Indice

Introduzione

Il lavoro di questa tesi si è svolto presso l'Istituto Nazionale di Fisica Nucleare nell'ambito del progetto APE. APE (*Array Processor Experiment*) ha lo scopo di progettare e costruire supercomputer paralleli in grado di eseguire le immense moli di calcolo necessarie per simulazioni fisiche: l'utilizzo principale delle macchine APE sono simulazioni di cromodinamica quantistica (*QCD*)[11, 12, 13].

Tutte le generazioni di macchine APE sono state programmate con un solo linguaggio ad alto livello: *TAO*. Il linguaggio TAO è particolarmente adatto alle applicazioni di fisica, ma la sua caratteristica principale è quella di basarsi su di un parser fuori dalla norma, chiamato *Zz* ("zeta zeta"). *Zz* è un parser dinamico in grado di riconoscere una grammatica evolvente: *all'interno di un programma si possono specificare nuove regole di grammatica che si possono usare nel seguito.* Questo permette una estendibilità del linguaggio senza precedenti e la definizione di operatori computazionalmente costosi in modo efficiente sfruttando appieno la macchina. Ricordiamo infatti che la caratteristica principale richiesta ai programmi che sono eseguiti sulle macchine APE, cosi come su tutti i supercomputer, è l'efficienza.

Zz è stato interamente progettato e realizzato all'interno del progetto APE. Ci sono però delle oggettive limitazioni all'utilizzo esclusivo di TAO e *Zz*. Una di queste è che TAO è conosciuto da poche persone al mondo, un centinaio al più. Questo fatto limita molto il diffondersi di APE e la conversione di algoritmi per questa famiglia di macchine. È stata quindi presa la decisione di iniziare un progetto a lungo termine che potesse dare ad APE un linguaggio ad alto livello potente, efficiente e conosciuto da tutti. La scelta naturale del linguaggio si è orientata sul C e sul C++, per la ovvia efficienza e diffusione. Riguardo al compilatore si è scelto di usare quello del progetto GNU per ragioni che discuteremo nel seguito.

Questa tesi è il primo lavoro all'interno del progetto APE in questa direzione: fornire al calcolatore APEmille, e in seguito ad apeNEXT, un compilatore C/C++ efficiente. Lo scopo di questo lavoro di tesi è duplice:

- definire una estensione dei linguaggi C e C++ per supportare il paradigma di programmazione SIMD[1].
- implementare un compilatore C e C++ per il calcolatore parallelo APEmille.

L'obiettivo é quello di aggiungere estensioni minime alla grammatica del C e del C++ standard, consentendo così un facile porting di applicazioni già esistenti o la scrittura *ex-novo* di nuovi programmi. La decisione di essere sempre

[1] *Single Instruction Multiple Data*, vedi capitolo 1.

il più vicini possibile allo standard ha portato alla definizione di un linguaggio C esteso con pochissimi nuovi costrutti, che chiameremo *SIMD C*, e ad un linguaggio C++ esteso, o *SIMD C++*, in cui le uniche estensioni sono quelle ereditate dal SIMD C. Questa scelta si discosta molto dagli approcci tradizionali che prevedono l'implementazione del paradigma SIMD attraverso librerie di classi, librerie di funzioni o espliciti costrutti paralleli. Nelle versioni SIMD proposte dei linguaggi C e C++ non vengono introdotti costrutti particolari per esplicitare il parallelismo: *le uniche aggiunte alla sintassi sono state fatte per risolvere delle ambiguità che una macchina parallela inevitabilmente ha.* Le minime estensioni ai linguaggi C e C++ standard rendono l'apprendimento delle rispettive versioni SIMD relativamente facile e veloce.

L'implementazione dei linguaggi SIMD C e SIMD C++ è stata realizzata a partire dalla catena di compilazione GNU. Questa scelta deriva da due motivi:

1. GNU è il compilatore di riferimento nel sistema operativo *Linux*, su cui si basa il sistema operativo di APEmille.

2. il codice sorgente GNU è disponibile a tutti con licenza di modifiche per scopi di ricerca.

GNU è un progetto iniziato nel 1984 da Richard Stallman, ricercatore del MIT (Massachusset Institute of Technology), a cui poi si sono aggiunti molti ricercatori di tutto il mondo con lo scopo di fornire software professionale disponibile a tutti, sorgenti inclusi. Attualmente il progetto GNU è coordinato e diretto dalla *Free Software Foundation*.

Nel seguito ci riferiremo a questo lavoro di "adattamento" del compilatore GNU ad APEmille con la parola "*porting*", specificando due tipi di *porting*: quello "ufficiale", ovvero previsto dal modello GNU e per cui esiste una interfaccia; e quello particolare di APEmille, in cui facciamo rientrare tutte le estensioni speciali fatte sul compilatore GNU per supportare APEmille.

La tesi è articolata in 7 capitoli:

1. Descrizione di una macchina SIMD astratta.

2. Storia del progetto APE e architettura di APEmille.

3. Estensioni dei linguaggi C e C++: SIMD C e SIMD C++.

4. Descrizione generale del modello di compilazione GNU.

5. Porting del compilatore GNU per APEmille.

6. Estensione del compilatore GNU per implementare SIMD C e C++ per APEmille.

7. Risultati

Capitolo 1

La macchina SIMD

In questo paragrafo descriviamo la architettura di una macchina SIMD con particolare attenzione alle soluzioni adottate nel calcolatore parallelo APE-mille. Altre macchine SIMD possono differire nei dettagli, ma l'architettura generale è quella esposta.

Macchine parallele SIMD consistono di elementi di calcolo paralleli sincronizzati con un unico processore di controllo. SIMD significa *"Single Instruction Multiple Data"* : ovvero un unico flusso di istruzioni che viene eseguito su tanti processori, ognuno dei quali esegue le operazioni su una base di dati diversa. Questa architettura si rivela adatta per la simulazione numerica della teoria dei quark detta *Cromo Dinamica Quantistica* (*QCD*) ed in campi applicativi quali la dinamica dei fluidi o l'elettromagnetismo mentre è totalmente inadatta in altri.

Il processore di controllo (in seguito CP) distribuisce contemporaneamente lo stesso flusso di istruzioni a tutti i nodi di calcolo. Questi eseguono le stesse istruzioni nello stesso tempo sui loro propri dati. Nella architettura SIMD considerata i nodi di calcolo sono processori (in seguito NP) specializzati per applicazioni numeriche. Gli NP sono collegati tra loro in una topologia di toro N-dimensionale. Ogni NP consiste in una ALU (unità logico aritmetica), un proprio register file, una memoria locale e una memoria di massa locale. Non esiste nessun tipo di memoria condivisa da tutti gli NP: la communicazione tra i nodi di calcolo viene realizzata attraverso un meccanismo di memory mapping che permette ad ogni NP di accedere alla memoria dei suoi vicini. La macchina è SIMD, ciò garantisce che non ci siano conflitti negli accessi in memoria.

1.1 Processore di Controllo, CP

Il CP gestisce i tipi di dati interi, esegue salti condizionati, chiamate a funzioni e genera indirizzi di memoria. Ogni istruzione è distribuita dal CP ad ogni NP della macchina, durante lo stesso ciclo di clock. Anche gli indirizzi di memoria tramite i quali sono acceduti i dati sono distribuiti dal CP a tutti gli NP: questi li useranno per accedere la propria memoria.

1.2 Processore Numerico, NP

Gli NP utilizati nel calcolatore APEmille sono processori specializzati in applicazioni numeriche, infatti supportano nativamente tipi di dati in virgola mobile, sia scalari (in precisione singola e doppia [7]) sia vettoriali o complessi (una coppia di scalari in singola precisione), ed infine numeri interi. “Supporto nativo” significa che il processore accetta istruzioni elementari per operazioni matematiche su dati dei tipi citati, con tempi di esecuzione molto ridotti. Non tutte le macchine SIMD supportano nativamente tipi vettoriali e complessi, quindi questa descrizione astratta di una macchina SIMD si avvicina in realtà a calcolatori paralleli molto avanzati come APEmille. Ogni NP ha al suo interno più unità di calcolo che possono lavorare in parallelo tra di loro e con l’unità che accede la memoria.

Gli NP, per gestire l’esecuzione condizionata di una parte di programma, possono testare condizioni locali (condizioni che interessano dati propri per ogni NP, e quindi vere o false a seconda dell’NP che le testa), e quando queste non sono verificate, possono disabilitare gli effetti delle istruzioni numeriche seguenti. Questo fa si che il CP continui a distribuire *tutte* le istruzioni ad ogni NP della macchina SIMD, ma che solo quelli in cui la condizione è vera le eseguono. L’istruzione *where* esegue il test della condizione locale. CP può anche determinare condizioni globali a seconda che tutti, nessuno o almeno uno degli NP soddisfano una condizione locale. Questo viene realizzato facendo testare agli NP la condizione locale e mandando al CP il risultato: questi opererà logicamente su questo insieme di risultati e deciderà il da farsi.

Ogni NP può eventualmente aggiungere un offset locale all’indirizzo di memoria che gli viene mandato dal CP: questo permette di realizzare un’architetture SIMD ibrida, nel senso che ogni NP può eseguire le stesse istruzioni non solo su dati diversi, ma anche su *indirizzi* di memoria diversi. Questa caratteristica si rivela fondamentale nell'implementazione di molti tipi di algoritmi, come ad esempio quelli che richiedono la lettura di un dato da una tavola di dati, come si vede nel programma seguente.

```
float indice,risultato, *tabella;
...
risultato = *(tabella+d);
...
```

Tra le possibili architetture parallele, quella SIMD è la piú semplice dal punto di vista della programmazione. Chi scrive un programma SIMD puó quasi sempre considerare la macchina come composta da un singolo processore che svolge le funzioni equivalenti alla coppia di processori CP e NP. Deviazioni da questo schema si rendono solo necessarie in alcune occasioni tra cui salti condizionati, accessi alla memoria di altri NP. É questa caratteristica delle SIMD che permette di adattare con minime modifiche linguaggi come C e C++ che sono stati concepiti per macchine a singolo processore.

Capitolo 2

Il calcolatore parallelo APEmille

2.1 Il progetto APE

Il progetto APE nacque alle fine del 1984. Alcuni fisici dell'INFN, ricercatori nel campo della *Lattice Gauge Theory* (o LGT), discutevano da tempo sulla possibilità di costruire una macchina con capacità di computazione almeno un ordine di grandezza superiore alle più veloci macchine vettoriali disponibili allora per il calcolo numerico.

La disponibilità sul mercato di processori per calcolo in virgola mobile segnò forse l'inizio del progetto APE, rendendo "*facile e forse anche possibile*" (come entusiasticamente riassunto da G. Parisi) la creazione di una macchina "spacca numeri" da zero. Lo sviluppo del progetto APE fu molto veloce: membri potenziali del gruppo APE si incontrarono al CERN alla fine del 1984 per la prima volta. Nel marzo del 1985, mentre i prototipi dei vari sistemi che componevano APE erano già sotto test, il progetto venne ufficialmente approvato dall'INFN. La commissione della prima macchina richiese alcuni mesi (periodo di tempo *estremamente* breve ma percepito come molto lungo e pieno di paure per i membri della collaborazione APE) e la prima macchina a 4 nodi (*APetto*) fu disponibile all'inizio del 1986, seguita da una macchina più grande della potenza di 1 GFlops nell'estate del 1987. Reali risultati di fisica furono presentati alla conferenza di LGT nel 1987, ed il primo articolo comparve sempre nel 1987. L'ultimo risultato basato sui dati forniti da APE fu pubblicato nel 1991.

La succesiva generazione di APE fu la famiglia APE100 con potenza di picco di 100 GFlops. La svolta decisiva in APE100 fu la progettazione completamente fatta in "casa" (*custom*) dei processori di calcolo, che ha permesso di definire i chip in completa sincronia con le esigenze del progetto e di raggiungere effettivamente una notevole potenza di calcolo. I primi risultati di LGT con APE100 apparvero alla fine del 1992. Le macchine di derivazione APE100 sono ancora largamente usate in molti centri di ricerca in tutta Europa. APE100 è stata utilizzata in campi anche differenti dalla QCD, come ad esempio fluidodinamica e metereologia.

L'ultima macchina realizzata dal progetto APE è APEmille, con una potenza di picco di 1000 GFlops. I primi prototipi di APEmille hanno iniziato a funzionare nell'estate del 1999, e la produzione in serie è iniziata alla fine dello stesso anno. APEmille si contraddistingue da APE100 per la maggiore potenza di calcolo dei singoli processori e per alcune differenze architetturali.

Un nuovo progetto sta iniziando: *apeNEXT*. La potenza di picco prevista per questa nuova macchina si aggira sull'ordine dei 10 TFlops. Poco si può dire su questa macchina in quanto la progettazione è ancora in una fase preliminare. I punti principali di questo nuovo progetto sono: tempi brevi, stabilitá, efficienza.

2.2 L'architettura di APEmille

APEmille è un calcolatore parallelo di tipo SIMD, scalabile, con topologia a reticolo cubico tridimensionale con condizioni al contorno periodiche e nodi di calcolo costituiti da processori VLIW e dalle loro memorie dati. La memoria programma dei processori e del controllore è separata dalle rispettive memorie dati in maniera conforme all'architettura Harvard.

I nodi di calcolo sono in grado di effettuare operazioni aritmetiche su dati interi a 32 bit, dati floating point in singola precisione, dati floating point in doppia precisione (64 bit), dati complessi floating point e coppie ordinate di dati in singola precisione. Una innovazione fondamentale è data dalla capacità di indirizzamento locale da parte dei nodi di calcolo: ad un indirizzo globale, cioè uguale per tutti i nodi, è possibile sommare uno spiazzamento calcolato sul nodo. Questo fa sì che l'istruzione eseguita (la stessa per tutti i nodi) possa operare su dati allocati in locazioni di memoria diverse. Per questa ragione si può classificare APEmille come un calcolatore SIMAMD (*Single Instruction stream - Multiple Address - Multiple Data stream*).

L'elemento base di APEmille è la *Processing Board* (PB), la scheda che

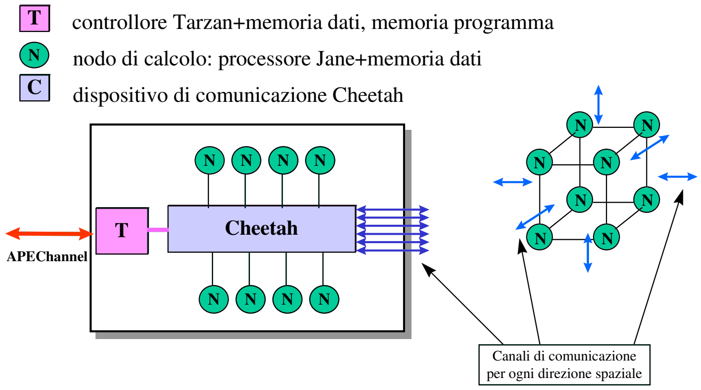

Figura 2.1: La Processing Board

ospita 8 processori aritmetici J1000 con le loro memorie dati, il controllore T1000 con la sua memoria dati, la memoria programma ed il dispositivo di comunicazione C1000, come mostrato in figura 2.1.

In *run mode* T1000 controlla il flusso del programma, invia le parole di microcodice e gli indirizzi globali agli 8 J1000 che svolgono i calcoli aritmetici, genera ed invia gli indirizzi remoti per C1000 che si occupa dei trasferimenti dei dati tra i diversi nodi di calcolo. In figura 2.1 sono mostrati i dispositivi presenti nella PB. Ogni PB è un sottosistema di APEmille che presenta tutte le funzionalità previste dalla macchina ed in cui i nodi di calcolo sono disposti ai vertici di un reticolo cubico 2x2x2 e comunicano con i loro 6 primi vicini.

La potenza di calcolo di picco del processore aritmetico J1000 si può calcolare facilmente considerando che esso è in grado di effettuare ad ogni ciclo di clock una operazione *normal* (a*b+c) con operandi complessi, che corrisponde ad 8 operazioni floating point in singola precisione. Alla frequenza di funzionamento di 66 MHz si ha quindi una potenza di 528 MFlop per il singolo nodo di calcolo. Nella tabella 2.1 sono elencate le possibili configurazioni di APEmille con le relative topologie e potenze di picco.

Configurazione	**Numero di nodi**	**Topologia**	**Potenza di picco** [GFlop]
PB	8	2x2x2	0.528
SUBCRATE	32	2x2x8	16.896
CRATE	128	2x8x8	67.584
TORRE	512	8x8x8	270.336
APEmille	2048	32x8x8	1081.344

Tabella 2.1: Configurazioni di APEmille e loro potenze di picco

Il funzionamento in *system mode* della macchina è gestito dall'esterno da una rete di host, la *Control Network*. Ogni host è una scheda PC dotata di più slot di espansione PCI (*Peripheral Computer Interconnect*), uno standard di fatto per la connessione ad alta banda passante di periferiche. Inoltre ciascun host, chiamato ACP (*Asynchronous Controller Processor*), è provvisto di una scheda di rete LAN (*Local Area Network*) e di dispositivi di memoria di massa e comunica con 4 schede PB su un bus parallelo, attraverso il protocollo dedicato APEChannel. L'interfaccia tra ACP e APEChannel è fornita da una scheda in standard PCI chiamata APEnic (*APE network interface card*) che è alloggiata in uno slot di espansione PCI della ACP. Il bus parallelo APEChannel è largo 39 bit (32 bit di parola + 7 bit di codice Hamming per il riconoscimento e la correzione degli errori) e lavora ad una frequenza di 33 MHz, corrispondente ad una banda passante per il canale di 132 Mbyte/s. L'APEnic si occupa anche di distribuire il segnale di *clock*, alle PB.

Il sistema operativo distribuito sulla rete di host provvede a dirigere le operazioni di I/O tra memorie e registri dei processori delle PB ed i dispositivi di memorizzazione di massa presenti sulla *Control Network*. Data la struttura modulare costituita dalle AU e dal sistema operativo distribuito, è prevista la possibilità di effettuare il partizionamento della macchina sia da parte di un singolo programma che potrà avviare in questo modo più *thread* SIMD simultanei, sia per eseguire più programmi su sottosistemi diversi della macchina.

Capitolo 3

Estensione dei linguaggi C e C++

Lo scopo di questo lavoro, come già detto, è l'implementazione in una macchina SIMD dei linguaggi C [3] e C++ [4] discostandosi il meno possibile dallo standard[1] dei linguaggi C e C++, estendendo minimamente la grammatica degli stessi.

La semantica però può essere molto diversa dai linguaggi tradizionali, perchè la macchina è SIMD e quindi ci sono molti processori che eseguono un programma, laddove il C e il C++ prevedono un solo processore.

In questa ottica tutte le estensioni per l'architettura SIMD sono state aggiunte al C, ed ovviamente sono presenti sul C++ come linguaggio evoluzione del C. Ciò per sottolineare che non sono state usate caratteristiche C++ per gestire o implementare l'architettura SIMD specificata.

Adattare e/o estendere un linguaggio per una particolare architettura

[1] Per la definizione dello standard del linguaggio C++ a cui si fa qui riferimento cifrare[8]

corrisponde principalmente a due attività:

- decidere come realizzare nell'architettura le caratteristiche del linguaggio
- estendere il linguaggio con costrutti che impiegano funzionalità particolari dell'architettura.

Di seguito vengono illustrate queste attività nel caso dei linguaggi C, C++ e della architettura SIMD illustrata.

3.1 SIMD C

Nell'ottica di minimizzare le estensioni dei linguaggi per permettere una facile migrazione ai SIMD C e C++ (e per poter usare la grande massa di applicazioni C esistenti) è stato deciso di implementare l'architettura SIMD direttamente nella semantica del linguaggio C, più alcuni nuovi costrutti. Questa scelta si differenzia da tutte le altre implementazioni correnti di estensioni parallele ai linguaggi C e C++ ([14], [15], [16]) che sfruttano librerie di classi o di funzioni per esplicitare il parallelismo. Questo approccio originale e innovativo può sfruttare la versatilità e la potenza del C++ per creare programmi più grandi e meglio organizzati. Notiamo che questo approccio non si applica a processori paralleli di tipo MIMD (*Multiple Instrucion Multiple Data*) in cui ogni processore puó eseguire un processo indipendente.

Analizziamo cosa si vuole intendere con “Linguaggio SIMD C”, per poi passare al “SIMD C++”.

Implementare un linguaggio in una architettura significa innanzitutto stabilire quale significato attribuire ai costrutti base di tale linguaggio: significato che è dettato da uno standard o è talmente ovvio che è tacitamente sottinteso. Si analizzeranno i seguenti aspetti del linguaggio C, e del loro nuovo o inalterato significato nell'architettura SIMD:

- variabili
- calcoli aritmetici
- conversioni di tipi
- espressioni logiche e comparazioni
- cicli e salti
- memoria dinamica
- gestione separata (CP e NP) di strutture e classi

3.1.1 Variabili

Nel linguaggio C standard dichiarare una variabile ha l'effetto di riservare (nel gergo tecnico e nel seguito *allocare*) una parte della memoria per immagazzinare dei dati. Usando poi il nome associato alla variabile in una espressione numerica è possibile accedere, in scrittura o lettura, al dato in quel momento presente nella locazione di memoria associata alla stessa variabile. Tutto ciò non presenta alcun problema per architetture in cui è presente un solo processore[2] ed una sola memoria. Nella architettura SIMD esposta sono presenti 2 tipi di processori, CP e NP, e gli NP sono per definizione molteplici proprio per rendere possibile l'esecuzione parallela dei calcoli. Ogni processore, CP e gli NP, possiede una propria memoria.

Cosa vuol dire quindi dichiarare una variabile? Nascono degli ovvi problemi che possono contraddire lo stesso principio di identità di Aristotele!

Ci sono almeno 2 ambiguità:

- in quale tipo di processore allocare la variabile
- riferendosi ad una varibile allocata negli NP, a quale di essi ci si riferisce.
- come gestire in una macchina SIMD variabili NP allocate diversamente in memoria.

Queste ambiguità sono state risolte in questo lavoro nel seguente modo:

- si sono divisi i vari tipi di dati del linguaggio C in due gruppi associati ai due tipi di processori: le variabili di un certo tipo sono allocate nel processore associato a quel tipo.
- delle varibili di tipi associati a NP esisteranno molteplici istanze, una per ogni NP. Si ribadisce che le variabili di tipo NP vengono allocate in ogni NP, cosicché ognuno di loro possa calcolare indipendentemente

[2]in cui ogni processo viene eseguito da un singolo processore, come avviene ad esempio nelle macchine con struttura SMP, adattata da workstation a server.

dagli altri. Tali variabili sono però allocate, in tutti gli NP, allo *stesso* indirizzo di memoria, per permettere il paradigma di programmazione SIMD.

- la allocazione della memoria è uguale per tutti gli NP.

Queste scelte eliminano ogni possibile ambiguità nella gestione dei dati. La divisione delle variabili in APEmille tra processore di controllo CP, detto T1000, e processore numerico NP, detto J1000, è dato nella tabella seguente.

Tabella 3.1: Tipi del linguaggio canonicamente associati ad un processore

Processore	Tipi canonicamente associati
CP	`int`[3], puntatori
NP	`float`, `double`, `complex`, `vector`, `localint`

Le scelte operate in questa divisione derivano dal fatto che gli NP vengono utilizzati principalmente per calcoli numerici, di qui la allocazione dei tipi numerici; mentre il processore di controllo CP viene usato principalmente per generare indirizzi e indici per accedere a vettori o matrici di dati, di qui l'allocazione dei tipi interi e dei puntatori.

I tipi `float` e `double` rappresentano rispettivamente numeri in virgola mobile in singola e doppia precisione, secono lo standard internazionale IEEE754 [7]. Il tipo `complex` si riferisce ad una coppia di numeri reali visti come parte reale e immaginaria di un complesso. Nella macchina SIMD considerata si possono in teoria avere complex con le parti costituenti in qualsiasi precisione:

[1]per "int" si intendono tutti i tipi interi, di ogni grandezza, sia con segno che senza segno.

nella realtà (APEmille) esistono solo i complessi in singola precisione come tipo nativo; i complessi in doppia precisione possono essere emulati via software dal compilatore, come avviene nei processori usuali che non dispongono di tipi complessi nativi. Il tipo `vector` rappresenta una coppia di numeri reali in singola precisione sui quali la macchina può operare istruzioni numeriche in parallelo ed indipendentemente. Il tipo `localint` rappresenta degli interi locali, gestiti ed allocati in ogni NP.

3.1.2 Calcoli Aritmetici

Avendo diviso i tipi di dati tra i due tipi di processori, la gestione dell'aritmetica ne deriva in modo naturale: ogni processore infatti possiede le istruzioni necessarie per eseguire calcoli sui tipi di dati che gestisce.

Non esistono quindi ambiguità su chi e come deve operare calcoli su variabili di un certo tipo.

3.1.3 Conversioni di tipi

Convertire dei dati da un tipo ad un altro può generare, come detto, ambiguità. Avendo infatti una variabile NP (di cui sono presenti molteplici istanze, una per ogni NP), e volendola convertire assegnandola ad una variabile CP (una sola istanza, in CP), quale delle tante istanze si deve scegliere per l'assegnazione?

Per evitare quasta ambiguità, le conversioni da tipi di dato NP a tipi di dato CP sono quindi proibite. Quelle inverse sono invece consentite: essendo presente infatti una sola istanza di ogni variabile CP, la si può trasmettere agli NP, ognuno dei quali ne creerà una copia locale.

Introduciamo il concetto di promozione che definiamo secondo l'uso comune: la promozione è una conversione da un tipo ad un altro implicitamente effettuata dal compilatore senza nessun esplicito costrutto. Esiste la possibilità per il programmatore di forzare esplicitamente la conversione da un tipo di dato ad un altro: chiamiamo questo *cast*.

Le possibili promozioni e i possibili cast consentiti nel linguaggio SIMD C per APEmille sono riassunti nella Tabelle 3.2 e 3.3.

Promozioni	`int`	CP ptr	NP ptr	`float`	`double`	`vector`	`complex`	`localint`
`int`	si	si	si	si	si	si	si	si
ptr tipi CP	si	si	si	no	no	no	no	no
ptr tipi NP	si	si	si	no	no	no	no	no
`float`	no	no	no	si	si	si	si	si
`double`	no	no	no	si	si	si	si	si
`vector`	no	no	no	no	no	si	no	no
`complex`	no	no	no	no	no	no	si	no
`localint`	no	no	si	si	si	si	si	si

Tabella 3.2: Promozioni consentite

3.1.4 Espressioni logiche e comparazioni

Espressioni logiche e comparazioni possono generare un problema di identità.

Facciamo un esempio.

```
int i,j,k;
```

Cast	int	CP ptr	NP ptr	float	double	vector	complex	localint
int	si	si	si	si	si	si	si	si
ptr tipi CP	si	si	si	no	no	no	no	no
ptr tipi NP	si	si	si	no	no	no	no	no
float	no	no	no	si	si	si	si	si
double	no	no	no	si	si	si	si	si
vector	no	no	no	no	no	si	no	no
complex	no	no	no	no	no	no	si	no
localint	no	no	si	si	si	si	si	si

Tabella 3.3: Cast consentiti

```
if (i>j || k>0)

  funzione_1();

else

  funzione_2();
```

Nel codice precedente non ci sono ambiguità: le variabili intere `i`, `j` e `k` vengono allocate e gestite da T1000, il processore di controllo. C'è un'unica istanza di queste variabili e verificare la condizione `i>j` non crea problemi di identità.

Analizziamo ora il caso di condizioni su variabili float:

```
float i,j,k;

if (i>j || k>0.0)

  se_vero();

else

  se_falso();
```

Cosa vuol dire `i>j` nel codice precedente? Ogni processore J1000 possiede la sua copia locale delle variabili `float`, quindi *è possibile* che in alcuni di loro la condizione dell'"if" possa essere vera, negli altri falsa.

Essendo la macchina SIMD, ovvero tutti i nodi di calcolo eseguono le stesse istruzioni, quale flusso deve seguire il programma? Quello associato alla veridicità della condizione (`se_vero()`), o quello associato alla sua falsità (`se_falso()`)?

In linea teorica si può scegliere di fare due cose in questa situazione:

1. cercare di simulare dei branch locali.

2. forzare tutti i processori a prendere una delle due strade.

Si ha quindi bisogno di qualcosa che possa far capire al compilatore quale delle due possibilità il programmatore vuole, in ogni situazione in cui ci siano ambiguità.

Ci sono due modi per risolvere il problema:

1. usare parole chiave speciali per esplicitare il comportamento voluto.

2. assegnare un comportamento di *default* che il compilatore deve seguire in situazioni di ambiguità e usare parole chiave per l'altra possibilità.

Vediamo dei codici:

```
// programma 1a
float i,j,k;
```

```
if (any(i>j) || all(k>0.0))
  se_vero();
else
  se_falso();


// programma 1b
float i,j,k;
where (i>j || k>0.0)
  se_vero();
elsewhere
  se_falso();


// programma 2
float i,j,k;
if (i>j || k>0.0)
  se_vero();
else
  se_falso();
```

Nei programmi 1a e 1b si è esplicitata la scelta che si vuole eseguire. Il costrutto `where...elsewhere` ha il seguente significato: i nodi in cui la condizione testata è vera eseguiranno la funzione `se_vero()`; i nodi in cui la condizione testata è falsa eseguiranno la funzione `se_falso()`. Per risolvere il problema di sincronizzazione delle macchine, nel calcolatore APEmille tutti i nodi di calcolo eseguono *entrambi* i blocchi. Nei nodi in cui la condizione testata è vera abiliteranno solo le istruzioni del blocco associato a “vero”, disabilitando gli effetti delle istruzioni del blocco associato a “falso”. Nei nodi in cui la condizione valutata è falsa si farà l'opposto.

I costrutti `any` e `all` hanno invece il significato di rendere globali delle condizione locali, condizionando il flusso del programma a seconda che ”qualcuno” o ”tutti” i nodi di calcolo verifichino la condizione in oggetto.

Nel terzo codice si lascia la decisione da seguire implicita: ovviamente ci deve essere un comportamento standard di cui il compilatore deve essere a conoscenza. Questo comportamento può essere sia quello di considerare la condizione ”globalizzata” con un costrutto `all` o `any`, oppure come simulazione di branch locale con il costrutto `where`.

È stata scelta la prima strada, ovvero quella di esplicitare il comportamento voluto, per diverse ragioni.

1. nuove architetture potrebbero necessitare di nuovi comportamenti.

2. dare un significato tacito a una condizione può generare errori di interpretazione, soprattutto nelle prime fasi di comprensione del linguaggio.

3. bisognerebbe comunque rendere esplicita la scelta non di *default*.

Per questi motivi si è deciso di estendere il linguaggio con le parole riservate:

`where`, `elsewhere`, `any`, `all` e `none`. Le prime hanno un significato chiarito precedentemente, mentre `none` è l'opposto logico di `all`.

3.1.5 Cicli e Salti

Abbiamo già praticamente detto tutto sui salti condizionati, con le soluzioni in una macchina SIMD per gestire condizioni locali.

Come detto in precedenza, il processore di controllo CP gestisce il flusso del programma, spetta a lui quindi effettuare i salti condizionati (con condizioni globali, ovvero su variabili CP o derivanti da `all`, `none` e `any`), i salti incondizionati e le chiamate a funzioni.

Entrambi sono implementati nella macchina SIMD con meccanismi tradizionali: il flusso di un programa si interrompe e si eseguono istruzioni di un'altra porzione di codice. Le chiamate a funzioni sono essenzialmente salti incondizionati, prima dei quali si memorizza il punto da dove si chiama, in modo da ripristinare l'esecuzione di quella parte di programma una volta terminata la funzione chiamata.

Esistono in generale due tipi di cicli: *contati* e *condizionati*; Nel linguaggio C i cicli `while` e `do ... while` sono condizionati; i cicli `for` possono essere sia condizionati che contati.

3.1.6 Memoria dinamica

Nel linguaggio C è possibile allocare dinamicamente dei blocchi memoria, ovvero riservare una quantità di spazio in memoria la cui grandezza è conosciuta solo a *run-time*, ovvero durante l'esecuzione del programma. Questo viene realizzato tramite la chiamata alla funzione `malloc`:

```
char * malloc (int size);
```

Dove `size` indica la grandezza della memoria da allocare. Il rilascio della memoria allocata avviene tramite la funzione `free`:

```
void free (char* address);
```

Dove `address` è l'indirizzo di memoria allocato in precedenza da `malloc`. Per ovviare al solito problema di identità tra i vari NP, abbiamo introdotto due tipi di `malloc`:

```
char * malloc_CP (int size);
char * malloc_NP (int size);
```

Ognuna delle due istruzioni alloca memoria per il rispettivo processore. La gestione delle memorie dei diversi NP è identica: la quantità di memoria allocata e gli indirizzi restituiti da `malloc_NP` sono uguali per tutti gli NP. I valori numerici presenti in quelle locazione di memoria saranno in generale diversi.

Nel linguaggio C++ la funzione `malloc` non viene usata direttamente dal programmatore, ma solo indirettamente, come ad esempio tramite il "new" del C++ che alloca dati in memoria e li inizializza. In questo caso la scelta corretta di allocazione viene gestita automaticamente.

3.1.7 Strutture

La gestione dell strutture pone un problema di fondo in una architettura SIMD con i due tipi di processore ognuno dei quali gestisce *solo* alcuni tipi di dati: dove allocare strutture con campi misti, ovvero sia con campi di tipo CP che campi di tipo NP.

La scelta adottata per strutture con campi misti è stata quella di allocare due blocchi, uno nella memoria CP e uno in quella NP, dove inserire rispettivamente i campi di tipo CP e i campi di tipo NP.

Nel primo campo del blocco CP ci sarà un puntatore nascosto al blocco di memoria NP. Ogni volta che si dovrà accedere al blocco NP si leggerà il puntatore nascosto, e si calcolerà l'indirizzo esatto nella memoria NP del campo che si vuole accedere. Un esempio di questa implementazione è dato in figura 6.3.

3.1.8 Indirizzamento locale

Come già visto nela macchina SIMD in oggetto i nodi di calcolo possono aggiungere un *offset* locale durante indirizzamenti della memoria.

Questo è implementato nel SIMD C++ attraverso l'utilizzo di espressioni di tipo `localint`.

Utilizzando il costrutto `jlofset(localint_expression)` sarà aggiunto, durante i prossimi accessi in memoria, un *offset* locale, diverso cioè in ogni processore di calcolo. Un esempio:

```
localint li;
complex array[20], c;
//ogni processore carica in ``li'' un valore diverso
li = carica_dato_locale();
c = array[12]; //accede al 12esimo elemento di array
jlofset(li),
c = array[12]; //accede all'elemento numero (12+li) di array
```

3.2 SIMD C++

Il linguaggio C++ unisce alla efficienza del C il supporto per la programmazione orientata agli oggetti, o *OOP*, la quale permette di realizzare software molto complessi con un elevato grado di astrazione e di manutenibilità. La scelta adottata nell'implementare tutte le caratteristiche e le potenzialità di una macchina SIMD, e di APEmille in particolare, nel C, ha permesso di poter utilizzare tutta l'astrazione concessa dal C++ senza dover rinunciare ad efficienza, e senza dover acquisire nuova conoscenza su come utilizzare classi o altro per sfruttare la macchina.

Unica variante alle implementazioni standard del C++ deriva dalla architettura a due tipi di processori della macchina SIMD in oggetto. Infatti, per realizzare il polimorfismo e l'ereditarietà multipla, caratteristiche fondamentali del C++, sono necessari dei puntatori nascosti all'interno dei dati di ogni istanza di una classe (*oggetto*).

Essendo i dati di un oggetto allocati separatamente su CP e sugli NP, a seconda del tipi di dato, ne segue la necessità di puntiatori nascosti (tutti residenti nella parte CP) per poter indirizzare entrambe le zone di memoria di ogni oggetto (quell relativa a CP e quella relativa a NP). Questo perché per accedere a determinati campi dell'oggetto è necessario conoscerne l'indirizzo, che varia a seconda che siamo su CP o NP.

3.3 Limiti dei SIMD C e C++

Le citate restrizioni sulle conversioni dei tipi rappresentano la più evidente limitazione nella definizione di questi linguaggi estesi. C'è un ulteriore aspetto secondario del linguaggio C che non può essere trasportato su un'architettura SIMD come quella esposta. Si tratta di istanze di *union* miste, ovvero contenenti sia tipi CP che tipi NP. Questo perchè una *union* è intrinsecamente uno stesso oggetto che si può accedere con diversi tipi di dati, mentre tipi CP e NP non possono risiedere nella stessa memoria. Facciamo notare che l'impossibilità di usare variabili `int` e variabili NP è relativa, in quanto si possono usare i `localint` al posto degli `int`. Il limite vero è nell'impossibilità di usare i puntatori nelle `union` miste.

Il SIMD C++ non ha nessuna ulteriore limitazione.

Capitolo 4

Modello GNU

In questo capitolo si dará una descrizione generale del modello di compilazione GNU e le fasi principali dell'attivitá di porting del compilatore per un nuovo processore, rimandando per approfondimenti e dettagli sul funzionamento al testo ufficiale "Using and Porting GNU CC" [9]. Nel capitolo 6 si motiveranno le estensioni fatte a questo modello per implementare i linguaggi SIMD C e SIMD C++ sul calcolatore parallelo APEmille, e se ne descriveranno le implementazioni.

La catena di compilazione GNU prevede diverse fasi per tradurre un programma sorgente scritto in C o in C++ nel corrispondente scritto in linguaggio macchina. Come descritto in fig.4.1, il file sorgente scritto in C o in C++ viene dapprima preprocessato dal preprocessore (identico per entrambi i linguaggi); successivamente inizia la compilazione vera e propria che dipende dal linguaggio.

Le fasi tradizionali della compilazione sono lo *scanning* e il *parsing* (per

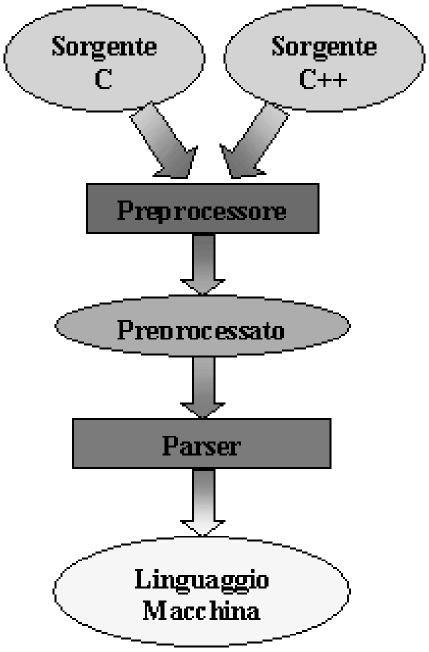

Figura 4.1: Catena di compilazione

approfondimenti cifrare [1, 2, 5]). Lo scanning viene eseguito da un modulo detto "scanner" o "lexer", il parsing dal "parser".

Il lexer crea dal programma sorgente una sequenza di *token*, ognuno dei quali descrive una parola o un simbolo del programma (vedi fig.4.2).

Il parser e' implementato attraverso un automa a stati finiti. L'automa, in ogni stato in cui si trova, e' in grado di riconoscere frasi che rispettano alcune regole grammaticali (dipendenti dallo stato in cui si trova) e di compiere le opportune azioni semantiche (vedi fig.4.3). Per azione semantica si intende una serie di istruzioni necessaria a creare e gestire una rappresentazione interna del programma che si sta compilando: questa rappresentazione servira' successivamente a generare il codice macchina finale.

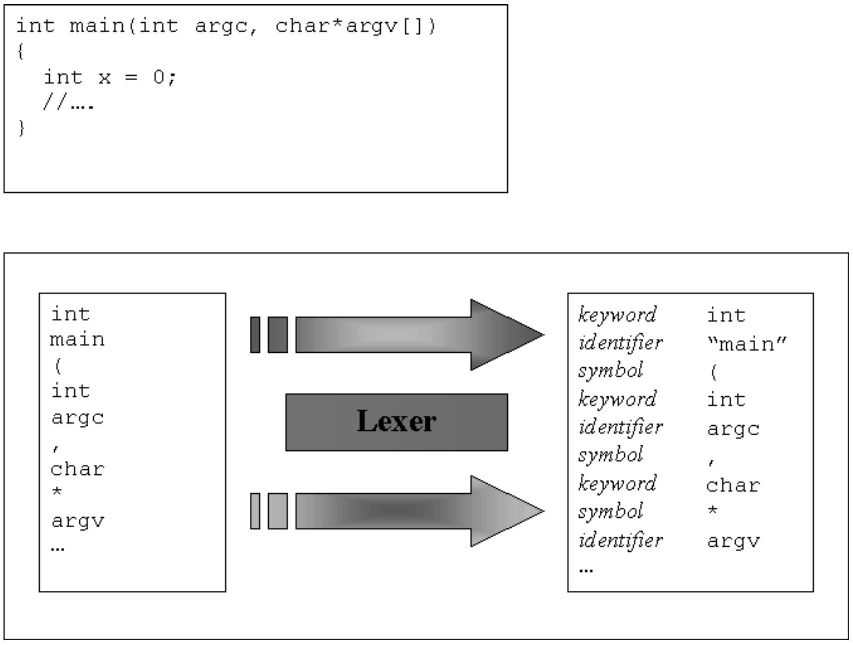

Figura 4.2: Lexer e tokens

Il parser legge un token alla volta dal lexer; si può quindi implementare il compilatore in modo che il lexer esegua lo scanning del programma sorgente man mano che il parser richiede token, oppure che legga tutti i token e poi li fornisca al parser in un secondo momento. Nel compilatore GNU il lexer e' implementato nel primo modo citato. Ad ogni token letto, il parser verifica se l'insieme dei token letti soddisfa una regola grammaticale associata al suo stato attuale: in caso affermativo applica la corrispondente azione semantica, in caso negativo legge il prossimo token.

Nell'implementazione GNU il lexer e il parser sono automaticamente generati a partire da un descrizione della grammatica con le azioni semantiche associate ad ogni regola. Un esempio di grammatica si può vedere in fig.4.4.

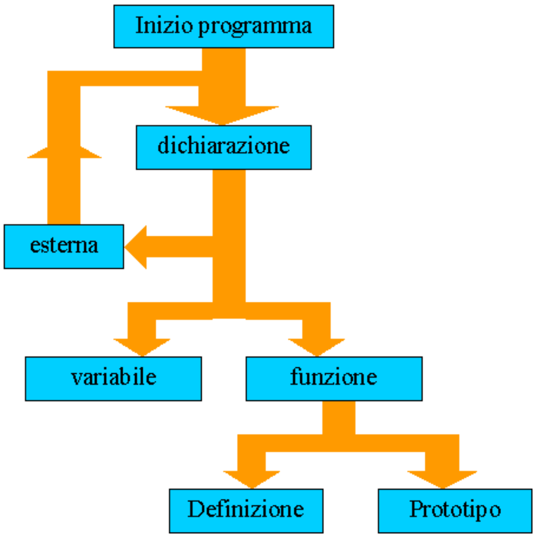

Figura 4.3: Parser

```
stmt:
        compstmt
            { stmt_count++; }
        | all_iter_stmt
      | FOR
        '(' xexpr ';'
            { stmt_count++;
              emit_line_note ($<filename>-1, $<lineno>0);
              emit_nop ();
              if ($3) c_expand_expr_stmt ($3);
            }
      | simple_if ELSE
            { c_expand_start_else ();
              $<itype>1 = stmt_count;
              position_after_white_space (); }
        lineno_labeled_stmt
            { c_expand_end_cond ();
              if (extra_warnings && stmt_count == $<itype>1)
                warning ("empty body in an else-statement"); }
```

Figura 4.4: Esempio di Grammatica

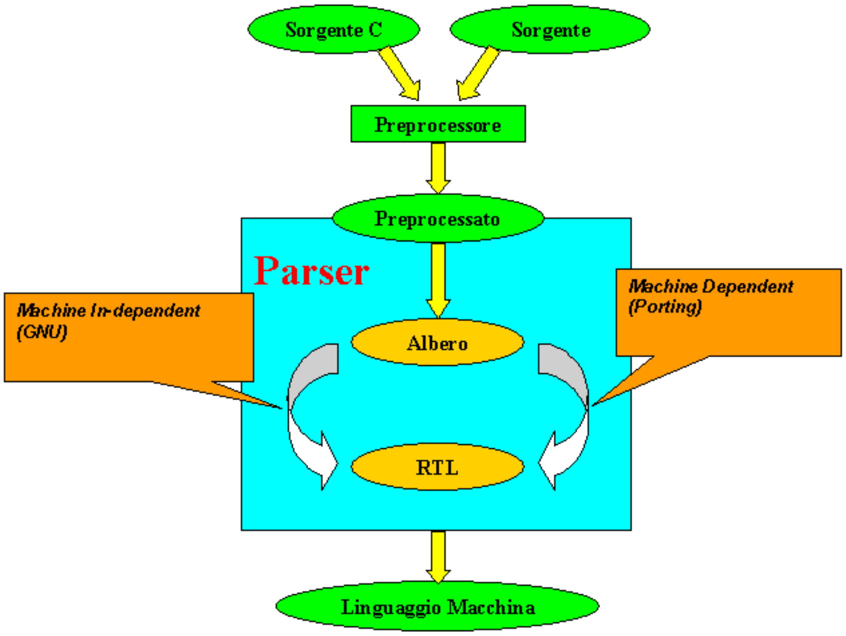

Figura 4.5: Internal

Il parser genera dunque una rappresentazione intermedia del programma sotto forma di albero. Successivamente questo viene *espanso* (per usare un termine GNU) in codice RTL che poi viene tradotto in linguaggio macchina (vedi fig.4.3). La struttura della catena di compilazione vista più in dettaglio è in fig.4.5.

Introdurremo ora brevemente la rappresentazione interna e il linguaggio RTL, descrivendoli come entità a se stanti. In seguito spiegheremo il processo di traduzione vero e proprio da albero a RTL a linguaggio macchina, e come questo sia operato a partire dalla descrizione della macchina (o *Machine Description*) fornita.

4.1 Rappresentazione Intermedia: albero

La rappresentazione intermedia è memorizzata sottoforma di albero radicato (ovvero in cui ogni nodo può avere un numero variabile di figli), e mantiene tutte le informazioni del programma sorgente. Tale rappresentazione è indipendente dal linguaggio in cui è scritto il programma e dalla macchina per cui si compila.

Ogni statement viene rappresentato come un albero, quindi un programma è rappresentato come una lista di alberi.

Nell'albero, i nodi possono rappresentare:

- variabili
- operatori i cui eventuali operandi sono sottoalberi.
- descrittori di tipi

Gli archi collegano gli operatori o i descrittori ai loro eventuali operandi, che possono essere singoli nodi o interi sottoalberi.

Diamo un esempio di come viene tradotto in rappresentazione intermedia una linea di codice C.

```
int a,b,c;
a = b*c+3;
```

Nella rappresentazione interna ad albero non esiste nessuna relazione temporale tra i nodi: essi non sono istruzioni, bensì solo una descrizione più astrat-

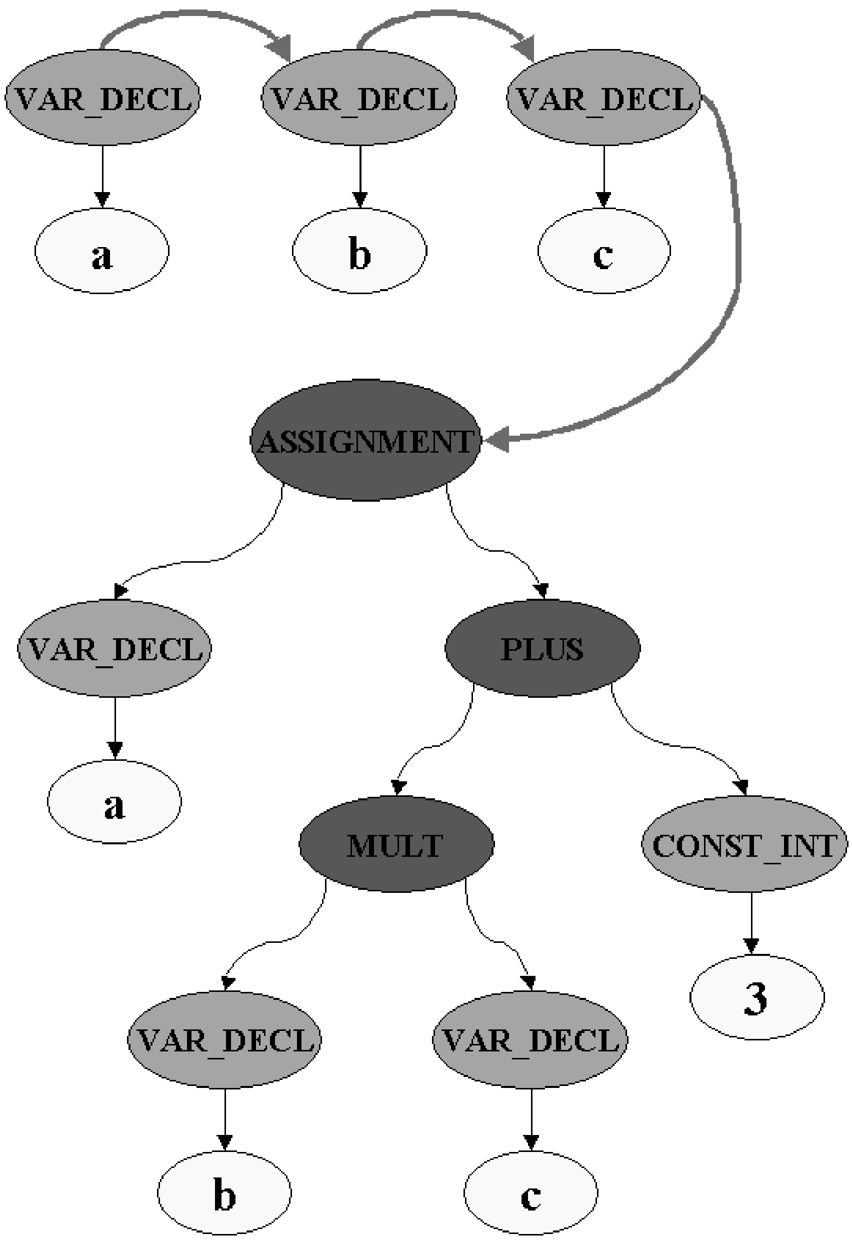

Figura 4.6: Esempio di albero

ta dell'algoritmo descritto nel programma sorgente. Non esiste inoltre nessuna descrizione dell'allocazione delle risorse della macchina, come la memoria o le sottounitá del processore. L'albero è completamente indipendente dalla particolare macchina per cui si sta compilando.

Durante la creazione e la manipolazione della rappresentazione ad albero, vengono eseguite due tipi di ottimizzazioni:

- *costant-folding* (raggruppamento delle costanti).
- semplificazioni aritmetiche (es.: $a * b + b * c = (a + c) * b$).

4.2 RTL

RTL sta per *Register Transfer Language*: esso descrive essenzialmente trasferimenti o operazioni su un insieme di registri o sulla memoria. Si può pensare all'RTL come ad un linguaggio di basso livello di una macchina virtuale. Un codice RTL è composto da una sequenza di istruzioni che possono avere un numero variabile di operandi: ogni operando può a sua volta essere un semplice oggetto base (registro o memoria) o una espressione RTL più complessa. Nell'esempio che segue, si trova una espressione RTL che assegna alla variabile `var` allocata in memoria il contenuto del registro 812 più la costante intera 2. I vari "SI" presenti indicano il *modo macchina* associato allle operazioni e agli operandi. Esistono vari modi macchina nel linguaggio RTL: ognuno di essi è canonicamente associato a un tipo di dati del linguaggio ad alto livello (C o C++). Nella fattispecie “SI” è il modo macchina canonicamente associato al tipo `int` del linguaggio C. Nella descrizione della macchina che si fornisce al GNU si possono implementare alcuni o tutti i possibili modi macchina, associandoli in modo adeguato ai tipi di dati supportati dalla macchina in oggetto.

```
(set (mem:SI (symbol_ref "var"))
     (plus:SI (reg:SI 812) (const_int 2)))
```

Una corrispondenza predefinita tra modi macchina e tipi del linguaggio ad alto livello è in 4.1.

Tabella 4.1: Associazione modi macchina - tipi linguaggi - tipi APEmille

Tipo linguaggio	Modo macchina	Tipo APEmille
`int`	SI	interi a 32 bit
`float`	SF	reali in singola precisione (32 bit)
`double`	DF	reali in doppia precisione (64 bit)
`complex`	SC	complessi come 2 `float` (2*32 bit)
`vector`	SV2	vettore di 2 `float` (2*32 bit)
`localint`	SLI	interi locali (J1000) (32 bit)

Il codice RTL rispecchia la sequenza temporale delle istruzioni del programma, con gli eventuali salti presenti. In un primo momento il compilatore, avendo bisogno di registri, alloca tutti ”registri virtuali”, senza limitazioni di numero. Successivamente, durante la fase detta *reload* (vedi paragrafo 4.3.4) esso rimappa tali registri virtuali su registri fisici, cercando di utilizzare al meglio le risorse della macchina. Le variabili allocate in memoria non hanno, nel codice RTL, un indirizzo preciso: nella catena di compilazione GNU l'allocazione esatta dei dati in memoria avviene durante la fase di "linking".

Durante la generazione di codice RTL vengono eseguite le seguenti ottimizzazioni:

- semplificazioni di condizioni (es: `a<b && b>a == FALSE`).
- scelta su dove inserire il controllo nei loop (all'inizio o alla fine).
- scelta su come generare `switch` statement.
- scelta se espandere o meno ogni funzione *inline*.

4.2.1 Condition Code

La macchina virtuale di cui l'RTL rappresenta il linguaggio ha un meccanismo di verifica delle condizioni e relativi salti condizionati bsato sul concetto di *Condition Code*, o CC. In generale, nella compilazione di un programma si ha bisogno di confrontare due grandezze, e di saltare ad un'altra istruzione se questa condizione è vera o falsa. La comparazione tra le due grandezze può essere di 6 tipi: uguale, diverso, maggiore, minore, maggiore o uguale, minore o uguale.

Nella macchina virtuale del modello GNU questo si svolge confrontando le due grandezze con una generica istruzione `compare`, e memorizzando il risultato nel *Condition Code*: quest'ultimo si può pensare come un insieme di 6 bit che memorizzano la veridicità dei 6 tipi di confronti possibili tra i due operandi. In verità sono sufficienti solo 2 bit, uno per memorizzare l'uguaglianza tra i due operandi e uno che indica se il primo è più grande del secondo (o vicevers). È banale ricavare poi tutte le altre condizioni da questi due dati.

Una istruzione immediatamente successiva esegue un salto condizionatamente al valore di uno dei 6 bit del *Condition Code*: ci sono dunque 6 possibili istruzioni che verificano il CC (vedi Tab. 4.2).

Tabella 4.2: Istruzioni RTL che verificano il Condition Code

eq	vero se gli operandi sono uguali
ne	vero se gli operandi sono diversi
lt	vero se il primo operando è più piccolo del secondo
gt	vero se il primo operando è più grande del secondo
le	vero se il primo operando è più piccolo del secondo o sono uguali
ge	vero se il primo operando è più grande del secondo o sono uguali

Nel modello GNU la verifica del CC avviene *sempre* dentro una istruzione `if_then_else`, che è il corrispettivo RTL degli *statement* `if...then...else` dei linguaggi C/C++.

Un esempio di codice RTL che effettua una comparazione si può trovare di seguito:

```
(set (cc0) (compare:SF (reg:SF 240)
           (mem:SF symbol_ref "variabile_float")))
(set (pc)
(if_then_else (lt (cc0)  (const_int 0))
```

```
      (label_ref  42))
      (pc)))
.... codice relativo al branch "if"
(set (pc) (label_ref 45))
(code_label 42)
.... codice relativo al branch "else"
(code_label 45)
.... codice che segue lo statement "if...then...else"
```

La prima istruzione assegna a `cc0`, un "registro" che memorizza lo stato del *Condition Code* il risultato della comparazione tra il registro 240 e la variabile residente in memoria `variabile_float`. Entrambi questi dati sono acceduti in modo macchina SF, associato canonicamente al tipo `float` dei linguaggi C/C++.

La seconda istruzione imposta il valore del *program counter* (registro che contiene l'indirizzo della prossima istruzione da eseguire: un cambio del *pc* risulta quindi una istruzione di salto) a due possibili valori: la label 42 (una label è un indirizzo di programma, ovvero l'indirizzo di una istruzione del programma) o lo lascia invariato. L'istruzione `if_then_else` restituisce il secondo o il terzo argomento (appunto la label 42 o *pc*) a seconda della veridicità del primo argomento. In questo caso il primo argomento è l'istruzione `lt` (dall'inglese "less than", "minore di") la quale verifica che nel CC (impostato dall'istruzione precedente) sia impostato a vero il bit relativo alla condizione "minore".

Questo meccanismo di gestione delle comparazioni e dei salti condizionati è abbastanza diverso dall' architettura di APEmille: l'implementazione adottata nel porting saraà spiegata più avanti.

4.3 Generazione di codice macchina

L'*espansione* della rappresentazione a albero al codice RTL avviene *statement* per *statement* e la parte del compilatore che opera questa espansione è generata automaticamente usando le informazioni sulla macchina proprie di ogni porting.

L'idea centrale è che gli operatori dell'albero corrispondono a precise istruzioni RTL, e nella *machine description* si fornisce il modo di *espandere* questi operatori in codice RTL, e successivamente in codice macchina.

Questa espansione avviene in vari passi, alcuni dei quali possono essere ripetuti. Nel seguito verranno spiegati questi passi.

4.3.1 Pattern di istruzioni

Nella machine description ci sono i pattern di istruzioni utilizzati nella fase di espansione degli operatori canonici da albero a RTL, e altri pattern che descrivono istruzioni non rappresentate direttamente nella rappresentazione interna. Tutti i pattern possono avere un nome: nella prima espansione dell'albero vengono utilizzati solo pattern con i nomi predefiniti. Questi pattern possono essere di due tipi: *define_expand* o *define_insn*.

In una *define_expand* si può inserire del codice che manipola l'istruzione per adeguare il codice RTL (che ricordiamo corrisponde al linguaggio di una macchina *virtuale*) al codice per la macchina in oggetto. Infatti, la macchina per cui si compila può non supportare nativamente tutte le istruzioni RTL, o può supportarne altre più o meno complesse di quelle RTL.

Alcune tra le manipolazioni usuali in una *define_expand* sono:

- dividere l'istruzione RTL in più istruzioni.
- forzare gli operandi a risiedere in memoria o nei registri.
- memorizzare gli operandi per usarli successivamente.

Un esempio di *define_expand* per il porting su APEmille è la seguente:

```
(define_expand "mulsf3"
[(set (match_operand:SF 0 "register_operand" "=j")
(mult:SF (match_operand:SF 1 "general_operand" "%j")
(match_operand:SF 2 "general_operand" "j")))]
""
"{
  if (GET_CODE(operands[1]) != REG)
      operands[1] = force_reg (SFmode, operands[1]);
  if (GET_CODE(operands[2]) != REG)
      operands[2] = force_reg (SFmode, operands[2]);
}")
```

Il nome "mulsf3" è il nome predefinito associato al pattern che il compilatore cerca quando deve espandere una moltiplicazione tra dati di tipo SF (Single Float, modo macchina canonicamente associato al tipo `float` del linguaggio C/C++). Nell'esempio, il codice di preparazione forza gli operandi a risiedere nei registri: questo è dovuto all'architettura RISC di APEmille, in cui le operazioni matematiche possono avvenire solo tra registri interni al processore. Ciò permette di parallelizzare i trasferimenti da memoria a registri e viceversa con le operazioni logico-matematiche.
L'altro tipo di pattern presenti nella machine description sono le *define_insn*. Queste consistono in una diretta corrispondenza tra la rappresentazione ad albero e codice RTL.

Ci sono però due tipi di *define_insn*: quelle con nomi predefiniti e quelle con nomi definiti nel porting. Le prime, come detto, sono usate per espandere operatori usati nella rappresentazione interna. Le seconde sono usate dalle *define_expand* per generare codice appropriato per la macchina in oggetto o dal compilatore per ottimizzare il codice o per tradurre le istruzioni RTL generate dalle *define_expand* che possono non essere operazioni canoniche.

Uno schema di questo meccanismo si può trovare in fig.4.7.

Il metodo delle *define_expand* e *define_insn* è facilmente comprensibile se lo si vede applicato a un caso specifico: gli operatori floating point dei microprocessori numerici di APEmille, J1000.

J1000 è ottimizzato per eseguire quelle che nella letteratura del progetto

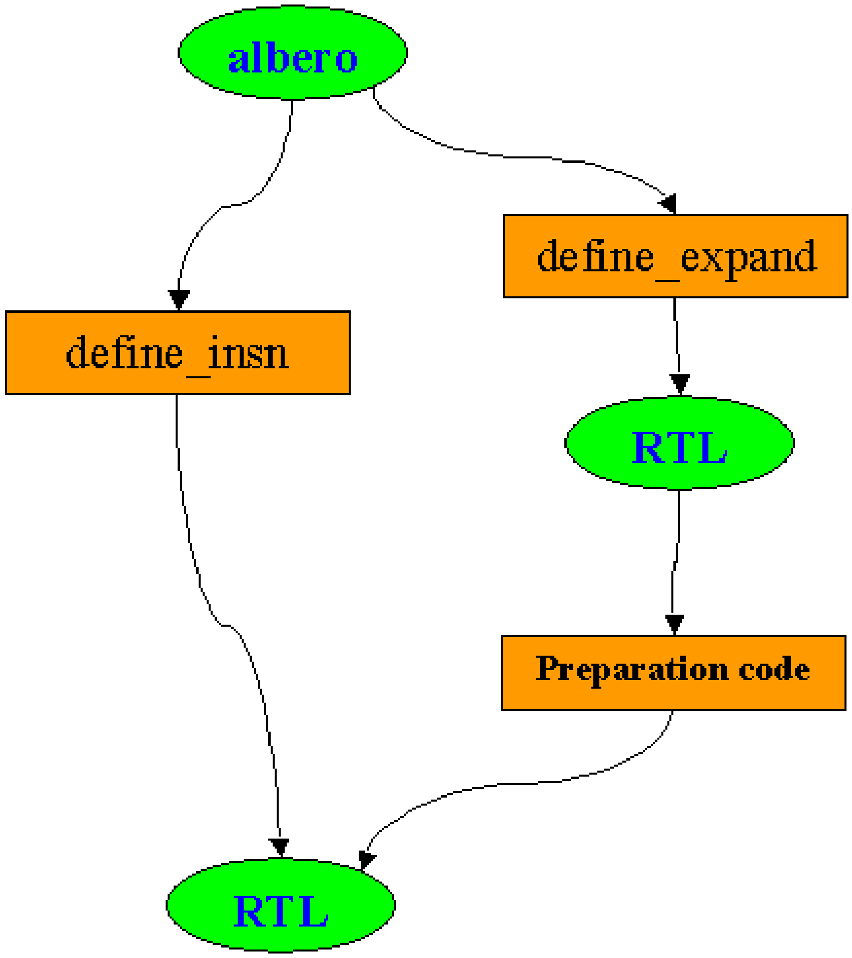

Figura 4.7: Generazione di codice RTL

APE si chiamano "normal". Una *normal* è una operazione del tipo `a*b+c` per un tipo di dato nativo del processore.

Non esistono quindi le operazioni matematiche "semplici" come l'addizione e la moltiplicazione. Per questo motivo gli operatori canonici "moltiplicazione" e "addizione" vengono implementati con delle *define_expand* che forzano gli operandi a risiedere nei registri. Vengono poi fornite sia la `define_insn`, `"normalsf"` che la `define_insn "hard_mulsf3"` e `define_insn "hard_addsf3"` che rispettivamente implementano la vera *normal* e le normal "finte" `a*b+0` `a*1+b`. Scrivendo prima nella machine description la normal "vere" e poi quelle "finte", il compilatore capisce che la prima è più efficiente delle seconde, e se riesce ad accorpare più operazioni in una *normal* emette il codice associato alla `define_insn "normalsf"`, altrimenti emette il codice associato ad una *normal* "finta". Una scelta oculata delle precedenze nella *machine description* porta quindi, a una maggiore efficienza delle istruzioni. Esempi di *define_insn* sono qui di seguito.

```
(define_insn "JSNORM_PP"
[(set (match_operand:SF 0 "register_operand" "=j")
     (plus:SF (mult:SF
                (match_operand:SF 1 "general_operand" "j")
(match_operand:SF 2 "general_operand" "j"))
  (match_operand:SF 3 "general_operand" "j")))]
""
```

```
"@
JSNORM_PP %0 %1 %2 %3")
(define_insn "JSNORM_stupid_mult"
[(set (match_operand:SF 0 "register_operand" "=j")
      (mult:SF (match_operand:SF 1 "general_operand" "%j")
       (match_operand:SF 2 "general_operand" "j")))]
""
"@
JSNORM_PP %0 %1 %2 0   !!stupid JS mult")
(define_insn "JSNORM_stupid_add"
[(set (match_operand:SF 0 "register_operand" "=j")
      (plus:SF (match_operand:SF 1 "general_operand" "%j")
       (match_operand:SF 2 "general_operand" "j")))]
""
"@
JSNORM_PP %0 %1 4 %2")
```

Le istruzioni RTL vanno lette così: tutto ciò che è dentro parentesi tonde rappresenta una (sotto)espressione RTL; le istruzioni possono avere degli operandi (`set`, `mult` e `plus` ne hanno 2) che si trovano subito dopo l'istruzione.

Nell'esempio l'istruzione "JSNORM_PP" sta per:

- J : istruzione per J1000

- S : single, operandi in singola precisione
- NORM: “normal”
- PP, operandi senza alterazioni di segno.

I numeri preceduti da percento indicano che in quel posto va emesso il valore dell’operando, nel nostro caso un registro. I numeri senza percento sono emessi così come sono, ed indicano quel preciso registro. I predicati “register_operand” e “general_operand” (secondo la terminologia GNU) indicano il tipo di operandi con cui si può generare una istruzione: nella fattispecie “register_operand” accetta solo registri, “general_operand” ogni tipo di operando.

L’ultima stringa all’interno di ogni operando rappresenta un vincolo, o *constraint*, ovvero un tipo di espressione RTL a cui bisogna convertire l’operando quando i registri virtuali vengono allocati nei registri fisici. Le lettere presenti in tali *constraint* rappresentano la classe di registri o il tipo di indirizzo di memoria da usare; il simbolo “=” comunica al compilatore che l’istruzione cambia l’operando in oggetto; il percento “%” indica che l’operazione tra l’operatore in oggetto e il successivo è commutativa.

4.3.2 Generazione di codice

Avendo introdotto la struttura generale della rappresentazione interna, il linguaggio RTL e i pattern di istruzioni presenti nella *machine description*, vediamo ora il funzionamento generale della generazione di codice macchina.

Come già detto, gli operatori della rappresentazione interna vengono *espansi* in espressioni RTL più o meno complesse, attraverso le `define_expand` e le `define_insn`. In questa fase i registri presenti nelle istruzioni RTL sono virtuali. A questo punto si eseguono molte delle ottimizzazioni che il compilatore GNU può effettuare, tra cui:

- ottimizzazione di salti
- eliminazione di sotto-espressioni comuni (*CSE*).
- eliminazione di sotto-espressioni comuni globali (*GCSE*).
- ottimizzazione dei cicli.
- analisi del flusso dei dati.
- combinazione delle istruzioni (*combiner*).
- ordinamento delle istruzioni (*scheduling*).
- scelta delle classi dei registri.

Alla fine di queste ottimizzazioni avviene la fase di "reload" (vedi paragrafo 4.3.4), durante la quale i registri virtuali vengono allocati in quelli fisici e si forzano gli operandi delle istruzioni RTL a soddisfare le *constraint* di cui sopra.

Durante tutti questi passi sono anche effettuate delle ottimizzazioni dipendenti dalla macchina, descritte nel porting. Esponiamo brevemente il funzionamento di queste fasi della compilazione

4.3.3 Ottimizzazioni indipendenti dalla macchina

Ottimizzazione di salti

In questa fase (*Jump Optimization* nella terminologia inglese GNU) vengono eseguite le seguenti ottimizzazioni:

- eliminazione di istruzioni di salto all'istruzione successiva.
- conversione di salti verso istruzioni che sono a loro volta dei salti.
- eliminazione di codice non raggiungibile (*dead code*).

La *Jump Optimization* viene eseguita due o tre volte durante tutta la compilazione. La prima volta alla fine della generazione del codice RTL; la seconda volta alla fine di *CSE*, ma solo se durante la fase *CSE* si operano dei cambiamenti che rendono necessaria un'altra ottimizzazione di salti; la terza volta subito prima dell'emissione del codice oggeto finale. In questa ultima occasione, oltre alle ottimizzazioni di salti citate sopra, vengono cancellate istruzioni *"NO-OP"* ("no operation").

CSE: Eliminazione delle sotto-espressioni comuni

Per sotto-espressioni comuni si intendono espressioni identiche che si trovano ripetute nel codice, e i cui operandi non subiscono cambiamenti tra le diverse occorrenze. Tali espressioni vengono calcolate solo una volta e memorizzate in un registro (se disponibile) o in una locazione di memoria. Tutti gli altri usi successivi di quella espressione accederanno direttamente al risultato memorizzato. Come esempio consideriamo il seguente codice prima (cse_1) e dopo (cse_2) le ottimizzazioni di questo passo.

```
//programma cse_1
complex function(complex rho, complex teta)
{
  float f = rho.im+2*teta.re+sqrt(rho.re);
  calcola_qualcosa(rho.im+2*teta.re,rho*f);
  return rho*teta/(rho.im+2*teta.re);
}
```

```
//programma cse_2
complex function(complex rho, complex teta)
{
  float cse_temp = rho.im+2*teta.re;
  float f = cse_temp+sqrt(rho.re);
  calcola_qualcosa(cse_temp,rho*f);
  return rho*teta/cse_temp;
}
```

GCSE: Eliminazione delle sotto-espressioni comuni globali

Questo modulo di ottimizzazione cerca delle sottoespressioni comuni globali, ovvero contenenti solo riferimenti a dati globali o costanti. Un esempio di questo passo di ottimizzazione si trova nei seguenti due codici: gcse_1, prima del passo GCSE; gcse_2, dopo il passo GCSE.

```
//programma gcse_1
const double pi = 3.14;
const double e = 2.718;
double num;
double function_1(double a)
{
   double x = 2*a-log(a)+exp(e,pi);
   where (x!=0)
      x = exp(e,pi)/x;
   elsewhere
      x = exp(e,pi);
   return x;
}
double function_2(double a)
{
   return function_1(a)+exp(e,pi);
}
```

```
//programma gcse_2
const double pi = 3.14;
```

```
const double e = 2.718;
double num;
double gcse_temp = exp(e,pi);
double function_1(double a)
{
   double x = 2*a-log(a)+gcse_temp;
   where (x!=0)
      x = gcse_temp/x;
   elsewhere
      x = gcse_temp;
   return x;
}
double function_2(double a)
{
   return function_1(a)+gcse_temp;
}
```

Ottimizzazione dei cicli

Durante questa fase, *Loop Optimization* nella terminologia GNU, vengono spostati all'esterno dei cicli i calcoli di espressioni indipendenti dalle iterazioni e vengono *srotolati* i cicli laddove ciò è possibile.

Lo srotolamento, o *unroll*, è una tecnica molto usata e molto efficace per ottimizzare i programmi, a discapito di una loro maggiore grandezza. Tale tecnica consiste nel riscrivere le istruzioni di più iterazioni, cosicché si possa avere maggior possibilità di ottimizzare il codice e di riempire tutte le sottounità del processore.

Alla fine di questo modulo di ottimizzazione, se richiesto dal programmatore, può essere eseguito di nuovo il modulo CSE.

Analisi del flusso dei dati

In questo modulo viene studiato l'utilizzo dei registri virtuali (la prima e l'ultima istruzione che li usa), vengono eliminate istruzioni i cui risultati non vengono mai utilizzati, viene tenuta traccia di tutti i registri virtuali utilizzati ad ogni istante del programma.

Combinazione delle istruzioni

Nell'esempio fatto per le *normal* di APEmille si è citata la capacità del compilatore GNU di ricombinare le istruzioni seguendo le indicazioni date nella *machine description* e nelle macro dell'interfaccia di porting sulle temporizzazioni delle istruzioni. Questo modulo, il *combiner*, ha proprio questo compito: riassemblare le istruzioni in modo che l'esecuzione sia il più efficiente possibile.

scheduling

Questo passo (*scheduler*) cerca istruzioni il cui risultato non sia disponibile per la prima (seguente) istruzione che ne necessita. Queste informazioni sono date al compilatore tramite l'interfaccia di porting. Lo *scheduler* cercherà, se possibile, di spostare le istruzioni in questione in modo da rendere disponibile il risultato al momento opportuno.

Nell'architettura APEmille questa attività è lasciata ad un modulo specifico della catena di compilazione: lo *shaker*. Non ci sono, dunque, informazioni sulla temporizzazione dei risultati nel porting per APEmille.

Lo scheduling viene ripetuto alla fine del passo di *reload* (vedi paragrafo 4.3.4).

Scelta delle classi di registri

Le *constraint* esposte brevemente prima vengono risolte in questa fase: esse, come detto, esprimono l'*assoluta* necessità per una istruzione di avere ogni operando di una determinata classe di registri. Se ciò non è vero per qualche istruzione, il compilatore crea delle copie temporanee dei dati che soddisfano le *constraint* richieste dalle istruzioni. Un esempio può essere il caso delle già citate *normal*, che possono essere eseguite *solo* con operandi che risiedono nei registri: se il compilatore deve generare codice per una istruzione RTL che opera una *normal* su dati che sono in memoria, viene creata una copia temporanea di quei dati in dei registri, e poi viene eseguita la *normal*. Questo si ottiene definendo la *normal* in modo da includere i necessari *constraint*, come segue:

```
(define_insn "JSNORM_PP"
[(set (match_operand:SF 0 "register_operand" "=j")
```

```
      (plus:SF (mult:SF (match_operand:SF 1 "general_operand" "j")
(match_operand:SF 2 "general_operand" "j"))
       (match_operand:SF 3 "general_operand" "j")))]
""
"@
JSNORM_PP %0 %1 %2 %3")
```

Qui di seguito viene presentato un esempio di questa situazione dove `mem_var` è una variabile `float` in memoria, nei registri fisici 45 e 46 si suppone ci siano risultati di calcoli eseguiti in precedenza o altre variabili già caricate dalla memoria. Nel registro fisico 44 andrà il risultato della *normal*, mentre il registro fisico 47 è quello utilizzato per una copia temporanea della variabile `mem_var`. Prima di applicare le *constraint* abbiamo per esempio la seguente espressione RTL.

```
(set (reg:SF 44)
     (plus:SF (mult:SF (mem:SF (symbol_ref "mem_var"))
                       (reg:SF 54))
              (reg:SF 90)))
```

Dopo aver applicato le *constraint* la precedente espressione RTL viene trasformata nella seguente:

```
(set (reg:SF 47) (mem:SF (symbol_ref "mem_var")))
(set (reg:SF 44)
```

```
(plus:SF (mult:SF (reg:SF 47)
                  (reg:SF 45))
         (reg:SF 46)))
```

Le *constraint* "`j`" indicano la classe di registri appartenenti a J1000. Il modello GNU infatti prevede un solo register file, mentre per modellare l'architettura di APEmille si ha il bisogno di due register file: uno per T1000, uno per J1000. Per raggiungere questo scopo abbiamo introdotto due classi di regsitri, "`t`" per T1000 e "`j`" per J1000.

Specificando *constraint* opportune per i vari modi macchina (che, ricordiamo, sono stati in questo porting associati a T1000 o a J1000) riusciamo a comunicare al compilatore GNU l'opportuno registro dove caricare i dati.

4.3.4 Reload

Durante questa fase i registri virtuali vengono *allocati* in registri fisici. Questo può generare la necessità di avere copie temporanee di dati se il numero di registri fisici è troppo esiguo: tali copie vengono fatte in memoria in quello che si chiama *stack*. Notiamo che questo non accade "quasi" mai in APEmille, dato il numero abbondante di registri; infatti T1000 dispone di 256 registri mentre J1000 di 512.

4.3.5 Ottimizzazioni dipendenti dalla macchina

In questo paragrafo verranno esaminate le informazioni relative a ottimizzazioni dipendenti dalla macchina che possono essere fornite nel porting del compilatore GNU per un nuovo processore. Queste informazioni non sono usate in un unico vero e proprio passo di compilazione, bensì vengono usate in molti dei passi citati sopra per rifinire le ottimizzazioni indipendenti dalla macchina. In verità anche le semplici informazioni sul numero dei registri, sulle classi di registri, sulle istruzioni RTL disponibili nella macchina, sui tipi nativamente gestiti dalla macchina, etc... alterano sensibilmente il modo con cui si genera il codice macchina.

Le informazioni che si comunicano al compilatore GNU che possono influenzare le suddette ottimizzazioni sono:

- il costo di istruzioni RTL.
- autoincremento negli indirizzamenti.
- possibilità e quantità di spostamenti di blocchi di dati.
- costo relativo di pattern di istruzioni nella *machine description*.
- suddivisione di istruzioni in multiple istruzioni.
- possibilità di raggruppare più istruzioni in un'unica istruzione.

Analizziamo con degli esempi ciascuna informazione.

Costo relativo di istruzioni RTL

L'interfaccia del porting comprende una serie di macro che devono essere fornite al compilatore GNU e che descrivono il costo relativo di operazioni sulla

Macro	Significato
CONST_COSTS(x,code)	Costo dell'espressione CONSTANT RTL "x" presente nell'espressione RTL "code"
RTX_COSTS(x,code)	Costo dell'espressione non CONSTANT RTL "x" presente nell'espressione RTL "code"
DEFAULT_RTX_COSTS (x,code)	Costo dell'espressione "x" presente nell'espressione RTL "code", solo se non fornito da RTX_COSTS
ADDRESS_COSTS(address)	Costo di un indirizzamento all'indirizzo "address"
REGISTER_MOVE_COST (from,to)	Costo per spostare dati dal registro "from" al registro "to"
MEMORY_MOVE_COST (mode,class,in)	Costo per spostare dati in modo macchina "mode" tra un registro di classe "class" e memoria: in vale "zero" se si scrive in memoria, "non-zero" altrimenti
BRANCH_COST	Costo di una istruzione di branch

Figura 4.8: Costi relativi di espressioni RTL

macchina per cui si scrive il porting. Queste macro con la relativa descrizione sono in Tabelle 4.8.

Autoincremento negli indirizzamenti

Per autoincremento si intende uno tra "pre-incremento","post-incremento","pre-decremento" o "post-decremento", ovvero la possibilità per la macchina di incrementare o decrementare un registro o una locazione di memoria subito prima o subito dopo aver eseguito un accesso in memoria, senza eseguire una istruzione esplicita di somma o sottrazione.

Una serie di macro, disponibili nell'interfaccia, istruisce il compilatore GNU sulla efficienza o meno di autoincrementi per ogni determinato modo.

Un esempio:

Macro	Significato
MOVE_RATIO	Numero di istruzioni macchina “move” memoria-memoria al di sotto del quale è conveniente generare più istruzioni invece di una “move” di un blocco
MOVE_MAX_PIECES	Grandezza della massima unità di memoria che una “load” o una “store” può spostare

Figura 4.9: Macro relative a spostamenti di blocchi di dati

```
#define USE_LOAD_PRE_INCREMENT(mode)          0
```

Nell’esempio sopra, si comunica al compilatore che per ogni modo macchina `mode` non esiste la possibilità di usare un pre-incremento. Questo è per esempio vero nell’architettura di APEmille, dove non esiste possibilità di autoincremento.

Spostamento di blocchi di dati

Ci sono 3 macro presenti nell’interfaccia di porting che specificano se e quando è possibile spostare grandi blocchi di dati in modo efficiente. Se ciò è possibile, viene cercata nella *machine description* un pattern con nome predefinito: movstr$m$, dove m è il modo macchina associato ai dati da trasferire. Le macro di cui sopra sono esposte in Tabella 4.9.

Costo relativo dei pattern di istruzioni

È stato già detto che la precedenza dei pattern nella *machine description* corrisponde per il compilatore GNU ad una maggiore efficienza, ovvero ad un minore tempo di esecuzione.

Splitting: suddivisione di istruzioni

Nella *machine description* è possibile fornire, oltre ai pattern di istruzioni `define_insn` e `define_expand`, dei pattern che dividono (*split*) singole istruzioni in più istruzioni, laddove questo rende il codice più efficiente.

Il costrutto usato nella *machine description* è la `define_split`. La sua sintassi è:

```
(define_split
[insn-pattern]
"condition"
[nuovo-insn-pattern-1
 nuovo-insn-pattern-2
 ...]
"statement di preparazione")
```

`insn-pattern` rappresenta l'istruzione RTL da, eventualmente, dividere; en-condition è una condizione che deve essere soddisfatta affinchè l'istruzione (che soddisfa il pattern *insn-pattern*) venga divisa; i vari *nuovo-insn-pattern-x* sono una sequenza di istruzioni RTL in cui viene spezzata l'istruzione in oggetto;

gli statement di preparazione (come per le `define_expand`) hanno il compito di forzare gli operandi, memorizzare informazioni o altro, e vengono eseguiti se l'istruzione in oggetto soddisfa il pattern della `define_split` e la condizione `condition` è soddisfatta.

Nel porting di APEmille non è stato necessario fornire nessuna `define_split`.

Raggruppamento di molteplici istruzioni in un'unica istruzione

Il costrutto `define_peephole` nella *machine description* fornisce al compilatore GNU informazioni e codice necessario ad accorpare più istruzioni insieme, quando questo rende il codice più efficiente.

Diamo un esempio di raggruppamento fornito nel porting di APEmille.

```
(define_peephole
[(set (match_operand:SI 0 "register_operand" "=t")
(match_operand:SI 1 "immediate_operand" "n"))
(set (match_operand:SI 2 "register_operand" "=t")
(mem:SI (plus:SI (match_dup 0)
(match_operand:SI 3 "register_operand" "t")))) ]
""
"@
MEMTOT2 %2 %1.%3")
```

Nell'esempio ci sono due istruzioni RTL: la prima è il caricamento di un registro con un immediato, ovvero una costante numerica nota al momento della compilazione; la seconda è un caricamento dalla memoria di un dato tramite un indirizzo calcolato comme somma di due registri (un `MOP3` con indirizzo immediato "zero" nella terminologia di APEmille). Uno dei due registri è però il registro precedentemente assegnato con una costante nota al momento della compilazione. Questo risulta essere uno spreco di un registro, anche per poche istruzioni, e può essere ottimizzato.

Infatti, la `define_peephole` genera una istruzione di caricamento dalla memoria con un `MOP2` (indirizzo immediato più registro) liberando così il registro caricato con un immediato ed utilizzando invece l'immediato stesso.

Queste ottimizzazioni vengono eeguite nel modulo *combiner*.

Capitolo 5

Porting GNU per APEmille

Per implementare il porting di un compilatore GNU per una specifica macchina, GNU mette a disposizione una interfaccia con cui si può modificare il comportamento del compilatore, e con cui si danno allo stesso compilatore informazioni indispensabili per generare correttamente del codice macchina.

Questa interfaccia è rappresentata da due file principali: *target.h* e *target.md*. *target.h* contiene macro che descrivono le caratteristiche della macchina; *target.md* contiene i pattern delle istruzioni, sia le `define_expand` che le `define_insn`. Facciamo notare che sia i pattern che le macro possono contenere codice C: questo permette elaborazioni complesse che permettono di alterare il processo di compilazione per adattarlo alla macchina in oggetto.

Di seguito per "interfaccia" si intenderanno le informazioni fornite nei due file descritti sopra.

Ci sono delle limitazioni evidenti e insormontabili per implementare i linguaggi SIMD C e SIMD C++ con un porting "ufficiale"[1] del compilatore

[1] Per "ufficiale" si intende senza modifiche del codice sorgente del compilatore GNU

GNU.

Riassumiamo brevemente le più importanti:

- impossibilità per GNU di allocare a *compile-time* le variabili, mentre in APEmille è necessario per ragioni di efficienza.

- GNU è esplicitamente costruito per processori che indirizzano la memoria con unità di 8 bit (o *byte*), mentre le unità elementari di accesso di APEmille sono di 32 bit.

- necessità di gestire nativamente i tipi `vector` e `localint`.

- gestione delle condizioni locali (`where`) e condizioni globali.

- supporto nativo alle funzioni matematiche $1/x$, log, exp,$1/sqrt$ come avviene in APEmille.

- gestione consistente della comunicazione tra i nodi di calcolo in stile APEmille.

Il lavoro originale di questa tesi è stato anche quello di implementare in modo efficiente il porting del compilatore GNU sulla architettura di APEmille, oltre che per sfruttarne la potenza di calcolo. Particolare studio è stato richiesto per alcuni aspetti del porting, in cui la complessità dell'architettura di APEmille o la sua diversitá dal modello GNU sono state particolarmente importanti.

Un elenco delle implementazioni più importanti effettuate è il seguente:

- gestione delle comparazioni e dei salti condizionati.
- gestione della corrispondenza dei banchi pari e dispari di memoria e dei registri.
- gestione dello *stack*.
- gestione dello accesso in *burst* dei processori J1000 di APEmille.

In questo capitolo descriviamo l'implentazione del porting per quanto riguarda l'interfaccia standard di GNU, lasciando al prossimo capitolo la trattazione delle estensioni apportate al modello GNU.

5.1 Comparazioni e salti condizionati

Nel paragrafo 4.2.1 è stato spiegato il modello di gestione delle comparazioni e dei salti condizionati del compilatore GNU: esso è basato sul concetto di *Condition Code*, o CC. Nel CC viene prima memorizzato il risultato di una comparazione generica tra due operandi; lo stato del CC viene poi letto da una istruzione che, condizionata appunto dal CC, esegue un salto in un altro punto del programma.

Essendo questo modello diverso dal funzionamento di APEmille, esponiamo qui l'implementazione adottata nel porting in oggetto. Ricordiamo che nel calcolatore APEmille, esistono 6 diverse istruzioni che testano le 6 comparazioni elencate nella tabella4.2. Ognuna di loro verifica la condizione associata

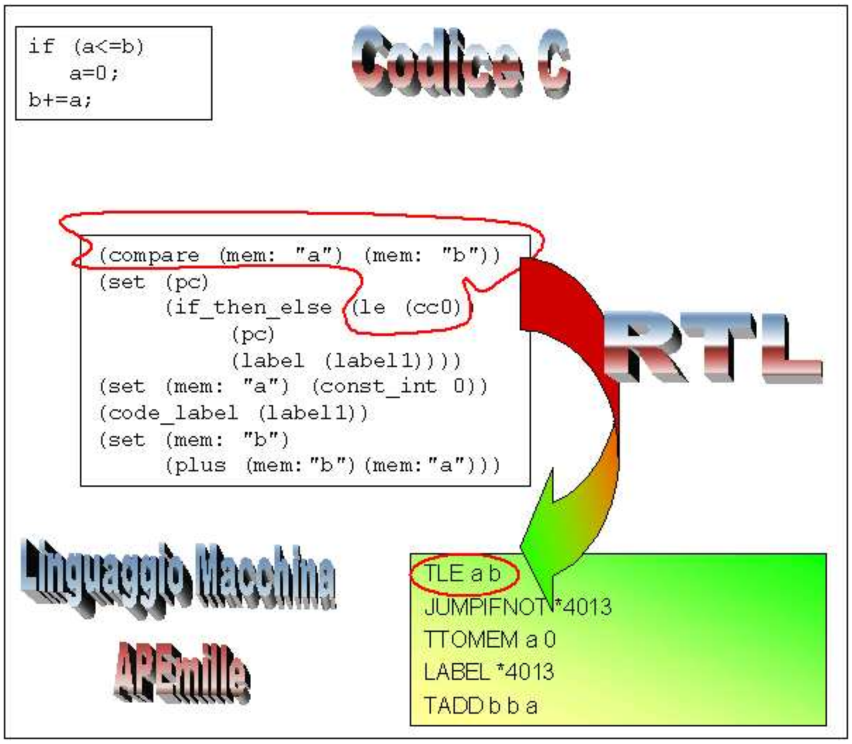

Figura 5.1: Esempio di compilazione di un salto condizionato

e imposta conseguentemente il CC di APEmille, che consiste perciò di un solo bit. Una successiva istruzione di salto condizionato eseguirà o no il salto condizionatamente dal CC di APEmille. Le istruzioni di salto condizionato di APEmille sono `JUMPIF` e `JUMPIFNOT`: la prima esegue il salto se il CC contiene il valore "vero", la seconda se il CC contiene "falso".

Un esempio della conversione dal modello GNU al modello APE della gestione deli salti condizionati è visibile in fig.5.1.

La generica comparazione tra due operandi del modello GNU viene tradotta nell'istruzione `compare` del linguaggio RTL, presente nella *machine de-*

scription in tanti pattern quanti sono i modi macchina. In questi pattern si memorizzano solo gli operandi, senza produrre nessun codice macchina o altre istruzioni RTL: l'unica azione è quella (solita nelle architetture RISC) di forzare gli operandi a risiedere nei registri. Successive istruzioni di salto condizionato (che nel modello GNU non hanno informazioni sugli operandi in quanto verificano solo il CC!) potranno finalmente emettere codice macchina esatto: conoscono infatti sia il tipo di comparazione da effettuare sia gli operandi, precedentemente memorizzati.

Qui di seguito esponiamo il pattern corrispondente alla generica comparazione tra dati di modo macchina `SI`, canonicamente associato al tipo `int`. Di seguito il pattern `ble` corrispondente al salto condizionato dalla comparazione "minore o uguale a".

```
(define_expand "cmpsi"
[(set (cc0)
      (compare:SI (match_operand:SI 0 "general_operand" "")
          (match_operand:SI 1 "general_operand" "")))]
""
"{
   extern rtx APEmille_compare_op0, APEmille_compare_op1;
   if (GET_CODE(operands[0]) != REG)
      operands[0] = force_reg(SImode, operands[0]);
   APEmille_compare_op0 = operands[0];
```

```
    if (GET_CODE(operands[1]) != REG)
       operands[1] = force_reg(SImode, operands[1]);
    APEmille_compare_op1 = operands[1];
   DONE;
 }")
(define_expand "ble"
[(set (pc)
      (if_then_else (le (match_dup 1) (const_int 0))
         (label_ref (match_operand:SI 0 "" ""))
  (pc)))]
   ""
   "{ extern rtx APEmille_compare_op0, APEmille_compare_op1;
   if (IS_T1000_MODE(GET_MODE (APEmille_compare_op0)))
   {
       emit_insn (gen_cmpLEsi (APEmille_compare_op0,
                   APEmille_compare_op1));
       emit_jump_insn (gen_cbranch (operands[0]));
       DONE;
    }
    fatal("Floating point or localint comparisons"
          " not allowed outsideg lobalizing keywords "
```

```
        "(where, any, all, none)\n");
}")
```

Notiamo di passaggio che non sono ammessi salti condizionati su condizioni associate a tipi floating point o localint (tipi di dati di J1000).

Diamo un esempio di un salto condizionato da una espressione intera (associata quindi a T1000) in linguaggio SIMD C e la relativa traduzione in linguaggio macchina.

```
int i,j;
if (i<=j)
  se_vero();
```

```
MEMTOT1  44 i
MEMTOT1  45 j
TLE  44 45
JUMPIFNOT *4100
.....
GOSUB se_vero
.....
LABEL *4100
```

Per consentire una lettura accettabile sono stati indicati gli indirizzi delle variabili con i loro stessi nomi, e sono state omesse le istruzioni di preparazione alla chiamata della funzione `se_vero` e quelle di riallineamento dello stack che seguono la stessa chiamata.

5.2 Corrispondenza banchi pari e dispari tra memoria e Register File

Una caratteristica avanzata dell'architettura del calcolatore APEmille è quella di dividere sia la memoria che il register file in due banchi, rispettivamente memoria "pari" e "dispari" e register file "pari" e "dispari", e di poter trasferire ad ogni ciclo di clock una parola da 32 bit nelle due direzioni:

- memoria pari - register file pari.
- memoria dispari - register file dispari.

Questo permette tra l'altro la possibilità di iniziare ad ongi ciclo di clock una operazione normal tra dati *complex* o *vector*, leggendo contemporaneamente entrambe le componenti.

Il vantaggio di efficienza si paga con una maggiore complessità dell'architettura, in quanto non è possibile trasferire direttamente una parola tra banchi di diversa parità. Per questo tipo di trasferimenti si deve utilizzare un registro di appoggio della giusta parità.

L'esempio più comune di questa situazione si incontra assegnando la parte reale di un complesso con quella immaginaria di un altro complesso (vedi programma di seguito). Per definizione, i dati complessi sono allineati in memoria e nel register file in modo che la parte reale sia *sempre* sul banco pari e quella immaginaria *sempre* su quello dispari.

```
register complex a;  // variabile a nei registri
complex b;           // variabile b in memoria
...
a.im = b.im;         // banco dispari in banco dispari
a.re = b.im;         // banco dispari in banco pari !!
```

Dovendo quindi copiare, per esempio, la parte immaginaria di una variabile complesso allocata in memoria nella parte reale di una variabile complessa residente in una coppia di registri, il compilatore deve prima copiare la parte immaginaria (banco dispari) in un registro d'appoggio nel banco dispari, e successivamente spostarlo nella parte reale (banco pari) destinazione. Copiare i dati all'interno del register file è sempre possibile, anche tra registri di banchi diversi.

```
// supponiamo a nella coppia di registri (50,51)
//   e b residente in memoria all'indirizzo 0x1230
// a.im = b.im
MEMTOJ1 51 0x1231
// a.re = b.im
MEMTOJ1 17 0x1231     // registro 17 di appoggio
JMOV    50 17         // JMOV: sposta registri di J1000
```

L'implementazione di questa tecnica risiede nel pattern dell'istruzione `movsf` nella *machine description*, ovvero quella canonicamente associata alla copia di dati in virgola mobile in singola precisione, modo macchina "Single Float" (`SF`) tipo `float` dei linguaggi SIMD C e C++. Questo perché le singole componenti di dati di tipo `complex` o `vector` sono in realtà dei `float`.

Riportiamo di seguito per completezza il codice dei pattern `movsf` e `hard_movsf` presenti nella machine description: tale codice non è di facile lettura, perciò riportiamo di seguito uno schema logico che riassume le caratteristiche fondamentali dell'implementazione, trascurando i dettagli.

```
(define_expand "movsf"
[(set  (match_operand:SF 0 "" "")
 (match_operand:SF 1 "" ""))]
""
"{
   extern rtx reduce_symbols_to_numbers(rtx);
   operands[1] = reduce_symbols_to_numbers(operands[1]);
   if(GET_CODE(operands[0])==MEM&&GET_CODE(operands[1])!=REG)
      operands[1] = force_reg (SFmode, operands[1]);
}")
(define_insn "hard_movsf"
[(set (match_operand:SF 0 "general_operand" "=m,=j,=j,=j")
      (match_operand:SF 1 "general_operand" "j,m,j,EF"))]
```

```
""
"* {
switch (which_alternative)
{
case 2:
    return "JSNORM_PP \t%0 %1 4 0\t\t";
      case 3:
    return "TTOJ \t\t%0 %1.0.0";
  case 0: case 1:    // memory <--> register file
{
 int scratch_reg = -1;
 rtx addr = XEXP(operands[which_alternative],0);
 char number = (address_is_MOP1(SFmode, addr) ? \'1\' :
      address_is_MOP2(SFmode, addr) ? \'2\' :
      address_is_MOP3(SFmode, addr) ? \'3\' : \'x\');
 char *instruction = (which_alternative==0
       ? "JTOMEM" :"MEMTOJ");
 char presimbol0 = (which_alternative==0 ? \'^\' : \' \');
 char presimbol1 = (which_alternative==0 ? \' \' : \'^\' );
 int dest = parity_of_operand(operands[0]);
 int source = parity_of_operand(operands[1]);
 int s1,s0;
```

```
 static char pre_adjust[32],post_adjust[32],
    final_buffer[128];
 pre_adjust[0] = post_adjust[0] = 0;
 if (number == \'x\' || source==-1 || dest==-1)
   fatal("(APE): hard_movsf \n");
   if (dest==source)
   { if (which_alternative==0)   //  mem <-- RF
       sprintf(final_buffer,
        "%s%c\\t\\t^%%0\\t %%1", instruction, number);
      else if (which_alternative==1)   //  RF <-- mem
       sprintf(final_buffer,
        "%s%c\\t\\t%%0\\t ^%%1",instruction,number);}
   else {
      if (which_alternative==0)   //  mem <-- RF
      {
        int temp = (dest%2==1) ?
            J1000_DISALIGNED_REG : J1000_ALIGNED_REG;
sprintf(final_buffer,
           "JSLUTMOVE \t%d \t%%1\n\t", temp);
        sprintf(final_buffer,"%s%s%c\t\t^%%0\t %d"
           ,final_buffer,instruction, number, temp);
      }
```

```
      else                          //  RF  <-- mem
      {
        int temp = (dest%2==0) ?
          J1000_DISALIGNED_REG : J1000_ALIGNED_REG;
        sprintf(final_buffer,"%s%c\t\t%d\t %%1\n\t",
             instruction,number, temp);
sprintf(final_buffer,"%sJSLUTMOVE \t%%0 \t%d",
             final_buffer, temp);
      }
   }
  return final_buffer;
}}}")
```

Il pattern `movsf` converte i riferimenti a variabili presenti nell'espressione RTL dell'operando 1 (sorgente della operazione di copia) con gli indirizzi loro corrispondenti (vedi paragrafo 6.1) e forza lo stesso operando a risiedere in un registro.

Notiamo che a questo punto *non si conoscono i registri fisici* su cui risiedono gli operandi: l'allocazione dei registri da virtuali a fisici viene infatti eseguita durante la fase di *reload* (vedi paragrafo 4.3.4).

Il secondo pattern, `hard_movsf`, viene utilizzato dal compilatore GNU *dopo* la fase di reload per generare il codice macchina finale: a questo punto si cono-

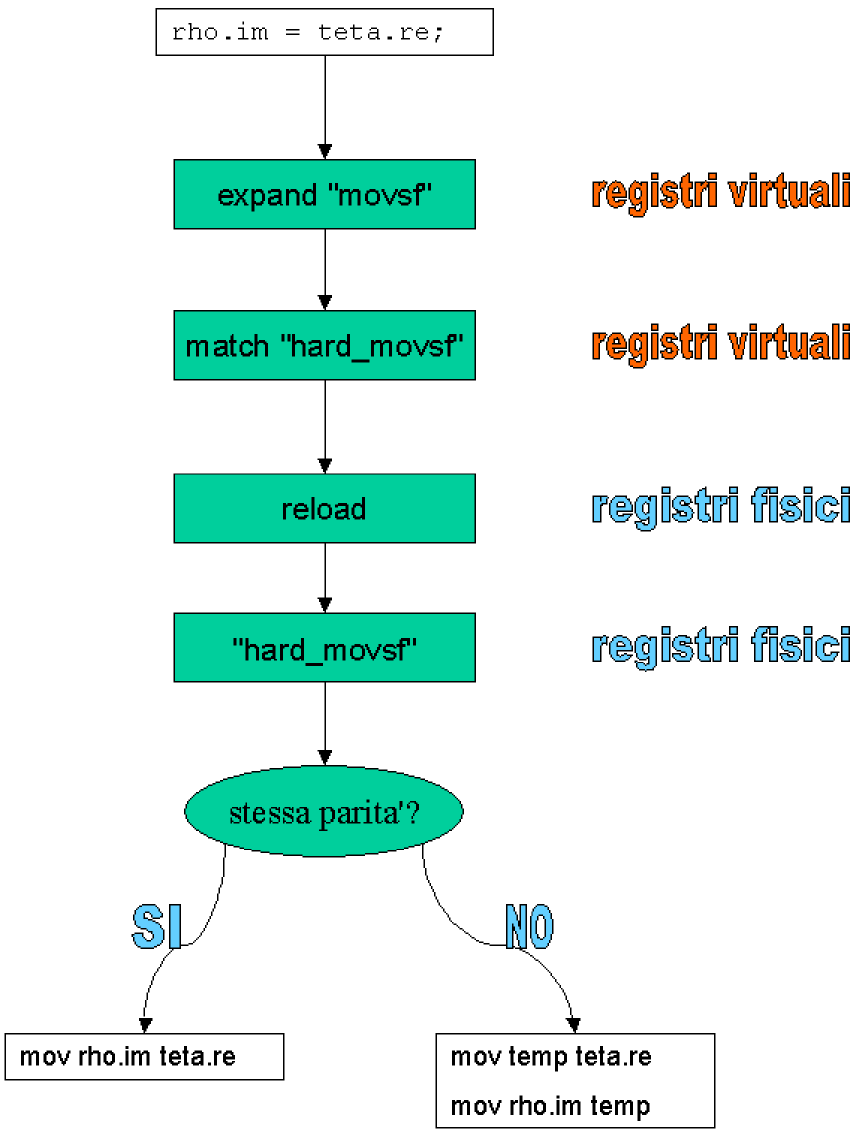

Figura 5.2: Sequenza temporale della gestione della parità

scono i registri fisici in cui risiedono gli operandi e si può quindi implementare la tecnologia della separazione dei banchi (vedi fig.5.2).

In `hard_movsf` ci sono 4 possibili istruzioni di copia RTL riconosciute, ognuna associata a una coppia di *constraint* (una per ogni operando):

1. da regsitro a memoria
2. da memoria a registro
3. da registro a registro
4. da costante a registro

La variabile `which_alternative` durante l'esecuione del programma contiene un numero che indica, appunto, quale in alternativa tra le 4 possibili rientra l'istruzione RTL che si sta convertendo in linguaggio macchina.

Le ultime due alternative (corrispondenti ai valori 2 e 3 di `which_alternative`) sono abbastanza semplici da generare: sono semplici istruzioni di "move" tra registri o trasferimento da T1000 a J1000 di costanti. Notiamo che la "move" tra registri di J1000 è implementata usando una normal banale, ovvero come $a = 1 * b + 0$: non esiste infatti una istruzione dedicata di "move" in APEmille.

Nel caso di "move" tra registri e memoria vengono calcolate le parità dei due operandi, sorgente e destinazione, e in caso di parità differenti viene utilizzato un registro d'appoggio come spiegato sopra. I due registri d'appoggio predefiniti sono `J1000_ALIGNED_REG` e `J1000_DISALIGNED_REG` rispettivamente nel banco pari e dispari. Queste sono semplici macro riconfigurabili.

5.3 Gestione dello stack

Lo stack è quella zona di memoria usata per l'allocazione di variabili locali e per il passaggio di argomenti a funzioni o procedure laddove non si possa o non si voglia usare i registri.

Esso cambia in grandezza durante l'esecuzione del programma e l'indirizzo esatto della cima (*top of the stack*) non è conosciuto durante la compilazione.

Lo stack infatti viene indirizzato tramite un registo speciale: il cosidetto *stack pointer* (registro di seguito indicato con SP), che memorizza in ogni istante l'indirizzo del top dello stack (o prima locazione libera seguente lo stack, a seconda delle implementazioni).

Lo stack può "crescere" (come grandezza) verso valori più grandi o più piccoli: nell'architettura APE si è scelto di farlo "crescere" verso "l'alto", ovvero verso valori più grandi.

Allocare spazio sullo stack significa dunque aumentare lo stack pointer, mentre deallocare significa decrementarlo.

Lo stack viene usato in APEmille per i seguenti scopi:

- allocare variabili locali *register*.
- passaggio di argomenti tra funzioni (in mancanza di registri, o su indicazione del programmatore).
- salvataggio dei registri che indirizzano lo stack stesso durante chiamate a funzioni.

Le variabili locali allocate sullo stack vengono indirizzate tramite un registro dedicato a questo scopo: il *frame pointer*, o FP. I parametri passati ad una funzione vengono acceduti da questa tramite un registro dedicato: l'*argument pointer*, o ARG.

Riassumiamo in tabella5.1 i registri che indirizzano lo stack e il loro significato. Notiamo che una ottimizzazione del compilatore GNU prevede la

Tabella 5.1: Registri che indirizzano lo stack

Registro	Nome Esteso	Indirizzo a cui punta
SP	Stack Pointer	prima locazione libera sullo stack
FP	Frame Pointer	indirizzo più basso del Frame
ARG	Arguments Pointer	indirizzo primo argomento passato

possibilità di usare solo 2 o 1 di tali registri in funzioni in cui ciò sia possibile: in questo lavoro è stato scelto di tralasciare questa ottimizzazione in quanto non influente nei normali programmi per APEmille. Infatti la maggior parte delle variabili e dei parametri possono risiedere nei registri grazie al regsiter file di grandi dimensioni (512 registri da 32 bit) di APEmille, non c'è quindi bisogno (nella maggior parte dei casi) dei regsitri FP e ARG.

Nel caso più generale, comunque, l'utilizzo dello stack in una generica funzione che richiede argomenti passati sullo stack e variabili locali allocate sullo stack è riportato in fig.5.3.

All'inizio ed alla fine di una generica funzione di sono due blocchi di codice chiamati *prologo* ed *epilogo*. Nel prologo vengono inizializzati i registri che

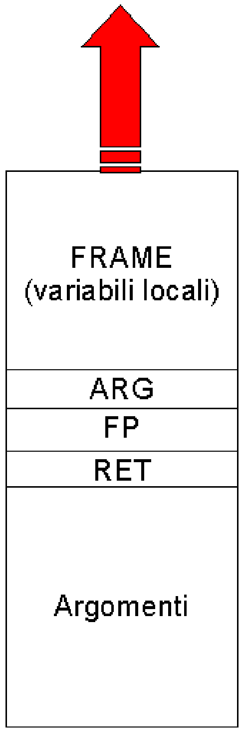

Figura 5.3: Layout dello stack dopo il prologo

indirizzano lo stack in modo consistente con la funzione in oggetto; nell'epilogo vengono ripristinati gli stessi registri con i valori precedentemente avuti prima del prologo: tali valori vengono memorizzati temporaneamente nello stesso stack.

5.4 Limiti del porting GNU per APEmille

I moduli di ottimizzazione dei GNU accettano solo le istruzioni RTL originali, cioé non riconoscono quelle aggiunte in questo lavoro. In alcuni punti estendere la logica dei moduli di ottimizzazione per gestire correttamente le nuove istruzioni è banale, in altri estremamente complesso. Quasi nessuna ottimizzazione è stata completamente portata ad un livello di implementazione, di affidabilità e di efficienza tali da poter consentire un test veritiero delle performance del compilatore GNU per APEmille.

5.4.1 Limiti del compilatore SIMD C++ per APEmille

Il compilatore SIMD C++ per APEmille è in uno stato embrionale e incompleto, a differenza del compilatore SIMD C. Il linguaggio SIMD C include, infatti, tutte le estensioni necessarie a sfruttare APEmille: è stato prioritario in questo progetto sperimentare la fattibilità e il costo (in termini di tempo e risorse) di un compilatore C. Gli unici programmi compilati in SIMD C++ sono stati dei programmi molto semplici.

Il linguaggio SIMD C++, così come qui esposto, prevede la gestione della classi basi virtuali tramite la duplicazione dei puntatori nascosti tradizionali. Questo per tener conto delle due parti che costituiscono un generico oggetto durante l'esecuzione del programma: la parte T1000 (CP) e la parte J1000 (NP). L'implementazione di questa caratteristica non è stata effettuata in questo lavoro, in cui è stata data precedenza a componenti del linguaggio più utili per poter scrivere da subito programmi di fisica in grado di sfruttare APEmille: non è possibile quindi definire classi basi virtuali nel compilatore SIMD C++ per APEmille.

Capitolo 6

Estensione del modello GNU per APEmille

In questo paragrafo descriviamo sia le estensioni apportate al modello GNU per poter implementare i linguaggi SIMD C e SIMD C++ e sfruttare la potenza di APEmille, sia le particolari tecnologie inventate per poter generare in modo semplice ed elegante codice macchina di APEmille con il compilatore GNU.

Ci sono delle limitazioni evidenti e insormontabili per implementare i linguaggi SIMD C e SIMD C++ con un porting ufficiale del compilatore GNU.

Riassumiamo brevemente le più importanti:

- impossibilità per GNU di allocare a *compile-time* le variabili, mentre in APEmille è necessario per ragioni di efficienza.

- GNU è esplicitamente costruito per processori che indirizzano la memoria con unità di 8 bit (o *byte*), mentre le unità elementari di accesso di APEmille sono di 32 bit.

- accesso alla memoria J1000 in *burst*.
- necessità di gestire nativamente i tipi `vector` e `localint`.
- gestione delle condizioni locali (`where`) e condizioni globali.
- supporto nativo alle funzioni matematiche [1] $1/x$, log, exp e $1/sqrt$ come avviene in APEmille.
- gestione consistente della comunicazione tra i nodi di calcolo in stile APEmille.

6.1 Allocazione statica della memoria

Questo meccanismo prevede una allocazione statica (ovvero durante la compilazione del programma) delle variabili globali e delle variabili locali (dichiarate dentro il corpo di una funzione) dichiarata con la keyowrd `static`. L'implementazione consiste di tre componenti chiave:

1. allocatore variabili.
2. convertitore simboli in indirizzi.
3. gestore degli "scope".

[1] La funzione $sqrt(x)$ è ovviamente ricavabile componendo la funzione inversa con la funzione $1/sqrt$

6.1.1 Allocatore

L'allocatore decide dove allocare le variabili, sulla memoria di T1000, di J1000 o di entrambi (nel caso di strutture o classi miste), e alloca fisicamente la memoria, associando al nome della variabile in oggetto l'indirizzo di memoria allocata.

Per consentire una facile migrazione a nuove versioni del compilatore GNU, si è deciso di usare una possibilità offerta dall'interfaccia verso il porting offerta da GNU, la macro `SET_DEFAULT_DECL_ATTRIBUTES` definita nel porting, che il compilatore GNU invoca ad ogni dichiarazione di variabile o funzione. Lo scopo della macro è di permettere l'impostazione di attributi nelle dichiarazioni a seconda delle esigenze specifiche di una implementazione del linguaggio C/C++. La definizione di questa macro è la seguente:

```
#define SET_DEFAULT_DECL_ATTRIBUTES(DECL, ATTRIBUTES)
```

dove `DECL` è un nodo dell'albero della rappresentazione interna con la dichiarazione appena parsata, e `ATTRIBUTES` è un nodo con gli attributi della variabile o funzione dichiarata. Si possono usare le informazioni presenti in `DECL` per alterare gli attributi in `ATTRIBUTES` o per altri scopi. L'uso che noi facciamo nel porting di GNU per APEmille è quello di decidere il tipo di allocazione che vogliamo fare (CP, NP o entrambi); allocare la memoria necessaria per la variabile; associare l'indirizzo della memoria appena allocata al simbolo dichiarato.

6.1.2 Convertitore

I simboli presenti nelle istruzioni RTL devono venire convertiti negli indirizzi precedentemente allocati. Questa conversione avviene tramite le macro:

```
#define GO_IF_LEGITIMATE_ADDRESS(MODE, EXP, WIN)
#define LEGITIMIZE_ADDRESS(EXP,OLDEXP,MODE,WIN)
```

Queste due macro fanno parte dell'interfaccia per il porting ed operano come filtri per consentire accessi in memoria con indirizzamenti appropriati per la macchina. La macro `GO_IF_LEGITIMATE_ADDRESS` accetta tre argomenti:

- `MODE` il modo macchina con cui si deve accedere la memoria
- `EXP` l'espressione RTL il cui valore rappresenta l'indirizzo da accedere
- `WIN` una label a cui saltare se l'indirizzo è valido

La macro `LEGITIMIZE_ADDRESS` accetta 4 argomenti:

- `EXP` l'espressione RTL modificata a partire da OLDEXP
- `MODE` il modo macchina con cui si deve accedere la memoria
- `OLDEXP` l'espressione RTL il cui valore rappresenta l'indirizzo da accedere
- `WIN` una label a cui saltare se l'indirizzo è valido

La macro `GO_IF_LEGITIMATE_ADDRESS` accetta solo indirizzi che *non* contengono simboli. Inoltre accetta espressioni RTL che esprimono le possibilità di APEmille di indirizzare la memoria, ovvero:

- `MOP1` costante numerica
- `MOP2` costante numerica + registro
- `MOP3` costante numerica + registro + registro

Le espressioni RTL non accettate da `GO_IF_LEGITIMATE_ADDRESS` vengono passate come `OLDEXP` alla macro `LEGITIMIZE_ADDRESS`. In questa macro viene effettuata la conversione da simbolo a indirizzo, leggendo i dati allocati in precedenza e associati ad ogni simbolo

6.1.3 Gestione degli "scope"

È possibile in un programma C o C++ che ci siano più variabili dichiarate con lo stesso nome in scope diversi: un utilizzo di una variabile che è stata dichiarata in precedenza più volte, viene visto come un accesso alla variabile dichiarata nell'ultimo scope visibile (vedi fig.6.1). Il compilatore GNU compila un programma in modo lineare, ovvero ciò che è scritto prima nel codice viene compilato prima. Questo permette di implementare uno stack (contenitore FILO, "first in last out") associato ad ogni gruppo di simboli con lo stesso nome. In ogni momento, quando il convertitore necessita dell'indirizzo di un simbolo, si accederà l'ultimo simbolo immesso nello stack e si prenderà il suo indirizzo.

Programma	y visibile
`float y;`	y globale
`int g()`	"
`{`	"
`   //…`	"
`   x = y+1;`	y globale
`}`	"
`int f()`	"
`{`	"
`   float y;`	y locale 1
`   //…`	"
`}`	"
`int b()`	"
`{`	y globale
`   y = 2.0;`	"
`   float x,y;`	y locale 2
`   //…`	"
`   while (x>0)`	"
`   {`	"
`      float y;`	y locale 3
`      //…`	"
`   }`	"
`}`	

Figura 6.1: Esempio di *scopes* di varabili con stesso nome

6.2 Indirizzamento della memoria a unità di 32 bit

Il compilatore GNU è stato progettato per macchine a 32 bit che indirizzano unità di memoria di 8 bit o *byte* (vedi [9], Cap. 12). APEmille è una macchina che indirizza parole di memoria a 32bit. L'interfaccia di porting mette a disposizione delle macro che definiscono la grandezza in bit delle unità di indirizzamento di memoria, e il numero di tali unità in una parola macchina (che rappresenta la grandezza in bit dei registri).

```
#define BITS_PER_WORD             32

#define BITS_PER_UNIT             32
```

La macro `BITS_PER_WORD` esprime il numero di bit presenti in una parola macchina, usata per conoscere la grandezza dei registri. La macro `BITS_PER_UNIT` esprime il numero di bit della più piccola unità indirizzabile della memoria.

Un cambiamento si è reso necessario: quello di aggiornare le definizioni dei modi macchina. Infatti le grandezze dei modi macchina sono date rispetto a BITS_PER_UNIT, quindi si è dovuto riscalarle di un fattore 4. Esempio

```
//Prima ....

DEF_MACHMODE (SImode, "SI", MODE_INT, 4, 4, DImode)

//Dopo ...

DEF_MACHMODE (SImode, "SI", MODE_INT, 1, 1, DImode)
```

Ogni macro `DEF_MACHMODE` viene usata per definire un modo macchina; gli argomenti sono:

1. nome del modo usato nella interfaccia del porting.
2. nome del modo usato nelle istruzioni RTL.
3. classe del modo macchina.
4. grandezza in word del modo.
5. grandezza in word di eventuali componenti più piccole.
6. modo macchina della stessa classe immediatamente più grande.

Per completezza esponiamo una lista completa dei modi macchina implementati nel compilatore GNU per APEmille.

```
DEF_MACHMODE (VOIDmode, "VOID", MODE_RANDOM, 0, 0, VOIDmode)
DEF_MACHMODE (SImode, "SI", MODE_INT, 1, 1, DImode)
DEF_MACHMODE (SLImode, "SLI", MODE_LOCAL_INT, 1, 1, VOIDmode)
DEF_MACHMODE (SV2mode, "SV2", MODE_FLOAT, 2, 1, VOIDmode)
DEF_MACHMODE (SFmode, "SF", MODE_FLOAT, 1, 1, DFmode)
DEF_MACHMODE (DFmode, "DF", MODE_FLOAT, 2, 2, XFmode)
DEF_MACHMODE (SCmode, "SC", MODE_COMPLEX_FLOAT, 2, 1, DCmode)
DEF_MACHMODE (BLKmode, "BLK", MODE_RANDOM, 0, 0, VOIDmode)
DEF_MACHMODE (CCmode, "CC", MODE_CC, 1, 1, VOIDmode)
```

Modo	Nome	Size	Tipo associato in C/C++	Tipo APEmille associato	Processore Associato
SI	Single Integer	32	Int, long, puntatori	Interi	T1000
SF	Single Float	32	Float	Singola precisione	J1000
DF	Double Float	64	Double	Doppia precisione	J1000
SC	Single Complex	64	complex	Complessi	J1000
SLI	Single Local Integer	32	Localint	Interi	J1000
SV2	Single Vector 2	64	Vector	Vettori	J1000
VOID	Void	?	Costanti	Costanti	?
CC	Condition Code	?	Condizioni	N/A	N/A

Figura 6.2: Tabella dei Modi macchina

Il modo `VOID` è usato per le costanti; il modo `BLK` è usato per strutture o tipi di dati non associati a nessun modo macchina; il modo `CC` è usato per il *condition code*.

L'associazione canonica dei modi macchina ai tipi del linguaggio e ai tipi nativi supportati da APEmille si può trovare in fig.6.2.

I nuovi modi aggiunti sono `SLI` e `SV2`, rispettivamente per i localint, gli interi locali sui processori di calcolo, e i vector, una coppia di numeri in virgola mobile in singola precisione sempre sui processori di calcolo.

6.3 Accesso della memoria in *burst* per J1000

I processori J1000 di APEmille sono in grado di leggere e scrivere la memoria locale in *burst*, ovvero accedendo a locazioni consecutive ad ogni ciclo di clock.

Questo si riflette nel linguaggio macchina in una sintassi abbastanza semplice associata alle istruzioni `JTOMEM` e `MEMTOJ`, rispettivamente istruzioni di copia dai registri di J1000 alla memoria e viceversa, con il simbolo ":" (due punti) seguito dal numero di locazioni consecutive da leggere o scrivere.

Vediamo degli esempi in SIMD C e in lunguaggio macchina.

```
complex a,b;
struct matrice left, right;   // sizeof(matrice) = 20
a = b;
right = left;

!! immaginiamo a all'indirizzo 0x3020
!!  e b nei registri (60,61)
!! a = b;
JTOMEM1 0x3020 60:2   !! burst di lunghezza 2
!! immaginiamo left all'indirizzo 0x4150 (lungo 20 word)
!!  e right nei registri (100,...,119)
!! right = left;
MEMTOJ  100:20 0x4150
```

Nella spiegazione dell'implementazione del burst distinguiamo due casi che spiegheremo separatamente: burst usati nella copia di dati nativi (`complex`, `vector` e `double`), e burst usati nella copia di strutture.

6.3.1 Burst in copia di dati nativi

Questi burst sono implementati direttamente nel codice generato dai vari pattern associati ai modi macchina DF,SC e SV2. Qui di seguito riportiamo per completezza i due pattern associati al modo DF (modo macchina "Double Float", `double` in SIMD C e C++), essendo i pattern associati agli altri 2 modi sostanzialmente identici.

In ognuno dei burst in oggetto la lunghezza del burst è ovviamente sempre conosciuta ed è sempre 2: questo rende l'implementazione elementare.

```
(define_expand "movdf"
[(set (match_operand:DF 0 "" "")
      (match_operand:DF 1 "" ""))]
""
"{
   extern rtx reduce_symbols_to_numbers(rtx);
   operands[1]=reduce_symbols_to_numbers(operands[1]);
   if(GET_CODE(operands[0])==MEM
       && GET_CODE(operands[1])!=REG)
     operands[1] = force_reg (DFmode, operands[1]);
}")
(define_insn "hard_movdf"
[(set
 (match_operand:DF 0 "general_operand" "=m,=j,=j,=j")
 (match_operand:DF 1 "general_operand" "j,m,j,E"))]
  ""
  "* {
  switch (which_alternative)
  {
  case 0:
```

```
 if (address_is_MOP1(DFmode,XEXP(operands[0],0)))
   return "JTOMEM1 \t^%0 %1 :2";
 if (address_is_MOP3(DFmode,XEXP(operands[0],0)))
   return "JTOMEM3 \t^%0 %1 :2";
 if (address_is_MOP2(DFmode,XEXP(operands[0],0)))
   return "JTOMEM2 \t^%0 %1 :2";
 fatal("DF MOV\n");
case 1:
 if (address_is_MOP1(DFmode,XEXP(operands[1],0)))
   return "MEMTOJ1 \t%0:2 ^%1";
 if (address_is_MOP3(DFmode,XEXP(operands[1],0)))
   return "MEMTOJ3 \t%0:2 ^%1";
 if (address_is_MOP2(DFmode,XEXP(operands[1],0)))
   return "MEMTOJ2 \t%0:2 ^%1";
 fatal("DF MOV\n");
case 2:
 return "JDNORM_PP \t%0 %1 6 0";
case 3:
 return "TTOJ \t\t%h0 %h1.0.0 \t\t"
        "TTOJ \t\t%e0 %e1.0.0 \t\t";}}")
```

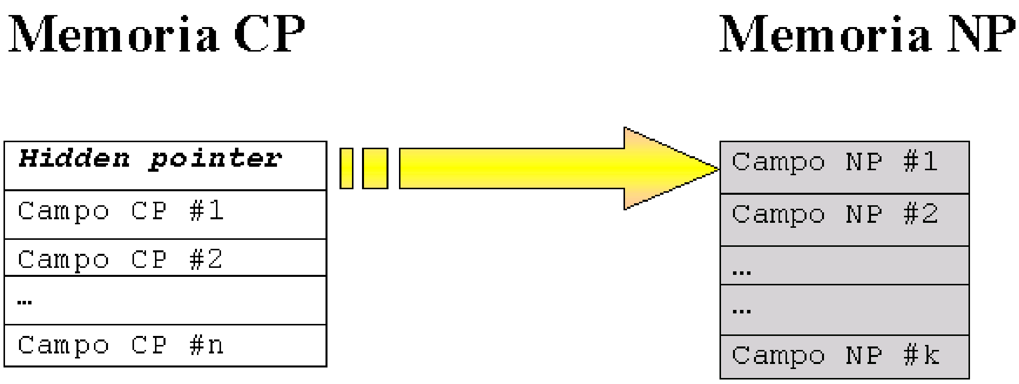

Figura 6.3: Strutture miste: layout su APEmille

6.3.2 Burst in copia di strutture

L'implementazione di questo tipo di burst si è rilevata molto più complessa e lunga della precedente per un semplice fatto: come detto il modello GNU prevede un solo processore in cui viene eseguito il programma e una sola memoria, mentre in APEmille ci sono molteplici nodi di calcolo ognuno con la propria memoria.

Come esposto nel paragrafo 3.1.7, strutture e classi miste vengono allocate in 2 blocchi: uno nella memoria di T1000 e uno in quella degli J1000 (vedi fig.6.3). Per copiare una struttura servono quindi due serie di istruzioni, una per ogni tipo di processore. Questo modello è abbastanza diverso dal modello GNU per cui è stato necessario un lavoro di estensione del linguaggio RTL. L'idea base, molto semplice, la seguente:

Durante l'allocazione (che ricordiamo viene effettuata *durante* la compilazione, vedi paragrafo 6.1) le variabili di tipi strutturati (`struct` e `class`) misti

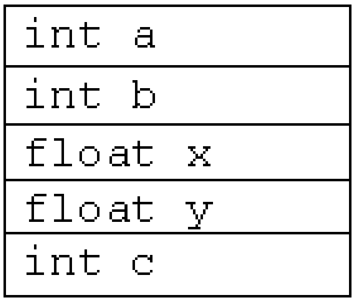

Figura 6.4: Strutture miste: layout su monoprocessore

vengono associate a due blocchi di memoria, uno su T1000 e uno su J1000, ognuno contenente i dati associati al rispettivo processore.

Durante la fase di generazione del codice RTL a partire dalla rappresentazione interna, le assegnazioni tra intere strutture vengono espanse in due copie tra i due blocchi.

Il compilatore GNU non distingue tra i due tipi di memoria, di T1000 e di J1000, quindi abbiamo bisogno di un modo per distinguere i due tipi di trasferimenti di blocchi che GNU associa sempre al modo macchina `BLK`.

È stato scelto di creare una nuova istruzione RTL, `hyper`, che ha un solo operando che contiene un indirizzo di memoria. Questa istruzione serve ad associare la copia della parte J1000 della struttura al pattern appropriato.

Infatti durante la generazione dell'RTL nella istruzione di copia della parte J1000 delle strutture, gli indirizzi sorgente e destinazione vengono sostituiti da espressioni `hyper` i cui operandi sono proprio gli indirizzi di cui sopra (vedi programma seguente).

```
//codice SIMD C/C++
struct mix a,b;  // mix ha size T1000 0x10,J1000 0x20
// a allocata in: T1000 0x8000, J1000 0x10000
// b allocata in: T1000 0x8010, J1000 0x10020
...
a=b
//codice RTL
(movstrsi (mem:BLK (0x8000))
          (mem:BLK (0x10000)) (0x10))
(movstrsi (hyper (mem:BLK (0x8000)))
          (hyper ((mem:BLK (0x10000)) (0x10)))
```

Dato che la scelta dei pattern di istruzioni nella *machine description* (dopo l'utilizzo di quelli con nomi predefiniti) viene effettuata in base alla corrispondenza della espressione RTL associata al pattern, le copie delle parti T1000 delle strutture verranno generate dal pattern `movstrsi`, mentre le copie delle parti J1000 verranno generate dal pattern `hard_movstr___J1000_`. Riportiamo questi due pattern qui di seguito per completezza.

```
(define_expand "movstrsi"
  [(parallel
     [(set (mem:BLK (match_operand:VOID 0 "" ""))
   (mem:BLK (match_operand:VOID 1 "" "")))
     (use (match_operand:SI 2 "nonmemory_operand" ""))
```

```
    (use (match_operand:SI 3 "immediate_operand" ""))
    (clobber (reg:DF 766))(clobber (reg:DF 764))
    (clobber (reg:DF 762))(clobber (reg:DF 760))
    (clobber (reg:DF 758))(clobber (reg:DF 756))
    (clobber (reg:DF 754))(clobber (reg:DF 752))
    (clobber (reg:DF 750))(clobber (reg:DF 748))
    (clobber (reg:DF 746))(clobber (reg:DF 744))
    (clobber (reg:DF 742))(clobber (reg:DF 740))
    (clobber (reg:DF 738))(clobber (reg:DF 736))])]
  ""
 "{
  int size = XINT(operands[2],0);
  int align = XINT(operands[3],0);
  int i;
   if (GET_CODE(operands[0]) == HYPER)
   {
     int size = XINT(operands[2],0);
     int align = XINT(operands[3],0);
     rtx *a0 = malloc(sizeof(rtx*)*size);
     rtx *a1 = malloc(sizeof(rtx*)*size);
     PUT_MODE(operands[0], SImode);
     PUT_MODE(operands[1], SImode);
```

```
      for (i=0; i<size; i++)
      {
        a0[i] = copy_rtx(operands[0]);
        a1[i] = copy_rtx(operands[1]);
        add_mop_to_const (&a0[i],i);
        add_mop_to_const (&a1[i],i);
        emit_insn (gen_movsi(a0[i], a1[i]));
      }
      DONE;
    }
}")
(define_insn "hard_movstr___J1000_"
[(parallel [(set
       (mem:BLK (hyper:BLK (match_operand:VOID 0 "" "")))
       (mem:BLK (hyper:BLK (match_operand:VOID 1 "" ""))))
   (use (match_operand:SI 2 "nonmemory_operand" ""))
   (use (match_operand:SI 3 "immediate_operand" ""))
   (clobber (reg:DF 766))(clobber (reg:DF 764))
   (clobber (reg:DF 762))(clobber (reg:DF 760))
   (clobber (reg:DF 758))(clobber (reg:DF 756))
   (clobber (reg:DF 754))(clobber (reg:DF 752))
   (clobber (reg:DF 750))(clobber (reg:DF 748))
```

```
   (clobber (reg:DF 746))(clobber (reg:DF 744))
   (clobber (reg:DF 742))(clobber (reg:DF 740))
   (clobber (reg:DF 738))(clobber (reg:DF 736))])]
""
"*{ int size, align, i;
   static char buffer[4096], instr[128];
   char number = (address_is_MOP1(SFmode,
       operands[which_alternative]) ? \'1\' :
   address_is_MOP2(SFmode,
           operands[which_alternative])?\'2\':
   address_is_MOP3(SFmode,
           operands[which_alternative])?\'3\':\'x\');
   size = XINT(operands[2],0);
   align = XINT(operands[3],0);
      buffer[0] = 0;
   size += size%2;
   sprintf(buffer,"MEMTOJ%c \t%d:%d\t^%%1\n\t"
           "JTOMEM%c \t^%%0\t%d:%d\t",number,
   510-size,size,number,510-size,size);
   return buffer;
}")
```

Notare che la copia della parte T1000 viene espansa in una serie di semplici “move”, non avendo T1000 l’accesso in burst della memoria. Inoltre nel pattern relativo alla copia della parte J1000 sono necessari una serie di registri temporanei per copiare i dati nel caso di copia da memoria a memoria.

6.4 Tipi nativi “localint” e “vector”

L’aggiuta dei tipi nativi `localint` e `vector` in modo consistente con la definizione di SIMD C e SIMD C++ è stata realizzata estendendo il motore di compilazione GNU in ogni sua fase: dalla rappresentazione interna, al linguaggio RTL, alla *machine description* ed in minima parte alla definizione della grammatica.

Innanzitutto sono stati aggiunti i modi macchina `SLI` e `SV2` rispettivamente associati ai tipi `localint` e `vector`. Per questi modi macchina sono state create due nuove “classi” di modi: `MODE_LOCAL_INT` e `MODE_VECTOR_FLOAT`. Questo perché non è possibile includere questi due nuovi tipi nelle classi preesistenti senza causare ambiguità nelle conversioni e nelle ottimizzazioni: il motore GNU infatti prevede che sia *naturale* eseguire conversioni tra tipi di dati associati a modi macchina della stessa classe, cosa non possibile (in modo *naturale*) tra dati di questi due nuovi tipi e gli altri.

Riportiamo di seguito per comodità le definizioni di questi due modi macchina (già esposti in fig.6.2) e la lista delle classi di modi macchina.

```
DEF_MACHMODE (SLImode, "SLI", MODE_LOCAL_INT, 1, 1, VOIDmode)

DEF_MACHMODE (SV2mode, "SV2", MODE_FLOAT, 2, 1, VOIDmode)
```

```
enum mode_class { MODE_RANDOM, MODE_INT, MODE_FLOAT,
MODE_LOCAL_INT, MODE_PARTIAL_INT, MODE_CC, MODE_VECTOR_FLOAT,
MODE_COMPLEX_INT, MODE_COMPLEX_FLOAT, MAX_MODE_CLASS};
```

Nella gestione rappresentazione interna è stato aggiunto codice che implementa tutte le operazioni aritmetiche fondamentali e le comparazioni (vedi paragrafo 6.5) in modo sostanzialmente identico agli altri tipi di dati.

Una aggiunta semplice come implementazione ma importante dal punto di vista del linguaggio (nonché comoda per il programmatore) è stata quella di estendere la grammatica per poter permettere di accedere le componenti di dati `vector` tramite gli operatori `.lo` e `.hi` (nello stile che rispecchia la tradizione del progetto APE).

Questo è stato realizzato estendendo la rappresentazione interna con due nuovi operatori che accedono la parte "alta" e "bassa" dei dati `vector`, e inserendo nella grammatica speciali azioni semantiche che usano questi operatori nel caso di riferimenti `.lo` o `.hi` a espressioni di tipo `vector`.

```
| primary '.' identifier
{
   if (TIPO_DI_ESPRESSIONE($1)==COMPLEX_TYPE) {
      if (strcmp(IDENTIFIER_POINTER($3),"re")==0)
          $$ = build_unary_op (REALPART_EXPR, $1, 0);
      else if (strcmp(IDENTIFIER_POINTER($3),"im")==0)
         $$ = build_unary_op (IMAGPART_EXPR, $1, 0);
```

```
    } else if (TIPO_DI_ESPRESSIONE($1)==VECTOR_TYPE) {
       else if (strcmp(IDENTIFIER_POINTER($3),"lo")==0)
           $$ = build_unary_op (V2LOPART_EXPR, $1, 0);
       else if (strcmp(IDENTIFIER_POINTER($3),"hi")==0)
           $$ = build_unary_op (V2HIPART_EXPR, $1, 0);
    } else
       $$ = build_component_ref ($1, $3);
}
```

Notiamo che una identica implementazione è stata introdotta per accedere le parti reale e immaginaria dei tipi `complex` i quali, pur essendo già nativamente gestiti dal compilatore GNU, non prevedevano accessi alle componenti in un modo così semplice ed elegante.

Il linguaggio RTL inteso come insieme di istruzioni non è stato esteso in questo particolare caso: infatti le sue istruzioni contengono il modo macchina su cui operare come parte degli operandi. Avendo quindi aggiunto a monte nuovi modi macchina non c'è stato bisogno di estendere il linguaggio RTL.

È stata però modificata l'interfaccia tra il codice e la *machine description* e la gestione delle ottimizzazioni, che non prevedeva affatto nuovi modi macchina.

Sono stati aggiunti tutti i *pattern* della *machine description* relativi ai nuovi tipi `vector` e `localint` (es. `addsli3`, `mulsli3`, ... , `addsv23`, `mulsv23`, etc...): questi *pattern* sono canonicamente associati ai vari `PLUS` e `MULT` etc. della rappresentazione interna con operandi di modo macchina `SV2` o `SLI`.

Esempi dei nuovi pattern (formalmente identici a tutti quelli dei modi macchina rimanenti) sono qui di seguito riportati.

```
(define_expand "addsv23"
[(set (match_operand:SV2 0 "register_operand" "")
(plus:SV2 (match_operand:SV2 1 "register_operand" "")
(match_operand:SV2 2 "register_operand" "")))]
""
"{
  if (GET_CODE(operands[1]) != REG)
      operands[1] = force_reg (SV2mode, operands[1]);
  if (GET_CODE(operands[2]) != REG)
      operands[2] = force_reg (SV2mode, operands[2]);
}")
(define_expand "mulsv23"
[(set (match_operand:SV2 0 "register_operand" "=j")
(mult:SV2 (match_operand:SV2 1 "general_operand" "%j")
(match_operand:SV2 2 "general_operand" "j")))]
""
"{
  if (GET_CODE(operands[1]) != REG)
```

```
      operands[1] = force_reg (SV2mode, operands[1]);
  if (GET_CODE(operands[2]) != REG)
      operands[2] = force_reg (SV2mode, operands[2]);
}")
(define_insn "mulsli3"
[(set (match_operand:SLI 0 "register_operand" "=j")
      (mult:SLI   (match_operand:SLI 1 "general_operand" "%j")
  (match_operand:SLI 2 "general_operand" "j")))]
""
"@
JIMULT \\t\\t%0 %1 %2")
(define_insn "addsli3"
[(set (match_operand:SLI 0 "register_operand" "=j")
      (plus:SLI   (match_operand:SLI 1 "general_operand" "%j")
  (match_operand:SLI 2 "general_operand" "j")))]
""
"@
JIADD \\t\\t%0 %1 %2")
```

Notiamo per completezza che i *pattern* delle istruzioni per il modo macchina `SV2` sono delle `define_expand` in quanto i processori J1000 posseggono *solo* istruzioni *normal* (ovvero, ricordiamo, $a * b + c$) che operano sui dati di tipo `vector`; al contrario, per il tipo `localint`, i processori J1000 posseggono *solo* le tradizionali istruzioni *add*, *mul*, etc...

6.5 Condizioni Locali e Globali

I costrutti dei linguaggi SIMD C e SIMD C++ per il supporto delle condizioni locali e globali sono già state discusse nel paragrafo 3.1.4.

In questo paragrafo illustreremo l'implementazione nel compilatore GNU di questi costrutti, mettendo in risalto le estensioni al linguaggio RTL, alla struttura della rappresentazione interna e al motore del parser.

Il modello GNU segue lo standard del linguaggio C (successivamente adottato anche nel linguaggio C++) per cui in presenza di operazioni logiche su più condizioni, se già dopo la valutazione della prima si conosce il risultato della condizione complessiva, si evita di valutare tutte le altre condizioni.

Ciò accade, per esempio, in un *and* logico tra due condizioni: se la prima è falsa non c'è ragione di valutare la seconda, in quanto l'*and* tra le due sarà comunque falso.

L'*and* tra le due condizioni viene perciò tradotto in :

- valutazione della prima condizione (risultato nel CC).
- salto condizionato (dal CC) a *dopo* il blocco "if".
- valutazione della seconda condizione (risultato nel CC).
- salto condizionato (dal CC) a *dopo* il blocco "if".

Durante l'esecuzione del programma la seconda condizione viene valutata *solo* se la prima risulta vera.

Questo comportamento non è consistente con l'architettura parallela di APEmille, dove i diversi processori numerici J1000 contengono dati differenti, e quindi le condizioni possono essere diverse.

L'approccio quindi, anche utilizzando le estensioni dei linguaggi SIMD C e SIMD C++ (ovvero i costrutti `where-elsewhere`, `all` etc...), è quello di valutare *entrambe* le condizioni come nell'esempio, e *successivamente* operare un *and* logico tra le due comparazioni. Ogni processore J1000 avrà quindi il suo risultato finale della condizione memorizzato in un bit: questo risultato potrà essere poi usato per implementare il `where` (condizionando l'esecuzione delle istruzioni successive in base al valore del suddetto bit) o i vari `all` etc. (mandando i risultati di tutti gli J1000 al controllore T1000 che poi deciderà il da farsi.

Nel caso di condizioni multiple questo meccanismo necessità di uno stack per memorizzare le varie sotto-condizioni: nei processori numerici di APEmille è presente uno stack in grado di contenere 8 risultati di condizioni locali.

Questo modello è completamente diverso[2] dal modello del *Condition Code* del compilatore GNU, e non è ricostruibile a partire da quest'ultimo.

Abbiamo quindi esteso sia la grammatica, la rappresentazione interna e il

[2]Ricordiamo che il modello di gestione delle comparazioni e dei salti condizionati del processore di controllo di APEmille, T1000, è perfettamente implementabile a partire dal modello GNU.

linguaggio RTL per gestire in modo consistente questo modello di condizioni locali per APEmille. Nella rappresentazione interna sono presenti operatori che incapsulano le 6 comparazioni di cui sopra. Tali operatori vengono poi rappresentati nel codice RTL come successione di comparazioni e salti condizionati, come spiegato sopra. Affiancata a questa "catena di compilazione" delle condizioni standard (necessaria per tradurre comparazioni che coinvolgono tipi di dati associati a T1000) abbiamo introdotto una catena con nuovi operatori nella rappresentazione interna e nuove istruzioni RTL per gestire comparazioni tra tipi di dati associati a J1000.

Le istruzioni aggiunte sono esposte (accanto alle rispettive "canoniche") e spiegate nella Tab. 6.1. È stata adottata la convenzione di aggiungere la lettera "J" a tutte le istruzioni RTL e gli operatori della rappresentazione interna per indicare le rispettive istruzioni e operatori per codice di J1000.

Tabella 6.1: Estensioni per la gestione delle condizioni locali

Comparazione	Operatore T1000	Operatore J1000	Istruzione RTL
uguale	EQ	JEQ	jeq
diverso	NE	JNE	jne
minore di	LT	JLT	jlt
maggiore di	GT	JGT	jgt
minore o uguale a	LE	JLE	jle
maggiore o uguale a	GE	JGE	jge

La biforcazione della "catena di compilazione delle condizioni" avviene durante la prima fase della compilazione: il parsing.

Nella definizione della grammatica stessa, infatti, ci sono azioni semantiche differenti a seconda che la comparazione che si sta parsando è tra tipi associati a T1000 o a J1000 (vedi parte della definizione dalla grammatica di seguito). Al momento del parsing sono perfettamente noti i tipi dei diversi operandi.

Nel resto della catena di compilazione le condizioni locali sono gestite in modo molto diverso dalle condizioni "normali". Infatti esse vengono direttamente tradotte nelle corrispettive istruzioni (aggiunte da questo lavoro) RTL, senza salti condizionati intermedi.

```
expr_no_commas:
  cast_expr
  | expr_no_commas '+' expr_no_commas
     { $$ = parser_build_binary_op ($2, $1, $3); }
  | expr_no_commas '-' expr_no_commas
     { $$ = parser_build_binary_op ($2, $1, $3); }
  ...
  | expr_no_commas ARITHCOMPARE expr_no_commas
     {    /* J1000 CC*/
        if (IS_J1000_MODE(TYPE_MODE(TREE_TYPE($1)))
      || IS_J1000_MODE(TYPE_MODE(TREE_TYPE($3)))) {
           switch ($2) {
     case GT_EXPR: $2 = JGT_EXPR; break;
```

```
    case LT_EXPR: $2 = JLT_EXPR; break;
    case LE_EXPR: $2 = JLE_EXPR; break;
    case GE_EXPR: $2 = JGE_EXPR; break;
            case NE_EXPR: $2 = JNE_EXPR; break;
          }
       }
       $$ = parser_build_binary_op ($2, $1, $3);
 if (IS_J1000_MODE(TYPE_MODE(TREE_TYPE($1)))
    || IS_J1000_MODE(TYPE_MODE(TREE_TYPE($3)))) {
        TYPE_MODE(TREE_TYPE($$))=TYPE_MODE(TREE_TYPE($1));
   $$ = build (JSTKSWAP_EXPR, TREE_TYPE($$), $$);
   }
    }
...
```

Anche nella *machine description* si sono ovviamente introdotti i pattern aggiuntivi corrispondenti alle nuove istruzioni RTL e i nuovi operatori della rappresentazione intermedia.. A differenza dei pattern relativi alle condizioni di T1000, quelli relativi a J1000 emettono le valutazione delle comparazioni e poi operazioni logiche sullo stack delle condizioni di J1000.

6.6 Supporto per le funzioni matematiche native di APEmille

In molti processori matematici moderni alcune operazioni tra numeri in virgola mobile non sono direttamente supportate con un'unica istruzione, bensì vengono effettuate a partire da un seme che il processore fornisce.

Le funzioni matematiche per cui i processori J1000 forniscono i semi per il calcolo e i relativi tipi di dati per cui sono disponibili sono in Tabella 6.2. Per trovare l'inverso e l'inverso della radice quadrata si usa l'algoritmo di Newton, mentre per l'esponenziale e il logaritmo vengono usati algoritmi che utilizzano polinomi di Chebytchev.

Tabella 6.2: Funzioni matematiche supportate da APEmille

Funzione	Significato	Tipi di dati per cui è disponibile
inv	Inverso	`float`, `double`, `vector`
invsqrt	Inverso della radice quadrata	`float`, `double`, `vector`
exp	Esponenziale	`float`, `double`, `vector`
log	Logartimo in base e	`float`, `double`, `vector`

6.7 Comunicazione tra i nodi di calcolo

La comunicazione tra i nodi di calcolo di APEmille è gestita in modo molto elegante. Da un punto di vista di linguaggio ad alto livello (SIMD C e SIMD C++), è sufficiente aggiungere delle opportune costanti per accedere elementi di array in nodi di calcolo "vicini" (ovvero adiacenti nella topologia toroidale di APEmille). Diamo un un esempio:

```
#define NODE_UP          0x03000000

complex array[20], c;

// la seguente istruzione accede

// il 12esimo elemento di array nella memoria locale

c = array[12];

// la seguente istruzione accede

// il 12esimo elemento di array nella memoria del processore

//in ``alto'' nella topologia di APEmille

c = array[12 + NODE_UP];
```

Notare che la costante che si aggiunge è un numero "grande": infatti gli 8 bit più significativi di ogni indirizzo stanno ad indicare su quale nodo di calcolo accedere l'indirizzo espresso dai restanti 24 bit meno significativi. Un valore zero per gli 8 bit più significativi indica che si vuole accedere la memoria locale del nodo di calcolo; altri valori corrispondono ai nodi adiacenti nella topologia di APEmille o altri nodi non adiacenti. È possibile infatti programmare alcune tra le 256 costanti disponibili (8 bit corrispondono a 256 possibilità) per corrispondere a determinati nodi di calcolo distanti nella topologia.

Questa caratteristica obbliga a particolare cura dell'aritmetica degli indirizzi.

Infatti la posizione dell'elemento j di un array i cui elementi hanno size s e il cui elemento zero si trova all'indirizzo b si calcola così: $p = b + js$.

Con l'implementazione adottata di esprimere l'indirizzamento remoto come costanti aggiunte negli indici degli array si ha un problema nella moltiplicazione js nel calcolo precedente.

L'indice j infatti ha gli 8 bit più significativi diversi da zero, e moltiplicando j per s quei bit cambiano. È necessario quindi usare una moltiplicazione speciale per accedere gli elementi di array: nella terminologia APE si chiama *MULA* ("Multiplication for Addresses").

La *MULA* è una moltiplicazione di T1000 tra due operandi trascurando gli 8 bit più significativi del primo operando, e successivamente aggiungendo quegli stessi 8 bit

6.8 Gestione di strutture e classi miste

Come esposto in 3.1.7 si rende assolutamente necessario allocare separatamente i diversi tipi di campi di una struttura o una classe nel caso di architetture, come quella di APEmille, che prevedono diversi tipi di processori per gestire diversi tipi di dati. L'implementazione di questa tecnica nel compilatore GNU é stato un lavoro non semplice: il modello GNU, infatti, non prevede due tipi di memorie e considera, naturalmente, una struttura come un blocco di dati contigui. Ciò non è assolutamente vero per strutture miste in programmi scritti per APEmille.

La scelta adottata nell'implementare questo modello di strutture separate con un puntatore nascosto (*hidden*) nella parte allocata in T1000 è stata quella meno distante dalla filosofia del modello GNU e che richiede meno modifiche al compilatore stesso. Nel seguito useremo il termine *strutture* per indicare sia classi che strutture, per brevità.

Dividiamo l'esposizione in tre passi, ognuno associato ad un aspetto di questa tecnica.

1. definizione di strutture miste.

2. accesso a campi di strutture miste.

3. copia di variabili (di strutture) miste.

6.8.1 Definizione di strutture miste

Data la definizione di una struttura, il compilatore GNU si crea una rappresentazione della stessa come una lista di campi ognuno con il suo tipo e l'*offset* all'interno dell'immaggine della struttura durante l'esecuzione del programma. Il compilatore GNU durante il parsing della definizione di variabili associa ad ognuna di loro il codice RTL per accederle.

Vediamo, ad esempio, nel codice C seguente la definizione di una struttura: quello che avviene in GNU (e in ogni altro compilatore per architetture monoprocessore) è l'allocazione contigua e sequenziale dei campi della struttura (vedi fig.6.4).

```
struct mista {
  int a,b;
```

```
  float x,y;
  int c;
};
```

Ciò che invece noi vogliamo è mostrato in fig.6.3.

Per implementare il modello scelto abbiamo bisogno di una tecnologia che distingua tra i diversi tipi di campi e che allochi separatamente (con due espressioni RTL *distinte*) le variabili di tipo misto. Il nome che è stato dato a questa tecnologia è *hyper*, derivante dal fatto che si immaggina di allocare i campi J1000 in qualche parte *al di fuori* della visibilità del compilatore GNU e accedibile tramite un puntatore *hidden*: ciò ricorda uno spazio topologico (quello GNU) immerso in uno spazio più grande (quello GNU+APEmille), un *iper*spazio appunto!

Notiamo che nella corrente implementazione non è possibile, in una chiamata a funzione, passare una variabile di tipo struttura per valore: la si deve passare per riferimento, tramite un puntatore.

Distinzione dei tipi dei campi

Durante il parsing dei campi di una struttura il compilatore aggiorna lo spazio occupato dai campi già parsati e associa ad ogni nuovo campo il suo *offset* all'interno della struttura in base allo spazio già occupato e all'allineamento richiesto. Ciò che è stato introdotto è un altro contatore di spazio allocato per i tipi associati a J1000: ai campi di tipi T1000 verrà assegnato un *offset* in base al contatore di spazio allocato proprio di GNU; ai campi di tipi J1000 verrà assegnato un *offset* in base al contatore speciale aggiuntivo.

Notare che non c'è bisogno di inserire dati per distinguere tra campi T1000 e J1000: il tipo di dato stesso indicherà il processore associato. Durante la compilazione del codice di un programma, eventuali accessi a campi di strutture vengono risolti aggiungendo all'indirizzo della variabile di quel tipo di struttura l'offset del campo in oggetto. È perciò sufficiente allocare due blocchi di memoria nei due processori e lasciare che GNU aggiunga l'offset associato ad ogni campo come richiesto. Per questo aspetto della tecnologia è stato necessario estendere la rappresentazione interna in modo che la descrizione di un tipo tenesse conto separatamente dello spazio totale dei campi di tipo T1000 e di quelli di tipo J1000. L'estensione è mostrata di seguito.

```
struct hyper_extensions
{
  union tree_node* hyper_size;
  unsigned int hyper_align;
};
struct tree_type
{
  char common[sizeof (struct tree_common)];
  union tree_node *values;
  union tree_node *size;
```

```
  union tree_node *size_unit;
  union tree_node *attributes;
  unsigned uid;

  unsigned char precision;
  enum machine_mode mode : 8;
  ...
```

Allocazione duplicata

La duplicazione dell'allocazione di variabili di tipi misti avviene in modo diverso a seconda di variabili locali o globali. Per quelle globali si alloca lo spazio relativo ai campi J1000 in questo processore, e si associa alla rappresentazione della variabile un'espressione RTL per accedere alla suddetta variabile. Questo è stato possibile grazie all'estensione della rappresentazione interna di variabili, come mostrato di seguito.

```
struct hyper_offset
{
  //RTL per i campi J1000
  struct rtx_def *hyper_rtl;
};
struct tree_decl
{
```

```
  struct hyper_offset *hyper; //APEmille extension
  char common[sizeof (struct tree_common)];
  char *filename;
  int linenum;
  unsigned int uid;
  union tree_node *size;
  ....
};
```

Per le variabili locali allocate sullo stack si allocano due blocchi di stack contigui, il primo per i campi T1000 e il secondo per quelli J1000. Alle variabili di tipi misti vengono come al solito associate due espressioni RTL per accedere i due blocchi di dati.

6.8.2 Accesso a campi di strutture miste

Gli accessi a campi di strutture miste richiedono ovviamente di distinguere i due tipi di campi a cui si accede, T1000 o J1000, e usare l'opportuna espressione RTL precedentemente associata (per ogni variabile di tipo misto) ai due blocchi di campi.

Questo è stato ottenuto tramite un'estensione del parser GNU nel codice che gestisce la trasformazione degli accessi a campi di strutture da rappresentazione ad albero a codice RTL. Vengono filtrati gli accessi a campi J1000 modificando l'espressione RTL che GNU usa per accedere il campo. Parte del codice relativo a questa estensione è di seguito presentato.

```
  switch (code)
    {
    case LABEL_DECL:
//...
    case PARM_DECL:
//...
    case VAR_DECL:
//...
//...
    case COMPONENT_REF:
//...
      //hyper: starts
    {
      rtx temp = DECL_RTL(tem);
      if (code==COMPONENT_REF&&IS_J1000_MODE(mode1))
{
  DECL_RTL(tem) = XEXP(DECL_HYPER_RTL(tem),0);
  APE_hyper_flag = 1;
}
      // hyper ends
```

```
      op0 = expand_expr (tem,
   (TREE_CODE (TREE_TYPE (tem)) == UNION_TYPE
    &&(TREE_CODE(TYPE_SIZE(TREE_TYPE (tem)))
     != INTEGER_CST)
             ? target : NULL_RTX),
   VOIDmode,
   modifier == EXPAND_INITIALIZER
   ? modifier : EXPAND_NORMAL);
      //hyper starts
      if (APE_hyper_strange_flag)
{
  DECL_RTL(tem) = temp;
  APE_hyper_flag = 0;
}
      }
      //hyper ends
```

La macro `DECL_HYPER_RTL` accede all'espressione RTL aggiuntiva associata ad ogni variabile di tpo misto per accedere i campi J1000, mentre la macro `DECL_RTL` accede l'espressione RTL usata comunemente da GNU per accedere ogni tipo di campo. Il primo pezzo di estensione salva temporaneamente l'espressione RTL associata ai campi T1000 e vi copia quella dei campi J1000 nei casi opportuni. Quindi la chiamata alla funzione `expand_expr` espande l'accesso al campo richiesto in una espressione RTL corretta, e il codice finale rimette le cose al loro posto ripristinando il valore precedente dell'espressione RTL associata alla variabile in oggetto.

6.8.3 Copia di variabili di tipi misti

La copia di variabili di tipi misti richiede due copie separate, una per ogni blocco. Il compilatore GNU usa per spostare blocchi in memoria una istruzione RTL speciale: `movstrsi` ("MOVe STRing Single Integer"). Non è quindi immediato ricondurla a due istruzioni diverse, con indirizzi diversi e *size* diverse.

Durante l'espansione dell'albero che rappresenta la copia di variabili di strutture miste, viene duplicato in modo opportuno il codice RTL che esegue la copia, facendo in modo che all'interfaccia di porting arrivino due istruzioni diverse per la copia del blocco T1000 e di quello J1000.

Si è scelto di estendere l'RTL con l'operatore unario `HYPER`.

```
DEF_RTL_EXPR(HYPER, "hyper", "e", 'x')
```

Nella copia del blocco J1000 incapsulando i due indirizzi, sorgente e destinazione, in due espressioni `HYPER`. Durante la generazione di codice, nel pattern `movstrsi` di tipo *define_expand* vengono gestite in modo diverso le istruzioni di copia con indirizzi normali (generando il codice per T1000) e quelle con indirizzi incapsulati in `HYPER` (generando codice per J1000).

Il codice della suddetta *define_expand* è di seguito riportato.

```
(define_expand "movstrsi"
  [(parallel [
    (set (mem:BLK (match_operand:VOID 0 "" ""))
 (mem:BLK (match_operand:VOID 1 "" "")))
   (use (match_operand:SI 2 "nonmemory_operand" ""))
   (use (match_operand:SI 3 "immediate_operand" ""))
   (clobber (reg:DF 766))(clobber (reg:DF 764))
   (clobber (reg:DF 762))(clobber (reg:DF 760))
   (clobber (reg:DF 758))(clobber (reg:DF 756))
   (clobber (reg:DF 754))(clobber (reg:DF 752))
   (clobber (reg:DF 750))(clobber (reg:DF 748))
   (clobber (reg:DF 746))(clobber (reg:DF 744))
   (clobber (reg:DF 742))(clobber (reg:DF 740))
   (clobber (reg:DF 738))(clobber (reg:DF 736))])]
  ""
"{
  int size = XINT(operands[2],0);
  int align = XINT(operands[3],0);
  int i;

   if (GET_CODE(operands[0]) != HYPER)
   {
```

```
      int size = XINT(operands[2],0);
      int align = XINT(operands[3],0);
      rtx *a0 = malloc(sizeof(rtx*)*size);
      rtx *a1 = malloc(sizeof(rtx*)*size);
      PUT_MODE(operands[0], SImode);
      PUT_MODE(operands[1], SImode);
      for (i=0; i<size; i++)
      {
        a0[i] = copy_rtx(operands[0]);
        a1[i] = copy_rtx(operands[1]);
        add_mop_to_const (&a0[i],i);
        add_mop_to_const (&a1[i],i);
        emit_insn (gen_movsi(a0[i], a1[i]));
      }
      DONE;
     }
   emit_insn(gen_movstr_J1000(operands));
   DONE;
 }")
```

Capitolo 7

Risultati

Il compilatore C per APEmille ha compilato con successo alcuni programmi di fisica di dimensioni abbastanza grandi; il compilatore C++, invece, ha solo compilato piccoli programmi di prova: questo perché è in uno stato meno stabile del compilatore C.

I programmi compilati, e la cui esecuzione ha avuto pieno successo [1] sono:

- *z2loop*, simulazione di Montecarlo di una teoria di Gauge basata sul gruppo Z_2.
- *ranfloat*, generatore di numeri pseudo-casuali.
- *dirac*, inversione dell'operatore di Dirac.

Riportiamo di seguito un estratto dal programma *ranfloat*: la routine principale che genera i numeri casuali.

[1] Per "pieno successo" intendiamo risultati identici, fino alla settima cifra significativa, ai medesimi programmi scritti in TAO ed eseguiti su APEmille o scritti in C ed eseguiti su personal computer.

```
void ranfloat(int steps) {
  int j,k;
  float delta;
  for (k=0; k < steps; k++ ) {
    j = rf_start-10;
    while (j<0) j += RF_B;
    delta = rf_vector[j]-rf_vector[rf_start]-rf_carry;
    where ( delta >= 0 ) {
      rf_vector[rf_start] = delta;
      rf_carry = 0;
    }
    elsewhere {
      rf_vector[rf_start] = delta + 1.0;
      rf_carry = RF_ONE;
    }
    rf_start = (rf_start+1) % RF_B;
  }
}
```

Questa routine utilizza il costrutto `where`,`elsewhere` proprio del SIMD C. Si noti che i cicli sulle variabili intere `k` e `j` (delle quali esiste un'unica istanza sul processore T1000) sono costrutti standard: `for` e `while`. Di seguito riportiamo la funzione `ranfloat` compilata dal compilatore C per APEmille:

```
!!Function ranfloat
LABEL *0x4010
JLOFRESET
!!ENTER frame size =10, args size=0
TTOMEM2 0x0.SP FP
TAGU2  FP  0x2.SP
TAGU2  SP 0xc.SP
TTOMEM2  %0x0.FP 64
TAGU1  16 0x0
TTOMEM2  %0x4.FP 16
LABEL *0x4011
JLOFRESET
MEMTOT2  16 %0x4.FP
MEMTOT2  17 %0x0.FP
TLT  16 17
JUMPIF          *0x4012.0
JUMP  *0x4013.0
LABEL *0x4012
JLOFRESET
MEMTOT1  16 %0x10000
TAGU2  17 0xfffffff6.16
TTOMEM2  %0x2.FP 17
```

```
LABEL *0x4014
JLOFRESET
MEMTOT2  16 %0x2.FP
TAGU1  17 0x0
TLT  16 17
JUMPIF          *0x4015.0
JUMP  *0x4016.0
LABEL *0x4015
JLOFRESET
MEMTOT2  16 %0x2.FP
TAGU2  17 0x18.16
TTOMEM2  %0x2.FP 17
JUMP  *0x4014.0
LABEL *0x4016
JLOFRESET
MEMTOT2  16 %0x2.FP
TADD   17 16 0
TADD  16 17 16
MEMTOT1  18 %0x10000
TADD   17 18 0
TADD  18 17 18
MEMTOJ2  34 ^0x20000.16
```

```
MEMTOJ2  36 ^0x20000.18

JSLUTMOVE  35 36

JSNORM_PM  34 34 4 35

MEMTOJ1  36 ^0x20030

JSLUTMOVE  35 36

JSNORM_PM  34 34 4 35

JTOMEM2 ^0x6.FP  34

JSTKREPUSH

MEMTOJ2  34 ^0x6.FP

TTOJ  35 0x0.0.0

JSPUSHGE  34 35

JSTKAND

MEMTOT1  17 %0x10000

TADD   16 17 0

TADD  17 16 17

MEMTOJ2  34 ^0x6.FP

JTOMEM2 ^0x20000.17  34

TTOJ  34 0x0.0.0

JTOMEM1 ^0x20030  34

JSTKNOT

JSTKANDBIS

MEMTOT1  16 %0x10000
```

```
TADD   17 16 0
TADD  16 17 16
MEMTOJ2 36 ^0x6.FP
JSTOD  34 36
TTOJ  37 0x3ff00000.0.0
TTOJ  36 0x0.0.0
JDNORM_PP  34 34 6 36
JDTOS  36 34
JTOMEM2 ^0x20000.16  36
TTOJ  34 0x33800000.0.0
JTOMEM1 ^0x20030  34
JPOP
MEMTOT1  16 %0x10000
TAGU2  18 0x1.16
TAGU1  19 0x18
TDIV          16 17 18 19
TTOMEM1  %0x10000 17
LABEL *0x4017
JLOFRESET
MEMTOT2  16 %0x4.FP
TAGU2  17 0x1.16
TTOMEM2  %0x4.FP 17
```

```
JUMP  *0x4011.0

LABEL *0x4013

JLOFRESET

LABEL *0x4018

JLOFRESET

!!LEAVE

TAGU2  SP 0xfffffff4.SP

MEMTOT2 FP 0x0.SP

JUMP  0.RET
```

Per chiarezza diciamo che nella sintassi del linguaggio macchina di APEmille tutto ciò che segue due punti esclamativi ("!") consecutivi, fino alla fine della linea, rappresenta un commento. Ricordiamo che non sono presenti le ottimizzazioni, quindi ci sono trasferimenti di dati superflui, *labels* ridondanti e altre lacune di compilazione che rallentano molto il codice.

Conclusioni

Il lavoro di questa tesi ha dimostrato in modo inequivocabile la possibilità di usare su macchine parallele SIMD i linguaggi di programmazione C e C++ con minime estensioni di sintassi. Pur essendo solo un primo lavoro sperimentale, questa tesi ha dischiuso nuovi orizzonti per un utilizzo delle macchine APE in un ambiente di sviluppo più standard di quello offerto dal linguaggio TAO.

I compilatori C e C++ per APEmille non sono ancora in uno stato stabile e terminale, necessitano quindi di un approfondito test per poter essere usati su programmi di simulazioni fisiche sulla macchina. Non si sono potuti effettuare test di performance per i problemi menzionati nel paragrafo 5.4, ovvero perché i moduli di ottimizzazione già presenti nel compilatore GNU non sono consistenti con le estensioni del linguaggio RTL[2] introdotte in questo lavoro. L'allineamento di questi moduli è un lavoro impegnativo in quanto necessita di comprendere appieno il loro funzionamento per poterli modificare.

Teniamo a sottilineare che, anche se non fa parte di questa tesi, il lavoro qui svolto ha permesso di avere dei compilatori C e C++ in stato avanzato

[2] *Register Transfer Language*, linguaggio usato dal compilatore GNU per descrivere il programma da compilare in una forma intermedia più facilmente manipolabile e ottimizzabile del linguaggio ad alto livello o di quello della macchina

di completamento per il supercomputer di prossima generazione del progetto APE: apeNEXT. La conoscenza della tecnologia GNU e l'esperienza acquisita nel campo dei compilatori C permetterà al gruppo APE di poter avere compilatori prototipi durante la progettazione e la simulazione di apeNEXT, prima cioé di avere fisicamente la macchina pronta. Questo permetterà di verificare l'architettura dei microprocessori con programmi molto più grandi di quelli che si possono scrivere direttamente in linguaggio macchina, evitando possibili errori ed aumentando l'efficienza del progetto.

GNU è un progetto maturo e stabile, è quindi naturale chiedersi se ci si possa appoggiare ad esso anche per altri compilatori del progetto APE, ovvero TAO e *Zz*. Sarebbe interessante, oltre che un progetto fortemente innovativo, poter implementare *Zz* sul motore di compilazione GNU: si avrebbe una grammatica evolvente compilabile su una qualsiasi macchina esistente!

Infine, una delle idee più belle ed anche più impegnative, è quella di estendere il modello di astrazione di GNU per poter supportare anche macchine parallele e supercomputer. Il lavoro di questa tesi ha dimostrato che ciò è realizzabile in tempi ragionevoli. Le estensioni più interessanti dal nostro punto di vista sono:

1. distribuire tipi di dati tra processori diversi.

2. possibilità di estendere dinamicamente il linguaggio e la rappresentazione interna per sfruttare particolari istruzioni macchina.

3. comparazioni gestite anche come stack o "predication"[3] (come in APE).

4. allocazione statica delle variabili.

5. possibilità di avere diversi livelli di memoria indipendenti (es. T1000 e J1000) con delle regole per gli accessi e i trasferimenti.

In conclusione, questo progetto ha portato novità nel gruppo APE, insieme a nuove sfide: faremo del nostro meglio per esserne all'altezza.

[3]Tecnica usata per condizionare l'esecuzione di singole istruzioni, gruppi di esse, con condizioni note durante l'esecuzione del programma: eventuali condizioni "false" semplicemente disattivano la scrittura dei risultati dell'istruzione condizionata, evitando le lente istruzioni di salto.

Bibliografia

[1] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers Priciples, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[2] A.V. Aho, J.D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Vol. I and II, Prentice Hall, Englewood Cliffs, CA, 1973.

[3] B.W. Kernighan, D.M. Ritchie, *Linguaggio C.* Gruppo Editoriale Jackson, 1989.

[4] B. Stroustrup, *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[5] J.P Tremblay, P.G. Sorenson, *Compiler Writing*, McGraw-Hill 1985

[6] J. Levine, T. Mason, D. Brown, *Lex and Yacc*, O'Reilly and associated, 1992.

[7] The Institute of Electrical and Electronics Engineers, Standard 754, 1985.

[8] Koenig, A.E. 1996. Working paper for Draft Proposed International Standard for Information Systems - Programming Langua-

ge C++. Document Number: X3J16/96-0225 WG21/N1043. URL: `http://www.maths.warwick.ac.uk/c++/pub/wp/`

[9] Stallman, R.: *Using and Porting GNU CC, for Version 2.95* Free Software Foundation ISBN: 1-882114-38-8

[10] Aglietti, F. et al.: *The teraflop supercomputer APEmille: architecture, software and project status report.* Computer Physics Communications 110 (1998) 216–219

[11] Tripiccione, R.: *APEmille.* Parallel Computing **25** (1999) 1297-1309

[12] Rapuano, F.: *Quenched physics on APE computers.* Parallel Computing **25** (1999) 1217-1226

[13] S. Güsken, Th. Lippert, K, Schilling, *Lattice QCD with two dynamical Wilson fermions on APE100 parallel systems.* Parallel Computing **25** (1999) 1227-1242

[14] The HPC++ working group. HPC++ white papers. Technical Report 95633, Center for Research on Parallel Computation, Dec. 1995

[15] `http://www.extreme.indiana.edu/hpc++`

[16] `http://www.extreme.indiana.edu/sage`