Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Corso di laurea in Ingegneria Informatica

Una Metodologia per l'identificazione dei punti critici in applicazioni hard real time

Relatore: Prof. Fabrizio Ferrandi
Correlatore: Ing. Marco Lattuada

Tesi di laurea di
Francesco Maria Felice Gastone Guffanti
Matr. 724813

Anno Accademico 2009/2010

a me

Sommario

La progettazione di un sistema dedicato è spesso guidata dal soddisfacimento di requisiti prestazionali. Esiste tuttavia una serie di campi d'utilizzo dove la proprietà fondamentale, richiesta a un elaboratore, è la capacità di fornire il risultato in un tempo massimo, calcolato precedentemente, ed imposto come requisito per l'utilizzo del sistema stesso. Questi sistemi prendono il nome di sistemi Hard Real Time e in essi il soddisfacimento dei vincoli temporali nel caso pessimo assume maggiore importanza del tempo medio d'esecuzione della funzionalità prevista. La predicibilità del sistema deve quindi essere accuratamente garantita sin dalle prime fasi della progettazione. Questo lavoro di tesi sviluppa una nuova metodologia di identificazione dei punti critici in applicazioni Hard Real Time basata sull'analisi dei possibili effetti della memoria cache. In particolare si ricercano all'interno del codice sorgente quelle sequenze di codice che, in accordo alla metodologia proposta, costituiscono una fonte di impredicibilità se eseguite su un sistema dotato di memoria cache. I risultati sperimentali mostrano come la metodologia possa essere facilmente integrata all'interno di un moderno compilatore.

Indice

Elenco delle tabelle

Elenco delle figure

Elenco dei codici

Capitolo 1

Introduzione

La richiesta di sempre maggiori prestazioni ha portato al design di architetture multiprocessore a singolo chip con gerarchie di memoria multilivello che sì determinano un significativo aumento di performance ma introducono ulteriori problematiche nella stima delle prestazioni del sistema.

Questo aspetto rappresenta un punto fondamentale dell'analisi dei sistemi Hard Real Time in cui un task che supera temporalmente la sua deadline provoca un danno irreparabile al sistema. Un esempio di task hard real time può essere rappresentato dal controllore della temperatura del nocciolo di una centrale nucleare, dove il mancato rispetto dei vincoli temporali può provocare un evidente disastro.

La criticità del calcolo del tempo di esecuzione del task ha spostato l'attenzione su quegli elementi (memorie cache, pipeline, branch predictor, etc.) che, all'inerno di un sistema hard real time, possono influenzarne il tempo di completamento. Se da un lato il tempo medio d'esecuzione dipende maggiormente dalle prestazioni del sistema, dall'altro il caso pessimo dipende da situazioni particolari che si possono verificare solo sotto particolari condizioni del sistema. Per esempio se il task viene eseguito all'interno del flusso di esecuzione di altri task o parallelamente ad essi, questi possono influenzare significativamente il tempo di esecuzione dal momento che influenzano lo stato della memoria al momento del lancio del task.

Per questo motivo l'analisi della predicibilità del sistema deve tener conto sia di fattori calcolabili staticamente sia del comportamento del sistema durante l'esecuzione. L'analisi di predicibilità diventa fondamentale in quanto fornisce parametri

importanti per il progettista e costituisce un indice di quanto sia attendibile e precisa la stima del WCET.

Il presente lavoro di tesi si basa sul progetto PEAL[1] in cui è stato analizzato come la presenza di una memoria cache possa influenzare l'analisi del Worst Case Execution Time.

Il contributo apportato da questo lavoro di tesi consiste in una nuova metodologia per l'identificazione dei punti critici che, sfruttando le rappresentazioni intermedie del compilatore GNU GCC, permette di individuare le sequenze di codice definite critiche dalla metodologia per il problema della predicibilità e che possono influenzare quindi in modo significativo la stima del Worst Case Execution Time.

Il presente lavoro di tesi è suddiviso in sette capitoli.
Il Capitolo 2 contiene le definizioni preliminari relative alla struttura del compilatore GNU GCC e delle sue rappresentazioni intermedie, alle tecniche di *software profiling* e infine alla struttura e al funzionamento della memoria cache.
Il Capitolo 3 propone una panoramica dello stato dell'arte relativa alla Stima del Worst Case Execution Time per architetture che presentino elementi che rendano incerta l'esecuzione dei programmi, con particolare attenzione alla memoria cache.
Il Capitolo 4 illustra la metodologia sviluppata durante l'intero lavoro di tesi.
Il Capitolo 5 riporta una breve descrizione di PandA, il framework all'interno del quale è stata implementata la metodologia proposta.
Il Capitolo 6 presenta i risultati sperimentali relativi all'applicazione della metodologia alla suite di benchmark MiBench.
Il Capitolo 7, infine, riporta l'analisi conclusiva del lavoro svolto evidenziando i possibili sviluppi futuri.

Capitolo 2

Definizioni Preliminari

Nella prima parte del Capitolo verranno descritte la struttura del compilatore GNU GCC e le sue rappresentazioni intermedie, prestando particolare attenzione al linguaggio GIMPLE. Nella seconda parte verranno introdotti i concetti di *Blocco Basico*, di *Control Flow Graph*, di ciclo *(Loop)* e di *archi di feedback*. Seguirà la descrizione del *Software Profiling* e infine sarà presentata la memoria *Cache*.

2.1 Il Compilatore GNU GCC

Il GNU Compiler Collection [6] (abbreviato in GCC) è un compilatore prodotto dal GNU Project ed è grado di supportare vari linguaggi di programmazione.

Usualmente il compilatore è un programma che prende come input un'applicazione scritta in codice sorgente e fornisce come output un'altra applicazione semanticamente equivalente, ma in un differente linguaggio (codice oggetto). Il processo di trasformazione da codice sorgente a codice oggetto attraversa tre fasi che interessano le tre componenti principali del GCC: *Front End*, *Middle End* e *Back End*, mostrati in Figura 2.1.

Durante la fase di Front End viene analizzato il file che contiene il codice sorgente in modo che la sua sintassi venga validata. Nel caso in cui la sintassi sia validata viene costruito l'*Albero Sintattico* astratto (AST[1]). L'AST rappresenta ogni istruzione del programma da compilare e si serve della rappresentazione in-

[1]Dall'inglese *Abstract Syntax Tree.*

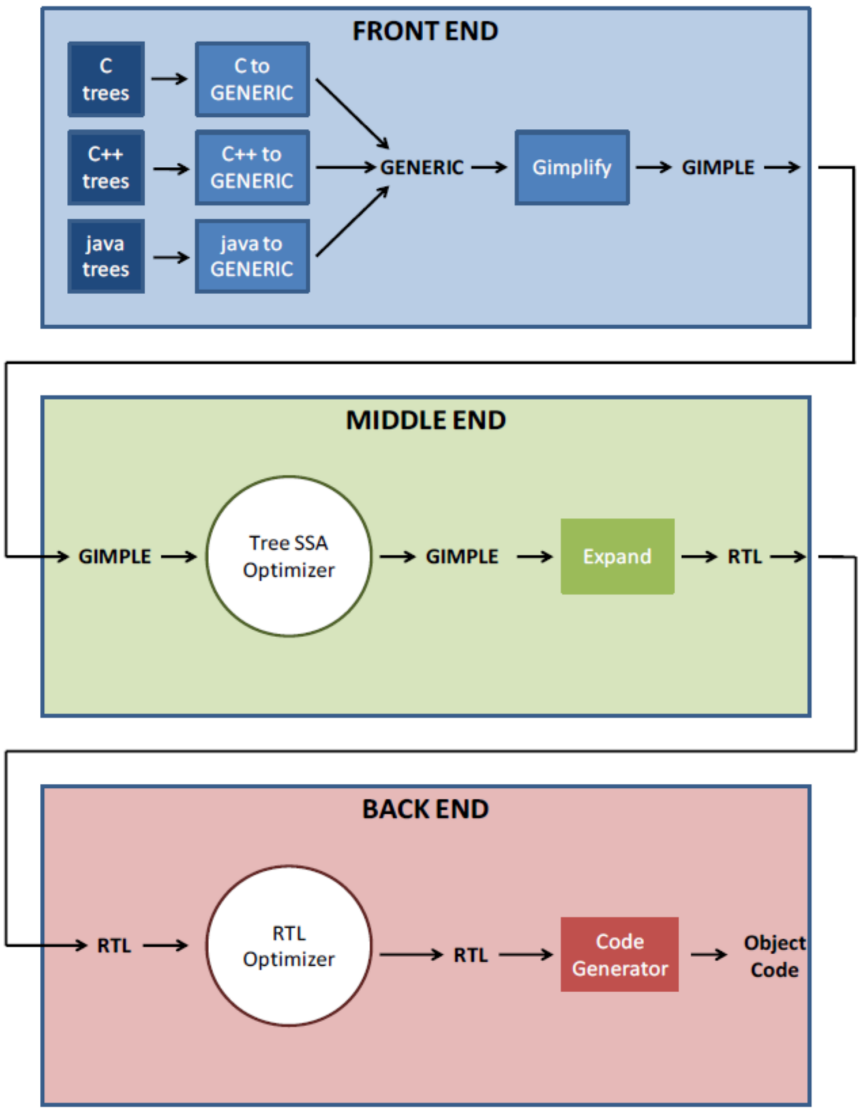

Figura 2.1: Struttura del compilatore GNU GCC.

termedia GENERIC. La struttura GENERIC è però troppo complessa per poi poterne fruire facilmente; si passa dunque a una rappresentazione meno complessa e sempre ad albero detta GIMPLE.

La fase intermedia detta Middle End ha come dato d'ingresso l'albero GIMPLE e in essa vengono effettuate ottimizzazioni del programma indipendenti dall'architettura.

Esempi ottimizzazioni applicate in questa parte sono le *semplificazioni algebriche* e il *constant folding*. Nelle prime vengono sfruttate le conoscenze algebriche per semplificazione delle espressioni originali (per esempio variabili moltiplicate per l'elemento neutro della moltiplicazione o sommati all'elemento neutro della somma vengono sostituiti dalle variabili stesse[2] Con constant folding invece si indica la risoluzione a *tempo di compilazione* delle istruzioni che hanno come operandi solo valori costanti[3].

Oltre al Middle End anche il Back End prevede delle ottimizzazioni e la trasformazione del programma in codice adatto all'esecuzione sull'architettura target.

2.2 Le Rappresentazioni Intermedie del Compilatore GCC

Il processo di trasformazione del codice attuato dal compilatore GCC prevede la creazione di tre rappresentazioni intermedie denominate, nell'ordine del flusso di compilazione, GENERIC, GIMPLE ed RTL.

2.2.1 GENERIC

GENERIC si pone come obiettivo quello di rappresentare il codice sorgente sotto forma di albero, indipendentemente da quale sia il linguaggio sorgente. Il compilatore trasforma il codice una funzione alla volta in strutture ad albero chiamate *Alberi Sintattici* dove le altre funzioni e tutte le istruzioni (dichiarazioni, asse-

[2]Per esempio : $b * 1 = b$ oppure $a + 0 = a$.
[3]Come ad esempio : $z = 1 + 2 + 1 + 4 + 3 = 11$.

gnamenti, costrutti condizionali, etc.) del codice sorgente sono tradotte in nodi dell'albero.

2.2.2 GIMPLE

Una volta che si ottiene la rappresentazione GENERIC si passa al linguaggio GIMPLE esplicitamente progettato per essere utilizzato in modo efficace nelle ottimizzazioni. Il passo del compilatore che genera la GIMPLE è detto *gimplifier*.

Il funzionamento di gimplifier è di tipo ricorsivo; le tuple GIMPLE sono generate seguendo la struttura dati delle espressioni originarie di GENERIC.

Verranno presentati gli elementi principali di GIMPLE e un esempio di rappresentazione.

- **Espressioni**: tipicamente un'espressione GIMPLE è una tupla composta da un operatore e un numero massimo di tre semplici operandi che possono essere costanti o variabili.
  Le operazioni più complesse vengono fattorizzate in espressioni temporanee ternarie dal gimplifier, seguendo un processo conosciuto come *expression temporaries*.
  Per esempio l'espressione:

```
a = b + c + d;
/* viene trasfomrata in */
Temp_1 = b + c;
a = Temp_1 + d;
```

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

```
a = b ? c : d;
/* è trasformata in */
```

```
if(b)
    Temp_1 = c;
else
    Temp_1 = d;
a = Temp_1;
```

- **Condizioni Logiche**: quando gli operandi logici *and* e *or* non appaiono come operandi di un'espressione condizionale vengono trasformati come segue:

```
a = b && c;
/* si tramuta in */
Temp_1 = bool b
if(Temp_1 == TRUE)
    Temp_1 = bool c;
a = Temp_1;
```

- **Costrutti 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.

- **Cicli**: nella versione attuale del GCC i cicli sono rappresentati per mezzo del costrutto *if-goto* [6]

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

2.2.3 RTL

La rappresentazione RTL (Register Transfer Language) è una Rappresentazione Intermedia utilizzata dal compilatore GCC nella fase di Back End, con lo scopo

di eseguire le ottimizzazioni target dependent.

Nonostante il linguaggio RTL (la sua grammatica) sia indipendente dall'architettura target, durante la traduzione dalla rappresentazione GIMPLE vengono prese in considerazione alcune caratteristiche dell'architettura di destinazione. Quest'aspetto fa sì che le rappresentazioni RTL della stessa applicazione su diverse architetture presentino delle differenze anche significative.

La struttura della rappresentazione RTL è composta da una lista di espressioni simboliche annidate.

2.3 Rappresentazioni basate su Grafi

Le rappresentazioni intermedie del compilatore GCC possono essere espresse graficamente tramite l'utilizzo di grafi orientati.

Nella Sezione 2.3.1 sarà definito cosa sia un *Blocco Basico* mentre nella Sezione 2.3.2 sarà mostrata una possibile rappresentazione basata su grafo detta *Control Flow Graph*. La Sezione 2.3.2.1 presenterà la rappresentazione dei *cicli*.

2.3.1 Blocchi Basici

Un blocco basico è una porzione di codice di un programma con alcune proprietà che lo rendono facilmente analizzabile.

- Il codice in un blocco basico ha un unico punto di ingresso (operazione di destinazione di un salto) e un unico punto d'uscita. Nessuna istruzione presente all'interno del blocco è un'istruzione di salto o è destinazione di un'altra istruzione di salto.

- Quando inizia l'esecuzione del blocco basico, tutte le istruzioni del blocco basico vengono eseguite in successione e una sola volta.

I blocchi basici a cui si può trasferire il controllo dopo aver raggiunto la fine di quello corrente sono chiamati *successori* del blocco, mentre i blocchi basici da cui proviene il controllo quando si entra in un blocco sono chiamati *predecessori* del blocco.

2.3.2 Control Flow Graph

Il Control Flow Graph (CFG) è un grafo che può essere costruito a partire da una delle rappresentazioni intermedie (RTL o GIMPLE). Sfruttando il CFG si può descrivere il flusso di controllo di una funzione.

I blocchi basici possono costituire i nodi di un CFG definito come $G_d(V, E)$ dove:

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

L'esempio riportato in Figura 2.2 illustra una rappresentazione intermedia del frammento di Codice 2.1. In questa immagine sono visibili i blocchi basici e i flussi d'esecuzione.

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

Codice 2.1: Codice associato all Grafo dei Blocchi Basici in Figura 2.2.

2.3.2.1 La Rappresentazione dei Cicli

Una componente fortemente connessa di un grafo orientato $G(V, E)$ è un insieme massimale di vertici $U \subseteq V$ tale che per ogni coppia di vertici $A, B \in U$ esiste sia

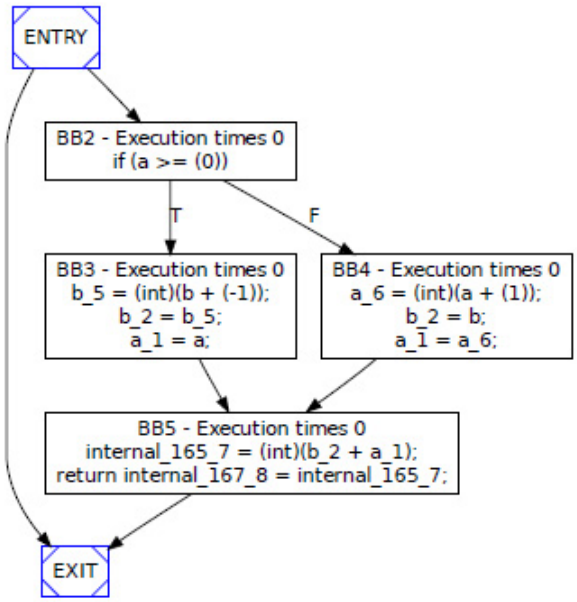

Figura 2.2: Grafo dei Blocchi Basici.

il cammino che va da A a B sia quello che va da B ad A. In Figura 2.3, i nodi A e B rappresentano un semplice esempio di una componente.

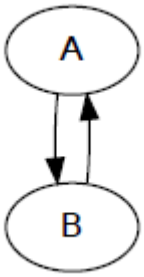

Figura 2.3: Esempio di componente fortemente connessa.

L'insieme dei nodi di ogni componente fortemente connessa all'interno di un Control Flow Graph costituisce un ciclo. Ogni ciclo ha un blocco di entrata, chiamato *header*, ed alcuni archi di *feedback* (descritti in Sezione 2.5). Il corpo di un ciclo è costituito da un insieme di blocchi dominati dall'header.

I cicli sono organizzati in una gerarchia ad albero dove i nodi figli rappresentano cicli innestati mentre i nodi allo stesso livello di profondità nell'albero rappresentano cicli allo stesso livello d'annidamento nel codice. Nell'esempio in Figura 2.4 generato dal Codice 2.2 il ciclo con identificativo *Loop - 5* è figlio di quello con identificativo *Loop - 7* ed è allo stesso livello di annidamento del *Loop - 3*.

```
for(i = 0; i < X; i++)          /* loop - 7 */
{
    for(j = 0; j < Y; j++)      /* loop - 5 */
        matrix[i][j] = i + j;
    for(k = 0; k < K; k++)      /* loop - 3 */
        printf(%d, matrix[4][5]);
}
printf(%d, matrix[4][5]);
```

Codice 2.2: Codice associato al Grafo dei Cicli in Figura 2.4.

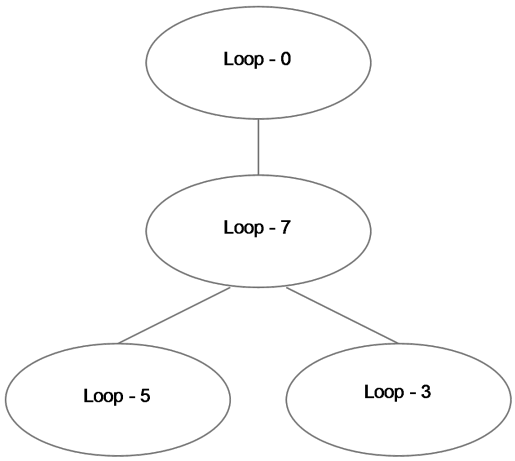

Figura 2.4: Grafo dei Cicli.

L'identificativo *Loop - 0* è riservato all'intero corpo della funzione.

2.3.2.2 Archi di Feedback

Gli *archi di feedback* sono quegli archi (di ritorno) che chiudono un ciclo avente origine nel nodo che rappresenta l'inizio della funzione. La presenza di cicli con

più archi di feedback è possibile quando nel corpo del ciclo sono presenti costrutti condizionali. In un grafo di controllo questi archi che sono presenti nel corpo dei cicli possono essere esplicitati.

Il Codice 2.3 genera il CFG con archi di feedback evidenziati rappresentato in Figura 2.5.

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

Codice 2.3: Codice per la generazione dei Grafi in Figura 2.5

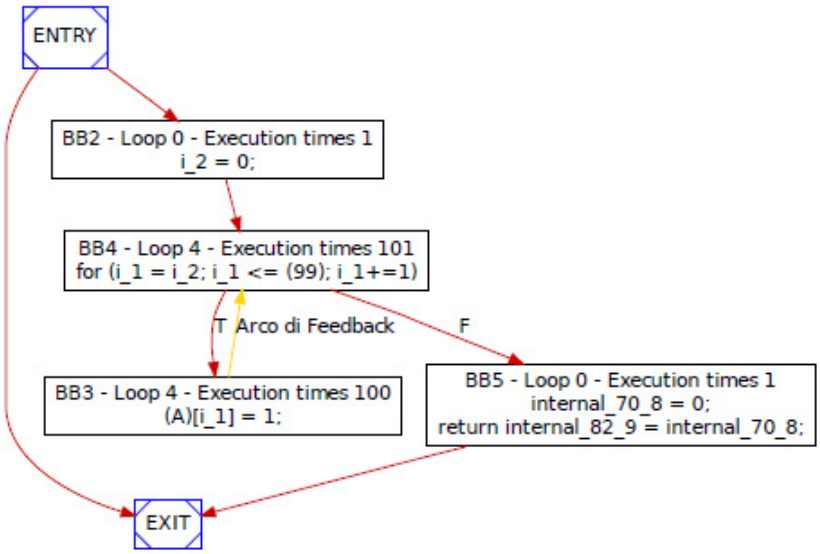

Figura 2.5: CFG con archi di feedback evidenziati ricavato dal Codice 2.3.

2.4 Software Profiling

Il *Program Profiling*, *Software Profiling* o semplicemente *Profiling* è una forma di analisi dinamica di un programma il cui obiettivo è scoprirne il comportamento servendosi delle informazioni derivanti dall'esecuzione del programma stesso.

Tipicamente si cerca di determinare quali siano le sezioni da ottimizzare in modo da aumentare le prestazioni e diminuire l'utilizzo di memoria.

Il codice viene eseguito su una architettura target reale o ideale (simulatore) e per far sì che la tecnica sia efficace il programma target deve essere eseguito per un numero significativo di input.

I vari tipi di *profiler*, l'applicazione che effettua l'analisi, si possono classificare in tre categorie: basati su campionamento, basati su eventi e basati sulle instrumentazioni. I profiler basati su campionamento controllano il *program counter* (PC) del programma target a intervalli regolari usando le routine di *interrupt* [4] del sistema operativo. Ogni volta che il programma viene interrotto si prende nota della porzione di codice che si sta eseguendo in quel momento e alla fine dell'esecuzione si utilizzano questi dati per determinare il tempo di esecuzione dei vari frammenti di codice del programma.

I profiler di questo tipo sono meno accurati di quelli che si servono delle altre tecniche poiché i dati non sono esatti, ma rappresentano un'approssimazione statistica. Questi profiler sono però molto utili in quanto riescono a fornire una panoramica più realistica dell'esecuzione dell'applicazione target non essendo intrusivi dal momento che non perturbano l'esecuzione stessa del programma. Questo permette di limitare gli effetti collaterali, come ad esempio quelli sulla cache. Tuttavia, con questa tecnica si potrebbero perdere alcuni eventi troppo brevi rispetto al tempo di campionamento, come, ad esempio, l'esecuzione di piccole funzioni.

I profiler basati su eventi si attivano quando nel programma profilato accade un evento come ad esempio l'entrata o l'uscita da una funzione. I profiler di questo tipo sono capaci di fornire dettagli di più basso livello ma hanno l'inconveniente di generare un maggiore overhead nell'esecuzione del programma target.

[4] un interrupt è un segnale asincrono che consente l'interruzione di un processo qualora si verifichino determinate condizioni oppure più in generale una particolare richiesta al sistema operativo da parte di un processo in esecuzione.

L'ultimo tipo di profiler è quello che instrumenta il codice del programma target con delle istruzioni aggiuntive che possono essere molto specifiche e facilmente controllate, cosicché il loro impatto sulle prestazioni sia minimo. L'impatto che le instrumentazioni possono avere dipende dalla posizione in cui vengono inserite all'interno del programma e dal meccanismo usato per ottenere la traccia di esecuzione del programma. Inoltre l'errore introdotto dalla instrumentazione spesso può essere dedotto ed eliminato dai risultati. Esistono diverse strategie di inserimento delle instrumentazioni (manuali o automatiche) e a diversi livelli (codice sorgente, rappresentazioni intermedie dei compilatori, oppure direttamente negli eseguibili).

Va detto che la distinzione tra tipologie di profiler presentata non è rigida in quanto sono state progettate soluzioni ibride; ad esempio si può utilizzare sia l'instrumentazione che il campionamento contemporaneamente: la prima per conoscere le chiamate a funzione e il secondo per ottenerne il tempo di esecuzione, in modo da poter usufruire dei pregi di entrambe le tecniche.

2.5 La Memoria Cache

Durante l'evoluzione del calcolatore si presentò il problema di arginare la sempre crescente differenza nell'aumento di prestazioni di processore e Memoria DRAM[5], visibile in Figura 2.6 [29].

Una soluzione efficace fu l'introduzione della memoria cache, una memoria *temporanea* (volatile), trasparente al software, che memorizza un insieme di dati che possono essere *velocemente* (rispetto a quanto lo sarebbero in RAM) recuperati su richiesta [29]. Obiettivo della cache è fornire una memoria la cui velocità si avvicini a quella delle memorie più veloci e che sia allo stesso tempo abbastanza grande, però al costo delle memorie più economiche. La stima di prestazioni di un sistema dotato di memoria cache dipende però fortemente dallo stato della cache al momento in cui viene eseguito il programma rendendo il comportamento impredicibile a seconda delle modalità di analisi e alla struttura del codice processato[6].

[5]La *Dynamic Random Access Memory*, è un tipo di RAM che immagazzina ogni bit in un diverso condensatore. Per le sue caratteristiche di carica è definita memoria dinamica oltre a essere definita volatile in base alla possibilità di perdere il proprio contenuto se non alimentata.

[6]Il determinismo di un sistema dotato di cache dipende anche dalla situazione in cui si decide di lanciare il programma; per esempio è deterministico se il programma da lanciare è l'unico a

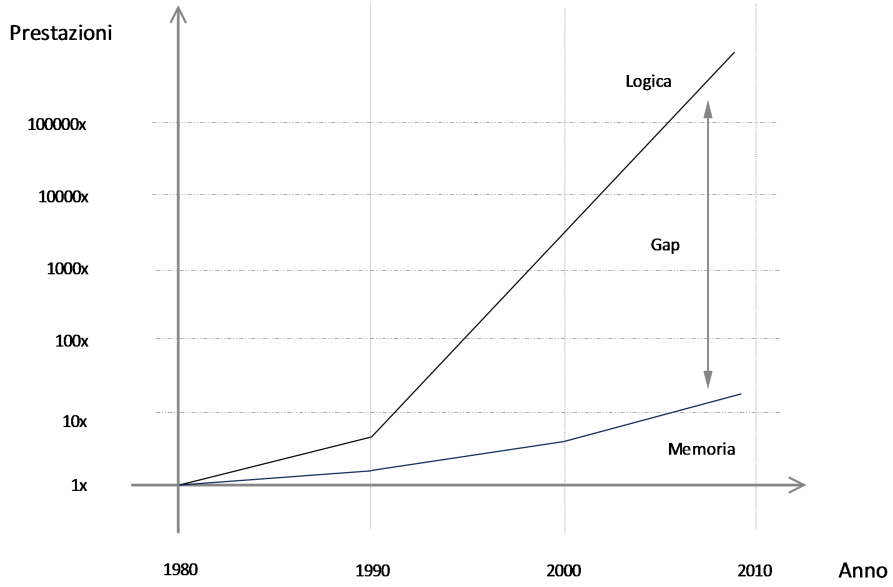

Figura 2.6: Gap di evoluzione tra CPU e DRAM.

In un sistema possono esserci N livelli di cache (solitamente da 1 a 3) denominati cache(L_i) con $i = 1, \ldots, N$ i quali sono solitamente *on-chip*, indicando con questa parola il loro posizionamento fisico sul medesimo chip del processore.

La memoria cache rappresenta il sistema di memorizzazione più veloce dopo i registri del processore e si colloca tra memoria principale e processore come in Figura 2.7. I trasferimenti fra cache(L1) e memorie di livello inferiore sono gestiti dallo hardware. Di norma la cache è trasparente al programmatore e al compilatore.

La cache può essere unificata, quindi un'unica cache ospita sia istruzioni che dati, o divisa in *Instruction-cache* (I-cache) e *Data-cache* (D-cache) come proposto nell'architettura Harvard[7] a seconda delle esigenze. La seconda scelta è preferita nei microprocessori più moderni mentre in alcune architetture per sistemi embedded la presenza della D-cache dipende dall'applicazione prevista.

essere eseguito in quel momento sul calcolatore e non lo è se invece lo stato iniziale della cache è ignoto.

[7]L'architettura Harvard è un'architettura che separa la memorizzazione e la trasmissioni dei dati da quella delle istruzioni, in particolare in alcuni sistemi la larghezza di parola delle istruzioni è superiore a quella dei dati, in altri ancora le istruzioni sono memorizzate in una memoria a sola lettura (ROM) mentre i per i dati si utilizza una memoria a scrittura e lettura (RAM).

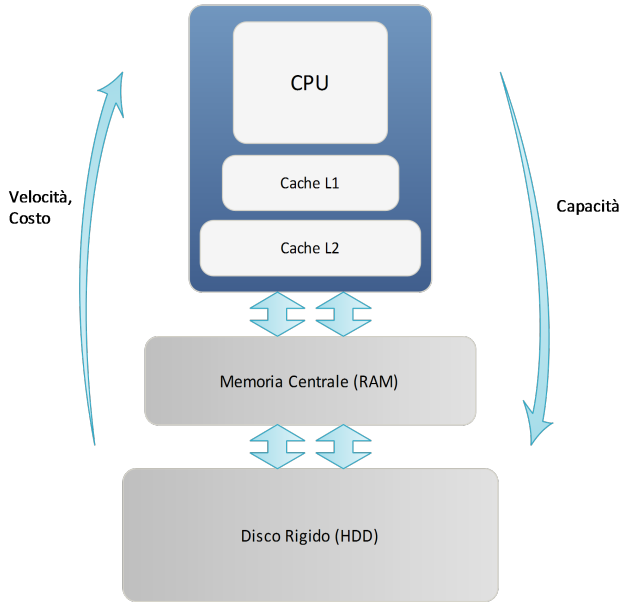

Figura 2.7: Gerarchia di memorie con due livelli cache on-chip.

Il numero dei livelli di cache e le dimensioni dei singoli elementi sono determinati dai requisiti di prestazioni.

In una architettura dotata di cache il funzionamento è di norma il seguente: la CPU accede al (più alto) livello di cache, il controllore della cache determina se l'elemento richiesto è effettivamente presente in cache, se trova l'elemento (*hit*) allora avviene un trasferimento fra cache e CPU, in caso contrario avviene un (*miss*).

In base a Hit_{rate} (frazione degli accessi a memoria ricompensati da uno hit) e a $Miss_{rate}$ (frazione degli accessi a memoria cui risponde un miss), con $Miss_{rate} = 1-(Hit_{rate})$, si può avere un quadro rappresentativo delle prestazioni di una cache solo tenendo conto dei relativi tempi d'accesso.

Infatti viene speso un diverso tempo, per il trasferimento degli elementi, nel caso si verifichi un hit o un miss: lo Hit_{time} è il tempo di accesso alla cache in caso di successo (include il tempo per determinare se l'accesso si conclude con hit o miss) mentre la $Miss_{penalty}$ è il tempo necessario per sostituire un blocco in cache con un altro blocco dalla memoria di livello inferiore, il $Miss_{time}$ è calcolato come $Miss_{penalty} + Hit_{time}$ ed è il tempo necessario per ottenere l'elemento richiesto in

caso di miss.

Nello studio di predicibilità di sistemi con cache ha molta importanza anche la corrispondenza tra blocchi in memoria principale e blocchi in cache (Figura 2.8) oltre alle *politiche di sostituzione* (di aggiornamento) dei blocchi in caso di cache miss, caratteristiche entrambe fissate per il corrente lavoro di tesi.

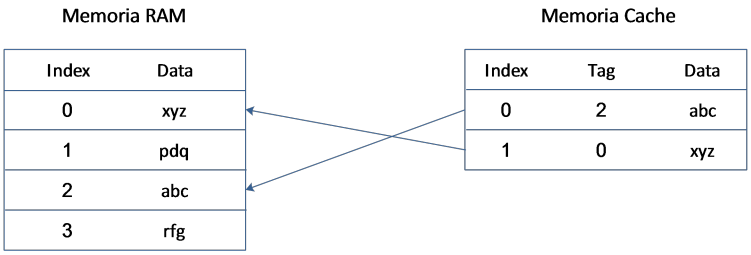

Figura 2.8: Confronto tra memoria Cache e memoria RAM.

Ogni blocco in memoria RAM ha un dato e un indice, cioè un indirizzo univoco utilizzato per riferirsi a quella specifica posizione. Ogni posizione nella cache ha un'etichetta che contiene l'indice in memoria principale del dato ivi caricato.

Si può andare dai due estremi *Direct Mapped* (in cui ogni blocco di memoria RAM trova il suo corrispondente in uno e un solo blocco in cache) e *Full Associative* (ogni blocco di RAM può essere mappato su un qualsiasi blocco di cache) passando attraverso le *set Associative N-vie*[8] (la cache è organizzata come insieme di set, ognuno dei quali include N blocchi)[9], con N potenza di 2. Esempi di corrispondenza tra blocchi in Memori RAM e blocchi in cache sono visibili in Figura 2.9.

La sostituzione dei blocchi può essere fatta in vari modi, i più comuni sono *Random* (i blocchi vengono sostituiti senza seguire una logica), *Least Recent Used* (LRU: come nella cache del LEON viene sostituito il blocco che è stato usato meno recentemente), *First In First Out* (FIFO: viene sostituito il blocco caricato meno recentemente come in una *coda*), *Last In First Out* (LIFO: viene sostituito l'ultimo blocco caricato come in una *pila*) e derivati.

[8]Con $N >= 2$; ovviamente se $N = 1$ si ha una cache direct-mapped.

[9]Anche la RAM è vista come organizzata in set, più numerosi dei set della cache; ogni set della RAM viene correlato a uno e un solo set della cache con una filosofia direct-mapped.

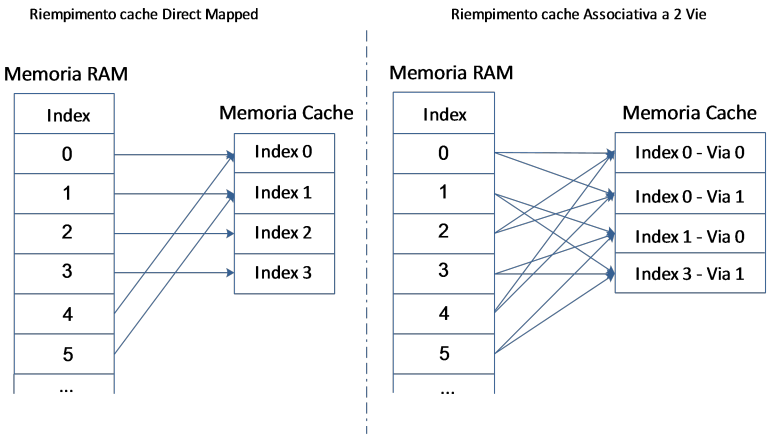

Figura 2.9: Confronto tra associatività Direct Mapped e set-associatività a 2 vie.

Capitolo 3

Analisi dello Stato dell'Arte

In questo capitolo sarà presentata una panoramica sullo stato dell'arte relativo al calcolo del *Worst Case Execution Time* e alla stima delle prestazioni dei sistemi dotati di memoria Cache.

Il problema di stima del WCET in presenza di cache estende le analisi del problema più generale di stima del WCET per l'architettura di Von Neumann[1].

Ogni approccio cerca di ridurre la complessità del problema introducendo delle limitazioni e fornendo quindi solo delle soluzioni parziali che rispondono a ben determinate condizioni iniziali. Risulta necessario combinare tra loro le varie soluzioni per rispondere esaustivamente agli interrogativi sulla materia di studio.

3.1 La Stima del WCET

Verrà innanzitutto introdotto il concetto di Worst Case Execution Time (WCET).

Il *Worst Case Execution Time* riferito a un task è il massimo tempo di esecuzione che il task impiegherebbe, al variare degli input, se eseguito su una determinata architettura. Conoscere il WCET diventa pertanto di primaria importanza per l'analisi di schedulabilità nei sistemi *Hard Real Time*[2].

[1] L'architettura di Von Neumann[29] è composta da quattro componenti fondamentali (CPU, memoria RAM, standard-Input e standard-Output) collegati al medesimo bus di interconnessione.

[2] Un sistema Hard Real Time è un sistema in cui il soddisfacimento dei vincoli temporali risulta critico. Questo non è vero nei sistemi Soft Real Time

La stima del WCET è oltremodo importante in uno scenario in cui i sistemi hard real time sono chiamati a soddisfare sempre più stringenti vincoli temporali e la determinazione di un *upper-bound* del tempo di esecuzione dei processi è sempre più difficile da ottenere. A causa della richiesta sempre maggiore di prestazioni negli anni si è introdotta una serie di componenti a livello microarchitetturale (quali cache, pipeline, etc.) all'interno della struttura base[3] del comune calcolatore.

Se lo stato dell'architettura studiata è perfettamente noto, il WCET è fissato; tuttavia, lo stato può essere solo parzialmente noto per sistemi in cui ai componenti di Von Neumann siano stati aggiunti elementi quali la cache, che rendono la stima del WCET più problematica a causa del parziale indeterminismo introdotto.

Questi componenti non assicurano, se non sotto certe condizioni, un comportamento sempre predicibile (particolare spazio verrà dato all'analisi del comportamento della memoria CACHE, descritta nella Sezione 2.5, essendo proprio questa al centro delle tematiche affrontate nello svolgimento di questo lavoro di tesi).

Considerando le prestazioni valutabili in termini di WCET e *tempo medio d'esecuzione*, se da un lato è indubbia la spinta che dispositivi citati abbiano apportato nel campo delle prestazioni, è reale il rischio che un sistema dotato di questi risenta di significative oscillazioni nel tempo di esecuzione a causa dell'impredicibilità introdotta.

3.1.1 Metodi analizzati

Una panoramica sui vari metodi e tool utilizzati finora per la stima del WCET è presentata da ***Wilhelm Reinhard*** *et al.* [31].

Non essendo possibile determinare facilmente l'upper-bound del tempo di esecuzione dei programmi (altrimenti si potrebbe risolvere il problema della terminazione, problema non decidibile) per ogni metodo vengono poste delle limitazioni al codice sorgente in modo da ridurne la complessità[4].

Per la stima del WCET bisogna analizzare il tempo d'esecuzione di ogni possibile *Execution Path*[5] che può essere percorso dal task. Se si conoscono gli input

[3]Composta da Unità di elaborazione dati, Memoria centrale e Disco rigido.

[4]Viene per esempio usata una forma ristretta di programma che permette di rendere tale problema decidibile, garantendo la terminazione.

[5]Il path d'esecuzione rappresenta il cammino d'esecuzione dei vari blocchi basici all'interno

e lo stato iniziale che porta all'esecuzione del task, il problema risulta di facile soluzione. Se invece non si hanno informazioni di questo tipo si procede sfruttando la rappresentazione tramite *Control Flow Graph* (descritta in Sezione 2.3.2) in modo da raggruppare i vari execution path in una unica struttura. Il problema della determinazione dei path è critico anche perché la maggior parte del tempo d'esecuzione dipende da cicli e funzioni ricorsive. Problematico è anche il fatto che se si eseguono due frammenti di codice A e B, ub_{A+B}[6] non è $ub_A + ub_B$ ma superiore. Considerare l'upper-bound in questo modo sui processori moderni porterebbe a ignorare il fatto che la stima dei tempi di B dipenda dallo stato iniziale che viene prodotto dall'esecuzione di A.

Le informazioni sul comportamento dei componenti come caches, memoria, pipeline e *branch predictor*[7] che influenzano il tempo di esecuzione sono utili per comprendere l'impredicibilità che questi componenti introducono. E' infatti il parziale non determinismo introdotto nel sistema che riduce l'applicabilità dei metodi di *timing-analysis* ai processori moderni[11]. L'assunzione che il solo *local worst case* basti a considerare l'upper-bound di un WCET globale risulta scorretta come del resto quella di credere di poter determinare uno *stato iniziale pessimo* corretto, senza sovrastimarlo.

La ricerca svolta produce due gruppi di metodi, classificati in base alle tecniche che usano, denominati *Static*[8], che dal codice generano i control-flow paths del task combinandoli con un modello astratto dell'architettura, e *Measurement-based*[9], che prendono i tempi misurati e derivano il massimo e il minimo tempo osservato oppure si servono dei frammenti di codice per formare un tempo globale del task.

del programma; dipende dalla struttura del programma stesso e in particolare dalla presenza di costrutti condizionali.

[6]Upper-bound della somma dell'esecuzione dei due frammenti di codice, prima A e immediatamente B.

[7]Speculazione sui salti.

[8]Questi metodi non si basano su l'esecuzione di codice su hardware reale.

[9]Questi altri eseguono il task o solo parte di esso su un determinato hardware o su un simulatore per alcune serie di input.

3.1.1.1 Modelli Calcolati Staticamente

I metodi **Static** come quello di *Wilhelm*[25] sono più sicuri visto che producono dei limiti sull'*execution-time* garantendo che il tempo d'esecuzione non superi questi limiti. L'analisi di tipo statica si articola in varie fasi chiamate *Value analysis*, *Control-flow analysis*, *Processor-behaviour analysis* e *Estimate calculation*.

Ogni metodo statico di analisi del comportamento della cache ha bisogno di conoscere gli effettivi indirizzi dei dati in memoria in modo da capire come avvengono gli accessi a questi ultimi.

Value Analysis

Nella *Value Analysis* il calcolo degli indirizzi avviene calcolando i *range* dei valori possibili sia dei registri del processore che delle variabili locali in ogni punto del programma; quindi necessariamente a *runtime*. Tuttavia la value analysis è implementata in tools quali aiT, Bound-T e SWEET.

- AiT[13] è un tool per ottere gli upper-bound dei tempi di esecuzione di porzioni di codice (subroutines) in modo da evitare sovrastime del WCET. Si serve di una combinazione tra un'astrazione basata su livelli e il *pattern matching* (che verrà descritto in seguito) lavorando sul codice macchina. Il tool non può essere usato universalmente in quanto dipende da annotazioni esterne che specifichino i valori di registri e variabili.

- Bound-T[18] determina come aiT l'upper-bound del tempo di esecuzione delle subroutines; opzionalmente è possibile determinare un upper-bound dell'uso dello stack.

  Bound-T soffre di alcune limitazioni come non elaborare programmi che al loro interno abbiano ricorsioni o in cui il CFG non sia riducibile e non analizzare i costrutti dinamici in cui ci siano molteplici indirizzi di destinazione. Calcola i limiti dei cicli modellizzando la computazione istruzione per istruzione e servendosi di equazioni affini e disuguaglianze (aritmetica di Presburger[10]). Inoltre il task analizzato deve usare la convenzione standard per le chiamate

[10] E' una forma semplice di aritmetica che ammette esclusivamente le operazioni di somma e uguaglianza.

a funzione, i puntatori a funzione non sono supportati e la variabile di induzione di un ciclo non può essere modificata da operazioni di moltiplicazione (ad eccezione di una costante), divisione e logiche come and, or, shift e rotate. Non è presente nemmeno un'implementazione per l'analisi della cache.

- SWEET (SWEdish Execution-Time Tool) è costruito in modo modulare sviluppando l'analisi nei tre passi di *Flow Analysis*, *Processor-Behaviour Analysis* e *estimate calculation*. Il tool permette di fare una analisi automatica del codice a livello intermedio legando tra loro Flow Analysis e Processor-Behaviour Analysis. Il tool può essere usato per processori RISC di medio livello di complessità e presenta una varietà di metodi che determinano l'upper-bound in base all'analisi del flusso e all'analisi della pipeline. SWEET è integrato con un compilatore e può dunque effettuare l'analisi a livello del codice intermedio (IC[11]) del compilatore stesso.

  L'analisi di SWEET sfrutta una rappresentazione simbolica per stimare i valori delle variabili in punti differenti del programma. Questa analisi è capace di calcolare automaticamente sia i cicli che le informazioni sui path non possibili.

  Inizialmente si effettua la *memory-access analysis*[12], che produce un set di annotazioni che informano sulle aree di memoria che saranno referenziate dall'istruzione o se l'istruzione produrrà una miss/hit in cache. Successivamente viene usata una *pipeline analysis*, che utilizza i dati della fase precedente e sfruttando un modello CPU *cycle-accurate* effettua lo studio del WCET.

  La flow analysis può processare programmi che includano puntatori e siano ANSI-C[13], codice destrutturato e funzioni ricorsive. Tuttavia per poter utilizzare la flow analysis automatica è necessario compilare il programma con il compilatore integrato in SWEET, in caso contrario bisogna introdur-

[11] I risultati provenienti dall'analisi del Codice Intermedio sono simili e quindi interscambiabili con quelli prodotti dalla Flow Analysis

[12] Nel caso in cui comparisse una istruzione che riguardi la cache verrebbe performata una I-cache analysis.

[13] ANSI C è lo standard pubblicato dalla American National Standards Institute (ANSI) per il linguaggio di programmazione C.

re manualmente le annotazioni sul flusso d'esecuzione. Non è supportata l'allocazione dinamica della memoria.

La memory-access analysis non gestisce le D-cache, è possibile analizzare un singolo livello di I-cache e effettuare la stima solo su determinati tipi di pipeline[14]. L'analisi sui path richiede inoltre che il codice sia ben strutturato.

Control-Flow Analysis

Nella *Control-Flow Analysis* (CFA) l'obiettivo è ottenere informazioni sul set finito di possibili paths di esecuzione[15]; questo set deve dunque essere il più piccolo possibile. I risultati prodotti da questa fase possono essere interpretati come vincoli sul comportamento dinamico del task dal momento che includono informazioni su quali funzioni possono essere chiamate e sulla percorribilità dei path. Risulta più semplice effettuare la CFA sul codice sorgente piuttosto che sul linguaggio macchina in quanto la compilazioni potrebbe modificare la struttura del flusso.

Gustafsson[9] invece usa una combinazione di più metodi: il *Pattern Matching*[16] per i cicli semplici e una interpretazione astratta[8] a livello intermedio del codice (IC) per i cicli più complessi.

Il pattern matching si serve dei risultati della flow analysis e calcola il numero di esecuzione dei gruppi di istruzioni. L'obiettivo è trovare il dominio dei possibili valori del contatore (il valore iniziale, incremento/decremento e valore finale).

Un inconveniente nell'uso di questo approccio è quello di non poter aggiornare o sostituire il compilatore, poiché potrebbe cambiare il pattern di istruzioni causando il fallimento del matching.

Due esempi che sfruttano la data-flow analysis sono:

- l'approccio di ***Christopher Healy*** *et al.*[10] che, in combinazione ad algoritmi speciali e un compilatore, calcola i limiti di cicli semplici e innestati;

[14]Sono gestite solo le *in-oreder pipeline* mentre le *out-of-order pipeline* non lo sono.

[15]Il numero di paths di esecuzione è necessariamente finito in quanto deve essere garantita la terminazione; è comunque molto difficile se non impossibile creare un set di soli paths possibili, tipicamente si crea un "superset" che sia una buona approssimazione che contenga anche paths non percorribili.

[16]Questa tecnica nasce dall'osservazione che il compilatore usa sempre lo stesso gruppo di istruzioni macchina per inizializzare, aggiornare e testare il contatore del ciclo.

- il lavoro di ***Friedhelm Stappert*** et al.[30] che sfrutta l'esecuzione simbolica sul codice sorgente per derivare le informazioni di flusso.

Il tempo di esecuzione di ogni singola istruzione (anche degli accessi alla memoria) dipende da ciò che è avvenuto nel recente passato, quindi per ottenere una giusta stima dell'esecuzione è necessario analizzare lo stato di occupazione dei componenti del processore per tutti i path che sono interessati dalle istruzioni del task.

Processor-behaviour Analysis

La *Processor-behaviour Analysis* determina ciò che non viene modificato nei vari stati dei componenti che influenzano maggiormente il tempo d'esecuzione come cache, memoria, pipeline e speculazione sui salti. Una analisi di questo tipo è necessaria per qualsiasi tool e si basa su un modello astratto del processore (con memoria del sottosistema, bus e periferiche) che è conservativo rispetto al comportamento temporale dell'hardware reale[17]. Le difficoltà che si possono incontrare nella stima con questo tipo di analisi dipendono fortemente dal tipo di processore analizzato e dal suo stato iniziale in quanto diversi stati iniziali possono portare a una significativa variazione dei tempi di esecuzione anche senza modificare gli input[31].

Estimate Calculation

La fase di *Estimate Calculation* calcola un upper-bound globale di tutti i tempi d'esecuzione dell'intero task basandosi sulle informazioni di flusso e di tempo derivate nelle fasi precedenti.

Secondo ***Wilhelm Reinard***[31] ci sono essenzialmente tre classi di metodi che combinano diverse stime dei tempi determinate analiticamente o misurate direttamente: *Structure-Based*, *Path Based* e *Implicit-Path Enumeration*.

- ***Structure-based***: L'upper-bound viene calcolato scorrendo il *Control Dependence Graph* (CDG)[4] dal basso verso l'alto e combinando i vari tempi

[17] Il modello astratto non stima mai un tempo di esecuzione inferiore a quello che si osserverebbe su un processore reale.

d'esecuzione. Serie di nodi vengono collassati in nodi unici e contemporaneamente vengono calcolati i nuovi tempi d'esecuzione dei macronodi.

Purtroppo non è possibile esprimere ogni flusso di controllo attraverso un albero sintattico e questo rende questa tecnica vincolata al codice sorgente che per questo motivo non può a sua volta essere ottimizzato; inoltre non risulta possibile fornire manualmente delle informazioni addizionali nemmeno sotto forma di annotazioni[31].

- ***Path-based***: L'upper-bound totale di un task è ottenuto calcolando i tempi dei diversi path all'interno del task, cercando il path che ha il tempo d'esecuzione massimo. Un punto fondamentale è che i path di esecuzione siano esplicitati.

  Il metodo va bene per cicli singoli ma non per cicli con più livelli d'annidamento. Inoltre il numero dei path è esponenziale nel numero dei punti di diramazione (branch)[31].

- ***Implicit-Path Enumeration (*IPET*)***: Nell'analisi IPET il flusso del programma e le stime dei tempi d'esecuzione dei blocchi basici sono combinati in set di vincoli aritmetici. E' una tecnica molto usata in cui ai blocchi basici e ai flussi viene assegnato, ogni volta che il blocco basico è incontrato nell'esecuzione, un coefficiente temporale *t-entity* (che esprime il contributo in tempo di quella entità sul tempo totale d'esecuzione) e una variabile contatore *x-entity* che conta le volte che l'entità viene eseguita. La somma del prodotto delle 2 variabili, soggetta alle limitazioni derivanti dalla struttura del task e dai flussi possibili, è una una stima dell'upper-bound. Alla fine si ha una stima di upper-bound e un worst-case count per ogni x-entity. Non è necessario che i path siano esplicitati. L'approccio appena descritto usa PLI (*Programmazione Lineare*[18]) o CP (*Constraint Programming*[19]) soffrendo di una complessità potenzialmente esponenziale nella dimensione[20] del task.

[18]Programmazione lineare per la risoluzione di problemi lineari interi

[19]La programmazione a vincoli (CP) è un paradigma di programmazione in cui le relazioni tra le variabili sono espresse in forma di vincoli.

[20]Le variabili da calcolare aumentano in relazione al numero di annotazioni sui flussi che vengono convertite in vincoli.

Symbolic Simulation

La *Symbolic Simulation* è un metodo statico che, simulando l'esecuzione del task in un modello astratto del processore che non riceve input, deve tener conto del semi-sconosciuto stato d'esecuzione. Vengono combinate Flow Analysis, Processor-behaviour Analysis e Bound Calculation.

Tuttavia la simulazione simbolica risulta ordini di grandezza più lenta rispetto agli altri metodi e l'analisi risulta per questo molto più lunga.

3.1.1.2 Modelli basati sulla Misurazione

I modelli **Measurement-Based** eseguono il task su un hardware o un simulatore per un determinato set di input e misurano il tempo totale di esecuzione o parte di questo. Se l'input del caso pessimo è conosciuto basta una sola esecuzione ma se non è noto allora l'output può solo fornire una stima in termini statistici. Si può anche misurare il tempo di esecuzione dei blocchi basici del CFG, successivamente combinare tra loro i tempi misurati tenendo conto di alcuni vincoli e procedere al calcolo del WCET. Queste misurazioni si sostituiscono alla Processor-behaviour Analysis utilizzata dai metodi statici. Il problema della stima del tempo d'esecuzione dei path è affrontato tramite le CFA come nell'approccio statico ma può produrre risultati inconsistenti nel caso sia erronea la stima del tempo d'esecuzione dei blocchi basici.

Un tool significativo che usa metodi Measurement Based è *RapiTime*.

- ***RapiTime***: RapiTime è un tool che deriva le informazioni necessarie all'analisi dal tempo di esecuzione del path più lungo di una porzione di codice (generalmente un blocco basico). Insieme al WCET viene fornita anche la distribuzione di probabilità del tempo di esecuzione totale dei path più lunghi attraverso il programma completo. Il tool non si basa su un modello di processore ma può, almeno per via teorica, modellizzarli tutti.

  Il problema consiste nell'estrarre le tracce di esecuzione dal sistema; non c'è supporto a questa attività che richiede un codice di instrumentazione aggiuntivo; non si possono analizzare programmi che presentano funzioni ricorsive o che facciano uso di puntatori e nemmeno programmi che non possano essere analizzati in modo statico.

Un altro problema è che viene testata solo una parte dei possibili contesti del processore per la stima dei blocchi basici. Aumentando il numero di contesti non è detto che si elimini il rischio di non analizzare un contesto significativo, per questo è preferibile avere a disposizione il contesto del worst case (lo stato iniziale del processore all'occorrenza del worst case) calcolabile attraverso analizzatori logici hardware o software che non siano intrusivi. I metodi di questo tipo sono utili per verificare l'effettiva ripetibilità dei tempi di esecuzione del task ma non possono comunque fornire upper-bound corretti molto inferiori agli upper-bound calcolati tramite metodi statici che non siano fallaci.

Entrambe le classi Static e Measurement-Based condividono problemi e soluzioni e possono collaborare tra loro nel tentativo di completarsi a vicenda. Il Front End è simile quando tutte e due le classi di metodi usano codice eseguibile come input; la Control-Flow Analysis è simile anch'essa, come lo è il calcolo delle stime dei tempi d'esecuzione. IPET per esempio è composto da alcuni metodi statici e altri basati sulla misurazione. Il problema principale dei metodi Static è la modellizzazione del comportamento del processore mentre per i Measurement-Based è complesso effettuare una stima accurata dei tempi a livelli fini di granularità senza che il programma sia perturbato da ciò che effettua la misura. La seconda classe inoltre soffre del problema del limitato set dei contesti iniziali del processore che rende il metodo poco accurato.

3.1.2 Altri Metodi

Secondo ***Bach Khoa Huynh*** *et al.* [12] l'analisi statica del WCET in sistemi non deterministici non è sufficiente poichè porterebbe a una inevitabile sovrastima.

E' necessario comprendere le fonti di impredicibilità a livello del codice sorgente e trasformare il programma in uno che sia facilmente analizzabile. Viene considerato estremamente complesso il tentativo di trasformare un programma già esistente, che sia *I/O-efficiente*[21], in una versione più predicibile su architetture dotate di D-cache. Viene quindi privilegiata la strada che prevede l'emulazione di una D-cache e l'abbandono del vincolo della *I/O-efficienza*.

[21] Sono programmi che minimizzano le operazioni I/O.

L'obiettivo è dunque quello di ottenere un programma che abbia solo una minima variazione in termini di tempo di esecuzione per molteplici input e che sia facile da analizzare attraverso tecniche di analisi statica. La sovrastima del tempo d'esecuzione ha portato alla creazione di strutture dati software denominate *cache buffer* usate per la persistenza in cache delle porzioni di codice necessarie più spesso (tecnica simile al cache Locking). La procedura di trasformazione del codice in una versione più adatta alla presenza di una D-cache può anche avvenire con più iterazioni seguendo un approccio *multilivello* nel tentativo di ridurre gli errori di stima presenti nell'analisi di persistenza di livello singolo.

L'analisi svolta su un sistema dotato di cache buffer riduce complessivamente l'errore di stima del WCET e la differenza in termini di tempo di esecuzione per input eterogenei.

Per ***Kim Sung-Kwan*** *et al.* [16] l'ottimizzazione delle operazioni di Load e Store avrebbe un ruolo fondamentale e quest'aspetto sarebbe stato valutato in modo troppo conservativo dagli altri approcci.

La proposta si presenta sotto forma di due tecniche: la prima si serve di una analisi del flusso d'esecuzione globale del programma per ridurre il numero di istruzioni L/S che vengono mal classificate come *Dynamic-L/S instruction*[22] mentre la seconda consiste nell'usare un'analisi sulle dipendenze dei dati per minimizzare l'influenza negativa delle istruzioni di L/S.

Il lavoro si pone come punto di partenza uno schema dei tempi più esteso rispetto a quello originale (dove l'informazione temporale associata ad ogni programma era un semplice tempo limite per la sua esecuzione). Quest'approccio tiene conto di operazioni di *concatenamento* e *pruning* (taglio) del WCTA[23][17] che permisero di ottenere una significativa diminuzione, nell'ordine del 50%, dell'errore di sovrastima nel calcolo del WCET.

I risultati sperimentali consentono di affermare che le due tecniche adottate

[22]Con questo termine sono state definite le istruzioni di L/S che si riferiscono a zone di memoria multiple come quelle utilizzate per implementare array o zone di memoria indicizzate da puntatori.

[23]Per WCTA si intende *Worst Case Time Abstraction*: contiene molte informazioni inutili a proposito dei vari percorsi di esecuzione che potrebbero essere il *Worst Case Execution Path*; Il Worst Case Execution Path a sua volta è il percorso dal quale si ottiene il WCET del programma.

permettono di aumentare la precisione nella determinazione del WCET in modo considerevole. Tuttavia un netto miglioramento è ottenibile soprattutto per i programmi che fanno un largo uso di array, che sono la fonte maggiore di istruzioni Dynamic-L/S.

Secondo ***Martin Schoeberl***[27] ha particolare importanza capire quali siano le caratteristiche ideali che rendono predicibile il comportamento di una data architettura.

Questo lavoro parte identificando i punti chiave d'incertezza quali il disallineamento tra necessità di prestazioni e analizzabilità del WCET oltre all'aumento di complessità dei modelli come conseguenza all'aumento di complessità delle caratteristiche architetturali. Viene anche discussa la diversa analizzabilità della I-cache e della D-cache. Poiché risulta difficile analizzare in modo statico la cache e in particolare la D-cache, vengono presi in considerazione tipi di cache particolari che permettono, conservando i benefici in prestazione, di fornire un WCET più preciso. Le architetture sono successivamente valutate da JOP[26], una implementazione di processore Java che permette di avere informazioni utili sulle varie configurazioni della cache quali la misura dei cache-miss e la miss-penalty (che dipende dalla memoria centrale del sistema).

Non è sempre facile scegliere tra una architettura più lenta ma predicibile e una architettura più veloce ma impredicibile; in questo lavoro si afferma che una soluzione predicibile sarà sempre mediamente più lenta di una soluzione meno predicibile ma ottimizzata per il caso medio.

Ghosh Somnath *et al.* [7] sostengono che per arginare il crescente *gap* di prestazioni tra CPU e Memoria Centrale una cache può rappresentare una buona soluzione soprattutto per quei programmi che manifestano una buona località (Sezione 4.2).

Sia l'ottimizzazione manuale che quella automatica servendosi di un compilatore riescono a ottenere risultati sub-ottimi a causa dell'impredicibilità dei conflitti in cache.

L'obiettivo è trovare precise relazioni tra gli indici dei cicli e analizzare come le dimensioni della cache possono influenzare il numero di *cache miss* nei cicli

annidati.

Il metodo propone l'uso delle Cache Miss Equations (CMEs) che permettono una rappresentazione dettagliata del comportamento della cache e dei conflitti, soprattutto per codice nella cui struttura siano presenti cicli. Viene sfruttato il compilatore SUIF[32] e vengono generate equazioni *Diofantine*[24] che descrivono l'uso della memoria da parte dei cicli.

Risolvere equazioni di questo tipo è complesso, ma non sempre necessario in quanto vanno analizzate solo in determinati e relativamente semplici contesti d'esecuzione. La maggior parte dei cicli innestati analizzati ha un accesso ai dati predicibile e regolare.

Dall'analisi dei conflitti risulta che la maggior parte dei miss avviene o a causa delle dimensioni del ciclo (*capacity miss*), che eccede le dimensioni del blocco della cache, o a causa di conflitti dovuti alla scarsa associatività della cache (*conflict miss*).

Gli autori forniscono un framework per supportare un compilatore nella trasformazione del codice e migliorare la stima del WCET.

Le condizioni che rendono efficaci i risultati prodotti con le CMEs sono però molteplici e rendendo analizzabile solo il 70% dei cicli appartenenti ai benchmarks SPECfp[25]. Tutti i cicli devono essere normalizzati in modo che il valore del passo sia uno[3], i cicli non possono contenere espressioni condizionali e vengono considerati solo cicli perfettamente annidati che contengono al massimo un blocco basico oltre al ciclo annidato. Altre limitazioni riguardano i tipi di *array* ammessi e il comportamento delle istruzioni di Load e Store all'interno di cicli annidati.

Le CME sono generate staticamente a tempo di compilazione e visto che devono analizzare i possibili conflitti di cache vanno arricchite con la posizione delle strutture dati. In generale viene creato un parametro per ogni variabile che dipende da informazioni disponibili solo *runtime*. Le variabili così generate sono calcolate dinamicamente a tempo d'esecuzione.

[24] In matematica, un'equazione *diofantea* (chiamata anche equazione diofantina) è un'equazione in una o più incognite con coefficienti interi di cui si ricercano le soluzioni intere.

[25] SPECfp è una suite di benchmark per computer studiato per testare le performance floating point di un compilatore.

Infine il lavoro di ***Enrico Mezzetti***[19] fornisce una serie di raccomandazioni e di buone consuetudini da osservare per minimizzare l'impredicibilità della cache. Si fa particolare attenzione soprattutto alla I-cache visto che le politiche di scrittura e lettura della D-cache la rendono troppo complessa e fortemente dipendente dall'applicazione (*application-specific dependent)* per poter essere analizzata. Questo lavoro è quello che fra tutti si mette nelle condizioni più simili a quelle del presente lavoro di tesi. Gli esperimenti sono svolti sul processore LEON AT697E servendosi di una I-cache da 32 KByte di memoria con associatività a 4 vie e una politica di sostituzione dei blocchi in cache di tipo Least Recent Used (LRU). La D-cache è disabilitata e lo stato della I-cache viene bloccato durante la gestione degli interrupt, durante l'esecuzione di specifici task e durante le chiamate del *Kernel*[26]. La traccia di esecuzione, necessaria per l'utilizzo del tool RapiTime, nella configurazione impiegata ha un'accuratezza del 95%.

L'uso di metodi *Statici* viene valutato sicuro ma troppo impreciso in quanto tendono a sovrastimare il WCET analizzando anche i path non percorribili e aspettandosi un numero troppo pessimistico di cache miss.

I metodi *basati sulla Misurazione* sono considerati migliori anche se risulta complesso e non sempre possibile individuare uno stato iniziale pessimo per il processore. Visto che l'analisi Measurement-Based parte dai blocchi basici e poi cerca di assemblare i blocchi stessi nel tentativo di ottenere il WCET, è possibile che vengano considerati dei path non possibili come capita per i metodi Static.

Per ottenere un sistema predicibile bisogna quindi limitare il comportamento della cache in modo da ridurre la sovrastima dei metodi Statici e limitare la complessità computazionale del caso pessimo iniziale dei metodi Measurement-based.

L'articolo promuove l'utilizzo di pratiche consolidate che permettono di aumentare considerevolmente la predicibilità della cache:

- Separare I-cache e D-cache. Una cache unificata è più difficile da analizzare rispetto a una che presenti questa separazione. Tutti i processori LEON hanno I/D-cache separate come separati sono i controller delle cache. E' preferibile

[26]Il Kernel costituisce il nucleo di un sistema operativo. Si tratta di un software avente il compito di fornire ai processi in esecuzione un accesso sicuro e controllato all'hardware. Il Kernel ha anche la responsabilità di gestire l'accesso concorrenziale all'hardware di ciascun programma.

una disabilitazione totale della D-cache quando si analizza il comportamento della I-cache per evitare delle interferenze indirette tra dati e istruzioni.

- Sfruttare una politica di sostituzione LRU. E' sperimentalmente la più analizzabile[14] ed è quella utilizzata da LEON AT697E.

- Prediligere cache set-associative a 2 o 4 vie. La cache direct-mapped è soggetta a troppi conflitti e quella full-associative è troppo impredicibile. LEON AT697E usa una I-cache a 4 vie. La scelta di tenere bassa l'associatività è dovuta al fatto che la politica LRU diventa complessa da implementare su una cache con associatività maggiore di 4[15].

- Inibire il comportamento della cache durante gli interrupt[27]. Una modifica di questo tipo permette di incrementare il determinismo senza penalizzare le performance. La I-cache di LEON può gestire l'occorrenza degli interrupt in questo modo.

A queste consuetudini si aggiungono le pratiche suggerite da questo lavoro con particolare riferimento alla famiglia di processori LEON:

- Partizionamento della Cache. Realizzabile sia via hardware[22] che via software[21] rappresenta il metodo più semplice per diminuire l'impredicibilità della cache. Consiste nel partizionare la cache in modo da garantire a ogni task o a ogni funzione la sua propria porzione di cache. Per questo genere di operazione è necessaria la presenza di un compilatore e di *linker*[28] che li supportino. L'assegnamento di una porzione di cache favorisce in particolare i task che sfruttano meglio la località spaziale della cache.

- Cache Locking . Si realizza via hardware e consiste nel controllare i contenuti della cache.
  Con questo metodo si può ignorare la politica di sostituzione della cache in

[27] Quando un task riprende la sua esecuzione dopo la *preemption* (l'atto di interrompere un task temporaneamente) troverà lo stato della cache diverso da come'era al momento della preemption, questi cambiamenti sono dovuti al gestore degli interrupt e agli altri task che sono stati eseguiti nel mentre.

[28] Un Linker è un programma che prende in input uno o più oggetti generati da un compilatore e li combina in un unico programma eseguibile.

modo da non cancellare le istruzioni che occorrono più spesso aumentando i numero di cache hit. Infatti se il contenuto della cache fosse fissato e calcolato, il comportamento della cache e degli accessi diventerebbe prevedibile in modo soddisfacente. Il cache locking si divide in *Static-locking*[23] e *Dynamic-locking*[5] a seconda che sia ottenuto con tecniche statiche o dinamiche.
Col metodo Static i dati vengono caricati nella cache all'avvio del sistema e vi rimangono all'interno fino al termine dell'esecuzione. Nei metodi dinamici invece il contenuto della cache viene aggiornato a determinati punti dell'esecuzione. I metodi di Dynamic-locking possono essere usati: a livello system, con supporto dei metodi di Static-locking per la gestione di task specifici, oppure a livello task, su determinate sezioni di codice di quest'ultimo.
Lo Static-locking porta, all'aumentare delle dimensioni del codice, a una diminuzione delle dimensioni della cache utilizzabile liberamente e quindi a un probabile aumento, anche se predicibile, dei tempi di esecuzione.
Il Dynamic-Locking invece causa un significativo aumento dell'*overhead* dovuto al cambio di contesto, overhead che comunque risulta costante del numero dei task e gestibile con lo scheduling. Entrambe le politiche di cache locking eliminano indeterminismo a costo di prestazioni.

- Sostituzione della cache. All'utilizzo della cache si predilige lo sfruttamento di una memoria dalle caratteristiche simili (piccole memorie on-chip, veloci e in cui sussiste la divisione tra istruzioni e dati) alla cache ma in cui i contenuti sono allocati in uno spazio di indirizzi separato e deterministico.
  Le *Scratchpad memoriy* sono gestite a livello software dall'utente o dal compilatore. Il programma viene diviso in piccole parti collegate tra loro da istruzioni *call*. il problema di questo approccio è la complessità.

- Evitare branch-prediction e speculative-execution. L'uso di questi due metodi può portare infatti all'aumento di anomalie temporali e quindi all'indeterminismo[24]. Il processore LEON non ammette nessuno dei due metodi. Può essere usata la predizione dei salti statica che permette un aumento di prestazioni e non apporta indeterminismo.

- Adottare uno stile di scrittura del codice che sia sensibile alle problematiche della cache e cercare di bilanciare le proprie necessità di prestazioni e predicibilità, ma questo dipende essenzialmente dall'ambito di utilizzo.

L'equilibrio tra prestazioni e determinismo risulta instabile anche per quei programmi che, considerati sicuri, vengono modificati solo superficialmente.

In conclusione si può affermare che se da un lato è stato fatto molto per la ricerca di quali siano i fattori che influenzano l'impredicibilità di un sistema dotato di cache, dall'altro si è ancora lontani da una soluzione univoca e non dipendente dai singolari casi d'analisi.

Capitolo 4

Approccio Metodologico

In questo Capitolo è descritta la metodologia proposta nel presente lavoro di tesi.

Le memorie cache, introdotte all'interno dell'architettura LEON per migliorare le prestazioni, possono avere un costo significativo in termini di variabilità del tempo di esecuzione a causa dei diversi tempi richiesti da un cache *hit* e un cache *miss*.

Nella Sezione 4.1 verranno presentate le assunzioni iniziali sulle quali si fonda la presente metodologia e le cause principali di impredicibilità del tempo di esecuzione.

Nella Sezione 4.2 verrà presentata la prima causa di impredicibilità, dovuta al *layout* del codice applicativo.

Nella Sezione 4.3 verrà presentata la seconda causa di impredicibilità, dovuta alle strutture di controllo all'interno dell'applicazione.

Nella Sezione 4.4 verrà presentata la terza causa di impredicibilità, dovuta alle dimensioni della cache.

Nella Sezione 4.5 verrà presentata la quarta causa di impredicibilità, dovuta alla concorrenza tra processi in esecuzione.

4.1 Assunzioni base e Cause di Impredicibilità

Dai risultati prodotti dal progetto PEAL[1] sono emerse alcune considerazioni che hanno portato all'uso delle seguenti assunzioni:

- sono considerati solo Linker[1] standard dei compilatori: non è dunque possibile controllare il layout della memoria;
- la struttura di ogni task può essere astratta come mostrato nel Codice 4.1:

```
while(1)
{
    /* attendi attivazione */
    /* esegui funzionalità*/
}
```

Codice 4.1: Struttura base del task

  in particolare l'analisi di predicibilità è focalizzata sulla parte del task che esegue la funzionalità;
- quando un task è pronto per riprendere l'esecuzione, la I-cache non ha memoria del precedente stato d'attivazione del task;
- il termine dell'esecuzione può fare riferimento solo all'attivazione di un unico task completo e non ad una singola parte del task.
- la cache è inibita durante la gestione degli interrupt.

Sono state individuate quattro possibili cause di impredicibilità del tempo di esecuzione dell'applicazione:

1) **layout**: la disposizione del codice e dei dati all'interno della memoria;
2) **sequenziale**: la storia e quindi l'ordine della sequenza d'accesso alle istruzioni del codice e ai dati;
3) **dimensione**: il rapporto tra le dimensioni della cache e la dimensione di istruzioni e dati;

[1]Il Linker è un programma che prende uno o più oggetti provenienti da un compilatore e li combina tra loro ottenendo un programma eseguibile.

4) **concorrenza**: il modo in cui interrupt e preemption interrompono e riprendono l'esecuzione del task in esecuzione;

L'analisi si concentra in particolare su quali siano i blocchi di codice che possono stimolare queste cause di impredicibilità e che sono quindi in grado di produrre un significativo aumento della variabilità del tempo di esecuzione.

Le sequenze di codice critiche per questo tipo di analisi possono essere classificate in:

- sequenza *cache killer*: sequenza che rende critica la predicibilità del tempo di esecuzione del task.
- sequenza *cache risk*: sequenza che, nel caso si cambiassero alcune delle condizioni correnti, potrebbe diventare una sequenza cache killer.

Data una sequenza cache killer bisogna valutare quanta impredicibilità questa introduce. Per le sequenze cache risk la valutazione si divide in probabilità che la sequenza stessa diventi di tipo cache killer e valutazione di quanta impredicibilità questa sequenza *cache killer* introdurrebbe.

4.2 Impredicibilità dovuta al Layout

Il layout di istruzioni e dati può incidere in modo significativo, durante l'esecuzione, sul numero di cache hit e cache miss. Per il principio di *Località Temporale*[2] quando si accede a un indirizzo usato recentemente ci si aspetta di avere un cache hit e in caso contrario un cache miss.

Tuttavia il comportamento della cache può provocare un andamento contrario a quello previsto in due situazioni: quando la *Località Spaziale*[3] (che dipende da dove il Linker posiziona il codice) inibisce quella temporale provocando un hit al posto di un miss oppure quando la *Località Temporale* inibisce quella spaziale

[2]Quando si fa riferimento a una posizione di memoria, con alta probabilità si farà di nuovo riferimento alla stessa posizione entro breve tempo (caso tipico: ripetuto accesso alle istruzioni del corpo di un ciclo).

[3]Quando si fa riferimento a una posizione di memoria, con alta probabilità si farà entro breve tempo riferimento a posizioni vicine (es.: istruzioni in sequenza; dati organizzati in vettori o matrici e a cui si accede sequenzialmente, etc.).

causando un miss al posto di un hit.

Un esempio di impredicibilità di tipo **layout** è rappresentato dal Codice 4.2; viene rappresentato un codice che definisce un ciclo che all'interno del suo corpo ha cinque chiamate a funzione in sequenza. La situazione è problematica nel caso in cui il codice descritto sia eseguito su una architettura dotata di cache con associatività inferiore al numero di chiamate a funzione presenti nel ciclo.

```
void p()
{
    while(1)
    {
        p1();
        p2();
        p3();
        p4();
        p5();
    }
}
```

Codice 4.2: Esempio di ciclo con cinque chiamate a funzione.

Dal momento che il Linker agisce autonomamente è possibile che i cinque blocchi delle cinque procedure siano mappati nello stesso set della I-cache.

Si supponga che questo conflitto occorra per tutti i blocchi delle funzioni. Si consideri inoltre che i blocchi di P1, P2, P3 e P4 occupino ognuno e in ordine i quattro blocchi del set della cache causando il seguente comportamento: durante la prima esecuzione della funzione P5 ci sarà un miss dovuto al tentativo di accesso al primo blocco (occupato dai blocchi appartenenti alle altre funzioni) e così anche per ogni accesso agli altri blocchi, dunque un cache miss, causando la rimozione dalla cache del blocco della funzione P1[4]. Durante l'iterazione successiva quando viene lanciata nuovamente P1 non troverà più il blocco caricato in cache precedentemente

[4]Il comportamento descritto cambia al variare della politica di sostituzione dei blocchi in cache (nell'esempio è stata adotta una politica First In First Out).

in quanto è stato rimosso per consentire l'allocazione del blocco di P5 e si verificherà un nuovo ciclo di miss che porterà alla rimozione di P2. Questo comportamento si ripete al susseguirsi del lancio delle funzioni causando nel peggior caso il verificarsi di un cache miss per ogni blocco richiesto da una funzione. Un comportamento di questo tipo indica che la porzione il Codice 4.2 rappresenta una sequenza cache killer.

Per poter individuare una sequenza cache killer è necessario che una metodologia:

- possa identificare gli I-cache miss come una causa di aumento del tempo di esecuzione
- possa individuare che questi miss avvengono durante l'esecuzione di un ciclo particolare e siano dovuti a conflitti nella cache;
- sia in grado di dire che i blocchi della cache in conflitto appartengano a diverse *unità di traduzione*[5].

4.3 Impredicibilità Sequenziale

Le strutture di controllo (come per esempio *if*, *switch* e *while*) presenti all'interno dell'applicazione possono produrre path d'esecuzione eterogenei.

Dal momento che gli stati della I/D-cache dipendono dalla storia d'esecuzione dell'applicazione, in un determinato punto, che sia raggiungibile da più path d'esecuzione, si possono avere diversi stati della cache che possono causare un diverso tempo di completamento dell'applicazione stessa.

Un esempio di impredicibilità di tipo **sequenziale** può essere generato modificando l'esempio di Codice 4.2 come in Codice 4.3.

[5]Una unità di traduzione è l'input finale per un compilatore C dal quale viene generato un file oggetto. Un programma C è costituito da unità chiamate file sorgenti (o file di pre-elaborazione). Quando il preprocessore C espande un file sorgente con tutti i file header dichiarati dalle direttive ♯*include*, il risultato è una unità di traduzione di pre-elaborazione. Il preprocessore traduce poi l'unità di pre-elaborazione in un'unità di traduzione. Da una unità di traduzione, il compilatore genera un file oggetto, che può essere ulteriormente elaborato e collegato (anche con file di altri oggetti) per formare un programma eseguibile.

L'esempio riportato richiama il codice precedente dunque il layout della memoria sarà il medesimo per entrambi i casi.

```
void p()
{
    while(1)
    {
        p1();
        p2();
        if(< condizione >) /*si verifica di rado*/
            p3();
        p4();
        p5();
    }
}
```

Codice 4.3: Esempio di ciclo con quattro/cinque chiamate a funzione.

Il codice proposto rappresenta dunque una sequenza cache risk in quanto se la condizione è falsa il ciclo è equivalente a un ciclo con quattro chiamate a funzione (non si verifica un problema di tipo layout), altrimenti il ciclo è equivalente a quello in Codice 4.2 con cinque chiamate a funzione.

L'identificazione di sequenze di questo tipo può essere fatta in modo sperimentale paragonando il tempo d'esecuzione del ciclo nel *caso migliore* e nel *caso pessimo* derivanti da un'analisi del WCET. Può comunque essere difficile procedere in questo modo in quanto le differenze nel tempo d'esecuzione che ci sono tra caso migliore e pessimo possono non essere dovute alla cache o possono non essere consistenti nel caso in cui non sia consistente il *caso migliore*. Una soluzione migliore è paragonare il WCET al tempo medio di esecuzione del programma.

Dal momento che il problema proposto in questa sezione dal Codice 4.3 rispecchia quello del Codice 4.2 valgono tutte le considerazioni fatte nella Sezione 4.2.

4.4 Impredicibilità dovuta alla Dimensione

La capacità totale della cache influenza il numero di miss e di hit: sarebbe dunque preferibile disporre di una cache abbastanza capiente per poter contenere interamente i programmi. Questo non è però sempre possibile a causa dei costi dello hardware e della complessità dei programmi, oltre al fatto che i cache miss non dipendono solo dal rapporto tra dimensione della cache e dimensione dei programmi ma anche dalla struttura dei programmi stessi.

La causa di impredicibilità di tipo **dimensione** può essere considerata come un caso speciale di causa di tipo sequential (vedi Sezione 4.3) in cui i conflitti sorgono successivamente alla scelta del layout. Un esempio che generalizza cause di impredicibilità di questo tipo è rappresentato nel Codice 4.4

```
while (...)
{
    /*sequenza di istruzioni la cui dimensione totale
    è maggiore della dimensione della I-cache */
}
```

Codice 4.4: Esempio di ciclo che causa un impredicibilità di tipo *Dimensione*.

Questo tipo di impredicibilità può presentarsi in molteplici occasioni che dipendono dalla tipologia delle istruzioni del blocco di codice interno al ciclo. Il caso più semplice è quello in cui il corpo del ciclo non contenga né chiamate a funzione né cicli innestati (non c'è quindi ripetizione del codice durante l'esecuzione di una singola iterazione del ciclo). Si supponga si presenti lo scenario seguente: la I-cache è vuota all'inizio dell'esecuzione del ciclo, i primi 8 KByte di codice sono caricati nella I-cache e vengono distribuiti uniformemente nei vari set della cache. A questo punto ogni set della cache contiene solo un blocco e quando altri 8 KByte di codice sono caricati vengono spartiti uniformemente tra i diversi cache set. A questo punto ogni set della cache contiene un blocco appartenente alla prima porzione di codice e uno appartenente alla seconda. Si supponga inoltre che vengano caricate altre porzioni di codice da 8 KByte ciascuna ripartita nel modo descritto. Necessa-

riamente si arriverà a un punto in cui la I-cache sarà piena distinguendosi così due casi: se il ciclo non ha più codice da allocare la sua successiva esecuzione produrrà solo cache hit, al contrario se il ciclo deve allocare altra memoria sarà necessario rimuovere un blocco in cache in base alla politica di sostituzione[6] dei blocchi. Nel secondo caso ad una seconda esecuzione del ciclo si avranno solo miss in quanto i blocchi che precedentemente occupavano la cache sono stati rimossi per consentire l'allocazione del codice rimanente.

Questo determinato tipo di miss viene denominato *capacity-miss* dal momento che è causato dalla dimensione della cache, troppo piccola rispetto alla dimensione del corpo del ciclo. La sequenza di codice così descritta è cache killer.

Due sequenze cache risk possono essere ricondotte a quella cache killer appena descritta: la primo è una sequenza in cui il corpo del ciclo non presenta né cicli innestati né chiamate a funzione e ha dimensione paragonabile a quella della I-cache; la seconda invece ha il corpo del ciclo di dimensioni superiori a quelle della I-cache e presenta cicli innestati che introducono delle ripetizioni.

L'analisi che una metodologia deve eseguire per poter identificare i modelli presentati è simile a quella descritta per riconoscere i modelli che presentano cause d'impredicibilità di tipo layout:

- identificare gli I-cache miss come una causa di aumento del tempo di esecuzione;
- individuare se questi miss avvengono durante l'esecuzione di un ciclo particolare e siano dovuti a conflitti nella cache;
- individuare se i conflitti sono dovuti alla dimensione del ciclo esaminato.

4.5 Impredicibilità dovuta alla Concorrenza

Un'applicazione può essere composta da numerosi task eseguiti contemporaneamente che concorrono per l'accesso alle risorse.

[6]Questo è vero in politiche di sostituzione come FIFO e LRU, ma non necessariamente valido per altre come LIFO.

L'interazione e l'interferenza tra task diversi può influire sul comportamento della cache e quindi introdurre variazioni del tempo di esecuzione totale e anche parziale dei vari task in base al loro ordine di esecuzione; questo non succede nel caso in cui lo stato della cache venga resettato quando avviene la *ripresa*[7] del task.

L'esecuzione di un task può essere interrotta in due situazioni da cui derivano due tipi di problematica concorrente: *pre-rilascio* e *sospensione*.

- Con il comando di pre-rilascio un task interrompe un altro task in un qualsiasi istante di esecuzione.
- Con il comando di sospensione invece il task si sospende volontariamente dopo aver concluso la sua attività.

L'assunzione che la cache non cambi durante gli interrupt permette di omettere gli interrupt stessi da questa trattazione.

Un modello che presenti il problema di tipo **concorrenza** è rappresentato in Figura 4.1. Si consideri il codice nell'ottica in cui le diverse procedure appartengano a diverse unità di traduzione e abbiano lo stesso layout in memoria come nel Codice 4.2.

I due task di Figura 4.1 hanno priorità diverse (Task1 ha priorità inferiore a Task2). Il task a priorità bassa contiene un ciclo nel cui corpo sono presenti quattro chiamate a funzione mentre il task a priorità alta contiene una sola chiamata a funzione. In assenza di pre-rilascio il Task1 completerebbe la sua esecuzione senza conflitti mentre quando il Task2 usa la pre-rilascio salva in cache il blocco della procedura P3che rimpiazza il blocco della procedura P4 (il meno usato, seguendo una politica LRU). In questo modo quando Task1 riprende la sua esecuzione (ipotizzando che la cache non venga azzerata) avverrà un cache miss perchè non sono presenti in cache tutti i blocchi relativi alle sue procedure (situazione analoga a quella del problema di tipo layout). Tuttavia, nel caso in cui non avvengano altre interruzioni dovute al comando di pre-rilascio, non ci saranno miss fino alla

[7]Per *pre-rilascio*, *sospensione* e *ripresa* si indica l'interruzione, la sospensione e il ripristino dell'esecuzione di un task; può avvenire per vari motivi come a causa di diverse priorità, l'esigenza di una risorsa non ancora disponibile o lo scadere del tempo dedicato al task dalla CPU, etc.

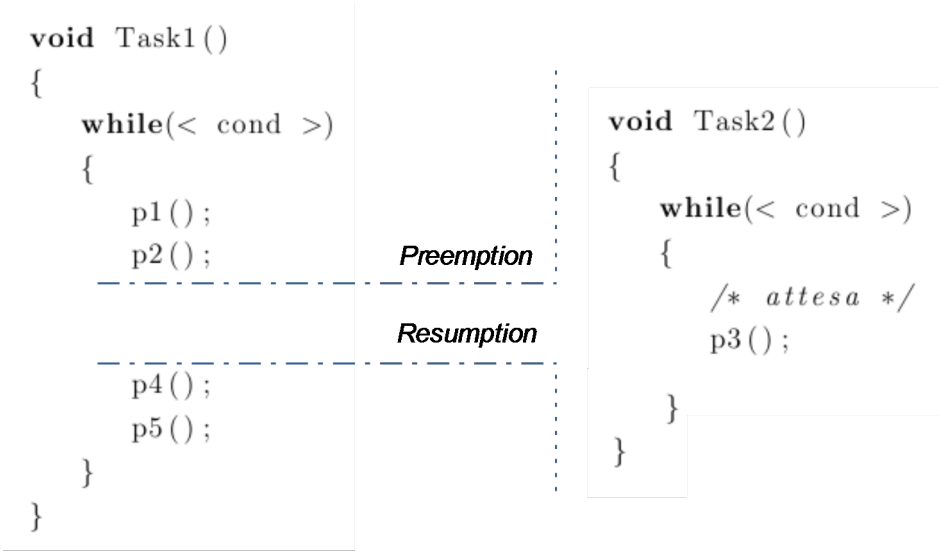

Figura 4.1: Esempio di *pre-rilascio* e *ripresa*: il *Task1* è in esecuzione prima di essere sospeso dal *Task2* che successivamente gli cede nuovamente il controllo.

successiva esecuzione di P4, quando gli effetti della chiamata a funzione P3 non saranno più visibili.

Altre considerazioni sono necessarie per permettere una valutazione completa degli effetti che la concorrenza ha sulla variabilità del tempo d'esecuzione.

La frequenza delle interazioni dovute ai comandi di pre-rilascio e il numero di iterazioni dei cicli all'interno dei task sono altri fattori di cui bisogna tener conto nella valutazione.

La prima caratteristica ci permette di considerare due possibili casi:

- Il task di partenza (Task1) è interrotto nell'intorno del medesimo punto d'esecuzione e sempre dallo stesso task (Task2): questo scenario è relativamente predicibile grazie alla sua regolarità.

- Il task di partenza (Task1) è interrotto in modo più irregolare: aumenta la variabilità del tempo d'esecuzione.

Combinando le due situazioni con il numero di possibili esecuzioni del ciclo si presentano due ulteriori modelli di impredicibilità di tipo concorrenza:

- pre-rilascio occasionale di cicli poco reiterati: un task a priorità alta che esegue un ciclo per poche iterazioni ogni volta che è attivato è occasionalmente interrotto dal pre-rilascio di un task con priorità maggiore e che presenta qualche conflitto I-cache con il primo.

- pre-rilascio persistente di cicli molto reiterati: un task a bassa priorità che esegue un ciclo per molte iterazioni ogni volta che è attivato è costantemente interrotto dal pre-rilascio di un task con priorità maggiore che presenta qualche conflitto I-cache con il primo.

Una metodologia per l'identificazione di questi modelli richiede la combinazione del calcolo del numero di iterazioni del ciclo (ottenibile con un profiler) e del numero di comandi di pre-rilascio (ottenibile con un simulatore o avendo a disposizione un dettagliato schema di schedulazione dei vari task).

Le due sequenze descritti rappresentano i *casi limite* tra le possibili interazioni di due task che fanno uso dei comandi di pre-rilascio e ripresa.

Capitolo 5

Implementazione della Metodologia

La metodologia proposta nel Capitolo 5 è stata implementata all'interno di *PandA* [20], 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;
- partizionamento dell'hardware/software;
- definizione di metriche per l'analisi ed il mapping su architetture multiprocessore;
- sintesi logica di circuiti digitali.

In particolare, l'implementazione del presente lavoro di tesi è stata integrata all'interno di uno degli strumenti del framework PandA: *Zebu*.

Nel presente Capitolo verrà presentata l'implementazione della metodologia discussa nel Capitolo 4.

Nella Sezione 5.1 verrà descritto il funzionamento di Zebu e le modifiche a esso apportate al fine di implementare la metodologia proposta.

5.1 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 2.1 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.

Il tool Zebu sfrutta tutte e tre le fasi di Front, Middle e Back End per eseguire la stima delle prestazioni.

- La fase *Front-End* di Zebu non accetta direttamente in input il codice sorgente C ma estrae una rappresentazione intermedia prodotta da una versione lievemente modificata del compilatore GNU GCC 4.3[6].

  Uno degli aspetti chiave della fase di Front End di Zebu è rappresentato dal tipo di rappresentazione intermedia, estratta dal GCC, che consente di effettuare un'analisi accurata del codice sorgente.

  Il compilatore GNU GCC è stato modificato in modo da avere informazioni su parti della struttura (come il CFG dei blocchi basici in Sezione 2.3.2) che prima non erano disponibili. E' stata anche integrata la possibilità di ottenere informazioni tra la corrispondenza delle operazioni GIMPLE e le rispettive operazioni RTL.

  La rappresentazione intermedia GIMPLE non è completamente indipendente dall'architettura destinazione: alcune informazioni, come per esempio la

dimensione dei tipi di dato, sono note solo in fase di trasformazione dal codice sorgente. Per questa ragione Zebu estrae la rappresentazione intermedia dallo *Sparc-cross-compiler*.

PandA permette di selezionare il tipo di ottimizzazione da eseguire in modo da apprezzarne gli effetti singolarmente. Le rappresentazioni intermedie GIMPLE e RTL sono estratte in due punti differenti della compilazione come descritto nella Sezione 2.1.

- Nella Middle-End viene eseguita la maggior parte delle analisi e delle trasformazioni possibili con Zebu. In particolare vengono prodotte due collezioni di grafi completi di annotazioni, la prima associa ai nodi del grafo le singole operazioni mentre la seconda ne associa i blocchi basici. Gli archi del grafo rappresentano le dipendenze e le precedenze tra i nodi.

  A ogni passo si usano le informazioni del GNU GCC per correlare le istruzioni del codice sorgente con le corrispondenti operazioni GIMPLE e RTL. Successivamente viene effettuato il calcolo dei CFG per ogni funzione.

  Vengono poi analizzati i CFG per l'identificazione dei costrutti *while* e *for* che vengono sostituiti con il costrutto *go-to* con l'aggiunta di *etichette* dal GCC; Zebu ha il compito di ricostruire i cicli con le nuove informazioni. Al momento Zebu non consente la ricostruzione perfetta di tutti i costrutti for e while presenti nel codice originario.

  Oltre agli aspetti statici è necessario tenere da conto anche di quelli dinamici come per esempio le probabilità di salto o il numero di iterazioni dei cicli. Queste informazioni sono spesso difficili da ottenere con buona approssimazione da un'analisi puramente statica.

  Zebu permette di memorizzare dati di profiling eseguendo l'applicazione sia sull'host sia sulla piattaforma target implementando una versione modificata dell'algoritmo Efficient Path Profiling. Questo è possibile sfruttando la Back End di PandA che permette di riscrivere una versione instrumentata dell'applicazione.

  Le informazioni prodotte durante questo passo consistono prevalentemente nella frequenza d'uso di ogni operazione nel CFG. Questo tipo di informa-

zione è indipendente dall'architettura target dal momento che dipende dalla rappresentazione intermedia GIMPLE. Dalla frequenza di ogni singolo percorso è poi possibile ottenere delle informazioni riguardanti la frequenza di esecuzione di ogni singolo blocco basico, salto e ciclo.

- Durante la Back-End si manipola la rappresentazione GIMPLE in modo da ottenere una versione instrumentata del codice sorgente su modello dell'architettura destinazione.

Sfruttando queste fasi, insieme alle rappresentazioni intermedie del compilatore, si possono integrare all'interno del framework PandA altre analisi utili per determinare predicibilità del codice:

- Stima delle dimensioni delle istruzioni;
- Analisi dei cicli;
- Identificazione dei pattern del codice.

5.2 Stima delle Dimensioni delle Istruzioni

Zebu lavora prevalentemente sulla rappresentazione GIMPLE che, essendo indipendente dall'architettura destinazione, non può fornire informazioni accurate sulle dimensioni dell'oggetto corrispondente nell'architettura target. L'uso della rappresentazione RTL permette al contrario, dal momento che dipende dall'architettura destinazione, di fornire una stima affidabile del codice prodotto.

Viene usato un contatore per il calcolo delle dimensioni delle istruzioni RTL. Il numero ottenuto rappresenta una sottostima del numero reale di istruzioni poiché il calcolo viene effettuato all'inizio del flusso di compilazione della Back End dove la maggior parte delle ottimizzazioni non è ancora stata eseguita.

Per questo motivo, dato un benchmark, il numero di istruzioni RTL risulta sempre inferiore al numero di istruzioni di tipo *assembly*[1]. Il rapporto tra il numero di istruzioni assembly e il numero di istruzioni RTL risulta comunque uniforme

[1]L'assembly, o linguaggio assemblatore è, tra i linguaggi di programmazione, quello più vicino al linguaggio macchina vero e proprio.

per i vari benchmark (il loro rapporto è 1,36). Tenendo in considerazione questa costante moltiplicativa, il numero di istruzioni RTL può essere considerato una buona stima della dimensione delle istruzioni.

5.3 Analisi dei Cicli

In questa fase Zebu colleziona alcune informazioni che verranno usate nella fase successiva per identificare i pattern cache risk e i pattern cache killer.

Questa parte d'analisi è composta da due passi:

1) Inizialmente viene analizzata l'applicazione a livello dei blocchi basici: ogni blocco basico è analizzato in modo da contare il numero di chiamate a funzione contenute nella corrispondente rappresentazione GIMPLE. Ogni singola operazione GIMPLE può contenere zero, una o più di una chiamata a funzione.

   L'analisi è in grado di verificare anche l'*inline*[2] delle funzioni durante il processo di compilazione del flusso dal quale viene prodotta la rappresentazione GIMPLE.
   Alla fine di questo passo viene calcolata la dimensione totale delle istruzioni del blocco basico tenendo in considerazioni la dimensione RTL corrispondente nel modo descritto nella Sezione 5.2.

2) Successivamente i dati calcolati al passo precedente vengono aggregati tenendo in considerazione le relazioni tra i cicli. Zebu analizza in modo iterativo tutti i cicli che sono stati identificati nella data funzione.
   L'ordine dell'analisi dei cicli corrisponde all'inverso dell'ordine dall'annidamento: i primi cicli analizzati sono le foglie dell'albero dei cicli (ovvero quei cicli che non hanno al loro interno altri cicli innestati) e procede verso la radice dell'albero (loop - 0) in modo che al momento dell'analisi di un ciclo siano già stati analizzati tutti i sui nodi figli (cicli innestati);
   L'ultimo ciclo a essere analizzato è il *Loop - 0*, il ciclo che rappresenta l'intera funzione.

[2] Ovvero inserire il codice intero della funzione ogni volta che la funzione viene usata all'interno del codice.

Durante l'analisi di ogni ciclo vengono calcolate quattro diverse quantità.

- la stima della dimensione totale del ciclo: questa quantità è calcolata come la somma delle dimensioni delle singole istruzioni di tutti i blocchi basici appartenenti al ciclo e ricorsivamente appartenenti ai cicli innestati.
- il numero dei distinti path d'esecuzione presenti nel Control Flow Graph del ciclo considerato: tenendo conto che i blocchi basici che appartengono a un ciclo che sia annidato nel ciclo corrente sono considerati come un nodo unico;
- il numero massimo di chiamate a funzione del ciclo: il numero di chiamate a funzione durante l'esecuzione di un ciclo dipende dal path d'esecuzione che si sta seguendo in quel momento; dunque questo parametro è calcolato tenendo in considerazione il numero massimo di chiamate a funzione di ogni path d'esecuzione per il ciclo corrente; se il ciclo analizzato contiene dei cicli annidati al suo interno, questi vengono considerati come singoli nodi il cui numero di chiamate a funzione consiste nel massimo numero di chiamate a funzione calcolato dall'analisi nell'iterazione precedente.
- il numero minimo di chiamate a funzione del ciclo: anche in questo caso se il ciclo contiene altri cicli innestati al suo interno, questi vengono trattati come nodi singoli; d'altro canto se il ciclo innestato non presenta il costrutto *do-while*, il numero di chiamate a funzione per questo vertice è calcolato tenendo in considerazione solo le chiamate a funzione presenti nell'header; quindi il numero minimo di chiamate a funzione eseguite dal ciclo in questo caso è equivalente a quelle presenti nell'header poiché non si entra nel ciclo.

5.4 Identificazione dei Pattern di Codice

Le informazioni calcolate dall'Analisi dei Cicli sono usate per valutare i pattern cache risk e cache killer descritti nel Capitolo 4.

5.4.1 Pattern dovuti al Layout

Dal momento che l'analisi di Zebu viene eseguita prima che il codice oggetto sia prodotto, non si possono avere delle informazioni su come il Linker ricomporrà il codice eseguibile. Tuttavia, anche se per questo motivo non è possiblile identificare con certezza i pattern cache killer, è possibile identificare i pattern cache risk che potrebbero a loro volta diventare cache killer. Per questo tipo di indagine si procede analizzando i dati prodotti dal passo precedente (identificazione dei cicli) considerando in particolare quei cicli in cui si può identificare un singolo path d'esecuzione.

Il singolo path può essere considerato un pattern cache risk se contiene più di quattro chiamate a funzione. Se il path identificato rappresenterà un pattern cache killer o meno dipenderà da come il Linker allocherà il codice delle funzioni chiamate.

5.4.2 Pattern Sequenziali

Questo tipo di analisi è molto simile a quella precedente. In questo caso non sono considerati solo i cicli che presentano un solo path d'esecuzione ma tutti i tipi di cicli. Si identificano come pattern cache risk per l'inconveniente di tipo Ordine i corpi dei cicli che hanno il numero massimo di chiamate a funzione maggiore dell'associatività della cache. Anche in questo caso il tool non è in grado di distinguere se si tratti di un pattern cache killer o di un pattern cache risk in quanto non ci sono abbastanza informazioni sul layout della memoria del codice eseguibile prodotto.

Nel caso in cui anche il numero minimo di chiamate a funzione, presenti nel corpo del ciclo analizzato, sia maggiore dell'associatività della cache si può affermare che il pattern corrente sia equivalente a un pattern che presenta l'inconveniente di tipo Layout.

5.4.3 Pattern dovuti alla Dimensione

L'identificazione di pattern che presentano l'inconveniente di tipo Dimensione può essere effettuata considerando la dimensione del corpo del ciclo stimata usando

i contatori delle istruzioni RTL. Se infatti la dimensione totale delle istruzioni del corpo del ciclo supera la dimensione del blocco della cache si otterranno solo miss per la richiesta di ogni altra istruzione (considerando l'uso di una politica di sostituzione dei blocchi in cache di tipo Least Recent Used).

Tuttavia la distinzione tra pattern cache risk e pattern cache killer risulta un compito difficile per il framework in quanto la dimensione delle istruzioni che costituiscono il corpo del ciclo è solo stimata e non calcolata.

Le ottimizzazioni di GNU GCC possono ridurre la dimensione del corpo del ciclo eliminando i pattern cache risk; tuttavia facendo ciò si riduce anche il controllo che il designer ha sull'applicazione e questo porterebbe a un aumento dell'impredicibilità. Al contrario, Zebu, permettendo la riscrittura del codice sorgente con le ottimizzazioni attuate dal compilatore GNU GCC, fornisce al designer l'opportunità di applicare ottimizzazioni per ridurre la dimensione del codice solo in determinate parti dell'applicazione. In questo modo la verifica dei risultati è effettuata direttamente sul codice sorgente.

5.4.4 Pattern dovuti alla Concorrenza

L'identificazione di pattern che presentano un inconveniente di tipo Concorrenza risulta difficile per un framework basato su un compilatore dal momento che un compilatore lavora analizzando i task singolarmente e non ha informazioni a proposito della loro interazione.

5.4.5 Altri Pattern

Un ulteriore causa di impredicibilità è la presenza di molteplici path di esecuzione: se l'analisi statica non è in grado di identificare la corretta successione dei path eseguiti, il Worst Case Execution Time deve essere calcolato basando la stima sempre sul path che richiede più risorse in termini di performance.

5.4.6 Identificazione del Codice non eseguito

L'eliminazione di codice non eseguibile, o *Dead Code Elimination*, eseguita da compilatori come il GCC, è in grado di rimuovere dell'applicazione porzioni di

codice che non possono essere eseguite. Tuttavia tecniche di analisi appartenenti alle tipologie citate nella Sezione **??** possono non rimuovere tutto il codice non eseguibile per due ragioni:

1. verificare che le condizioni che attivano particolari path di esecuzione sono insoddisfacibili è un compito arduo. La seconda è che alcune parti di codice possono essere considerate morte solo perché gli input delle applicazioni reali sono un sottoinsieme degli input possibili. Zebu è in grado di aiutare il progettista a riscrivere l'applicazione in modo da aumentare la sua predicibilità evidenziando le porzioni di codice non eseguibili per il particolare set di input considerato. L'eliminazione di codice non eseguibile può migliorare la predicibilità dell'applicazione in due modi:

1. Può ridurre il numero di sequenze cache killer e cache risk dovuti a cause di tipo Dimensione o Sequenziale;

2. Può aiutare strumenti per il calcolo del WCET rimuovendo parti di codice morto che contribuiscono al WCET ma che in realtà non verranno mai eseguite.

5.4.7 Identificazione di Cicli on Numero Variabile di Iterazioni

L'analisi WCET eseguita da strumenti come *a*3 è basata sui vincoli al numero di iterazioni dei cicli, o *loop iteration boundaries*. La presenza di cicli con un numero variabile di iterazioni può influenzare la predicibilità della funzione analizzata. Le informazioni di partizionamento raccolte da Zebu consentono di identificare facilmente cicli con questo genere di problematiche. Possibili soluzioni alle cause di impredicibilità sono due:

1. Se è possibile identificare una relazione tra il numero di iterazioni del ciclo e qualche caratteristica della particolare esecuzione del ciclo stesso (ad esempio il numero di iterazioni dipende dal contesto in cui viene chiamato il ciclo), i limiti al numero iterazioni del ciclo possono essere parametrizzati;

2. Se non esiste alcuna relazione, una possibile soluzione è quella di riscrivere parte dell'applicazione in modo da migliorare la sua regolarità.

Capitolo 6

Risultati Sperimentali

Nel presente capitolo vengono presentati i risultati sperimentali ottenuti attraverso la metodologia descritta nel Capitolo 4 e attraverso la relativa implementazione descritta nel Capitolo 5.

I test sono stati svolti su un server del laboratorio di Microarchitetture del Dipartimento di Elettronica e Informazione del Politecnico di Milano: processore Intel Xeon X5355, 4 core di esecuzione a frequenza 2,33 GHertz con 4 MByte di cache(L2) condivisa da dati e istruzioni e 16 GByte di memoria RAM.

L'architettura destinazione è rappresentata da un processore LEON3: tale processore è dotato di un architettura 32 bit single core di esecuzione con pipeline a 7 stadi e memoria cache configurabile da 1 a 4 vie, da 1 a 256 KByte/via con politiche di sostituzione Random o Least Recent Used.

La memoria del processore LEON AT697E utilizzato durante la fase di test è configurata con una I-cache di capacità 32 KByte con associatività a 4 vie e politica di sostituzione dei blocchi Least Recent Used.

Nella Sezione 6.1 verranno presentati i benchmark utilizzati per ottenere i risultati di stima del presente Capitolo.

Nella Sezione 6.2 verranno presentati i risultati del presente lavoro di tesi.

6.1 Benchmark utilizzati

Le analisi proposte in questo lavoro di tesi e integrate nel framework PandA sono state validate su una serie di programmi C provenienti dalla suite di benchmark MiBench[2].

Le caratteristiche dei benchmark utilizzati sono riportate nella Tabella 6.1.

Nome Benchmark	Numero linee di codice sorgente	Numero funzioni	Numero cicli	Numero cicli interni ai path	Numero Blocchi Basici	Tempo di analisi in secondi
Basic Math - Large	495	5	18	45	68	0.3
Basic Math - Small	456	5	13	35	53	0.25
Bitcount	917	19	7	52	77	0.47
Susan	2122	19	47	111681	649	1793.29
Qsort_Large	55	2	2	15	23	0.13
Qsort_small	45	2	2	11	19	0.1

Tabella 6.1: Caratteristiche dei benchmark appartenenti alla suite MiBench.

I benchmark del tipo *Basic Math* sono composti da calcoli matematici quali la risoluzione di equazioni di terzo grado e di radici quadrate. Il benchmark *Bitcount* è composto da alcune implementazioni di una funzione per il calcolo dei bit diversi da zero nelle rappresentazioni dei numeri. Il benchmark *Susan*, il più complesso di quelli analizzati, consiste in funzioni per l'elaborazione di immagini (il motivo per cui Zebu impiega molto tempo rispetto agli altri benchmark nell'analisi di Susan è che sono presenti molti path da analizzare). I due benchmark *Qsort* rappresentano l'implementazione dell'algoritmo Quicksort.

In Tabella 6.1 vengono riportate le caratteristiche salienti dei banchmark:

- il numero delle righe di codice sorgente;
- il numero di funzioni definite nel programma (non sono incluse le funzioni di libreria o quelle del sistema operativo);
- il numero dei cicli presenti nel programma;
- il numero dei path interni a un ciclo (non sono considerati i path che attraversano più di un ciclo o sono composti da più iterazioni di un singolo ciclo);
- il tempo di esecuzione dell'analisi di Zebu in secondi.

6.2 Risultati

In questa Sezione verranno presentati i risultati ottenuti attraverso l'applicazione della metodologia proposta. I risultati saranno divisi per categoria in base al tipo di stima effettuata e alla causa di impredicibilità ricercata. Per ogni tipologia di impredicibilità sarà mostrata una tabella che ne descrive l'occorrenza per la suite di benchmark selezionata.

6.2.1 Stima della Dimensione delle Istruzioni

La correttezza della stima della metodologia descritta nel Capitolo 4 è stata verificata attraverso la libreria Binary File Description[1].

I risultati sono riportati nella Tabella 6.2.

Nome Benchmark	Parametri	Numero totale Blocchi Basici	Numero Blocchi Basici eseguiti	Percentuale Blocchi Basici eseguiti
Basic Math - Large	-	68	66	97.05%
Basic Math - Small	-	53	50	94.33%
Bitcount	1125000	77	32	41.55%
Susan	-s	694	76	10.95%
Susan	-e	694	173	24.92%
Susan	-c	694	156	22.47%
Qsort_Large	-	23	19	82.60%
Qsort_small	-	19	17	11

Tabella 6.2: Relazione tra istruzioni del codice sorgente e istruzioni RTL.

I risultati ottenuti indicano il rapporto che intercorre tra la dimensione del codice sorgente di una applicazione reale e il corrispondente numero di istruzioni RTL; dai risultati si evince che il rapporto è relativamente costante. Da questi dati è possibile ottenere la dimensione totale delle istruzioni come risultato della

[1]Il BFD, o Binary File Descriptor library, è il meccanismo principale del progetto GNU per la manipolazione del file oggetto in una varietà di formati. A partire dal 2003, supporta circa 50 formati di file per circa 25 architetture di processore.

moltiplicazione tra numero di istruzioni RTL e valor medio[2]. Il massimo errore introdotto nella stima è quello del benchmark Susan del 21,98%.

6.2.2 Occorrenza impredicibilità dovute al Layout

Le informazioni ottenute sulle caratteristiche dei cicli permettono la ricerca di pattern cache risk e pattern cache killer all'interno dei programmi analizzati.

La Tabella 6.3 riporta l'analisi eseguita su ogni ciclo di tipo foglia (che non contiene quindi altri cicli innestati al suo interno) e che non ha alcuna espressione di controllo nel proprio corpo (quelli che presentano quindi un singolo path d'esecuzione).

Per ogni ciclo appartenente a queste categorie sono riportate le seguenti caratteristiche:

- il suo ID (che corrisponde all'indice del blocco basico dell'header del ciclo, assegnatogli dal GNU GCC);
- il numero di chiamate a funzione presenti nell'header e nel corpo del ciclo;
- se costituisca o meno un pattern di tipo cache risk o cache killer.

Il numero di chiamate a funzione è prevalentemente zero o uno, dunque per questo tipo di programmi è impossibile che si verifichi una causa di impredicibilità di tipo layout e si vengano a creare pattern cache risk, tanto meno si presenteranno pattern cache killer. Solamente un ciclo, che rappresenta una funzione presente nel *main* dell'algoritmo di Quicksort all'interno del benchmark Qsort, potrebbe costituire un pattern problematico ma, dal momento che non è disponibile nessuna informazione sul layout finale del codice eseguibile, non si può affermare con certezza che si tratti di un pattern cache risk o un pattern cache killer.

[2]Il valor medio è ottenuto dalla media aritmetica dei singoli rapporti in Tabella 6.2; in questo caso è 5,54

Nome Benchmark	Nome Funzione	Loop ID	Numero di chiamate a funzione	Cache-Risk / Cache-Killer
Basic Math - Large	main	32		NO
		52	4	NO
		49	2	NO
		62	2	NO
		66	2	NO
		25	1	NO
		22	1	NO
		19	1	NO
		16	1	NO
		10	1	NO
		7	1	NO
		4	1	NO
Basic Math - Small	main	20	1	NO
		37	4	NO
		34	2	NO
		31	2	NO
		13	1	NO
		7	1	NO
		4	1	NO
Bitcount	bit_count	3	1	NO
	main	7	1	NO
	bit_shifter	4	0	NO
	bitstring	4	0	NO
	bstr_i	4	1	NO
Susan	int_to_uchar	11	0	NO
		9	0	NO
	susan_principle_small	9	0	NO
	susan_principle	9	0	NO
	enlarge	11	0	NO
		7	1	NO
	susan_smoothing	22	1	NO
		14	1	NO
	getint	10	1	NO
Qsort_Large	main	16	1	NO
		6	7	YES
Qsort_small	main	12	1	NO
		6	1	NO

Tabella 6.3: Identificazione della causa di impredicibilità di tipo Layout.

6.2.3 Occorrenza di impredicibilità Sequenziale

Per l'identificazione di pattern che manifestino cause di impredicibilità di tipo Sequenziale si analizzano tutti i cicli che rappresentano una foglia nell'albero dei cicli.

L'analisi è volta a identificare i corpi dei cicli in cui persistono molteplici path d'esecuzione tali che il numero delle chiamate a funzione eseguite durante una singola iterazione del ciclo sia maggiore dell'associatività della cache. I risultati dell'analisi sono riportati in Tabella 6.4.

Nome Benchmark	Nome Funzione	Loop ID	Numero dei path	Numero minimo di chiamare a funzione	Numero massimo di chiamare a funzione
Basic Math - Large	usqrt	6	2	0	0
Basic Math - Small	usqrt	6	2	0	0
Bitcount	bitstring	9	2	0	0
Susan	int_to_uchar	8	4	0	0
	median	7	2	0	0
	edge_draw	12	2	0	0
		7	2	0	0
	susan_thin	30	2	0	0
	corner_draw	7	2	0	0
	susan_principle_small	7	2	0	0
	susan_principle	7	2	0	0
	susan_corner_quick	82	50	0	0
		25	20	0	0
	susan_corners	105	50	0	0
		48	52	0	2
	susan_edges_small	48	1251	0	1
		7	2	0	0
	susan_edges	52	3363	0	1
		7	2	0	0
	function_setup_brightness	6	2	1	1
	susan_smoothing	38	2	0	1
	Getint	3	2	1	2
	Main	24	13	0	1

Tabella 6.4: Identificazione della causa di impredicibilità di tipo Sequenziale.

Dai risultati si evince che, all'interno dei programmi usati per testare l'eventuale

presenza di impredicibilità di tipo Sequenziale, non è probabile che si crei un pattern cache risk in quanto il numero massimo di chiamate a funzione che è possibile riscontrare è pari a due.

6.2.4 Occorrenza di impredicibilità di tipo Dimensione

La stima della dimensione del corpo dei cicli è effettuata usando il numero di istruzioni RTL corrispondenti al codice sorgente di ogni programma.

Vengono esaminati non solo i cilci che rappresentano le foglie dell'albero dei cicli ma tutti i cicli presenti nelle funzioni contenute nei programmi. La grandezza del corpo dei cicli che non rappresentano una foglia è calcolata tenendo in considerazione anche le grandezze dei cicli innestati.

La Tabella 6.5 rappresenta i risultati sperimentali di questo tipo d'indagine.

Dal momento che la dimensione della cache dell'architettura destinazione (32 KByte) è molto maggiore del ciclo più pesante, in termini di dimensione in Byte (rappresentato dal loop 79 del benchmark Susan con circa 8 KByte), anche considerando un improbabile quanto grande errore di stima, si può assumere dai dati che non ci siano le condizioni per la presenza di un pattern di tipo cache risk per il verificarsi di una causa di impredicibilità di tipo Dimensione.

6.2.5 Codice non eseguito

I benchmark della suite MiBench sono stati profilati fornendo un significativo set di input alla macchina host in modo da calcolare quanti blocchi basici vengano effettivamente eseguiti. Ad ogni esecuzione della simulazione è stato riportato il numero dei blocci basici identificati dall'analisi, il numero dei blocchi eseguiti ed è stato poi calcolato il rapporto tra questi.

Il banchmark Susan è stato eseguito in tre diverse modalità : edge detection, corner detection e smoothing.

I risultati sono riportati in Tabella 6.6.

I risultati evidenziano come il codice dei benchmark Basic Math e Qsort sia eseguito quasi completamente. Al contrario, solo una piccola parte del codice dei benchmark Susan e Bitcount corrisponde a una parte che è stata eseguita. Nel benchmark Susan, in particolare, questo è dovuto al fatto che ognuna delle tre

Nome Benchmark	Nome Funzione	Loop ID	
Basic Math - Large	main	40	448
	usqrt	6	128
Basic Math - Small	main	28	448
	usqrt	6	832
Bitcount	bit_count	3	60
	main	13	436
	bit_shifter	4	112
	bitstring	9	204
	bstr_i	4	172
Susan	int_to_uchar	8	96
	median	9	188
	edge_draw	7	282
	susan_thin	79	8400
	corner_draw	7	356
	susan_principle_small	9	568
	susan_principle	9	1852
	susan_corners_quick	84	5928
	susan_corners	48	6416
	susan_edges_small	50	2072
	susan_edges	54	6560
	setup_brightness_lut	6	152
	Enlarge	13	372
	susan_smoothing	40	1192
	getint	3	140
	main	24	1084
Basic Math - Large	main	6	780
Basic Math - Small	main	6	

Tabella 6.5: Identificazione della causa di impredicibilità di tipo Dimensione.

Nome Benchmark	Parametri	Numero totale Blocchi Basici	Numero Blocchi Basici eseguiti	Percentuale Blocchi Basici eseguiti
Basic Math - Large	-	68	66	97.05%
Basic Math - Small	-	53	50	94.33%
Bitcount	1125000	77	32	41.55%
Susan	-s	694	76	10.95%
Susan	-e	694	173	24.92%
Susan	-c	694	156	22.47%
Qsort_Large	-	23	19	82.60%
Qsort_small	-	19	17	**89.47%**

Tabella 6.6: Percentuale dei blocchi basici eseguiti durante i test.

funzionalità citate sopra può essere attivata solo individualmente. Bitcount invece ha all'interno porzioni di codice che non vengono mai lanciate durante i test eseguiti.

6.2.6 Cicli con un Numero Variabile di Iterazioni

Se il numero medio di iterazioni di un ciclo non è costante vuol dire che dipende dal particolare contesto di esecuzione. I cicli che appartengono ai programmi della suite MiBench che hanno manifestato queste caratteristiche sono riportati in Tabella 6.7

Nome Benchmark	Parametri	Funzione	Ciclo	Numero medio di iterazioni	Numero massimo di iterazioni
Basic Math - Large	-	main	32	1.1	3
Basic Math - Small	-	main	20	1.11	3
Bitcount	1125000	bit_count	3	15.5	27
Susan	-s	getint	10	1.33	2
			3	0.66	2
Susan	-e	susan_thin	77	68.04	70
		getint	10	1.33	2
			3	0.67	2
Susan	-c	getint	10	1.33	2
			3	0.66	2

Tabella 6.7: Numero di Iterazioni dei Cicli.

Nella Tabella 6.7 le differenze tra il numero di iterazioni dei cicli sono dovute alla difficile definizione del contesto d'esecuzione. Questo determinato tipo di errore è risolvibile in alcuni casi instrumentando il codice problematico. Come per esempio in situazioni simili a quella del benchmark Basic Math, dove non risulta possibile isolare la porzione di codice problematico in quanto non è possibile distinguere i contesti d'esecuzione per i quali i cicli cambiano numero di iterazione, l'accuratezza del Worst Case Execution Time ne risente maggiormente.

Capitolo 7

Conclusioni e possibili sviluppi futuri

In questo lavoro di tesi è stata proposta una possibile metodologia per l'individuazione di sequenze di codice problematiche per la stima del Worst Case Execution Time. Partendo da sequenze di codice proposte per una analisi a livello assembly in un precedente lavoro[1], la metodologia proposta analizza le rappresentazioni intermedie fornite dal compilatore GCC per individuare lo stesso tipo di sequenze ma a un più alto livello di astrazione. In particolare dall'analisi di queste rappresentazioni intermedie è possibile estrarre informazioni a proposito delle caratteristiche di cicli e funzioni come la stima della dimensione del codice oggetto e il numero di chiamate a funzione. Sulla base di queste informazioni viene effettuata una stima della presenza di possibili cause di impredicibilità dovute a layout, sequenza e dimensione, senza la necessità di avere alcuna informazione sul codice assembly.

I risultati dei test effettuati hanno dimostrato che la metodologia è in grado di individuare all'interno di una applicazione le sequenze critiche ricercate. Tuttavia, l'analisi di un significativo set di benchmark estratto da una suite usata in ambito industriale ha mostrato come tali sequenze siano in realtà piuttosto rare all'interno delle applicazioni. Per questo motivo è necessario estendere il presente lavoro al fine di individuare le sequenze di codice che effettivamente possono introdurre problematiche di predicibilità nella stima del Worst Case Execution Time.
In particolare i possibili sviluppi futuri della metodologia proposta consistono in:

- estendere la metodologia in modo da consentire di individuare altre tipologie di sequenze critiche di codice;
- applicare la metodologia in un contesto di simulazione a più processi in modo da verificare la presenza di cause di impredicibilità di tipo concorrenza previste dalla metodologia stessa;
- estendere la metodologia in modo da poter analizzare architetture con memoria D-cache o memoria condivisa istruzioni-dati;

Riferimenti bibliografici

[1] PEAL Final Report, Doc. Ref. TR-PEAL-FR-001. 2, 37, 69

[2] *MiBench: A free, commercially representative embedded benchmark suite*, 2001. 60

[3] Randy Allen e Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9, October 1987. 31

[4] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998. 25

[5] Alexis Arnaud e Isabelle Puaut. Dynamic instruction Cache Locking in Hard Real-Time Systems. In *Conference on Real-Time Systems*, 2006. 34

[6] Free Software Foundation. GNU Compiler Collection GCC, version 4.3. 3, 7, 50

[7] Somnath Ghosh, Margaret Martonosi, e Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21:703–746, July 1999. 30

[8] Jan Gustafsson, Andreas Ermedahl, e Björn Lisper. Towards a Flow Analysis for Embedded System C Programs. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 287–300, Washington, DC, USA, 2005. IEEE Computer Society. 24

[9] Jan Gustafsson, Björn Lisper, Christer Sandberg, Christer S, e Nerina Bermudo. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *In 8 th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003*, 2003. 24

[10] Christopher Healy e David Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, RTAS '99, pp. 79–, Washington, DC, USA, 1999. IEEE Computer Society. 24

[11] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, e Wilhelm Reinhard. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7), 2003. 21

[12] Bach Khoa Huynh, Lei Ju, Sudipta Chattopadhyay, e Abhik Roychoudhury. Program Transformations for Predictable Cache Behavio. 05 2009. 28

[13] Angewandte Informatik. AbsInt. 22

[14] Jan Reineke, Daniel Grund, Christoph Berg, e Reinhard. 33

[15] J.Engblom. On hardware and hardware models for embedded real-time systems. 2001. 33

[16] Sung-Kwan Kim, Sang Lyul Min, e Rhan Ha. Efficient worst case timing analysis of data caching. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:230, 1996. 29

[17] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, e Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21:593–604, 1995. 29

[18] Tidorum Ltd. Bound-T. 22

[19] Enrico Mezzetti, N.Holsti, A.Colin, G. Bernat, e T. Vardanega. Attacking the sources of unpredictability in the instruction cache behaviour. November 2008. 32

[20] μ-lab Dipartimento di Elettronica e Informazione. PandA Project, 2006. 49

[21] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, November 1995. 33

[22] Henk Muller, David May, James Irwin, e Dan Page. Novel Caches for Predictable Computing. Relazione tecnica, Bristol, UK, UK, 1998. 33

[23] I. Puaut. Cache Analysis vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems. In *Proc. of the 2nd International Workshop on worst-case execution time analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002. 34

[24] Jan Reineke, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, e Bernd Becker. A definition and classification of timing anomalies. 2010. 34

[25] Wilhelm Reinhard. Determining Bounds on Execution Times. 2008. 22

[26] Martin Schoeberl. A Java processor architecture for embedded real-time systems. 2008. 30

[27] Martin Schoeberl. Time-predictable computer architecture. *EURASIP JOURNAL ON EMBEDDED SYSTEMS*, 2009, 2009. 30

[28] Vugranam C. Sreedhar, Guang R. Gao, Yong fong Lee, e Yong fong Leey. Identifying loops using DJ graphs, 1995.

[29] William Stallings. *Sistemi operativi*. Jackson Libri, 2000. 14, 19

[30] Friedhelm Stappert e Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. Syst. Archit.*, February 2000. 25

[31] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Heckmann Reinhold, Mitra Tulika, Mueller Frank, Puaut Isabelle, Puschner Peter, Staschulat Jan, e Stenstrom Per. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7, May 2008. 20, 25, 26

[32] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih wei Liao, Chau wen Tseng, Mary W. Hall, Monica S. Lam, e John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29, 1994. 31

[33] Koren Yehuda. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Trans. Knowl. Discov. Data*, 4(1):1–24, 2010.

[34] Hu Yifan, Koren Yehuda, e Volinsky Chris. Collaborative Filtering for Implicit Feedback Datasets. In *ICDM '08: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pp. 263–272, Washington, DC, USA, 2008. IEEE Computer Society.

Ringraziamenti

Vorrei ringraziare me stesso!!!