativa alla funzione correntemente in esecuzione.

Sono inoltre previsti sei *segment-registers* da 16 bit, chiamati `CS`, `DS`, `SS`, `ES`, `FS`, `GS`, che però non sono di interesse qualora si utilizzi un sistema operativo `Unix-like`.

In aggiunta sono presenti i registri `EFLAGS` (per il controllo dello stato del programma) e `EIP` (*instruction pointer*, anche chiamato *program counter*).

1.1.1 Formato delle istruzioni

Il formato delle istruzioni dell'IA-32 prevede vari campi, alcuni dei quali possono essere o meno presenti, come viene rappresentato in figura 1.1.

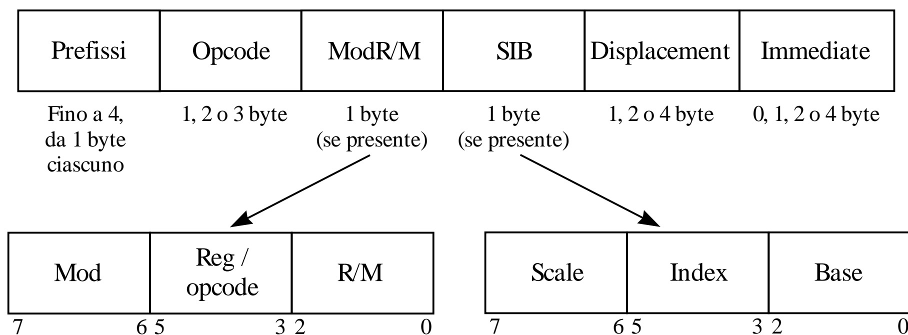


Figura 1.1: Schema del formato delle istruzioni per l'IA-32

I **prefissi** sono suddivisi in quattro gruppi e, al più, può essere presente un solo prefisso per gruppo. I prefissi che possono essere di interesse per la presente trattazione sono quelli dei gruppi 1, 3 e 4.

I prefissi di gruppo 1 sono dei prefissi che permettono di ripetere l'istruzione immediatamente seguente, fintanto che non venga verificata una condizione. Sono riassunti, con una breve spiegazione, nella tabella 1.1.

I prefissi di gruppo 3 e 4, invece (`0x66` e `0x67`) specificano che, rispettivamente, il dato immediato o l'indirizzo in memoria avranno dimensioni differenti rispetto a quanto specificato dall'opcode.

`REPNE/REPNZ (0xf2)`	Ripete mentre il registro `CX` è diverso da zero o fino a quando `CX` è uguale a zero
`REP/REPE/REPZ (0xf3)`	Ripete fino a quando `CX`=0

Tabella 1.1: Prefissi di ripetizione (gruppo 1)

L'**opcode** principale è anch'esso di formato variabile. Per la precisione, il primo byte dell'opcode identifica una *classe di istruzioni*. Qualora l'opcode sia composto da un solo byte, esso identifica univocamente un'istruzione. Qualora, invece, sia composto da più di un byte, il primo di essi identifica una famiglia di istruzioni, simili per *semantica* o per *campi utilizzati* oppure per *rappresentazione dei dati* adottata.

Alcune volte all'opcode primario viene associato un campo di 3 bit (denominato *Reg/Opcode*), all'interno del byte ModR/M.

Tutte quelle istruzioni che devono riferire un operando in memoria utilizzano un byte che specifica qual è la forma di indirizzamento. Questo byte prende il nome di **ModR/M**. Esso è composto da 3 sottocampi:

- Il campo *mod* (*mode*) viene combinato con il campo *r/m* (*re-gis-ter/memo-ry*) per formare 32 possibili valori: otto registri e 24 modalità di indirizzamento, come si può osservare nella tabella 1.3;
- Il campo *reg/opcode* specifica il numero di un registro, oppure tre bit aggiuntivi di informazioni per l'opcode. Il significato che deve essere assegnato a questo campo può essere desunto dall'opcode principale;
- Il campo *r/m* può specificare un registro come operando, oppure può essere combinato con il campo mod per codificare una modalità di indirizzamento. A volte alcune combinazioni del campo *mod* e del campo *r/m* vengono utilizzate per esprimere delle informazioni sugli opcode per alcune istruzioni.

Alcune codifiche del byte ModR/M specificano la presenza di un ulteriore byte per l'indirizzamento, che assume il nome di **SIB**. Esso è composto dai seguenti sottocampi:

- Il campo *scale* specifica il fattore di scala, che può valere 1, 2, 4 o 8;
- Il campo *index* specifica il numero del registro indice;
- Il campo *base* specifica il registro di base.

Nel paragrafo 1.1.3 saranno descritti nel dettaglio gli usi possibili dei byte ModR/M e SIB.

Alcune modalità di indirizzamento si servono di uno **spiazzamento** (che, nella terminologia Intel, viene chiamato *displacement*). Esso viene posto immediatamente dopo il byte ModR/M (oppure dopo il byte SIB, se quest'ultimo è presente). Se viene utilizzato uno spiazzamento, esso può avere una lunghezza di 1, 2 o 4 byte, a seconda dell'istruzione che lo utilizza

Istruzione	Prefisso	Sorg/Dest	Registri
movs	REP, REPE, REPZ	Entrambi	DS:SI, ES:DI
stos	REP, REPE, REPZ	Destinazione	ES:DI

Tabella 1.2: Utilizzo delle istruzioni di formato stringa

Se un'istruzione specifica un **dato immediato** (o *operando immediato*, come viene chiamato nella terminologia Intel), esso viene collocato sempre in coda all'istruzione (pertanto, dopo i byte ModR/M, SIB e dopo lo spiazzamento, se essi sono presenti).

Qualsiasi combinazione degli elementi del formato istruzioni è ammesso. L'unico limite imposto è che la lunghezza di una singola istruzione può essere al più di 16 byte.

1.1.2 Istruzioni che operano su stringhe

Un insieme importante di istruzioni è costituito da quelle che permettono di lavorare sulle stringhe[1]. Quelle a cui sono interessato per questo lavoro sono le istruzioni che consentono la scrittura o la copia su aree di memoria di dimensione arbitraria. Esse sono, rispettivamente, le istruzioni `stos` e `movs`. La prima permette di scrivere un certo numero di ripetizioni del valore contenuto nel registro `AX`. La seconda, invece, consente di copiare un'area di memoria in un'altra, della stessa dimensione.

Il funzionamento di queste istruzioni è semplice:

1. Si imposta il *Direction flag* all'interno del registro `EFLAGS`: se esso vale 0, la stringa verrà processata dall'inizio alla fine, altrimenti dalla fine all'inizio;
2. Nel registro `CX` viene caricato il numero di iterazioni dell'operazione;
3. L'indirizzo iniziale della stringa sorgente viene caricato nel registro `DS:SI`, mentre l'indirizzo iniziale di quella destinazione viene caricato nel registro `ES:DI`;
4. Il prefisso di ripetizione descrive quanto grande sarà l'area di memoria coinvolta dalla scrittura;
5. L'opcode dell'istruzione descriverà se l'unità della copia sarà costituita da uno, due, quattro od otto byte.

Nella tabella 1.2 vengono riportate le istruzioni con i prefissi che esse possono utilizzare. I prefissi sono quelli di ripetizione, descritti nella tabella 1.1.

L'esecuzione di una di queste istruzioni (con uno dei prefissi consentiti) segue i seguenti passi:

[1] Per *stringa* si intende, secondo la definizione classica, una sequenza ordinata di simboli. In questo caso particolare i simboli sono costituiti dai valori assunti da singoli byte.

1. Viene controllato il valore del registro `CX`. Se esso è 0, viene incrementato `EIP` e si passa quindi all'istruzione successiva;
2. Viene eseguita l'istruzione, secondo il formato stabilito dall'opcode;
3. Viene incrementato o diminuito il valore di DI (ed anche di SI, se utilizzato) a seconda che il valore del Direction flag sia 0 o 1;
4. Viene decrementato il valore di `CX`;
5. Si torna al punto 1.

1.1.3 Modalità di indirizzamento

Le modalità di indirizzamento supportate dall'architettura IA-32 sono molteplici e complesse.

Come mostrato in figura 1.2, è possibile identificare un operando in memoria, di taglia arbitraria, specificando 5 variabili:

- un registro di segmento. In realtà, questa variabile non può essere impostata esplicitamente: l'indirizzamento sarà relativo al segmento di codice all'interno del quale viene individuata l'istruzione di accesso.
- un *indirizzo di base* contenuto all'interno di uno dei registri *general purpose* (che assume di conseguenza il nome di *registro di base*);
- un *valore di indice* contenuto all'interno di uno dei registri *general purpose* (che assume di conseguenza il nome di *registro indice*);
- una *scala*, moltiplicatrice del valore di indice, che viene codificata direttamente nei byte dell'istruzione;
- uno *spiazzamento*, anch'esso codificato direttamente nei byte dell'istruzione.

Appare subito evidente che la complessità di questa modalità di indirizzamento è stata concepita per identificare in maniera concisa dati appartenenti a strutture dati non primitive, quali array o `struct`.

Nelle tabelle 1.3 e 1.4 sono riassunte le codifiche di tutte le modalità di indirizzamento in memoria a 32 bit descritte in [15].

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

Figura 1.2: Metodo di indirizzamento in memoria per l'IA-32

reg (16 bit)			AX	CX	DX	BX	SP	BP	SI	DI
reg (32 bit)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
binario			000	001	010	011	100	101	110	111
Indirizzo	**Mod**	**R/M**	**Byte ModR/M (esadecimale)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[–]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX] + disp8	01	000	40	48	50	58	60	68	70	78
[ECX] + disp8		001	41	49	51	59	61	69	71	79
[EDX] + disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX] + disp8		011	43	4B	53	5B	63	6B	73	7B
[–] + disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP] + disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI] + disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI] + disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX] + disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX] + disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX] + disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX] + disp32		011	83	8B	93	9B	A3	AB	B3	BB
[–] + disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP] + disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI] + disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI] + disp32		111	87	8F	97	9F	A7	AF	B7	BF

Tabella 1.3: Indirizzamento (in memoria) a 32 bit con il byte ModR/M

base (32 bit)			EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
binario			000	001	010	011	100	101	110	111
Indice scalato	**SS**	**Index**	**Byte SIB (esadecimale)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
nessuno		100	04	0C	14	1C	24	2C	34	3C
[EBP]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX * 2]	01	000	40	48	50	58	60	68	70	78
[ECX * 2]		001	41	49	51	59	61	69	71	79
[EDX * 2]		010	42	4A	52	5A	62	6A	72	7A
[EBX * 2]		011	43	4B	53	5B	63	6B	73	7B
nessuno		100	44	4C	54	5C	64	6C	74	7C
[EBP * 2]		101	45	4D	55	5D	65	6D	75	7D
[ESI * 2]		110	46	4E	56	5E	66	6E	76	7E
[EDI * 2]		111	47	4F	57	5F	67	6F	77	7F
[EAX * 4]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX * 4]		001	81	89	91	99	A1	A9	B1	B9
[EDX * 4]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX * 4]		011	83	8B	93	9B	A3	AB	B3	BB
nessuno		100	84	8C	94	9C	A4	AC	B4	BC
[EBP * 4]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI * 4]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI * 4]		111	87	8F	97	9F	A7	AF	B7	BF
[EAX * 8]	11	000	C0	C8	D0	D8	E0	E8	F0	F8
[ECX * 8]		001	C1	C9	D1	D9	E1	E9	F1	F9
[EDX * 8]		010	C2	CA	D2	DA	E2	EA	F2	FA
[EBX * 8]		011	C3	CB	D3	DB	E3	EB	F3	FB
nessuno		100	C4	CC	D4	DC	E4	EC	F4	FC
[EBP * 8]		101	C5	CD	D5	DD	E5	ED	F5	FD
[ESI * 8]		110	C6	CE	D6	DE	E6	EE	F6	FE
[EDI * 8]		111	C7	CF	D7	DF	E7	EF	F7	FF

Tabella 1.4: Indirizzamento (in memoria) a 32 bit con il byte SIB

Le nomenclature disp8 e disp32 nella tabella 1.3 stanno ad indicare che uno spiazzamento di 1 byte o di 4 byte seguirà il byte ModR/M o il byte SIB (se presente).

La presenza o meno del byte SIB è determinata sempre dal byte ModR/M: in tabella 1.3, laddove il campo *r/m* vale 100, l'indirizzo viene presentato come [–]. Ciò indica che dopo il byte ModR/M sarà presente il campo SIB. Altrimenti, il byte SIB non sarà presente.

Nella tabella 1.4, invece, si può notare che qualora il valore del campo *base* sia 101, viene omesso il registro ebp. Questo è dovuto al fatto che, come si può desumere dalla tabella 1.3, se il campo *mod* vale 00, deve essere presente unicamente uno spiazzamento a 32 bit. Pertanto, se il campo *base* del byte SIB varrà 101, questo valore identificherà il registro ebp soltanto se il campo *mod* del byte ModR/M avrà un valore diverso da 00.

1.1.4 Istruzioni di salto

Le istruzioni di salto trasferiscono il controllo del flusso del programma ad un'istruzione differente, senza salvare alcuna informazione sul punto di ritorno. Questo tipo di istruzioni si divide in due categorie principali, i salti *condizionati* ed i salti *incondizionati*.

I **salti incondizionati** si suddividono in quattro famiglie: salti *near*, salti *short*, salti *far* e *task switch*. Per il lavoro che ho svolto, in cui non è previsto del software che possa effettuare salti verso segmenti differenti o che possa richiedere un cambio di processo, sono di interesse unicamente i primi due tipi di salto.

I **salti near** sono salti che puntano ad un'istruzione all'interno dello stesso segmento di codice (puntato quindi dal segmento CS). L'operando di destinazione, se direttamente codificato nell'istruzione, è uno spiazzamento di quattro byte. La destinazione del salto, pertanto, corrisponderà al valore del registro %eip al quale sarà sommato lo spiazzamento.

In alternativa la destinazione del salto può essere specificata come indirizzo assoluto, sempre all'interno del segmento di codice corrente. In questo caso però, la destinazione è memorizzata in un registro oppure in una locazione di memoria. Si parla, in questo caso, di *salti indiretti* o *salti a registro*. Si può fare riferimento alla tabella 1.5, in cui vengono forniti alcuni esempi di istruzioni di salto, in sintassi AT&T.

I **salti short** sono dei salti che, come operando per specificare la destinazione, utilizzano un solo byte. Pertanto, il raggio d'azione di questi salti è limitato a [-128, +127] byte dal valore corrente di %eip.

I **salti condizionati** si possono dividere anch'essi in salti *short* e *near*, con le stesse identiche differenze che caratterizzavano i salti incondizionati.

La differenza tra queste due tipologie di salto sta nel fatto che i salti condizionati modificano realmente il flusso d'esecuzione dell'applicazione se e solo se una qualche condizione viene verificata. La condizione si riferisce tipicamente a dei valori dei flag contenuti nel registro EFLAGS.

In particolare, un'istruzione di salto condizionato verifica il valore di uno o più flag (tra il *Carry Flag*, l'*Overflow Flag*, il *Parity Flag*, il *Sign Flag* e lo *Zero Flag*) e, se essi sono in uno stato specificato dall'opcode (ossia, se verificano la

Istruzione	Descrizione
jmp .Label	Trasferisce il flusso di controllo all'istruzione situata nella locazione di memoria individuata dall'etichetta .Label
jmp short .Label	Trasferisce il flusso di controllo all'istruzione indicata dall'etichetta .Label, posta in un raggio di [-127, +128] byte
jmp *%eax	Trasferisce il flusso di controllo all'indirizzo contenuto nel registro %eax
jmp *(%eax)	Trasferisce il flusso di controllo all'indirizzo contenuto nella posizione di memoria puntata da %eax
jmp *(,%eax,8)	Trasferisce il flusso di controllo all'indirizzo di memoria calcolato come %eax * 8
jmp *$array(,%eax, 4)	Trasferisce il flusso di controllo all'indirizzo di memoria calcolato come [%eax * 4] + array[2]

Tabella 1.5: Esempi di salti

condizione), viene effettuato il salto alla destinazione specificata dall'operando che, come nel caso dei salti incondizionati, può essere di uno o quattro byte.

I flag vengono, ovviamente, impostati da un'istruzione di tipo `cmp` situata in una porzione di codice precedente all'istruzione di salto. In questo modo è possibile utilizzare le strutture di controllo della programmazione strutturata (*if-then-else* o cicli).

Fa eccezione l'istruzione `jcxz` (come d'altra parte gli equivalenti a 32 e 64 bit `jecxz` e `jrcxz`) che effettua il salto se il valore del registro `%cx` (o `%ecx`, o `%rcx`) è zero. Inoltre questa istruzione, pur effettuando un'operazione simile ad un confronto tra 0 ed il valore del registro, non modifica il valore di `EFLAGS`.

Infine, quest'istruzione prevede solamente un operando di un byte: non esiste una controparte di tipo *near*.

1.1.5 Esempi riassuntivi

Riporto, di seguito, alcuni frammenti di codice con relativo commento, che possono meglio mettere in evidenza quale sia la struttura intrinseca del formato istruzioni dell'architettura IA-32 e quali siano le problematiche legate alla loro interpretazione automatizzata.

Prendiamo in considerazione il frammento di codice in figura 1.3. In questa porzione di codice vengono mostrati, sulla sinistra, i byte corrispondenti alle istruzioni mostrate sulla destra.

Appare subito evidente il formato variabile. Inoltre è stato messo in risalto il differente significato dei vari byte. Con sottolineatura singola è stato indicato l'opcode, con sottolineatura doppia il byte ModR/M, con un cerchio il byte SIB, con sottolineatura ondulata lo spiazzamento e con un riquadro i dati immediati.

Il formato variabile fa sì, quindi, che istruzioni più *semplici* possano essere

rappresentate con meno byte, ottenendo così realmente il risparmio di memoria di cui ho parlato all'inizio. È altresì evidente come, per poter identificare la singola istruzione, sia necessario interpretarla in modo completo: ciascun byte, infatti, descrive il significato dei byte successivi. Non essendo possibile conoscere a priori la dimensione di un'istruzione, occorre leggere il codice byte per byte, associando a ciascuno di essi un *significato semantico*.

```
85 c0                            test %eax,%eax
75 09                            jnz 4c
c7 45 ec  00 00 00 00            movl $0x0, 0x14(%ebp)
eb 59                            jmp a5
8b 45 08                         mov 0x8(%ebp), %eax
8d 4c 24 04                      lea 0x4(%esp), %ecx
0f b7 40 2e                      movzwl 0x2e(%eax), %eax
```

Figura 1.3: Esempio di codice Assembly per IA-32

In figura 1.4 viene invece messo a confronto il codice assembly necessario a scandire ed impostare a 0 gli elementi di un array composto da 16 interi, che in linguaggio `C` corrisponde a:

```
int arr[16];
int main(void) {
  register int i = 0;
  for(i = 0; i < 16; i++)
    arr[i] = 0;
  return 0;
}
```

In figura 1.4(a) è presentato il codice assembly per IA-32, in quella 1.4(b) è presentato il codice per la stessa routine su architettura Sparc ed in figura 1.4(c) quello su architettura MIPS.

Risulta evidente che il formato variabile delle istruzioni consente, con estrema facilità, di indirizzare aree di memoria con un'unica istruzione ed in maniera più efficiente: laddove in architettura Intel per salvare un dato in un'area di memoria indirizzata con 32 bit è sufficiente una sola istruzione `mov`, nelle architetture RISC è necessario caricare prima la parte alta dell'indirizzo in un registro, poi sommarvi la parte bassa ed infine andare ad effettuare la scrittura sull'area di memoria puntata dal registro. Questo è dovuto al fatto che le architetture RISC, prevedendo un'istruzione di 32 bit, non possono fornire sufficiente spazio nei dati immediati per rappresentare l'indirizzo completo.

Infine è interessante notare (in tabella 1.6) come istruzioni complesse del linguaggio `C`, come ad esempio la `bzero`, trovino delle corrispondenze native, semplici ed efficienti nell'Instruction Set dell'Intel (le operazioni su stringa) mentre, in altre architetture, è necessario rappresentare quell'istruzione con una perifrasi, oppure utilizzando chiamate a funzioni.

1.2 Architettura `CISC` x86-64

L'architettura **x86-64** è un *sovrainsieme* dell'architettura IA-32. Questo significa che un processore x86-64 può eseguire sia codice a 32 bit, sia codice a 64 bit. Questa estensione dell'architettura è stata sviluppata da `AMD` ed è stata soltanto in seguito implementata anche da *Intel*, con i nomi *IA-32e* o *Intel64*.

```
31 c0                                  xorl %eax, %eax
                                       .Loop:
c7 04 85 00 96 04 08 00 00 00 00       movl $0, arr(,%eax,4)
40                                     incl %eax
83 f8 0f                               cmpl $15, %eax
7e ef                                  jle .Loop
```

(a)

```
03 00 00 81    sethi %hi(0x20400), %g1
86 10 63 34    or %g1, 0x334, %g3
84 10 20 00    clr %g2
               .Loop:
89 28 a0 02    sll %g2, 2, %g4
84 00 a0 01    inc %g2
80 a0 a0 0f    cmp %g2, 0xf
04 bf ff fd    ble 1047c
c0 20 c0 04    clr [ %g3 + %g4 ]
```

(b)

```
3c 1c 0f c0      lui gp,0xfc0
27 9c 78 80      addiu gp,gp,30848
03 99 e0 21      addu gp,gp,t9
8f 83 80 44      lw v1,-32700(gp)
24 02 00 0f      li v0,15
               .Loop:
24 42 ff ff      addiu v0,v0,-1
ac 60 00 00      sw zero,0(v1)
04 41 ff fd      bgez v0,.Loop
24 63 00 04      addiu v1,v1,4
```

(c)

Figura 1.4: Codice Assembly a confronto

Intel	Sparc	MIPS

cld
mov $size,%ecx
mov $address,%edi
xor %eax,%eax
repz stos %eax,%es:(%edi)
	sethi %hi(0x20400), %g1
	or %g1, 0x368, %o0
	mov $size, %o1
	call bzero
		lui gp,0xfc0
		addiu gp,gp,30816
		addu gp,gp,t9
		addiu sp,sp,-32
		li a1,$size
		lw t9,-32712(gp)
		jalr t9

Tabella 1.6: Traduzione di un'istruzione `bzero`

Nei paragrafi seguenti metterò in evidenza quali sono le differenze sostanziali tra l'architettura *Intel-compliant* a 32 e 64 bit.

1.2.1 Formato delle istruzioni

Il formato istruzioni dell'architettura x86-64 è sostanzialmente identico al corrispettivo a 32 bit. La differenza più significativa risiede nel fatto che è stato introdotto un nuovo prefisso, denominato `REX`, posizionato tra i 4 prefissi originali e l'opcode principale.

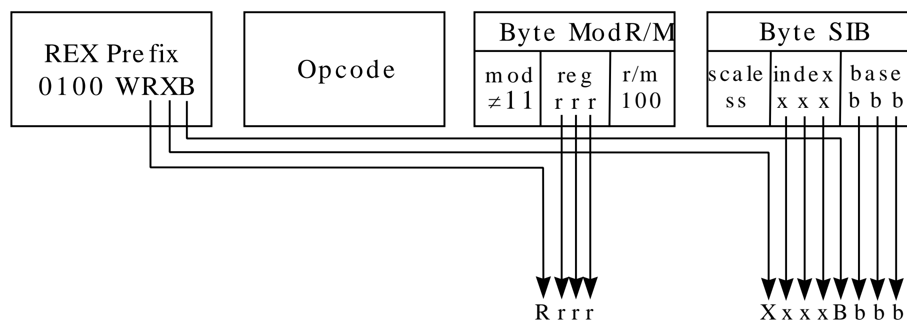


Figura 1.5: Prefisso REX

Il prefisso `REX`, come mostrato in figura 1.5, ha i primi 4 bit preimpostati a `0100`[3]. I restanti quattro bit del prefisso (denominati `REX.W`, `REX.R`, `REX.X`, `REX.B`) permettono di estendere l'architettura a 32 bit. In particolare:

- `REX.W`, se impostato a 1, specifica che la dimensione degli operandi coinvolti nell'istruzione sarà di 64 bit;
- `REX.R` è un'estensione del campo *reg* del byte ModR/M;
- `REX.X` è un'estensione del campo *index* del byte SIB;
- `REX.B` è un'estensione del campo *r/m* del byte ModR/M o del campo *base* del byte SIB oppure del campo *reg* dell'opcode.

La definizione dell'architettura specifica che viene preso in considerazione soltanto il byte REX immediatamente precedente all'opcode principale. Ciò significa che, in una singola istruzione, possono essere presenti più byte REX consecutivi, ma soltanto l'ultimo verrà tenuto in considerazione.

Il resto del formato istruzioni è generalmente identico: la dimensione dei vari campi (*spiazzamento*, *dati immediati*, ...) resta sempre di 32 bit.

[3]Nell'architettura IA-32 i prefissi `REX` corrispondono ad un insieme di 16 opcode, che occupano un'intera riga della mappa degli opcode. In particolare, occupano le posizioni `0x40` ÷ `0x4f`.

Questi opcode (a byte singolo) corrispondono a delle istruzioni valide (`inc` e `dec`). Nell'architettura a 64 bit, pertanto, queste istruzioni non sono più disponibili, ma se ne possono usare alcune equivalenti a 2 byte, inizianti con l'opcode di escape `0xff`.

Tuttavia vi è un'eccezione che compare per un piccolo insieme di istruzioni di tipo `mov`, quelle con gli opcode `0xb8` ÷ `0xbf`, che prevedono la presenza di un dato immediato di 64 bit qualora il campo `REX.W` sia impostato a 1. In tutti gli altri casi, il dato immediato a 32 bit subisce un'estensione del segno.

Inoltre il prefisso di gruppo 3 (`0x66`), che nell'architettura a 32 bit specificava che la dimensione dell'operando doveva essere di un byte, nell'architettura a 64 bit specifica invece, se `REX.W` è impostato ad 1, che la dimensione dell'operando dovrà essere di 16 bit.

È evidente, dal momento che alcuni campi del byte `REX` permettono di estendere i codici di accesso ai registri, che il numero di registri *general purpose* cambia. Vengono infatti aggiunti, nell'architettura a 64 bit, 8 nuovi registri che prendono i nomi `r8` ÷ `r15`. Inoltre la dimensione di tutti i registri passa da 32 bit a 64 bit.

1.2.2 Modalità di indirizzamento

Le modalità di indirizzamento in memoria dell'architettura x86-64 continuano ad essere quasi completamente quelle discusse per la versione a 32 bit, descritte quindi in figura 1.2 e nelle tabelle 1.3 e 1.4.

Tuttavia una nuova forma di indirizzamento è stata inserita nell'architettura a 64 bit. Essa prende il nome di **RIP-Relative Addressing**.

Se nell'architettura a 32 bit un indirizzamento relativo all'*instruction pointer* è possibile soltanto con le istruzioni di trasferimento del controllo, nell'architettura a 64 bit le istruzioni che utilizzano il byte ModR/M possono indirizzare in maniera relativa al valore corrente di `RIP`.

Nella tabella 1.3 viene specificato che, qualora il campo *mod* valga `00` ed il campo *r/m* valga `101`, viene indirizzato un dato con un semplice spiazzamento a 32 bit (che può essere inteso anche come un offset relativo all'indirizzo 0). Nell'architettura a 64 bit, invece, questo spiazzamento di 32 bit viene considerato come un offset relativo al valore corrente del registro `RIP`, a prescindere dalla presenza o meno di un prefisso `REX`.

Appare evidente, in questo scenario, che la modalità di indirizzamento `RIP`-Relative può essere utilizzata esclusivamente per indirizzare variabili globali.

1.3 Formato ELF

Come sarà descritto in seguito, per poter identificare le scritture su memoria occorre modificare il codice del programma applicativo in modo tale che possano essere eseguite, in maniera trasparente all'utente, delle routine di tracciamento delle scritture.

Si presenta subito, pertanto, l'esigenza di dover modificare gli eseguibili. Di seguito, presenterò brevemente il formato degli oggetti `ELF` (`E`xecutable and `L`inkable `F`ormat), così come viene presentato in [35].

1.3.1 Formato del file

Un file `ELF` è un file che permette di rappresentare del *codice rilocabile* (generato dagli *assemblatori* o dai *linker*, che può essere nuovamente linkato ad altri oggetti rilocabili), un *file eseguibile* (che contiene un programma pronto per l'esecuzione) oppure uno *shared object* (ossia un oggetto che può essere linkato staticamente

dal linker oppure caricato dinamicamente dopo che un altro programma è stato caricato in memoria. In questo caso assume anche il nome di *libreria*).

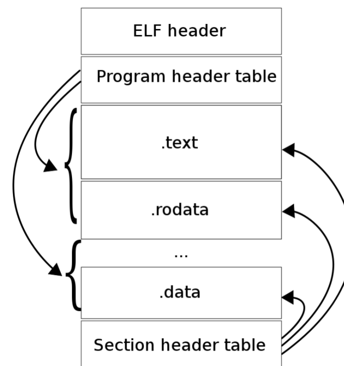


Figura 1.6: Struttura di un file ELF

Come mostrato in figura 1.6, un file `ELF` è suddiviso in sezioni. A seconda che descriva un oggetto rilocabile o eseguibile, alcune sezioni saranno o non saranno presenti.

In particolare, tutti i file `ELF` hanno, all'inizio, un header generale (chiamato *Elf header*) che descrive la struttura del file. In questa sezione, oltre a dare informazioni riguardanti il tipo di codice contenuto nel file (come ad esempio l'architettura per cui è stato compilato il programma, l'ordinamento *little-endian* o *big-endian* dei dati, la versione dell'oggetto, ...), si definisce dove potranno essere trovate le tabelle degli *header del programma* e degli *header di sezioni* in termini di offset dall'inizio del file.

Le *intestazioni del programma* descrivono al sistema come creare l'immagine del processo, specificando ad esempio come organizzare i segmenti e quali privilegi assegnare loro. Un oggetto *eseguibile* deve obbligatoriamente avere queste informazioni.

Le *intestazioni delle sezioni* descrivono invece le sezioni del file in termini di dimensione, di nome, di attributi e definiscono inoltre qual è la tabella di rilocazione associata ad una singola sezione. Un oggetto *rilocabile* deve obbligatoriamente avere queste informazioni.

Dal momento che in questo progetto utilizzerò unicamente oggetti rilocabili, nel seguito analizzerò alcuni dettagli della struttura di queste intestazioni. Inoltre analizzerò nel dettaglio alcune sezioni di interesse.

1.3.2 Header delle sezioni

Gli *header delle sezioni* sono organizzati in una tabella. Ciascun elemento di essa permette di identificare una sezione, specificando qual è il suo nome, quali sono i suoi attributi, qual è la sua dimensione e quale la sua posizione nel file (sempre intesa come offset dall'inizio del file).

Di interesse per questo lavoro sono, in particolare, tre campi dell'header: `sh_size`, `sh_offset` ed `sh_flags`. Il primo determina quale sarà la dimensione della sezione (in byte), mentre il secondo descrive dove, nel file `ELF`, è possibile individuarla. Quest'informazione è, ancora una volta, fornita come spiazzamento a partire dall'inizio del file. L'ultimo, invece, permette di determinare se una sezione contiene codice e se è di tipo eseguibile, scrivibile e/o allocabile

A tutte quelle sezioni che, al loro interno, racchiudono del codice che riferisce delle variabili è associata una tabella di rilocazione, posta anch'essa all'interno di un'altra sezione. Essa ha lo scopo di fornire al linker le informazioni su dove sono situati, nel flusso dei byte del codice, i riferimenti alle variabili e su quali sono i punti di ingresso per le chiamate alle funzioni. In questo modo il linker, una volta decisa la posizione all'interno dell'immagine del processo di una determinata variabile o funzione, andrà a sostituire quell'indirizzo laddove necessario.

Le due sezioni sono collegate tra loro tramite il campo `sh_info` della tabella di rilocazione. In esso, infatti, viene memorizzato il numero della sezione alla quale la tabella di rilocazione si riferisce. La tabella di rilocazione, inoltre, nel campo `sh_link`, memorizza l'indice della sezione contenente l'elenco dei simboli.

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;
```

(a)

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
    Elf32_Sword r_addend;
} Elf32_Rela;
```

(b)

Figura 1.7: Elementi della tabella di rilocazione

1.3.3 Tabelle di rilocazione

Le tabelle di rilocazione sono formate da una serie di elementi, la cui struttura è rappresentata in figura 1.7. In particolare, in figura 1.7(a) viene presentato un elemento della tabella di rilocazione senza addendo, mentre in figura 1.7(b) viene presentata una tabella di rilocazione con addendo.

In questi elementi i campi rappresentano quanto segue:

- `r_offset`: definisce la posizione su cui operare l'azione di rilocazione. Per un file rilocabile, esso è espresso come offset a partire dall'inizio della sezione;

- `r_info`: fornisce l'indice all'interno della tabella dei simboli in cui recuperare altre informazioni di interesse ed il tipo di rilocazione che deve essere eseguita. Le tipologie di rilocazioni accettabili dipendono dall'architettura hardware per cui l'oggetto è stato compilato.

- `r_addend`: se presente, definisce un addendo costante da sommare all'indirizzo che verrà sostituito nell'operazione di rilocazione. Viene tipicamente utilizzato per accedere ad elementi specifici di array o `struct`.

Qualora venga utilizzata una tabella di rilocazione formata da elementi senza addendo, all'interno del flusso di byte del testo possono essere inseriti degli

spiazzamenti relativi all'indirizzo di base della rilocazione: il linker, infatti, sommerà il valore dell'indirizzo invece di sovrascriverlo. In questo modo è possibile ottenere lo stesso risultato.

Dal campo `r_info` è possibile estrarre, tramite una macro predefinita (di nome `ELF_32_R_SYM()`), un indice relativo alla tabella dei simboli specificata da `sh_link`, all'interno dell'header della sezione.

1.3.4 Tabelle dei simboli

La tabella dei simboli raccoglie informazioni relative a tutte le variabili e a tutte le funzioni dell'oggetto, indipendentemente dalla sezione in cui essi vengano riferiti, ed è organizzata in elementi strutturati come in figura 1.8.

```
typedef struct {
    Elf32_Word st_name;
    Elf32_Addr st_value;
    Elf32_Word st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half st_shndx;
} Elf32_Sym;
```

Figura 1.8: Elementi della tabella dei simboli

L'unico campo tra questi, che effettivamente è di interesse per lavoro che sto descrivendo, è `st_name`. Esso contiene un offset all'interno della tabella delle stringhe che può essere facilmente individuata all'interno del file grazie al valore memorizzato nel campo della tabella iniziale del file `ELF`, che prende il nome di `e_shstrndx`. Esso contiene l'indice della tabella delle stringhe.

1.3.5 Tabella delle stringhe

Indice	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
00	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Figura 1.9: Struttura della tabella delle stringhe

La tabella delle stringhe è organizzata come in figura 1.9. Essa è composta da una serie di sequenze di caratteri, terminate dal carattere `NULL`[4]. L'intero object utilizza queste stringhe per rappresentare i nomi dei simboli ed i nomi delle sezioni.

Una stringa può essere riferita come un indice all'interno della tabella (come, ad esempio, viene fatto tramite il campo `st_name`).

Il primo byte, di indice zero, contiene un carattere `NULL`. Similmente, l'ultimo byte della tabella conterrà anch'esso un carattere `NULL`, assicurando così la

[4] Tipicamente chiamate *stringhe di testo*.

Indice	**Stringa**
`0`	Nessuno
`1`	`name`
`7`	`Variable`
`11`	`able`
`16`	`able`
`24`	*stringa nulla*

Tabella 1.7: Utilizzo della tabella delle stringhe

corretta terminazione delle stringhe. Se un oggetto punta alla stringa di indice 0, ciò significa o che l'oggetto non ha nome, o che il suo nome è nullo. Nella tabella 1.7 presento degli esempi relativi alla figura 1.9, per meglio chiarire come funziona il sistema di indicizzazione delle stringhe.

Capitolo 2

Scelte Progettuali e Implementazione

Nel capitolo 1 ho evidenziato brevemente quali siano le caratteristiche e le problematiche riguardanti le architetture ed i formati dei file utilizzati per la realizzazione di questo progetto.

In questo capitolo descriverò le scelte progettuali e implementative adottate per la realizzazione del sistema di tracciamento di accessi su memoria. In particolare descriverò per prima cosa in che modo vengano realizzati in maniera automatizzata il riconoscimento delle istruzioni e l'individuazione di tutte quelle istruzioni da modificare/tracciare. In seguito descriverò operativamente come avvenga l'inserimento delle routine di tracciamento e di correzione dei salti. Infine descriverò nel dettaglio come siano state realizzate le suddette routine, soffermandomi sulle scelte effettuate per far sì che aggiungano un basso overhead all'esecuzione.

2.1 Analizzatore lessicale

Come accennato nel paragrafo 1.1.1 a pagina 11, non è possibile conoscere a priori quale sarà la lunghezza della successiva istruzione incontrata nel flusso di byte del programma.

Per questo motivo ho realizzato un parser capace di interpretare l'intero Instruction Set delle architetture *Intel-compliant* a 32 ed a 64 bit.

Il fulcro del parser è una tabella: in essa sono raggruppate, in ordine di opcode, le famiglie di istruzioni. Così come viene definito nell'appendice B di [16], tutte le istruzioni sono raggruppate in alcune famiglie. Inoltre, sempre dall'appendice B di [16], è possibile risalire facilmente agli operandi afferenti ad ogni singola istruzione, siano essi 1, 2 o 3.

Ciascuna riga della tabella, pertanto, è organizzata come con la `struct` presentata in figura 2.1.

Questa struttura permette di conoscere, a partire dal primo byte dell'opcode, numerose informazioni sull'istruzione che si sta incontrando:

- il **menmonico** dell'istruzione, ossia il suo nome in codice assembly, grazie al campo `instruction`. Se la riga descrive una famiglia di istruzioni o non si tratta di un opcode valido, questo campo punterà a `NULL`.

```
struct _insn {
  char *instruction;
  enum addr_method addr_method[3];
  enum operand_type operand_type[3];
  void (*esc_function)(struct disassembly_state *);
  bool to_be_instrumented;
};
```

Figura 2.1: Struttura delle righe della tabella di istruzioni

- il **metodo di indirizzamento** di ciascun operando dell'istruzione (ad esempio, se l'operando sarà un registro, una locazione di memoria, un offset di salto, ...), tramite il campo `addr_method`[1];
- il **tipo di operando**, ossia informazioni quali la dimensione standard dell'operando in questione e la tipologia di registri in cui può essere trovato, tramite il campo `operand_type`;
- la **funzione del parser** che permette di analizzare la particolare classe di istruzioni, tramite il puntatore `*esc_function`. Queste funzioni ricevono come parametro la struttura `disassembly_state` che tiene traccia del lavoro fatto fino ad un certo istante per interpretare l'istruzione corrente. Se l'istruzione ha un opcode di un solo byte, il campo punta a `NULL`;
- se l'istruzione è da instrumentare, il campo `to_be_instrumented` è impostato a 1. Questo campo permette di escludere dall'analisi (ma non dal parsing!) tutte le istruzioni non di interesse.

Poiché la tabella delle istruzioni è ordinata secondo il valore del primo byte di opcode (da `0x00` a `0xff`), è chiaro che l'accesso a tutte queste informazioni è immediato, dopo la lettura di un singolo byte.

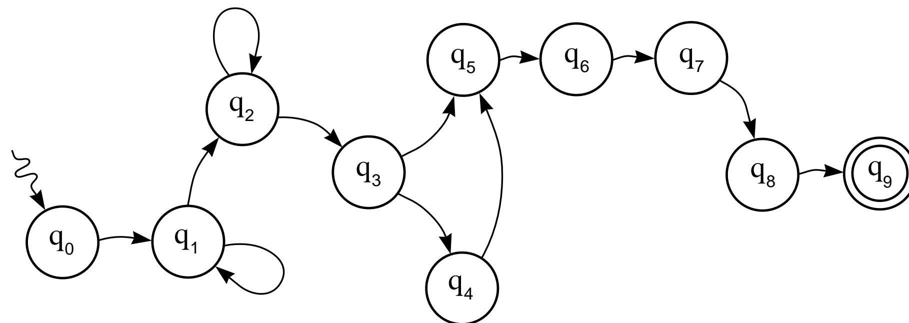
Per scendere nel dettaglio, il parser è in grado di identificare un'istruzione collocata in una certa posizione del testo ed estrarne tutte le informazioni di interesse. I passi seguiti sono rappresentati dall'*Automa a Stati Finiti* mostrato in figura 2.2.

Come si può notare, il parser è in grado di gestire tutte le istruzioni sia per l'architettura a 32 bit, sia per quella a 64 bit. I prefissi REX, infatti, sono memorizzati nella tabella di istruzioni come le istruzioni `inc` e `dec` dell'architettura a 32 bit. Se però l'oggetto analizzato è stato compilato per l'architettura a 64 bit, verrà eseguita una routine di estrazione dei prefissi `REX`, facendo in modo che quei byte non possano essere interpretati come istruzioni.

Per sapere se i byte ModR/M e SIB sono presenti, sono state utilizzate delle `define`: nell'appendice B di [16], infatti, si può notare che (salvo alcune eccezioni gestite dalle funzioni associate alle famiglie di istruzioni), gli opcode che determinano la presenza del byte ModR/M sono conosciuti.

È pertanto possibile sapere, dal byte dell'opcode, se sia presente o meno un byte ModR/M. Una volta letto, per determinare se sia presente anche un

[1]Poiché nell'architettura Intel un'istruzione può avere al più tre operandi, il campo è costituito da un array di tre elementi.



q_0: Inizializza il parser e determina se si opera a 32 o 64 bit
q_1: Estrae gli eventuali prefissi
q_2: Se a 64 bit, estrae i prefissi REX
q_3: Recupera il primo byte dell'opcode
q_4: Invoca la funzione associata alla classe di istruzioni
q_5: Se presente, legge il byte ModR/M
q_6: Se presente, legge il byte SIB
q_7: Se presente, legge il displacement
q_8: Per ciascun operando dell'istruzione, analizza il suo formato
q_9: L'istruzione è stata processata, pertanto termina

Figura 2.2: Automa a Stati Finiti del parser

byte SIB, è sufficiente analizzarne soltanto alcuni bit, ciò che viene fatto tramite un'altra `define`.

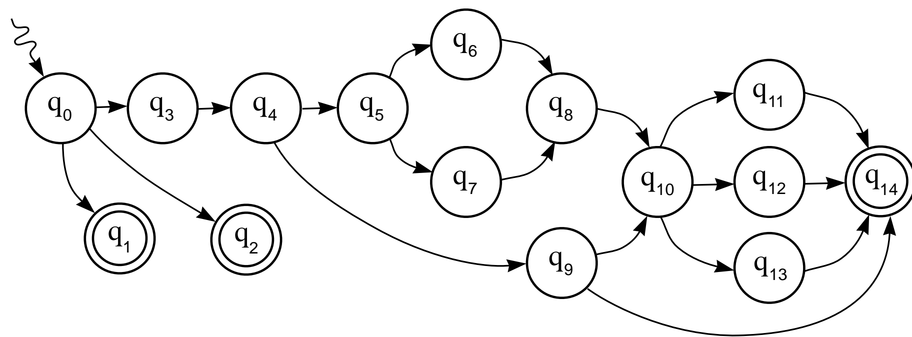
L'analisi dei primi byte dell'istruzione è pertanto veloce ed efficace. Ciò che interessa di più per il nostro scopo (ossia conoscere dov'è che un'eventuale istruzione scriverà in memoria) viene ottenuto nello stato q_8.

In quello stato, infatti, per ciascun operando viene invocata una funzione di analisi che estrae informazioni relative al suo formato e alla sua specie.

Un'apposita funzione, chiamata `format_addr_m`, è stata scritta per gestire gli operandi che rappresentano locazioni in memoria. Questa funzione opera come rappresentato nell'*Automa a Stati Finiti* in figura 2.3.

Per semplificare, la funzione `format_addr_m` procede con un'analisi incrementale a partire dal byte ModR/M. Si è certi che un byte ModR/M sia presente poiché questa funzione viene invocata esclusivamente quando occorre gestire un operando in memoria (che, come minimo, sfrutta il byte ModR/M).

Successivamente la funzione controlla tutti quei campi che portano alla presenza del SIB e dello spiazzamento, controllando proprio se si incappa in alcune delle eccezioni presentate nel paragrafo 1.1.3. Una volta determinato se si stanno verificando alcune eccezioni, la funzione si limita a copiare in un'apposita struttura chiamata `insn_info` tutti i campi di interesse.



q_0: Determinazione della dimensione della scrittura
q_1: Se l'operando è un registro, lo tratta come tale
q_2: Se mod == 11, si è verificato un errore
q_3: Determina se si opera su registri a 32 o 64 bit
q_4: Se mod == 00 e r/m == 101, allora c'è disp32
q_5: Se rm == 100 allora c'è il byte SIB
q_6: Se mod == 00 e base == 101, SIB non determina una base
q_7: Estrae il codice del registro di base
q_8: Estrae la scala e il codice del registro indice
q_9: Non c'è SIB: r/m determina il registro di base
q_{10}: Di seguito potrebbe esserci uno spiazzamento
q_{11}: Se mod == 01, lo spiazzamento è di 8 bit
q_{12}: Se mod == 10, lo spiazzamento è di 32 bit
q_{13}: Se si è passati per q_6, lo spiazzamento è di 8 bit
q_{14}: L'indirizzo in memoria è stato interpretato correttamente

Figura 2.3: Automa a Stati Finiti di `format_addr_m`

2.2 Instrumentazione

Il parser descritto nel paragrafo precedente permette l'interpretazione e l'individuazione, in maniera statica, di tutte quelle istruzioni che effettuano scritture su memoria.

Una volta che il parser ha identificato un'istruzione di interesse, il modulo di instrumentazione del codice antepone ad essa una `call` ad un modulo chiamato `update_tracker` che si preoccupa di processare le scritture in memoria. Questo modulo verrà descritto in modo dettagliato nel paragrafo 2.3.

Questa modalità operativa è tipica nell'ambito del tracciamento dei riferimenti in scrittura a memoria, come si può leggere in modo approfondito in [37]. Tuttavia, poiché l'obiettivo finale che mi sono posto è quello di integrare questo sistema di tracciamento in una piattaforma di calcolo ottimistico, si presentano delle questioni prestazionali necessariamente più stringenti, per tentare di effettuare l'operazione di tracciamento con il minor numero di istruzioni macchina possibile, scegliendo quelle che abbiano necessità di un numero minore di *μops*.

2.2.1 Generazione della tabella delle istruzioni

Dal momento che il parser è in grado di estrarre tutte le informazioni di interesse per individuare le aree di memoria soggette ad una scrittura, è possibile organizzare il processo di instrumentazione (e di successivo tracciamento) in modo tale che debba essere eseguito il minor numero possibile di operazioni a run-time.

In particolare, dopo aver inserito la `call`, il modulo di instrumentazione si preoccupa di inserire in una tabella apposita (chiamata *tabella di istruzioni di* `update_tracker`) una riga formata dai campi mostrati in figura 2.4.

```
struct update_tracker_entry {
    unsigned long ret_addr;
    unsigned int size;
    char flags;
    char base;
    char index;
    char scale;
    long displacement;
};
```

Figura 2.4: Elementi della tabella delle istruzioni per `update_tracker`

Questi campi mantengono le seguenti informazioni:

- `ret_addr`: il valore di ritorno della chiamata ad `update_tracker`. Corrisponde all'indirizzo in cui è situata l'istruzione di scrittura che ha causato l'invocazione del modulo di tracciamento;
- `size`: la dimensione della scrittura. In caso di istruzioni `movs` o `stos`, la dimensione si riferisce ad una singola esecuzione dell'istruzione (non viene tenuto in considerazione quanto espresso dai prefissi `rep` o `repe`);
- `flags`: permette di distinguere se si tratta di istruzioni di tipo `mov` o di operazioni su stringa. Qualora si tratti di un'istruzione di tipo `mov`, descrive anche quali campi (*base*, *indice*, *scala*, *displacement*) vengono utilizzati dall'istruzione. I bit di questo campo, ed il loro significato, sono elencati nella tabella 2.1;

Bit	Significato
0	Determina se si tratta di un'operazione di tipo mov o su stringhe
1	Determina se viene utilizzato un registro di base
2	Determina se viene utilizzato un registro di indice

Tabella 2.1: Flag utilizzati dal modulo `update_tracker`

- `base`: mantiene il codice numerico (come descritto nel paragrafo 1.1 a pagina 4) associato al registro di base. Se non viene utilizzato alcun registro, il valore del campo è impostato a 0. Il valore 0, però, può anche corrispondere al registro `%eax`: a questo campo viene assegnato un significato piuttosto che l'altro in funzione del valore del campo `flags`;
- `index`: mantiene il codice numerico del registro di indice. Come per il registro di base, se esso non è utilizzato, questo campo viene impostato a 0;
- `scale`: il valore della scala, ossia 1, 2, 4 oppure 8. Se non viene utilizzato un registro di indice, il campo viene impostato a 0;
- `displacement`: lo spiazzamento (direttamente estratto dai byte dell'istruzione).

Ho scelto di utilizzare come campo per descrivere quali registri vengano utilizzati come base e come indice i codici utilizzati negli opcode perché questo permette al modulo `update_tracker` di recuperare il valore contenuto in quei registri con una sola istruzione, come verrà descritto nel dettaglio nel paragrafo 2.3 a pagina 32.

In particolare, gli elementi della tabella vengono ordinati in modo tale che essa risulti essere una tabella di hash, la cui chiave di accesso è proprio il valore di ritorno delle istruzioni. Eventuali collisioni sono gestite tramite trabocco, ma, per migliorare l'efficienza, il modulo di instrumentazione è in grado di accorgersi se esse crescono troppo, e può decidere quindi di aumentare la dimensione della tabella (ampliando il numero di chiavi disponibili).

2.2.2 Modifica degli ELF

L'aver inserito delle istruzioni (in particolare, delle `call`) all'interno del codice rende necessario modificare la struttura del file `ELF` che si sta processando.

Per quanto in [27] sia stata presentata una libreria in grado di gestire questo formato, essa è fortemente orientata alla generazione di file di questo tipo (ad esempio, ad uso di assemblatori e linker). Dal momento che il mio scopo è quello di modificare file già precedentemente generati, ed in particolare modificarne soltanto delle parti ben precise, ho deciso di scrivere da zero un modulo dedicato (e quindi più efficiente) di gestione dei file di questo tipo.

Alla luce di quanto visto nel capitolo 1.3, gli oggetti `ELF` hanno una struttura basata su tabelle di header associate a delle sezioni. Il modulo di gestione che

ho scritto, pertanto, dopo aver aperto il file specificato ed aver controllato che il *magic number* del file sia corretto, carica in memoria gli header delle sezioni.

La libreria, in seguito, fornisce delle API che consentono al resto del software di ingrandire e ridurre la dimensione delle sezioni, di sostituire una sezione con un'altra (conservando l'header) e di modificare l'ordine delle sezioni. Inoltre, tramite un'ulteriore API, la libreria salva nel file la nuova versione modificata.

Tutte queste operazioni di modifica, relativamente semplici, vengono effettuate principalmente sfruttando i campi `sh_size` ed `sh_offset` precedentemente descritti.

2.2.3 Modifica delle tabelle di rilocazione

Poiché vengono mossi elementi all'interno della sezione testo, è necessario modificare anche i riferimenti a quelle locazioni in cui il linker, al momento della generazione dell'eseguibile finale, andrà ad svolgere le operazioni di rilocazione descritte nel paragrafo 1.3.3.

Questa operazione viene effettuata appoggiandosi sempre al modulo di gestione dei file `ELF`. Infatti essa fornisce in aggiunta due API apposite:

- `shift_functions`: dato un file `ELF` ed una posizione nel testo, applica a tutte le entry di rilocazione relative a *simboli* ed afferenti all'area successiva alla posizione specificata uno shift desiderato;

- `shift_reloc_entry`: dato un file `ELF` ed una posizione nel testo, applica a tutte le entry di rilocazione relative a *funzioni* ed afferenti all'area successiva alla posizione specificata uno shift desiderato;

La scelta di dividere le operazioni di rilocazione delle funzioni dalla rilocazione dei simboli di variabile è stata dettata dall'esigenza di rendere il codice quanto più semplice possibile: le informazioni relative ai simboli, infatti sono collocate nella tabella di rilocazione, mentre le informazioni relative alle funzioni, poiché non sono associate a vere operazioni di rilocazione, ma servono piuttosto a specificare degli *entry point* per istruzioni di `call`, sono collocate unicamente nella tabella dei simboli.

Ogni volta che il modulo di instrumentazione del codice individua un'istruzione da instrumentare, inserisce la `call` ed in seguito, con le API appena descritte, rende coerenti al nuovo layout di codice tutte le tabelle contenenti riferimenti a posizioni nel codice. Inoltre effettua un'operazione di aggiunta di simboli (necessaria per far sì che il linker si accorga dell'esistenza del modulo `update_tracker)` che implica una modifica apposita delle tabelle dei simboli e delle stringhe precendentemente descritte.

2.2.4 Gestione dei salti

Ciò che non viene corretto con l'approccio di modifica delle informazioni di rilocazione appena descritto sono le destinazioni dei salti. Se prendiamo in considerazione un'istruzione di codice assembly, esso specificherà la destinazione tramite una *label.* Essa però non sopravvive fino all'object: l'assemblatore, infatti, traduce già questa destinazione con dei byte che esprimono uno spiazzamento relativo, come descritto precedentemente nel paragrafo 1.1.4.

Risulta pertanto necessario adottare un approccio differente per la correzione dei salti. Durante l'operazione di inserimento delle `call` viene generata una

lista contenente un'associazione tra indirizzi e spostamento in avanti applicato: in questo modo è possibile sapere, dato un indirizzo, di quanto esso è stato in realtà modificato.

Modifica dei salti diretti

Dopo aver completato l'instrumentazione, pertanto, il codice viene nuovamente scandito alla ricerca di istruzioni di salto da correggere. Ogni volta che si incontra una tale istruzione, viene processato lo spiazzamento per determinare il riferimento cui punta l'istruzione di salto. Una volta determinata la destinazione del salto, viene consultata la lista precedentemente popolata alla ricerca di un range di indirizzi all'interno del quale cada quello corrente. Una volta individuato, viene calcolato qual è lo spiazzamento da applicare. Esso viene poi sommato alla destinazione del salto, che viene riscritta all'interno del codice, correggendo così la jump.

Facendo riferimento ai salti di tipo *short*, appare subito evidente che, con l'inserimento di istruzioni all'interno del codice, l'intervallo di [-127, +128] byte potrebbe non essere più sufficiente. Ho deciso, pertanto, di adottare un approccio conservativo: tutte le istruzioni di salto di tipo *short* vengono convertite automaticamente dal modulo di instrumentazione in salti di tipo *near*, facendo attenzione a scegliere istruzioni che abbiano lo stesso significato semantico (nel caso dei salti condizionali). Questa scelta non impatta sulle prestazioni dell'applicazione perché, a partire dai modelli di processore successivi all'80486, i salti di tipo *short* e quelli di tipo *near* richiedono esattamente lo stesso numero di *μops* per essere eseguiti. Inoltre, per entrambi, le architetture effettuano lo stesso tipo di *predizione dei salti*.

Per l'istruzione `jcxz`, che, come spiegato nel paragrafo 1.1.4, non ha una controparte di tipo *near*, ho adottato una tecnica di sostituzione differente. L'ho rimpiazzata, infatti, con la porzione di codice presentata in figura 2.5.

Questo frammento di codice esegue l'istruzione `jcxz`. Se la condizione viene verificata (ossia se il registro `CX` ha valore 0) viene effettuato un salto all'istruzione `jmp` *rel32* che punterà alla destinazione corretta (ossia, alla destinazione originale cui è stato applicato l'offset).

Se la condizione, invece, non è verificata, il salto indicato da `jcxz` non viene eseguito, ma viene eseguita invece la `jmp short` che scavalca il salto contenente la destinazione corretta, facendo proseguire l'esecuzione del codice.

La semantica dell'istruzione `jcxz` viene pertanto rispettata (infatti, tra le altre cose, non viene modificato il valore del registro `EFLAGS`), sicuramente però a discapito delle prestazioni. È da sottolineare tuttavia che la frequenza di occorrenza di istruzioni di questo tipo è molto bassa.

```
  jcxz .Salta
  jmp short .NonSaltare
.Salta:    jmp rel32
.NonSaltare:
```

Figura 2.5: Sostituzione dell'istruzione *jcxz*

Instrumentazione dei salti indiretti

Non tutte le istruzioni di salto, però, possono essere corrette staticamente. Nel capitolo 1.1.4 a pagina 10 si è discusso dei *salti indiretti* (o *salti a registro*). Poiché la destinazione del salto, in quei casi, dipende dal flusso di esecuzione dell'applicazione, non è possibile conoscere in maniera apprioristica quale sarà la destinazione.

Per questo motivo, ho deciso di adottare una tipologia di correzione a runtime del tutto simile al tracciamento degli accessi in memoria. Dopo aver corretto i salti diretti, infatti, il modulo di instrumentazione effettua un ulteriore passaggio sul codice alla ricerca di salti indiretti. Nel momento in cui uno di essi viene individuato, il modulo gli antepone una `call` ad una routine di correzione chiamata `branch_corrector`, che verrà descritta nel dettaglio nel paragrafo 2.4.

Come per il tracciamento degli accessi in memoria, il modulo di instrumentazione genera una tabella (chiamata `branch_table`) strutturata con elementi riportati in figura 2.6.

```
struct _branch_insn {
    unsigned long ret_addr;
    char flags;
    char base;
    char idx;
    char scala;
    long offset;
};
```

Figura 2.6: Elementi della tabella `branch_table`

In questo modo il modulo di correzione dei salti sarà in grado di calcolare, a tempo di esecuzione, qual è la destinazione del salto che, per un'istruzione di questo tipo, viene rappresentata con il metodo di indirizzamento proprio delle locazioni in memoria, descritto in 1.1.3.

Ricordando quanto detto nel paragrafo 1.1.4 a pagina 10, i salti indiretti nelle architetture *Intel-compliant* consentono di memorizzare la destinazione del salto in registri oppure in locazioni di memoria. Appare evidente, quindi, che il contenuto di un registro può avere un doppio significato: può essere la destinazione di un salto, oppure l'indirizzo in memoria in cui recuperare la destinazione.

Per questo motivo, il campo `flags` permette di identificare se l'opcode dell'istruzione determina l'uno o l'altro caso. I bit del campo `flags`, con si relativi significati, sono riportati in tabella 2.2.

Anche questa tabella viene organizzata in modo tale che risulti essere una tabella hash, con la possibilità di essere ridimensionata dinamicamente qualora il numero di collisioni aumenti troppo.

2.2.5 Riepilogo del processo di instrumentazione

In questo paragrafo voglio riassumere il processo di analisi ed instrumentazione del codice. Si tratta di un processo complesso e strutturato in più fasi, che vengono riportate qui di seguito:

1. una prima scansione del testo dell'applicazione permette di identificare quante saranno le chiamate al modulo `update_tracker`. Contemporanea-

Bit	Significato
`0`	Non utilizzato
`1`	Determina se viene utilizzato un registro di base
`2`	Determina se viene utilizzato un registro di indice
`3`	Determina se l'indirizzo calcolato è la destinazione oppure una posizione in memoria

Tabella 2.2: Flag utilizzati dal modulo `branch_corrector`

mente, vengono identificati quanti salti *short* dovranno essere convertiti in salti *near*, quante istruzioni `jcxz` sono presenti, e quante chiamate al modulo `branch_corrector` andranno inserite;

2. tramite queste informazioni viene calcolata la dimensione della nuova sezione di codice. È un'operazione relativamente semplice, dal momento che le dimensioni degli oggetti di partenza (ad esempio, i salti *short*) e le dimensioni degli oggetti di destinazione (le `call`, i salti *near* e il blocco di codice per sostituire l'istruzione `jcxz`) sono note;

3. una seconda scansione si occupa di inserire le `call` ai due moduli di aggiornamento. Contemporaneamente vengono popolate la tabella di istruzioni per `update_tracker`, quella per `branch_corrector`, la lista di shift delle posizioni e vengono salvate informazioni relative alle posizioni degli oggetti che il linker dovrà rilocare;

4. ogni volta che viene inserita una `call`, tramite le API fornite dal modulo di gestione degli `ELF`, vengono spostate in avanti (di 5 byte, la dimensione di una `call`) tutte le entry di rilocazione interessate;

5. ogni volta che viene individuata un'istruzione di salto, viene effettua la conversione (se necessaria) in salto *near* e viene memorizzata la destinazione originale del salto in una lista di metadati temporanei. Se viene incontrato un salto indiretto, oltre l'aggiunta della `call`, esso viene sostituito con una jump ad una differente sezione di codice. Il motivo di questa scelta verrà spiegato nel paragrafo 2.4;

6. al termine della seconda scansione, viene controllato se le tabelle di hash hanno un numero troppo elevato di collisioni (nel qual caso la loro dimensione viene fatta crescere). Esse vengono poi salvate su file. Inoltre vengono inserite nell'object delle entry di rilocazione che permetteranno al linker di aggiungere nelle `call` i riferimenti ai moduli. Infine, vengono aggiunte delle entry di rilocazione che permetteranno al linker di inserire i riferimenti corretti alle variabili nelle tabelle dei moduli a run-time;

7. una terza (ed ultima) scansione del codice si preoccupa di correggere le destinazioni dei salti statici, andando (per ciascuna istruzione di salto) a controllare qual era la destinazione originale e cercando, nella lista di shift, di quanto essa dev'essere corretta;

8. al termine della terza scansione, la lista di shift viene convertita in una tabella, per permettere al modulo `branch_corrector` di correggere a runtime i salti indiretti. Questa tabella viene anch'essa salvata su file;

9. in ultimo, (i) tramite le API del modulo di gestione degli `ELF`, viene salvata la versione modificata dell'object, (ii) tramite un tool apposito vengono inserite nell'object le tre tabelle che in precedenza erano state salvate su file e, (iii) tramite gli strumenti standard di compilazione (`gcc` ed `ld`) vengono effettuate tutte le operazioni di linking tra i vari moduli, producendo così un oggetto contenente il software di livello applicativo (posto in una posizione ben determinata[2]), le tabelle ed i moduli di correzione a run-time.

2.3 Tracciamento degli aggiornamenti

Una volta completato il processo di instrumentazione, il tracciamento degli accessi a runtime risulta abbastanza semplice. Ogni volta che un'istruzione di scrittura in memoria sta per essere eseguita, viene prima effettuata una chiamata alla routine `update_tracker`.

Per far sì che la routine di tracciamento degli accessi abbia un overhead minimo, essa è stata scritta direttamente in codice assembly: questo mi ha permesso di effettuare una serie di ottimizzazioni (bilanciando la scelta tra istruzioni richiedenti un minor numero di *μops* ed istruzioni con espressività semantica maggiore, che consentissero in maniera più sintetica di effettuare operazioni complesse).

Immediatamente dopo la chiamata, la funzione `update_tracker` crea nello stack una fotografia dello stato del processore al momento della chiamata. In particolare, esso verrà assemblato come riportato in figura 2.7.

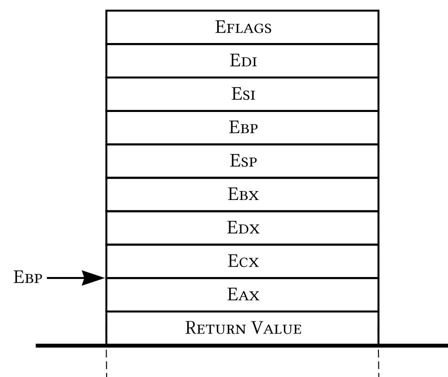


Figura 2.7: Finestra dello Stack di `update_tracker`

Nello stack vengono inseriti i valori dei registri al momento della chiamata di funzione, in ordine di codice numerico (così come descritto in 1.1 a pagina 4). Inoltre poiché nell'esecuzione della routine vengono effettuati dei confronti,

[2] È necessario specificare una posizione ben determinata nel software di livello applicativo poiché gli indirizzi utilizzati in un file non rilocato sono relativi a partire dall'inizio della sezione, mentre i valori di ritorno sono degli indirizzi assoluti. Se così non si facesse, il modulo `update_tracker` non avrebbe informazioni circa l'istruzione che ha causato la sua attivazione.

viene salvato anche il registro EFLAGS. Il valore del registro ebp, inoltre, viene modificato in modo tale che esso punti al valore originale di eax.

In seguito, update_tracker recupera nello stack il valore di ritorno della funzione chiamante, seguendo lo standard specificato in [36]. Questo valore viene utilizzato come chiave per l'accesso alla tabella degli indirizzi. Dal momento che essa è strutturata come una una tabella hash, il recupero della riga associata avviene in tempo $O(1)$.

A questo punto, dalla riga selezionata nella tabella, tramite il campo flags è possibile distinguere se si tratta di un'istruzione di tipo mov o di tipo stringa.

In caso di istruzioni di tipo stringa (cioè movs e stos, nel nostro scenario), il codice eseguito è riportato in figura 2.8.

```
  movsbl insn_table+4(%edx), %esi
  imul -4(%ebp), %esi
  mov -28(%ebp), %edi
  pushfw
  popw %bx
  bt $10, %bx
  jnc .DF0
  sub %esi, %edi
.DF0:
  jmp .CallDymelor
```

Figura 2.8: update_tracker per le istruzioni di tipo stringa

Il registro %edx, all'inizio dell'esecuzione di questo frammento di codice, contiene lo spiazzamento in byte all'interno della tabella per accedere alla riga associata all'istruzione. Lo spiazzamento aggiuntivo di 4 byte permette di accedere al campo size. In questo modo è possibile conoscere la dimensione atomica riguardante l'istruzione di tipo mov (richiamo quanto detto nel paragrafo 1.1.2 a pagina 6).

La seconda istruzione recupera dallo stack il valore originale di %ecx. Questo registro, come detto in precedenza, contiene il numero di iterazioni che coinvolgeranno l'istruzione di tipo stringa. Moltiplicando la taglia dell'operazione atomica per il numero di iterazioni si otterrà la dimensione totale della scrittura.

In seguito viene recuperato dallo stack il valore originale di %edi. Come descritto precedentemente, questo registro contiene già l'indirizzo di destinazione iniziale della scrittura in memoria effettuata da un'istruzione stos o movs. Ciò che occorre fare, è determinare se la scrittura procede in avanti o all'indietro, a partire dall'indirizzo contenuto in %edi.

Il parametro che discrimina la direzione della scrittura è il flag DF (Direction Flag) all'interno del registro EFLAGS[3]. Poiché, però, EFLAGS non è direttamente accessibile, i suoi 16 bit meno significativi vengono inseriti nello stack (pushfw) ed in seguito il dato affiorante dallo stack viene copiato nel registro %bx (popw %bx).

A questo punto, se il decimo bit vale 1, viene sottratta all'indirizzo di base calcolato la taglia della scrittura, calcolando così un nuovo indirizzo di base. Ciò è corretto, dal momento che lo scopo che mi sono prefissato è identificare le aree di memoria interessate da scrittura, non l'ordine con cui vengono effettuate le scritture all'interno di esse.

[3] Il flag DF è memorizzato nel decimo bit meno significativo del registro.

Al termine della procedura, il controllo viene passato alla sezione CallDymelor. Essa si occupa di interfacciarsi con il gestore della memoria della piattaforma di calcolo distribuito che verrà presentata in seguito. Parlerò nel dettaglio del sistema di interfaccia nel paragrafo 3.4.2.

Qualora, invece, dal campo flags si determini che l'operazione di scrittura è di tipo mov, il codice eseguito è quello riportato in figura 2.9.

```
  xor %edi, %edi
  testb $4, %al
  jz .NoIndex
  movsbl insn_table+10(%edx), %ecx
  negl %ecx
  movl (%ebp, %ecx, 4), %edi
  movsbl insn_table+11(%edx),%ecx
  imul %ecx, %edi

.NoIndex:
  testb $2, %al
  jz .NoBase
  movsbl insn_table+9(%edx), %ecx
  negl %ecx
  addl (%ebp, %ecx, 4), %edi

.NoBase:
  add insn_table+12(%edx),%edi
  movsbl insn_table+4(%edx),%esi
```

Figura 2.9: update_tracker per le istruzioni di tipo mov

Dopo aver azzerato il registro %edi, il modulo update_tracker effettua una serie di controlli per verificare se l'indirizzo utilizzato dall'istruzione utilizza i campi indice e base.

Qualora sia presente un indice, il codice del registro precedentemente salvato nella tabella del monitor (come spiegato nel paragrafo 2.2.1 a pagina 24) viene caricato nel registro %ecx. Di questo valore viene calcolato il complemento a due, tramite l'istruzione negl. Questo valore, moltiplicato per la dimensione di un registro, viene utilizzato come spiazzamento all'interno della finestra dello stack.

Ricordando la figura 2.7 ed i codici numerici associati ai registri (presentati nel paragrafo 1.1 a pagina 4), appare evidente che in questo modo è possibile recuperare il valore del registro di indice con un'unica istruzione: movl (%ebp, %ecx, 4), %edi.

A questo punto il valore della scala viene recuperato dalla tabella del monitor e moltiplicato per il valore di indice appena calcolato.

Qualora sia presente una base, analogamente a quanto effettuato per il registro di indice, viene recuperato dallo stack il contenuto del registro originale e sommato all'indirizzo in fase di calcolo.

Per quanto riguarda lo spiazzamento, ricordo che, se non ne è presente alcuno, il campo nella riga della tabella viene impostato a 0. Ora, poiché controllare se lo spiazzamento è presente ha lo stesso costo del sommarlo all'indirizzo in fase di calcolo, esso vi viene direttamente sommato. Se non è presente, quindi, viene sommato 0, lasciando inalterato il valore dell'indirizzo calcolato.

Infine, viene caricata la dimensione della scrittura (calcolata in fase di instrumentazione statica) nel registro `%esi`.

A questo punto, sia che l'istruzione fosse di formato stringa, sia che fosse di formato `mov`, nel registro `%esi` è memorizzata la dimensione della scrittura e in `%edi` il suo indirizzo iniziale, ed il controllo viene passato ad una funzione di gestione della mappa di memoria che, come detto in precedenza, verrà descritta nel paragrafo 3.4.2.

2.4 Correzione dei salti

L'operazione di correzione dei salti *indiretti* è un'operazione più dispendiosa. L'importanza della progettazione e dell'implementazione di una tecnica di questo tipo può essere compresa consultando quanto viene detto in [32, 36]: in questi documenti viene infatti spiegato chiaramente l'utilizzo di questo tipo di salti. Essi permettono di tradurre in maniera efficiente costrutti complessi, come lo `switch-case` del linguaggio `C`. In figura 2.10 vengono messi a confronto il codice `C` di un costrutto *multiway* di questo tipo ed una classica implementazione in Assembly.

```
switch(j) {
  case 0:
    ...
  case 2:
    ...
  case 3:
    ...
  default:
    ...
}
```

(a)

```
  cmpl $3, %eax
  ja .Ldef
  jmp *.Ltab(,%eax,4)
.Ltab:
  .long .Lcase0
  .long .Ldef
  .long .Lcase2
  .long .Lcase3
```

(b)

Figura 2.10: Elementi della tabella di rilocazione

In figura 2.10(b) si nota come, per migliorare l'efficienza, venga costruita una tabella di indirizzi, in cui ciascuna riga mantiene la destinazione relativa al blocco di codice di uno dei rami `case`.
Il confronto, pertanto, viene fatto non con la variabile realmente testata (si possono costruire degli `switch` con valori arbitrari), ma con dei valori identificanti il numero di ramo del `case`. In seguito, il salto viene effettuato andando a recuperare, nella tabella, la destinazione relativa al ramo che deve essere raggiunto, tramite un'istruzione di salto *indiretto* (l'istruzione `jmp *.Ltab(,%eax,4)`).

Consentire l'utilizzo di una struttura di programmazione di questo tipo è di fondamentale importanza: l'obiettivo finale di questo progetto è di integrare questo sistema di tracciamento degli accessi in una piattaforma di calcolo di tipo ottimistico che, come verrà descritto più in dettaglio nel paragrafo 3.1, è basata su eventi. Tipicamente, per determinare quale tipo di evento si ci accinge a processare, vengono utilizzate delle strutture di *multiway branching*.

L'approccio più immediato per tentare di correggere questi salti potrebbe essere quello di correggere il valore contenuto nel registro. Questo però porte-

rebbe l'applicazione in uno stato incorretto: non è possibile prevedere, infatti, se quel valore contenuto nel registro verrà utilizzato nel seguito del codice in una maniera differente.

Quello che viene fatto, in realtà, è far modificare al modulo `branch_corrector` il codice dell'applicazione, facendo sì che il salto diventi un salto diretto, che punti alla nuova destinazione corretta a run-time.

Una scelta di questo tipo, però, si scontrerebbe con i privilegi del segmento di testo che, per motivi di sicurezza, viene reso automaticamente non scrivibile. L'opzione più semplice sarebbe quella di rendere il segmento scrivibile, ciò che potrebbe essere ottenuto con poco sforzo, ma questo comporterebbe che eventuali errori di programmazione che, generalmente, alzerebbero il segnale di `SEGFAULT`, provochino invece delle modifiche al codice, generando un comportamento imprevedibile dell'applicazione (con conseguenti grandi difficoltà di debugging).

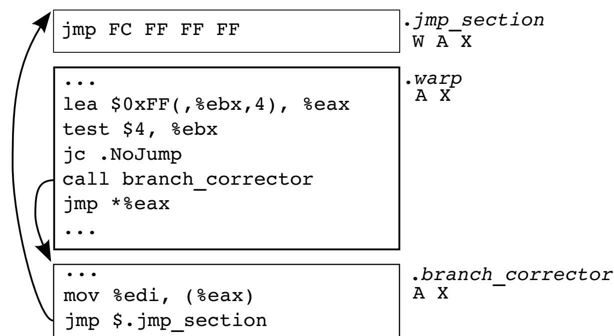


Figura 2.11: Funzionamento della correzione a run-time dei salti

Per questo motivo, nel paragrafo 2.2.5 è stato anticipato che le istruzioni di salto indiretto vengono sostituite con dei salti ad un'altra sezione di codice. Il layout della gestione dei salti, quindi, può essere riassunto dalla figura 2.11.

Si può notare che la sezione aggiuntiva cui puntano le jump sostituite ai salti *indiretti* contiene un'unica istruzione: un salto *diretto* di tipo *near*. Inoltre, questa sezione contiene del testo *eseguibile* e *scrivibile*.

Il modulo `branch_corrector`, pertanto, in maniera del tutto analoga a quanto compiuto da `update_tracker`, recupera dallo stack il valore di ritorno (che viene assemblato esattamente come in figura 2.7) e lo utilizza come chiave per recuperare nella tabella di hash le informazioni relative all'istruzione di salto originale che ha causato la sua attivazione. A questo punto il modulo calcola la destinazione del salto originale, appoggiandosi al campo `flags` per determinare se la destinazione del salto sarebbe stata in un registro oppure in memoria.

In seguito il modulo deve determinare quanto shift deve essere applicato alla destinazione originale per ottenere la destinazione corretta. Questo compito viene svolto effettuando una *ricerca binaria*[4] sulla tabella degli shift, alla ricerca

[4] La *ricerca binaria* eseguita è, in realtà, una versione leggermente modificata della ricerca binaria classica: dando per scontato, infatti, che almeno uno shift è presente (se così non fosse, non sarebbe mai avvenuta alcuna chiamata al modulo `branch_corrector`), la condizione di controllo di verificare se il limite superiore della ricerca è maggiore del limite inferiore viene modificata: se, infatti, il limite superiore viene a coincidere con il limite inferiore, si è trovato

del più grande tra i minori degli indirizzi.

Una volta individuato lo shift da applicare, `branch_corrector` lo somma alla destinazione originale del salto. In seguito la nuova destinazione corretta viene convertita in un offset a 32 bit che viene direttamente scritto nella sezione contenente il nuovo salto.

In seguito `branch_corrector` ripristina lo stato del processore e termina, restituendo il controllo all'applicazione. La prima istruzione eseguita sarà il salto alla sezione appena aggiornata dalla quale il controllo di esecuzione passerà alla destinazione corretta del salto originale.

esattamente il più grande tra i minori.

Capitolo 3

Integrazione con un ambiente per il calcolo ottimistico

In questo capitolo discuterò di come il sistema di tracciamento degli accessi su memoria dinamica descritto nel capitolo 2 sia stato applicato al salvataggio incrementale degli stati in una piattaforma di calcolo ottimistico.

Nel paragrafo 3.1 descriverò brevemente quali siano le caratteristiche dei sistemi di calcolo parallelo e distribuito che sono utilizzati in questo progetto. Nel paragrafo 3.2 descriverò più nel dettaglio le caratteristiche generali della piattaforma all'interno della quale ho integrato il sistema di tracciamento. Nel paragrafo 3.3 traccerò una breve panoramica sul sottosistema di gestione della memoria dinamica della piattaforma sulla quale ho operato. Nel paragrafo 3.4 descriverò nel dettaglio quali modifiche siano state apportate al sottosistema di gestione della memoria di riferimento per far sì che potesse essere effettuato il salvataggio incrementale degli stati. Infine, nel paragrafo 3.5 presenterò alcune misure ed alcuni risultati riguardanti le prestazioni del nuovo sistema presentato.

3.1 Cenni sulla simulazione parallela e distribuita

L'indagine nel campo della simulazione parallela e distribuita ha inizio nel 1979 con l'articolo di Chandy e Misra in [7]. Il concetto di PDES (Parallel Discrete Event Simulation), descritto per la prima volta in [12], nasce come evoluzione del precedente DES (Discrete Event Simulation). Si trattava di un paradigma distribuito per l'esecuzione di modelli simulativi.

Con il termine *simulazione* si intende un modello logico-aritmetico capace di imitare una caratteristica di un qualunque sistema fisico presente in natura, schematizzabile attraverso algoritmi e/o formule matematiche. Ad ogni simulazione è associato uno *stato globale* che rappresenta la totalità delle informazioni applicative gestite, e un insieme di *eventi* continui o discreti generati casualmente, che fanno sì che lo stato subisca delle modifiche.

Una simulazione è detta *discreta* quando le operazioni previste dagli eventi occorrono istantaneamente ed hanno una durata impulsiva. Nella parte conclusiva di questo progetto mi concentrerò unicamente su questo tipo di simulazioni.

L'idea del PDES, la cui architettura è mostrata in figura 3.1, è quella di concretizzare un programma di simulazione installato su calcolatori paralleli (remoti o locali) e basato sul processamento di eventi discreti. Ciascun evento può causare modifiche più o meno complesse allo stato globale o parziale della simulazione. Essi sono messi in correlazione temporale da un tempo logico discreto chiamato *timestamp* (o Logical Virtual Time, LVT). Questa correlazione fa sì che sia possibile una forma di sincronizzazione e coordinazione tra tutti i processi partecipanti alla simulazione, così da poter raggiungere un risultato finale comune e corretto.

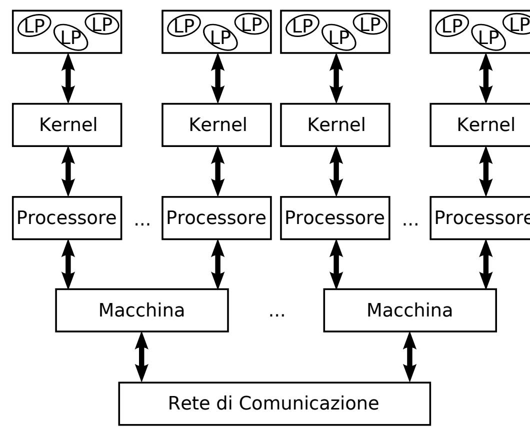


Figura 3.1: Architettura di un sistema PDES

Da un punto di vista modellistico la simulazione può essere vista come una collezione di N oggetti detti *logical processes* (LP), a ciascuno dei quali viene associato uno stato, denominato S_i, contenente un sottoinsieme di variabili $S_i \subseteq S$ strettamente necessario all'evoluzione della singola istanza della simulazione. L'insieme S corrisponde alla totalità degli stati e quindi raccoglie tutte le informazioni rilevanti ai fini della simulazione.

Lo scopo di un LP è di cooperare con gli altri in diversi contesti, mediante lo scambio di messaggi di formato predefinito, al fine di raggiungere il risultato di elaborazione desiderato. Ciascun messaggio contiene al suo interno le informazioni necessarie alla gestione di un evento associato ad un particolare timestamp. La gestione delle operazioni di scambio di messaggi e di sincronizzazione, così come l'esecuzione di tutte quelle procedure volte a garantire la correttezza della simulazione, sono affidate ad un *kernel di simulazione*, su cui si appoggia il programma di *livello applicativo*, che contiene le reali specifiche del modello da simulare.

Il kernel di simulazione si occupa anche di determinare l'ordine con cui ciascun LP dovrà processare gli eventi in attesa (generati dallo stesso LP o ricevuti da un altro). La strategia più usuale è quella di selezionare l'evento E_{min} con timestamp T_{min}. La ragione di queste scelte dipende dal fatto che, qualora venisse eseguito prima un evento E_x con timestamp $T_x > T_{min}$, esso potrebbe modificare lo stato dell'oggetto di simulazione, producendo un effetto che andrebbe ad alterare il processamento dell'evento E_{min}.

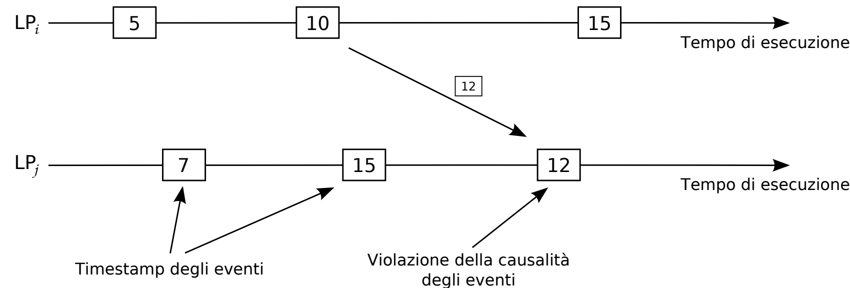


Figura 3.2: Violazione della causalità

Qualora un LP ricevesse, in alcuni contesti, un messaggio contenente un evento con timestamp precedente all'LVT corrente, come si può vedere in figura 3.2, occorrerà adottare tecniche stringenti per ripristinare uno stato precedente ed assicurare l'esecuzione secondo l'ordine corretto.

3.1.1 Strategia di sincronizzazione ottimistica

Un approccio di *sincronizzazione di tipo ottimistico*, rispetto ad un approccio di *sincronizzazione di tipo conservativo*[1], sceglie un evento E_i unicamente tra quelli locali, senza preoccuparsi dello stato della simulazione degli altri LP.

L'esempio più significativo di simulazione ottimistica è dato dal meccanismo Time Warp presentato in [18]: processa comunque gli eventi disponibili, è in grado di rivelare gli errori, interrompere il flusso degli eventi ed effettuare il ripristino di uno stato perfettamente coerente dal quale ripartire, tenendo in considerazione le nuove informazioni sull'evento che ha causato questa interruzione. Con questo approccio viene sfruttato appieno il concetto di parallelismo, poiché non vengono effettuate operazioni di controllo sugli altri LP per verificare se un evento sia safe o unsafe.

In caso di un errore (ossia se si riceve uno *straggler*[2]) è necessario avviare una procedura di *recovery* attraverso la quale gli effetti di tutti gli eventi eseguiti in modo prematuro vengano cancellati. Completata quest'operazione, si procede con una fase denominata *rollback*: in essa viene eseguito il riprocessamento di tutti gli eventi (compreso quello che ha generato l'errore) rispettando l'ordine temporale. Il numero di eventi da riprocessare prende il nome di *distanza di rollback*.

Poiché l'esecuzione incorretta di questi eventi può aver causato l'invio di messaggi ad altri LP, è prevista una tecnica di invio di *anti-messaggi* che forzano la cancellazione (eventualmente causando rollback) degli eventi negli altri LP. In figura 3.3 viene presentato un possibile scenario di questo tipo.

[1] Un approccio di sincronizzazione di tipo conservativo stabilisce che, prima di processare un evento, è necessario accertarsi se esso sia *safe* o *unsafe*. Per ottenere questo risultato è necessario adottare tecniche di previsioni tali che consentano di determinare se è possibile ricevere un evento con timestamp minore di quello T_{min} locale. È inoltre necessario adottare tecniche per evitare il *deadlock*.

[2] Un *messaggio straggler* è un messaggio contenente un evento il cui timestamp è precedente all'LVT corrente dell'LP coinvolto.

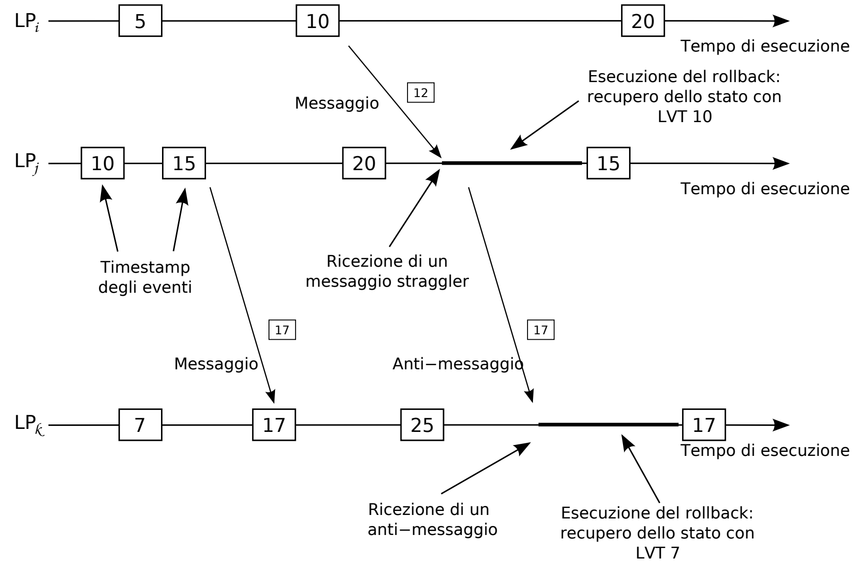


Figura 3.3: Esempio di rollback

In [18] si suggerisce l'uso di un tempo logico comune a tutti gli LP, denominato Global Virtual Time (GVT). Esso corrisponde ad un valore di virtual time che costituisce un *lower bound* di ogni futuro rollback e viene calcolato mediante un protocollo distribuito valutando il minore tra i timestamp degli eventi in attesa su tutti gli LP sparsi sulla rete. Le operazioni eseguite ad un tempo logico inferiore al GVT possono essere considerate *committed*, ovvero concluse ed impossibili da disfare.

3.1.2 Salvataggio degli stati

Per effettuare rollback è necessario ripristinare lo stato degli oggetti di simulazione relativo ad un timestamp precedente. In [6] viene proposta una tecnica di *reverse computation*, che consiste nell'invertire il flusso dell'esecuzione del software di livello applicativo (in modo automatico o semiautomatico) al fine di ripristinare una configurazione precedente.

Tuttavia, la tecnica di *state saving* (correlata a quella dello *state restoring*) è giudicata più matura ed evoluta. Le versioni più significative di questo approccio sono riportate in [18] e [17].

Gli aspetti riguardanti il ripristino di uno stato precedente (*modalità* di checkpointing, *periodo* di checkpointing, *risorse impiegate*, tecniche di *cancellazione dei log* non più necessari) non devono coinvolgere in alcun modo il programmatore dell'applicazione: il suo codice non deve contenere alcun riferimento a routine che trattino di salvataggio e ripristino degli stati[3].

Di seguito viene proposta una breve panoramica sulle tecniche di salvataggio

[3]Tuttavia, come si vedrà nel capitolo 4, nella letteratura sono stati proposti alcuni modelli di programmazione che richiedono allo sviluppatore di fornire informazioni su come effettuare il salvataggio incrementale degli stati.

degli stati che, nel corso degli ultimi anni, sono state proposte dalla letteratura sul calcolo ottimistico.

3.1.3 Copy State Saving (CSS)

La tecnica più semplice di salvataggio degli stati, presentata per la prima volta in [18], è quella del Copy State Saving (CSS). Si tratta di effettuare una copia completa dello stato [4] di un LP ogni volta che lo scheduler degli eventi determina quale dovrà essere l'evento successivo da eseguire.

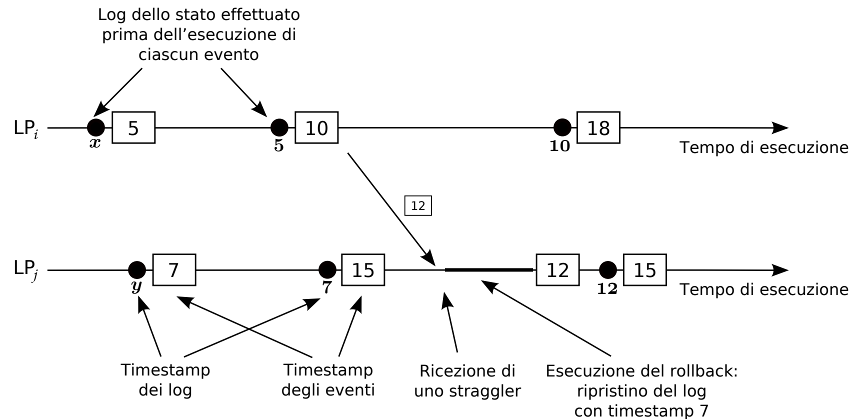


Figura 3.4: Esempio di rollback con CSS

I log vengono marcati con il timestamp relativo all'evento processato immediatamente prima della cattura dello snapshot. In questo modo, qualora sia necessario effettuare un rollback a causa di uno straggler, si potrà determinare con facilità da quale stato occorrerà far ripartire la simulazione (basterà ripristinare il log con l'LVT associato immediatamente minore del timestamp dello straggler).

Poiché, come mostrato in figura 3.4, si avrà un log per ciascun evento, non è necessario riprocessare alcun evento per allinearsi al timestamp dello straggler[5].

Questa tecnica di checkpointing presenta alcuni svantaggi, tra cui l'ingente dispendio di risorse in termini di spazio (per conservare le fotografie degli stati) e di tempo (per eseguire l'operazione di log a ciascun evento).

Dal momento, quindi, che vi è un ingente consumo di memoria, assume un ruolo fondamentale l'operazione di *fossil collection* (cfr. [17]). La fossil collection è quell'operazione che si occupa di eliminare in modo definitivo i log più vecchi, che vengono giudicati inutili per future operazioni di rollback[6].

3.1.4 Sparse State Saving (SSS)

Per tentare di migliorare la tecnica del CSS, sono state proposte alcune varianti chiamate Sparse State Saving (SSS). L'idea alla base di queste modalità di

[4] Tipicamente il sistema, insieme allo stato, effettua una copia anche dei metadati necessari ad un successivo ripristino.

[5] L'operazione di riallineamento, qualora necessaria, prende il nome di *coasting forward*.

[6] L'operazione di *fossil collection* è strettamente legata al concetto di calcolo del GVT espresso nel paragrafo 3.1.1.

checkpointing è quella di fare delle fotografie degli stati in maniera *sparse*, ossia in maniera non sistematica rispetto al processamento di ciascun evento, ma selettivamente in diversi istanti temporali, con periodo costante (PSS, Periodic State Saving) o variabile (ASS, Adaptive State Saving).

Periodic State Saving

La tecnica del *periodic state saving*, detta anche *state skipping*, viene introdotta per la prima volta in [20] ma è stata analizzata più nel dettaglio soltanto in [3]. Questa tecnica tenta di limitare l'overhead dello state saving salvando lo stato di un LP con frequenza minore rispetto al numero di eventi processati, mantenendo dei log relativi ad alcuni eventi $E' \subseteq E$.

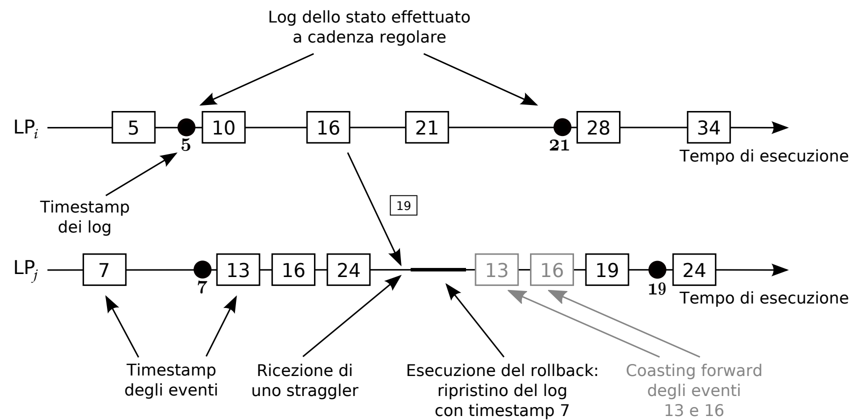


Figura 3.5: Esempio di rollback con SSS

Poiché non si ha più a disposizione un log per ciascun evento, potrebbe verificarsi il caso, mostrato in figura 3.5, della ricezione di uno straggler al tempo T_s non corrispondente ad un istante di checkpoint. Il sistema ricerca allora nella coda degli stati il log più recente e tuttavia inferiore a T_s. Dopo il ripristino di questo stato si effettua il coasting forward (o *state rebuilding*) che consiste in una veloce riesecuzione di eventi in precedenza già processati (ma di cui si era persa ogni nozione) in *modalità silenziosa*[7].

Nell'esecuzione del coasting forward è necessario garantire che l'esecuzione degli eventi segua la stessa traiettoria seguita in precedenza: in presenza delle stesse condizioni di input e dello stesso stato, il riprocessamento di un evento deve fornire lo stesso output e generare le stesse interazioni con l'ambiente esterno. Questo comportamento viene detto Piece-Wise-Deterministic (PWD) ed è stato descritto in [9]. Esso è necessario per la corretta ricostruzione dello stato di un LP.

Questa tecnica ha il significativo vantaggio di ridurre il consumo della memoria di lavoro, causando però un aumento di overhead relativo all'operazione di rollback, dal momento che è necessario eseguire il coasting forward. Diventa, quindi, di cruciale importanza determinare un intervallo di checkpointing χ adeguato: se χ è troppo piccolo, si rischia di avere un utilizzo poco efficiente

[7] Per *modalità silenziosa* si intende un riprocessamento di eventi che esuli dall'inivio dei messaggi agli altri LP, dal momento che essi sono già stati inviati in precedenza.

delle risorse di memoria, se è troppo elevato si rischia di indurre un'operazione di coasting forward troppo costosa.

Adaptive State Saving

Lo studio in [22] analizza un periodo di checkpointing adattivo per mezzo di un modello basato sul tempo di esecuzione di un LP. Supponendo che durante l'esecuzione di un evento non possa verificarsi *preemption* dovuta a rollback, né alcun invio di messaggi, e supponendo che le lunghezze dei rollback siano indipendenti tra loro, si può individuare l'intervallo di checkpointing ottimale come:

$$\chi_{opt} = \left\lceil \sqrt{\frac{2\delta_s}{\delta_c} + \left(\frac{N}{k_r} + \gamma - 1\right)} \right\rceil$$

dove δ_s e δ_c sono i tempi impiegati in media dal sistema per eseguire lo state saving ed il coasting forward, N è il numero totale di eventi committed, k_r il numero di rollback compiuti e γ la lunghezza media di un rollback.

Analogamente e sotto le stesse precondizioni, in [28] si propone di osservare durante un periodo T_{obs} il numero di rollback k_{obs} e di eventi eseguiti R_{obs} (sia *committed* che *rolled back*), e di generare una successione numerica di intervalli di checkpoint χ_n, il cui primo elemento (ed i parziali successivi) è dato da:

$$\chi_n = \left\lceil \sqrt{2\frac{R_{obs}}{k_{obs}}\frac{\delta_s}{\delta_c}} \right\rceil$$

I seguenti valori saranno calcolati secondo lo pseudocodice:

if $n = 0$ **then** $\chi_n \leftarrow \chi_{init}$
else if $k_{obs} = 0$ **then** $\chi_n \leftarrow \lceil (1-\rho)\chi_{n-1} + \rho\chi_{max} \rceil$
else $\chi_n \leftarrow max(1, \lceil (1-\rho)\chi_{n-1} + \rho min(\chi_{min}, \chi_{max}) \rceil)$

dove il valore $\rho \in (0, 1)$ determina se si sta dando più peso allo storico di χ_n piuttosto che alle misurazioni correnti; χ_{min} e χ_{max} definiscono dei limiti alle variazioni del periodo.

3.1.5 Incremental State Saving (ISS)

Per ovviare al problema della gestione della memoria, in uno scenario in cui siano presenti un alto numero di LP con degli stati grandi, ed evitare quindi il trashing delle risorse (con un conseguente calo di performance dell'intero sistema) si può adottare un approccio incrementale (ISS), per cui il sistema, invece di salvare l'intero stato di ogni oggetto, salva solamente le aree modificate dall'ultimo restore.

Poiché questo progetto verte proprio alla modellizzazione di un approccio di questo tipo, analizzerò più nel dettaglio alcune di queste proposte nel capitolo 4.

3.2 ROOT-Sim

ROOT-Sim è una piattaforma open source di tipo PDES. La sua architettura è mostrata in figura 3.6.

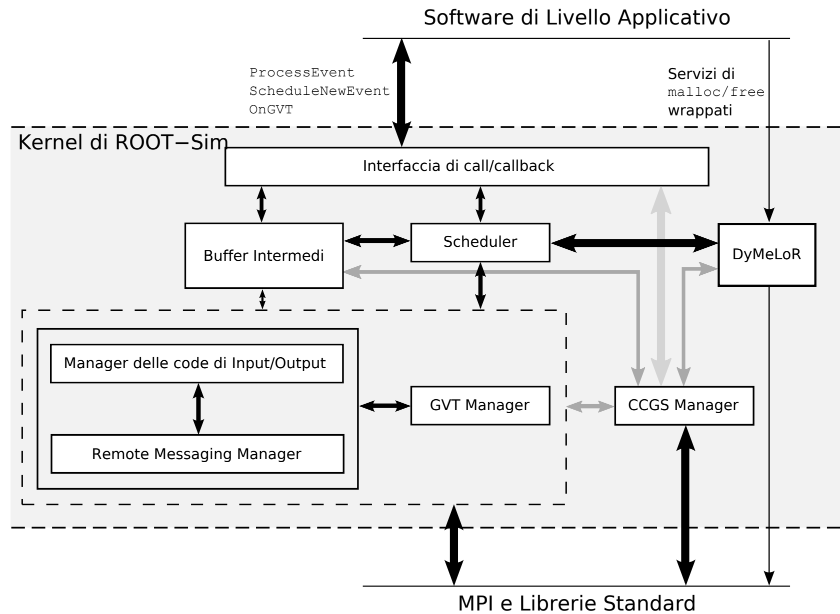


Figura 3.6: Schema dell'architettura di ROOT-Sim

Come si può vedere, essa prevede una stratificazione su più livelli, ognuno dei quali ha il compito di svolgere dei compiti particolari:

- **livello applicativo**: è lo strato al cui interno risiede il programma di simulazione creato dall'utente;
- **livello del kernel di simulazione**: è uno strato intermedio all'interno del quale risiede la piattaforma di simulazione, composta da tutti i suoi sottosistemi;
- **livello MPI**: è il livello costituito dalla libreria MPI (Message Passing Interface), utilizzata dalla piattaforma per rendere distribuita la simulazione, permettendo lo scambio di messaggi tra i vari LP.

Ciascun livello espone ai livelli superiori (tramite alcune API ad-hoc) delle funzionalità e, al tempo stesso, usufruisce delle funzionalità dei livelli inferiori. Per rendere estremamente semplice l'aggancio reciproco dei livelli, nonostante la complessità delle operazioni svolte da ciascuno di essi, le API esposte sono estremamente semplici.

Di seguito presenterò brevemente le funzionalità offerte ed i servizi di cui si avvale ciascun livello.

3.2.1 Livello applicativo

Il livello applicativo è quello strato software all'interno del quale si colloca l'applicazione di simulazione ideata dal programmatore, concepita per riprodurre un comportamento simulato di un qualche sistema realmente esistente.

Esso comunica con il livello del kernel attraverso l'utilizzo di tre funzioni:

- `ScheduleNewEvent()`: permette di comunicare al kernel di simulazione un nuovo evento appena generato. Questo evento dovrà essere spedito dal kernel al suo corretto destinatario che si occuperà di processarlo secondo il suo scheduling;
- `ProcessEvent()`: permette al kernel della piattaforma di consegnare a ciascun LP il prossimo evento da eseguire. L'ordine di causalità viene pertanto gestito dal kernel della piattaforma. Il processamento dell'evento viene discriminato in base alla tipologia cui esso appartiene, sfruttando del codice definito dallo stesso programmatore, basato su un costrutto di *multiway branching* descritto nel paragrafo 2.4;
- `OnGVT()`: tramite questa funzione la piattaforma comunica all'applicazione che è stato raggiunto uno stato *commited* per tutti gli LP, pertanto è possibile verificare, tramite l'analisi di alcuni predicati, se la simulazione ha raggiunto la sua configurazione finale di terminazione.

3.2.2 Livello del kernel di simulazione

Il livello del kernel è il nucleo centrale dell'architettura, in cui risiede la singola istanza della piattaforma di simulazione. Dal momento che la piattaforma è sia parallela che distribuita, l'ambiente di simulazione è ripartito su un certo numero di kernel, ciascuno dei quali ospita al proprio interno un certo numero di LP, assegnato in modo esclusivo ed equilibrato, per bilanciare il carico di lavoro.

Uno di questi kernel assume il ruolo di *master*: esso svolge il ruolo di coordinatore tra i kernel (gli altri sono chiamati *slave*) in tutte quelle operazioni che richiedono che qualcuno prenda una decisione.

Ciascuna istanza del kernel è organizzata in una serie di sottosistemi, ciascuno dei quali si preoccupa di svolgere determinate mansioni indispensabili all'esecuzione di una corretta simulazione. Questi moduli sono interallacciati tra loro tramite alcune opportune interfacce interne, non visibili al livello dell'applicazione. Di seguito descriverò le loro funzionalità.

Sottosistema di gestione degli eventi

Il *sottosistema di gestione degli eventi*, che prende il nome di `queue_mgnt`, si preoccupa del controllo degli eventi associati a ciascun LP e dei metadati ad essi associati. Gli eventi sono organizzati in una serie di code doppiamente concatenate, all'interno delle quali vengono rispettati i vincoli di causalità.

A ciascun evento, infatti, sono associati due valori di *tempo logico* che ne definiscono l'istante in cui esso è stato generato e l'istante in cui deve essere processato. Il sottosistema `queue_mgnt` si occupa dell'aggiornamento e della gestione degli accessi alle code, rispettandone l'ordine temporale[8] stabilito dai timestamp degli eventi.

Qualora il gestore degli eventi riceva un messaggio straggler, esso viene segnalato al sistema, comportando così il verificarsi di un rollback.

[8]Gli eventi ricevuti da un LP sono contenuti all'interno di messaggi che non arrivano necessariamente in ordine di timestamp, dal momento che la rete sottostante non garantisce alcun ordinamento.

Sottosistema per il GVT

Il *sottosistema per il GVT*, denominato `gvt_mgnt`, si occupa di calcolare periodicamente il valore del Global Virtual Time, secondo un protocollo master-slave distribuito, tra tutti i kernel di simulazione dell'intera piattaforma.

Periodicamente il kernel master invia a tutti gli slave un messaggio di notifica. Questo fa sì che gli slave, in risposta, inviino un valore di timestamp pari al minimo locale tra i timestamp degli eventi in attesa di processamento per tutti gli LP. Il master, una volta ricevute tutte le risposte, calcola il minore tra questi valori (il GVT) e lo comunica nuovamente alle istanze del kernel, che si preoccuperanno di adottarlo.

Alla ricezione del nuovo GVT, inoltre, tutti i kernel di simulazione potranno effettuare una potatura delle strutture dati: tutte quelle etichettate con un timestamp inferiore al valore del GVT verranno considerate obsolete e rilasciate.

Sottosistema per la gestione degli stati

Il *sottosistema per la gestione degli stati* (che prende il nome di `state_mgnt`) si occupa di gestire le operazioni di salvataggio e di ripristino (in caso di rollback) degli stati di ciascun LP, preservando la correttezza della simulazione.

La tecnica di checkpointing di ROOT-Sim è quella dello *sparse state saving* con periodo di log prefissato (non adattivo).

Sottosistema per lo scheduling

Il *sottosistema per lo scheduling* si occupa della gestione della sequenzialità degli eventi di ciascun LP. Stabilisce l'ordine con cui si dovranno processare gli eventi, così da mantenere il più possibile inalterata la loro causalità, nel rispetto dell'ordine temporale.

Lo scheduler si occupa di selezionare l'LP (tra quelli gestiti dal kernel) che deve effettuare un passo di simulazione. Gli LP vengono selezionati con un algoritmo STF (Shortest Timestamp First): il gestore valuta qual è l'LP cui fa riferimento l'evento con il timestamp più basso e lo attiva.

Sottosistema per lo scambio dei messaggi

La generazione di un evento provoca l'invio di un messaggio all'LP destinatario (che potrebbe coincidere con il mittente). In un tale contesto è necessario fornire un sistema di spedizione e ricezione di messaggi che coinvolga coppie di LP.

Tutti gli eventi generati dall'applicazione vengono inseriti in un buffer circolare inizialmente vuoto. Successivamente il sottosistema per lo scambio dei messaggi analizza il buffer e consegna ai destinatari i messaggi.

Quando il kernel riceve in maniera asincrona i messaggi, li inserisce nelle code tramite il gestore degli eventi.

3.2.3 Livello MPI

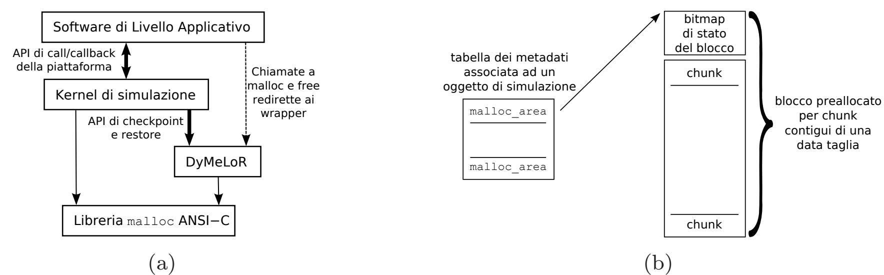
Il kernel di simulazione di ROOT-Sim si appoggia, per lo scambio dei messaggi, alla libreria MPI. Essa viene utilizzata per la realizzazione di un ambiente di lavoro distribuito.

L'implementazione della logica MPI utilizzata in questa piattaforma è fornita dalla libreria open source OpenMPI.

3.3 DyMeLoR

La libreria DyMeLoR, già integrata in ROOT-Sim come gestore della memoria dinamica, può essere considerata, da un punto di vista architetturale, come un wrapper dei servizi `malloc`/`free` dell'ANSI-C che viene frapposto, in maniera del tutto trasparente al programmatore, tramite delle semplici direttive a tempo di linking, tra il codice di livello applicativo e la libreria `malloc` tradizionale. Una schematizzazione di questo approccio è rappresentata in figura 3.7(a).

Come precedentemente mostrato in figura 3.6, DyMeLoR offre un'API per l'integrazione con il kernel di simulazione, che consiste in un insieme di servizi che supportano operazioni di gestione della memoria orientate specificatamente al salvataggio ed al ripristino degli stati.



(a) (b)

Figura 3.7: Architettura di DyMeLoR

DyMeLoR mantiene, per ciascun oggetto ospitato dal kernel di simulazione, una tabella di metadati di entry di `malloc_area`, come mostrato in figura 3.7(b). Ciascun elemento della tabella mantiene informazioni riguardanti un blocco di chunk contigui in memoria (come, ad esempio, la posizione in memoria del blocco), eventualmente allocata per servire richieste di memoria per quell'oggetto. Elementi differenti sono utilizzati per gestire chunk di dimensioni differenti.

Nel momento in cui viene ricevuta una richiesta per un chunk di una determinata dimensione, il blocco corrispondente viene allocato da DyMeLoR tramite una chiamata al vero servizio `malloc`. In pratica, in questo modo viene preallocato un certo numero contiguo di blocchi della stessa taglia, pronti per servire richieste future.

Questa preallocazione permette a DyMeLoR di utilizzare metadati molto concisi per l'identificazione dello stato di ciascun chunk (se *occupato* o *libero*) all'interno di un blocco. In particolare, viene utilizzata una bitmap di cosiddetti *status bit*. Per ottimizzare ancora di più l'utilizzo di memoria, essa viene posta in cima al blocco dei chunk, e viene allocata solamente qualora il blocco in questione venga realmente allocato.

Qualora la tabella delle `malloc_area` arrivi a saturazione, essa può essere espansa nel caso in cui l'oggetto di simulazione faccia richiesta di nuovi chunk.

Le operazioni di log e restore in DyMeLoR sono eseguite con semplici tecniche di packing ed unpacking dei dati. In un'operazione di log i chunk correntemente in uso vengono impacchettati in un buffer contiguo (allocato dinamicamente tramite la sottostante `malloc`), insieme alle entry di `malloc_area` attive e le bitmap di stato.

In un'operazione di restore, le strutture dati di un log vengono estratte dal

buffer contiguo e rimesse al loro posto. Per far sì che le operazioni di deallocazione possano essere reversibili, la `malloc_area` mantiene anche le informazioni relative al tempo logico (se disponibili) in cui i chunk all'interno di un dato blocco siano stati tutti rilasciati. Un blocco con tutti i chunk non allocati ed il cui ultimo rilascio sia avvenuto prima del GVT può essere deallocato tramite una chiamata a `free` verso la libreria `malloc` sottostante. In quel caso, la corrispondente `malloc_area` viene impostata a non attiva.

L'allocazione dei chunk all'interno di ciascun blocco è simile all'algoritmo di Linux per la selezione del prossimo file descriptor da assegnare, quando si apre un canale di I/O. Questo algoritmo mantiene bassa la frammentazione e tende a mantenere raggruppati i chunk assegnati in cima al blocco di memoria, con una conseguente diminuzione della latenza delle operazioni di log, causata da un'esaltata località.

3.4 Di-DyMeLoR

Come si può intuire, Di-DyMeLoR (Dirty Dynamic Memory Logger and Restorer) è un'evoluzione di DyMeLoR basata sull'intercettazione delle scritture tramite il sistema di instrumentazione che ho proposto in questo lavoro.

Tramite l'utilizzo di nuove strutture dati e moduli, questa libreria è in grado di utilizzare le informazioni relative alle operazioni sulla memoria dell'applicazione per tracciare le attività di aggiornamento. In questo modo è possibile migliorare l'efficienza del sistema, tramite l'utilizzo di un approccio incrementale.

In questo paragrafo descriverò le modifiche effettuate alle strutture dati ed ai moduli per l'integrazione con il sistema di tracciamento degli aggiornamenti. In seguito entrerò nei dettagli delle operazioni di log e restore svolte con approccio incrementale.

3.4.1 Gestore della memoria dinamica

In Di-DyMeLoR le strutture dati originali per la gestione della mappa di memoria sono state espanse per potersi occupare in maniera esplicita della costruzione di log degli stati completi a partire dal salvataggio incrementale di quelle sole aree che sono state sporcate dall'ultima operazione di log.

Per tracciare i chunk sporcati è stata associata a ciascun blocco di chunk una seconda bitmap di cosiddetti *dirty bit*. Come si può osservare in figura 3.8, la bitmap è situata all'interno dello stesso segmento di memoria puntato dalla `malloc_area` corrispondente, che contiene anche l'originale use bitmap. Questa bitmap eredita le stesse caratteristiche della bitmap di stato originale.

Pertanto, l'occupazione aggiuntiva di memoria necessaria a determinare quali chunk sono stati sporcati dall'ultima operazione di log è ben scalata rispetto alla dimensione della memoria assegnata all'applicazione.

Per le operazioni di log e restore è necessario anche tracciare quali tra i metadati hanno subito delle modifiche: per questo motivo all'interno della struttura `malloc_area` sono stati aggiunti i seguenti campi di tipo intero (come si può vedere dalla figura 3.8):

- `dirty_area`: viene utilizzato come flag per indicare se è avvenuta un'operazione di qualsiasi tipo (tra allocazione, deallocazione, o scrittura di un chunk) all'interno dell'area, dall'ultima operazione di log;

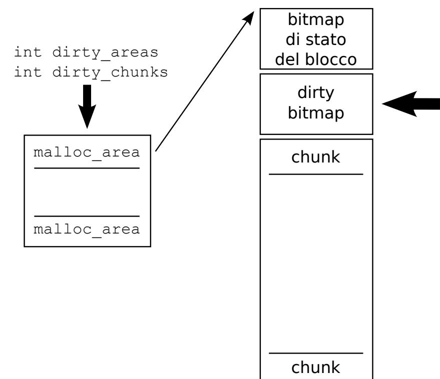


Figura 3.8: Strutture dati di Di-DyMeLoR

- `dirty_chunks`: tiene il conto del numero di chunk in uso che sono stati sporcati, all'interno dell'area, dall'ultima operazione di log.

Conformemente al modello di DyMeLoR originale, nel caso in cui l'indirizzo interessato da una scrittura risieda all'esterno della mappa di memoria dell'oggetto di simulazione correntemente in esecuzione (ad esempio, riferimenti a variabili globali esterne all'heap), il gestore della memoria restituisce il controllo.

3.4.2 Tracciamento delle scritture

Come accennato nel paragrafo 2.3, l'interfaccia tra il modulo `update_tracker` ed il gestore della memoria avviene tramite un'API esposta dalla libreria Di-DyMeLoR: `dirty_mem()`.

Tramite questa funzione il modulo di tracciamento degli aggiornamenti alla memoria notifica al gestore della mappa di memoria quale area (in termini di indirizzo di base e dimensione) è stata interessata da una scrittura. Il gestore della mappa, quindi, andrà a scandire la tabella delle `malloc_area` dell'LP correntemente in esecuzione. Qualora l'indirizzo notificato alla funzione `dirty_mem` sia afferente ad una qualche area dell'LP corrente, il gestore della mappa di memoria calcola quale (o quali) chunk contengono la regione aggiornata. A questo punto, i bit della dirty bit relativi ai chunk interessati vengono impostati ad 1.

Qualora l'indirizzo notificato non appartenga a nessuna area della mappa di memoria dell'LP corrente, il modulo `dirty_mem` non modifica alcuna delle strutture dati.

3.4.3 Operazioni di salvataggio degli stati

Le attività di salvataggio degli stati di Di-DyMeLoR sono state differenziate tra *log completi* e *log incrementali*. Entrambi i tipi di log consistono in una serie di operazioni di impacchettamento di informazioni all'interno di un buffer contiguo in memoria, allocato tramite una chiamata a `malloc`. Tuttavia vengono impacchettate informazioni differenti.

Un'operazione di **log completo** coincide con l'operazione di log originariamente supportata da DyMeLoR. In essa, pertanto, non vengono salvate le dirty

bitmap. L'unica differenza risiede proprio nella loro gestione: completato il log, infatti, le dirty bitmap vengono azzerate.

Un'operazione di **log incrementale** effettua, invece, operazioni di *packing* differenti a seconda del valore corrente delle strutture dati. Per ciascuna `malloc_area` attiva, si possono verificare i seguenti casi:

A: `dirty_area` vale 1 e `dirty_chunks` vale 0. In questo caso la `malloc_area` viene impacchettata nel buffer di log insieme alla bitmap di stato, indicando così l'allocazione dei chunk all'interno di un dato blocco. La dirty bitmap e i chunk, tuttavia, non vengono salvati;

B: `dirty_area` vale 1 e `dirty_chunks` è maggiore di 0. In questo caso la `malloc_area` viene impacchettata all'interno del buffer di log, insieme a tutti chunk correntemente assegnati che risultano essere stati sporcati. Tutti gli altri chunk in uso non vengono salvati;

C: `dirty_area` vale 0. In questo caso, non viene memorizzata alcuna informazione circa la `malloc_area`.

Come nel caso dei log completi, i log incrementali causano un reset completo di tutte le strutture dati necessarie al tracciamento delle modifiche. Questo avviene indipendentemente da quale dei tre casi appena descritti si verifichi.

Voglio sottolineare, infine, che le operazioni di salvataggio incrementale degli stati non richiedono assolutamente di essere eseguite prima del processamento di ogni evento. Esse infatti si basano sul riconoscimento di porzioni di memoria sporcate dall'ultimo log, indipendentemente dal numero di eventi che causano le modifiche alla memoria. La ricostruzione dello stato si adatta quindi perfettamente sia alla modalità CSS che a quella SSS, descritte nei paragrafi 3.1.4 e 3.1.3

3.4.4 Operazioni di ripristino degli stati

In modo simile a DyMeLoR, ciascun log viene etichettato con il tempo di simulazione corrente e tutti i log (sia completi, sia incrementali) vengono collegati tra loro in una catena.

Quando si presenta la necessità di eseguire un'operazione di ripristino verso un tempo di simulazione T, viene effettuata una ricerca all'interno della catena per determinare il log più recente con tempo minore o uguale di T.

Qualora il log trovato sia completo, viene eseguita un'operazione di ripristino analoga a quella compiuta originariamente da DyMeLoR. Qualora il log incontrato, invece, sia di tipo incrementale, entra in gioco un algoritmo differente. In particolare, vengono iterati i seguenti passi, all'indietro attraverso la catena di log, partendo da quello identificato come stato da ripristinare:

1. una `malloc_area` trovata all'interno di un buffer di log, che non sia ancora stata ripristinata, viene rimessa a posto all'interno della tabella dei metadati. Inoltre, viene ripristinata dal buffer anche anche la bitmap di stato[9];

2. ciascun chunk all'interno del buffer di log (associato alla `malloc_area` corrente) che non sia ancora stato ripristinato in un'iterazione precedente viene ricopiato al suo posto nel suo blocco di memoria.

[9] Si ricordi che, indipendentemente dal tipo di log e dal caso specifico di log incrementale, una `malloc_area` viene sempre associata alla sua bitmap di stato, per garantire la possibilità di ripristino di operazioni di allocazione e deallocazione.

La procedura iterativa di ripristino si ferma quando tutte le `malloc_area` sono state ripristinate insieme a tutti i chunk assegnati. Anche se, in principio, questo potrebbe richiedere un numero non definito di passi iterativi all'indietro lungo la catena di log, in pratica l'operazione di restore può essere finalizzata nel momento in cui viene incontrato un log completo durante l'attraversamento della catena. Infatti tutti i chunk in uso che non sono ancora stati ripristinati diventano immediatamente disponibili per la copia nel log completo.

Per ottimizzare l'individuazione dei chunk non ancora ripristinati, la procedura iterativa di ripristino si appoggia a delle bitmap temporanee (una per ciascuna `malloc_area`) su cui vengono eseguite una coppia di veloci operazioni bit a bit di tipo OR-XOR, ogni volta che viene estratta da un buffer di log una dirty bitmap. Questo procedimento è illustrato in figura 3.9.

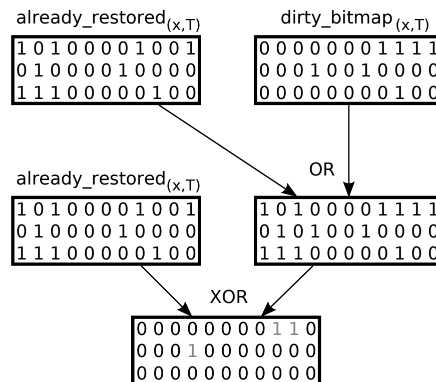


Figura 3.9: Operazione OR-XOR sulle bitmap

La bitmap temporanea $already_restored_{(x,T)}$ è quella associata ad una `malloc_area` x e ad un log con timestamp T e mantiene le informazioni relative a quali chunk sono già stati ripristinati. La bitmap $dirty_bitmap_{(x,T)}$ è invece una di quelle estratte da un buffer di log marcato con timestamp T ed afferente ad una `malloc_area` di indice x. Come si può vedere, l'operazione descritta in figura permette di identificare con due sole veloci operazioni quali siano i soli chunk presenti nel log in fase di processamento che dovranno essere ripristinati.

3.4.5 Caching dei riferimenti

In DyMeLoR sono stati volutamente evitati degli header per i chunk, per evitare di dover effettuare log e restore di metadati troppo grandi. Pertanto, quando un chunk viene rilasciato, non è possibile utilizzare alcuna struttura dati per accedere in maniera rapida alla `malloc_area` coinvolta nell'operazione.

Per velocizzare queste operazioni DyMeLoR forniva un sottosistema di *direct-map caching* a livello software, implementato come una tabella hash, con le righe della tabella formate dalla coppia:

$$< chunk_start_address, m_area_index >$$

La questione di identificare la `malloc_area` a partire da un indirizzo di memoria diventa ancora più critica nella nuova versione Di-DyMeLoR: il gestore della mappa di memoria, infatti, deve recuperare la `malloc_area` per aggiornarne i metadati ogni volta che riceve una segnalazione da `update_tracker`, che è un evento molto più frequente di quello di una deallocazione.

Inoltre, in Di-DyMeLoR c'è bisogno di recuperare l'area corretta a partire da un indirizzo di memoria che non coincide necessariamente con il bordo superiore del chunk (come avviene invece per le operazioni di `free`), poiché `update_tracker` potrebbe catturare un'operazione di scrittura relativa ad una locazione di memoria posta a metà di un chunk. Per affrontare questa situazione la cache è stata estesa, portando le linee ad avere una forma rappresentata dalla tripla:

$$< chunk_start_address, chunk_end_address, m_area_index >$$

La cache diventa pertanto in grado di accettare un input multiset. L'indirizzo iniziale di una scrittura intercettato da `update_tracker` viene privato di n bit meno significativi, dove n è scelto in modo tale da far collidere tutti gli indirizzi di uno stesso chunk nella stessa riga. In realtà, posto che la dimensione dei chunk forniti all'applicazione possa essere differente[10], n è stato scelto come valor medio tra il numero di bit necessari a far collidere i chunk più piccoli e quelli più grandi (di quelli gestiti da Di-DyMeLoR), moltiplicato per un fattore di scala teso a polarizzare il valore verso i chunk più piccoli[11].

3.5 Dati Sperimentali

La piattaforma hardware utilizzata nello studio sperimentale è una macchina Quad-Core dotata di quattro processori a 64-bit Intel da 2.4-GHz con 4MB di cache e 4GB di memoria RAM, con installato Linux (versione del kernel 2.6.22). Ciascun core ospita una delle istanze del kernel di simulazione ottimistica.

La frequenza di calcolo del GVT (e quindi le relative operazioni di recupero della memoria) sono state impostate in modo tale che l'utilizzo di RAM non superi mai il 60/70% della memoria totale, così da non incappare in fenomeni di swapping che altererebbero l'attendibilità delle misure riportate.

L'applicazione di prova è un simulatore di sistemi cellulari parametrizzabile che modella esplicitamente i fenomeni di attenuazione e di interferenza dei canali [19]. Ciascuna istanza di oggetto di simulazione modella una singola cella e traccia, con l'utilizzo di strutture dati allocate dinamicamente, l'assegnazione dei canali ed alcune informazioni sulla gestione della potenza, per le chiamate in uscita.

In particolare, all'avvio di una chiamata destinata ad una periferica cellulare attualmente ospitata nella cella, l'oggetto di simulazione genera un nuovo record di chiamata tramite una coppia di strutture dati allocate dinamicamente e lo collega ai record già attivi. Ciascuno di essi verrà rilasciato qualora la chiamata termini o sia trasferita (*hand-off*) verso un'altra cella (in questo secondo caso, una nuova procedura di istanziazione del record di chiamata verrà eseguita nella cella di destinazione).

[10] Ricordo che, così come avviene per la libreria `malloc`, Di-DyMeLoR gestisce dimensioni dei chunk che siano potenze di 2, con una dimensione massima parametrizzabile, generalmente impostata a 32KB.

[11] I chunk di dimensione medio-piccola sono, statisticamente, quelli più richiesti da un software di livello applicativo.

Quando viene instaurata una chiamata, viene eseguita la regolazione della potenza, che comprende la scansione della lista di record sovraccitata per calcolare la minore potenza di trasmissione che permetta alla chiamata che sta per essere avviata di raggiungere il livello di soglia SIR, così come descritto dalla tecnologia GSM.

Le strutture dati che tengono traccia dei coefficienti di attenuazione vengono anch'esse aggiornate mentre si scandisce la lista.

Sono state simulate delle macro celle, ciascuna delle quali gestisce fino a 1000 canali wireless, utilizzando impostazioni classiche come la distribuzione esponenziale del tempo di interarrivo delle chiamate ed una durata media della chiamata di 2 minuti (guardare, per esempio, [4]). Inoltre, la frequenza di interarrivo verso ciascuna cella è stata fatta variare nell'intervallo tra 1 e 6.25 chiamate per tempo unitario di simulazione, inducendo così valori del fattore di utilizzo del canale sempre maggiori (tra il 12% ed il 75%), e facendo aumentare la lunghezza attesa della lista di record in uso di cui ho parlato precedentemente.

In questo modo si ottiene un doppio effetto:

1. la memoria richiesta per lo stato di ciascun oggetto di simulazione varia tra i 4KB ed i 32KB (meta-dati esclusi);
2. la granularità degli eventi cresce, da più grossolana a più fine.

Queste variazioni mi hanno consentito di valutare alcuni effetti delle capacità innovative fornite da Di-DyMeLoR con configurazioni differenti. Per quanto riguarda l'instrumentazione del software, in figura 3.10 ho misurato i valori dei parametri descritti più avanti, con misure ottenute da una configurazione di benchmark di piccola dimensione, formata da quattro oggetti di simulazione (ciascuno ospitato da una singola istanza del kernel di simulazione, in esecuzione sulla macchina Quad-Core):

(A) La latenza media di esecuzione di un evento di simulazione;

(B) La latenza media di un'operazione di log;

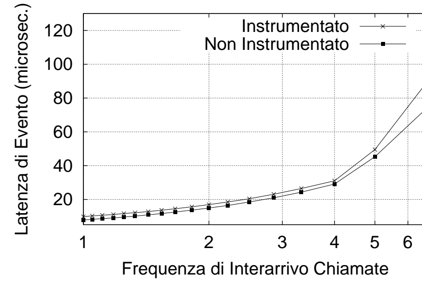
(C) La latenza media per ripristinare la mappa di memoria ad uno stato salvato;

(D) La dimensione media di un log salvato.

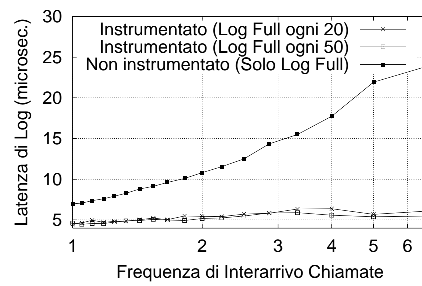
Come detto nel paragrafo 3.4.4, la latenza di un'operazione di ripristino di uno stato in Di-DyMeLoR dipende dall'interleaving tra i log completi ed i log incrementali, lungo la catena di log. Pertanto, per quanto riguarda i parametri in (B), (C) e (D), i grafici si riferiscono a differenti passi di interleaving tra log completi ed incrementali, ossia log completi presi ogni 20 e 50 operazioni di log, rispettivamente.

Dai risultati si nota che l'overhead, indotto dal meccanismo di tracciamento degli aggiornamenti della memoria sulla latenza di esecuzione degli eventi, è estremamente limitato. Inoltre, i requisiti di memoria per ciascuna operazione di log, nel caso in cui sia stata effettuata l'instrumentazione, sono decisamente più bassi rispetto a quelli osservati nel software non instrumentato.

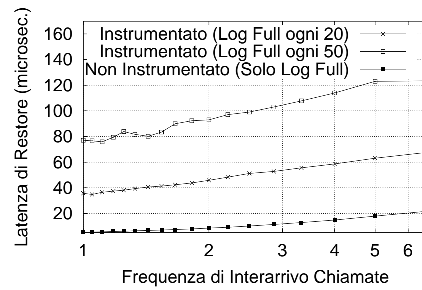
La configurazione senza codice instrumentato fornisce un netto guadagno per le operazioni di restore degli stati. In ogni caso, dai grafici si può dedurre che la diminuzione di rendimento nel caso del software instrumentato può essere controllata (mantenendo, al contempo, i vantaggi dal lato del logging) con una selezione particolare di un passo di interleaving non eccessivamente grande tra i log completi ed incrementali.



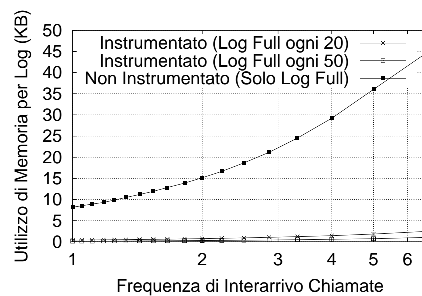
(a)



(b)



(c)



(d)

Figura 3.10: Statistiche di base per l'applicazione di test

Capitolo 4

Lavori Collegati

In questo capitolo discuterò alcuni lavori presentati più o meno recentemente sugli stessi ambiti, evidenziandone chiaramente similitudini e differenze.

Mi concentrerò su due ambiti più strettamente connessi con le idee e le tecniche utilizzate: nel paragrafo 4.1 discuterò di approcci simili nell'ambito del debugging e del vulnerability assessment, mentre nel paragrafo 4.2 tratterò più specificatamente dell'ambito delle operazioni di log e restore all'interno delle piattaforme di calcolo ottimistico.

4.1 Debug e Vulnerability Assessment

La metodologia di Vulnerability Assessment (VA) è un processo volto a testare il livello di sicurezza dei sistemi.

Nell'ambito più specificatamente informatico, la metodologia di VA cerca di individuare tecniche e pratiche per l'identificazione più o meno automatizzata di difetti all'interno delle applicazioni. La sempre maggiore crescita della complessità delle applicazioni, tuttavia, pone nuove sfide, tra cui la presenza di dead end ed il verificarsi di falsi positivi che possono portare via del tempo allo sviluppo stesso del software.

Classici esempi di controlli di Vulnerability Assessment sono la ricerca di funzioni che effettuano copie in memoria senza limiti (ad esempio le funzioni `strcpy()` o `strcat()`), di errori di tipo *off-by-one*, del possibile verificarsi di overflow o underflow, di errori sull'aritmetica dei puntatori.

Il progetto che ho sviluppato ha relazioni con un certo numero di lavori nel campo del tracciamento dell'esecuzione dei programmi (ad esempio [13, 1, 24, 39]) con lo scopo di effettuare debugging e vulnerability assessment.

Volendo fare un confronto con il mio lavoro, molti degli approcci lì proposti forniscono un'analisi dettagliata dei cambiamenti nello stato del programma e del flusso di esecuzione. In ogni modo, quest'analisi viene ottenuta tramite tecniche che degradano le prestazioni basate su instrumentazione dinamica e/o servizi del kernel, che sono decisamente inadatti in alcuni contesti (come ad esempio le simulazioni in parallelo) dove il rendimento non può essere sacrificato.

Di nuovo, per quanto riguarda il supporto al debugging dei programmi, l'unico lavoro che mostra una modalità operativa confrontabile con la mia (cioè l'utilizzo di meccanismi di trap basati sull'inserimento/sostituzione di codice per identificare gli accessi in scrittura sulla memoria) sono quelli che si occupano dei watchpoint (come ad esempio [37]). In ogni caso, questo lavoro ha degli obiettivi

di performance differenti dai miei: le ottimizzazioni si preoccupano principalmente delle tecniche di ricerca per verificare se un riferimento in memoria cade o meno all'interno di una regione che è correntemente soggetta ad un watchpoint.

In altre parole, tutti quegli aspetti connessi con l'identificazione di aree che sono state sporcate e le problematiche relative al salvataggio ed al ripristino degli stati non vengono considerate.

4.2 Calcolo Parallelo

Nel contesto di simulazioni ottimistiche sono state introdotte alcune soluzioni per salvare l'intero stato di un oggetto di simulazione (al momento dell'esecuzione di un evento o dopo un certo intervallo di eventi eseguiti) [10, 23, 25, 26, 28], o per salvare incrementalmente le porzioni di stato modificate [5, 29, 34, 38], o per supportare un misto dei due approcci [11, 21].

Queste soluzioni hanno la necessità di (i) fornire il codice necessario per catturare gli snapshot dello stato degli oggetti all'interno del software di livello applicativo o di (ii) impiegare chiamate a funzioni tra le API di apposite librerie di checkpointing oppure di (iii) identificare staticamente (ad esempio, a tempo di compilazione) quali porzioni dello spazio di indirizzamento debbano essere considerate parte dello stato.

Di conseguenza, non viene supportata una trasparenza perfetta, dal momento che il programmatore deve necessariamente scontrarsi con le questioni legate agli snapshot degli stati. Inoltre l'identificazione statica delle locazioni di memoria da includere negli snapshot non è compatibile con l'allocazione e la deallocazione dinamica della memoria (ad esempio tramite librerie standard) a livello degli oggetti di simulazione.

È questo il caso del lavoro in [38] che ha alcune somiglianze con il mio sul piano dell'instrumentazione automatica, ma che non permette l'utilizzo di memoria allocata dinamicamente.

Confrontata con questi altri approcci, la mia soluzione supporta la gestione degli stati, basata su capacità di log incrementale, senza la necessità di moduli specifici per il log e per il restore all'interno del codice dell'applicazione, né di un'interfaccia esplicita con delle librerie di log e restore. Inoltre permette di distribuire gli stati degli oggetti di simulazione su dei frammenti di memoria allocata dinamicamente, senza requisiti stringenti sulla loro dimensione.

La questione degli stati basati su memoria dinamica per gli oggetti di simulazione ottimistica è stata anche affrontata dai framework in [33, 8]. In ogni modo, in essi vengono adoperate delle API apposite per notificare esplicitamente al kernel di simulazione che delle operazioni specifiche di allocazione o deallocazione e, più in generale, operazioni su strutture dati basate su memoria dinamica, dovranno essere ripristinabili. Pertanto, differentemente dall'approccio da me adottato, non sono supportati layout di memoria basati su servizi di allocazione e deallocazione propri dell'ANSI-C.

In termini di capacità del sottosistema di gestione della memoria, i lavori più vicini al mio approccio sono probabilmente quelli in [30, 31], che presentano dei livelli software per effettuare in maniera trasparente operazioni di log e restore nelle simulazioni ottimistiche basate sullo standard di interoperabilità High-Level-Architecture (HLA).

Questi livelli si appoggiano a meccanismi di protezione della memoria propri dei Sistemi Operativi (e in particolare ai servizi offerti dalla *system call* `UNIX` `mprotect()`) per identificare gli aggiornamenti in memoria ed effettuare il log incrementale delle pagine sporcate che siano appartenenti ad un layout di memoria degli LP. Il kernel della piattaforma protegge da scrittura tutte le pagine che contengono gli stati degli oggetti di simulazione, così da poter intercettare i segnali di errore alzati dal kernel del Sistema Operativo ogni volta che l'applicazione effettua un'operazione di scrittura. In questo modo, ad ogni intercettazione, il kernel della piattaforma si occuperà di eseguire una copia dell'intera pagina, abilitarne la scrittura per consentire l'aggiornamento dello stato, ed infine disabilitarne nuovamenta la scrittura.

Se confrontato con il mio approccio, l'overhead per tracciare gli aggiornamenti ed effettuare le operazioni di log incrementale è verosimilmente maggiore[1] ed è conveniente soltanto quando sia comparabile con il costo dei servizi di interoperabilità supportati dal middleware di HLA.

Tutto ciò rende questi approcci inadatti alle tradizionali piattaforme di simulazione ottimistica, che hanno requisiti di efficienza estremamente stringenti, e che sono l'obiettivo di questo lavoro.

Alcuni risultati recenti [2, 6, 21] hanno mostrato la fattibilità e l'efficienza della gestione ottimistica degli stati tramite una computazione inversa: una versione inversa del codice di simulazione di livello applicativo viene generata (automaticamente o semiautomaticamente, tramite un'analisi statica) ed utilizzata per il calcolo all'indietro, mirato al ripristino dello stato dell'oggetto di simulazione.

Questa tecnica, comunque, è più adatta per applicazioni a granularità di eventi fine, dal momento che il codice inverso è generalmente più efficiente quando a ciascun evento è associata una computazione breve. Il mio approccio, invece, tende a concentrarsi su eventi a granularità arbitraria.

Inoltre in contesti generali di simulazione (ad esempio laddove si possono verificare dei percorsi d'esecuzione non invertibili, come l'assegnazione distruttiva di variabili o la generazione di numeri pseudocasuali), questo approccio deve essere necessariamente affiancato da tecniche di log e restore simili a quelle che ho presentato in questo lavoro.

[1] I continui passaggi da *user-mode* a *kernel-mode* e viceversa, necessari per cambiare i privilegi associati alle pagine e causati dai tentativi di modifiche a pagine protette da scrittura, diventano estremamente frequenti durante la simulazione, ed essi sono operazioni intrinsecamente molto costose.

Capitolo 5

Conclusioni

In questo lavoro ho presentato una metodologia per il tracciamento trasparente degli accessi su memoria dinamica, l'implementazione di questa metodologia e ho descritto come ho integrato questi moduli software all'interno di un layer open source preesistente, che consente operazioni di log e restore trasparenti per oggetti di simulazione ottimistica con layout degli stati basati su servizi standard di allocazione e deallocazione di memoria dinamica.

Il tracciamento avviene in maniera trasparente al programmatore, tramite direttive a tempo di compilazione e linking e modifica diretta sui file object di formato ELF, con strumenti pensati per operare su architetture di tipo *Intel-compliant*.

Le routine che si preoccupano dell'intercettazione delle scritture e della correzione dei salti a run-time sono state scritte in maniera tale da minimizzare il più possibile l'overhead aggiunto.

Il sistema di tracciamento è stato integrato in DyMeLoR, un sottosistema di gestione di memoria allocata dinamicamente, con sostanziali modifiche alla struttura interna di questo modulo. Questa integrazione ha consentito di estendere le capacità di ROOT-Sim, una piattaforma di calcolo ottimistico, permettendo l'adozione di operazioni di salvataggio e ripristino degli stati incrementali, mediante l'individuazione a tempo di esecuzione delle variazioni relative alla mappa di memoria degli oggetti di simulazione.

La completa trasparenza del processo di instrumentazione del codice garantisce al programmatore del software di livello applicativo la possibilità di continuare a scrivere utilizzando tutti gli strumenti del linguaggio ANSI-C, senza doversi preoccupare in alcun modo della gestione degli stati, essendo ciò completamente gestito (autonomamente) dalla piattaforma.

Ho inoltre presentato alcuni risultati sperimentali, per una valutazione dei benefici ottenibili tramite l'approccio proposto che rivelano come, in contesti di forte carico, i vantaggi apportati dalla mia metodologia siano realmente significativi.

Bibliografia

[1] V. Bala, E. Duesterwald, e S. Banerjia. DYNAMO: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, 2000.

[2] D. W. Bauer e E. H. Page. An approach for incorporating rollback through perfectly reversible computation in a stream simulator. In *21st International Workshop on Principles of Advanced and Distributed Simulation*, pp. 171–178. IEEE Computer Society, 2007.

[3] S. Bellenot. State skipping performance with the Time Warp operating system. In *Proceedings of the 6 th Workshop on Parallel and Distributed Simulation (PADS '92*, pp. 53–64, 1992.

[4] A. Boukerche, S. K. Das, A. Fabbri, e O. Yildz. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 166–173. IEEE Computer Society, maggio 1999.

[5] D. Bruce. The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 40–49. IEEE Computer Society, giugno 1995.

[6] C. D. Carothers, K. S. Perumalla, e R. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, luglio 1999.

[7] K.M. Chandy e J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, Sept. 1979.

[8] S. Das, R. Fujimoto, K. Panesar, D. Allison, e M. Hybinette. GTW: a time warp system for shared memory multiprocessors. In *Proceedings of the 26th conference on Winter simulation*, pp. 1332–1339. Society for Computer Simulation International, 1994.

[9] M. Elnozahy, L. Alvisi, Y. Wang, e D. Johnson. A survey of rollback-recovery protocols in message-passing systems. Relazione tecnica, ACM Computing Surveys, 1996.

[10] J. Fleischmann e P.A. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 50–58. IEEE Computer Society, giugno 1995.

[11] S. Franks, F. Gomes, B. Unger, e J. Cleary. State saving for interactive optimistic simulation. In *Proceedings of the 11th workshop on Parallel and Distributed Simulation*, pp. 72–79. IEEE Computer Society, 1997.

[12] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, ottobre 1990.

[13] GDB: The GNU Project Debugger. *http://www.gnu.org/software/gdb/*.

[14] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual Volume 1: Basic Architecture*.

[15] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*.

[16] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*.

[17] D. Jefferson. Virtual Time II: storage management in conservative and optimistic systems. In *Proceedings of the 9th annual ACM symposium on Principles of distributed computing*, pp. 75–89. ACM, 1990.

[18] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, luglio 1985.

[19] S. Kandukuri e S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[20] Y. B. Lin e E. D. Lazowska. *Reducing the saving overhead for Time Warp parallel simulation*. University of Washington Department of Computer Science and Engineering, febbraio 1990.

[21] A. Naborskyy e R. M. Fujimoto. Using reversible computation techniques in a parallel optimistic simulation of a multi-processor computing system. In *21st International Workshop on Principles of Advanced and Distributed Simulation*, pp. 179–188. IEEE Computer Society, 2007.

[22] A. C. Palaniswamy e P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 127–134. IEEE Computer Society, 1993.

[23] B. R. Preiss, W. M. Loucks, e D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, luglio 1994.

[24] F. Qin, C. Wang, Z. Li, H. S. Kim, Y. Zhou, e Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pp. 135–148, 2006.

[25] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, febbraio 2001.

[26] F. Quaglia e A. Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, giugno 2003.

[27] M. Riepe. `libelf`, a library to read, modify or create ELF files in an architecture-independent way. 0.8.9-stable released on 2006-08-22.

[28] R. Ronngren e R. Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pp. 110–117. Society for Computer Simulation, luglio 1994.

[29] R. Ronngren, M. Liljenstam, R. Ayani, e J. Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 70–77. IEEE Computer Society, maggio 1996.

[30] A. Santoro e F. Quaglia. Transparent state management for optimistic synchronization in the High Level Architecture. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pp. 171–180. IEEE Computer Society, giugno 2005.

[31] A. Santoro e F. Quaglia. A version of MASM portable across different UNIX systems and different hardware architectures. In *Proceedings of the 9th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, ottobre 2005.

[32] R. A. Sayle. A superoptimizer analysis of multiway branch code generation. In *Proceedings of the GCC Developers' Summit*, pp. 103–110, 2008.

[33] SPEEDES. *http://www.speedes.com*, 2005.

[34] J. Steinman. Incremental state saving in SPEEDES using C Plus Plus. In *Proceedings of the Winter Simulation Conference*, pp. 687–696. Society for Computer Simulation, dicembre 1993.

[35] The SCO Group, Inc. *System V Application Binary Interface*, fourth edizione, March 1997.

[36] The SCO Group, Inc. *System V Application Binary Interface, Intel386 Architecture Processor Supplement*, fourth edizione, March 1997.

[37] R. Wahbe, S. Lucco, e S. L. Graham. Practical Data Breakpoints: Design and implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, 1993.

[38] D. West e K. Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 78–85. IEEE Computer Society, maggio 1996.

[39] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, e W. F. Wong. How to do a million watchpoints: Efficient debugging using Dynamic Instrumentation. In *CC*, volume 4959 di *Lecture Notes in Computer Science*, pp. 147–162. Springer, 2008.