POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione
Corso di laurea in Ingegneria Informatica

Una Metodologia per la Stima di Prestazioni di Sistemi Embedded basata su Distribuzioni di Probabilità

Relatore: Prof. Fabrizio FERRANDI
Correlatore: Ing. Marco LATTUADA

Tesi di laurea di
Luca DE MARCO
Matr. 724969

Anno Accademico 2009/2010

Allo zio Carlo

Sommario

I sistemi multiprocessori integrati sono al giorno d'oggi lo standard nel campo della progettazione di sistemi integrati. La fase di analisi delle prestazioni dell'applicazione ricopre un ruolo di importanza critica durante la progettazione di tali sistemi, poiché soddisfare i vincoli di prestazioni è uno degli obiettivi fondamentali del flusso di progettazione. Verificare tali vincoli e correggere il sistema solo nelle fasi finali della progettazione può richiedere cambiamenti significativi del sistema stesso ritardando la produzione di tempi non accettabili. Le prestazioni del sistema devono quindi essere accuratamente stimate sin dalle prime fasi della progettazione. Questo lavoro di tesi sviluppa una nuova metodologia di stima delle prestazioni basata su distribuzioni di probabilità che sfrutta il metodo di regressione lineare e la statistica per fornire un intervallo di confidenza del tempo di esecuzione di un dato codice sorgente. In particolare utilizza una tecnica di regressione lineare in grado di effettuare una stima simultanea del tempo medio di esecuzione e della relativa varianza, applicandola a un insieme numeroso ed eterogeneo di benchmark. Tramite il calcolo della varianza del tempo di esecuzione vengono quindi generati gli intervalli di confidenza per il tempo medio di esecuzione tramite i quali si può valutare la confidenza nella stima. Risultati sperimentali mostrano la bontà della metodologia nel fornire una stima del tempo di esecuzione e la relativa confidenza.

Indice

1 Introduzione **1**

2 Definizioni Preliminari di Geometria e Statistica **5**

2.1 Definizioni Preliminari di Statistica 5

2.1.1 Intervalli di Confidenza 8

2.1.2 Teoremi Centrali del Limite 10

2.2 Definizioni Preliminari di Geometria 11

2.3 La Regressione Lineare Multivariata 11

2.3.1 Il modello generico di regressione 12

2.3.2 Ipotesi alla base del Modello e il Metodo dei Minimi Quadrati 13

2.3.3 Stima della Varianza dei coefficenti di Regressione 15

2.3.4 Stima delle varianze delle osservazioni **Y** e intervalli di confidenza 17

2.3.5 Test sul Modello Utilizzato 18

2.3.6 Verifica dell'Assenza di Correlazione fra i Residui 20

2.3.7 Verifica della Significatività dei Regressori 21

2.3.8 Calcolo del Coefficiente di Determinazione R^2 21

3 Definizioni Preliminari **23**

3.1 La Struttura del Compilatore GNU GCC 23

3.2 Le Rappresentazioni Intermedie del Compilatore GCC 25

3.2.1 GENERIC 26

3.2.2 GIMPLE 26

3.2.3 RTL 28

3.3 Rappresentazioni Intermedie basate su Grafi 32
3.3.1 I Blocchi Basici 32
3.3.2 Control Flow Graph 32
3.4 Il Software Profiling 34

4 Analisi dello Stato dell'Arte nella Stima di Prestazioni 39
4.1 Tecniche di Stima Basate su Simulatore 39
4.2 Tecniche di Stima Basate su Modelli Matematici 40
4.2.1 La stima di prestazioni con intervalli di confidenza secondo Jantsch e Bjuréus 41
4.2.2 Considerazioni sulla metodologia 43
4.2.3 Le basi della presente tesi 44
4.2.4 Altre tecniche di stima basate su modelli matematici . . . 45

5 Approccio Metodologico 49
5.1 La stima di prestazioni con intervalli di confidenza proposta . . . 49
5.1.1 Estensione del metodo di stima e della tecnica di regressione 50

6 L'Implementazione all'interno del Progetto PandA 61
6.1 Zebu 62
6.1.1 Il Front-End di Zebu 62
6.1.2 Il Middle-End di Zebu 63
6.1.3 Modifiche effettuate a Zebu 63
6.2 Spider 66

7 Risultati Sperimentali 73
7.1 Il processore Leon e tsim 73
7.2 Caratteristiche dei benchmark 74
7.3 Risultati 75
7.4 Risultati con operazioni GIMPLE 77
7.5 Risultati con operazioni RTL 78
7.5.1 Risultati con benchmark interi 79
7.5.2 Commenti sul modello 82
7.5.3 Risultati con benchmark float 83

7.5.4 Commenti sul modello 83
7.5.5 Risultati con tutti i benchmark 84
7.5.6 Commenti sul modello 85

8 Conclusioni e possibili sviluppi futuri 87

Riferimenti bibliografici 92

Elenco delle tabelle

4.1 Modello di VIS 41

7.1 Caratteristiche dei modelli GIMPLE ottenuti con algoritmo di rimozione benchmark mal normalizzati 77
7.2 Caratteristiche dei modelli GIMPLE ottenuti con algoritmo MinMax 78
7.3 Risultati algoritmi rimozione benchmark 79
7.4 Medie, varianze e significatività dei regressori con modello intero normalizzato 79
7.5 Caratteristiche del modello intero normalizzato 80
7.6 Risultati test su modello intero normalizzato 80
7.7 Medie, varianze e significatività dei regressori con modello intero e applicazione algoritmo MinMax 81
7.8 Caratteristiche del modello intero risultato dell'applicazione dell'algoritmo MinMax 82
7.9 Risultati test su modello intero e applicazione algoritmo MinMax 82
7.10 Risultati algoritmi rimozione benchmark 83
7.11 Caratteristiche dei modelli RTL float 84
7.12 Risultati algoritmi rimozione benchmark 84
7.13 Caratteristiche dei modelli RTL float 85

Elenco delle figure

3.1 Struttura del compilatore GCC 24
3.2 Esempio di codice sorgente 28
3.3 Rappresentazione GIMPLE del codice sorgente in Figura 3.2 . . . 29
3.4 Esempio di codice RTL 30
3.5 Codice Sorgente per la traduzione nella rappresentazione RTL . . 31
3.6 Esempio di Rappresentazione RTL per processori ARM 31
3.7 Esempio di Rappresentazione RTL per processori SPARC 33
3.8 Esempio di codice sorgente 34
3.9 CFG del blocchi basici del codice in Figura 3.8 35
3.10 Esempio di codice sorgente 35
3.11 FCFG del codice in Figura 3.10 36

6.1 Distribuzione di probabilità Costante 65
6.2 Esempio di file XML prodotto da Zebu 65
6.3 Listato dei comandi eseguiti da R per effettuare regressione 68
6.4 Coefficenti risultanti dai comandi di Figura 6.3 69
6.5 Esempio file XML risultato generato da Spider 71

Capitolo 1

Introduzione

Il continuo aumento della potenza di calcolo richiesta, combinato con la necessità di ridurre il consumo di potenza dei *sistemi integrati*, ha causato la diffusione di *sistemi multiprocessori integrati* (*Multiprocessor Systems-on-Chip* o MPSoCs), architetture a singolo chip composte da elementi di calcolo, omogenei o eterogenei tra loro, una gerarchia di memoria e componenti di Input/Output connessi attraverso un infrastruttura di rete.

Questo tipo di sistemi è in grado di fornire una potenza di calcolo significativa, ma il loro uso può introdurre nuovi problemi nella progettazione di sistemi integrati, come ad esempio la scelta dei componenti, la divisione dell'applicazione in processi (o *partizionamento*), la descrizione del parallelismo fra processi, l'associazione dei processi ai relativi elementi di calcolo (o *assegnamento*) e l'analisi delle prestazioni dell'applicazione parallela risultante per verificare se i requisiti di progettazione sono soddisfatti.

Nonostante l'aumento della complessità della fase di progettazione, il *time-to-market* dei sistemi integrati non è stato aumentato. Al contrario, un flusso di design più veloce è richiesto dai produttori al fine di rendere i prodotti disponibili sul mercato il prima possibile. Questa accelerazione nel processo di design impone al progettista di utilizzare euristiche molto veloci per ridurre i tempi di esplorazione dello spazio di progettazione, impoverendo la qualità dei risultati di progettazione.

Ci sono però aspetti della fase di progettazione in cui l'analisi non può essere

troppo approssimativa. Uno di questi è l'analisi delle prestazioni dell'applicazione dato che soddisfare i vincoli di prestazioni è l'obiettivo fondamentale del flusso di progettazione e i vincoli temporali sono tra gli aspetti più critici nella progettazione di sistemi integrati. Verificare tali vincoli solo nelle fasi finali della progettazione non è possibile: correggere il sistema nelle fasi finali del processo di design può richiedere cambiamenti significativi del sistema stesso ritardando la produzione di tempi spesso non accettabili. Per questa ragione le prestazioni del sistema devono essere stimate sin dalle prime fasi della progettazione. Tale stima deve essere veloce, per non allungare ulteriormente i tempi di progettazione, ma allo stesso tempo affidabile, al fine di fornire un analisi veritiera della prestazioni dell'applicazione. L'affidabilità diventa quindi un parametro importante in quanto fornisce al progettista un indice di quanto attendibile e precisa sia la stima delle prestazioni del sistema ossia fornisce un indice di *confidenza*, utile per la verifica del soddisfacimento dei vincoli temporali.

Essendo ignoto il valore esatto delle prestazioni del sistema poiché si è scelto di non effettuare alcuna simulazione per non allungare i tempi di progettazione, risulta impossibile calcolare esattamente l'affidabilità della stima. Si può ricorrere alla statistica e, in particolare, agli *intervalli di confidenza* per definire degli intervalli numerici che contengano il valore vero incognito delle prestazioni del sistema con alta probabilità, ossia con alta confidenza.

Questo lavoro di tesi si inserisce proprio all'interno di uno strumento di progettazione automatico ed in particolare nella sua parte dedicata alla stima di prestazioni e al calcolo della relativa confidenza.

Il presente lavoro di tesi si basa su due lavori precedenti. Il primo, di *Ferrandi e Lattuada* [21], propone una metodologia di stima di prestazioni che sfrutta il metodo di regressione lineare applicato a un insieme di benchmark per costruire un modello matematico di stima delle prestazioni di un'applicazione. Il secondo, di *Jantsch e Bjuréus* [9], propone una metodologia di stima di prestazioni basata su un modello matematico di un architettura obiettivo in grado di fornire anche un indice di affidabilità della stima tramite l'utilizzo degli *intervalli di confidenza*.

Il contributo portato da questo lavoro di tesi consiste quindi in una nuova metodologia di stima di prestazioni basata su modelli matematici che, sfrut-

tando una estensione della tecnica di regressione lineare, permette di effettuare una stima simultanea del tempo medio di esecuzione di un applicazione e della relativa varianza tramite una stima del tempo di esecuzione e della varianza delle singole operazioni che compongono l'applicazione. Utilizzando la varianza della stima, fornisce un indice dell'affidabilità della stessa tramite l'uso della statistica e il calcolo di un intervallo di confidenza associato alla stima. Sfruttando la regressione in media e varianza, la metodologia proposta consente una modellizzazione flessibile e a carattere generale delle operazioni, permettendo di associarvi diverse distribuzioni di probabilità.

Il presente lavoro di tesi è suddiviso in otto capitoli; il Capitolo 2 contiene definizioni, teoremi e procedimenti statistici necessari per la comprensione dell'intero lavoro.

Il Capitolo 3 contiene le definizioni preliminari relative alla struttura del compilatore GNU GCC e delle sue rappresentazioni intermedie, ai linguaggi GIMPLE ed RTL (*Register Transfer Language*), i concetti di ciclo (*Loop*), di *Blocco Basico* e di *Control Flow Graph*, e le tecniche di *software profiling*.

Il Capitolo 4 propone l'analisi dello stato dell'arte relativo alla Stima di Prestazioni per processori Embedded, con particolare attenzione alla definizione di intervalli di confidenza per il tempo di esecuzione.

Il Capitolo 5 illustra la metodologia sviluppata durante l'intero lavoro di tesi.

Nel Capitolo 6 viene fornita una breve descrizione di PandA, il framework all'interno del quale è stata implementata la metodologia proposta.

Il Capitolo 7 riporta i risultati sperimentali relativi all'applicazione della metodologia.

Il Capitolo 8 infine riporta l'analisi conclusiva del lavoro svolto evidenziando i possibili sviluppi futuri.

Capitolo 2

Definizioni Preliminari di Geometria e Statistica

In questo Capitolo vengono presentati definizioni, teoremi e procedimenti statistici utilizzati nel corso del presente lavoro di tesi. Nella prima Sezione verranno forniti le definizioni e i teoremi di base necessari per la comprensione del modello e dei procedimenti di stima utilizzati a partire da esso. Nella seconda Sezione verranno poi presentati e discussi tali procedimenti.

2.1 Definizioni Preliminari di Statistica

Vengono ora presentati le definizioni e i teoremi sui quali si basa il presente lavoro di tesi.

Definizione 1 (Spazio di Probabilità)**.** *Si definisce spazio di probabilità la terna* (Ω, $\mathcal{F}$, $\mathcal{P}$)*, indicando con* Ω *lo spazio campionario, ossia un insieme di possibili valori di* $\mathbb{R}$ *e con* $\mathcal{P}$ *la funzione di misura di probabilità definita sull'algebra* $\mathcal{F}$*.*

Definizione 2 (Variabile stocastica o variabile aleatoria o variabile casuale o v.a.)**.** *Sia* $(\Omega, \mathcal{F}, \mathcal{P})$ *uno spazio di probabilità . Una variabile aleatoria* X *è una funzione da* Ω *in* $\mathbb{R}$ *tale che per ogni* $x \in \mathbb{R}$*, l'insieme*$\{X \leq x\} := \{\omega \in \Omega : X(\omega) \leq x\} \in \mathcal{F}$*.*

Definizione 3 (Variabile aleatoria continua o v.a. continua). *Una variabile aleatoria X si dice continua se l'insieme dei possibili valori che X può assumere ha la potenza del continuo ossia ha la cardinalità di* $\mathbb{R}$.

Definizione 4 (Densità di Probabilità Continua). *Data una v.a. X si definisce Densità di Probabilità Continua l'applicazione* $f_X(x)$ *reale di variabile reale, non negativa e integrabile tale che la probabilità che X appartenga a un dato insieme A è data da:*

$$P(X \in A) = F(x) = \int_{x \in A} f_X(x)dx \tag{2.1}$$

$\forall$ *sottoinsieme A dello spazio campionario* Ω *di X.*

Definizione 5 (Media v.a. Continua). *Sia X una variabile aleatoria continua e sia* f_X *la sua densità. Se*

$$\int_{\mathbb{R}} |x| f_X dx < +\infty \tag{2.2}$$

si definisce media di X o valore atteso di X il numero

$$E[X] := \int_{\mathbb{R}} x f_X dx \tag{2.3}$$

altrimenti si dice che X non ammette valore atteso. *Ross[31]*

Definizione 6 (Varianza). *Sia X una variabile aleatoria discreta o continua, tale che esista E(X). Se inoltre esiste*

$$E[(X - E(X)^2] \tag{2.4}$$

allora si pone $Var(X) := E[(X - E(X)^2]$ *e si chiama Varianza di X. La radice quadrata della varianza*

$$\sqrt{Var(X)}$$

prende il nome di deviazione standard di X. *Ross[31]*

Definizione 7 (Variabile Casuale Normale). *La variabile casuale Normale (detta anche variabile casuale Gaussiana) è una variabile casuale continua avente distribuzione di probabilità a due parametri, indicata tradizionalmente con:*

$$N(\mu, \sigma^2) \tag{2.5}$$

caratterizzata dalla seguente funzione di densità di probabilità

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x-\mu^2}{2\sigma^2}} \tag{2.6}$$

Ross[31]

Definizione 8 (Densità Congiunta Continua o di Variabili Aleatorie Continue)**.** *Date due v.a. X e Y si definisce Densità di Probabilità Congiunta Continua l'applicazione $f_{X,Y}(x,y)$ reale, non negativa e integrabile rispetto a x e rispetto a y, tale che la probabilità che X appartenga a un dato insieme A e Y appartenga a un dato insieme B è data da:*

$$P(X \in A, Y \in B) = F_{X,Y}(x,y) = \int_{x \in A} \int_{y \in B} f_{X,Y}(x,y) dy dx \tag{2.7}$$

per ogni sottoinsieme A dello spazio campionario Ω_X di X e per ogni sottoinsieme B dello spazio campionario Ω_Y di Y. La densità di probabilità associata alla singola variabile aleatoria è detta Densità Marginale ed calcolabile tramite la formula:

$$f_X(x) = \int_{y \in B} f_{X,Y}(x,y) dy \tag{2.8}$$

Definizione 9 (Indipendenza tra Variabili Aleatorie)**.** *Due variabili aleatorie X, Y si dicono indipendenti se la densità di probabilità congiunta associata a X e Y, $f_{X,Y}(x,y)$ è data dal prodotto delle due densità marginali, ossia:*

$$f_{X,Y}(x,y) = f_X(x) \cdot f_Y(y) \tag{2.9}$$

Definizione 10 (Funzione Generatrice dei Momenti per v.a. Continua)**.** *Si definisce Funzione Generatrice dei Momenti di una v.a. continua X o f.g.m. la funzione:*

$$M_X(t) = E[e^{tX}] = \int_{-\infty}^{+\infty} e^{tx} f(x) dx \tag{2.10}$$

Definizione 11 (Momento di ordine n-esimo)**.** *Si definisce Momento di ordine n-esimo o semplicemente momento n-esimo di una v.a. continua X la derivata di ordine*

n-esimo della f.g.m. associata alla v.a. X:

$$\mu_n = \frac{d^n M_X}{dt^n}|_{t=0} \tag{2.11}$$

Teorema 1 (Teorema di Additività). *Siano ξ e η due variabili stocastiche indipendenti. Si ha che $E[\xi + \eta] = E[\xi] + E[\eta]$ e $Var[\xi + \eta] = Var[\xi] + Var[\eta]$*

Teorema 2 (Teorema di Addizione per variabili Normali). *Se ξ_1 è una v.a. $N(\mu_1, \sigma_1^2)$ e ξ_2 è una v.a. $N(\mu_2, \sigma_2^2)$, allora la variabile aleatoria ξ data dalla somma di ξ_1 e ξ_2 si distribuisce anch'essa come normale:*

$$\xi = \xi_1 + \xi_2 \sim N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2) \tag{2.12}$$

Definizione 12 (Campione Casuale). *Siano X_1,...,X_n n variabili aleatorie indipendenti e identicamente distribuite (aventi stessa funzione di densità di probabilità) con comune funzione di densità $f(x, \theta)$, $\theta \in \Theta$. Diremo che X_1,...,X_n è un campione casuale di dimensione n estratto dalla popolazione di densità $f(x, \theta)$.*

Definizione 13 (Statistica). *Una Statistica è una variabile aleatoria T funzione del campione casuale $X_1, ..., X_n$ ossia $T = g(X_1, ..., X_n)$. La distribuzione (o legge) di una statistica T è detta Distribuzione o Legge Campionaria.*

Definizione 14 (Media Campionaria). *Sia $X_1, ..., X_n$ un campione casuale estratto da una popolazione di densità $f(x, \theta)$ avente media μ. Si definisce Media Campionaria la statistica:*

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{2.13}$$

Le media campionaria è uno stimatore puntuale non distorto della media μ.

2.1.1 Intervalli di Confidenza

Verranno ora fornite le formulazioni degli intervalli di confidenza per popolazioni gaussiane come descritto in [31].

Definizione 15 (Intervallo di Confidenza per la media di popolazione gaussiana a varianza nota). *Sia $x_1, ..., x_n$ una realizzazione del campione casuale $X_1, ..., X_n$*

estratto della popolazione $N(\mu, \sigma^2)$. Fissato $\gamma \in (0,1)$ definito come livello di confidenza dell'intervallo, se la varianza σ^2 è nota, si definisce intervallo di confidenza per la media μ quell'insieme di valori tale per cui la probabilità che il valore vero incognito μ, di cui $\overline{X}$ è stimatore, sia contenuto nell'intervallo è esattamente pari a γ. Esprimendo quanto detto in formule si ha che:

$$P_{\mu,\sigma^2}\left(-\epsilon < \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} < \epsilon\right) = \gamma \tag{2.14}$$

Un intervallo di confidenza per la media μ di livello $\gamma 100\%$ è dato quindi da:

$$\left(\overline{x} - z_{\frac{1+\gamma}{2}}\frac{\sigma}{\sqrt{n}}, \overline{x} + z_{\frac{1+\gamma}{2}}\frac{\sigma}{\sqrt{n}}\right) \tag{2.15}$$

dove con $z_{\frac{1+\gamma}{2}}$ si identifica il quantile di ordine $\frac{1+\gamma}{2}$ della normale $N(0,1)$ definito come:

$$P(X \leq k) = \gamma \Leftrightarrow k = z_\gamma \tag{2.16}$$

Intevalli di Confidenza Notevoli

nel caso in cui è nota una sola osservazione x del campione casuale composto dalla singola v.a. X proveniente da popolazione gaussiana ossia $X \sim N(\mu, \sigma^2)$ si può semplificare l'Equazione 2.15 ottenendo l'Equazione 2.17:

$$\left(x - z_{\frac{1+\gamma}{2}}\sigma, x + z_{\frac{1+\gamma}{2}}\sigma\right) \tag{2.17}$$

Inoltre, per particolari valori di γ i quantili della variabile aleatoria $N(0,1)$ assumono valori interi. Per semplicità di calcolo si è quindi scelto di utilizzare i seguenti livelli di confidenza γ:

$$\text{Se } \frac{1+\gamma}{2} = 99.7\% \text{ si ha che } z_{\frac{1+\gamma}{2}} = 3 \text{ e quindi l'IC diventa: } (x - 3\sigma, x + 3\sigma) \tag{2.18}$$

$$\text{Se } \frac{1+\gamma}{2} = 95.5\% \text{ si ha che } z_{\frac{1+\gamma}{2}} = 2 \text{ e quindi l'IC diventa: } (x - 2\sigma, x + 2\sigma) \tag{2.19}$$

Se $\frac{1+\gamma}{2} = 68.3\%$ si ha che $z_{\frac{1+\gamma}{2}} = 1$ e quindi l'IC diventa: $(x - \sigma, x + \sigma)$ (2.20)

2.1.2 Teoremi Centrali del Limite

I *Teoremi Centrali del Limite* sono una famiglia di teoremi di convergenza debole nell'ambito della teoria della probabilità . A tutti i teoremi è comune l'affermazione che la somma(normalizzata) di un grande numero di variabili casuali è distribuita approssimativamente come una variabile casuale normale standard $N(0, 1)$. Ciò spiega l'importanza che quest'ultima variabile casuale assume nell'ambito della statistica e della teoria della probabilità in particolare. Questi teoremi affermano quindi che se si ha una somma di variabili aleatorie X_i *indipendenti e identicamente distribuite* (con densità uguali) con media μ e varianza σ^2, allora indipendentemente dalla distribuzione di partenza, al tendere della dimensione campionaria a infinito la somma tende a distribuirsi come una variabile casuale normale. Quanto appena detto viene espresso in formule come:

$$\sum_{i=1}^{n} X_i \sim N(n\mu, n\sigma^2) \tag{2.21}$$

Standardizzando l'Equazione 2.21 si ottiene:

$$\lim_{n\to\infty} \frac{\overline{X}_n - \mu}{\sigma} \sqrt{n} = N(0, 1) \tag{2.22}$$

dove

$$\overline{X}_n = \frac{\sum_{i=1}^{n} X_i}{n} \tag{2.23}$$

è la variabile aleatoria *Media Campionaria*.

La più nota formulazione di un teorema del limite centrale è quella dovuta a Lindeberg e Levy, che dimostrarono tale teorema nel 1922 [31].

Definizione 16 (Teorema Centrale del Limite di Lindeberg-Levy)**.** *Si consideri una successione di variabili casuali* $\{x_i\}_{i=1}^{n}$ *indipendenti e identicamente distribuite, tali che esistano, finiti, i loro momenti di ordine primo e secondo, e sia in particolare*

$E[x_i] = \mu < \infty$ e $var(x_i) = \sigma^2 < \infty$ $\forall i$. Definita allora la nuova variabile casuale:

$$S_n = \frac{\overline{X} - \mu}{\sigma / \sqrt{n}} \tag{2.24}$$

dove

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2.25}$$

è la media aritmetica degli x_i, si ha che S_n converge in distribuzione a una variabile casuale normale avente valore atteso 0 e varianza 1, ossia la distribuzione di S_n, al limite per $n \to \infty$, coincide con quella di una variabile casuale normale.

Teorema 3 (Teorema Centrale del Limite Applicato)**.** *Sia ξ_k^{op} una variabile stocastica di tipo k-esimo. Sia $\mu_{op} = E[\xi_k^{op}]$ e $\sigma_{op}^2 = Var[\xi_k^{op}]$. Se n_{op} è elevato e $\xi_1^{op}, \xi_2^{op}, ..., \xi_{n_{op}}^{op}$ sono indipendenti, allora $t_{op} = \sum^n \xi_k^o p \sim N(n\mu_{op}, n\sigma_{op}^2)$.*

2.2 Definizioni Preliminari di Geometria

In questa Sezione vengono presentate le definizioni di geometria necessarie alla comprensione del presente lavoro di tesi.

Definizione 17 (Prodotto Scalare di Vettori)**.** *Il* Prodotto Scalare *di due vettori* **a** *= [$a_1, a_2, ..., a_n$] e* **b** *= [$b_1, b_2, ..., b_n$] è definito come:*

$$\mathbf{a}^{\mathbf{T}} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n \tag{2.26}$$

Definizione 18 (Norma 2 di Vettore)**.** *Dato un vettore* **x** *si definisce norma 2 del vettore $||\mathbf{x}||_2$ la funzione:*

$$||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}. \tag{2.27}$$

2.3 La Regressione Lineare Multivariata

In statistica, la *Regressione Lineare Multivariata* rappresenta un metodo di stima del valore atteso di una variabile Y, detta variabile endogena, dati i valori

di altre N variabili indipendenti, dette anche esogene o variabili esplicative, $X_1, ..., X_N$, legati alla variabile endogena dalla relazione funzionale lineare:

$$Y = \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_N X_N + \varepsilon \tag{2.28}$$

quando sono noti K valori della variabile Y, detti *osservazioni della variabile dipendente*, e K valori di ciascuna variabile indipendente $X_1, ..., X_N$, detti *osservazioni della variabili indipendenti*. La tecnica di stima è basata sul *Metodo dei Minimi Quadrati*, sviluppato agli inizi del '800 da Legendre e Gauss come metodo di determinazione delle orbite di corpi celesti [31].

2.3.1 Il modello generico di regressione

Il generico modello di regressione lineare multivariata, detto *Modello di Spiegazione di Y*, può essere riassunto nella seguente formula:

$$\begin{aligned} Y_i = \beta_1 X_{i1} + \beta_2 X_{i2} + \cdots + \beta_N X_{iN} + \varepsilon_i = \\ \sum_{j=1}^{N} \beta_j X_{ij} + \varepsilon_i \qquad i = 1:K \quad j = 1:N \end{aligned} \tag{2.29}$$

dove:

- K è il numero di osservazioni delle variabili dipendenti $i = 1...K$
- N è il numero di *regressori* o variabili indipendenti $j = 1...N$
- Y_i rappresenta l'osservazione dell'i-esima variabile dipendente $i = 1...K$
- X_{ij} sono le i-esime osservazioni di ciascuno dei N regressori
- β_j sono i coefficienti di regressione, la stima dei quali è obiettivo della regressione

  ε_i sono i *residui* di stima o *errori statistici* associati alle K variabili dipendenti espressi come differenza tra i dati osservati Y_i e quelli predetti

$\overline{Y_i}$

$$\varepsilon_i = Y_i - \overline{Y_i} \quad i = 1...K \tag{2.30}$$

Raggruppando le osservazioni delle variabili esplicative in una matrice X di dimensioni $K \times N$ detta *Matrice di Segno*

$$\mathbf{X} = \begin{bmatrix} 1 & X_{1,1} & \cdots & X_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_{K,1} & \cdots & X_{K,N} \end{bmatrix} \tag{2.31}$$

raggruppando i coefficienti di regressione nel vettore

$$\mathbf{b} = \begin{pmatrix} \beta_1 \\ ... \\ \beta_N \end{pmatrix} \tag{2.32}$$

e raggruppando i residui nel *Vettore di Residui*

$$\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ ... \\ \varepsilon_K \end{pmatrix} \tag{2.33}$$

è possibile scrivere il *Modello di Spiegazione di* Y 2.29 in forma matriciale

$$\mathbf{Y} = \mathbf{X}\mathbf{b} + \boldsymbol{\varepsilon} \tag{2.34}$$

2.3.2 Ipotesi alla base del Modello e il Metodo dei Minimi Quadrati

Al fine di applicare la il metodo di regressione lineare, il modello in esame deve soddisfare alcune condizioni, ossia:

1. **Omoschedasticità dei Residui**:

Definizione 19 (Omoschedasticità). *Una distribuzione casuale di residui ε si dice* omoschedastica *quando la sua media è pari a zero e la sua varianza è costante ossia si mantiene invariata per ciascun residuo.*

L'omoschedasticità (dal greco, stessa varianza intesa come stessa varianza tra residui) è una condizione ideale nella quale si trova una funzione di dati rappresentabili graficamente come dispersi in modo piuttosto omogeneo al di sopra od al di sotto di una linea retta. Il termine deriva direttamente da *homoscedasticity* usato come sinonimo di omogeneità delle varianze e fu introdotto da Karl Pearson nel 1905. Nel modello di stima utilizzato l'omoschedasticità si traduce nelle condizione di distribuzione normale dei Residui, che espressa in formule risulta:

$$\boldsymbol{\varepsilon} \sim N(\mathbf{0}, \sigma^2 \boldsymbol{I}) \quad ossia \quad E[\varepsilon_i] = 0 \quad e \quad E[\varepsilon_i^2] = \sigma^2 \quad \forall i \in K \tag{2.35}$$

2. **Assenza di Correlazione nei residui**: $E[\varepsilon_i \varepsilon_j] = 0 \; \forall i \neq j \; ij = 1...K$

Assumendo soddisfatte tali condizioni, è possibile ottenere le stime del vettore dei coefficenti di regressione (i parametri del modello) $\mathbf{b}$ tramite il metodo dei minimi quadrati trovando il vettore $\hat{\mathbf{b}}$ che risolve il problema di minimo:

$$\min_{\hat{\mathbf{b}}} (\mathbf{Y} - \mathbf{X}\hat{\mathbf{b}})^T (\mathbf{Y} - \mathbf{X}\hat{\mathbf{b}}) \tag{2.36}$$

Le condizioni del primo ordine per un minimo definiscono il sistema (detto delle *equazioni normali*

$$-2\mathbf{X}\mathbf{Y} + 2\mathbf{X}^T\mathbf{X}\hat{\mathbf{b}} = 0 \tag{2.37}$$

da cui:

$$\hat{\mathbf{b}} = \hat{\mathbf{b}}_{OLS} = \begin{pmatrix} \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_N \end{pmatrix} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{2.38}$$

che è il vettore di stima ai minimi quadrati (OLS: Ordinary Least Squares) dei regressori del modello. Per le proprietà della forma quadratica minimizzanda, si è certi che la soluzione trovata corrisponde a un minimo, non solo locale ma

globale. Inoltre si può dimostrare che tale stimatore è uno stimatore *BLUE* ossia *Best, Linear, Unbiased Estimator*.

Sfruttando il vettore $\hat{\mathbf{b}}$ è possibile riscrivere la forma matriciale del modello di spiegazione di Y visto nell'Equazione 2.34 nella forma espressa dall'Equazione 2.39:

$$\mathbf{Y} = \mathbf{X}\hat{\mathbf{b}} + \varepsilon \tag{2.39}$$

Chiamando $\mathbf{X}\hat{\mathbf{b}} = \hat{\mathbf{Y}}$ intendendo con $\hat{\mathbf{Y}}$ il *vettore delle osservazioni stimato* l'Equazione 2.39 può essere ulteriormente riscritta nell'Equazione 2.40:

$$\mathbf{Y} = \hat{\mathbf{Y}} + \varepsilon \tag{2.40}$$

2.3.3 Stima della Varianza dei coefficenti di Regressione

Il metodo di regressione lineare multivariata ai minimi quadrati permette di calcolare in maniera agevole anche la varianza dei coefficenti di regressione, sfruttando la seguente relazione:

$$cov(\hat{\mathbf{b}}) = \boldsymbol{\sigma}^{\mathbf{2}}(\mathbf{X}^{\mathbf{T}}\mathbf{X})^{-\mathbf{1}} \tag{2.41}$$

dove con $\boldsymbol{\sigma}^2$ si è indicato il *Vettore delle Varianze dei Residui*

$$\boldsymbol{\sigma}^2 = var(\boldsymbol{\varepsilon}) \tag{2.42}$$

e con $cov(\hat{\mathbf{b}})$ si è indicata la *Matrice di Covarianza dello Stimatore* .Tale matrice ha forma:

$$cov(\hat{\mathbf{b}}) = \begin{bmatrix} var(\hat{\beta}_1) & cov(\hat{\beta}_1, \hat{\beta}_2) & \cdots & cov(\hat{\beta}_1, \hat{\beta}_N) \\ cov(\hat{\beta}_2, \hat{\beta}_1) & var(\hat{\beta}_2) & \cdots & cov(\hat{\beta}_2, \hat{\beta}_N) \\ \vdots & \vdots & \ddots & \vdots \\ cov(\hat{\beta}_N, \hat{\beta}_1) & cov(\hat{\beta}_N, \hat{\beta}_2) & \cdots & var(\hat{\beta}_N) \end{bmatrix} = \boldsymbol{\sigma}^{\mathbf{2}}(\mathbf{X}^{\mathbf{T}}\mathbf{X})^{-\mathbf{1}} \tag{2.43}$$

Analizzando la forma della Matrice di Covarianza si nota che la diagonale principale contiene le varianze associate ai regressori. Nell'Equazione 2.41 il vettore delle varianze dei residui è incognito.

Sotto l'ipotesi di omoschedasticità dei residui, il vettore incognito è in realtà costante e si può quindi effettuare una stima di tale vettore incognito utilizzando l'*SSR, Sum of Squared Residuals* o *RSS, Residual Sum of Squares* ossia la somma dei quadrati dei Residui Stimati per ogni variabile dipendente:

$$SSR = \sum_{i=1}^{K} [Y_i - (\hat{\beta}_1 X_{i1} + \cdots + \hat{\beta}_N X_{iN})]^2 = \sum_{i=1}^{K} [\varepsilon_i]^2 \tag{2.44}$$

Utilizzando l'SSR il vettore delle varianze dei residui può essere stimato usando lo stimatore:

$$\widehat{\boldsymbol{\sigma}}^2 = \frac{SSR}{K - N - 1} \tag{2.45}$$

Empiricamente, si è osservato che una tale stima è accurata solo se:

$$N \ll K \tag{2.46}$$

ossia se il numero di osservazioni della variabile dipendente è maggiore del numero di regressori di almeno un fattore 10. Sotto tale condizione si ha la proprietà di non distorsione dello stimatore:

$$E[\widehat{\boldsymbol{\sigma}}^2] = \boldsymbol{\sigma}^2 \tag{2.47}$$

Utilizzando lo stimatore appena calcolato si può effettuare una stima della Matrice di Covarianza sostituendo nell'Equazione 2.43 il risultato dell'Equazione 2.45 ottenendo:

$$\widehat{cov(\hat{\mathbf{b}})} = \widehat{\boldsymbol{\sigma}^2}(\mathbf{X}^T \mathbf{X})^{-1} \tag{2.48}$$

Estraendo la questa matrice la diagonale principale si ottiene il vettore di stima delle varianze associate ai regressori:

$$var(\widehat{\mathbf{b}}) = \begin{pmatrix} var(\hat{\beta}_1) \\ var(\hat{\beta}_2) \\ \vdots \\ var(\hat{\beta}_N) \end{pmatrix} \tag{2.49}$$

Scritto in maniera compatta, il vettore risulta essere:

$$var(\widehat{\mathbf{b}}) = \begin{pmatrix} var(\hat{\beta}_1) \\ var(\hat{\beta}_2) \\ \vdots \\ var(\hat{\beta}_N) \end{pmatrix} = diag\left(\boldsymbol{\sigma}^2(\mathbf{X}^T\mathbf{X})^{-1}\right) = diag\left(\frac{\sum_{i=1}^{K}[\varepsilon_i]^2}{K-N-1}(\mathbf{X}^T\mathbf{X})^{-1}\right) \tag{2.50}$$

dove con *diag* si intende l'operatore di estrazione delle diagonale principale dalla matrice a cui viene applicato.

2.3.4 Stima delle varianze delle osservazioni Y e intervalli di confidenza

Sfruttando il vettore delle varianze dei coefficienti di regressione, è possibile ottenere una stima della varianza associata a ogni singola componente Y_i del vettore delle osservazioni Y sfruttando l'Equazione 2.51:

$$var(Y_i) = var(\widehat{Y_i}) = \sum_{j=1}^{N} var(\widehat{b_j}X_{ij}) \tag{2.51}$$

Essendo ogni componente X_{ij} della matrice X costante, l'Equazione 2.51 viene semplificata nell'Equazione 2.52:

$$var(Y_i) = var(\widehat{Y_i}) = \sum_{j=1}^{N} var(\widehat{b_j})X_{ij}^2 \tag{2.52}$$

L'Equazione 2.52 può essere riscritta in forma matriciale nell'Equazione 2.53

$$var(\mathbf{Y}) = var(\widehat{\mathbf{Y}}) = var(\widehat{\mathbf{b}})\mathbf{X}.^{\mathbf{2}} \tag{2.53}$$

dove con l'operatore $\mathbf{X}.^{\mathbf{2}}$ si intende il prodotto scalare della matrice X per se stessa.

Essendo ogni j-esima componente del vettore $\hat{\mathbf{b}}$ una variabile aleatoria avente media $\hat{b}_j$ e varianza $var(\hat{b}_j)$ contenute rispettivamente nel vettore $\hat{\mathbf{b}}$ e nel

vettore $var(\hat{\mathbf{b}})$, se $\sum_{j=1}^{N} X_{ij} \quad \forall i = 1...K$ è elevato, ossia maggiore di 60 [31], è possibile applicare il *Teorema Centrale del Limite* [Teorema 16] e considerare ciascuna variabile aleatoria $\widehat{b_{ij}} = \widehat{b}_j X_{ij}$ come avente distribuzione normale $N(\widehat{b}_j X_{ij}, var(\widehat{b}_j) X_{ij}^2)$.

Inoltre, dal momento che la somma di variabili aleatorie normali si distribuisce come una normale [Teorema 2], si ha che anche la singola osservazione Y_i è una variabile aleatoria normale $N(Y_i, var(Y_i)\ ) = N(Y_i, \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2)$. Sfruttando la formulazione espressa dalle Equazioni 2.18, 2.19 e 2.20, è possibile definire degli intervalli di confidenza per ciascuna componente Y_i del vettore Y con le Equazioni 2.54, 2.55 e 2.56:

$$\text{Se } \frac{1+\gamma}{2} = 99.7\% \text{ si ha che } z_{\frac{1+\gamma}{2}} = 3 \text{ e quindi l'IC diventa:}$$

$$\left(Y_i - 3 \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2, Y_i + 3 \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2 \right) \tag{2.54}$$

$$\text{Se } \frac{1+\gamma}{2} = 95.5\% \text{ si ha che } z_{\frac{1+\gamma}{2}} = 2 \text{ e quindi l'IC diventa:}$$

$$\left(Y_i - 2 \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2, Y_i + 2 \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2 \right) \tag{2.55}$$

$$\text{Se } \frac{1+\gamma}{2} = 68.3\% \text{ si ha che } z_{\frac{1+\gamma}{2}} = 1 \text{ e quindi l'IC diventa:}$$

$$\left(Y_i - \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2, Y_i + \sum_{j=1}^{N} var(\widehat{b}_j) X_{ij}^2 \right) \tag{2.56}$$

2.3.5 Test sul Modello Utilizzato

Vengono ora presentati i test statistici ai quali viene sottoposto il modello al fine di verificare le condizioni presentate nella Sezione 2.3.2

Verifica Omoschedasticità dei Residui

La prima sezione di test è dedicata alla verifica dell'omoschedasticità dei residui.

1) Test sulla Media dei Residui: si verifica innanzitutto che la media dei residui non sia significativamente diversa da zero operando in uno dei seguenti modi:

- calcolando in maniera esplicita la media aritmetica dei residui:

$$\overline{\varepsilon} = \frac{\sum_{i=1}^{N} \varepsilon_i}{N}$$

  e verificando che sia circa zero ($< 10^{-6}$) [31]

- applicando il test di ipotesi *t-di-student* sulla media di popolazione gaussiana (avendo supposto la normalità dei residui) nel caso di varianza incognita al vettore dei residui:

  Ipotesi Nulla: $H_0 : \mu = 0$
  Ipotesi Alternativa: $H_1 : \mu \neq 0$
  Condizione di Rifiuto: rifiuto H_0 se

$$\frac{\overline{x} - 0}{s/\sqrt{N}} \geq t_{N-1}(1 - \alpha) \tag{2.57}$$

  dove con s si è indicata la varianza campionaria

$$s = \frac{1}{N-1} \sum_{i=1}^{N} (\varepsilon_i - \overline{\varepsilon})^2$$

  e con $t_{N-1}(1 - \alpha)$ si è indicato il *quantile di ordine di significatività 1-α* della distribuzione t-di-student con N-1 gradi di libertà. Valori di significatività considerati sufficienti sono circa del 95%.

2) Verifica di Normalità dei Residui: successivamente, si verifica la normalità della distribuzione degli errori con il test di *Shapiro-Wilk* [32]. Il test di Shapiro-Wilk è un test per la verifica di ipotesi statistiche ed è considerato in letteratura uno dei test più potenti per la verifica della normalità, soprattutto per piccoli campioni. Venne introdotto nel 1965 da Samuel Shapiro e Martin Wilk. La verifica della normalità avviene confrontando due stimatori alternativi della va-

rianza σ^2 che compaiono come numeratore e denominatore della statistica test mostrata nell'Equazione 2.58:

$$W = \frac{(\sum_{i=1}^{K} a_i \varepsilon_i)^2}{\sum_{i=1}^{K} (\varepsilon_i - \overline{\varepsilon})^2} \tag{2.58}$$

I pesi a_i della combinazione lineare sono disponibili su opportune tabelle contenute in [32]. La statistica W può assumere valori da 0 a 1. Sfruttando la tabella contenuta in [32] detta *Percentage points of the W test* tabulata in base della cardinalità K del campione, si verifica il punto percentuale in cui cade il valore W, approssimandolo al più vicino se il dato W non compare sulle tavole. Qualora il valore percentuale trovato sia inferiore a 0.10, il test rifiuta l'ipotesi nulla che i valori campionari siano distribuiti come una variabile casuale normale.

2.3.6 Verifica dell'Assenza di Correlazione fra i Residui

L'assenza di correlazione fra i residui può essere effettuata con il *Test di Ljung-Box e variante Box-Pierce*

Test di Ljung-Box e variante Box-Pierce: come il test di Durbin-Watson, il test di Ljung-Box e la sua variante Box-Pierce consentono la verifica dell'assenza di correlazione fra i residui, sfruttando due statistiche test leggermente differenti:

$$LB = N(N+2) \sum_{i=1}^{h} \frac{\varepsilon_i^2}{N-i} \tag{2.59}$$

$$BP = N \sum_{i=1}^{h} \varepsilon_i^2 \tag{2.60}$$

dove h è un intero prescelto. Se è vera l'ipotesi nulla (assenza di autocorrelazione) le statistiche convergeranno asintoticamente a una variabile casuale χ_{h-1}^2 chi-quadro con h gradi di libertà. Le due statistiche differiscono semplicemente per i coefficienti utilizzati per pesare i campioni. Si dimostra, tuttavia che LB ha una convergenza più rapida alla distribuzione asintotica e, per tale motivo, risulta preferibile al test di Box-Pierce.

2.3.7 Verifica della Significatività dei Regressori

Un importante parte di test concerne i singoli coefficienti del modello. Si vuole stabilire il livello di *significatività di un regressore*, ossia voler stabilire se la j-esima variabile della matrice X abbia o meno potere esplicativo nei confronti della Y. Ciò equivale a sottoporre a verifica l'ipotesi nulla che il corrispondente coefficiente $\hat{\beta}_j$ sia nullo. A tal fine si ricorre alla statistica test:

$$\hat{t} = \frac{\hat{\beta}_j}{\sqrt{\sum_{jj}}}$$

dove $\sum_{jj} = [\boldsymbol{\sigma}^2(\mathbf{X}^T\mathbf{X})^{-1}]_{jj}$. Sotto l'ipotesi nulla $H_0 : \hat{\beta}_j = 0$ $\sum_{jj}$ ha distribuzione t di Student. Dal valore della statistica test e attraverso le tavole della distribuzione t di Student o software di calcolo opportuni, si può ottenere il valore di significatività di ciascun regressore (il *p-value*). Più alto è il valore di significatività, maggiore sarà la capacità del regressore di spiegare la variabile dipendente.

2.3.8 Calcolo del Coefficiente di Determinazione R^2

Il modello risultante dall'applicazione del metodo di regressione può essere infine valutato attraverso il *Coefficiente dj Determinazione* R^2. Il coefficiente R^2 rappresenta la proporzione tra la variabilità dei dati e la correttezza del modello statistico utilizzato, ossia fornisce una misura di quanto accuratamente i valori della variabile indipendente Y possono essere predetti dal modello. Vi sono differenti definizioni del coefficiente R^2 a seconda del campo statistico in cui viene applicato; nell'ambito della regressione lineare, il coefficiente di determinazione è definito attraverso l'Equazione 2.61

$$R^2 = \frac{SSR}{\sum_{i=1}^{K}[Y_i - \overline{Y}]} \tag{2.61}$$

dove con $\overline{Y}$ si intende la media delle osservazioni della variabile dipendente, esprimibile tramite l'Equazione 2.62:

$$\overline{Y} = \frac{1}{K} \sum_{i=1}^{K} Y_i \tag{2.62}$$

mentre con SSR si intende la somma dei quadrati dei residui mostrata nell'Equazione2.44. Il coefficiente di determinazione può assumere valori da 0 a 1: valori prossimi a 1 indicano che il modello spiega perfettamente i dati attraverso i regressori, mentre valori prossimi a 0 indicano che il modello non spiega affatto i dati.

Capitolo 3

Definizioni Preliminari

Nella prima parte di questo Capitolo verranno descritte la struttura del compilatore GNU GCC [13] e le sue rappresentazioni intermedie, prestando particolare attenzione ai linguaggi GIMPLE ed RTL (*Register Transfer Language*).

Nella seconda parte verranno introdotti i concetti di ciclo (*Loop*), di *Blocco Basico* e di *Control Flow Graph*.

Nella parte conclusiva si descriveranno le tecniche di *software profiling*.

3.1 La Struttura del Compilatore GNU GCC

Il GNU Compiler Collection [13](abbreviato in GCC) è un compilatore prodotto dal *GNU Project* in grado di supportare vari linguaggi di programmazione. Tipicamente un compilatore è un software che prende in input un'applicazione scritta in codice sorgente e la trasforma in un'altra semanticamente equivalente, ma descritta in un differente linguaggio (codice oggetto). Il processo di compilazione trasforma il codice sorgente nel codice macchina attraverso tre fasi che coinvolgono le tre componenti principali del GCC: Front End, Middle End e Back End, mostrati in Figura 3.1.

Nella prima fase di *Front End*, il file sorgente viene analizzato e viene validata la sintassi del codice in esso contenuto. Se la validazione ha esito positivo, viene costruito l'*Abstract Syntax Tree* (AST) che rappresenta ogni istruzione del

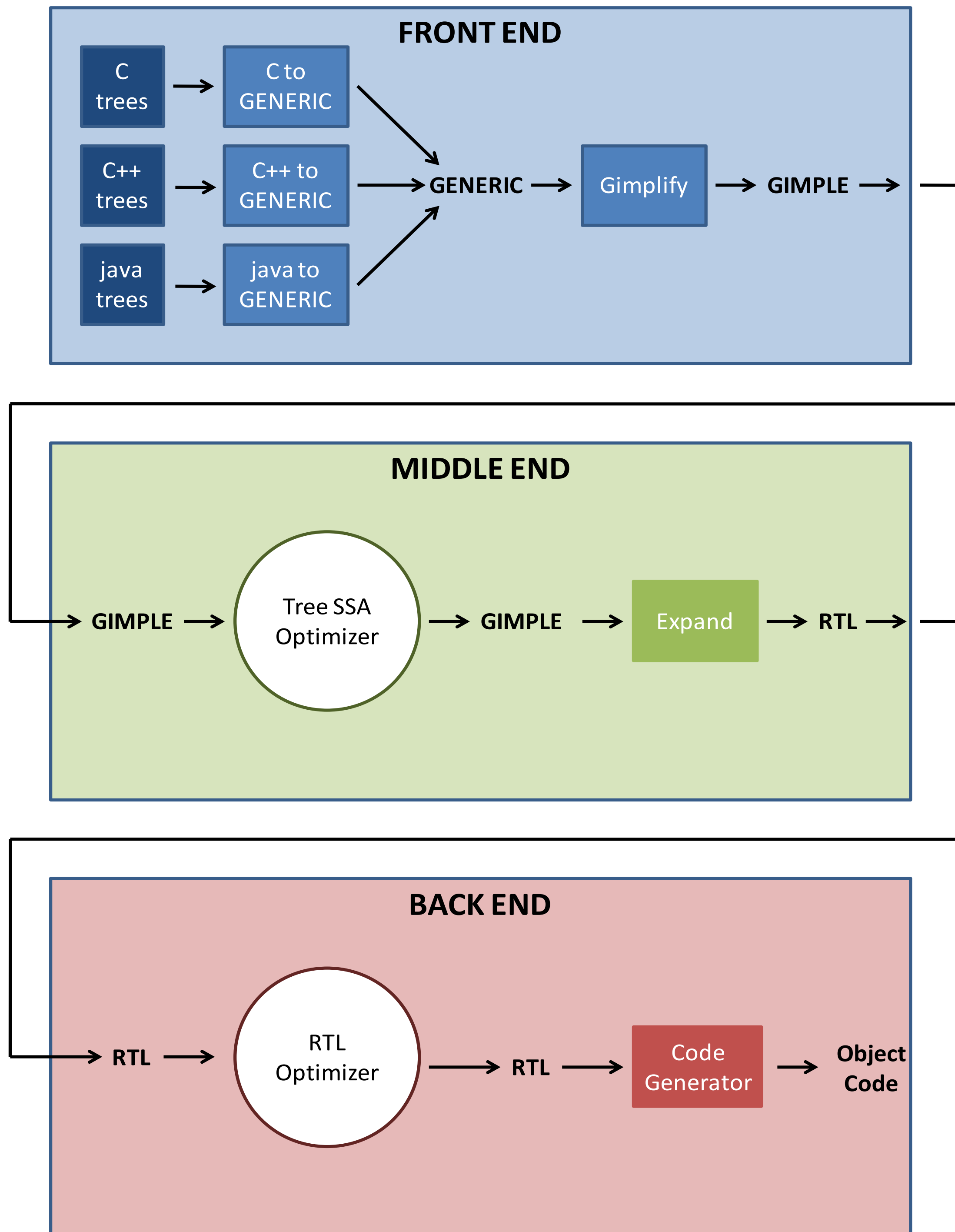

Figura 3.1. Struttura del compilatore GCC

programma da compilare in una struttura gerarchica ad albero attraverso l'uso della rappresentazione intermedia GENERIC. Data la complessità della struttura creata, la rappresentazione GENERIC viene tradotta in una struttura ad albero più semplice detta GIMPLE.

Nella fase di *Middle End*, partendo dalla rappresentazione intermedia GIMPLE, vengono effettuate ottimizzazioni del programma indipendenti dall'architettura. Due possibili esempi di ottimizzazioni eseguite in questa fase sono:

- *Semplificazioni Algebriche*: vengono utilizzate le proprietà algebriche per semplificare le espressioni. Ad esempio l'espressione $1 + i - 1$ viene semplificata in i;

- *Constant Folding*: la tecnica di constant folding si basa sul concetto che le espressioni che hanno come operandi solo valori costanti possono essere valutate durante la compilazione. Ad esempio l'espressione $a = 4 + 3 - 8$ può essere sostituita da $a = -1$.

Il compilatore in questa fase realizza ottimizzazioni indipendenti dall'architettura operando direttamente sulla rappresentazioni GIMPLE. Successivamente tale rappresentazione viene convertita in quella RTL sulla quale verranno fatte ulteriori ottimizzazioni.

La fase di *Back End* produce il codice macchina per l'architettura target. In questa fase il compilatore ha bisogno di conoscere le caratteristiche dell'architettura hardware che eseguirà il codice. Durante il processo di traduzione della rappresentazione intermedia nel codice macchina vengono applicate ulteriori ottimizzazioni dipendenti dall'architettura destinazione.

3.2 Le Rappresentazioni Intermedie del Compilatore GCC

In questa Sezione verranno presentate le tre rappresentazioni intermedie utilizzate dal compilatore GCC.

3.2.1 GENERIC

L'obiettivo di GENERIC è quello di fornire una rappresentazione ad albero del codice sorgente indipendente dal linguaggio. In GENERIC un'intera funzione del codice sorgente viene tradotta in una struttura ad albero, ossia in un Abstract Syntax Tree. In essa le funzioni e di conseguenza tutte le istruzioni del codice sorgente (dichiarazioni, assegnamenti, costrutti condizionali, etc.), sono tradotte in nodi dell'albero.

3.2.2 GIMPLE

Dalla rappresentazione GENERIC si ottiene quella GIMPLE, ossia un sottoinsieme semplificato della prima, esplicitamente progettata per essere utilizzata efficientemente nelle ottimizzazioni. Il passo del compilatore che genera la GIMPLE è detto gimplifier. Il gimplifier funziona ricorsivamente generando tuple GIMPLE dalle espressioni originarie di GENERIC. Di seguito saranno brevemente illustrati le caratteristiche principali di GIMPLE e in ultimo sarà mostrato un esempio di rappresentazione:

- **Espressioni**: in generale, un'espressione GIMPLE è una tupla composta da un operatore e da un numero massimo di tre semplici operandi; questi operandi possono essere sia costanti che variabili. Operazioni più complesse vengono fattorizzati in espressioni temporanee dal *gimplifier*, seguendo un processo conosciuto come *expression temporaries*. Per esempio l'espressione:

  ```
  a = b + c + d ;
  ```

  viene trasformata in

  ```
  T1 = b + c ;
  a = T1 + d ;
  ```

- **Espressioni Condizionali**: le istruzioni condizionali come '?' sono convertite in costrutti *if* . Ad esempio, l'istruzione:

```
a = b ? c : d
```

diventa

```
if (b)
   T1 = c;
else
   T1 = d;
a = T1;
```

- **Condizioni Logiche**: quando gli operandi logici *and* e *or* non appaiono come operandi di un'istruzione condizionale, sono semplificati come segue:

```
a = b && c
```

diventa

```
T1 = (bool)b
if T1 = TRUE ;
T1 = (bool)c ;
a = T1 ;
```

- **Loop**: nella versione 4.3 del GCC i loop sono rappresentati per mezzo di *if-goto* [13].

- **Istruzioni Condizionali**: una semplice istruzione condizionale come l'*if* viene tradotta in GIMPLE con l'espressione *COND_EXPR*. Quando l'espressione condizionale contiene formule logiche (AND, OR, etc.), per rispettare le regole della cortocircuitazione, viene trasformata in due o più *COND_EXPR*.

- **Salti Incondizionati**: i salti incondizionati sono espressi con le due espressioni *GOTO_EXPR* e *RETURN_EXPR*. Esempi di istruzioni di salto incondizionato nel linguaggio C sono: break, continue, goto, return.

```
void f(int n){
   int i, x;
   int j = (--i, i ? 0 : 1);
   for (x = 42; x > n; --x){
      i += x*4 + 32;
   }
   printf("%d",i);
}
```

Figura 3.2. Esempio di codice sorgente

3.2.3 RTL

La rappresentazione RTL (Register Transfer Language) è una Rappresentazione Intermedia utilizzata dal compilatore GCC nella fase di Back End, per eseguire le ottimizzazioni *target dependent*. Nonostante il linguaggio RTL (la sua grammatica) sia indipendente dall'architettura considerata, durante la traduzione dalla rappresentazione GIMPLE vengono presi in considerazione alcuni aspetti dell'hardware. Perciò le rappresentazioni RTL della stessa applicazione su diverse architetture possono differire fra loro anche significativamente. La rappresentazione RTL è composta da espressioni simboliche che consistono in una struttura a lista annidata.

In particolare la rappresentazione del corpo di una funzione è una sequenza di istruzioni di tipo `insn` o `jump_insn`. Ogni istruzione RTL è composta da una lista annidata di operazioni RTL: un operazione RTL è principalmente caratterizzata da un operatore (ad esempio `plus` , `minus`), un tipo di dato (ad esempio intero a singola precisione) alcuni operandi (ad esempio registri o risultati di altre operazioni RTL annidate) e annotazioni.

In Figura 3.3 viene mostrata la rappresentazione GIMPLE del codice sorgente di Figura 3.2.

```
; Function f (f)

;; f (n)
{
   int x;
   int i;
   int D.752;
   int D.751;

<bb 2>:
   i_4 = i_3(D) + -1;
   x_6 = 42;
   goto <bb 4>;

<bb 3>:
   D.751_8 = x_2 + 8;
   D.752_9 = D.751_8 * 4;
   i_10 = D.752_9 + i_1;
   x_11 = x_2 + -1;

<bb 4>:
   # x_2 = PHI <x_6(2), x_11(3)>
   # i_1 = PHI <i_4(2), i_10(3)>
   if (x_2 > n_7(D))
      goto <bb 3>;
   else
      goto <bb 5>;

<bb 5>:
   # i_12 = PHI <i_1(4)>
   printf (&"%d"[0], i_12);
   return;
}
```

Figura 3.3. Rappresentazione GIMPLE del codice sorgente in Figura 3.2

```
(set:SI (reg:SI 140) (plus:SI (reg:SI 138) (reg:SI 139)))
```

Figura 3.4. Esempio di codice RTL

I Modi Macchina

Ad ogni operazione RTL viene associato un *modo macchina* che associa alla singola operazione la dimensione, il tipo e la rappresentazione dei suoi operandi. Nelle operazioni RTL, il modo macchina viene scritto dopo l'expression code e preceduto dal carattere separatore ':'. Per esempio, (`mult:SI`) rappresenta l'espressione `mult` (moltiplicazione) con il modo `SImode` (ossia intero a singola precisione a 32-bit). Se l'operazione RTL è priva di operandi, il modo macchina viene omesso.

Esempio di rappresentazione RTL

Per facilitare la lettura, si illustrerà prima una singola istruzione RTL. Successivamente sarà mostrata la rappresentazione RTL generata dalla compilazione di una serie di istruzioni sorgenti.

Nel codice in Figura 3.4, l'espressione *plus* somma il contenuto del registro 138 al contenuto del registro 139. Il risultato viene salvato, dall'espressione set nel registro 140.

Possiamo adesso mostrare un esempio di confronto fra le rappresentazioni RTL per il processore ARM (in Figura 3.6) e SPARC (in Figura 3.7). Il codice sorgente di entrambe è riportato in Figura 3.5. Come si può vedere le due rappresentazioni RTL sono differenti. Ad esempio, la prima istruzione viene tradotta con tre somme (`plus:SI`) nel caso di processore ARM, mentre per il processore SPARC la stessa istruzione si implementa con una sola somma.

```
   int func(int a, int b){
1:    a = a + 10000000;
2:    b = b + 10;
3:    a = a + b;
4:    return 1 - a*b;
   }
```

Figura 3.5. Codice Sorgente per la traduzione nella rappresentazione RTL

```
;;  Function func (func)
1: (insn 8 7 9 source.c:3 (set (reg:SI 142)
         (const_int 9961472 [0x980000])) -1 (nil))
1: (insn 9 8 10 source.c:3 (set (reg:SI 143)
         (plus:SI (reg:SI 142)
            (const_int 38400 [0x9600]))) -1 (nil))
1: (insn 10 9 11 source.c:3 (set (reg:SI 141)
         (plus:SI (reg:SI 143)
            (const_int 128 [0x80]))) -1 (expr_list:
            REG_EQUAL (const_int 10000000 [0x989680])(nil)))
1: (insn 11 10 0 source.c:3 (set (reg/v:SI 135 [ a.4 ])
         (plus:SI (reg/v:SI 139 [ a ])
            (reg:SI 141))) -1 (nil))
2: (insn 7 6 0 source.c:4 (set (reg/v:SI 134 [ b.5 ])
         (plus:SI (reg/v:SI 140 [ b ])
            (const_int 10 [0xa]))) -1 (nil))
3: (insn 12 11 0 source.c:5 (set (reg/v:SI 133 [ a.6 ])
         (plus:SI (reg/v:SI 135 [ a.4 ])
            (reg/v:SI 134 [ b.5 ]))) -1 (nil))
4: (insn 13 12 0 source.c:6 (set (reg:SI 136 [ D.1176 ])
         (mult:SI (reg/v:SI 134 [ b.5 ])
            (reg/v:SI 133 [ a.6 ]))) -1 (nil))
4: (insn 14 13 0 source.c:6 (set (reg:SI 137 [ D.1175 ])
         (minus:SI (const_int 1 [0x1])
            (reg:SI 136 [ D.1176 ]))) -1 (nil))
4: (insn 15 14 16 source.c:6 (set (reg:SI 138
         [ <result> ]) (reg:SI 137 [ D.1175 ])) -1 (nil)
4: (jump_insn 16 15 17 source.c:6 (set (pc)
         (label_ref 0)) -1 (nil))
```


Figura 3.6. Esempio di Rappresentazione RTL per processori ARM

3.3 Rappresentazioni Intermedie basate su Grafi

Le rappresentazioni intermedie del compilatore GCC viste nella Sezione 3.2 hanno la caratteristica di poter essere espresse graficamente tramite l'utilizzo di grafi orientati.

Nella Sezione 3.3.1 verrà definito il concetto di *blocco basico*

Nelle Sezione 3.3.2 verrà mostrata una possibile rappresentazione intermedia basata su grafo detta *Control Flow Graph* (CFG).

Nella Sezione 3.4 verrà discusso della tecnica di *software profiling*.

3.3.1 I Blocchi Basici

Un blocco basico è una sequenza di operazioni (RTL o GIMPLE) che gode delle seguenti proprietà:

- la prima operazione è una destinazione di un salto;
- l'ultima operazione è un salto;
- non vi sono altre operazioni di salto o destinazione (operazioni etichettate) all'interno della sequenza.

Pertanto se un'operazione di un blocco basico viene eseguita in una certa traccia, tutte le operazioni di quel blocco basico verranno eseguite in quella traccia.

3.3.2 Control Flow Graph

I blocchi basici di una descrizione di sistema possono a loro volta costituire i nodi di un grafo, chiamato *Control Flow Graph* o CFG. Formalmente il CFG è un grafo orientato $G_d(V, E)$ dove:

- $V = v_i; i = 1, 2, ..., b$ è l'insieme dei nodi che si trovano in relazione uno a uno con i blocchi basici;
- $E = (v_i, v_j); i, j = 1, 2, ..., b$ è l'insieme degli archi orientati del grafo: esiste un arco tra v_i e v_j se:

```
;; Function func (func)
1: (insn 8 7 9 source.c:3 (set (reg:SI 116)
         (const_int 9999360 [0x989400])) -1 (nil))
1: (insn 9 8 10 source.c:3 (set (reg:SI 115)
         (ior:SI (reg:SI 116)
            (const_int 640 [0x280]))) -1
               (expr_list:REG_EQUAL
                  (const_int 10000000 [0x989680])
         (nil)))
1: (insn 10 9 0 source.c:3 (set (reg/v:SI 109 [ a.6 ])
         (plus:SI (reg/v:SI 113 [ a ])
            (reg:SI 115))) -1 (nil))

2: (insn 7 6 0 source.c:4 (set (reg/v:SI 108 [ b.7 ])
         (plus:SI (reg/v:SI 114 [ b ])
            (const_int 10 [0xa]))) -1 (nil))
3: (insn 11 10 0 source.c:5 (set (reg/v:SI 107 [ a.8 ])
         (plus:SI (reg/v:SI 109 [ a.6 ])
            (reg/v:SI 108 [ b.7 ]))) -1 (nil))
4: (insn 12 11 13 source.c:6 (set (reg:SI 8 %o0)
         (reg/v:SI 107 [ a.8 ])) -1
            (insn_list:REG_LIBCALL 15 (nil)))
4: (insn 13 12 14 source.c:6 (set (reg:SI 9 %o1)
         (reg/v:SI 108 [ b.7 ])) -1 (nil))
4: (insn 15 14 16 source.c:6 (set (reg:SI 117)
         (reg:SI 8 %o0)) -1 (insn_list:REG_RETVAL 12
            (expr_list:REG_EQUAL
               (mult:SI (reg/v:SI 107 [ a.8 ])
                  (reg/v:SI 108 [ b.7 ]))
               (nil))))
4: (insn 16 15 0 source.c:6 (set (reg:SI 110 [ D.774 ])
         (reg:SI 117)) -1 (nil))
4: (insn 17 16 18 source.c:6 (set (reg:SI 118)
         (const_int 1 [0x1])) -1 (nil))
4: (insn 18 17 0 source.c:6 (set (reg:SI 111 [ D.773 ])
         (minus:SI (reg:SI 118)
            (reg:SI 110 [ D.774 ]))) -1 (nil))
4: (insn 19 18 20 source.c:6 (set (reg:SI 112 [ <result> ])
         (reg:SI 111 [ D.773 ])) -1 (nil))
4: (jump_insn 20 19 21 source.c:6 (set (pc)
         (label_ref 0)) -1 (nil))
```

Figura 3.7. Esempio di Rappresentazione RTL per processori SPARC

```
int funct(int a, int b)
    if(a>=0){
        b--;
    }else{
        a++;
    }
    return b+a;
```

Figura 3.8. Esempio di codice sorgente

- la prima operazione del j-esimo blocco basico segue l'ultima operazione dell'i-esimo blocco basico;
- la prima operazione del j-esimo blocco basico è una delle possibili destinazioni del salto con cui termina l'i-esimo blocco basico.

In Figura 3.9 viene riportato un esempio di Control Flow Graph creato a partire dei blocchi basici associati al codice sorgente in Figura 3.8.

Archi di Feedback

In un grafo di flusso di controllo possono venire esplicitati gli archi di $feedback$ presenti nel corpo dei loop. Gli archi di feedback sono quegli archi che chiudono un percorso circolare avente origine nel nodo che rappresenta l'inizio della funzione. Un CFG contenente anche archi di feedback tra i blocchi basici è detto *Feedback Control Floaw Graph* o FCFG.

In Figura 3.11 è mostrato l'FCFG dei blocchi basici associati al codice sorgente mostrato in Figura 3.10, dove è evidenziato l'arco di feedback tra il blocco basico `BB3` e il blocco basico `BB4`.

3.4 Il Software Profiling

Il *software profiling*, o più semplicemente profiling, è una forma di analisi dinamica del programma per ottenere informazioni sulla sua esecuzione.

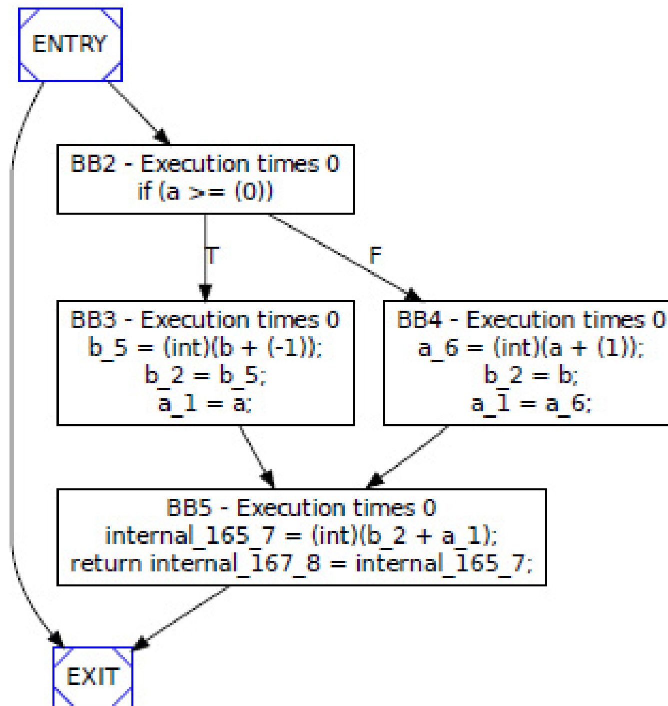

Figura 3.9. CFG del blocchi basici del codice in Figura 3.8

```
int funct2(int A[])
for (i = 0 ; i < X*Y; i++)
    A[i] = 1 ;
return((int)0) ;
```

Figura 3.10. Esempio di codice sorgente

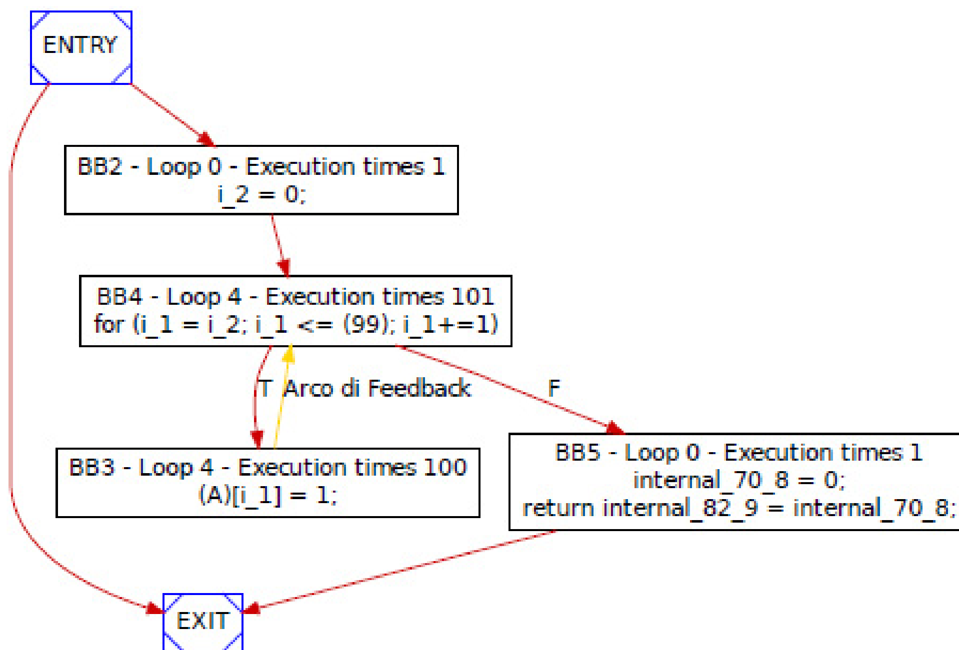

Figura 3.11. CFG del codice in Figura 3.10

Durante il profiling il codice viene eseguito su un processore reale o virtuale. Perché questa tecnica sia efficace, il programma target deve essere eseguito su un insieme significativo di dati di input.

L'applicazione che effettua l'analisi è chiamata *profiler*. Il profiler è in grado di raccogliere diversi tipi di informazioni riguardanti l'esecuzione del codice. Sono stati sviluppati diversi tipi di profiler, ma a grandi linee è possibile classificarli in tre categorie: profiler basati su campionamento, profiler basati su eventi e quelli basati sulle instrumentazioni.

I profiler del primo tipo controllano il *program counter* del programma target a intervalli regolari usando gli *interrupt* del sistema operativo. Ogni volta che il programma viene interrotto si prende nota del pezzo di codice che si sta eseguendo in quel momento. Alla fine dell'esecuzione si utilizzano questi dati per determinare il tempo di esecuzione dei vari pezzi di codice del programma. I profiler di questo tipo sono meno accurati di quelli che utilizzano altre tecniche, perché i dati non sono esatti, ma sono un'approssimazione statistica. Comunque nella pratica i profiler basati su campionamento possono spesso fornire una fo-

tografia dell'esecuzione dell'applicazione target più veritiera degli altri approcci, perché non sono intrusivi nell'esecuzione del programma. Grazie a questo gli effetti collaterali, come ad esempio quelli sulla cache, sono limitati. Tuttavia, con questa tecnica si potrebbero perdere degli eventi troppo brevi rispetto al tempo di campionamento, come, ad esempio, l'esecuzione di piccole funzioni.

I profiler basati su eventi, invece, si attivano quando nel programma profilato accade un evento come ad esempio l'entrata o l'uscita da una funzione. Questo tipo di profiler rispetto al precedente riesce a fornire dettagli di più basso livello, ma genera più overhead nell'esecuzione del programma target.

Infine, l'ultimo tipo di profiler instrumenta il codice del programma target con delle istruzioni aggiuntive. Le instrumentazioni possono essere molto specifiche e facilmente controllate, in modo che il loro impatto sulle prestazioni sia minimo. Tale impatto dipende dalla posizione in cui le instrumentazioni vengono inserite all'interno del programma e dal meccanismo usato per ottenere la traccia di esecuzione del programma (*execution trace*). Inoltre l'errore introdotto dalla instrumentazione spesso può essere dedotto ed eliminato dai risultati.

Le operazioni instrumentate possono essere introdotte con diverse strategie (manualmente o in automatico) e a diversi livelli (nel codice sorgente, all'interno della compilazione, oppure direttamente negli eseguibili).

Per completezza va detto che la classificazione presentata non è rigida ed infatti vari profiler sono stati progettati con soluzioni ibride, ad esempio utilizzando sia l'instrumentazione che il campionamento: la prima per conoscere le chiamate a funzione, il secondo, invece, per ottenerne il tempo di esecuzione.

Capitolo 4

Analisi dello Stato dell'Arte nella Stima di Prestazioni

In questo capitolo verrà mostrata una panoramica sullo stato dell'arte relativo alla Stima di Prestazioni per processori Embedded, con particolare attenzione alla definizione di intervalli di confidenza per il tempo di esecuzione.

In letteratura il problema della stima delle prestazioni di un'applicazione è stato affrontato secondo due approcci differenti. Da un lato sono state sviluppate tecniche basate su simulatori, che verranno illustrate nella Sezione 4.1, dall'altro tecniche basate su modelli matematici, che verranno presentate nella Sezione 4.2.

4.1 Tecniche di Stima Basate su Simulatore

Le tecniche di Stima di Prestazioni basate su Simulatore richiedono l'esecuzione dell'applicazione su un simulatore *cycle-accurate* ovvero un simulatore che riproduce il comportamento dell'architettura target ciclo di clock per ciclo di clock. Esempio di simulatori cycle-accurate sono *Legion Sparc* [19] che è in grado di simulare architetture basate su processori SPARC, e SimIt-ARM [1], che è in grado di simulare architetture basate su processori ARM.

Tali tecniche risultano essere molto accurate, ma si basano sulla conoscenza approfondita dell'architettura target. Soffrono quindi di mancanza di generalità e di portabilità su differenti piattaforme hardware: se si modifica l'architettura target, è necessario modificare anche il simulatore e per questo le tecniche basate su simulatore risultano inadatte ad essere impiegate in progettazioni di tipo iterativo e, nello specifico, in progettazioni di sistemi dedicati dove tipicamente il software è co-progettato insieme all'hardware.

Inoltre, le tecniche basate su simulatore risultano essere lente poiché richiedono la simulazione del comportamento dell'intera architettura (processore, gerarchie di memoria ecc..) durante l'esecuzione dell'applicazione di cui si vuole effettuare la stima. Per ovviare a queste limitazioni spesso si utilizzano simulatori meno accurati e perciò più veloci da realizzare, con conseguente però perdita di precisione nella stima.

4.2 Tecniche di Stima Basate su Modelli Matematici

Le tecniche di stima basate su modelli matematici consentono un'analisi dell'applicazione e dell'architettura target a diversi livelli di astrazione. L'analisi viene condotta al fine di estrapolare determinate caratteristiche significative per la stima, caratteristiche che vengono poi utilizzate per la costruzione di modelli che descrivono le prestazioni dell'applicazione su una specifica architettura obiettivo.

L'analisi può essere statica oppure dinamica (profiling attraverso l'esecuzione del codice su una macchina host) ed il programma può essere considerato a livello di codice sorgente o a livello di una delle differenti rappresentazioni intermedie del processo di compilazione.

Nel seguito verranno presentati i principali lavori che utilizzano tecniche basate su modelli matematici e la loro possibile estendibilità alla stima della varianza del tempo di esecuzione. Tutti gli studi descritti in questa sezione effettuano una fase di analisi su un insieme di benchmark per ottenere caratteristiche rilevanti per le prestazioni. Tali caratteristiche vengono correlate con i

reali tempi di esecuzione dei benchmark per ottenerne i costi. In ultimo, con i costi delle caratteristiche è possibile stimare nuove applicazioni.

4.2.1 La stima di prestazioni con intervalli di confidenza secondo Jantsch e Bjuréus

Il primo lavoro presentato, sul quale si basa il presente lavoro di tesi, è quello di *Per Bjureus* e *Axel Jantsch* [9]. E' una metodologia di stima di prestazioni tramite intervalli di confidenza basata su un modello di *Instruction Set* (IS) detto *Virtual Instruction Set* (VIS), in cui a ogni istruzione elementare contenuta nell'IS dell'architettura obiettivo viene associata una variabile stocastica avente distribuzione di probabilità incognita ma media μ e varianza σ^2 nota, dove la media μ rappresenta il tempo di esecuzione in cicli della singola istruzione e σ rappresenta la dispersione statistica attorno a tale valor medio.

Nella Tabella 4.2.1 viene riportato il VIS proposto dai due autori (derivato in maniera empirica attraverso un piccolo insieme di benchmark contenenti il programma da stimare stesso) relativo all'architettura ARM7TDMI.

Instruction	μ	σ^2
ADD	1.5	0.5
LOOP	5.0	0.5
LE	5.5	0.5
FLOOR	1.0	0.1
MUL	1.5	1.0
SUB	1.5	0.5
GT	5.5	1.0
DIV	1.5	1.0
subsref	2.0	0.2

Tabella 4.1. Il modello probabilistico utilizzato da Jantsch e Bjuréus

Il modello proposto sfrutta le cosiddette *tracce di esecuzione* o *Execution Trace*. Secondo la definizione fornita, una traccia di esecuzione è una lista di operazioni a cui corrisponde un istante di esecuzione, non decrescente.

Secondo il modello proposto quindi il tempo di esecuzione di ciascun programma è dato dalla somma dei tempi di esecuzione di ciascuna istruzione ele-

mentare che compone la traccia di esecuzione. Essendo ciascuna istruzione modellizzata con l'uso di una variabile stocastica, allora anche il tempo complessivo di esecuzione del programma può essere modellizzato con una variabile stocastica, secondo la seguente formulazione:

$$t = \sum_{\forall op} \sum_{k=1}^{n_{op}} \xi_k^{op} \tag{4.1}$$

dove:

1. n_{op} è il numero di volte in cui l'operazione di tipo op-esimo è eseguita;

2. ξ_k^{op} denota il tempo di esecuzione della istruzione di tipo op-esimo;

3. $\sum_{k=1}^{n_{op}} \xi_k^{op}$ denota la somma del tempo di esecuzione di tutte le volte in cui l'istruzione di tipo op-esimo viene eseguita;

4. $\sum_{\forall op}$ somma i tempi di esecuzione per ogni tipo di istruzione del VIS.

Se si considera la sommatoria interna, dove vengono sommati i tempi di esecuzione di ogni operazione

$$t_{op} = \sum_{k=1}^{n_{op}} \xi_k^{op} = \xi_1^{op} + \xi_2^{op} + ... + \xi_{n_{op}}^{op} \tag{4.2}$$

e si sfrutta il Teorema 3 della Sezione 2 si ha che $t_{op} \sim N(n\mu_{op}, n\sigma_{op}^2)$. Assumendo ulteriormente che ogni t_{op} sia indipendente l'una dall'altra e applicando il Teorema 2 si ha che la distribuzione della variabile stocastica associata al tempo totale di esecuzione si distribuisce come una normale:

$$t \sim N\left(\sum_{\forall op} n_{op}\mu_{op}, \sum_{\forall op} n_{op}\sigma_{op}^2\right) \tag{4.3}$$

Nel caso in cui n_{op} non sia elevato, n_{op} non avrà distribuzione normale. Il Teorema [1] della Sezione 2 garantisce che il tempo medio di esecuzione possa essere visto sempre come somma dei tempi medi di esecuzione di ciascuna istruzione. La non normalità impone però di usare metodi particolari per

la ricerca degli intervalli di confidenza, metodi che i due autori accennano solamente.

Tramite l'utilizzo del framework *MASCOT* [8] e in particolare del tool *EstiMate* [8], sviluppato dagli autori, il codice sorgente in ingresso viene analizzato e tradotto in tracce di esecuzione, composte dalle istruzioni elementari. Le tracce di esecuzione vengono analizzate e, sommando il tempo medio di esecuzione di ciascuna istruzione secondo l'Equazione 4.1, si ottiene la stima, in media e varianza, del tempo totale di esecuzione della traccia.

Da questa stima, viene generato un intervallo di confidenza per il tempo medio totale di esecuzione.

Dopo la fase di stima, il framework MASCOT simula il codice sorgente in ingresso, ottenendo il tempo di esecuzione reale del codice utilizzato. Il risultato di simulazione viene quindi confrontato con l'intervallo di confidenza ottenuto in fase di stima, calcolando l'errore relativo e effettuando una stima dell'accuratezza tramite il confronto dello scostamento tra il punto medio dell'intervallo di confidenza e il valore reale simulato.

4.2.2 Considerazioni sulla metodologia

La metodologia di stima proposta da Jantsch e Bjuréus presenta aspetti positivi e negativi. Per quanto riguarda i primi, la metodologia proposta:

- ha carattere molto generale per quanto concerne la distribuzione di probabilità con cui sono modellizzabili le singole istruzioni: in virtù del Teorema 3, se le operazioni sono numerose è garantita la gaussianità asintotica e la validità del metodo di stima;

- è estremamente semplice stimare un intervallo di confidenza per la media del tempo di esecuzione essendo il tempo di esecuzione distribuito (asintoticamente) come una normale. Ad esempio un intervallo di confidenza di livello $\gamma 99.7\%$ è $(\mu - 3\sigma, \mu + 3\sigma)$;

- consente di modellizzare particolari architetture tramite la semplice modifica del VIS.

La metodologia proposta presenta però il seguente limite: il VIS a cui viene associato un tempo medio e una varianza di esecuzione alle singole istruzioni dell'instruction set, è stato derivato in maniera troppo semplicistica: utilizzare un piccolo set di benchmark, tra cui il programma da stimare stesso, conduce, come dimostrato dagli autori, a risultati buoni per la stima del programma stesso. Se si utilizza però lo stesso modello probabilistico per effettuare la stima su altri programmi, il modello proposto rischia di mancare di generalità e condurre a risultati di stima insoddisfacenti.

4.2.3 Le basi della presente tesi

Il lavoro svolto in questa tesi ha preso avvio da un progetto realizzato da *Fabrizio Ferrandi* e *Marco Lattuada*, [21]. Gli autori hanno sviluppato un modello di stima a partire da un insieme di benchmark, sfruttando un tool di simulazione, la tecnica di regressione lineare e una versione modificata del compilatore GCC.

Sfruttando il simulatore e il compilatore, l'intero insieme di benchmark viene compilato e simulato. Per ogni benchmark vengono registrati il numero di cicli totale di esecuzione e il numero di volte in cui ciascuna operazione RTL generata dal compilatore viene eseguita all'interno del codice.

Grazie a questi dati e alla regressione lineare, gli autori sono stati in grado di fornire una stima il tempo di esecuzione, in termini di cicli di clock, per ogni istruzione RTL generata dal GCC.

La metodologia appena descritta, grazie all'uso delle operazioni RTL, permette di considerare alcuni aspetti architetturali del processore target che altrimenti non potrebbero essere considerati, utilizzando una rappresentazione di più alto livello.

Il mantenimento della correlazione tra RTL e la rappresentazione indipendente dall'architettura GIMPLE permette alla metodologia di rimanere sufficientemente flessibile, tanto da essere adottata, senza necessità di modifiche, per altre architetture supportate dal compilatore GCC.

In conclusione, nel lavoro gli autori hanno realizzato un modello abbastanza accurato da riuscire a stimare le prestazioni di un insieme eterogeneo di bench-

mark. Tale modello rappresenta il punto di partenza e di sviluppo fondamentale per il presente lavoro di tesi.

4.2.4 Altre tecniche di stima basate su modelli matematici

Suzuki, *Sangiovanni* e *Vincentelli* [34] propongono una metodologia in cui tempo di esecuzione del processore target viene stimato sfruttando un semplice modello additivo basato sulle istruzioni ad alto livello del linguaggio C. Il costo di ogni istruzione è ottenuto eseguendo un insieme di benchmark su un ISS (Instruction Set Simulator) ed estraendo il numero medio di cicli. Questo lavoro ha qualche limite, ossia non sono considerate le ottimizzazioni del compilatore e le caratteristiche dell'architettura, la stima può essere applicata solo a programmi con una struttura molto semplice senza, ad esempio, cicli e ricorsione e non è possibile effettuare stima della varianza di esecuzione a meno di non modificare l'ISS.

Liu, *Lajolo* e *Vincentelli* [22] misurano, invece, le prestazioni della prima esecuzione di ogni blocco basico in codice C usando un ISS. Le successive esecuzioni di ogni blocco basico sono direttamente misurate o predette a seconda delle caratteristiche del blocco. Questo lavoro presenta però il limite di richiedere l'utilizzo di un simulatore.

Lajolo, *Lazarescu* e *Vincentelli* [20] utilizzano il compilatore GNU GCC [13] per effettuare un'analisi temporale sul codice assembly, aggiungendo al codice assembly derivato dal codice C di partenza delle annotazioni dei tempi di esecuzione delle singole istruzioni. Il codice C di partenza viene quindi rigenerato con però l'aggiunta delle annotazioni temporali. Il tempo di esecuzione delle istruzioni assembly sul processore target è ottenuto dai file descrittori della macchina (*machine description files*) del GCC. Il codice C viene compilato ed eseguito per ottenere la stima delle prestazioni del programma. I limiti di tale approccio sono la necessità di una pesante instrumentazione essendo un approccio a basso livello e che dai machine description files non è possibile ottenere degli intervalli di confidenza. Ulteriori limiti di questo lavoro sono l'avere considerato solo tre benchmark di controllo con aritmetica intera, il fatto che la rigenerazione del codice C originale richiede la conoscenza dell'Instruction Set

di ogni processore che si desideri stimare e che la stima di singole parti di codice non è facile da effettuare e richiede la riscrittura sostanziale del codice.

Il lavoro presentato da *Bammi et al.* [6] introduce il concetto di istruzioni virtuali per la stima basata sul codice sorgente. L'applicazione viene compilata in una rappresentazione intermedia che ha tutte le operazioni rappresentative di un processore RISC. Il codice risultante viene arricchito di annotazioni temporali che servono a fornire statistiche sull'esecuzione delle istruzioni virtuali e successivamente viene ritradotto in C, compilato ed eseguito sulla macchina host. Il costo di ogni istruzione virtuale sul processore target è ottenuto attraverso due differenti tecniche: derivando i cicli dal manuale del processore o dal suo ISS, oppure mediante l'uso di tecniche statistiche. Per l'utilizzo delle tecniche statistiche occorre calcolare i tempi di esecuzione di un insieme di benchmark. Tali tempi vengono successivamente correlati, utilizzando la regressione lineare, alle statistiche di esecuzione di ogni istruzione virtuale. Sebbene il modello proposto sia facilmente adattabile alla stima di intervalli di confidenza per il tempo di esecuzione, poiché può fare uso della statistica e in particolar modo utilizza la regressione lineare, non riesce a prevedere gli effetti delle ottimizzazioni del compilatore. Per risolvere il problema gli autori sono ricorsi nuovamente ad una stima a livello assembly. Infine la strategia è stata convalidata solo su applicazioni di tipo produttore/consumatore.

Brandolese et al. [11] usano il concetto di *atomi* per descrivere gli elementi basici di un codice sorgente. L'idea chiave è che le prestazioni di un'applicazione possono essere modellizzate come la somma dei ritardi di esecuzione di ogni atomo. Il ritardo stimato di un atomo è espresso come somma di due contributi: un tempo di riferimento, che prende in considerazione tutti gli aspetti deterministici in condizioni ideali, ed una *deviazione statistica* che dipende dagli aspetti dell'architettura e del compilatore. I limiti di questa metodologia risiedono nella complessità nell'ottenere, per ogni atomo, i due contributi descritti in precedenza.

Beltrame et al. [7] propongono un approccio flessibile per stimare le prestazioni di applicazioni su architetture super-scalari. In questo lavoro i cicli per istruzione di ogni operazione assembly sono modellizzati come combinazione di tre contributi: un termine fisso per il ritardo nominale di esecuzione

di un'operazione, un termine statistico che prende in considerazione gli overhead dovuti agli stalli e un coefficiente di parallelismo che modellizza quanto del parallelismo teorico venga effettivamente sfruttato. I limiti di questa metodologia consistono nei dati richiesti per calcolare i parametri del modello; infatti, per ogni istruzione sono necessari i tempi di inizio e di fine esecuzione al fine di costruire il modello.

Anche *Giusto e Martin* [16] utilizzano un insieme di istruzioni virtuali. Nel loro lavoro il costo di un'istruzione virtuale si stima applicando un metodo di regressione lineare a un insieme di benchmark *(training set)*. Con l'utilizzo delle istruzioni virtuali però un approccio di tipo lineare è accurato solo quando le applicazioni del training set sono simili a quelle che si vogliono stimare.

Bontempi e Kruijtzer [10] usano un modello non lineare per la stima di un intero programma. Viene effettuato un profiling con il tool *IPROF* per ottenere statistiche sull'esecuzione dei programmi su un processore virtuale con un insieme di 42 istruzioni. Per la creazione del modello viene utilizzata la tecnica di *lazy learning*: viene costruita una funzione di stima secondo il criterio della vicinanza tra l'applicazione e il training set. Anche questo approccio risente della mancanza di dettagli architetturali, poiché vengono usate istruzioni virtuali. Nel lavoro sono stati utilizzati sei benchmark con 15 differenti insiemi di dati.

Oyamada et al. [27][28] adottano un altro modello non lineare basato sulle reti neurali. L'approccio è accurato, ma è basato sul set di istruzioni del processore target. In questo modo, per ogni processore considerato è necessario utilizzare non solo una rete neurale addestrata, ma anche il compilatore ed i metodi per estrarre il numero dinamico delle istruzioni assembly dell'applicazione da stimare. I metodi non lineari potrebbero prevedere più accuratamente le prestazioni del software in casi specifici, ma l'esplorazione dello spazio di progetto diventa più complessa. Tipicamente tali metodi non sono additivi. Inoltre l'uso delle reti neurali rende difficile la stima della varianza di esecuzione.

Una limitazione comune a tutti i modelli analitici presentati finora è che non vengono prese in considerazione le gerarchie di memoria e tutte le ottimizzazioni dinamiche del processore. Inoltre, i modelli potrebbero non funzionare con la stessa accuratezza su architetture differenti dai processori RISC (CISC o VLIW).

Per migliorare i modelli di stima sono stati proposti approcci ibridi che

utilizzano sia la simulazione che la stima.

Meyr et al.[25] uniscono i modelli del processore, generati da un linguaggio descrittore dell'architettura, che forniscono una stima dei tempi di esecuzione di ciascuna istruzione dell'Instruction Set dell'architettura, alle simulazioni basate su *SystemC*, che forniscono il numero esatto di cicli di esecuzione di ciascuna istruzione. Tale tecnica è più veloce di quelle basate su ISS *(Instruction Set Simulator)*, ma richiede più tempo rispetto ad eseguire direttamente il codice instrumentato su una macchina host.

HySim [15] combina l'utilizzo di un ISS con l'esecuzione del codice su una macchina host per raggiungere alte velocità di simulazione. L'esecuzione del codice su una macchina host utilizza un modello additivo che approssima il comportamento della *data-cache* per processori RISC. Tale approccio effettivamente aumenta la velocità di simulazione fino a 5x, ma è comunque ancora troppo lento per una vasta esplorazione dello spazio di progetto su architetture complesse. Non rimuove, inoltre, la necessità di avere nel simulatore un design abbastanza accurato dell'hardware target e richiede, anche, di effettuare una fase di training dei modelli di stima.

Stolberg et al.[33] suggeriscono di lavorare a più alto livello costruendo un modello di un'applicazione basato sui task, indipendente dalla piattaforma hardware e dalla specifica implementazione del software. Tale metodologia, però, è applicabile solo ad applicazioni di tipo intensivo, perché il tempo di esecuzione del task non dipende da costrutti di controllo ed inoltre non è possibile modellizzare le interazioni tra task.

Capitolo 5

Approccio Metodologico

In questo capitolo verrà illustrata la metodologia utilizzata durante l'intero lavoro di tesi.

Nella prima sezione verrà esposta la metodologia di stima sviluppata come tentativo di superare i limiti della metodologia proposta da [9] analizzata nel Capitolo 4.

5.1 La stima di prestazioni con intervalli di confidenza proposta

La metodologia di stima di prestazioni con intervalli di confidenza proposta prende avvio dai due lavori analizzati nelle Sezioni 4.2.3 e 4.2.1 e realizza i seguenti obiettivi:

- sfrutta il lavoro precedente descritto nella sezione 4.2.3 di stima del tempo medio di esecuzione ma utilizza la tecnica di regressione proposta al fine di effettuare una stima anche della varianza del tempo di esecuzione e quindi un intervallo di confidenza;

- mantiene il carattere generale del lavoro di Jantsch e Bjuréus per quanto concerne la distribuzione di probabilità con cui sono modellizzabili le sin-

gole istruzioni. Infatti consente di modellizzare le operazioni con diverse distribuzioni di probablità;

- sfrutta la metodologia di stima degli intervalli di confidenza proposta da Jantsch e Bjuréus, mantenendone la semplicità di calcolo.

Nei paragrafi successivi verrà analizzato come tali obiettivi vengono raggiunti.

5.1.1 Estensione del metodo di stima e della tecnica di regressione

Il modello di stima basato sulle operazioni RTL sviluppato nel precedente lavoro sfrutta una versione modificata del compilatore GCC per effettuare una stima del tempo medio di esecuzione dell'applicazione tramite l'uso di annotazioni temporali di ciascuna operazione RTL, della simulazione e della regressione lineare. Il lavoro inizialmente proposto però prevedeva di utilizzare solamente i tempi medi di esecuzione, ottenuti tramite regressione lineare, di ciascuna operazione RTL e di annotare le operazioni RTL, frutto della traduzione del codice sorgente da parte del compilatore, con tali tempi. Da un punto di vista probabilistico, il modello precedente associava alla singola operazione RTL una distribuzione di probabilità *costante*, a varianza quindi nulla, in cui era pari a 1 la probabilità che l'istruzione RTL impiegasse il tempo specificato. Tradotto in formule risulta:

$$P(t_{RTL} = t_{Regressione}) = 1 \tag{5.1}$$

dove con t_{RTL} si è indicato il tempo in cicli necessari all'esecuzione della singola istruzione RTL e con $t_{Regressione}$ il tempo di esecuzione stimato per quell'istruzione tramite la regressione lineare.

Il modello di stima

Il modello di stima proposto nel presente lavoro è una modifica del modello progettato da *Ferrandi e Lattauda* [21]. Alle singole operazioni RTL generate

dal compilatore è stato associato non solo un tempo medio di esecuzione ma anche una varianza di esecuzione tramite l'uso di particolari distribuzioni di probabilità con media e varianza non nulla. Grazie alla possibilità di associare una distribuzione di probabilità qualsiasi alle operazioni RTL, viene mantenuto il carattere generale del lavoro di Jantsch e Bjuréus e si rende possibile la stima anche nel caso di cui le operazioni RTL non siano in numero sufficiente a garantire la gaussianità asintotica.

Il flusso di esecuzione della metodologia di stima già elaborato da *Ferrandi e Lattuada* [21] è però rimasto invariato. Infatti, in virtù del *Teorema Centrale del Limite* [Teorema 16], secondo il quale la distribuzione di partenza delle singole istruzioni RTL non conta poiché è garantita l'asintotica gaussianità, e in virtù del *Teorema di Additività* [Teorema 1], non solo il tempo medio di esecuzione, ma anche la varianza del tempo medio esecuzione gode della proprietà di additività, quindi la varianza totale del tempo di esecuzione è data dalla somma delle varianze di ciascuna istruzione RTL ed è distribuita come una variabile aleatoria *Normale*.

Il metodo di regressione

Analogamente al modello di stima, anche il metodo di regressione utilizzato nel presente lavoro di tesi è un estensione di quello proposto da *Ferrandi e Lattuada* [21] modificato, come descritto nel Capitolo 2, al fine di effettuare una stima non solo delle medie ma anche delle varianze da associare alle singole operazioni RTL

Se precedentemente la regressione lineare consentiva solo di stimare i tempi medi di esecuzione a partire dai tempi di esecuzione di un insieme di numerosi benchmark, con la tecnica di regressione proposta è stato possibile stimare anche le varianze associate alle singole operazioni RTL. Il modello di regressione utilizzato è stato il seguente:

$$Y_i = \beta_1 X_{i1} + \beta_2 X_{i2} + \cdots + \beta_N X_{iN} + \varepsilon_i = \sum_{j=1}^{K} \beta_j X_{ij} + \varepsilon_i \qquad i = 1:K \tag{5.2}$$

dove:

- K è il numero di benchmark utilizzati per la stima $i = 1...K$
- N è il numero di tipi di operazioni appartenenti all'Instruction Set, da stimare $j = 1...N$
- Y_i rappresenta il numero di cicli di esecuzione totali dell'i-esimo benchmark $i = 1...K$
- X_{ij} rappresenta il numero di volte in cui l'operazione di tipo j-esimo viene eseguita nel benchmark i-esimo
- β_j rappresenta il numero medio di cicli di esecuzione per ogni operazione, la stima dei quali è obiettivo della regressione
- ε_i sono i *residui* di stima o *errori statistici* associati alle K variabili dipendenti, ossia la differenza tra il numero totale di cicli stimato e il numero di cicli reale, ottenuto tramite simulazione, per il benchmark i-esimo;

In forma matriciale diviene:

$$\mathbf{Y} = \mathbf{X}\mathbf{b} + \boldsymbol{\varepsilon} \tag{5.3}$$

La stima dei regressori β_j, ossia del numero medio di cicli di esecuzione per ogni operazione, viene effettuata attraverso la relazione:

$$\hat{\mathbf{b}}_{OLS} = \begin{pmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_N \end{pmatrix} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{5.4}$$

che è il vettore di stima ai minimi quadrati (OLS: Ordinary Least Squares) del numero medio di cicli di esecuzione per ogni operazione. La varianza associata alle singole operazioni si stima con la seguente relazione:

$$var(\widehat{b}_{OLS}) = \begin{pmatrix} var(\hat{\beta}_1) \\ var(\hat{\beta}_2) \\ \vdots \\ var(\hat{\beta}_N) \end{pmatrix} = diag\left(\boldsymbol{\sigma}^2(\mathbf{X}^T\mathbf{X})^{-1}\right) = diag\left(\frac{\sum_{i=1}^{K}[\varepsilon_i]^2}{K-N-1}(\mathbf{X}^T\mathbf{X})^{-1}\right) \tag{5.5}$$

Normalizzazione del modello

Uno dei limiti delle tecniche di regressione lineare consiste nel criterio utilizzato costruire il modello. Nella regressione lineare classica si costruisce il modello minimizzando l'errore dei minimi quadrati (*MSE*):

$$MSE = \frac{1}{n}\sum_{i=1}^{k}\varepsilon_i^2 \tag{5.6}$$

dove i ε_i sono i *residui* di stima o *errori statistici*, già discussi nel Capitolo 2, espressi come differenza tra il numero di cicli di esecuzione reale di ogni benchmark e quello stimato, cioè la differenza tra i dati reali e quelli predetti.

$$\varepsilon_i = Y_i - \overline{Y_i} \quad i = 1...K \tag{5.7}$$

Questo criterio di minimizzazione può creare problemi quando si cerca di costruire un modello su un insieme non omogeneo di dati. Per esempio, considerando le due osservazioni $Y_1 = 100$ e $Y_2 = 10000$ ed i valori stimati $\overline{Y_1} = 200$ e $\overline{Y_2} = 10100$ per entrambi avremo lo stesso errore ai minimi quadrati (100^2). In realtà però nella predizione della prima osservazione si commette un errore del $100\% = \left(\frac{|200-100|}{100}100\right)$ mentre nella seconda l'errore dello stimatore è solo del $1\% = \left(\frac{|10100-10000|}{10000}100\right)$.

Con il criterio della minimizzazione dei minimi quadrati non si prende in considerazione l'errore relativo. Ignorare tale errore è particolarmente critico

nella stima di prestazioni, perché applicazioni diverse possono avere tempi di esecuzione differenti di alcuni ordini di grandezza. Inoltre, un modello costruito solo minimizzando l'errore dei minimi quadrati (MSE) si adatta meglio ad applicazioni di lunga durata (nell'ordine di qualche milione di cicli) e non descrive correttamente quelle brevi. Per ridurre la varianza delle variabili dipendenti e cercare di rendere l'insieme dei dati il più omogeneo possibile, si è deciso di adoperare la seguente tecnica di normalizzazione. Invece di costruire un modello per stimare il tempo totale di esecuzione di un'applicazione, si utilizza la regressione lineare per valutare il tempo medio di esecuzione di ogni singola operazione. Tale tempo medio può essere ottenuto dividendo il numero di volte in cui una data operazione viene eseguita e il tempo di esecuzione reale dell'applicazione per il numero totale di operazioni eseguite all'interno del benchmark. Dividendo l'Equazione 5.2, presentata nel Paragrafo 5.1.1:

$$Y_i = \sum_{j=1}^{K} \beta_j X_{ij} + \varepsilon_i \qquad i = 1 : K \tag{5.8}$$

per il numero totale T di operazioni eseguite all'interno del benchmark i-esimo, ossia:

$$T = \sum_{j=1}^{N} Xij \quad \forall i = 1...K \tag{5.9}$$

si ottiene l'Equazione 5.10:

$$\frac{Y_i}{T} = \sum_{j=1}^{N} \left(\frac{\beta_j X_{ij}}{T} \right) + \frac{\varepsilon_i}{T} \qquad \forall i = 1 : K \quad j = 1...N \tag{5.10}$$

che scritta in maniera compatta utilizzando una notazione analoga all'Equazione 5.8 diventa:

$$W_i = \sum_{j=1}^{N} \beta_j Z_{ij} + r_i \qquad \forall i = 1 : K \tag{5.11}$$

dove:

$$W_i = \frac{Y_i}{T} \quad \forall i = 1 : K$$

rappresenta il numero medio di cicli per operazione all'interno del benchmark i-esimo,

$$Z_{ij} = \frac{X_{ij}}{T} \quad \forall i = 1 : K$$

rappresenta la frazione di volte in cui l'operazione di tipo j-esimo viene eseguita rispetto al totale di operazioni eseguite nel benchmark i-esimo,

$$r_i = \frac{\varepsilon_i}{T} \quad \forall i = 1 : K \tag{5.12}$$

rappresenta il residuo medio per ogni istruzione eseguita nel benchmark i-esimo. I coefficienti Z_{ij} così trovati diventano i nuovi pesi associati alle operazioni.

Introduzione dell'operazione main

Solitamente, sia con l'uso di simulatori che con l'esecuzione diretta del codice sull'hardware, quando per un dato processore si ottengono i tempi di esecuzione delle applicazioni, nella misura è compreso anche il tempo di start-up del programma. Per tempo di start-up si intende il tempo che intercorre tra l'avvio di un programma e l'inizio vero e proprio dell'esecuzione del suo codice. Si può ipotizzare che questo overhead sia indipendente dall'applicazione, ma comunque il suo impatto relativo è molto più alto sui programmi brevi che su quelli lunghi. Per aggirare il problema è stata introdotta una operazione fittizia, chiamata *main*. La nuova operazione è trattata come tutte le altre, quindi viene stimata in regressione al pari delle altre, con la differenza che presenta coefficiente unitario, ossia viene eseguita una sola volta in ciascun benchmark. Il suo valore, dopo la normalizzazione, sarà più grande nelle applicazioni brevi e più piccolo in quelle lunghe. Questo comportamento permette di caratterizzare l'impatto dello start-up, a seconda della dimensione del programma.

Algoritmi di rimozione di benchmark

La tecnica di regressione presentata nella Sezione 5.1.1 è fortemente dipendente dai benchmark sui quali viene applicata. In particolare, la tecnica di regressione lineare soffre di due problematiche:

- *Overfitting*
- *Polarizzazione*

L'overfitting è dovuto a particolari uniformità di certi insiemi di benchmark e impatta negativamente sulla significatività dei regressori: alle operazioni contenute nei benchmark ripetuti viene dato maggior peso in fase di regressione e ciò porta ad un abbassamento della significatività dei regressori non contenuti nei benchmark ripetuti, diminuzione a volte anche consistente.

La polarizzazione è dovuta all'eccessiva diversità di alcuni benchmark. Nonostante la normalizzazione effettuata sui benchmark secondo la metodologia mostrata nella Sezione 5.1.1, a causa dell'eterogeneità dell'insieme di benchmark può accadere che alcuni benchmark presentino un numero totale di cicli piuttosto basso, nell'ordine del migliaio di cicli, mentre altri benchmark più complessi presentassero un numero di cicli molto elevato dell'ordine dei milioni di cicli. Tale eccessiva dispersione causa di aumenti dell'errore relativo di stima del modello di regressione anche consistenti.

Si è reso quindi necessario sviluppare alcune tecniche algoritmiche di rimozione dei benchmark che mirassero a ridurre le problematiche analizzate finora. Ne sono state sviluppate due ossia:

1. Rimozione dei benchmark mal normalizzati
2. Rimozione con algoritmo MinMax

Queste due tecniche di rimozione vengono applicate alla matrice X e al vettore Y seguendo due ordini diversi e generando due matrici di modello differenti:

1. Matrice X_c generata applicando l'algoritmo di rimozione dei benchmark mal normalizzati alla matrice X;
2. Matrice X_{mc} generata applicando l'algoritmo MinMax e quindi quello di rimozione dei benchmark mal normalizzati alla matrice X;

Applicando la tecnica di regressione lineare a ciascuna della due matrici sopra elencate vengono generati due diversi modelli di risultato che nel seguito verranno indicati con $\widehat{b}_c$, $\widehat{b}_{mc}$.

Le due tecniche consentono di ridurre il *training set*, ossia l'insieme di benchmark sui quali verrà applicata la regressione lineare, migliorandone i risultati senza però far perdere di generalità al modello risultato. I benchmark rimossi con queste tecniche non vengono però scartati, bensì costituiscono il *validation set* di ciascun dei due modelli di risultato ossia l'insieme di benchmark attraverso i quali viene verificato che il modello generato a partire dal training set abbia validità generale, ossia generi intervalli di confidenza per il tempo di esecuzione di un applicazione che contengano con elevata probabilità il valore vero del tempo di esecuzione dell'applicazione in esame.

Verrà ora mostrato come le tecniche di rimozione finora discusse vengono implementate.

Rimozione MinMax

Lo schema dell'Algoritmo di rimozione *MinMax* viene presentato in 5.1.1:

Algoritmo 5.1.1: MINMAX(X, Y)

$$
\textbf{while } Y_size_after < Y_size_before \\
\textbf{do} \begin{cases} Y_size_before = size(Y) \\ max \leftarrow vect_max(Y) \\ min \leftarrow vect_min(Y) \\ \textbf{if } max/min > 100 \\ \quad \textbf{then} \begin{cases} m = avg(Y) \\ \textbf{if } max/m > 10 \\ \quad \textbf{then} \begin{cases} Y \leftarrow remove_line(Y, row_index(max)) \\ X \leftarrow remove_line(X, row_index(max)) \end{cases} \\ \textbf{if } m/min > 10 \\ \quad \textbf{then} \begin{cases} Y \leftarrow remove_line(Y, row_index(min)) \\ X \leftarrow remove_line(X, row_index(min)) \end{cases} \end{cases} \\ Y_size_after = size(Y) \end{cases}
$$

L'algoritmo MinMax è stato elaborato allo scopo di ridurre insiemi di benchmark troppo dispersi, ossia che presentassero benchmark con valore massimo

del tempo di esecuzione *max* distante più di due ordini di grandezza rispetto al valore minimo *min* del tempo di esecuzione. Se l'insieme è eccessivamente disperso, l'algoritmo calcola il numero di cicli medio m dell'insieme di benchmark tramite la funzione *avg(Y)* applicata al vettore Y che contiene i tempi di esecuzione totali di ciascun benchmark. Se il rapporto tra il numero di cicli *max* e il numero di cicli medio m è maggiore di 10, tramite la funzione *remove_line(Y, row_index(max))* viene rimossa la riga contenente il valore *max*, il cui indice viene estratto attraverso la funzione *row_index(max)*, dal vettore Y e tramite la funzione *remove_line(X, row_index(max))* viene rimossa la corrispondente riga della matrice X per mantenere dimensioni congruenti tra la matrice X e il vettore Y. Se il rapporto tra il numero di cicli medio m e il numero di cicli *min* è maggiore di 10, si effettua l'eliminazione della riga di indice contenente il valore *min* sia dal vettore Y che dalla matrice X. L'algoritmo continua a effettuare eliminazioni fino a che la condizione di non dispersione non è soddisfatta, ossia fino a che non vengono più effettuate rimozioni modificando la dimensione del vettore Y. Per verificare che il vettore Y non venga modificato si controlla la dimensione all'inizio e alla fine di ogni iterazione dell'algoritmo tramite la funzione *size(Y)*. Se la dimensione rimane invariata, ossia max e min distano meno di due ordini di grandezza, l'algoritmo termina.

Test applicati al modello

I modelli di stima generati applicando la regressione lineare alle matrici risultanti degli algoritmi di rimozione benchmark vengono sottoposti ai test statistici discussi nella Sezione 2.3.5 al fine di verificare la bontà dei modelli generati e la loro applicabilità alla stima di prestazioni.

Applicazione del Modello

Al termine del processo di generazione dei modelli di stima è possibile utilizzarli per stimare le prestazioni delle applicazioni e per generare degli intervalli di confidenza per tale stima. Grazie ai dati relativi al tempo di esecuzione stimato per ogni singola operazione (RTL o GIMPLE) e alla relativa varianza, viene effettuata una stima del tempo totale di esecuzione di un applicazione in ingres-

so tramite la somma della semplice moltiplicazione del tempo di esecuzione di ogni istruzione per il numero di volte in cui essa compare nel benchmark.

Infatti, essendo noti al termine della regressione le stime $\hat{\beta}_j$ dei coefficienti β_j associati alla j-esima operazione e i singoli X_{ij}, è sufficiente applicare l'Equazione 5.13 per ottenere il numero totale $\hat{Y}_i$ di cicli stimato per il benchmark i-esimo, ossia:

$$\hat{Y}_i = \sum_{j=1}^{N} \hat{\beta}_j X_{ij} \tag{5.13}$$

Anche la varianza del tempo totale di esecuzione può essere espressa in maniera analoga all'Equazione 5.13 tramite l'equazione 5.14:

$$var(\hat{Y}_i) = \sum_{j=1}^{N} var(\hat{\beta}_j) X_{ij} \tag{5.14}$$

dove con $var(Y_i)$ si è indicata la varianza associata al tempo di esecuzione totale per il benchmark i-esimo, con $var(\hat{\beta}_j)$ si è indicata la varianza del tempo medio di esecuzione del j-esimo tipo di istruzione e con x_{ij} la frequenza, ossia il numero di volte in cui l'istruzione di tipo j-esimo viene eseguita nel benchmark i-esimo.

Tramite le Equazioni 5.13 e 5.14 è possibile ottenere degli intervalli di confidenza per il tempo totale di esecuzione sfruttando le Equazioni 2.18, 2.19 e 2.20 contenute nel Capitolo 2 ottenendo le espressioni 5.15, 5.16 e 5.17:

$$99.7\% \Longrightarrow (\ Y_i - 3var(Y_i),\ Y_i + 3var(Y_i)\ ) \tag{5.15}$$

$$95.5\% \Longrightarrow (\ Y_i - 2var(Y_i),\ Y_i + 2var(Y_i)\ ) \tag{5.16}$$

$$68.7\% \Longrightarrow (\ Y_i - \ \ var(Y_i),\ Y_i + \ \ var(Y_i)\ ) \tag{5.17}$$

Nel presente lavoro di tesi di è scelto di adottare l'espressione 5.15 poiché consente di ottenere l'intervallo di confidenza di maggior precisione.

Capitolo 6

L'Implementazione all'interno del Progetto PandA

La metodologia proposta nel Capitolo 5 è stata implementata all'interno di *PandA* [26], framework sviluppato in linguaggio C++ utilizzato per la ricerca e la sperimentazione nel campo dell'HW-SW Co-Design e basato sul compilatore GNU GCC. Il framework, realizzato dal Laboratorio di Microarchitetture del Dipartimento di Elettronica e Informazione presso il Politecnico di Milano, integra numerosi tool in grado di fornire supporto per:

- sintesi ad alto livello di sistemi hardware;
- estrazione del parallelismo per il software e partizionamento dell'hardware/software;
- definizione di metriche per l'analisi ed il mapping su architetture multiprocessore;
- sintesi logica di circuiti digitali.

L'implementazione del presente lavoro di tesi è stata integrata all'interno di due strumenti del framework PandA: *Zebu* e *Spider*.

Nel presente Capitolo verrà presentata l'implementazione della metodologia discussa nel Capitolo 5.

Nelle Sezioni 7.1, 6.1 e 6.2 verrà descritto il funzionamento di ciascuno strumento interessato e le modifiche a esso apportate al fine di implementare la metodologia proposta.

6.1 Zebu

Zebu è un compilatore *C-to-C* che realizza il profiling, il partizionamento ed il mapping di specifiche C. La struttura di Zebu è simile a quella del compilatore GCC presentata nella Sezione 3.1 e può essere divisa in tre componenti principali:

- *Front-End*: analizza l'applicazione tramite l'uso di una versione modificata del compilatore GCC elaborando le rappresentazioni intermedie generate;
- *Middle-End*: effettua il partizionamento e il mapping del codice sorgente dell'applicazione analizzata;
- *Back-End*: riscrive il codice sorgente C di partenza nella forma corrispondente alla soluzione di mapping e partizionamento desiderata.

6.1.1 Il Front-End di Zebu

Il *front-end* di Zebu esegue una versione modificata del compilatore GCC (detta *TreeGCC*, modificata al fine di produrre il maggior numero possibile di informazioni circa le rappresentazioni intermedie utilizzate) passandovi in ingresso il codice sorgente dell'applicazione in esame. In questa fase vengono generate le rappresentazioni GIMPLE ed RTL dell'applicazione.

La rappresentazione GIMPLE, che viene estratta alla fine della fase di middle-end in modo da considerare tutte le ottimizzazioni eseguite in questa seconda fase, è in grado di rendere l'analisi effettuata da Zebu più semplice e più accurata, migliorando i risultati della fase successiva di partitioning.

La rappresentazione RTL, estratta invece all'inizio della fase di back-end per mantenere la correlazione con la rappresentazione GIMPLE, dipende dalla particolare architettura obiettivo che viene considerata durante la compilazione ad

opera di GCC. Zebu non è in grado ci considerare ogni architettura, al momento Zebu opera su architetture di tipo SPARC e ARM.

6.1.2 Il Middle-End di Zebu

Durante la fase di *middle-end* Zebu effettua il partizionamento e il mapping dell'applicazione. Il partizionamento viene effettuato tramite il raggruppamento o *clustering* delle espressioni GIMPLE che compongono la rappresentazione intermedia dell'applicazione.

In questa fase possono essere effettuati due tipi di partizionamento:

- Parallelo: il partizionamento parallelo è ottenuto suddividendo l'applicazione in *task* paralleli al fine di velocizzare l'esecuzione dell'applicazione sfruttando più processori.
- Sequenziale: il partizionamento sequenziale è ottenuto suddividendo porzioni dell'applicazione in catene di task in sequenza al fine di velocizzare l'esecuzione dell'applicazione eseguendo tali task tramite l'uso di hardware dedicato come *Digital Signal Processors* (DSP) o *Field Programmable Gate Array* (FPGA).

6.1.3 Modifiche effettuate a Zebu

La versione presente nel framework PandA prima del presente lavoro di tesi associava ad ogni istruzione della rappresentazione intermedia GIMPLE o RTL un tempo, in termini di cicli di esecuzione, estratto da un file di modello in formato XML [35], che rappresenta il numero di cicli richiesto da quella istruzione per essere eseguito sulla data architettura obiettivo. Tali file erano generati tramite l'uso della regressione lineare e contenevano solamente il numero di cicli medio stimato per ogni operazione RTL o GIMPLE. Ai fini della presente tesi era però necessario associare non solo un tempo medio alle singole istruzioni ma anche una varianza del tempo di esecuzione.

La libreria *probability_distribution*: per consentire a Zebu di effettuare una stima anche in termini di varianza del tempo di esecuzione, è stata sviluppata

e inserita al suo interno la libreria C++ *probability_distribution*. Grazie ad essa Zebu è in grado di associare a ogni istruzione RTL o GIMPLE una variabile stocastica, rappresentata dalla classe astratta chiamata *abstract_distribution*.

Tale classe astratta rappresenta una generica distribuzione di probabilità che viene concretizzata attraverso alcune classi che rappresentano distribuzioni di probabilità vere e proprie. Contiene la definizione delle operazioni matematiche applicabili alle distribuzioni di probabilità ossia somma, sottrazione, moltiplicazione e divisone per una costante numerica. In questo modo il tipo di dato elaborato da Zebu risulta trasparente al flusso di esecuzione, che rimane immutato.

Le distribuzioni che sono state sviluppate sono:

- **Distribuzione Costante**: la distribuzione costante $cost(\mu)$ di valore μ ha densità di probabilità concentrata in un singolo punto, ossia ha probabilità pari a 1 di assumere il valore μ e probabilità pari a 0 di assumere un qualunque valore $\neq \mu$. In Figura 6.1 viene mostrata la densità di probabilità di una distribuzione costante di valore μ.

- **Distribuzione Normale**: la distribuzione normale $N(\mu, \sigma^2)$ ha densità di probabilità della forma vista nell'Equazione 2.5.

Data la semplicità dell'interfaccia abstract_distribution sarà facile implementare anche altre distribuzioni nel caso in cui vengano richieste da futuri sviluppi del presente lavoro di tesi.

Modifica dell'output: l'output di Zebu è stato modificato in modo da tener traccia delle distribuzioni di probabilità: alla fine della fase di stima viene prodotto in uscita un file XML contenente le operazioni RTL o GIMPLE che componevano le rispettive rappresentazioni intermedie, assieme al numero di volte in cui tali operazioni vengono eseguite e al numero totale di cicli di esecuzione ottenuto grazie al simulatore *tsim*.

Nella figura 6.2 viene mostrato un esempio di file XML prodotto da Zebu contenente il risultato della simulazione effettuata da tsim e il numero di volte in cui vengono eseguite alcune operazioni RTL clusterizzate.

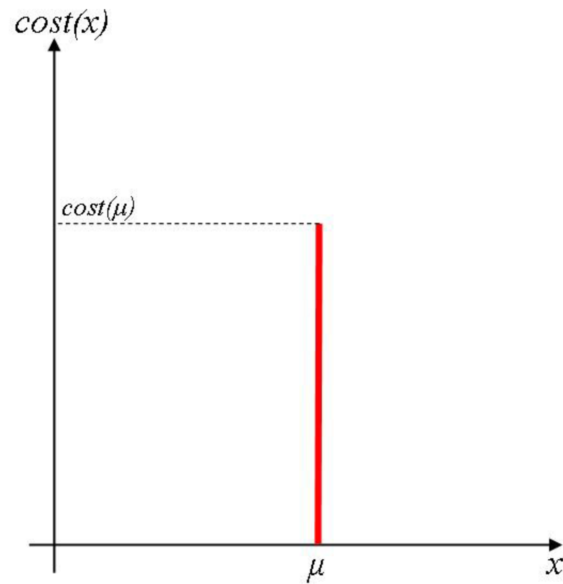

Figura 6.1. Distribuzione di probabilità Costante

```
<?xml version="1.0"?>
<sym_distribution name="20000112-1.c::__test_1">
  <sequential simulated_cycles="2168">
    <op name="assignment" count="28"/>
    <op name="branch_expr" count="10"/>
    <op name="call_expr" count="5"/>
    <op name="const_readings" count="18"/>
    <op name="integer_memory_readings" count="5"/>
    <op name="integer_register_accesses" count="60"/>
    <op name="integer_register_writings" count="34"/>
    <op name="is_main" count="1"/>
  </sequential>
</sym_distribution>
```

Figura 6.2. Esempio di file XML prodotto da Zebu

Tramite l'utilizzo di script bash, Zebu viene invocato sull'intero insieme di benchmark del *training set* producendo un file XML per ciascun benchmark, file che saranno utilizzati poi da Spider.

Modifica dei modelli di peso con tsim: nella sua versione inziale, Zebu utilizzava dei *modelli di peso* in formato XML contenenti il numero di cicli di esecuzione stimato per ogni operazione RTL e GIMPLE. Grazie al presente lavoro di tesi, Zebu è stato modificato in modo tale da interpretare i file XML prodotti da Spider contenenti non solo il tempo medio di esecuzione ma anche la varianza di ogni istruzione RTL e GIMPLE. Utilizzando questi nuovi file prodotti dalla versione modificata di Spider, Zebu è ora in grado di stimare anche la varianza del tempo di esecuzione di un dato codice sorgente C e generare intervalli di confidenza per il tempo di esecuzione sfruttando la metodologia proposta nel Capitolo 5 e la relativa implementazione discussa nel presente Capitolo.

6.2 Spider

Spider è lo strumento del framework PandA preposto all'elaborazione dei dati. Questi è in grado di generare i file di modello raccogliendo i dati prodotti durante le analisi effettuate da Zebu su ciascun benchmark del training set. In base al tipo risultati prodotti da Zebu, Spider automaticamente effettua determinate operazioni di pre-processing.

Modifiche a Spider

Per realizzare la metodologia proposta, Spider è stato modificato al fine di interpretare i risultati prodotti dalla nuova versione di Zebu. In particolare, è stata sviluppata la libreria *variance_estimation* dedicata alla stima della varianza dei tempi di esecuzione, contenente la classe *regression_estimation*.

All'interno della nuova classe è stata implementata la metodologia di regressione mostrata in 5.1.1 basandosi sull'utilizzo del software *R* [17]. R è un software open source di analisi statistica in grado di effettuare velocemente regressione lineare in media e varianza e di applicare inoltre numerosi test al mo-

dello risultato. Al fine di interfacciare Spider a R è stata richiesto l'utilizzo delle librerie *RInside* e *Rcpp* in modo da inviare comandi e prelevare risultati a e da R. Nei paragrafi seguenti verrà presentato il flusso di esecuzione della classe *regression_estimation*.

Costruzione della matrice X di modello: Spider è stato modificato in modo da poter processare i nuovi file XML prodotti da Zebu. Dopo aver ricevuto in ingresso tutti i file XML relativi ai risultati dei benchmark del training set, Spider li analizza e invoca la classe *regression_estimation* che costruisce il modello di regressione, ossia la matrice X e il vettore Y della Sezione 2.34 riga per riga. Infatti, ciascun file XML prodotto da Zebu contiene i coefficenti x di una riga della matrice $\mathbf{X}$ all'interno dell'attributo *count* di ogni nodo *op* e contiene il corrispettivo coefficiente y del vettore $\mathbf{Y}$ all'interno dell'attributo *simulated_cycles* del nodo *sequential*.

Applicazione degli Algoritmi di Rimozione Benchmark: a partire dalla matrice $\mathbf{X}$, vengono applicati gli algoritmi di rimozione benchmark nell'ordine discusso nella Sezione 5.1.1 e generati un file .txt e un file .CSV (*Comma Separated Values*) contenente la matrice di modello risultante. In particolare, vengono generati due file .txt, un primo file contenente il modello $\mathbf{X_c}$ risultante dall'applicazione dell'algoritmo di rimozione dei benchmark mal normalizzati e un secondo contenente il modello $\mathbf{X_{mc}}$ risultante dall'applicazione dell'algoritmo *MinMax* seguito da quello di rimozione dei benchmark mal normalizzati alla matrice X.

I benchmark rimossi tramite tali algoritmi vengono salvati in due matrici, contenenti le righe rimosse dalle rispettive matrici X, dette *matrici di validazione* dette M_c e M_{mc}, e in due vettori, contenenti gli elementi rimossi dai rispettivi vettori Y, detti V_c e V_{mc} sfruttando R.

Applicazione del metodo di Regressione Lineare: tramite l'uso di Rcpp e RInside, che consentono di inviare comandi ad R direttamente dal codice C++, viene eseguito l'algoritmo di regressione lineare tramite la funzione `lm`. Il comando `lm` viene eseguito sulle due matrici X_c e X_{mc} . Il risultato dell'appli-

```
data <- read.table("coeffsumnorm_regression_model.cvs");
dataDF <- data.frame(data)
model <- lm(Y~.-1, data=dataDF)
summary(model)
```

Figura 6.3. Listato dei comandi eseguiti da R per effettuare regressione

cazione della funzione alle due matrici sono due vettori dei coefficenti stimati detti *modelli di stima*, $\widehat{\mathbf{b}}_{\mathbf{c}}$ e $\widehat{\mathbf{b}}_{\mathbf{mc}}$ i quali contengono i coefficienti di regressione stimati con il metodo dei minimi quadrati con associate le relative significatività. I comandi eseguiti per effettuare la regressione lineare sono elencati in Figura 6.3.

In Figura 6.4 viene mostrato il vettore dei regressori $\widehat{\mathbf{b}}_{\mathbf{c}}$ risultante dai comandi elencati in Figura 6.3 applicati alla matrice di modello X_c.

Calcolo delle varianze associate ai regressori: utilizzando la funzione `vcov` di R applicata ai vettori $\widehat{\mathbf{b}}_{\mathbf{c}}$ e $\widehat{\mathbf{b}}_{\mathbf{mc}}$ vengono estratti i vettori $var(\widehat{\mathbf{b}}_{\mathbf{c}})$ e $var(\widehat{\mathbf{b}}_{\mathbf{mc}})$ contenenti le varianze dei singoli coefficienti di regressione stimati.

Calcolo delle varianze del tempo di esecuzione: sfruttando le due matrici $\mathbf{X_c}$ e $\mathbf{X_{mc}}$, i due vettori $\widehat{\mathbf{b}}_{\mathbf{c}}$ e $\widehat{\mathbf{b}}_{\mathbf{mc}}$ e i due vettori $\mathbf{var}(\widehat{\mathbf{b}}_{\mathbf{c}})$ e $\mathbf{var}(\widehat{\mathbf{b}}_{\mathbf{mc}})$, Spider utilizza R per calcolare le varianze del tempo di esecuzione di ciascun benchmark sfruttando le operazioni tra matrici che R mette a disposizione e l'Equazione matriciale 2.53.

Validazione del modello: sfruttando i risultati presentanti nei paragrafi precedenti, Spider valida i due modelli generati attraverso l'uso di R. La validazione avviene tramite la generazione di intervalli di confidenza per i benchmark appartenenti agli insiemi di validazione contenuti nelle matrici M_c e M_{mc}. Seguendo la metodologia illustrata nella Sezione 5.1.1 vengono generati IC a tre livelli di significatività (99.7%, 95.5%, 68.7%) per ogni benchmark appartenente agli insiemi di validazione.

```
Call:
lm(formula = Y ~ . - 1, data = dataDF)

Residuals:
    Min      1Q  Median      3Q     Max
-4.4242 -0.9650 -0.3441  0.3489 27.1488

Coefficients:
                          Estimate Std. Error t value Pr(>|t|)
assignment                 74.6573    20.0422   3.725 0.000217 ***
bit_int_expr              -19.8363    10.7218  -1.850 0.064874 .
branch_expr                80.8428    23.6524   3.418 0.000681 ***
call_expr                 -25.5854    27.1344  -0.943 0.346167
comp_unknown_expr        -117.7300    42.3950  -2.777 0.005687 **
const_readings              6.2578     8.7678   0.714 0.475723
integer_memory_readings   -29.3085    13.8621  -2.114 0.034970 *
integer_memory_writings  -106.7693    25.4257  -4.199 3.16e-05 ***
integer_register_accesses  14.8544     7.5102   1.978 0.048473 *
integer_register_writings -89.7107    18.5571  -4.834 1.77e-06 ***
is_main                  2280.8953    24.5647  92.853  < 2e-16 ***
plusminus_int_expr         -0.7485    11.7766  -0.064 0.949345
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.683 on 515 degrees of freedom
Multiple R-squared: 0.9882, Adjusted R-squared: 0.9879
F-statistic:  3589 on 12 and 515 DF,  p-value: < 2.2e-16
```

Figura 6.4. Coefficenti risultanti dai comandi di Figura 6.3

Applicazione dei test: utilizzando il pacchetto *lmtest* di R, vengono infine applicati i test mostrati nella Sezione 5.1.1 sui due modelli di stima contenenti $\widehat{\mathbf{b}}_{\mathbf{c}}$ e $\widehat{\mathbf{b}}_{\mathbf{mc}}$.

Salvataggio dei risultati: al termine del procedimento di stima, Spider genera un file XML contenente il numero stimato di cicli di esecuzione per ogni istruzione con la relativa varianza e il corrispettivo valore di significatività del regressore, il risultato dei test statistici descritti nella Sezione 2.3.5 e i risultati della fase di validazione.

In Figura 6.5 viene mostrato un esempio di file XML generato da Spider.

```xml
<?xml version="1.0"?>
<regression_results>
  <coeff_results>
    <op class="assignment" mean="74.657" variance="401.689" signif_lev="***"/>
    <op class="bit_int_expr" mean="-19.836" variance="114.957" signif_lev="."/>
    <op class="branch_expr" mean="80.842" variance="559.434" signif_lev="***"/>
    <op class="call_expr" mean="-25.585" variance="736.276" signif_lev=" "/>
    <op class="comp_expr" mean="-117.730" variance="1797.334" signif_lev="**"/>
    <op class="const_readings" mean="6.257" variance="76.873" signif_lev=" "/>
    <op class="integer_memory_readings" mean="-29.308" variance="192.157" />
    <op class="integer_memory_writings" mean="-106.769" variance="646.467" />
    <op class="integer_register_accesses" mean="14.854" variance="56.403" />
    <op class="integer_register_writings" mean="-89.710" variance="344.364" />
    <op class="is_main" mean="2280.895" variance="603.424" signif_lev="***" />
    <op class="plusminus_int_expr" mean="-0.748" variance="138.687" />
  </coeff_results>
  <model_results>
    <residual_mean value="2.9282963548727938e-08"/>
    <rel_fitting_error value="0.13291239458910012"/>
    <rel_fitting_error value="0.13291239458910012"/>
    <validation value="0.94586"/>
  </model_results>
  <test_results>
    <t-test_result value="2.9282e-08" result="PASSED"/>
    <shapiro_test_result value="3.832" result="PASSED"/>
    <box_test_result value="0.538" result="PASSED"/>
  </test_results>
</regression_results>
```

Figura 6.5. Esempio file XML risultato generato da Spider

Capitolo 7

Risultati Sperimentali

Nel presente capitolo vengono presentati i risultati sperimentali ottenuti attraverso la metodologia descritta nel Capitolo 5 e attraverso la relativa implementazione descritta nel Capitolo 6. La macchina utilizzata per ottenere tali risultati è un portatile basato sul architettura Intel Merom Core 2 Duo con processore a 1.66 Ghz e 2 Gb di memoria RAM PC 5300.

Nella Sezione 7.1 verrà presentata l'architettura utilizzata per la simulazione e il simulatore utilizzato per il calcolo del tempo di esecuzione reale di ciascun benchmark su tale architettura.

Nella Sezione 7.2 verranno presentate le suite di benchmark utilizzati per ottenere i risultati di stima del presente capitolo e della relativa classificazione.

Nella Sezione 7.3 verranno presentati infine i risultati del presente lavoro di tesi.

7.1 Il processore Leon e tsim

Al fine di effettuare la regressione lineare sono necessari i tempi reali di esecuzione di ciascun benchmark del training set. Tali tempi di esecuzione possono essere ottenuti utilizzando un simulatore che esegua il codice sorgente simulandone l'esecuzione su una particolare architettura. Nel presente lavoro l'architettura target considerata è stata quella del processore *LEON3*. Tale processore è

dotato di un architettura a 32-bit a 8 core di esecuzione a 1,4 Mhz con pipeline a 7 stadi. Per simulare il LEON3 è stato utilizzato *tsim*. Tsim [3] è un un simulatore cycle-accurate capace di simulare processori RISC (Reduced instruction Set Computer) come ERC32 e LEON. Sfruttando tsim è possibile ottenere il numero reale di cicli di esecuzione di ciascun benchmark e poter così applicare la tecnica di regressione lineare.

7.2 Caratteristiche dei benchmark

Per validare la metodologia proposta è stato richiesto l'utilizzo di un vasto numero di benchmark al fine di non violare la condizione 2.3.3 definita nella Sezione 2.3.2, la quale afferma che il numero di benchmark utilizzato per effettuare la stima con il metodo di regressione lineare sia maggiore di un fattore 10 del numero di classi di operazioni da stimare. In particolare sono stati utilizzati 742 benchmark provenienti da differenti suite:

- *DSP Stone* [37]: una suite di benchmark dedicata alla valutazione e al confronto di diversi Digital Signal Processor;
- *MiBench* [18]: un insieme di benchmark che rappresenta diverse categorie commerciali: automazione e controllo industriale, dispositivi commerciali, networking, sicurezza e telecomunicazioni;
- *Splash 2* [36]: un insieme di applicazioni parallele per lo studio di architetture a memoria distribuita e a memoria centralizzata;
- *Powerstone* [24]: una suite di benchmark utilizzata per valutare le performance e il risparmio energetico di sistemi integrati e di applicazioni portatili come controllo di automobili, elaborazione di segnali, elaborazione di immagini e fax;
- *OpenMP Source Code Repository (OmpSCR)* [12]: una suite di benchmark per valutare le prestazioni di architetture multiprocessore a memoria condivisa;

- *NAS Parallel Benchmarks* [4]: una suite di benchmark progettata per valutare le prestazioni di architetture di supercalcolo.

Inoltre per estendere il più possibile l'insieme delle applicazioni del training set, oltre ai benchmark delle suite precedentemente descritte, si è utilizzato un insieme di applicazioni progettate per i test di regressione del GCC.

I benchmark sono stati divisi in tre categorie sulla base delle operazioni in essi contenuti. In particolare, sono stati divisi in base ai tipi di dato elaborati. Sono stati quindi classificati in:

- *integer* benchmark contenenti solo operazioni su tipi di dato interi;
- *float* benchmark contenenti prevalentemente operazioni su tipi di dato in virgola mobile;
- *all* benchmark contenenti operazioni su tipi di dato interi e in virgola mobile. L'insieme all contiene i due insiemi precedenti e altri benchmark che non appartengono ai due precedenti.

Sulla base di questa classificazione nelle Sezioni successive verranno presentati i risultati ottenuti.

7.3 Risultati

In questa sezione verranno presentati i modelli ottenuti attraverso l'applicazione della metodologia proposta in 5. I risultati saranno divisi in due categorie, una prima categoria contenente i risultati di stima applicati alle sole operazioni GIMPLE e una seconda categoria contenente i risultati di stima relativi alle operazioni RTL.

All'interno delle due categorie i risultati saranno ulteriormente divisi utilizzando la classificazione dei benchmark presentata nella Sezione 7.2 e secondo l'algoritmo di rimozione benchmark utilizzato.

Per ogni modello di stima saranno mostrate tre tabelle. Una prima tabella contenente media, varianza e significatività di ciascun regressore. Sarà utilizzata la seguente convenzione per definire la significatività dei regressori:

*** regressore molto significativo, risultato del t-test di significatività minore di 0.001 [30]

** regressore piuttosto significativo, risultato del t-test di significatività compreso tra 0.001 e 0.01 [30]

* regressore significativo, risultato del t-test di significatività compreso tra 0.01 e 0.05 [30]

. regressore poco significativo, risultato del t-test di significatività compreso tra 0.05 e 0.1 [30]

regressore non significativo, risultato del t-test di significatività compreso tra 0.1 e 1 [30]

Verrà quindi presentata una seconda tabella contenente le caratteristiche del modello ossia:

- Coefficiente R^2 di Determinazione del modello di regressione;
- Media dei residui;
- Errore medio relativo del modello, calcolato come $e = \sum_{i=1}^{k} \frac{\frac{Y_i - \widehat{Y}_i}{\widehat{Y}_i}}{k}$ dove con Y_i si intende il numero di cicli simulato del benchmark i-esimo, con $\widehat{y}_i$ si intende il numero di cicli stimato attraverso il modello del benchmark i-esimo;
- Numero di benchmark appartenenti al training set il cui tempo totale di esecuzione cade nell'IC di significatività 99.7%, 95.5%, 68.7% calcolato grazie al modello risultato;
- Numero di benchmark appartenenti al training set e al validation set (ossia l'insieme completo di benchmark della classe considerata) il cui tempo totale di esecuzione cade nell'IC di significatività 99.7%, 95.5%, 68.7% calcolato grazie al modello di stima risultante dalla metodologia.

Infine sarà presentata una terza tabella contenente i risultati dei test statistici discussi nella Sezione 2.3.5.

7.4 Risultati con operazioni GIMPLE

L'applicazione della metodologia proposta nel Capitolo 5 alle operazioni GIMPLE ha prodotto risultati che concordano con quanto emerge dal lavoro di *Ferrandi e Lattuada* [21]. Nel lavoro citato, i due autori giungono alla conclusione che la stima di prestazioni usando le operazioni GIMPLE presenta errori di stima di gran lunga superiori a quelli ottenuti utilizzando le operazioni RTL. Analoga conclusione è deducibile dai modelli che risultano dall'applicazione del metodo di regressione alle operazioni di classe GIMPLE. Nel seguito verranno mostrate tabelle che riassumono le caratteristiche dei modelli risultanti dall'applicazione della metodologia. In esse, un regressore sarà considerato significativo se raggiunge almeno un risultato nel t-test di significatività inferiore a 0.05 (almeno *).

Nella Tabella 7.1 vengono riassunte le caratteristiche e i risultati dei test sui modelli risultanti dall'applicazione dell'algoritmo di rimozione benchmark mal normalizzati alle operazioni di classe GIMPLE sui tre diversi insiemi di benchmark.

Caratteristica \ Benchmark	Integer	Float	All
Coefficente R^2	0.71638	0.84883	0.71537
Regressori Significativi	4/12 33.3%	3/12 25%	5/13 38.46%
Errore Relativo	34.19%	38.52%	37.69%
T-Test	✔	✗	✔
Shapiro-Wilk Test	✗	✗	✗
Box-Pierce Test	✔	✔	✗

Tabella 7.1: Caratteristiche dei modelli GIMPLE ottenuti con algoritmo di rimozione benchmark mal normalizzati

Nella Tabella 7.2 vengono riassunte le caratteristiche e i risultati dei test sui modelli risultanti dall'applicazione dell'algoritmo MinMax alle operazioni di classe GIMPLE sui tre diversi insiemi di benchmark.

Le Tabelle 7.1 e 7.2 mostrano quanto accennato in precendenza circa la bontà la stima con operazioni GIMPLE. Tutti i modelli presentano un coefficiente R^2 di determinazione piuttosto basso (minore di 0.85), sintomo della scarsa bontà

Caratteristica \ Benchmark	Integer	Float	All
Coefficente R^2	0.73758	0.84883	0.71537
Regressori Significativi	4/12 33.3%	3/12 25%	5/13 38.46%
Errore Relativo	33.22%	36.63%	35.46%
T-Test	✔	✗	✔
Shapiro-Wilk Test	✗	✗	✗
Box-Pierce Test	✔	✔	✗

Tabella 7.2: Caratteristiche dei modelli GIMPLE ottenuti con algoritmo MinMax

di predizione del numero medio totale di cicli da parte dei regressori. Il numero di regressori significativi è piuttosto basso (in media solamente il 30%), segno della bassa significatività complessiva del modello. L'errore di fitting relativo è piuttosto elevato, arriva a errori del 39%, rendendo i modelli difficilmente utilizzabili per la stima di prestazioni. Inoltre dai risultati dei test statistici emerge che i modelli sono sospetti di non essere OLS lineari: infatti tutti i modelli falliscono almeno il test di normalità dei residui di Shapiro-Wilk, segno che complessivamente i modelli non possono essere considerati lineari.

In conclusione, i modelli di stima generati a partire dalle operazioni di classe GIMPLE possono difficilmente essere utilizzati sia per effettuare una stima delle prestazioni di un applicazioni sia per costruire intervalli di confidenza significativi per il tempo totale di esecuzione di un applicazione.

7.5 Risultati con operazioni RTL

In questa sezione vengono presentati i risultati di stima ottenuti applicando la metodologia proposta nel Capitolo 5 alle operazioni RTL. I risultati sono classificati in base al tipo di dato e in base all'algoritmo di rimozione benchmark utilizzato.

7.5.1 Risultati con benchmark interi

Vengono ora presentati i modelli ottenuti con benchmark di soli interi operanti su operazioni RTL. La tabella 7.3 riassume i dati relativi al numero iniziale di benchmark e al numero di benchmark rimossi grazie agli algoritmi di rimozione benchmark.

Benchmark iniziali	623	
Algoritmo applicato	**Benchmark Rimossi**	**Benchmark Rimanenti**
Normalizzazione	96	527
MinMax	18	605

Tabella 7.3: Risultati algoritmi rimozione benchmark

Risultati solo con normalizzazione

Il modello ottenuto con solo operazioni RTL, benchmark appartenenti alla classe integer e rimozione di benchmark con algoritmo di normalizzazione è riassunto nella Tabella 7.4.

Classe RTL	μ	σ^2	Significatività
assignment	74.657	401.689	***
bit_int_expr	-19.836	114.957	.
branch_expr	80.842	559.434	***
call_expr	-25.585	736.276	
comp_unknown_expr	-117.730	1797.334	**
const_readings	6.257	76.873	
integer_memory_readings	-29.308	192.157	*
integer_memory_writings	-106.769	646.467	***
integer_register_accesses	14.854	56.403	*
integer_register_writings	-89.710	344.364	***
is_main	2280.895	603.424	***
plusminus_int_expr	-0.748	138.687	

Tabella 7.4: Medie, varianze e significatività dei regressori con modello intero normalizzato

Nella Tabella 7.5 sono riassunte le caratteristiche del modello intero normalizzato.

Media dei Residui	2.92e-08	
Coefficiente R^2	0.98818	98.818%
Errore Relativo	0.132	13.2%
Training benchmark nell'IC 99.7%	527/527	100%
Training benchmark nell'IC 95.5%	527/527	100%
Training benchmark nell'IC 68.3%	527/527	100%
Benchmark nell'IC 99.7%	621/623	99.68%
Benchmark nell'IC 95.5%	620/623	99.52%
Benchmark nell'IC 68.3%	619/623	99.36%

Tabella 7.5: Caratteristiche del modello intero normalizzato

Nella Tabella 7.6 sono riassunti i risultati dei test statistici applicati al modello mostrato in Tabella 7.4.

T-Test	✔
Shapiro-Wilk Test	✔
Box-Pierce Test	✔

Tabella 7.6: Risultati test su modello intero normalizzato

Commenti sul modello

Il modello mostrato in Tabella 7.4 risultante dall'applicazione dell'algoritmo di normalizzazione all'insieme di benchmark di classe *integer* stimando le operazioni RTL, è un buon modello di stima. Innanzitutto il coefficiente R^2 di determinazione è elevato, il che significa che i regressori predicono bene il valore della variabile dipendente in campione ossia il tempo totale di esecuzione di ciascun benchmark. Inoltre, quasi tutti i regressori hanno una significatività alta nello spiegare il tempo di cicli totale per ciascun benchmark, segno, assieme all'elevato coefficiente R^2, che il modello è significativo.

La media dei residui è praticamente nulla, nell'ordine di 10^{-8}, e l'errore relativo è del 13%, segno della bontà del fitting del modello. La percentuale di benchmark del training set che cadono in tutti i tre gli intervalli di confidenza è del 100% e la percentuale dell'insieme totale dei benchmark è molto alta, più dell 99%, segno anche questo della precisione del modello.

Il modello inoltre supera tutti e tre i test statistici e risulta quindi essere OLS, la regressione lineare applicata al modello è quindi efficace poichè il modello è lineare.

In conclusione, il modello 7.4 è in grado di stimare efficacemente media e varianza di operazioni RTL operanti su dati interi ed è in grado di costruire intervalli di confidenza significativi per il tempo totale di esecuzione di un applicazione.

Risultati con algoritmo MinMax

Il modello ottenuto con solo operazioni RTL, benchmark appartenenti alla classe integer e rimozione di benchmark con algoritmo MinMax è riassunto nella Tabella 7.7.

Classe RTL	μ	σ^2	Significatività
assignment	78.618	418.941	***
bit_int_expr	-20.030	122.108	.
branch_expr	84.094	575.242	***
call_expr	-25.815	782.727	
comp_unknown_expr	-126.048	1869.967	**
const_readings	6.939	84.297	
integer_memory_readings	-31.830	217.827	*
integer_memory_writings	-109.737	715.052	***
integer_register_accesses	13.427	60.612	.
integer_register_writings	-90.788	357.095	***
is_main	2282.292	626.295	***
plusminus_int_expr	0.986	145.142	

Tabella 7.7: Medie, varianze e significatività dei regressori con modello intero e applicazione algoritmo MinMax

Nella Tabella 7.8 sono riassunte le caratteristiche del modello intero risultante dall'applicazione dell'algoritmo MinMax.

Nella Tabella 7.9 sono riassunti i risultati dei test statistici applicati al modello mostrato in Tabella 7.7.

Media dei Residui	5.504e-08	
Coefficiente R^2	0.98884	98.884%
Errore Relativo	0.115	11.5%
Training benchmark nell'IC 99.7%	510/510	100%
Training benchmark nell'IC 95.5%	510/510	100%
Training benchmark nell'IC 68.3%	510/510	100%
Benchmark nell'IC 99.7%	621/623	99.68%
Benchmark nell'IC 95.5%	620/623	99.52%
Benchmark nell'IC 68.3%	619/623	99.36%

Tabella 7.8: Caratteristiche del modello intero risultato dell'applicazione dell'algoritmo MinMax

T-Test	✔
Shapiro-Wilk Test	✗
Box-Pierce Test	✔

Tabella 7.9: Risultati test su modello intero e applicazione algoritmo MinMax

7.5.2 Commenti sul modello

Il modello mostrato in Tabella 7.4 risultante dall'applicazione dell'algoritmo MinMax all'insieme di benchmark di classe *integer* stimando le operazioni RTL, è molto simile al modello mostrato in Tabella 7.4 e come tale presenta caratteristiche simili. E' anch'esso un modello di stima significativo, il coefficiente R^2 di determinazione è elevato, il che significa che i regressori predicono bene il valore della variabile dipendente in campione ossia il tempo totale di esecuzione di ciascun benchmark, e quasi tutti i regressori hanno una significatività alta nello spiegare il tempo di cicli totale per ciascun benchmark.

La media dei residui è praticamente nulla, nel'ordine di 10^{-8}, e l'errore relativo è dell'11.5%, segno della bontà del fitting del modello anche superiore al modello di Tabella 7.4. La percentuale di benchmark del training set che cadono in tutti i tre gli intervalli di confidenza è del 100% e la percentuale dell'insieme totale dei benchmark è molto alta, più dell 99%, segno anche questo della precisione del modello.

Il modello non supera però il test di Saphiro Wilk di verifica di normalità dei residui. Il modello non può essere propriamente considerato OLS e quindi

lineare, anche se la media dei residui, essendo praticamente nulla, ci consente di approssimarlo come tale. Infatti l'errore relativo si mantiene comunque basso, addirittura inferiore al caso con normalizzazione per somma di coefficienti, che poteva invece essere considerato OLS.

In conclusione, il modello 7.7 è in grado di stimare efficacemente media e varianza di operazioni RTL operanti su dati interi ed è in grado di costruire intervalli di confidenza significativi per il tempo totale di esecuzione di un applicazione, nonostante non sia propriamente un modello lineare.

7.5.3 Risultati con benchmark float

Vengono ora presentati i risultati ottenuti con benchmark *float* contenenti in prevalenza operazioni in virgola mobile di classe RTL. La tabella 7.10 riassume i dati relativi al numero iniziale di benchmark e al numero di benchmark rimossi grazie agli algoritmi di rimozione benchmark.

Benchmark iniziali	119	
Algoritmo applicato	**Benchmark Rimossi**	**Benchmark Rimanenti**
Normalizzazione	21	98
MinMax	0	119

Tabella 7.10: Risultati algoritmi rimozione benchmark

Nella Tabella 7.11 vengono riassunte le caratteristiche e i risultati dei test sui modelli risultanti dall'applicazione degli algoritmi di rimozione benchmark alle operazioni di classe RTL sull'insieme di benchmark *float*.

7.5.4 Commenti sul modello

Dalla Tabella 7.11 si può notare che i modelli risultanti dall'applicazione della metodologia su benchmark in virgola mobile sono difficilmente utilizzabili per la stima di prestazioni. Sebbene il ceofficente R^2 di determinazione sia elevato, segno che la regressione è stata effettuata correttamente, la significatività dei regressori è, per entrambi gli algoritmi di rimozione benchmark, è complessivamente bassa, un solo regressore significativo, ossia l'operazione

Caratteristica \ Benchmark	Rim. Mal Normalizzati	MinMax
Coefficente R^2	0.93183	0.97228
Regressori Significativi	1/20 5%	1/20 5%
Media dei Residui	-3.718e-08	-2.092e-08%
Errore Relativo	32.8%	33.2%
T-Test	✗	✓
Shapiro-Wilk Test	✗	✗
Box-Pierce Test	✓	✗

Tabella 7.11: Caratteristiche dei modelli RTL float

`is_main`. L'errore relativo è di conseguenza elevato, circa il 33%, e i modelli non sono considerabili come OLS lineari dato che falliscono 2 su 3 test statistici. Complessivamente, i due modelli RTL *float* non sono buoni modelli di stima.

Le motivazioni di tale conclusione vanno ricercate nei benchmark stessi. E' molto difficile trovare benchmark dotati di una certa complessità (non semplici main) che contengano una prevalenza di operazioni in virgola mobile; infatti l'insieme di benchmark float è composto da soli 119 benchmark. Se si vogliono stimare le 20 classi di operazioni RTL contenute nei benchmark *float* sono necessari almeno 200 benchmark per non violare la Condizione 2.3.3. Con soli 199 benchmark la condizione risulta violata e di conseguenza il modello risulta poco significativo.

7.5.5 Risultati con tutti i benchmark

Vengono ora presentati i risultati ottenuti con tutti i benchmark e operazioni di classe RTL. La tabella 7.12 riassume i dati relativi al numero iniziale di benchmark e al numero di benchmark rimossi grazie agli algoritmi di rimozione benchmark.

Benchmark iniziali	742	
Algoritmo applicato	**Benchmark Rimossi**	**Benchmark Rimanenti**
Normalizzazione	117	625
MinMax	117	625

Tabella 7.12: Risultati algoritmi rimozione benchmark

Nella Tabella 7.13 vengono riassunte le caratteristiche e i risultati dei test sui modelli risultanti dall'applicazione degli algoritmi di rimozione benchmark alle operazioni di classe RTL sull'insieme di benchmark *float*.

Caratteristica \ Benchmark	Rim. Mal Normalizzati	MinMax
Coefficente R^2	0.98018	0.98034
Regressori Significativi	3/20 5%	4/20 5%
Media dei Residui	3.697e-08	5.653e-08%
Errore Relativo	13.6%	12.6%
T-Test	✔	✔
Shapiro-Wilk Test	✗	✗
Box-Pierce Test	✔	✔

Tabella 7.13: Caratteristiche dei modelli RTL float

7.5.6 Commenti sul modello

Dalla Tabella 7.13 si può notare che i modelli risultanti dall'applicazione della metodologia su benchmark in virgola mobile risultano peggiori dei modelli risultanti dai benchmark di tipo *integer* ma migliori di quelli risultantti dai benchmark *float*. Il coefficiente R^2 di determinazione è elevato, è simile al caso *integer*, segno che la regressione è stata effettuata correttamente e che i regressori predicono bene il valore della variabile dipendente in campione ossia il tempo totale di esecuzione di ciascun benchmark. La significatività dei regressori è simile al caso *float*: per entrambi gli algoritmi di rimozione benchmark è complessivamente piuttosto bassa, 3 o 4 regressori significativi su 20, segno che non tutti i regressori hanno grosso potere esplicativo nei confronti del tempo totale di esecuzione. L'errore relativo però è buono, circa il 13%, come nel caso *integer*. Entrambi i modelli non sono propriamente considerabili come OLS lineari dato che entrambi falliscono il test di Shapiro-Wilk di normalità dei residui, anche se la media dei residui, essendo praticamente nulla in entrambi i casi, ci consente di approssimarli come tali.

Le motivazioni delle caratteristiche dei modelli sono da attribuirsi anche in questo caso ai benchmark *integer*. L'aggiunta all'insieme di benchmark *integer*,

che conduce a buoni modelli di stima, dei benchmark *float*, che invece conduce a modelli non buoni, e di altri benchmark, produce modelli di stima aventi significatività dei regressori più bassa rispetto al caso con soli benchmark *integer*. Dal momento però che i benchmark float sono in numero decisamente inferiore rispetto all'insieme complessivo dei benchmark, il peggioramento delle caratteristiche dei modelli è limitato, il coefficente R^2 si mantiene elevato e l'errore relativo si mantiene contenuto.

In conclusione, i modelli derivanti dai benchmark *all* possono essere utilizzati per la stima di prestazioni e la generazione di intervalli di confidenza ottenendo risultati di stima accettabili.

Capitolo 8

Conclusioni e possibili sviluppi futuri

In questo lavoro di tesi è stata proposta una possibile metodologia di stima di prestazioni con uso di intervalli di confidenza basata su modelli matematici. La metodologia proposta si avvale delle rappresentazioni intermedie GIMPLE ed RTL e sfrutta la regressione lineare per il calcolo delle medie e delle varianze del tempo di esecuzione di ogni singola istruzione GIMPLE o RTL contenuta in una data applicazione sorgente. Sulla base delle medie e delle varianze calcolate viene effettuata una stima del tempo totale di esecuzione dell'applicazione e vengono generati intervalli di confidenza per il tempo di esecuzione a diversi livelli di significatività (o confidenza).

I risultati dei test proposti hanno dimostrato che avendo a disposizione un numero sufficientemente elevato di benchmark, la metodologia è in grado costruire efficacemente modelli di stima e generare intervalli di confidenza atti a fornire un indice dell'affidabilità della stima stessa, utile nel flusso di progettazione ai fini della verifica dei vincoli temporali.

Esistono tuttavia possibili margini di miglioramento e possibili estensioni.

- estendere la classe *variance_estimation* tramite l'introduzione di nuove distribuzioni di probabilità che possano migliorare i risultati di stima;

- considerare anche altre tecniche di regressione per modelli non strettamente lineari OLS come la *Ridge Regression* la *Principal Components Regression*, la *Regressione Quantilica* e la *Regressione Robusta*, non trattate in questo lavoro di tesi, ma che potrebbero fornire modelli di stima significativi anche in presenza di modelli non lineari [30];

- considerare e applicare ulteriori test statistici rispetto ai tre utilizzati nella metodologia proposta, al fine di verificare meglio l'effettiva linearità del modello;

- costruire intervalli di confidenza significativi anche per i benchmark dell'insieme *float* ricercando altri benchmark contenenti in prevalenza operazioni in virgola mobile in modo da avere un fattore dieci tra il numero di classi di operazioni RTL e il numero di benchmark ottenendo così modelli di stima significativi;

- aumentare il numero di benchmark utilizzati per l'applicazione del metodo di regressione lineare, sfruttando altre suite di benchmark in modo da migliorare l'eterogeneità del training set limitando i problemi di polarizzazione e overfitting e di conseguenza la necessità degli algoritmi di rimozione benchmark.

Riferimenti bibliografici

[1] SimIt-ARM 3.0: an ARM instruction-set simulator.

[2] Boost C++ libraries, 2006.

[3] Aeroflex Gaisler AB. Tsim, a generic SPARC architecture simulator.

[4] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, e M. Yarrow. The NAS parallel benchmarks 2.0. technical report.

[5] Thomas Ball e James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pp. 46–57. IEEE Computer Society, 1996.

[6] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, e Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pp. 82–86. ACM, 2000.

[7] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, e V. Trianni. Modeling assembly instruction timing in superscalar architectures. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pp. 132–137. ACM, 2002.

[8] Per Bjureus e Axel Jantsch. MASCOT: A specification and cosimulation method integrating data and control flow. In *Proceedings of the Design, Automation and Test in Europe*, March 2000.

[9] Per Bjureus e Axel Jantsch. *Performance Analysis with Confidence Intervals for Embedded Software Processes*, 2005.

[10] G. Bontempi e W. Kruijtzer. A data analysis method for software performance prediction. In *Design, Automation and Test in Europe Conference and Exhibition*, pp. 0–971, 2002.

[11] C. Brandolese, W. Fornaciari, F. Salice, e D. Sciuto. Source-level execution time estimation of c programs. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pp. 98–103, New York, NY, USA, 2001. ACM.

[12] A. J. Dorta, C. Rodriguez, F. de Sande, e A. Gonzalez-Escribano. The OpenMP Source Code Repository. In *PDP*, pp. 244–250, 2005.

[13] Free Software Foundation. GNU Compiler Collection GCC, version 4.3.

[14] Wikimedia Foundation Inc. Wikipedia, The Free Encyclopedia.

[15] Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, e Heinrich Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pp. 325–330. ACM, 2008.

[16] Paolo Giusto e Grant Martin. *Reliable Estimation of Execution Time of Embedded Software*. 70 Billerica Rd, Chelmsford MA, 01824, U.S.A, 2001.

[17] R Group. The R project for statistical computing.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, e R. B. Brown. Mibench: A free, commercially representative, embedded benchmark suite. In *WWC*, pp. 3–14, 2001.

[19] Project Kenai. Legion SPARC, a fast instruction accurate simulator.

[20] Marcello Lajolo, Mihai Lazarescu, e Alberto Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *CODES '99: Proceedings of the seventh international work- shop on Hardware/software codesign*, pp. 85–89. ACM, 1999.

[21] M. Lattuada e F. Ferrandi. Performance Modeling of Embedded Applications with Zero Architectural Knowledge. In *CODES 2010+ISSS: International Conference on Hardware/Software Codesign and System Synthesis*, p. To appear, 2010.

[22] Jie Liu, Marcello Lajolo, e Alberto Sangiovanni-Vincentelli. Software timing analysis using hw/sw cosimulation and instruction set simulator. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pp. 65–69. IEEE Computer Society, 1998.

[23] Andrew Makhorin e altri. GLPK: GNU Linear Programming Kit, 2008.

[24] A. Malik, B. Moyer, e D. Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pp. 241–243, NY, USA, 2000. ACM.

[25] Heinrich Meyr, Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, e Rainer Leupers. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pp. 468–473, 3001 Leuven, 2006. European Design and Automation Association.

[26] μ-lab Dipartimento di Elettronica e Informazione. PandA Project, 2006.

[27] M. Oyamada, F. R. Wagner, M. Bonaciu, W. Cesario, e A. Jerraya. Software performance estimation in mpsoc design. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pp. 38–43. IEEE Computer Society, 2007.

[28] Marcio Seiji Oyamada, Felipe Zschornack, e Flávio Rech Wagner. Applying neural networks to performance estimation of embedded software. In *J. Syst. Archit.*, pp. 54(1–2):224–240, 2008.

[29] Vito Ricci. *Fitting distributions with R*, febbraio 2005.

[30] Vito Ricci. *Principali tecniche di regressione con R*, settembre 2006.

[31] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press, 4 edizione, 2009.

[32] Sam S. Shapiro e Martin Bradbury Wilk. An analysis of variance test for normality (complete samples). In *Biometrika*, pp. 591–611. 1965.

[33] Hans-Joachim Stolberg, Mladen Berekovic, e Peter Pirsch. A platform-independent methodology for performance estimation of multimedia signal processing applications. In *J. VLSI Signal Process. Syst.*, pp. 1(2):139–151, 2005.

[34] Kei Suzuki e Alberto Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pp. 605–610. AP, 1996.

[35] World Wide Web Consortium (W3C). eXtensible markup language (XML).

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, , e A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, pp. 24–36, 1995.

[37] V. Zivojnovic, J. M. Velarde, C. Schlager, e H. Meyr. DSPSTONE: a DSP-oriented benchmarking methodology. In *ICSPAT'94: International Conference on Signal Processing and Technology*, 1994.

Ringraziamenti

Finalmente arrivo a scrivere i ringraziamenti di questa tesi! Pensavo che questo momento non arrivasse mai! E invece eccomi qui! Prima di tutto voglio ringraziare un giorno, il 1 Ottobre 2007, e voi direte perché? Perché in quella data ho conosciuto tre persone senza le quali probabilmente oggi non sarei qui alla fine di questo (lungo!) percorso universitario, cioè Cecio, Oli e lo zio Andre. Grazie per tutto ciò che avete fatto per me in questi sei anni, Grazie perché senza di voi a volte non ce l'avrei fatta. Grazie perché amici come voi non si trovano tutti i giorni!

Il secondo, ma non meno importante, Grazie va alla mia famiglia adottiva, che mi ha convinto a venire a Milano, mi ha accolto come un figlio e soprattutto sopportato! Nonostante le minacce di trasloco in un igloo in giardino se non studiavo! Grazie a Marina, alla mia sorellona Valentina (ora posso colonizzare la mansarda!), a Socks e Margi ma soprattutto a Carlo! Zio questa tesi la dedico a te, perché so quanto ci tenevi a vedermi laureato, perché sei sempre stato il mio sostenitore N.1, perché lo so che da lassù ci hai sempre aiutato... e perché ci manchi tantissimo... a me in particolare.

Come non ringraziare poi la mia di famiglia! Quella vera stavolta! Grazie Mamma e Papà per tutti i sacrifici che avete fatto per me, perché non mi avete fatto mancare niente, perché mi avete sempre sostenuto e soprattutto per tutte le volte che mi avete dovuto sopportare, lo so che ho un caratteraccio ma sono io il primo che mi devo sopportare! Grazie a mio fratello Andrea, perché forse è lui la persona che più di tutti mi ha dovuto sopportare e mi sembra ci sia riuscita e Grazie a tutti i miei parenti perché anche loro ci sono stati sempre!

C'è poi un'altra famiglia che devo ringraziare ed è la famiglia Guffanti! Carlotta, il Titti e soprattutto mamma Isa che mi ha trattato come un figlio riempiendomi di orgoglio quando mi diceva che ero il figlio bravo!

Mi sembra doveroso poi ringraziare Marco! Grazie perché senza di te questa tesi non sarebbe stata possibile e per tutto il tempo che mi hai dedicato! Assieme a te ringrazio tutti gli insegnanti e i professori che, una lezione alla volta, mi hanno portato fino a raggiungere questo traguardo.

Chi altri devo ringraziare? Bé ovviamente la mia compagnia! Anzi le mie

compagnie! Grazie a Mattia, Bricco, Bibi, Giulio, Caste, alla mia collega ingegnera Viola, Serena, il nostromo Leo, Marche, Pitta, Richy, Ruppia, Gheorghe, Fede per tutti questi anni passati insieme e perché con voi sono cresciuto e spero di continuare crescere!

Grazie a tutti quelli con cui ho condiviso un pezzo di strada fino a qui, il fatto che non ci sia il vostro nome non vi rende meno importanti!

E se mi sono dimenticato qualcuno mi scuso ma mi conoscete, ho pessima memoria!