Tecniche di instrumentazione statica per il supporto alla trasparenza verso il programmatore nelle STM

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

Candidato
Fernando Visca
1205740

Relatore
Francesco Quaglia

a.a. 2010-2011

A Monica,
che mi è sempre stata vicino
durante i miei studi.

Indice

Introduzione **vii**

1 Tecnologie coinvolte **1**
1.1 Architettura CISC IA-32 2
1.1.1 Formato delle istruzioni 2
1.1.2 Istruzioni che operano su stringhe 5
1.1.3 Modalità di indirizzamento 6
1.1.4 Istruzioni di salto 10
1.1.5 Esempi riassuntivi 12
1.2 Architettura `CISC` x86-64 15
1.2.1 Formato delle istruzioni 15
1.2.2 Modalità di indirizzamento 17
1.3 Particolari classi di istruzioni 17
1.3.1 L'unità x87 FPU 18
1.3.2 Tecnologia Intel© MMX™ 20
1.3.3 Estensioni SSE 20
1.3.4 Estensioni SSE2 22
1.4 Formato ELF 23
1.4.1 Formato del file 23
1.4.2 Header delle sezioni 24
1.4.3 Tabelle di rilocazione 25
1.4.4 Tabelle dei simboli 26
1.4.5 Tabella delle stringhe 27

2 Progetto degli strumenti di supporto **29**
2.1 Analizzatore lessicale 29
2.2 Instrumentazione 34

2.2.1 Generazione della tabella delle istruzioni 35
2.2.2 Modifica degli ELF 38
2.2.3 Modifica delle tabelle di rilocazione 38
2.2.4 Gestione dei salti 39
2.2.5 Riepilogo del processo di instrumentazione 42
2.3 Correzione dei salti 44

3 Integrazione in TL2 **49**
3.1 Introduzione ai sistemi STM 49
3.1.1 Motivazione delle STM 49
3.1.2 Sincronizzazione non-bloccante 52
3.1.3 Transazioni 53
3.1.4 Supporto hardware 54
3.1.5 Tecniche per algoritmi non-bloccanti 56
3.1.6 Un esempio di algoritmo non-bloccante 57
3.2 Transactional Locking II 58
3.2.1 Breve panoramica di TL2 59
3.3 Il modulo `transactional_rw` 62
3.4 Wrapping delle `STM_READ` e `STM_WRITE` 66

4 Conclusioni **71**

Bibliografia **75**

Introduzione

Il presente lavoro riguarda la trasparenza al programmatore nel progetto di applicazioni basate su sistemi *Software Transactional Memory* (STM). Più in dettaglio, si vuol far sì che, durante la scrittura di applicazioni basate su STM, sia possibile, *per le operazioni su memoria* (i.e., letture e scritture), utilizzare i costrutti messi a disposizione dal linguaggio di programmazione scelto *astraendo* da qualsiasi riferimento alle funzioni di lettura/ scrittura messe a disposizione dall'implementazione STM e dalla loro logica di funzionamento.

Già da tempo sul mercato dei microprocessori si è affermata la tendenza ad ottenere prodotti più tecnologici non già migliorando le prestazioni del singolo processore ma combinando più processori che eseguano *applicazioni parallele*. Ciò è dovuto anche al fatto che nella tecnologia di produzione dei circuiti integrati si sono raggiunti dei *limiti fisici* che obbligano a ricercare migliori performance esclusivamente tramite la concorrenza. Per queste ragioni, ricopriranno importanza sempre maggiore i *sistemi paralleli e distribuiti*. Lo sviluppo di applicazioni concorrenti non è semplice per il programmatore ed i paradigmi più diffusi, in primo luogo l'approccio *lock-based*, tendono ad introdurre un alto tasso di errori nel progetto. Il problema maggiore è quello dovuto agli *stati di memoria inconsistenti* i quali si verificano a causa della forte *contention* da parte dei *thread* in esecuzione per l'accesso ad aree di memoria condivise. Le STM — come sarà più volte ricordato nel corso della trattazione — offrono un'interfaccia di programmazione più intuitiva e potrebbero, in un futuro, risolvere molti di questi problemi. Per questo motivo, si è cercato di sviluppare una soluzione per la semplificazione dell'uso di questi strumenti da parte del progettista.

L'obiettivo prefisso sarà perseguito tramite l'uso di tecniche di *instrumentazione statica*. In parole povere, si analizza il file eseguibile (processo

di *parsing*) e si intercettano tutte le istruzioni che accedono in memoria sostituendole, nel flusso stesso delle istruzioni, con delle chiamate a routine che si occupino di demandare, tramite opportuni meccanismi, l'esecuzione di quelle istruzioni alle API dell'implementazione STM.

In prima istanza, saranno analizzate le tecnologie e gli strumenti coinvolti (capitolo 1). Dopo tale analisi, si discuteranno le scelte progettuali di base nonché i dettagli dell'implementazione, sia dell'instrumentatore che del parser (capitolo 2). Successivamente, si apprenderà come sono stati integrati tali moduli in una particolare implementazione STM denominata *TL2*; verrà fatta una introduzione a TL2 così come alle STM in generale (capitolo 3). Infine, si presenteranno i risultati e verranno tratte le dovute conclusioni (capitolo 4).

Il presente lavoro si inserisce all'interno dell'attività di ricerca condotta dal gruppo *HPDCS* (High Performance and Dependable Computing Systems) del Dipartimento di Informatica e Sistemistica dell'università degli studi di Roma "La Sapienza" [9].

Capitolo 1

Tecnologie coinvolte

Essendo sorta la necessità di intercettare le istruzioni di lettura/scrittura in memoria nel flusso di istruzioni macchina per la successiva fase di instrumentazione, si è scelto di modificare un parser sviluppato in precedenza presso il gruppo di ricerca HPDCS [9], il quale effettua l'analisi lessicale dell'*instruction set* delle architetture Intel© 64 e IA-32 (perciò, l'intero lavoro si basa su quella tecnologia).

Il parser, tuttavia, non "intercetta" le letture in memoria. Infatti, esso è stato sviluppato per il tracciamento delle sole scritture in memoria (*dirtying* della memoria) per il salvataggio, con prestazioni ottimali, dello stato di simulazione della piattaforma di calcolo distribuito `ROOT-Sim` (vedere [14] per approfondimenti).

Tale problema è stato risolto tramite una lieve, ma sostanziale modifica alle strutture dati: è stato introdotto un campo "flags" di tipo `integer` alla struttura dati restituita dal parser in output. Questo ha permesso non solo di supportare il tracciamento delle letture, ma ha anche trasformato il software in un analizzatore lessicale *general-purpose* dell'*instruction set* delle architetture Intel© 64 e IA-32, come sarà meglio esemplificato più in avanti.

Per concludere, si è scelto di concentrarsi sul sistema operativo GNU/Linux e sul formato di eseguibili ELF sia, come prima, per il riuso del codice del parser e dell'instrumentatore preesistenti, sia per la successiva fase di integrazione in TL2.

Di seguito verrà fatta una breve panoramica sulle architetture e sul formato degli eseguibili coinvolti, per consentire di illustrare con maggiore

dettaglio i problemi che sono sorti e giustificare le soluzioni via via adottate.[1]

1.1 Architettura CISC IA-32

Con *IA-32*, oppure a volte anche con *i386* od *x86*, si definisce l'*instruction set* dei microprocessori prodotti principalmente da *Intel* ed *AMD*. Caratteristica principale di questo *instruction set* è la lunghezza variabile.

La lunghezza variabile 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 [10]. 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*, detto anche *program counter*).

1.1.1 Formato delle istruzioni

Il formato delle istruzioni dell'IA-32 prevede vari campi, alcuni dei quali opzionali, come viene rappresentato in figura 1.1.

I **prefissi** sono suddivisi in quattro gruppi e può essere presente al più un solo prefisso per gruppo. I prefissi 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, rispettivamente, che il dato immediato o l'indirizzo in memoria avranno dimensioni differenti rispetto a quanto specificato dall'opcode.

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

[1] Le sezioni sull'architettura dei microprocessori Intel© e sul formato ELF sono state riadattate da [13].

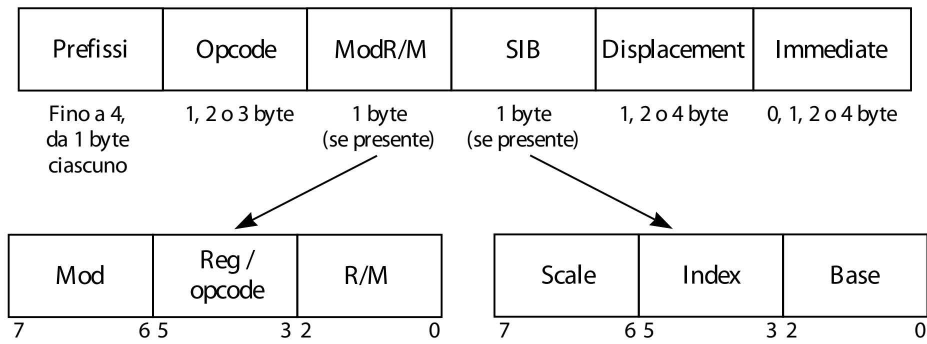


Figura 1.1: Schema del formato delle istruzioni per l'IA-32

`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)

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* (*register/ memory*) 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

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. Tali istruzioni consentono la scrittura/lettura o la copia su aree di memoria di dimensione arbitraria. Esse sono, rispettivamente, le istruzioni `stos`, `lods` o `movs`. La prima permette di scrivere un certo numero di ripetizioni del valore contenuto nel registro `AX`. La seconda, permette di leggere un certo numero di byte a partire dall'indirizzo contenuto in `DS:(E)SI` (`DS` è un registro *segment selector*). La terza, invece, consente di copiare un'area di memoria in un'altra, della stessa dimensione.

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

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.

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

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

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 [11].

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

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

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 byte 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`.

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

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

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

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 alla regola precedente 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, dunque, fa sì che istruzioni più *semplici* possano essere rappresentate con meno byte, ottenendo così realmente il risparmio di memoria di cui si è accennato all'inizio. Inoltre, è 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*.

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:

```
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

```
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.

```
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`

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*.

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

Il prefisso `REX`, come mostrato in figura 1.5, ha i primi 4 bit preimpostati a `0100`. 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;

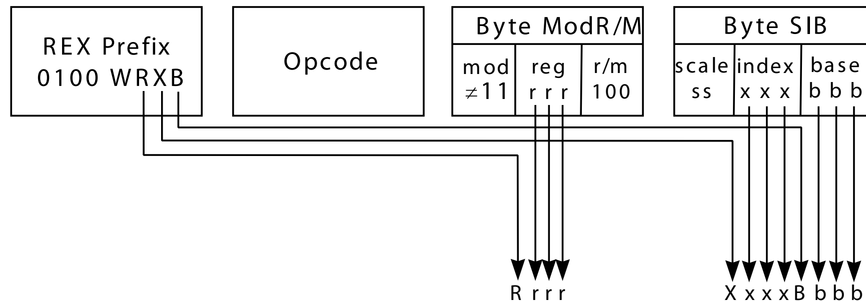


Figura 1.5: Prefisso REX

- `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.

Si noti che 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`.

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. 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.

Oltre a ciò, 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 Particolari classi di istruzioni

Nel capitolo 2 si apprenderà come sia stato modificato il parser al fine di includere il supporto all'intercettazione delle letture in memoria, oltre che delle scritture. Nel medesimo capitolo sarà spiegato come lo stesso parser sia stato incrementato al fine di renderlo uno strumento *general-purpose* e riusabile.

Una delle nuove funzionalità del parser è la sua capacità di assegnare ad ogni istruzione legale incontrata un insieme (talvolta vuoto) di *flag*. I flag utilizzati sono riassunti in tabella 2.1.

Per meglio comprendere il significato dei flag `I_SSE`, `I_SSE2`, `I_FPU`, `I_MMX` ed `I_XMM` si accennerà nel seguito alcune utili nozioni sulle componenti hardware e le classi di istruzioni dei microprocessori in esame. Si consulti [10] per approfondimenti.

1.3.1 L'unità x87 FPU

La *x87 Floating-Point Unit* (FPU), unità di calcolo autonoma che costituisce un ambiente di esecuzione separato, fornisce capacità di calcolo in virgola mobile ad alte prestazioni per applicazioni di grafica, scientifiche ed ingegneristiche.

L'unità supporta i tipi di dato floating-point, integer e packed BCD integer. Essa opera secondo gli algoritmi di processamento ed ha la stessa architettura di gestione delle eccezioni definita in [1].

Come appena detto, la x87 FPU rappresenta un ambiente di esecuzione a se stante all'interno dell'architettura IA-32 (vedi 1.6). Questo ambiente di esecuzione consiste di otto registri generici (detti *x87 FPU data registers*) e dei seguenti registri speciali:

- status resister;
- control register;
- tag word register;
- last instruction pointer register;
- last data (operand) pointer register;
- opcode register.

Le istruzioni che si avvalgono di tale unità sono contrassegnate dal parser con il flag `I_FPU`.

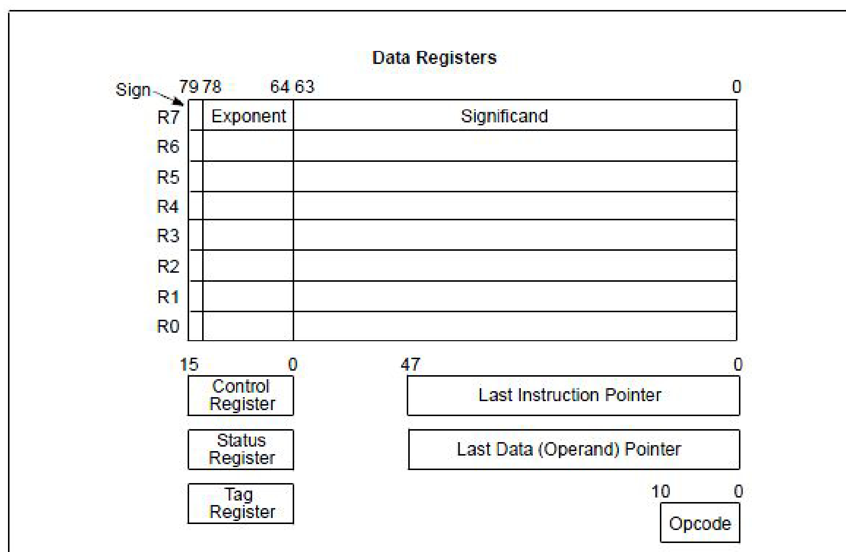


Figura 1.6: Ambiente di esecuzione della x87 FPU

1.3.2 Tecnologia Intel© MMX™

La tecnologia MMX dell'Intel fu introdotta nell'architettura IA-32 con i processori della famiglia Pentium II ed i Pentium con tecnologia MMX. L'estensione supporta il modello di esecuzione *single-instruction, multiple-data* (SIMD) progettato per migliorare le performance di applicazioni multimediali avanzate.

Il modello di esecuzione SIMD supporta dati a 64-bit di tipo *packed integer* e, sempre mantenendo la compatibilità con i processori, le applicazioni ed il codice di sistema precedenti, introduce le seguenti novità:

- Otto nuovi registri a 64 bit, detti appunto registri MMX.
- Tre nuovi tipi di dato *packed*:
  - Interi 64-bit *packed byte* (con e senza segno).
  - Interi 64-bit *packed word* (con e senza segno).
  - Interi 64-bit *packed doubleword* (con e senza segno).
- Istruzioni a supporto dei nuovi tipi di dato e per la gestione dello *stato MMX*.
- Altre estensioni di supporto.

Le istruzioni che si avvalgono di tale tecnologia e dei registri MMX sono contrassegnate dal parser con il flag `I_MMX`.

1.3.3 Estensioni SSE

Le Streaming SIMD Extensions (SSE) sono state introdotte nell'architettura IA-32 a partire dai processori della famiglia Pentium III. Queste estensioni migliorano le performance dei processori IA-32 in relazione ad applicazioni di grafica 2-D e 3-D, nonché ad altre applicazioni multimediali.

Le estensioni SSE espandono il modello di esecuzione SIMD introdotto con la tecnologia MMX apportando nuove capacità quali la gestione di valori floating-point a precisione singola, *packed* e *scalar*, contenuti in appositi registri a 128 bit.

Le estensioni SSE aggiungono le seguenti funzionalità all'architettura IA-32, pur mantenendo la compatibilità con i precedenti processori, programmi applicativi e codice di sistema:

- otto nuovi registri a 128 bit detti registri *XMM* in modalità non-64-bit e sedici registri XMM disponibili in modalità 64-bit;
- registro MXCSR di 32 bit di stato per le operazioni che sfruttano i registri XMM;
- tipo di dato floating-point packed a precisione singola da 128 bit;
- istruzioni che effettuano operazioni SIMD su dati floating-point a precisione singola e istruzioni che estendono le operazioni SIMD che possono essere effettuate su interi;
- istruzioni per il salvataggio ed il ripristino dello stato;
- istruzioni per il *prefetching* esplicito dei dati e per altre ottimizzazioni nell'uso della cache (a beneficio delle applicazioni che richiedono regolarmente l'accesso a grandi quantità di dati).

Nella figura 1.7 è riportato uno schema con le modifiche architetturali al processore per il supporto delle estensioni SSE.

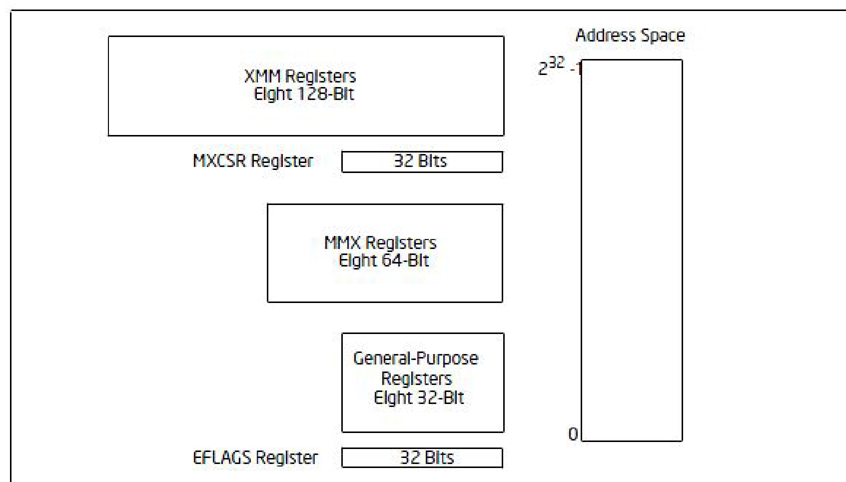


Figura 1.7: Ambiente di esecuzione Streaming SIMD Extensions (SSE)

Le istruzioni SSE vengono contrassegnate dal parser con l'attribuzione del valore di verità al flag `I_SSE`. Le istruzioni usano i registri XMM vengono, invece, contrassegnate dal flag `I_XMM`

1.3.4 Estensioni SSE2

Le Streaming SIMD Extensions 2 (SSE2) sono state introdotte nell'architettura IA-32 a partire dai processori Pentium 4 ed Intel Xeon. Queste estensioni incrementano le prestazioni dei processori IA-32 in relazione ad applicazioni di grafica tridimensionale, codifica/decodifica video, riconoscimento vocale ed altre applicazioni scientifiche ed ingegneristiche.

Le estensioni SSE2 si basano sul modello di esecuzione SIMD usato dalla tecnologia MMX e dalle estensioni SSE. Le nuove funzionalità introdotte riguardano il supporto dei valori floating-point packed a *doppia precisione* ed il supporto di interi packed a 128 bit.

Le nuove caratteristiche dell'architettura IA-32 per il supporto di tali estensioni mantengono — come solito — la compatibilità con i processori, le applicazioni ed il codice di sistema precedenti, e consistono principalmente in:

- Cinque nuovi tipi di dato:
  - Floating-point packed a doppia precisione da 128 bit.
  - Interi packed byte da 128 bit.
  - Interi packed word da 128 bit.
  - Interi packed doubleword da 128 bit.
  - Interi packed quadword da 128 bit.
  - Interi packed byte da 128 bit.
- Nuove istruzioni in supporto dei nuovi tipi di dato.
- Altre modifiche di supporto alle istruzioni IA-32.

Le modifiche strutturali necessarie al supporto delle estensioni SSE2 non differiscono da quelle per le SSE, illustrate in figura 1.7.

Le istruzioni SSE2 sono contrassegnate dal parser tramite il flag `I_SSE2`.

1.4 Formato ELF

Come sarà descritto in seguito, per poter identificare le letture e 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 letture e delle scritture.

Si presenta subito, pertanto, l'esigenza di dover modificare gli eseguibili. Di seguito, sarà presentato brevemente il formato degli oggetti ELF (Executable and Linkable Format), così come viene presentato in [17].

1.4.1 Formato del file

Un file ELF è un file che permette di rappresentare del *codice rilocabile* (generato dagli *assemblatori* o dai *linker* e 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 precedentemente caricato in memoria lo richieda). Nel caso in cui l'ELF rappresenti uno shared object caricato dinamicamente, il file assume anche il nome di *libreria*.

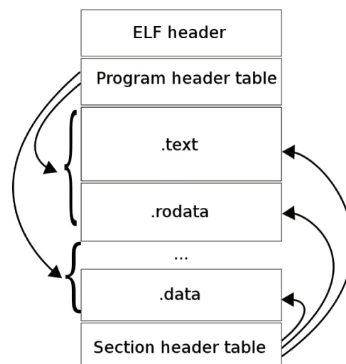


Figura 1.8: Struttura di un file ELF

Come mostrato in figura 1.8, un file ELF è suddiviso in sezioni. A seconda che descriva un oggetto rilocabile o eseguibile, alcune sezioni saranno presenti o meno.

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,...), viene definito dove si trovano le tabelle degli *header del programma* e degli *header di sezione* 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 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 saranno utilizzati unicamente oggetti rilocabili, nel seguito saranno analizzati alcuni dettagli della struttura di queste intestazioni. In più, saranno analizzate nel dettaglio alcune sezioni di interesse.

1.4.2 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 — al solito, espressa come offset dall'inizio del file.

Di interesse per questo lavoro sono tre campi dell'header: `sh_size`, `sh_offset` ed `sh_flags`. Il primo determina quale sarà la dimensione della sezione (in byte), il secondo la posizione nel file ELF dove è possibile individuarla, l'ultimo 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, la tabella di rilocazione ha lo scopo di fornire

al linker informazioni sulla posizione, nel flusso di byte del codice, dei riferimenti alle variabili, e su quali siano 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 1.9: Elementi della tabella di rilocazione

1.4.3 Tabelle di rilocazione

Le tabelle di rilocazione sono formate da una serie di elementi, la cui struttura è rappresentata in figura 1.9. In particolare, in figura 1.9(a) viene presentato un elemento della tabella di rilocazione senza addendo, mentre in figura 1.9(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.

1.4.4 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 1.10.

```
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 1.10: Elementi della tabella dei simboli

L'unico di questi campi che sia effettivamente di interesse per il lavoro descritto, 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.

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

Tabella 1.7: Utilizzo della tabella delle stringhe

1.4.5 Tabella delle stringhe

La tabella delle stringhe è organizzata come in figura 1.11. Essa è composta da una serie di sequenze di caratteri terminate dal carattere `NULL`, tipicamente chiamate *stringhe di testo*. L'intero object utilizza queste stringhe per rappresentare i nomi dei simboli ed i nomi delle sezioni.

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 1.11: Struttura della tabella delle stringhe

Una stringa può essere riferita con 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 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 sono presentati degli esempi relativi alla figura 1.11, per meglio chiarire come funziona il sistema di indicizzazione delle stringhe.

Capitolo 2

Progetto degli strumenti di supporto

In questo capitolo verranno illustrate le scelte progettuali e alcuni dettagli implementativi adottati per la realizzazione del sistema di tracciamento di accessi su memoria. In particolare, sarà descritto 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 sarà descritto operativamente come avvenga l'inserimento delle routine di tracciamento e di correzione dei salti. Infine sarà analizzato nel dettaglio come siano state realizzate le suddette routine, soffermandosi 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 12, non è possibile conoscere a priori quale sarà la lunghezza della successiva istruzione incontrata nel flusso di byte del programma. Per questo motivo si è reso necessario un parser capace di interpretare l'intero instruction set delle architetture *Intel-compliant* a 32 ed a 64 bit.

Già in precedenza, presso il gruppo di ricerca `HPDCS`, era in sviluppo un parser nato per altri obiettivi (consultare [14, 9]). Tuttavia, esso presenta un problema: non traccia le *letture* dalla memoria. Il lavoro svolto ha riguardato la modifica di questo parser per introdurre il supporto al trac-

ciamento delle letture e, al contempo, per renderlo uno strumento riusabile e incrementarne le potenzialità.

In ciò che segue verrà soltanto accennato al funzionamento del parser, argomento per cui si invita a consultare [13] per maggiori dettagli. Piuttosto, si metteranno in evidenza le differenze sostanziali tra la nuova versione del parser e quella precedente spiegando come si sono raggiunti gli scopi prefissi.

Il fulcro del parser era, ed è, una tabella: in essa sono raggruppate, in ordine di opcode (primo byte), le famiglie di istruzioni. Così come viene definito nell'appendice B di [12], tutte le istruzioni sono raggruppate in alcune famiglie. Tramite questa tabella si risale al nome dell'istruzione, al formato ed al tipo degli operandi; nel caso in cui per ottenere tali informazioni si debba invece leggere un ulteriore byte, ci sarà in tabella un puntatore ad una funzione che si occuperà di gestire quest'ultima casistica ed accederà un'altra tabella, simile alla prima, indicizzata tramite il leggendo secondo byte di opcode, e così via. . .

Rispetto alla `struct` che rappresentava la entry della tabella nel parser precedente, il nuovo parser introduce una lieve quanto fondamentale modifica, la quale si evince con chiarezza dalla figura 2.1: lì dove c'era un campo `to_be_instrumented` di tipo `bool`, ora è presente il campo `unsigned long flags`.

(a)

```
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;
};
```

(b)

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

Figura 2.1: Struttura delle righe della tabella di istruzioni (a) nella vecchia e (b) nella nuova versione del parser

Il campo `flags` svolge una funzione particolare. Lì dove il valore booleano della prima tabella indicava semplicemente se l'istruzione analizzata fosse da instrumentare o meno, questo nuovo campo è formato da una composizione in `OR` binario di valori interi predefiniti — appunto, i "flags". La presenza o meno di un flag si può testare tramite delle macro ad-hoc. I flag portano con se numerose informazioni utili sull'istruzione e, per determinare se una istruzione sia da instrumentare o meno, si può ricorrere ad una espressione logica sul valore di ritorno delle macro per il test dei flag (queste sono tutte le istruzioni che leggono e/o scrivono, nonché le `jump` per i motivi che vedremo più in avanti).

I flag che si è determinato di dover introdurre, in vista sia di applicazioni future che dell'integrazione nell'instrumentatore, sono elencati, con la rispettiva spiegazione, nella tabella 2.1. Si noterà che la presenza di tali flags ha trasformato il parser, tramite una modifica relativamente semplice, da uno strumento non riusabile e poco espressivo in un analizzatore lessicale completamente riusabile ed anche "espressivo", nel senso che è in grado di fornire preziose informazioni su diverse qualità semantiche delle istruzioni tramite i flag.

L'introduzione di questo campo `flags` ha obbligato a modificare tutte le tabelle del parser, le quali contengono una entry per ogni istruzione dell'instruction set. Per ogni entry, si è studiata semantica della relativa istruzione, consultando le schede sui manuali [11] e [12]. Una volta compresa la semantica e la tipologia di istruzione in esame, si è deciso quali flag attribuire alla stessa, tra cui i più importanti per la presente trattazione sono sicuramente i flag `I_MEMRD` ed `I_MEMWR` che indicano, rispettivamente, che l'istruzione legge e scrive in memoria.

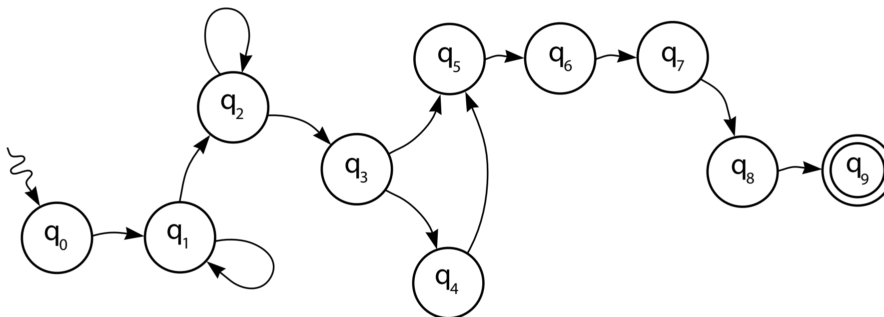
Si noti come l'introduzione del supporto alle letture ha comportato una modifica non solo delle tabelle, ma anche della logica del parser, in quanto alcune modalità di indirizzamento o tipi di operando sono di interesse solo se si considerano anche le letture e, per questo motivo, la precedente versione del parser ne ignorava la gestione. Inoltre, si sono dovute gestire diverse eccezioni (ad esempio, istruzioni che leggono dall'operando di destinazione e non dal sorgente. . .) e si è esteso lo spettro delle istruzioni riconosciute dal parser che, sottolineiamo, continua ad ignorarne un'ampia classe — nella pratica, queste non vengono mai generate dai compilatori.

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

I_ALU	L'istruzione esegue operazioni di tipo aritmetico/logiche e sfrutta la ALU del processore
I_CALLRET	Istruzione della famiglia CALL o della famiglia RET
I_CONDITIONAL	L'istruzione esegue se e solo se una certa condizione è soddisfatta (ad esempio, se e solo se un bit dello *status register* EFLAGS è true)
I_CTRL	L'istruzione svolge dei test su dati o bit (essenzialmente, sono le funzioni della famiglia TEST e CMP)
I_FPU	Istruzioni che utilizzano l'unità x87 FPU (operano su dati in virgola modile)
I_JUMP	Istruzioni della famiglia JUMP che modificano il flusso di esecuzione del programma
I_MEMRD	L'istruzione legge dalla memoria
I_MEMWR	L'istruzione scrive sulla memoria
I_MMX	Istruzioni che si avvalgono della tecnologia MMX™ e che comunque usano i registri MMX
I_PUSHPOP	Istruzioni della famiglia PUSH e POP che operano sullo stack del processo
I_SSE	Istruzioni Streaming SIMD Extensions (vedere la sezione 1.3.3)
I_SSE2	Istruzioni Streaming SIMD Extensions 2 (vedere la sezione 1.3.4)
I_STRING	L'istruzione opera su stringhe
I_XMM	Istruzioni che si avvalgono dei registri XMM

Tabella 2.1: Flag per la classificazione semantica delle istruzioni

di interesse. I passi seguiti sono rappresentati dall'*Automa a Stati Finiti* mostrato in figura 2.2.



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

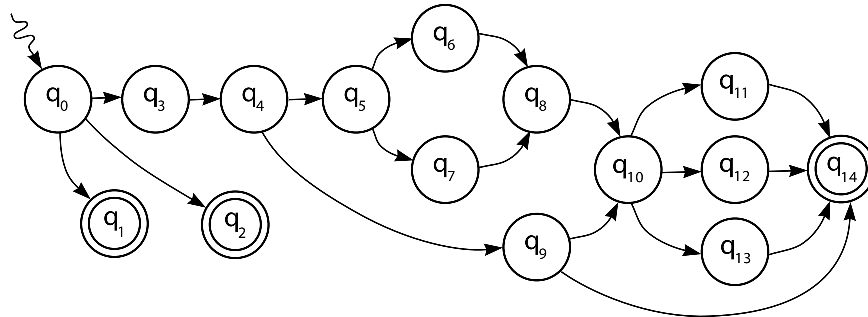
Figura 2.2: Automa a Stati Finiti del parser

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.

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, già presente nella precedente versione del parser, ha subìto lievi modifiche e adattamenti ed opera come rappresentato nell'*Automa a Stati Finiti* in

figura 2.3. La sua logica di funzionamento risulta essere valida anche per ricavare l'indirizzo di memoria nel caso di letture.



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 e letture su memoria.

Una volta che il parser ha identificato un'istruzione di interesse, il modulo di instrumentazione del codice la sostituisce con una `call` ad un modulo chiamato `transactional_rw` che si preoccupa del *wrapping* delle funzioni di lettura e scrittura di TL2. Questo modulo a sua volta richiama al suo interno una funzione C che funge da *demultiplexer di operazioni* e si occupa di richiamare, a seconda dei casi, la funzione di lettura o scrittura (o entrambe) di TL2. Questo "triplo passaggio" (routine assembly → funzione C → API di TL2) e le sue motivazioni verranno date nella sezione 3.4.

Questa modalità operativa è tipica nell'ambito del tracciamento dei riferimenti in scrittura a memoria, come si può leggere in modo approfondito in [20] ed è stata perciò adottata per il tracciamento delle scritture in memoria per il salvataggio ottimale dello stato di simulazione della piattaforma ROOT-Sim come illustrato nel lavoro [14] al quale è collegato il presente. La tecnica usata (nonché parte degli strumenti sviluppati per supportarla) continua ad essere valida per gli scopi della presente trattazione. Tuttavia, dati gli obiettivi preposti, è chiaro che si presenteranno delle richieste prestazionali più stringenti rispetto a [20]. In particolare, si vuole che il modulo `transactional_rw` esegua 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 lettura/scrittura, è possibile organizzare il processo di instrumentazione 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* `transactional_rw`) una riga formata dai campi mostrati in figura 2.4.

Questi campi mantengono le seguenti informazioni:

- `ret_addr`: il valore di ritorno della chiamata a `transactional_rw`. Corrisponde all'indirizzo in cui è situata l'istruzione di scrittura/lettura che ha causato l'invocazione del modulo di tracciamento;

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

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

- `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. Infine, viene indicato se l'istruzione effettua una lettura od una scrittura od entrambe, a seconda dei casi; I bit di questo campo, ed il loro significato, sono elencati nella tabella 2.2;

- `base`: mantiene il codice numerico (come descritto nel paragrafo 1.1 a pagina 2) 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;

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
3	Determina se è presente un registro di destinazione
4	Determina se viene scritto un dato immediato
5	Determina se l'operazione legge dalla memoria
6	Determina se l'operazione scrive sulla memoria

Tabella 2.2: Flag utilizzati dal modulo `transactional_rw`

- `displacement`: lo spiazzamento (direttamente estratto dai byte dell'istruzione);
- `reg`: il codice del registro di destinazione, nel caso in cui siamo in presenza di una istruzione di lettura che ha come destinazione un registro (il suo uso è determinato dal campo `flags`);
- `immediate`: valore del dato immediato, nel caso in cui siamo in presenza di una istruzione di scrittura che faccia uso di dati immediati (il suo uso è determinato dal campo `flags`);

Vengono utilizzati, per descrivere quali registri vengano utilizzati come base e come indice, i codici utilizzati negli opcode perché questo permette al modulo `transactional_rw` di recuperare il valore contenuto in quei registri con una sola istruzione, come verrà descritto nel dettaglio nella sezione 3.3 a pagina 65.

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, ossia il campo `unsigned long ret_addr`. 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 [15] 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 nostro scopo è quello di modificare file già precedentemente generati, ed in particolare modificarne soltanto delle parti ben precise, si è deciso di utilizzare un libreria sviluppata in precedenza per il medesimo motivo, sempre presso il gruppo HPDCS (vedere in particolare [13]).

Alla luce di quanto visto nel capitolo 1.4, gli oggetti `ELF` hanno una struttura basata su tabelle di header associate a delle sezioni. Il modulo di gestione, 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 a svolgere le operazioni di rilocazione descritte nel paragrafo 1.4.3.

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 `transactional_rw`) che implica una modifica apposita delle tabelle dei simboli e delle stringhe precedentemente 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. Si è adottato 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 nella sezione 1.1.4, non ha una controparte di tipo *near*, si è adottata una tecnica di sostituzione differente. Essa è stata rimpiazzata 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, ciò avviene a discapito delle prestazioni (è da sottolineare tuttavia che l'occorrenza di istruzioni di questo tipo ha una frequenza molto bassa).

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 ese-

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

Figura 2.5: Sostituzione dell'istruzione *jcxz*

cuzione dell'applicazione, non è possibile conoscere in maniera aprioristica quale sarà la destinazione.

Per questo motivo, si è deciso di adottare una tipologia di correzione a run-time 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 nella sezione 2.3.

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. 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.

```
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`

Ricordando quanto detto nel paragrafo 1.1.4 a pagina 10, i salti indiretti

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 2.3: Flag utilizzati dal modulo `branch_corrector`

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 i relativi significati, sono riportati in tabella 2.3. Anche questa tabella viene organizzata come 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 si vuole 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 `transaction_rw`. Contemporaneamente, 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, quali i salti *short*,

e le dimensioni degli oggetti di destinazione quali 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 `transactional_rw`, quella per `branch_corrector` e 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 nella sezione 2.3.

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 aumentata. 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 a controllare, per ciascuna istruzione di salto, 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 run-time 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; (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 — le tabelle ed i moduli di correzione a run-time.[1]

Il risultato ultimo di questo processo, in estrema sintesi, è la sostituzione di tutte le istruzioni che accedono in memoria con una `call` a `transactio-nal_rw`. Abbiamo dunque instrumentato il codice iniettando delle chiamate ad una routine esterna al posto di ogni istruzione di lettura/scrittura.

La routine richiamata può svolgere qualsiasi tipo di operazione a seconda dei nostri scopi. Ad esempio, come già accennato, in [14] una tale routine, che precedeva — senza sostituire — le sole scritture, serviva a tracciare in opportune strutture dati quali locazioni fossero state scritte. Oltre a ciò, si ha possibilità di scelta su quale classe di istruzioni considerare per l'instrumentazione, combinando a piacere i flag in tabella 2.1. Dicendo questo, si vuole sottolineare la generalità dell'approccio e degli strumenti sviluppati.

Nella sezione 3.3 verrà spiegato il funzionamento del modulo `transac-tional_rw` richiamato nel codice. Nella sezione ?? si spiegherà il funzionamento della funzione C `demultiplexer` che funge, appunto, da demultiplexer di operazioni per le API di lettura/scrittura di TL2.

2.3 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 [16, 19]: in questi documenti viene infatti spiegato chiaramente l'utilizzo di questo

[1] 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 `transactional_rw` non avrebbe informazioni circa l'istruzione che ha causato la sua attivazione.

tipo di salti. Essi permettono di tradurre in maniera efficiente costrutti complessi, come lo `switch-case` del linguaggio `C`. In figura 2.7 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 2.7: Elementi della tabella di rilocazione

In figura 2.7(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 per l'efficienza.

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

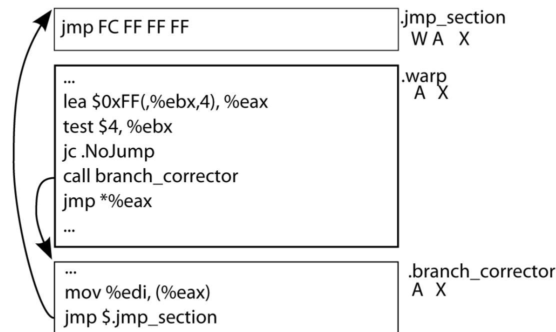


Figura 2.8: Funzionamento della correzione a run-time dei salti

Per questo motivo, nella sezione 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 2.8.

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 3.6) 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* sulla tabella degli shift, alla ricerca del più grande tra i minori degli indirizzi.[2]

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.

[2] 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: se, infatti, il limite superiore viene a coincidere con il limite inferiore, si è trovato esattamente il più grande tra i minori.

Capitolo 3

Integrazione in TL2

Nel presente capitolo verrà fatta una panoramica sulle *Software Transactional Memory* in generale ed una introduzione a *TL2*: l'implementazione STM alla quale verranno applicate le tecniche di instrumentazione statica degli eseguibili esposte nel capitolo 2. Nelle sezioni 3.3 e 3.4 verrà mostrato come si sono incluse le funzioni di lettura/scrittura di TL2 in un modulo "contenitore", che è quello richiamato dal codice instrumentato per effettuare le letture e le scritture. Nella sezione 4 si passano un rassegna gli obiettivi prefissi in partenza e le modalità con cui sono stati realizzati.

3.1 Introduzione ai sistemi STM

Le *memorie transazionali software* (il termine stesso "Software Transactional Memory" deriva da un articolo di Nir Shavit e Dan Touitou [18] in cui gli autori così chiamano il loro approccio risolutivo al problema della sincronizzazione) sono un insieme di tecniche che costituiscono argomento di discussione, sia nell'ambito del calcolo parallelo e distribuito che nell'ambito della progettazione dei linguaggi di programmazione, come un modo alternativo di intendere e trarre beneficio dalla *concorrenza*.

3.1.1 Motivazione delle STM

Da tempo è noto che la *legge di Moore* (vedi figura 3.1) è destinata a non valere più, a causa del fatto che la tecnologia di produzione dei transistor ha raggiunto distanze molecolari e perciò, semplicemente, non sarà più

La *legge di Moore* è l'osservazione che la densità dei transistor nei processori raddoppia ogni 18 mesi

possibile costruire microprocessori con maggiore densità di transistor integrati di quelli attuali. Nei microprocessori moderni ormai le informazioni si propagano a velocità prossime a quelle della luce.

Una soluzione per ottenere prestazioni sempre migliori per affrontare problemi di calcolo complessi sono, in attesa che le future tecnologie modifichino lo stato dell'arte nella produzione dell'hardware, i *sistemi paralleli*. Per "sistemi paralleli" si intende l'uso di più processori utilizzati simultaneamente per aumentare la velocità di esecuzione in un ambiente *time-sharing*. Tuttavia, i paradigmi di programmazione adottati nei linguaggi ad alto livello convenzionali — C, C++, Fortran, Java... — non sempre sono l'ideale per l'implementazione di applicazioni concorrenti.

Un paradigma di programmazione sicuramente supportato da tutti i linguaggi di programmazione moderni (e chiaramente da tutti i sistemi operativi moderni, come WinNT, Unix/Linux, MacOS...) è il paradigma *multi-thread*. Un *thread* è una unità di esecuzione *schedulabile* la quale differisce da un *processo* per un punto fondamentale: due thread pari (*peer threads*)condividono l'intero spazio di indirizzamento — ciò che non condividono invece è soltanto lo *stack*.

Risulterà chiaro che l'accesso alle variabili da parte dei singoli thread deve essere in qualche maniera *sincronizzato* per evitare che vengano lette o scritte informazioni inesatte. I lock a mutua esclusione (detti anche *mutex*) sono largamente utilizzati per raggiungere tale obiettivo nonostante siano forieri di numerosi errori di programmazione, data la difficoltà intrinseca legata al loro utilizzo. Esempi di errori comprendono [4, 8]:

- *Deadlock.* Se due thread possono richiedere lock multipli può succedere che l'uno richieda una risorsa detenuta dall'altro e viceversa: i due attenderanno l'un l'altro all'infinito.

- *Livelock.* Simile al deadlock, solo che i due thread non attendono all'infinito bensì cambiano continuamente stato, senza mai progredire.

- *Inversione di priorità.* Si ha quando un thread a bassa priorità può ottenere il lock su un oggetto richiesto anche da un thread ad alta priorità, progredendo a discapito di questo.

- *Convoying.* Potrebbe accadere che thread *deschedulati* continuino a detenere risorse richieste da altri thread impedendo a questi ultimi di progredire.

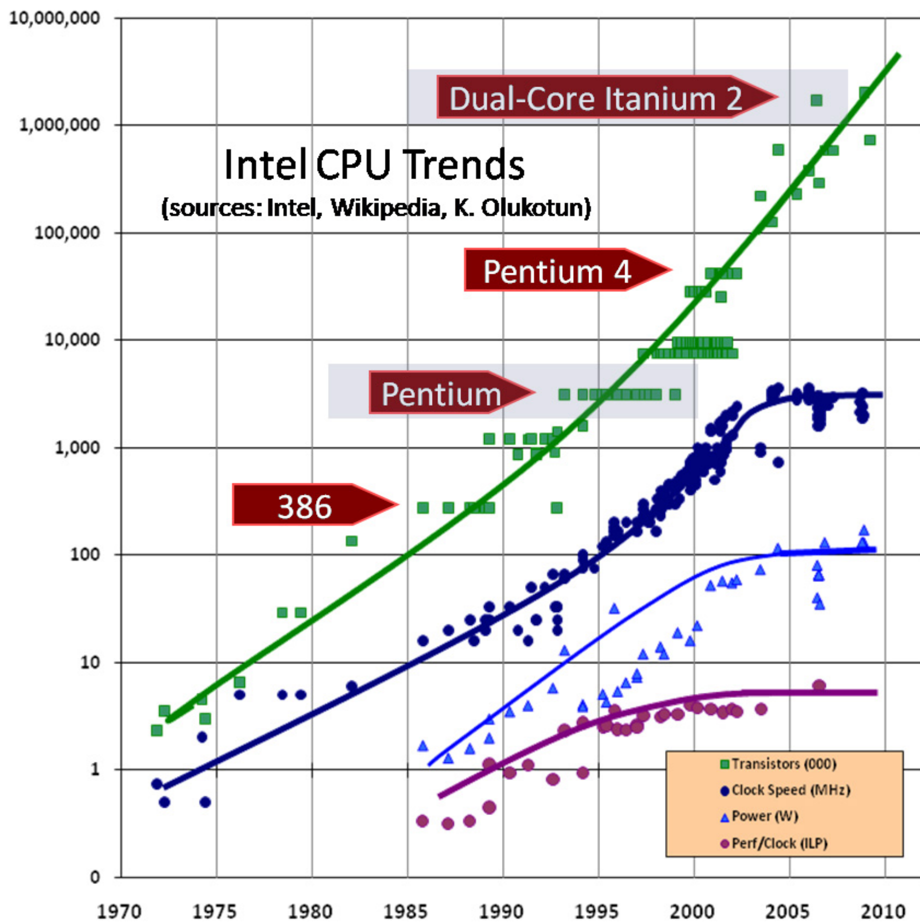


Figura 3.1: Dati esemplificativi della legge di Moore

Le memorie transazionali offrono una interfaccia più intuitiva verso il programmatore per risolvere il problema della sincronizzazione e possono, se usate con correttezza, diminuire l'incidenza dei problemi sopra elencati — i quali sono legati principalmente, lo ricordiamo, all'esperienza ed all'abilità del programmatore.

3.1.2 Sincronizzazione non-bloccante

Le memorie transazionali appartengono alla classe degli algoritmi di sincronizzazione *non-bloccanti*. In questa classe ricadono tutti quegli algoritmi di sincronizzazione che non prevedono la presenza di meccanismi *espliciti* di mutua esclusione. Le STM rientrano in tale classe perché offrono al programmatore un paradigma di programmazione orientato alle *transazioni* (vedi sezione 3.1.3) e non prevedono dunque una interfaccia di programmazione che contempli istruzioni di esclusione mutua.

All'interno della classe degli algoritmi di sincronizzazione non-blocccanti esiste una gerarchia di livelli di sincronizzazione:

- *obstruction-freedom*,
- *lock-freedom*,
- *wait-freedom*.

Obstruction-freedom, che offre garanzie più deboli rispetto alle altre, garantisce che un thread operante su una struttura dati condivisa completerà la sua esecuzione, *se* eseguito isolatamente. Ciò vuol dire che non c'è il rischio di incorrere in situazioni di deadlock — ma nulla mette al riparo dal rischio di livelock [7].

Gli algoritmi che invece garantiscono l'assenza di livelock, ovvero garantiscono progresso per i thread in esecuzione *considerati nel loro complesso*, sono gli algoritmi lock-free. L'unica garanzia non offerta da questi ultimi è quella che ogni thread *considerato individualmente* non sia soggetto a *starvation* [4]. Gli algoritmi lock-free sono anche algoritmi obstruction-free.

Wait-freedom, invece, è la più forte delle forme di sincronizzazione non-bloccante in termini di garanzie e assicura che ogni thread può terminare qualsiasi operazione in un *numero finito di passi* di esecuzione. Gli algoritmi wait-free sono anche algoritmi lock-free [6].

Tra le tre classi esiste la relazione: *obstruction-free* $\supset$ *lock-free* $\supset$ *wait-free*. In tabella 3.1 è riportato un riassunto di quanto appena esposto.

Classe	Riassunto	Problemi
obstruction-free	Ogni thread completerà la sua esecuzione se eseguito isolatamente	Livelock
lock-free	I thread fanno progresso, se considerati *nel loro complesso*	Starvation *singolo* thread
wait-free	Ogni thread completerà la sua esecuzione se eseguito isolatamente	

Tabella 3.1: Algoritmi di sincronizzazione non-bloccanti

3.1.3 Transazioni

Il modello di memoria gestito ed acceduto dalle STM ha — dal nome stesso — una logica "transazionale". Il concetto di transazione viene più ampiamente sfruttato nei *sistemi di basi di dati*, nell'ambito dei quali storicamente è nato e si è sviluppato.

Una transazione (consultare [5] per approfondimenti) è una procedura che esegue operazioni ripetitive e predefinite sui dati (come, nel caso delle STM, la lettura e la scrittura, ad esempio) e si compone di più passi, *temporanei* e *revocabili*, i quali vengono resi *permanenti* tramite un singolo comando, eseguito atomicamente, detto *commit*.

Richiamiamo in quanto segue le quattro proprietà fondamentali delle transazioni, note come proprietà ACID:

- *Atomicità.* La transazione verrà finalizzata completamente o sarà abortita completamente.

- *Consistenza.* Le modifiche apportate alla memoria la lasciano in uno stato consistente.

- *Isolamento.* La transazione ha lo stesso risultato sui dati in ingresso che avrebbe se eseguita non in concorrenza (isolatamente).

- *Durabilità.* Una volta *committed*, la transazione lascia una traccia permanente nella struttura acceduta.

Nell'ambito delle software transactional memory siamo maggiormente interessati alle proprietà di atomicità e isolamento [8].

Per un qualsiasi insieme di transazioni eseguite in concorrenza in un sistema parallelo e distribuito, infine, si vuole garantire la proprietà di *serializzabilità.* Richiamiamo le seguenti definizioni per completezza.

Definizione. SCHEDULE. Dato un insieme di transazioni $T_1, T_2 \dots T_n$, sia S una sequenza di azioni (appartenenti alle transazioni); S si dice essere uno *schedule* sull'insieme di transazioni $T_1, T_2 \dots T_n$ se in esso viene rispettato l'ordine di esecuzione stabilito all'interno di ogni transazione (i.e., se l'azione a viene prima dell'azione b in T_i, allora a viene prima di b in S.

Definizione. SCHEDULE SERIALE. Uno schedule S è detto seriale se le azioni di due qualsivoglia transazioni appartenenti ad S non vengono eseguite in *interleaving.*

Definizione. SERIALIZZABILITÀ. Uno schedule S è detto *serializzabile* se esso è equivalente ad uno schedule seriale sulle stesse transazioni (ciò vuol dire che i due schedule producono lo stesso risultato per dati in ingresso identici).

3.1.4 Supporto hardware

La maggior parte dei moderni microprocessori fornisce funzionalità di base, ma importanti, per implementare operazioni atomiche. L'istruzione COMPAREANDSWAP (o più brevemente CAS) è una di queste. Essa è una operazione atomica presente su molti microprocessori (vedere algoritmo in figura 3.2).

L'istruzione CAS viene eseguita *in un ciclo macchina* e può essere utilizzata, come vedremo a breve, per implementare diverse tipologie di algoritmi non-bloccanti — questo a discapito dalla sua apparente semplicità. Oltre a

```
CompareAndSwap(a : WordAddress, old : Word, new : Word)
 1  if *a = old
 2     then *a ← new
 3          return True
 4     else return False
```

Figura 3.2: CompareAndSwap

ciò, CompareAndSwap serve da base per l'implementazione di altre operazioni atomiche interessanti come *double-word compare & swap* (DCAS) o la più generale *multi-word compare & swap* (MCAS) [4].

Altre importanti operazioni atomiche disponibili sui comuni processori sono la LoadLinked e la StoreConditional. La prima carica un dato dalla memoria e pone un "lock" su di esso, di modo che una successiva StoreConditional sulla stessa locazione abbia effetto *solo se* nessuna altra operazione di scrittura sia intervenuta fra le due ed abbia modificato la locazione stessa (vedi figura 3.3).

```
LoadLinked(r : Register, a : Wordaddress)
 1  r ← *a
 2  Linked(a) ← True
```

```
StoreConditional(r : Register, a : Wordaddress)
 1  if Linked(a) = True
 2     then *a ← r
 3          r ← 1
 4     else r ← 0
```

Figura 3.3: LoadLinked e StoreConditional

Herlihy e Moss hanno proposto in [8] un set di istruzioni macchina alternativo e più significativo rispetto ai precedenti — benché simile — derivato dalla nozione di *linea di cache*. Si riportano di seguito le istruzioni proposte:

1. LoadTransactional e LoadTransactionalExclusive, le quali caricano una locazione di memoria in un registro privato, aggiungono la locazione di memoria nel *read-set* o nel *write-set* della transazione (la seconda delle due "suggerisce" che la locazione letta verrà modificata a breve).

2. StoreTransactional fa un *tentativo* di memorizzare un valore da un registro ad una locazione di memoria, ossia prepara il valore da scrivere e pone la locazione scrivenda nel *write-set* della transazione.

3. Commit tenta di scrivere sulla locazione precedentemente preparata da StoreTransactional. Se nessun'altra transazione ha modificato locazioni contenute nel *read-set* o nel *write-set* di questa transazione, i valori scritti divengono *committed* e un valore di successo è ritornato. Altrimenti, la transazione viene abortita ed un valore di fallimento viene ritornato.

4. Abort distrugge la transazione corrente senza *side-effects*.

5. Validate testa se il valore letto dalla LoadTransactional o LoadTransactionalExclusive sia ancora consistente con il valore in memoria.

L'uso di tali istruzioni ha la chiara implicazione introdurre limitaizoni ai programmi dovute alla dimensione dalla cache ed alle decisioni di scheduling.

3.1.5 Tecniche per algoritmi non-bloccanti

Per analizzare meglio la concorrenza e gli algoritmi di sincronizzazione non-bloccanti, è conveniente introdurre degli strumenti formali per ragionare su di essi.

Definizione. Consensus number. Un oggetto si dice avere un consensus number n quando al massimo n processi paralleli possono usare quell'oggetto per risolvere un problema di consenso (i.e., al massimo n processi possono usare quell'oggetto per raggiungere un accordo comune sulla soluzione ad un certo problema).

Risulta dimostrato che un oggetto con numero di consenso n *non può* essere usato per implementare un oggetto *wait-free* per un'operazione di consensus number $n_1 > n$; tuttavia, ogni oggetto di consensus number n è *universale* nel senso che può essere usato per implementare qualsiasi oggetto di consensus $n_1 = n$ (per approfondimenti, vedere [6]). CompareAndSwap ha consensus number $n = \infty$, il che lo rende universale per qualsivoglia oggetto.

Le primitive atomiche vengono tipicamente usate, come vedremo con un esempio nella sezione 3.1.6, in un ciclo *try-validate-commit* in cui una operazione è ripetuta diverse volte finché non si riesce a raggiungere il commit della transazione. Herlihy e Moss hanno proposto questo tipo di approccio per la loro memoria transazionale [8].

In molte proposte di sistemi di memoria transazionale software, queste primitive vengono impiegate in uno schema di *object versioning*. Ciò vuol dire che le modifiche ad un oggetto od alla memoria creano nuove *versioni* (copie) di quell'oggetto in memoria; una volta che le modifiche sono complete tali versioni divengono quelle "canoniche", ossia divengono quelle di riferimento per le successive letture e/o scritture di tutte le altre transazioni su quell'oggetto. L'azione di rendere canonica una versione è ottenuta tramite l'impiego, appunto, delle primitive atomiche.

"object versioning".

Esistono svariati meccanismi di gestioni delle versioni; si può semplicemente attribuire dei *"tag"* di versione e proprietà alle celle di memoria e renderle accessibili a tutti, permettendo che ogni thread in esecuzione possa validare una certa versione a discrezione (assumendo, è chiaro, che ogni thread agisca in maniera equa) [18]. Altre versioni prevedono la presenza di una semplice lista collegata di oggetti che per ogni oggetto mantiene una descrizione storica delle modifiche e lo stato della transazione che ha creato quella versione dell'oggetto stesso [2]. L'approccio multiversione è anche legato ai concetti di *logging*, *rollback* e *replay* delle transazioni; ciò permette di modificare tranquillamente le locazioni di memoria, senza farne copie, ma tenendo un registro delle azioni eseguite (appunto, un *log*) che consenta di disfarle nel caso in cui la transazione sia da abortire.

3.1.6 Un esempio di algoritmo non-bloccante

Illustriamo, per esemplificare quanto esposto, un esempio di applicazione di CompareAndSwap in un complesso algoritmo obstruction-free per l'im-

plementazione di una coda *double-ended*, proposta in [7]. Le procedure per aggiungere e rimuovere gli elementi dall'estremità a destra della coda sono mostrati in figura 3.4 ed in figura 3.5.

```
RightPush(A, value)
 1  while True
 2  do k ← Oracle(A, right)
 3      cur ← A_{k-1}
 4      prev ← A_k
 5      if prev.val ≠ End_r ∧ cur.val = End_r
 6        then if k = Max + 1
 7               then return Full
 8             if CAS(&A_{k-1}, prev, ⟨prev.val, prev.ctr + 1⟩)
 9               then if CAS(&A_{k-1}, cur, ⟨value, cur.ctr + 1⟩)
10                      then return OK
```

Figura 3.4: La procedura RightPush

In tali procedure, End_l e End_r sono sentinelle che segnano, rispettivamente, gli estremi sinistro e destro della coda. La procedura Oracle "indovina" dove si trovi l'estremo sinistro o destro e ne ritorna l'indice. Gli oggetti dela struttura dati sono coppie $\langle val, ctr \rangle$ dove *val* è del tipo corrispondente all'elemento e *ctr* è un intero.

3.2 Transactional Locking II

Dopo aver introdotto brevemente i fondamenti teorici e le proprietà fondamentali delle STM e degli algoritmi di sincronizzazione, ci focalizzeremo su una particolare implementazione STM per comprendere ad un livello più pratico in cosa consistano e come vengano utilizzate fattivamente le STM. Prenderemo in considerazione *Transactional Locking II* (d'ora in avanti semplicemente TL2), una implementazione di una STM presentata nell'articolo [3] e sulla quale si è basato il lavoro di instrumentazione statica.

RightPop(A)
1 **while** True
2 **do** $k \leftarrow$ Oracle($A, right$)
3 $\quad cur \leftarrow A_{k\text{-}1}$
4 $\quad next \leftarrow A_k$
5 $\quad$ **if** $cur.val \neq \text{End}_r \wedge next.val = \text{End}_r$
6 $\quad\quad$ **then if** $cur.val = \text{End}_l \wedge A_{k\text{-}1} = cur$
7 $\quad\quad\quad$ **then return** Empty
8 $\quad\quad$ **if** $\text{CAS}(\&A_k, next, \langle \text{End}_r, next.ctr + 1\rangle)$
9 $\quad\quad\quad$ **then if** $\text{CAS}(\&A_{k\text{-}1}, cur, \langle \text{End}_r, cur.ctr + 1\rangle)$
10 $\quad\quad\quad\quad$ **then return** $cur.val$

Figura 3.5: La procedura RightPop

3.2.1 Breve panoramica di TL2

TL2 è una implementazione di una STM che utilizza un meccanismo di sincronizzazione sui dati *basato su lock*.

L'algoritmo di TL2 adotta una nuova tecnica di validazione basata su *global version-clock*. A differenza di altre STM, TL2 si adatta con semplicità a qualsiasi tipo di sistema di memoria — compresi quelli che utilizzano `malloc/free`. Periodi di esecuzione non sicuri vengono evitati accuratamente grazie a questo nuovo global version-clock, il quale garantisce che il codice utente operi solo su stati di memoria consistenti. Allo stesso tempo, vengono offerte prestazioni comparabili (ed in alcuni casi migliori) rispetto alle altre implementazioni STM, sia lock-based che non-blocking.

Come già accennato, le STM che utilizzano meccanismi di sincronizzazione lock-based tendono ad avere prestazioni migliori rispetto a quelle che adottano meccanismi non-bloccanti. Tuttavia, entrambe le soluzioni presentano due importanti limitazioni che richiedono di essere rimosse per una esecuzione efficiente e per consentire, in futuro, che le STM abbiano un'ampia commercializzazione.

Una prima limitazione è quella di *closed memory systems*. Questo vuol dire che le locazioni di memoria usate da operazioni transazionali non sono

riutilizzabili da operazioni non transazionali, e viceversa. Ciò che invece si vorrebbe ottenere è un modello di memoria non chiuso, ma "riciclabile". Questa proprietà è facilmente ottenibile in quei linguaggi dotati di *garbage collector* ma sarebbe desiderabile anche in quei linguaggi che, come il C, utilizzano le operazioni standard `malloc` e `free`. Sfortunatamente, tutte le implementazioni STM non-bloccanti richiedono sistemi a memoria chiusa così come quelle lock-based le quali, però, necessitano anche di speciali versioni di operazioni come `malloc` e `free`.

Una seconda limitazione è quella di *specialized managed runtime environments*. Infatti, implementazioni STM efficienti richiedono degli ambienti di esecuzione speciali, capaci di gestire i comportamenti irregolari dovuti a quelle operazioni transazionali effettuate su stati inconsistenti della memoria. Tali situazioni comprendono l'esecuzione di transazioni *zombie*: sono zombie quelle transazioni che hanno osservato un read-set inconsistente, ma che ancora devono essere abortite. La lettura di dati inconsistenti, se non viene gestita in modo corretto dall'ambiente di esecuzione, può causare comportamenti inaspettati come *loop infiniti* ed *accessi illegali in memoria*.

TL2 rappresenta la prima implementazione STM che supera entrambe queste limitazioni. Essa lavora su sitemi a memoria aperta (sostanzialmente, con qualsiasi tipo di funzione `malloc` e di `free`) e consente alle transazioni di lavorare unicamente su stati consistenti della memoria.

La nuova idea, forse controintuitiva, introdotta da TL2, è quella di utilizzare il global version-clock, il quale viene incrementato ogni volta che una transazione scrive in memoria (e il quale può essere letto da qualsiasi transazione). Questo global clock può essere progettato di modo che per tutte le transazioni, ma soprattutto per quelle di piccola taglia, gli effetti dovuti alla *contention* siano minimi.

In TL2, tutte le locazioni di memoria sono accresciute con un lock che contiene un numero di versione (*versioned lock*). Le transazioni leggono il global version-clock e *validano* ogni locazione di memoria che presenti quel numero di versione (ciò per controllare che sulla locazione non sia già stato imposto il lock di qualche altra transazione). Questo meccanismo ci garantisce, a basso costo, che vengano lette solamente locazioni di memoria consistenti.

Le transazioni che effettuano soltanto operazioni di lettura (*read-only transactions*) non devono collezionare alcun read-set, mentre quelle che effettuano anche operazioni di scrittura (*write transactions*) ne hanno bisogno.

Una volta terminato di collezionare tutti i read/write set, le write transactions acquisiscono i lock sulle locazioni di memoria che devono essere scritte, incrementano il global version-clock e provano a validare (secondo le modalità appena descritte) il read-set: se la validazione va a buon fine le transazioni eseguono il commit, altrimenti abortiscono. Dopo aver fatto commit, le transazioni aggiornano le locazioni di memoria con il nuovo valore del global version-clock e rilasciano i lock a loro associati.

L'implementazione delle transazioni che effettuano solo operazioni di lettura è meno complessa. Le read-only transactions, infatti, leggono il valore del global version-clock e lo salvano in una variabile locale chiamata *read-version* (d'ora in avanti semplicemente `rv`). La validazione dell'operazione di lettura avviene solo successivamente, nel momento del commit, controllando che il *versioned write-lock* associato alla locazione di memoria che si era precedentemente letta risulti "libero" (ovvero, che nessuna altra transazione detenga il lock sulla locazione) e che il valore del versioned-lock stesso sia minore o uguale ad `rv`. Se vengono rispettate queste condizioni la transazione effettua il commit, altrimenti abortisce. Si noti come una tale implementazione delle transazioni read-only sia molto efficiente, anche perché non richiede la costruzione e la validazione di un read-set. Uno degli obiettivi di TL2, infatti, era proprio garantire un'esecuzione efficiente per questo tipo di transazioni — cosa che le altre STM non riescono a fare a causa del forte overhead introdotto.

L'algoritmo usato da TL2 adotta un *two-phase locking scheme* il quale implementa una politica ottimistica di acquisizione dei lock *commit-time* (versione *lazy)*. La rilevazione dei conflitti invece viene effettuata a livello di *stripe* (*per stripe*, od anche semplicemente *PS*), cioè viene allocato separatamente un array di lock e la memoria viene partizionata a stripe. Dopodiché, tramite funzioni hash, si mappa ogni locazione di memoria su di una stripe.

Possiamo affermare, in conclusione, che TL2 fornisce prestazioni superiori rispetto alle altre implementazioni STM lock-based, risolvendo molti problemi riguardanti la sicurezza e le prestazioni. Viene notevolmente semplificato, tra l'altro, il processo meccanico di trasformazione del codice sequenziale in codice transazionale.

3.3 Il modulo transactional_rw

Una volta completato il processo di instrumentazione, l'invocazione delle istruzioni di lettura e scrittura di TL2, chiamate TxLoad e TxStore, risulta abbastanza semplice. In realtà, però, verranno invocate due macro, STM_READ e STM_WRITE, che, all'interno del codice di TL2, servono da "wrapper" delle TxLoad e TxStore.

La routine assembly transactional_rw richiama al suo interno, sempre con una call, un'apposita funzione C: un demultiplexer di operazioni (come accennato) che richiamerà al suo interno le STM_READ e STM_WRITE a seconda dei parametri che riceve in ingresso. Come meglio esposto nella sezione 3.4, questa funzione, che chiameremo demultiplexer, è una API *aggiunta all'interno di TL2* — è questo il motivo per cui essa riesce a "vedere" le macro STM_READ e STM_WRITE, tra l'altro.

Per far sì che la routine di invocazione abbia un overhead minimo, essa è stata scritta direttamente in codice assembly: questo 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 transactional_rw crea nello stack una fotografia dello stato del processore al momento della chiamata. In particolare, esso verrà assemblato come riportato in figura 3.6.

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 2). Inoltre poiché nell'esecuzione della routine vengono effettuati dei confronti, 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, transactional_rw recupera nello stack il valore di ritorno della funzione chiamante, seguendo lo standard specificato in [19]. 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

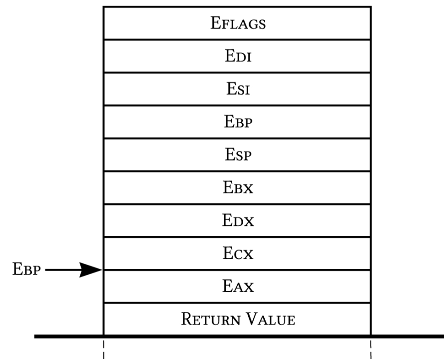


Figura 3.6: Finestra dello Stack di `transactional_rw`

tipo stringa, nonché se sia di lettura o scrittura o entrambe e se sia previsto o meno un registro di lettura o un dato immediato da scrivere.

In caso di istruzioni di tipo stringa (cioè `movs`, `stos` e `lods`), il codice eseguito è riportato in figura 3.7.

```
  movsbl insn_table+4(%edx), %eax
  imul -4(%ebp), %eax
  mov -28(%ebp), %edi
  mov -24(%ebp), %esi
  pushfw
  popw %bx
  bt $10, %bx
  jnc .DF0
  sub %eax, %edi
  sub %eax, %esi
.DF0:
  jmp .CallDemultiplexer
```

Figura 3.7: `transactional_rw` 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 5).

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 lettura/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`. Oltre al valore originale di `%edi`, viene anche recuperato il valore originale di `%edi`, il quale contiene invece l'indirizzo sorgente iniziale della lettura in memoria effettuata da un'istruzione `lods` o `movs` (nel caso di `movs` da memoria verso memoria). Ciò che occorre fare è determinare la direzione dell'operazione, ossia se questa avviene in avanti o all'indietro rispetto all'indirizzo contenuto in `%edi` ed in `%esi`.

Il parametro che discrimina la direzione della scrittura è il flag `DF` (Direction Flag) all'interno del registro `EFLAGS` (in particolare, esso è memorizzato nel decimo bit meno significativo del registro. 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 ai due indirizzi di base calcolati la taglia della scrittura calcolando, così, un nuovo indirizzo di base. Successivamente, ci penserà la funzione C `demultiplexer` richiamata successivamente dal codice assembly ad effettuare la lettura/scrittura.

Al termine della procedura, il controllo viene passato alla sezione `CallDemultiplexer`. Essa si occupa di richiamare al suo interno la funzione C `demultiplexer` che, a sua volta, richiama dal suo interno le `STM_READ` e `STM_WRITE` di TL2, passando alle stesse gli opportuni parametri. La funzione `demultiplexer` restituirà un puntatore ad una opportuna locazione di memoria che contiene l'eventuale dato letto. Sarà poi il modulo a livello assembly, tramite il campo `reg` accessibile dalla tabella riportata in figura 2.4 ad immagazzinare questo dato nel registro di destinazione nel caso in cui si sia effettivamente in presenza di una lettura e sia effettivamente presente un registro destinazione, cosa questa che si può evincere dagli opportuni bit

Notare che il salvataggio dei dati nel registro dovrà tener conto anche del valore del Direction Flag, come descritto in [10, Sez. 7.3.9].

del campo `flags` della medesima tabella, descritti nella sezione 2.2.1 a pagina 2.2.1. La funzione `demultiplexer`, lo ricordiamo ancora, verrà descritta invece nella sezione 3.4.

Qualora dal campo `flags` si determini che l'operazione è di tipo `mov`, il codice eseguito è quello riportato in figura 3.8.

```
  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 3.8: `transactional_rw` per le istruzioni di tipo `mov`

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

Qualora sia presente un indice, il codice del registro precedentemente salvato nella tabella del monitor (come spiegato nella sezione 2.2.1 a pagina 35) 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 3.6 ed i codici numerici associati ai registri (presentati nella sezione 1.1 a pagina 2), 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 si ricorda 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 si verificano le seguenti casistiche:

1. Se l'istruzione è di formato stringa e legge soltanto (i.e., è una `lods` od una `movs` da memoria a registro) in `%esi` abbiamo l'indirizzo sorgente ed in `%eax` la dimensione della lettura.

2. Se l'istruzione è di formato stringa e scrive soltanto (i.e., è una `stos` od una `movs` da registro a memoria) in `%edi` abbiamo l'indirizzo destinazione ed in `%eax` la dimensione della scrittura.

3. Se l'istruzione è di formato stringa e legge e scrive allo stesso tempo (i.e., è una `movs` da memoria a memoria) in `%esi` abbiamo l'indirizzo della locazione leggenda, in `%edi` abbiamo l'indirizzo della destinazione scrivenda ed in `%eax` la dimensione della scrittura.

4. Se l'istruzione è, come abbiamo definito, di tipo `mov`, avremo nel registro `%esi` la dimensione della scrittura e in `%edi` il suo indirizzo iniziale.

Questi parametri vengono opportunamente passati al modulo C `demultiplexer` dalla `transactional_rw` affinché venga eseguita la lettura o la scrittura od entrambe dalle opportune già citate funzioni di TL2.

3.4 Wrapping delle STM_READ e STM_WRITE

Facciamo un piccolo passo indietro per chiarire a che punto siamo giunti e cosa ci manca per proseguire verso la nostra meta:

- abbiamo sostituito ogni occorrenza di letture/scritture con una chiamata ad un modulo assembly `transactional_rw`;
- `transactional_rw`, il quale ha accesso alla tabella in figura 2.4, provvede a calcolare l'indirizzo di lettura e/o quello di scrittura, nonché a calcolare la dimensione della regione di memoria interessata, immagazzinando tali dati in registri dedicati;
- a questo punto, `transactional_rw` deve invocare la funzione C `demultiplexer` che effettuerà materialmente l'accesso in memoria, tramite le API fornite dalla implementazione STM di riferimento (nel nostro caso è TL2, ma nulla vieta di riscrivere la `demultiplexer` all'interno del codice di una qualsiasi altra implementazione).

Il motivo di includere la funzione `demultiplexer` nelle API di TL2, estendendo le stesse, è quello di supportare la chiamata alle `STM_READ` e `STM_WRITE` *senza modificare la logica interna* di TL2 e quindi avere uno strumento che sia *"compliant"* con la definizione di TL2. `demultiplexer` funge da modulo intermedio di raccordo tra il livello della routine assembly ed il livello del codice di TL2. Esso gioca un ruolo chiave nel processo di *disaccoppiamento* tra il codice scritto dal programmatore e le API offerte da TL2. [1]

Il modulo di raccordo `demultiplexer` — il cui nome mette in rilievo la sua funzione di *demultiplexer di operazioni* per le `STM_READ` e `STM_WRITE` — è progettato come un'estensione delle API messe a disposizione da TL2 (esso *entra a far parte di TL2*) e questo ha due importanti conseguenze:

- garantisce, allo stesso tempo, che non ci sia bisogno di modifiche alla logica interna di TL2 e che il programmatore non debba essere a conoscenza di tale logica (eccetto per la marcatura delle transazioni);
- permette al modulo di avere completa visibilità dei simboli presenti nel codice di TL2.

Ciò premesso, l'invocazione ed il funzionamento di `demultiplexer` non presentano particolari difficoltà. Essa ha bisogno soltanto di sapere:

[1] Si precisa, come evidenziato in più punti nel testo, che si vuole celare al programmatore la logica delle *operazione di lettura/scrittura* di TL2 e non già esonerarlo dal marcare le transazioni — cosa che necessariamente deve avvenire tramite le macro `STM_BEGIN_RD()`, `STM_BEGIN_WR()` e `STM_END()`.

1. se leggere o scrivere (o entrambe);
2. l'eventuale indirizzo di lettura;
3. l'eventuale indirizzo di scrittura (almeno un indirizzo deve essere presente);
4. quanti byte leggere;
5. quanti byte scrivere;
6. dato immediato da scrivere, se presente.

Alla fine della sua esecuzione, c'è bisogno che venga ritornato a `transactional_rw` il dato letto, se alcuno. Da quanto appena detto, è semplice indovinare la segnatura di `demultiplexer` (vedere figura 3.9). C'è bisogno di passare dei flags per comunicare se c'è bisogno di leggere o di scrivere (o entrambe). C'è bisogno di un indirizzo di lettura e di uno di scrittura (almeno uno dei due). Siccome le istruzioni che leggono e scrivono allo stesso tempo sono delle `movs` da memoria a memoria, in tutti i casi in cui ci sia lettura *e* scrittura la dimensione delle due operazioni è una sola: `size`.

```
void * demultiplexer(char flags, void *to_read, void *to_write, long size, long immediate);
```

Figura 3.9: Segnatura di `demultiplexer`.

Nella sezione 3.3, a pagina 66, viene riportato come e dove si trovano gli indirizzi di lettura/scrittura, nell'istante prima dell'invocazione di `demultiplexer`. Inoltre, è possibile estrapolare le informazioni da includere nei flag ed il valore del dato immediato dalla tabella, accessibile a `transactional_rw`, le cui entry sono rappresentate in figura 2.4. Non resta altro da fare che porre sullo stack (nell'ordine inverso in cui compaiono nella segnatura, come vuole lo standard [19]) i parametri da passare e chiamare la funzione tramite l'istruzione `call demultiplexer`.

A questo punto, il funzionamento interno di `demultiplexer` è intuitivo ed il passo fondamentale è richiamare le `STM_READ` e `STM_WRITE`. Un puntatore al dato eventualmente letto viene restituito e, tramite il campo `reg` della

tabella in figura 2.4, `transactional_rw` saprà in quale registro immagazzinarlo. Detto questo, abbiamo concluso la descrizione del funzionamento delle `transactional_rw` e `demultiplexer`.

In figura 3.10, viene riportata la definizione delle `STM_READ` e `STM_WRITE` (che, lo ricordiamo di nuovo, sono delle macro) con la relativa funzione che loro stesse mappano: notare la semplicità con cui sono state concepite le macro le quali ci occultano molti dettagli implementativi di TL2 (grande vantaggio offerto dall'estensione delle API includendovi la `demultiplexer` stessa). Si sottolinea di che, per riutilizzare il resto del codice cambiando l'implementazione STM di riferimento, basta riscrivere la *demultiplexer* affinché sia "compliant" con la medesima: passo questo necessario per raggiungere il *disaccoppiamento* dalla implementazione STM contingente.

```
#define STM_READ(var)        TxLoad(STM_SELF, (vintp*)(void*)&(var))
#define STM_WRITE(var, val)    TxStore(STM_SELF, (vintp*)(void*)&(var), (intptr_t)(val))
```

Figura 3.10: Le `STM_READ` e `STM_WRITE`.

Capitolo 4

Conclusioni

Nel presente lavoro ci si è proposti di rendere trasparente al programmatore, in fase di sviluppo, la logica delle funzioni di gestione della memoria di una implementazione STM.

L'obiettivo primario è stato rendere possibile al programmatore l'uso, per le funzioni di lettura/scrittura dalla memoria, dei costrutti messi a disposizione dal proprio linguaggio di programmazione (in questo caso, l'ANSI-C) senza doversi curare della logica e dell'implementazione delle relative istruzioni offerte dalla STM (tramite la sua API). La ragione di tutto ciò è la semplificazione del processo di produzione del software.

Per realizzare lo scopo si è scelto di agire direttamente in fase di compilazione e linking sull'output di compilazione, sostituendo ogni istruzione che riferisse la memoria con una chiamata alle istruzioni di lettura/scrittura della STM, tramite instrumentazione statica. L'invocazione avviene tramite il passaggio, al livello del codice assembly "iniettato", dei parametri necessari ad attivare una funzione C `demultiplexer` che estende l'API offerta dalla STM — e che è stata sviluppata ad-hoc.

Si sono introdotte le tecnologie coinvolte nell'intero processo, come riassunto in seguito:

- Si è scelto di focalizzarsi sul linguaggio macchina delle architetture Intel© 64 e IA-32. All'uopo, è stato sviluppato un *parser* del linguaggio per interpretare le istruzioni nel flusso di byte dell'eseguibile e permetterne così la modifica.

- Si è scelto di indirizzare il lavoro verso l'uso nei sistemi Unix-like — in particolare *Linux* — e per questo si sono studiate tecniche e librerie per la lettura e la modifica del formato di file *ELF* (Executable and Linkable Format) tipico degli eseguibili presenti su questi sistemi.
- Come sistema STM di riferimento si è scelto *"Transactional Locking II"* (TL2), una implementazione STM lock-based che presenta ottime caratteristiche in termini di prestazioni ed affidabilità.

L'obiettivo è stato raggiunto appieno e si è prestata particolare cura agli aspetti prestazionali ottimizzando al massimo le porzioni di codice asembly da iniettare nell'originale, di modo che avessero un impatto trascurabile, o per lo meno sostenibile, sull'efficienza complessiva in termini di numero di μops.

Gli strumenti sviluppati in supporto del progetto (parser, libreria di gestione degli ELF file, instrumentatore . . .), parte dei quali erano già in sviluppo presso il gruppo di ricerca HPDCS, si prestano al riuso in altre applicazioni per la loro generalità. Anche la tecnica utilizzata per l'instrumentazione risulta utilizzabile in altri contesti e per altri scopi con l'apporto di semplici modifiche e, per questo motivo, può essere di riferimento per progetti futuri.

Bibliografia

[1] Eldhose K. A., K.R. Lekshmi, e K. S. Lalmohan. IEEE 754 Floating Point Standard, FPGA'S and HDL'S in VLSI Design. *IJCA Proceedings on International Conference on VLSI, Communications and Instrumentation (ICVCI)*, (6):6–11, 2011. Published by Foundation of Computer Science.

[2] C. Scott Ananian e Martin Rinard. Efficient object-based software transactions. In *Proceedings of Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, Ottobre 2005.

[3] Dave Dice, Ori Shalev, e Nir Shavit. Transactional Locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pp. 194–208, 2006.

[4] Keir Fraser. *Practical lock-freedom.* Tesi di Dottorato di Ricerca, King's College, University of Cambridge, 2003.

[5] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pp. 144–154. IEEE Computer Society, 1981.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[7] Maurice Herlihy, Victor Luchangco, e Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pp. 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Maurice Herlihy e J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pp. 289–300, New York, NY, USA, 1993. ACM.

[9] Gruppo di ricerca "High Performance and Dependable Computing Systems" (HPDCS). `http://www.dis.uniroma1.it/~hpdcs`, 2011.

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

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

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

[13] Alessandro Pellegrini. Tracciamento trasparente ed efficiente di scritture su memoria dinamica con granularità arbitraria in architetture per il calcolo ottimistico. Tesi per Master, Computer Science Departement, DIS, Sapienza Univerità di Roma, Rome, Italy, 2007.

[14] Alessandro Pellegrini, Roberto Vitali, e Francesco Quaglia. DiDyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pp. 45–53, Washington, DC, USA, 2009. IEEE Computer Society.

[15] M. Riepe. `libelf`, a library to read, modify or create ELF files in an architecture-independent way. `http://directory.fsf.org/project/libelf/`. 0.8.9-stable released on 2006-08-22.

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

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

[18] Nir Shavit e Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997.

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

[20] Robert Wahbe, Steven Lucco, e Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–12, 1993.