POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Sviluppo e validazione sperimentale di un algoritmo per ottimizzare l'efficienza energetica di più funzioni computation intensive tramite tecniche di memoizzazione

Relatore: Prof. Chiara Francalanci
Correlatori: Ing. Eugenio Capra
Ing. Marco Bessi

Tesi di Laurea di:
Domenico Iero
739407

Anno Accademico 2010-2011

Alla mia famiglia

Sommario

Compito del Green IT è quello di trovare soluzioni che riducano il consumo energetico in ambito IT. Il consumo energetico è dovuto sia alle componenti hardware sia alle applicazioni software. Nonostante ciò, fino a pochi anni fa la ricerca nell'ambito Green IT si è concentrata maggiormente sull'efficienza energetica dell'hardware. Al fine di ottenere un risparmio energetico anche lato software, è necessario ottimizzare gli algoritmi il che richiede però la conoscenza del dominio e di una accurata analisi del codice, che può essere impraticabile e troppo costosa da effettuare per basi di codice di grandi dimensioni. In questa Tesi si è partiti da un approccio basato sulla memoizzazione dinamica, tramite cui viene identificato un sottoinsieme di funzioni pure che possono essere tabulate e di cui vengono memorizzati i risultati. Obiettivo della Tesi è definire un modello tramite cui si possa gestire dinamicamente la memoria all'interno dell'architettura di memoizzazione. In seguito alla creazione di questo modello è stato sviluppato un blocco di *Trade off* il cui compito è di gestire la memoria disponibile, al fine di rendere la metodologia della memoizzazione flessibile rispetto a cambiamenti delle caratteristiche delle funzioni precedentemente identificate come pure. Il blocco di *Trade off* va a sostituire una precedente gestione statica della memoria, con una dinamica il cui fine è aumentare l'efficienza energetica del software. Infine il prototipo è stato testato su un benchmark set di librerie finanziarie e statistiche. I risultati empirici mostrano un risparmio energetico medio del 96.8% delle prestazioni e risparmio di tempo del 97%.

Abstract

Green IT aims is to find solutions to reduce energy consumption in IT context. Energy consumption is due to both hardware and software. Nevertheless, up to a few years ago, Green IT research had focused more on energy efficiency of the hardware. In order to have energy savings also on software side, it is necessary to optimize algorithms, studying performances and tuning applications. This requires domain knowledge and an accurate analysis of the code, which can be infeasible and too expensive to perform for large code bases. In this Thesis we started from an approach based on dynamic memoization, through which it is identified a subset of pure functions that can be tabulated and the results of which are stored. The Thesis aims to define a model by which you can dynamically manage memory within the memoization architecture. After the creation of this model has been developed a *Trade off* module whose task is to manage the available memory in order to make memoization methodology flexible with respect to changes in the characteristics of the functions previously identified as pure. The *Trade off* module replaces a previous static memory management with a dynamic one whose purpose is to improve energy efficiency of the software. We implemented a prototype software system to apply memoization and tested it on a set of financial functions. Empirical result show average energy savings of 96.8% and time performance savings of 97%.

Ringraziamenti

Dopo questi anni passati al Politecnico è d'obbligo ringraziare coloro che hanno segnato questo percorso.

Desidero innanzitutto ringraziare i miei genitori che in questi anni di università non hanno mai smesso di credere in me, permettendomi di portare avanti i miei studi e di coltivare i miei interessi.

Un grazie va alle mie sorelle che con i loro suggerimenti e ammonimenti mi hanno indirizzato verso scelte senza dubbio migliori. In particolare ringrazio Manuela per l'enorme supporto concesso ad un neodiplomato un po' spaesato in una città grande come Milano; ringrazio Lidia per avermi tollerato nonstante tutto e per essere stata la mia corelatrice per la laurea triennale; e ringrazio Roberta per essermi stata così vicina nonostante fosse così lontana.

Un grazie a zia Lina e zio Tommaso per la loro costante presenza, un grazie anche a tutti gli altri zii e cugini.

Ringrazio tutti i miei amici di Reggio Calabria che in questi anni ho visto poco per via della lontananza, ma che mi hanno regalato momenti di felicità durante i miei ritorni a casa. E ringrazio Giancarlo, Davide, Roberto e Antonio per le partite di basket sotto il sole cocente.

In questi anni ho avuto inoltre la fortuna di incontrare degli insostituibili compagni di studio, ma anche di gioco, che desidero ringraziare perchè hanno reso il mio percorso di studi più leggero, quindi ringrazio Daniele, Gianlu-

ca, Adolfo e Antonella per la categoria informatici, ringrazio Maurizio per la categoria informatici pentiti e passati a gestionale, ed infine ringrazio i gestionali puri Davide e Matteo, che nonostante abbiano perso un sacco di sfide a Call of Duty presentano dei margini di miglioramento.

Un ringraziamento va agli amici del dojo ed in particolare al maestro Stefano che ha saputo creare un gruppo così solido e affiatato.

Per quanto riguarda il presente elaborato invece ringrazio la prof. Chiara Francalanci, l'ing. Eugenio Capra, l'ing. Marco Bessi e il prof. Giovanni Agosta, li ringrazio per la loro disponibilità e per l'opportunità concessami.

Ringrazio inoltre tutti coloro non esplicitamente citati che con il loro contributo hanno fatto si che riuscissi a raggiungere questo traguardo.

Come ultimo ringraziamento, solo per ordine e non per importanza, ringrazio la mia fidanzata Maria che negli ultimi due anni mi è stata sempre vicina, dandomi un prezioso supporto in ogni situazione, sopportando tutte le mie paranoie e le mie ansie.

Domenico

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il Green IT è la disciplina che studia l'efficienza energetica dell'IT e da alcuni anni è al centro dell'attenzione a livello sia accademico che industriale per via degli enormi benefici che potrebbe apportare. In quest'ambito è possibile distinguere diversi campi di azione, tra i quali i più importanti sono:

- Postazioni di lavoro,
- Data center,
- Green hardware,
- Green software.

Diverse ricerche sono state svolte sulle prime tre, portando a risultati significativi, mentre il tema dell'efficienza energetica del software applicativo risulta ancora relativamente nuovo e inesplorato, nonostante il suo impatto possa risultare significativo. Infatti è vero che i dispositivi hardware siano fisicamente responsabili del consumo di energia, dissipata sotto forma di calore, ma è altrettanto vero che il responsabile primo di tale consumo è il software, che guida l'hardware e determina quali e quante operazioni elementari devono essere eseguite.

Il green software, quindi, è la disciplina che studia le modalità secondo cui il software influisce sui consumi energetici dell'IT e come ottimizzarle. Per ottenere del software che minimizza il consumo energetico sono possibili diverse vie: si potrebbe investire nella fase di sviluppo del codice, ingaggiando programmatori esperti, oppure applicare tecniche di modifica del software già creato. La prima scelta potrebbe portare a una riduzione del consumo di energia pari a 3 ordini di grandezza [7], ma richiede programmatori con una profonda esperienza del dominio e degli algoritmi di programmazione. Dal punto di vista economico scegliere programmatori esperti costituisce un costo elevato sia in termini di progetto software sia in termini di manutenzione, infatti il codice potrebbe essere più difficile da interpretare e modificare. Per quanto riguarda la seconda scelta possibile, cioè la modifica di software già creato, è possibile applicare la tecnica della memoizzazione.

La tecnica di memoizzazione consiste nel salvataggio in memoria dei risultati delle funzioni, in modo tale da riutilizzarne i valori in futuro, in caso di una chiamata di funzione con gli stessi parametri di ingresso. Applicando questa tecnica la CPU non deve rifare calcoli già eseguiti in passato, perchè verrà demandato alla memoria il compito di ricavare i risultati.

Studi precedenti hanno inoltre mostrato che il 60% circa del consumo energetico medio di un server è dovuto alla CPU. Le memorie e dischi invece consumano una quantità di energia notevolmente minore e si è osservato che la loro potenza assorbita sia quasi indipendente dall'utilizzo, a differenza della CPU che consuma di più se utilizzata maggiormente. In seguito a tali osservazioni si è pensato di demandare alla memoria, ove possibile, compiti che altrimenti avrebbe dovuto svolgere la CPU, al fine di ottenere un risparmio energetico.

Quindi, applicando la tecnica della memoizzazione e prendendo in considerazione questi studi, è nato un intero filone di ricerca, orientato al risparmio energetico, all'interno del quale trova posto il seguente lavoro di Tesi.

Bisogna però considerare che la memoizzazione è vantaggiosa allorquando si considerano funzioni complesse, la cui esecuzione richiederebbe un tempo elevato e i cui parametri di ingresso siano distribuiti secondo una gaussiana, cioè che si ripetano con un'alta probabilità. Per la scelta delle funzioni si è fatto riferimento a [9], queste infatti dovranno possedere certe caratteristiche per essere memoizzabili: essere deterministiche e non presentare side-effect.

In questo lavoro di Tesi è stato definito un modello tramite cui si possa gestire dinamicamente la memoria, nel caso vengano tabulate più di una funzione. Proprio per la presenza di più funzioni, che occupano la memoria in maniera concorrente, è necessario un modello che ne valuti le caratteristiche al fine di ottenere una corretta gestione della memoria. Successivamente, in base a questo modello, si è sviluppato un blocco che è stato inserito all'interno di un'architettura di memoizzazione già esistente.

La struttura con cui è organizzata questa Tesi è la seguente:

- Nel Capitolo 2 viene illustrato il concetto di Green IT e di Green Software, e l'importanza che rivestono attualmente queste tematiche. Inoltre viene descritta la tecnica di memoizzazione e viene definito il concetto di funzione pura.

- Nel Capitolo 3 viene presentato il metodo utilizzato per la risoluzione del problema connesso alla riduzione del consumo energetico in programmi Java, per mezzo dell'utilizzo di funzioni pure. Viene in seguito descritta l'architettura generale del framework di memoizzazione e ne vengono analizzati le singole componenti.

- Nel Capitolo 4 viene analizzato nel dettaglio il modello del *Trade off*, necessario per la successiva implementazione del blocco di *Trade off* che all'interno dell'architettura di memoizzazione ha lo scopo di gestire le aree di memoria. Viene introdotto il concetto di guadagno, metrica tramite la quale si valutano le funzioni pure. Il guadagno viene ap-

punto utilizzato all'interno di due algoritmi, spiegati sempre in questo capitolo, tramite cui vengono prese decisioni in merito alla quantità di memoria concessa alle funzioni, in modo tale da avere un'allocazione dinamica della memoria dettata dalle caratteristiche delle chiamate di funzione.

- Nel Capitolo 5 viene analizzato il codice finalizzato all'implementazione del blocco di *Trade off* e con esso le modifiche al codice esistente nell'architettura di memoizzazione precedente.

- Nel Capitolo 6 vengono riportati i risultati ottenuti in questo lavoro di Tesi, relativi all'introduzione del *Trade off*, in termini di riduzione del consumo energetico nei programmi Java.

- Nel Capitolo 7 sono tracciate le conclusioni finali del lavoro svolto e proposte nuove linee di sviluppo, sia per migliorare le funzionalità già realizzate, sia in prospettiva di aggiungerne delle nuove.

Capitolo 2

Stato dell'arte

In questo capitolo viene presentato lo stato attuale della ricerca sull'efficienza energetica nell'ambito dell'Information Technology (IT) e vengono definiti i termini tecnici principali relativi a questo progetto di Tesi. Nella Sezione 2.1 viene introdotto il concetto di Green IT, di cui vengono descritti gli aspetti fondamentali. La Sezione 2.2 mostra come il mondo della ricerca affronti il tema dell'ottimizzazione energetica del software, che muove solo di recente i suoi primi passi. Nella sezione 2.3.1 verrà descritta invece la tecnica della tabulazione, nota da tempo ormai nel settore con il nome di memoizzazione. La memoizzazione, come avremo modo di vedere più avanti, permette di ottenere un miglioramento sia delle performance che del consumo energetico di un metodo software, consentendo di realizzare il calcolo e la memorizzazione di valori intermedi che verranno successivamente riutilizzati al fine di velocizzare le operazione necessarie. Le funzioni che si prestano ad essere trattate con tale metodo prendono il nome di funzioni pure e nella sezione 2.4 se ne da una definizione formale secondo quanto presente in letteratura.

2.1 Green IT

Il risparmio energetico nell'IT è uno dei temi che cominciano oggi ad assumere un'importanza fondamentale sia per quanto riguarda il problema dell'inquinamento globale, sia per quanto riguarda l'innalzamento dei costi sostenuti dalle aziende per gestire e mantenere i propri sistemi informativi. L'Information and Communication Technology (ICT) ha un forte impatto sulle emissioni di carbonio nell'atmosfera e sul cambiamento del clima mondiale. Si calcola che le infrastrutture IT siano responsabili di più del 2% delle emissioni di CO2 mondiale [14]. Una ricerca del Gartner Group ha stimato, infatti, che un miliardo di tonnellate di emissioni di CO2 sono da imputare al settore IT mentre le emissioni globali ammontano a 49 miliardi di tonnellate l'anno. La riduzione del consumo di energia elettrica è quindi la chiave per diminuire le emissioni di diossido di carbonio, il loro impatto sull'ambiente e sul riscaldamento globale. L'IT influisce sull'ambiente in diverenti modi. L'intero ciclo di vita di un computer, dalla produzione alla sua dismissione, apporta diversi problemi ambientali. La produzione dei computer consuma elettricità, prodotti chimici ed altro, divenendo così causa di inquinamento. Inoltre l'energia elettrica utilizzata per far funzionare server, computer, monitor, reti di comunicazione e sistemi di raffreddamento di data center, è in continuo aumento. Ogni PC, infatti, genera ogni anno una tonnellata di diossido di carbonio [6, 13], mentre un server consuma la stessa energia per la cui produzione viene emessa la stessa quantità di CO2 prodotta da un SUV per percorrere 25km. Il crescente sviluppo tecnologico ha portato alla creazione di processori sempre più veloci e potenti con un conseguente incremento dell'energia necessaria al processing. I comuni processori Intel Core dissipano in media 110W con un incremento del consumo del 110% rispetto ai vecchi 486 che consumavano mediamente circa 10W. Se si considera poi che un moderno blade server consuma 1kW, quanto consuma un frigorifero di casa, un rack di server blade ne consuma 40kW, quanto un'in-

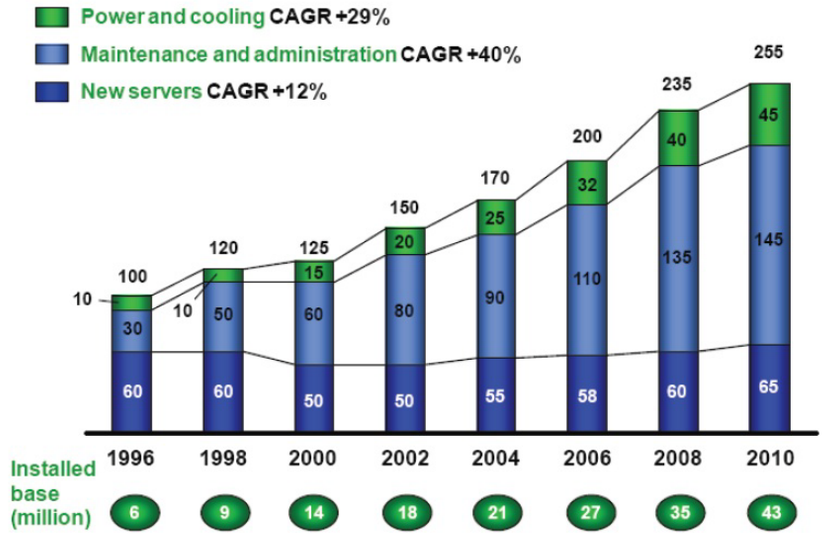

Figura 2.1: Tasso di crescita annuale dei costi delle infrastrutture IT. Fonte: IDC(2006).

tera palazzina. Un data center di medie dimensioni consuma come un intero quartiere, 250kW, mentre data center di grandi dimensioni possono arrivare a consumare 1MW, consumo paragonabile a quello di un'intera cittadina.

Come si può vedere nelle Figure 2.1 e 2.2, negli ultimi dodici anni il costo dell'hardware è cresciuto meno rispetto ai costi per l'energia ed il raffreddamento; il rapporto di crescita è infatti di quattro volte. Intelligent Distributed Computing (IDC) stima [8] che per ogni $ 1.00 speso oggi per un nuovo server, $ 0.50 centesimi vengono spesi in energia e raffreddamento. Se non ci saranno drastici cambiamenti si calcola che la spesa crescerà fino a $ 0.70 a partire dal 2011. In Italia, in due anni, il prezzo dell'energia elettrica per il settore industriale è aumentato del 18,6 %. Nonostante ciò, solo di recente ed ancora in numero ridotto, le aziende hanno cominciato ad annoverare i costi dell'energia tra i costi delle infrastrutture IT nel Total Costof Ownership (TCO).

La maggior parte delle aziende, infatti, non mette direttamente in relazione questi costi ai costi del settore IT rendendo così molto diffcile l'analisi

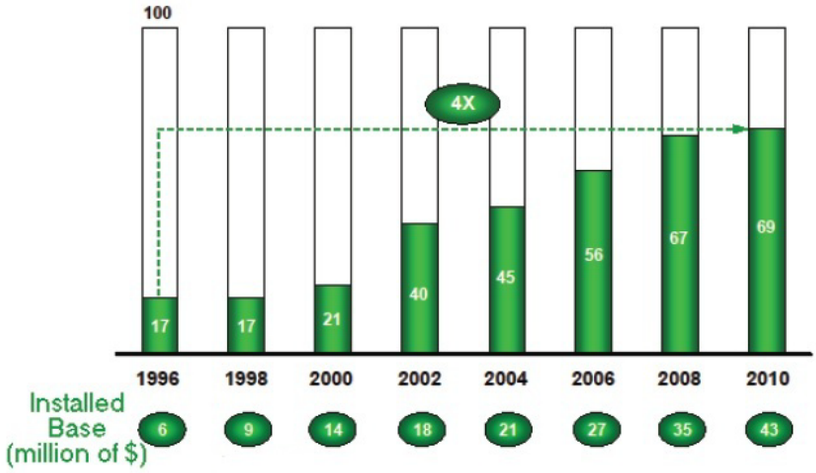

Figura 2.2: Rapporto tra i costi dell'energia e del raffreddamento rispetto ai costi di acquisizione di nuovi server. Fonte: IDC(2006).

dettagliata e la percezione del problema. Non è solo il costo dell'energia ad essere in continua crescita; anche il suo consumo cresce costantemente. Questo fenomeno sta diventando un limite alla scalabilità dei data center di medie e grandi imprese dislocate in aree ad alta densità abitativa. Ogni anno la potenza elettrica richiesta cresce dell'8-10 % ed i gestori di energia rischiano di non riuscire a convogliare l'energia necessaria in aree urbane ristrette. La potenza assorbita per metro quadro dai nuovi blade server è spesso incompatibile con le caratteristiche elettriche degli attuali data center. Una soluzione potrebbe essere il decentramento di queste infrastrutture in aree con una minor densità abitativa; ciò comporterebbe, però, ulteriori costi e danni ambientali.

2.1.1 Green IT per un IT sostenibile

Green IT (o Green Computing), è lo studio e la sua conseguente applicazione alla progettazione, produzione, uso e smaltimento di computer, server e sottosistemi associati, in modo effciente e con il minor impatto ambientale possibile. Il Green IT cerca inoltre di far coincidere interessi economici

e miglioramenti delle performance con le responsabilità etiche e sociali che derivano dalle attuali condizioni ambientali precarie. Tre sono le macroaree principali di cui il Green Computing si occupa:

- l'effcienza energetica dell'IT;
- la gestione ecocompatibile del ciclo di vita dell'IT
- l'utilizzo dell'IT come strumento per una governance "green".

L'effcienza energetica può essere migliorata agendo sia sulla progettazione che sulla gestione delle strutture e dei data center, modificando la cultura aziendale e le pratiche di utilizzo dell'hardware a disposizione. In realtà l'effcienza energetica, cioè il rapporto tra consumo energetico e prestazioni, ha visto una crescita negli ultimi anni poichè le prestazioni sono migliorate più di quanto non sia cresciuta l'energia richiesta. La seconda macroarea include tutte le azioni riconducibili a una gestione sostenibile del ciclo di vita dell'IT, dalla produzione alla dismissione dei sistemi. L'inquinamento causato dall'IT non è, infatti, unicamente riferibile al consumo di energia elettrica ma deriva, in parte, dall'utilizzo e dall'errato smaltimento delle sostanze tossiche utilizzate per la produzione. Questo fenomeno è così diffuso da meritare un acronimo che lo identifichi: Waste of Electric and Electronic Equipment (WEEE). L'inquinamento derivato dall'IT è diventato un problema sempre più sentito, infatti il 70% dell'inquinamento del suolo derivato da piombo, cadmio e mercurio è causato direttamente o indirettamente dall'IT. Il Green Computing si occupa quindi di ridurre l'utilizzo delle componenti inquinanti delle attrezzature IT, dell'ottimizzazione del packaging, dell'eco-etichettatura delle varie parti e del ricondizionamento e recupero dei diversi componenti. L'ultima accezione di Green IT si concentra sull'utilizzo dell'IT come strumento di misura e rilevamento di parametri "green", come il consumo energetico, la temperatura o i rifiuti tossici prodotti, per tutti i processi di business. Recenti studi [14] hanno rilevato che l'86% dei dipartimenti ICT

nel Regno Unito non è a conoscenza del peso delle proprie emissioni di CO2 nell'ambiente, e l'80% delle organizzazioni non è a conoscenza del dispendio energetico delle proprie attività. Una ricerca analoga del Dipartimento di Elettronica e Informazione del Politecnico di Milano [6] rivela che, su un campione di 140 piccole e medie imprese italiane, l'89% degli amministratori IT non conosce il consumo di energia delle proprie infrastrutture. Diviene quindi impossibile cercare di migliorare qualcosa di cui non si è a conoscenza; risulta ancora più diffcile che un responsabile IT sia incentivato a diminuire il consumo energetico dei suoi apparati, quando i relativi costi non afferiscono al suo budget. Per avere un IT più sostenibile deve quindi avvenire un cambiamento profondo nella cultura e nella gestione aziendale. L'impatto ambientale dei processi di un'azienda dovrebbe essere monitorato tramite Key Performance Indicator (KPI) appropriati. I KPI rilevati dovrebbero poi essere analizzati secondo opportune dimensioni quali: tempo, fase di lavorazione, prodotto, e successivamente rielaborati per entrare a far parte della Knowledge Base (KB) aziendale. Il Green Computing non si limita così solamente all'analisi dell'impatto ambientale dei sistemi IT, ma diventa anche studio dell'impiego dell'IT stesso per monitorare ed ottimizzare le prestazioni green di un'azienda. L'IT può quindi essere utilizzato in un duplice modo:

- può supportare la misurazione di parametri green tramite sensori o reti intelligenti capaci di raccogliere e analizzare i dati;

- può permettere l'analisi e la sintesi dei dati tramite strumenti di data mining per supportare le decisioni aziendali.

Per tutte queste ragioni l'interesse del mondo dell'IT nel Green Computing è in continuo aumento. Essere "green" apporta, come si è detto, molti benefici; un maggior consumo energetico comporta costi elevati e può portare ad aggravi fiscali. Inoltre, poichè la cultura green è in rapida diffusione, i

clienti sono maggiormente incentivati ad avere fornitori attenti ai problemi ambientali.

2.1.2 Un approccio multilivello

La propagazione dell'informazione richiede energia. Un bit, che rappresenta la minima quantità d'informazione, è associato allo stato di un sistema fisico (ad esempio un bit può rappresentare la carica di un insieme di elettroni o il campo magnetico su una porzione di disco), e per commutarlo è necessario cambiare lo stato del sistema con conseguente consumo energetico. Studi condotti al Massachussets Institute of Technology hanno stabilito il valore del lower bound del consumo di energia necessario alla commutazione di un bit da uno stato a un altro ad una certa velocità. Questo consumo minimo nei computer tradizionali arriva solamente a 10-16 Joule per eseguire l'operazione. Questi valori di consumo energetico sono ovviamente lontani dai consumi reali di un sistema IT. La differenza deriva dal fatto che la commutazione del bit è il livello più basso di un sistema informatico: tutti i livelli infrastrutturali superiori arrivano, infatti, a moltiplicare la singola energia di commutazione di un fattore trenta. In Figura 2.3 viene mostrata la suddivisione del consumo d'energia in un data center medio (la suddivisione può variare a causa di differenti carichi computazionali o caratteristiche hardware del sistema). Dal grafico in figura è chiaro come il problema dell'effcienza nel consumo energetico debba essere affrontato a diversi livelli infrastrutturali del data center (utilizzo della CPU, distribuzione del carico sui server, raffreddamento, etc.) poichè ogni componente contribuisce al consumo totale. É inoltre importante notare come sia fondamentale ottimizzare l'effcienza energetica dei livelli più bassi del sistema. Ad esempio, se il software che gestisce le commutazioni eseguite dal processore è ineffciente, il processore dovrà eseguire più operazioni; di conseguenza ci sarà una maggiore richiesta di memoria ed un ulteriore bisogno di raffreddare il sistema.

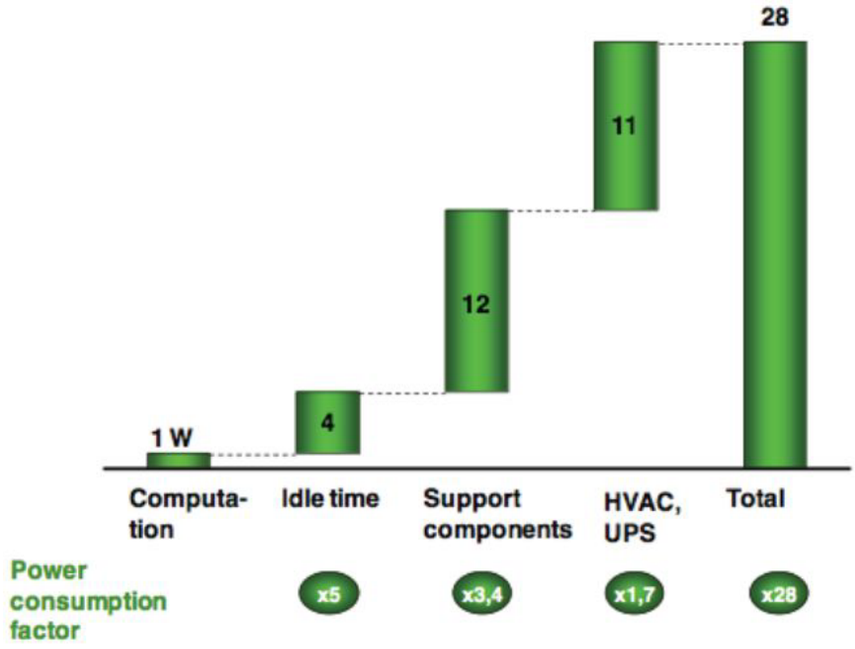

Figura 2.3: Ripartizione percentuale del consumo d'energia in un data center. Fonte: IBM (2007).

Gli sprechi energetici dei livelli inferiori si propagano quindi ai livelli superiori amplificandosi sempre più. Attualmente l'effcienza energetica di un sistema IT viene stimata al 50% dell'effcienza massima teorica ottenibile. In tabella 2.1 sono presentate alcune stime sull'effcienza dei vari livelli dei sistemi IT più comunemente utilizzati [14, 13]. Sono molti i fattori che contribuiscono all'ineffcienza dei vari livelli IT e molte sono le leve sulle quali si può agire per ottenere dei miglioramenti. A livello infrastrutturale le principali cause d'ineffcienza energetica sono i gruppi di continuità e i sistemi di condizionamento, che potrebbero essere regolati dinamicamente per poter raffreddare quando e dove necessario a seconda del carico di lavoro effettivo del data center. Periferiche e componenti ausiliari sono poi causa di dissipazione di energia a livello server. Cercare di ottimizzarne l'utilizzo può portare ad un buon guadagno di energia. Ad esempio è possibile regolare il funzionamento delle ventole secondo i bisogni di raffreddamento. Con que-

Livello	Efficienza energetica attuale stimata
Infrastruttura	50%
Sistema	40%
Server	60%
Microprocessore	0,001%
Software	20%
Rete	10%
Database	60%
Pratiche di utilizzo dell'IT	30%

Tabella 2.1: Efficienza energetica stimata rispetto all'effcienza massima teorica degli attuali sistemi IT.

sto metodo si può arrivare a risparmiare fino al 50% dell'energia dissipata. Altri miglioramenti si ottengono suddividendo la cache in segmenti alimentati singolarmente, o introducendo l'utilizzo di memorie allo stato solido in sostituzione degli hard disk tradizionali.

Sono i processori i componenti con l'efficienza energetica peggiore. I processori odierni sono ben lontani dal raggiungere l'efficienza teorica prevista dalle leggi della fisica quantistica; oltre al loro limite architetturale, inoltre, non sono utilizzati al massimo delle potenzialità e, di conseguenza, diventano inefficienti. É stato dimostrato che abbassando la frequenza di clock (underclocking) e passando da processori single-core a quad-core, si possono ridurre i consumi energetici relativi fino al 50Il software dei sistemi IT è molto complesso e implica diversi sottolivelli: dal sistema operativo al middleware fino all'application layer. Solitamente, durante la progettazione e lo sviluppo dei software di sistema, non si tiene in considerazione l'efficienza energetica, che non viene nemmeno annoverata nel trade-off fra i costi, le prestazioni e la qualità. Un nuovo filone di ricerca si occupa oggi di analizzare quali aspetti della qualità interna del software siano direttamente o indirettamen-

te collegati all'efficienza energetica. In questo ambiente teorico trova spazio questo lavoro di Tesi ed altri lavori conclusi ed attualmente in svolgimento all'interno del gruppo di Sistemi Informativi del Politecnico di Milano. La qualità dei dati a livello DB ha un forte impatto sull'efficienza energetica. Dati di scarsa qualità possono portare ad un maggior numero di transazioni e operazioni con un conseguente dispendio di energia. Le strutture dati dei database, i dbms, e i parametri, devono quindi essere ottimizzati affinch'è si raggiunga un risparmio complessivo di energia. É tuttavia fondamentale un cambiamento della cultura aziendale sia per gli utenti che per gli amministratori del sistema. Ogni persona coinvolta nel sistema IT deve essere correttamente istruita per comprendere il problema dell'efficienza energetica, poichè anche gli accorgimenti più semplici possono apportare miglioramenti sensibili. Ad esempio, l'acquisto di un'apparecchiatura IT costruita secondo parametri green può richiedere un investimento maggiore; considerando però i risparmi energetici ricavati rispetto al costo odierno dell'energia, l'investimento sarebbe ripagato, procurando successivamente ulteriori vantaggi economici.

2.2 Green software

Benchè il software non consumi direttamente energia, influisce profondamente sul dispendio energetico dell'intero sistema IT. Dai livelli più bassi a quelli più alti il consumo energetico si propaga e si moltiplica, come descritto nella sezione precedente. É per questo che intervenire sul software apportando modifiche per ottenere risparmi energetici può contribuire ad una notevole riduzione delle spese energetiche complessive del sistema IT. In questa sezione viene descritto l'impatto del software sul Green IT evidenziando lo sviluppo globale per l'applicazione del Green Computing nel settore dell'Information Tecnology.

2.2.1 Consapevolezza globale

Il Green IT è una realtà sempre più diffiusa nel mondo aziendale. L'adesione a questa metodologia comporta miglioramenti non solo per l'azienda, che riduce i costi e migliora la propria immagine verso il cliente, ma anche sull'ambiente in quanto la riduzione del consumo di energia elettrica porta ad una diminuzione delle emissioni di carbonio nell'atmosfera. Fino ad ora la maggior parte degli studi sul Green Computing sono stati focalizzati sul risparmio energetico dell'infrastruttura IT nel complesso, concentrandosi principalmente sui problemi di raffreddamento e gestione dell'energia. Occorre però concentrarsi sullo sviluppo di codice di qualità orientato al risparmio energetico permettendo di ottenere una notevole riduzione delle spese energetiche complessive del sistema IT, poichè, come già riportato, dai livelli più bassi a quelli più alti dell'architettura il consumo energetico si propaga e si moltiplica. Precedenti studi sull'argomento si sono focalizzati sul Low Power Software per i sistemi embedded[1].

Il Low Power Software si occupa di progettazione orientata al risparmio energetico per sistemi che hanno stringenti requisiti di consumo. La diminuzione del consumo, in questi dispositivi, permette di ridurre i limiti imposti dalla miniaturizzazione, dilatare i tempi tra le ricariche dei dispositivi, ed essere quindi competitivi sul mercato. I sistemi embedded sono caratterizzati anche da ristretti limiti di spazio di memoria a disposizione. Una metodologia utilizzata per ovviare al problema è la Code Compression, cioè la compressione del codice sorgente che riduce la quantità di memoria richiesta e, necessitando di un numero minore di accessi alla main memory, riduce i consumi e i dispendi energetici, diminuendo inoltre la dissipazione di ener-

[1] In informatica, con il termine sistema embedded, si identificano genericamente tutti quei sistemi elettronici a microprocessore, progettati appositamente per un'applicazione, ovvero non riprogrammabili dall'utente per altri scopi. Sono integrati direttamente nel sistema che controllano, dispongono quindi di una piattaforma hardware ad-hoc.

gia nel bus e nelle interconnessioni. Un altro metodo utilizzato per ottenere migliori performance ed efficienza computazionale è l'impiego di compilatori che ottimizzino il codice, servendosi di librerie focalizzate sulle performance che contengano algoritmi ottimizzati e instruction set volti a ridurre i consumi energetici richiesti. Anche una gestione efficiente dei dati può ridurre i costi dell'energia tramite la progettazione di algoritmi che ne minimizzino il movimento, suddividendo in gerarchie la memoria disponibile, mantenendo vicini i dati degli elementi da processare, e usando la cache memory in modo ottimale. Obiettivo della Context Awareness è invece quello di creare applicazioni che possano rispondere o adattarsi ai cambiamenti ambientali. Per ambienti fisici, ad esempio, sensori esterni possono generare eventi ai quali le applicazioni reagiscono, come in seguito alla variazione dell'intensità luminosa nell'ambiente circostante, o la variazione della temperatura. In un'ottica energetica, il software deve rispondere ai cambiamenti del sistema intraprendendo azioni atte a conservare energia. La ricerca sull'efficienza energetica si è finora concentrata sul software embedded, cercando di migliorare le performance e i consumi configurando dinamicamente i componenti a basso livello architetturale. Ciò permette di ridurre l'utilizzo complessivo della CPU e, quindi, il consumo energetico. Non molto è stato fatto però nel campo del green software design per applicazioni di largo impiego, nonostante sia stato mostrato come l'ottimizzazione energetica del software abbia ancora ampio spazio di miglioramento e, soprattutto, come benefici tangibili in termini di diminuzione dei costi dell'energia possano essere apportati. Molti domini applicativi offrono la possibilità di valutare un trade off tra Quality of Service (QoS) e risultato, per ottenere miglioramenti nelle performance e al contempo una riduzione del consumo energetico. L'implementazione di green software dovrebbe permettere al programmatore di modificare funzioni particolarmente costose e cicliche, mediante l'impiego di tecniche di programmazione energicamente più efficienti che offrano i medesimi risultati,

riducendo i carichi computazionali destinati al sistema.

2.3 Precalcolo e tabulazione in memoria

L'utilizzo di precalcolo e tabulazione di valori in memoria è una tecnica largamente impiegata nel settore della crittografia, allo scopo di rendere più efficiente l'impiego di onerosi algoritmi di calcolo, o di rendere possibili attacchi crittoanalitici in tempi ragionevoli. Questi attacchi, basati sulla ricerca esaustiva delle chiavi di decriptazione, richiedono grandi potenze di calcolo e tempi lunghi d'esecuzione. Quando lo stesso attacco è riprodotto diverse volte, può essere utile realizzare dapprima una ricerca esaustiva delle chiavi, memorizzando i risultati in memoria e, una volta eseguito il precalcolo, condurre gli attacchi istantaneamente in maniera simultanea. La tecnica della tabulazione è molto utilizzata anche per migliorare l'efficienza nelle performance di diversi algoritmi crittografici a chiave pubblica, che sfruttano ripetutamente il calcolo matematico di elevato a potenza. Il problema di ottenere un'esponenziazione veloce è molto importante per realizzare un'implementazione efficiente della maggior parte dei sistemi crittografici. Più in particolare, la tecnica del precalcolo dei risultati di una funzione e la loro tabulazione in memoria prende il nome di memoizzazione.

2.3.1 Memoizzazione

Il termine memoizzazione è derivato dal termine "memo function" coniato da Donald Michie (1968) [12] riferendosi al processo con cui una funzione può ricordarsi del risultato di una computazione precedente.

Definizione 2.1 (Memoizzazione)**.** *La memoizzazione è una tecnica di programmazione che consiste nel salvare in memoria i valori restituiti da una funzione in modo da averli a disposizione per un riutilizzo successivo senza doverli ricalcolare.*

Memoizzazione, che significa "mettere in memoria", è una tecnica caratteristica della programmazione dinamica. Spesso questo termine viene confuso con "memorizzazione" che, sebbene descriva lo stesso procedimento, ha un significato più ampio. Una funzione può essere "memoizzata" soltanto se soddisfa alcuni requisiti che corrispondono alla definizione di funzione pura (vedi sezione 2.4). L'idea base è quella di memorizzare una tabella di coppie `<input,risultato>` e, per ogni invocazione della funzione, di controllare nella tabella se esiste già la entry corrispondente. Per spiegare il concetto in modo più approfondito usiamo l'esempio della funzione in pseudo-codice `int fibonacci(int n)`, dove n è la posizione del numero che vogliamo ritornato all'interno della serie di Fibonacci. Un possibile algoritmo potrebbe essere:

```
int fibonacci(int n){
        if (n==1 || n==2) return 1;
        else return fibonacci(n-1) + fibonacci(n-2);
}
```

Come possiamo notare `fibonacci(n)` viene chiamata ricorsivamente, se applichiamo la tecnica della memoizzazione a `fibonacci(n)` il codice della funzione diverrà:

```
int fibonacci(int n){
        int result = lookupTable(n);
        if(result risulta attendibile) return result;
        else{
                if (n==1 || n==2) return 1;
                else{
                        result = fibonacci(n-1)
                                        + fibonacci(n-2);
                        saveResult(n, result);
                        return result;
                }
```

```
        }
}
```

La funzione `fibonacci(n)` possiede una tabella in cui salvare i risultati delle sue computazioni. Per prima cosa, all'ingresso della funzione, verrà fatta una ricerca nella tabella per controllare se il dato richiesto, per l'input in esame, è già presente (ossia è già stato calcolato in una computazione precedente), se il risultato ottenuto dalla `lookupTable` è un valore attendibile (ossia se non è un valore definito come impossibile per la funzione) allora la funzione termina restituendo il risultato pescato dalla tabella. In caso contrario la funzione viene eseguita normalmente e, prima che questa ritorni il valore richiesto, provvede a salvarsi il risultato appena calcolato per usi futuri all'interno della tabella di salvataggio tramite la funzione `saveResult` che associa, nella tabella della funzione, il risultato appena calcolato all'input in esame.

2.4 Funzioni pure

La definizione di funzione pura può essere derivata dal concetto di funzione matematica, ossia un mapping uno a uno tra input e output [10].

Definizione 2.2 (Metodo puro)**.** *Un metodo di un programma è considerato puro se soddisfa due proprietà:*

- *non genera side-effect, ossia la sua esecuzione non crea nessun effetto visibile se non quello di generare un risultato;*
- *è deterministico, cioè il comportamento del metodo dipende solo dagli argomenti passati come parametri nella sua invocazione.*

Più in dettaglio possiamo andare a definire le proprietà sopra citate: assenza di side-effects e determinismo della funzione.

Definizione 2.3 (Metodo libero da side-effect)**.** *Un metodo è libero da side-effect se i soli oggetti che il metodo modifica sono creati come parte dell'esecuzione del metodo.*

Questa definizione permette al metodo di creare e modificare nuovi oggetti ma non permette di modificare tutti i valori osservabili dall'esterno del metodo. In aggiunta è necessario che un metodo puro, come da Definizione 2.2, non causi nessun side-effect fuori dall'ambiente del sistema, ad esempio non potrà scrivere su file, stampare messaggi sulla console o comunicare in rete. Una relazione matematica è considerata funzione se ogni diverso input è associato ad un singolo specifico output; per due valutazioni una funzione matematica con gli stessi input genererà lo stesso risultato. Il requisito di determinismo per un metodo è analogo a quello delle funzioni matematiche.

Definizione 2.4 (Determinismo di un metodo)**.** *Il determinismo di una metodo è una proprietà parametrica: data la definizione di cosa si intende per equivalenza tra parametri, un metodo è deterministico se tutte le chiamate con argomenti equivalenti ritornano risultati indistinguibili tra di loro.*

Il risultato deve dipendere solo dagli argomenti passati come parametri al metodo e non da altri stati globali (ad esempio, l'ora, il giorno corrente oppure il valore di un attributo globale della classe). Per questa proprietà dobbiamo definire il concetto di uguaglianza tra parametri nel caso di un linguaggio di programmazione. Se nel caso dei tipi primitivi l'uguaglianza può essere fatta semplicemente in base al valore che essi assumono, per quanto riguarda le classi ci troviamo di fronte ad altre questioni: possono due chiamate ad un metodo essere equivalenti se gli argomenti hanno gli stessi valori ma diversi alias? E se posseggono gli stessi alias ma risiedono in diversi indirizzi di memoria? Non c'è una risposte definitiva alle domande sopra proposte perchè per ogni problema che dobbiamo affrontare sarà ottimale una scelta piuttosto che l'altra.

In letteratura è possibile trovare molti lavori incentrati sull'analisi di programmi con lo scopo di individuare funzioni pure o più semplicemente di individuare funzioni senza side-effect. Prima dell'avvento di Java la situazione era più semplice (ad esempio, con i linguaggi ad oggetti, specialmente in Java, è ora possibile l'override di metodi) e le analisi venivano svolte direttamente sul codice sorgente delle applicazioni. Adesso con Java le analisi si sono specializzate su due fronti: analisi statica o analisi dinamica del bytecode. Con l'analisi statica vengono riconosciute le funzioni che sono sempre pure, mentre con l'analisi dinamica (durante l'esecuzione del programma che vogliamo analizzare) possiamo andare a definire più stadi di purezza e quindi riuscire a individuare delle funzioni anche che sono pure solo in per determinati parametri di input. Lo studio del programma si è spostato dal codice sorgente per i linguaggi diversi da Java al bytecode Java. Il bytecode permette l'analisi grazie alle istruzioni presenti al suo interno pur essendo di più basso livello rispetto al codice sorgente Java. Per una più approfondita trattazione circa l'analisi del bytecode per la scelta delle funzioni pure si rimanda a [9].

Capitolo 3

Architettura del framework

In questo capitolo verrà illustrato l'approccio utilizzato per la soluzione del problema relativo alla riduzione del consumo energetico in programmi Java tramite l'utilizzo della memoizzazione di funzioni pure. Il target di utilizzatori del progetto è un'utente che vuole analizzare il software Java che possiede, suo o di terze parti, e vuole migliorarne il consumo energetico. Per essere usufruibile da un numero maggiore di utenti l'applicazione permette di analizzare il software Java direttamente dal bytecode (file .class), in questo modo anche l'utenza che non disponesse del codice sorgente può comunque analizzare e modificare il proprio codice riducendone il consumo energetico.

In seguito all'analisi del problema del consumo energetico del software si procede alla progettazione di un'architettura software generale che consenta la modifica di un'applicazione con l'obiettivo di ottenere una riduzione del consumo energetico per la propria esecuzione e quindi la realizzazione e l'integrazione di tutti i moduli necessari per questo compito.

Il punto di partenza è rappresentato dallo studio effettuato precedentemente in [2] e [1], che ha portato alla luce come è ripartito il consumo energetico per l'esecuzione del software su una macchina. Il componente che consuma maggiormente energia, ed il cui consumo è dipendente dalle

istruzioni eseguite, è il processore, mentre il consumo di memoria RAM è indipendente dal suo utilizzo e rimane costante nel tempo. Infatti la RAM consuma lo stesso quantitativo di energia a causa del continuo refresh delle celle di memoria, a prescindere da come viene utilizzata per il salvataggio dei dati. Da questo interessante fattore è nata l'idea di sfruttare il più intensamente possibile la memoria RAM inutilizzata memorizzando al suo interno i risultati principali di alcune funzioni usate intensamente o computazionalmente pesanti presenti nel software. In questo modo l'applicazione, durante la sua esecuzione, eviterà di calcolarsi il risultato per alcuni dati di ingresso con conseguente riduzione dell'utilizzo della CPU, il cui consumo energetico varia in base all'utilizzo, a favore dell'utilizzo della RAM , il cui consumo risulta costante.

Nella Sezione 3.1 verrà descritta l'architettura generale all'interno della quale si colloca la Tesi e verrà analizzata nella sua interezza. Nella Sezione 3.2 verranno descritti i moduli principali presenti nell'architettura e la loro cooperazione. Nella Sezione 3.3 verranno riportati i vari problemi riguardo alla gestione dei dati nella memoria per la memoizzazione delle funzioni e le rispettive soluzioni individuate.

3.1 Architettura

L'architettura è composta da quattro parti principali (Figura 3.1) :

- Analisi statica del bytecode dell'applicazione per la scoperta delle funzioni pure in esso contenute e la loro successiva modifica,
- Esecuzione del meta-modello *Decision maker*,
- Analisi delle informazioni per reportistica,
- Salvataggio delle tabelle dati in memoria su file XML.

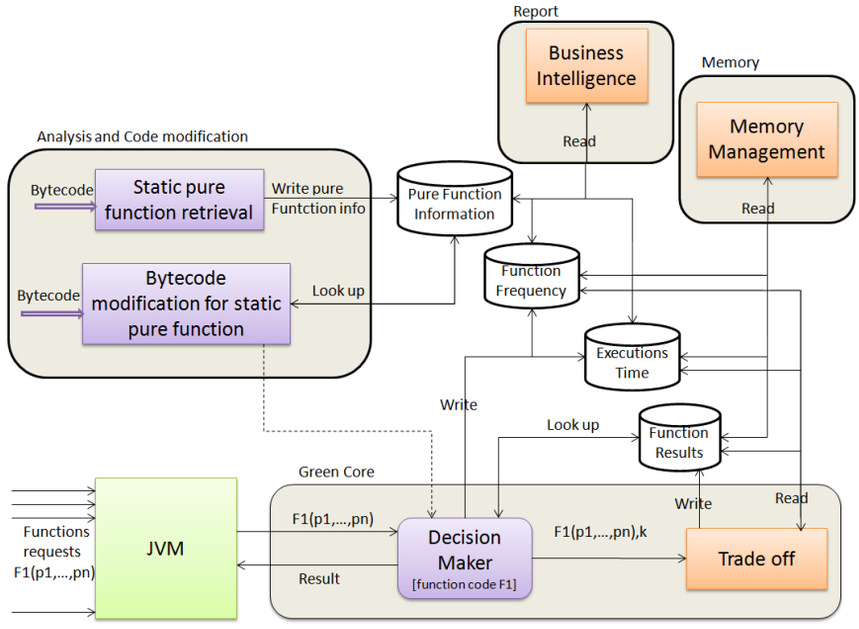

Figura 3.1: Architettura

L'analisi statica analizza l'intero bytecode dell'applicazione alla ricerca di funzioni pure. Le informazioni (nome del metodo, signature, numero di istruzioni bytecode contenute, ecc.) riguardanti ogni singola funzione pura vengono quindi salvate in memoria. Quando tutto il codice è stato analizzato ed è stato deciso quali funzioni pure è utile tabulare, può essere eseguito il modulo *Bytecode modification* che si occupa della modifica delle funzioni pure inserendo dentro di esse il codice necessario per l'esecuzione del meta-modulo *Decision maker*. La scelta delle funzioni da tabulare è fatta in base alle condizioni di purezza viste nella Sezione 2.4 Capitolo 2 e in base all'efficacia ottenuta dalla loro memoizzazione. Il parametro dell'efficacia verrà trattato invece nel Capitolo 4 Sezione 4.2.2.

Durante l'esecuzione dell'applicazione la JVM analizza ed esegue normalmente il bytecode. Nel caso di esecuzione di una funzione pura, la JVM non carica il codice originario della classe in cui si trova la funzione richiesta ma carica il codice che è stato modificato nella precedente parte dal modu-

lo *Bytecode modification* (eseguito *una-tantum* per applicazione). Il metodo, prima dell'esecuzione normale del codice della funzione, si occupa di effettuare un'operazione di *look up* sulla tabella relativa alla funzione in esecuzione con i parametri effettivi per cui è stata richiamata: se esiste nella tabella il risultato relativo alla chiamata della funzione per quei determinati input la funzione restituisce direttamente questo valore, altrimenti essa continua la sua esecuzione normalmente. Prima dell'istruzione *return* viene chiamata una funzionalità del modulo di *Trade off* che si occupa di decidere se salvare o meno il risultato dell'esecuzione nell'apposita tabella in base alle sue logiche interne per il risparmio di energia e per l'ottimizzazione dell'uso della memoria.

La parte relativa alla reportistica gestisce l'analisi dei dati relativi alle esecuzioni delle funzioni (tempi di esecuzione e frequenze di chiamate delle funzioni pure) e dei dati relativi al codice, identificati nella prima parte di analisi statica, fornendo, tramite tabelle e grafici, delle indicazioni visive sull'effettivo risparmio energetico reso dall'uso della soluzione adottata e sulle possibili modifiche per aumentarne ancora le potenzialità.

La parte di gestione della memoria effettua infine il salvataggio delle tabelle presenti in memoria su file XML secondo uno specifico DTD che identifica i tag in cui saranno memorizzate le istruzioni.

3.2 Moduli

In questa sezione verrà spiegato in maniera approfondita il comportamento dei singoli moduli dell'architettura generale descritta nella sezione precedente. Inoltre per ogni modulo verrà descritta l'interazione che ha con gli altri in modo da permettere il funzionamento dell'approccio. I moduli di *Pure function retrieval*, *Bytecode modification* e il meta-modulo *Decision maker* (sviluppati in [9]) e il modulo di *Business intelligence* (sviluppato in [8])

verranno illustrati nelle sezioni che seguono, mentre il modulo di *Trade off* verrà ampiamente trattato nei capitoli successivi.

3.2.1 Pure function retrieval

Il modulo di *Pure function retrieval* si occupa di eseguire un'analisi statica del bytecode dell'applicazione in esame. In accordo a specifiche regole, definite in [9], determina se un metodo è puro o no e, se è considerato puro, salva la signature (nome della funzione, tipo del valore restituito, numero e tipo dei parametri passati in ingresso) ed altre informazioni, come ad esempio il numero di istruzioni di bytecode contenute, il numero di chiamate a funzioni interne, ecc., sul database per uno studio statistico successivo (Sezione 3.3). In questo caso è possibile salvare i dati sul database perchè i tempi di accesso alle informazioni non sono un valore critico per l'esecuzione dei moduli interessati. Un'interfaccia grafica permette l'interazione con le API del modulo per l'analisi del bytecode dell'applicazione che vogliamo esaminare. L'analisi del codice può avvenire in due modi:

- partendo da un metodo di una classe selezionata (solitamente il metodo di partenza è il metodo `main` dell'applicazione),
- selezionando un insieme di classi e analizzando tutti i metodi al loro interno.

Il modulo di *Pure function retrieval* è il punto di partenza per lo studio di ogni applicazione che vogliamo ottimizzare dal punto di vista del consumo energetico. Infatti, senza aver individuato quali sono le funzioni ottimali per permettere la memoizzazione, ossia le funzioni pure presenti all'interno dell'applicazione, non è possibile andare a modificare il bytecode. L'utente del framework interagendo con l'interfaccia grafica del modulo di *Pure function retrieval* seleziona il metodo o le classi da cui iniziare l'analisi, in base al metodo di ricerca scelto, e ottiene come risultato le funzioni pure

trovate mostrando le loro caratteristiche principali e un punteggio che ne identifica la dimensione in base alle istruzioni bytecode che lo compongono. L'utente è quindi messo in condizione di effettuare la scelta più opportuna per selezionare quali metodi andare a modificare per inserire il meta-modulo *Decision Maker*. Per permettere la modifica delle funzioni, vegono salvate sul database le informazioni necessarie per identificare univocamente i metodi prescelti, organizzati per applicazioni analizzate (Sezione 3.3), che saranno poi utilizzate dal metodo di *Bytecode modification*.

Dopo questa parte di analisi del codice viene eseguito il modulo *Bytecode modification* che modifica il bytecode delle funzioni pure trovate.

3.2.2 Bytecode modification

Il modulo di *Bytecode modification* controlla sul database le informazioni che identificano le funzioni pure trovate e, per ognuna di queste, esegue la funzione di modifica del bytecode per l'inserimento delle istruzioni bytecode che permettono l'esecuzione del meta-modulo *Decision maker*. Una funzione viene identificata tramite una stringa `methodId` che è composta da tre parti principali:

- `className`, comprensivo di classpath della classe (a titolo di esempio per la classe Integer, la stringa `className` sarebbe `java.lang.Integer`),
- `methodName`, che identifica il metodo all'interno della classe className (sempre per l'esempio della classe `Integer` potremmo avere `methodName toBinaryString`),
- `signature`, ovvero la stringa che identifica la signature del metodo `methodName` (come descritta nella Sezione 2.5.1 di [9]).

Per fare un'esempio di un `methodId` generato a partire dalla funzione Fourier Series e utilizzata durante i test: `section2_MOD.series.SeriesTest; fourierTransform;(IDDI)D`.

Anche questo modulo offre l'utilizzo di un'interfaccia grafica che permette un più semplice uso delle API. L'utente che usa l'interfaccia ha la possibilità di selezionare l'applicazione di cui vuole modificare i metodi puri presenti. Selezionata l'applicazione, vengono mostrati i `methodId` dei metodi precedentemente salvati nel database con il modulo di *Pure function retrieval* e si procede con la modifica degli stessi inserendo il meta-modulo *Decision maker*. L'inserimento del meta-modulo, così come per l'analisi dell'applicazione, avviene a livello di istruzioni bytecode e non richiede quindi una compilazione ulteriore del codice prodotto.

3.2.3 Decision maker

Il modulo di *Decision maker* è un meta-modulo, nel senso che non è un modulo a se stante bensì consiste nella modifica del codice della funzione per mezzo dell'introduzione dei blocchi di *Look up* e di *Trade off*, grazie all'utilizzo del modulo di *Bytecode modification*. Il meta-modulo *Decision maker* rappresenta le istruzioni bytecode che permettono il funzionamento e l'integrazione generale di tutti i moduli dell'approccio proposto. Quindi effettivamente il modulo di *Bytecode modification* si occupa dell'inserimento di due blocchi di istruzioni bytecode: un blocco per la *Look up* sulla tabella relativa alla funzione in esame ed un blocco per la chiamata al modulo di *Trade off*.

Il blocco di *Trade off* inserito in questa fase ha come unico compito quello di controllare se è disponibile ulteriore spazio in memoria per il salvataggio delle coppie parametri di ingressi e risultato. Come vedremo nei capitoli successivi sono state introdotte nuove funzionalità a tale componente per ottenere un minor consumo energetico per mezzo di un migliore utilizzo della memoria.

Per esempio presa una funzione pura `returnType className.methodName(params)` implementata come nel Listato 3.1 (in pseudo-codice),

la relativa funzione modificata dopo l'inserimento del meta-modulo *Decision maker* risulta come lo pseudo-codice nel Listato 3.2.

Listing 3.1: Funzione methodName(params) della classe className

```
public class className {

        ReturnType methodName(Parameters p){
                ReturnType result = null;

                //CODICE ORIGINALE
                        //effettua il calcolo di result
                //CODICE ORIGINALE

                return result;
        }
}
```

Come è possibile vedere, evidenziati in colori diversi possiamo notare i blocchi di codice relativi a:

- verde - blocco di *Look up* in cui viene effettuata la ricerca nella tabella della funzione per controllare se è già stato tabulato il risultato per i suoi parametri di ingresso,
- blu - blocco di codice originario della funzione,
- arancione - blocco di *Trade off* gestisce la memoria controllando se c'è spazio sufficiente per un nuovo inserimento.

Listing 3.2: Funzione methodName(params) della classe className modificata

```
public class className {

  ReturnType methodName(Parameters p){
    ReturnType result = null;
```

```
    //BLOCCO DI LOOK UP
      //controlla in memoria se e' presente la coppia
      //parametri di ingresso e risultato
    //BLOCCO DI LOOKUP

    //CODICE ORIGINALE
      //effettua il calcolo di result
    //CODICE ORIGINALE

    //BLOCCO DI TRADE OFF
      //controlla se e' disponibile spazio in memoria
      //per il salvataggio della nuova coppia
      //parametri di ingresso e risultato appena
      //calcolati
    //BLOCCO DI TRADE OFF

    return result;
  }
}
```

Il flusso di esecuzione della funzione dopo l'inserimento del meta-modulo *Decision maker* risulta come riportato in Figura 3.2.

I blocchi del diagramma di flusso riprendono i colori usati in precedenza per evidenziare nuovamente come sono distribuite le operazioni all'interno della funzione.

Nella sezione successiva viene spiegato più nel dettaglio il funzionamento del blocco di *Look up*. La funzione del blocco di *Trade off* allo stato attuale consiste solo nel controllare che non si ecceda la dimensione massima della memoria. Si rimanda ai capitoli successivi l'analisi approfondita di tale blocco relativa alle modifiche apportate da questo lavoro di Tesi.

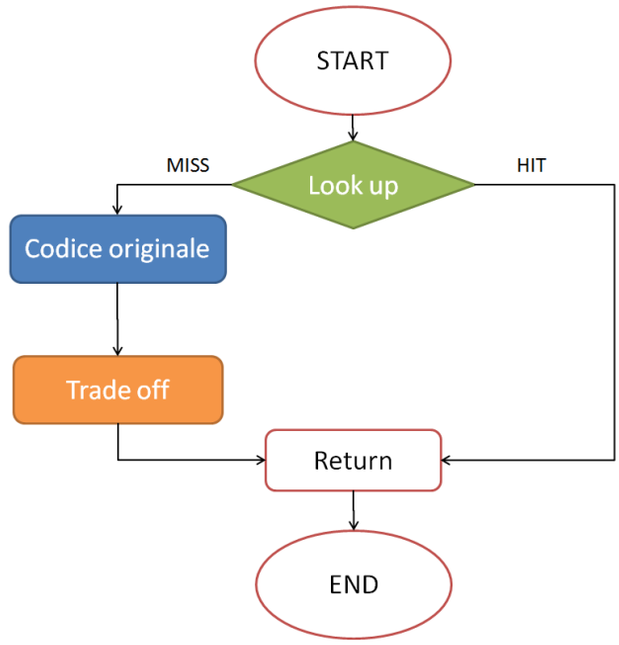

Figura 3.2: Flusso di esecuzione per una funzione dopo l'inserimento del meta-modulo Decision maker.

Blocco di Look up

Il blocco di *Look up* si occupa di effettuare la ricerca sulla tabella della funzione per controllare se è presente il risultato precalcolato per gli input dell'attuale esecuzione della funzione. In particolare il blocco effettua le seguenti operazioni primarie:

1. crea la stringa `methodId` usata successivamente per effettuare la *Look up* sulla tabella,
2. crea un array di `Object` contente i valori dei parametri della funzione per l'attuale esecuzione e lo usa per instanziare un oggetto di tipo `TOParameters`, anch'esso richiesto per la *Look up*,
3. effettua una *Look up* sulla tabella relativa alla funzione in esecuzione per scoprire se esiste il risultato per il metodo, identificato dalla stringa `methodId`, eseguito dati i parametri in ingresso salvati in `toparam`,

4. se la *Lookup* fa un HIT:

   (a) aggiorna la frequenza di utilizzo della funzione con id `methodId` chiamata con parametri `toparam`,

   (b) il metodo ritorna il corretto valore prelevato dalla memoria per la funzione e i parametri con la quale è stata chiamata ottenuto dalla *Look up* precedente.

Per una questione di debugging, e quindi da utilizzare solo in fase di test a causa dell'overhead introdotto dalle operazioni aggiuntive, è possibile inserire all'interno del blocco ulteriori istruzioni che permettono di avere più informazioni riguardo i tempi di esecuzione e il numero di hit o miss ottenuti su ogni funzione salvata in memoria.

3.2.4 Business intelligence

Il modulo crea i report riguardo la memoria occupata dall'applicazione, il grafo dell'uso di memoria e le possibili modifiche all'hardware (ed esempio l'inserimento di un nuovo modulo di RAM) per incrementare ancora di più la possibilità di risparmio di energia grazie alla tabularizzazione dei risultati. L'obiettivo principale della *Business intelligence* è di valutare i dati e fornire una rappresentazione grafica in modo da aiutare l'utente a determinare efficientemente l'architettura migliore per l'applicazione in esame. Il modulo presenterà alcune valutazioni sulla memoria usata e sul tempo di esecuzione riportando un consumo energetico teorico ottenuto considerando un modello dei componenti di memoria e CPU. Inoltre fornisce la possibilità di stimare il consumo energetico con appositi cambiamenti dal punto di vista hardware (come ad esempio l'aggiunta di nuovi banchi di RAM alla macchina).

3.3 Gestione dei dati

Le informazioni necessarie per l'esecuzione dell'applicazione, come visibile dall'architettura in Figura 3.1, sono memorizzate in quattro strutture dati: *Pure Function Information*, *Functions Frequency*, *Time Executions* e *Function Results*. Le strutture dati necessarie per il salvataggio possono essere di tipi diversi in quanto le informazioni contenute sono accedute in modalità diverse. Per quanto riguarda le informazioni relative a *Pure Function Information*, possono essere salvate in un database o su un file in quanto sono dati il cui tempo di accesso non influisce sull'esecuzione dell'applicazione; per quanto riguarda *Functions Frequency*, *Time Executions* e *Function Results* invece, esse devono necessariamente essere salvate in memoria in quanto un accesso al database durante l'esecuzione dell'applicazione renderebbe vano l'uso dell'architettura per quanto riguarda sia i tempi di esecuzione e il consumo energetico. In particolare poi, le informazioni riguardo la frequenza, il tempo di esecuzione ed i risultati sono gestiti con una unica classe, chiamata *MemoryManagement*, illustrata in Sezione 3.3.2. É necessario tenere in considerazione la necessità di salvataggio sul database o su file di queste tabelle per evitare problemi di perdite di dati dovute a problemi hardware, in quanto, essendo salvati in RAM, anche un semplice riavvio della macchina porterebbe alla perdita totale dei dati. Nelle sezioni successive verranno analizzate più in dettaglio le strutture dati precedentemente citate.

3.3.1 Pure Function Information

Le informazioni che identificano le funzioni pure e i dati che le caratterizzano singolarmente sono salvate sul database (Figura 3.3). Le informazioni che vengono salvate in queste tabelle del database sono utilizzate dai moduli di *Pure function retrieval* e di *Bytecode modification*.

Con la tabella application vengono salvati: `applicationName`, ossia il nome dell'applicazione analizzata (inserita dall'utente nell'apposito campo

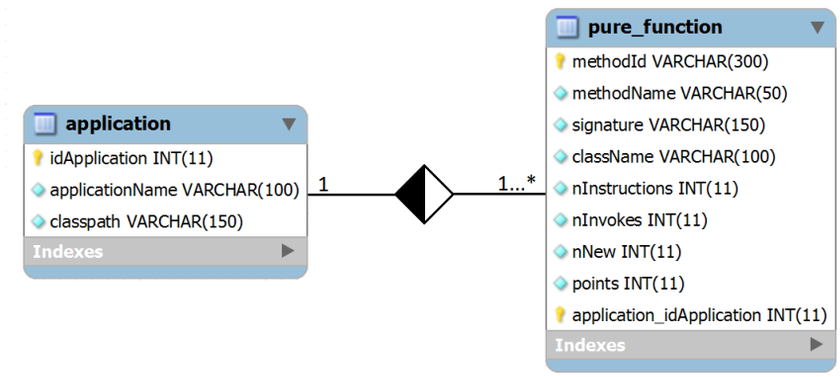

Figura 3.3: Tabelle del database per la memorizzazione delle informazioni delle funzioni pure.

dell'interfaccia grafica del modulo di *Pure function retrieval*); `classpath`, ossia il il percorso all'interno del filesystem per raggiungere i metodi all'interno delle classi salvate nella tabella di `pure_function`. Con la tabella pure function vengono invece salvati i dati dei metodi puri che vogliamo andare a modificare identificati univocamente dal methodId, ossia `className;methodName;signature`. Inoltre la tabella memorizza altre informazioni riguardanti il metodo puro, come ad esempio, `nInstructions` e `points` che sono rispettivamente il numero di istruzioni bytecode che compongono il metodo e il punteggio del metodo. Il punteggio viene calcolato per mezzo di una stima dei costi delle istruzioni bytecode [15] e viene utilizzato per valutare se una funzione è *computation intensive* o meno.

3.3.2 Memory Management

Per quanto riguarda le tabelle di memorizzazione delle frequenze, dei tempi di esecuzione e dei risultati delle funzioni pure come detto precedentemente, esse non possono essere salvate in un database durante l'esecuzione dell'applicazione perchè altrimenti questo andrebbe a compromettere il guadagno ottenuto dall'uso dell'architettura sia dal punto di vista temporale che dal

punto di vista dei consumi energetici. É stato scelto quindi di tenere le informazioni necessarie all'esecuzione dell'applicazione in memoria RAM e di salvarne il contenuto solo nei periodi di inattività su dei file XML per non aver problemi nel caso di guasti al sistema che potrebbero comportarne una perdita. Nelle prossime sezioni saranno analizzati più in dettaglio i due tipi di salvataggio dei dati utilizzati.

Salvataggio su file Xml

Le tabelle devono essere memorizzate anche su un dispositivo di memoria non volatile per evitare perdita dei dati a causa di qualsiasi inconveniente tecnico, come ad esempio un guasto hardware. In questo caso è molto più semplice il salvataggio dei valori su file invece del salvataggio su database a causa della natura delle funzioni pure di cui vogliamo salvare le informazioni. Il file XML ha una struttura che ricalca la struttura presente in memoria, mantenendo al suo interno i dati riguardanti la memoria RAM e poi una parte riguardante le funzioni identificate da `methodName` e signature con i dati di occupazione di memoria, priorità e frequenza. Anche il salvataggio su file XML risulta oneroso per l'applicazione e quindi andrebbe effettuato raramente. Per dei salvataggi durante l'esecuzione dell'applicazione sarebbe utile anche solo il dump della memoria su file, molto più semplice e con meno consumi energetici e temporali.

Salvataggio in memoria

Durante l'esecuzione i dati vengono memorizzati in memoria RAM per un accesso più veloce e meno costoso dal punto di vista energetico, infatti il salvataggio dei dati su database o su Hard Disk risulterebbe inefficiente e addirittura controproducente sia dal punto di vista temporale che dal punto di vista del consumo energetico. Per la gestione della memoria viene usata la

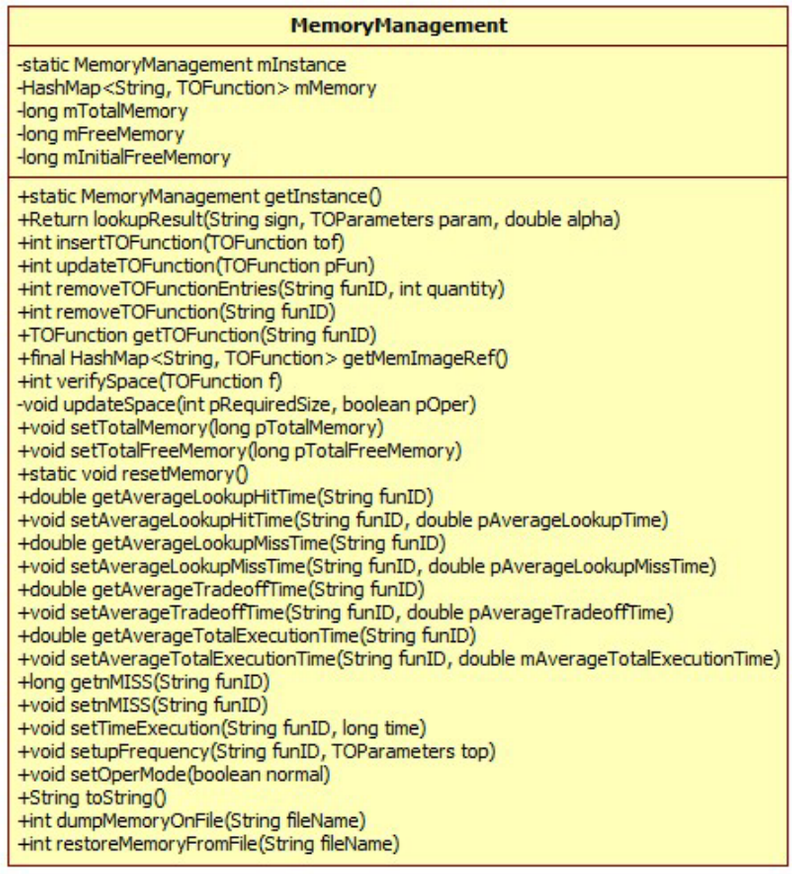

Figura 3.4: La classe MemoryManagement.

classe *MemoryManagement* che fornisce le API necessarie sie per la gestione della RAM che dei file XML.

In Figura 3.4 viene riportata la classe `MemoryManagement` che racchiude principalmente un `HashMap<String,TOFunction> mMemory`, che gestisce le coppie `methodId` e relativa `TOFunction`.

Internamente alla `TOFunction` invece è presente un oggetto della classe `MemoizationHashSFArr`, che sarebbe un `Hashmap` modificato. L'oggetto `mValues` appartenente alla classe `MemoizationHashSFArr` gestisce le coppie `TOParameters`, ossia i valori dei parametri passati per ogni chiamata della

funzione in oggetto, e il relativo valore di ritorno salvato in `Return`. Le principali differenze rispetto ad un `Hashmap` classico e il suo utilizzo verranno analizzate nel Capitolo 5.

Capitolo 4

Modello per l'allocazione dinamica della memoria

Nel seguente capitolo verranno analizzate le caratteristiche del modello per l'allocazione dinamica della memoria, detto modello del *Trade off*, che porterà alla creazione del blocco di *Trade off*, utilizzato all'interno dell'architettura di memoizzazione. Inizialmente verranno presentate nella Sezione 4.1 le caratteristiche generali che riguardano il modello del *Trade off*. Nella Sezione 4.2 sarà introdotto il concetto di efficacia, fondamentale per valutare la bontà del paradigma di memoizzazione. Infine nella Sezione 4.3 verranno analizzate nel dettaglio le due fasi in cui opera il modello.

4.1 Caratteristiche del modello del Trade off

Il modello del *Trade off* è responsabile della corretta gestione della memoria infatti, avendo più di un metodo memoizzabile e un elevato numero di inserimenti possibili, la memoria, in quanto limitata, diventa un fattore critico. Per fare in modo che i metodi che beneficiano maggiormente della memoizzazione abbiano più spazio in memoria di quelli scarsamente memoizzabili o

di quelli per cui la memoizzazione risulta svantaggiosa, si fa uso del modello.

Oltre a svolgere questo compito il modello del *Trade off* viene anche utilizzato per controllare che la somma delle zone di memoria concesse ai vari metodi non ecceda il tetto massimo di memoria utilizzabile e offre anche la possibilità di rimuovere da quest'ultima inserimenti che non apportano alcun beneficio.

Facendo riferimento all'architettura di memoizzazione (Figura 3.1 Sezione 3.1) il blocco di *Trade off*, costruito sul modello, si colloca all'interno del *Green core*. In seguito alla sua chiamata verrà deciso se salvare o meno la coppia parametri di ingresso e risultato in *Function Results* e inoltre verrà aggiornato il campo relativo alla frequenza delle funzioni in *Function Frequency*.

Per quanto riguarda l'introduzione del meta-modulo *Decision maker* vista nel capitolo precedente Sezione 3.2.3 la Figura 4.1 anticipa i cambiamenti apportati al blocco di *Trade off* che verranno analizzati in questo capitolo e nel Capitolo 5.

4.2 Valutazione dell'efficacia del Trade off

Quando si parla di corretta gestione della memoria si intende che l'utilizzo del *Trade off* serve a minimizzare il consumo energetico. Affinché ciò avvenga è necessario che l'energia consumata in fase di esecuzione del metodo da parte del processore sia minore di quella consumata per attuare il paradigma della memoizzazione. Per ottenere un consumo minore la memoizzazione deve andare a buon fine il maggior numero possibile di volte: tale valore è espresso dall' *hit rate* α .

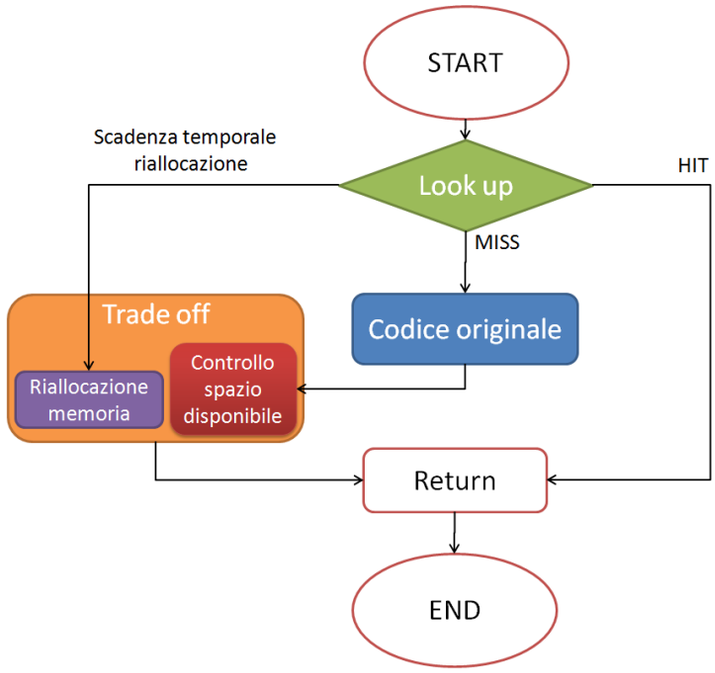

Figura 4.1: Flusso di esecuzione per una funzione dopo l'inserimento del meta-modulo Decision maker.

4.2.1 Hit rate

L' *hit rate* è la percentuale di volte che l'operazione di *Look up*, descritta nel Capitolo 3, trova in memoria la coppia parametri di ingresso e risultato. Come vedremo più avanti sono stati utilizzati due tipi di *hit rate*: uno che fa uso di un'approssimazione per mezzo di una gaussiana, l'altro invece che ne trova il valore effettivo calcolato a run time per mezzo di due contatori *#hit* e *#miss*, che rappresentano rispettivamente il numero di volte che la *Look up* va a buon fine e il numero di volte che fallisce.

Prendendo in considerazione il secondo valore di *hit rate* si potrebbero verificare casi in cui a fronte di un aumento di *#miss* sia necessario l'intervento del processore che esegue sia la funzione sia i moduli aggiuntivi di *Look up* e di *Save result*. Tutto ciò provoca un considerevole aumento del consumo energetico.

L' hit rate inoltre, dipende direttamente dalla quantità di memoria concessa alla singola funzione, dalla varianza dei parametri in ingresso e dalla loro precisione. La precisione è relativa hai parametri di ingressi e consiste nella minima distanza possibile tra due valori contigui. Per quanto riguarda la varianza, a parità di precisione, più basso è il suo valore maggiore è l'*hit rate*, infatti in questo caso il numero dei possibili ingressi è ridotto; essendo ridotto il numero di ingressi sarà possibile salvarne il valore in memoria e ottenere così un numero elevato di hit. Ovviamente all'aumentare della precisione richiesta anche avendo una varianza molto bassa il numero di ingressi da salvare è comunque elevato. Invece considerando la memoria disponibile se il suo valore è elevato allora è possibile salvare molte coppie parametri di ingresso e risultato, ottendo così un aumento della probabilità di trovare l'ingresso cercato.

4.2.2 Il modello del Trade off

Il funzionamento del modello del *Trade off* consiste in una suddivisione della memoria tra i vari metodi utilizzando come metrica il guadagno G, espresso come funzione inversa dell'efficacia η. L'efficacia del metodo consiste nel valutare di quanto diminuisce il consumo energetico nel caso di utilizzo del paradigma della memoizzazione rispetto al consumo dovuto all'esecuzione del metodo da parte del processore. Indichiamo con P la potenza consumata utilizzando la memoizzazione e con P_e la potenza consumata dal processore durante l'esecuzione dei metodi. L'efficacia è espressa dal rapporto di queste due potenze:

$$\eta = \frac{P}{P_e} \tag{4.1}$$

Al posto dell'energia utilizziamo i tempi di memoizzazione (T) e di esecuzione (T_e), sia per semplicità nelle misure, sia perché nelle applicazioni *computation intensive* l'energia consumata risulta essere direttamente colle-

gata al tempo necessario per la computazione. Esprimiamo quindi l'efficacia come rapporto dei due tempi:

$$\eta = \frac{T}{T_e} \tag{4.2}$$

Il tempo T necessario per l'esecuzione della memoizzazione può essere scomposto in varie componenti ottenendo le seguenti uguaglianze:

$$T = \alpha T_{hit} + (1 - \alpha) T_{miss} \tag{4.3}$$

$$T_{hit} = T_m \tag{4.4}$$

$$T_{miss} = T_m + T_e + \beta T_t \tag{4.5}$$

dove T_m è il tempo di accesso alla memoria, T_e il tempo in cui il processore esegue il calcolo richiesto dal metodo e T_t il tempo necessario per eseguire il blocco di *Trade off*. La costante β infine rappresenta la frequenza con cui il blocco di *Trade off* viene richiamato. Se la *Look up* va a buon fine vale la Formula 4.4 e quindi il tempo di memoizzazione coincide con il tempo necessario per leggere da memoria la coppia ingresso risultato (T_m). Viceversa se la *Look up* fallisce allora dobbiamo considerare il tempo impiegato per accedere alla memoria (T_m), il tempo per calcolare il risulatato da parte del processore (T_e) e infine il tempo di *Trade off* (T_t), necessario per decidere se inserire o meno il dato in memoria.

I due tempi, T_{hit} e T_{miss}, sono pesati all'interno del tempo di memoizzazione in base al valore dell'hit rate. Quindi sostituendo le formule 4.4 e 4.5 in 4.3, e sostituendo la Formula 4.3 in quella di η (4.2) otteniamo la seguente:

$$\eta = \frac{T_m}{T_e} + (1 - \alpha)\left(1 + \beta \cdot \frac{T_t}{T_e}\right) \tag{4.6}$$

4.3 Le due fasi del Trade off

Esistono due fasi del Trade off:

1. Suddivisione della memoria in zone. Avviene prima dell'effettiva esecuzione dei metodi e assegna una zona per ogni metodi in base a parametri noti da studi pregressi e forniti in ingresso dall'utente attraverso un'interfaccia grafica.

2. Aggiornamento della dimensione delle zone. Avviene durante l'esecuzione dei metodi, a intervalli prefissati, al fine di ridimensionare le zone ottenute dallo studio iniziale in base al comportamento delle funzioni a *run time*.

4.3.1 Fase 1: Suddivisione delle memoria in zone

In questa fase l' *hit rate* α è approssimato con una gaussiana secondo la formula:

$$\alpha = erf\left(\frac{N_d \cdot \tau/2}{\sigma \cdot \sqrt{2}}\right) \tag{4.7}$$

In generale l' *hit rate* dipende da quante coppie ingresso-risultato vengono salvate in memoria e dalla distribuzione statistica di tali valori, che nel nostro caso abbiamo posto uguale a una gaussiana, scelta che ci porta nella fase di testing a generare i dati di ingresso secondo tale distribuzione. I parametri τ e σ^2 sono rispettivamente la precisione, cioè la minima distanza possibile tra due valori contigui di ingresso, e la varianza, che indica quanto variano i valori in ingresso rispetto al valore medio.

Nel nostro studio avendo in ingresso n parametri, con varianze e precisioni differenti, nel calcolare l'efficacia η utilizziamo un' *hitrate* α dato dal prodotto delle varie α_n, una per ogni parametro:

$$\alpha_n = erf\left(\frac{N_{d,n} \cdot \tau/2}{\sigma_n \cdot \sqrt{2}}\right) \tag{4.8}$$

$$\alpha = \prod_n \alpha_n \tag{4.9}$$

In questa fase l'unico valore che varia nel tempo è il parametro N_d, che rappresenta il numero dei possibili ingressi, definito come segue:

$$N_d = \frac{S_m}{S_d} \tag{4.10}$$

dove S_m è la dimensione della memoria allocata per un dato metodo ed S_d è la dimensione del dato, composto dalla somma delle dimensioni dei parametri di ingresso e del risultato. Il valore di S_d viene calcolato durante l'inserimento dei dati, riguardanti i vari metodi, da parte dell'utente per mezzo del metodo di `sizeOf`.

Se i parametri in ingresso ad una funzione sono n il valore $N_{d,n}$ verrà calcolato come segue:

$$N_{d,n} = \sqrt[n]{N_d} \tag{4.11}$$

Come si può osservare nella Figura 4.2 la funzione dell'efficacia ha un andamento decrescente e tende al valore minimo $\frac{T_m}{T_e}$, che è raggiunto nel caso ottimo in cui si ha quando l' *hit rate* assume valore 1 che si verifica solo quando tutti i possibili inserimenti sono salvati in memoria. Osservando infatti la Formula 4.6, se $\alpha = 1$ il secondo termine della somma si annulla e l'efficacia $\eta = T_m/T_e$. Il metodo è tanto vantaggioso quanto minore è il tempo di accesso alle memoria rispetto al tempo necessario per effettuare i calcoli.

Nella costruzione del modello del *Trade off* è stato utilizzato il guadagno G, definito come funzione inversa dell'efficacia *eta*.

$$G = \frac{1}{\eta} \tag{4.12}$$

Il suo andamento è dunque crescente (Figura 4.3). Nel valutare a quale metodo assegnare la memoria scelgo la configurazione che mi fa ottenere il guadagno massimo.

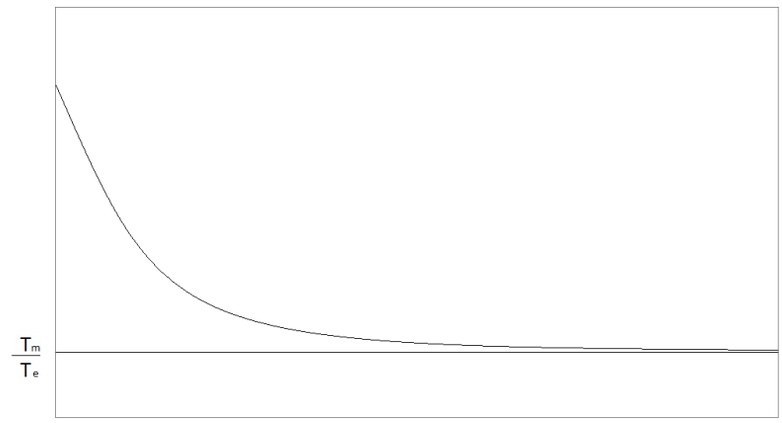

Figura 4.2: Efficacia

Il guadagno di per se non rappresenta in maniera esaustiva l'importanza di un metodo ai fini del nostro studio, per cui viene utilizzato un altro valore per pesarlo, e cioè la frequenza f che esprime quanto spesso richiamo il metodo in questione e da cui si ottiene il guadagno totale:

$$G_{tot} = \sum_i G_i \cdot f_i \tag{4.13}$$

L'obiettivo è massimizzare il guadagno totale.

La frequenza gioca un ruolo fondamentale perché in sua assenza e con l'utilizzo del solo guadagno G si ha il seguente problema: un metodo che presenta un elevato guadagno ma che viene richiamato solamente poche volte, occupa una porzione di memoria elevata, togliendo spazio a quei metodi che sono richiamati con una frequenza maggiore ma con un guadagno di poco inferiore.

Algoritmo di suddivisione della memoria

L'Algoritmo considera in una prima fase noti tutti i parametri a eccezione della quantità di memoria concessa all'*i-esima* funzione, che inizialmente è

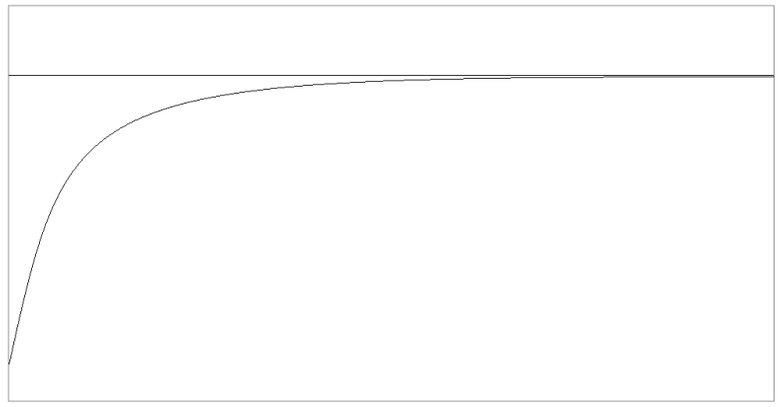

Figura 4.3: Guadagno

pari a zero e aumenta in seguito all'applicazione dell'algoritmo di assegnamento descritto di seguito.

L'algoritmo sopra descritto alloca in maniera incrementale la memoria disponibile alle funzioni a blocchi di dimensione $S_{d,i}$. Il valore $S_{d,i}$ è specifico per ogni funzione i e dipende dalla dimensione dei parametri di ingresso e dal risultato dell'*i-esima* funzione:

$$S_{d,i} = size(x_{i,1}) + ... + size(x_{i,n}) + size(f_i(x_{i,1}, ..., x_{i,n})) \tag{4.14}$$

Nel calcolare le dimensioni reali della singola coppia ingresso-risultato vanno considerati dei byte aggiuntivi, dovuti agli oggetti che inglobano i dati. Per una trattazione più approfondita del calcolo di $S_{d,i}$ si rimanda all'Appendice A.

Data la dimensione delle coppie ingresso-risultato per ogni funzione e conoscendo la dimensione massima della memoria disponibile S_M imposta dall'hardware che si sta utilizzando, finché c'è memoria disponibile ($S < S_M$) si valuta ciclicamente quale funzione apporta il maggiore guadagno con l'aggiunta di una singola coppia ingresso-risultato.

Algorithm 1 Algoritmo 1

Inputs : F insieme di funzioni pure $f_i(x_{i,1}, ..., x_{i,n})$;

Parameters : $S_{d,i} = size(x_{i,1}) + ... + size(x_{i,n}) + size(f_i(x_{i,1}, ..., x_{i,n}))$, dimensione data della f_i, memoria massima $S_{m,i}$, memoria allocata per f_i, inizialmente poste a zero $S = \sum_i S_{m,i}$, percentuali p_i di memoria concessa per ogni funzione f_i

1: **Begin**
2: **while** $(S < S_M)$ **do**
3: si prende la f_i con $G_i/(S_{m,i} + S_{d,i})$ massimo
4: si incrementa il valore di $S_{m,i} = S_{m,i} + S_{d,i}$
5: si aggiorna S_i
6: **end while**
7: $p_i = S_{m,i}/S_M$, per ogni i
8: **End**

Il valore S è la somma delle memorie allocate alle funzioni e serve per evitare di eccedere la memoria disponibile.

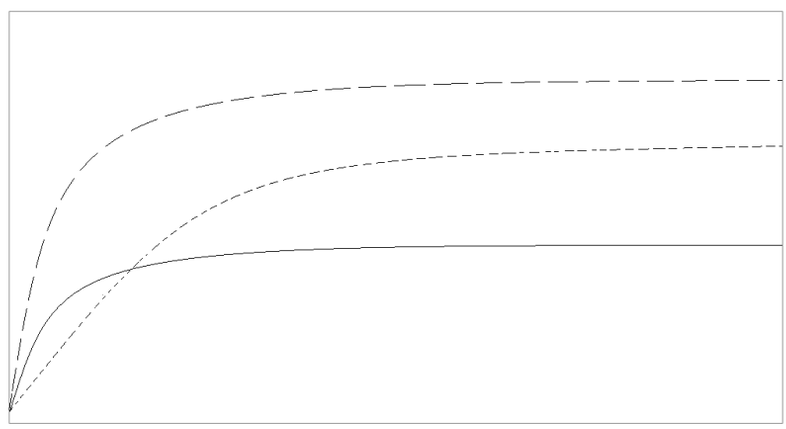

Figura 4.4: Guadagni

All'interno del ciclo viene calcolato il valore del guadagno per ogni fun-

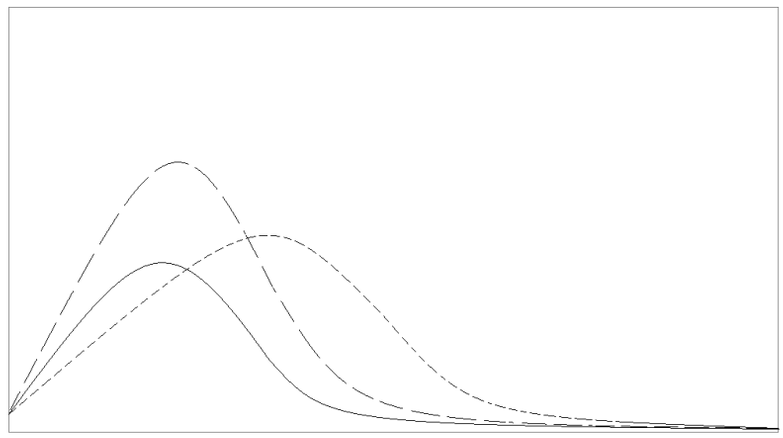

Figura 4.5: Guadagni pesati

zione i che viene pesato con un valore dato dalla somma della dimensiona corrente allocata per l'i-esima funzione più la dimensione di un nuovo inserimento. Viene selezionata la funzione con il valore di guadagno pesato maggiore e ne viene incrementato il campo di memoria allocata ($S_{m,i}$). Infine viene aggiornato il valore S.

Nella Figura 4.4 viene presentato a titolo di esempio un caso in cui sono messi a confronto i guadagni di tre funzioni.

Il guadagno viene pesato (Figura 4.5) in base alla memoria correntemente concessa al metodo per evitare che la funzione con guadagno massimo lasci senza memoria le altre funzioni.

Alla fine dell'esecuzione dell'algoritmo per ogni funzione è impostato un limite di memoria, utilizzabile della funzione stessa per salvare le coppie ingresso-risultato.

In conclusione, terminato tale assegnamento, i dati relativi alla percentuale di memoria concessa all'i-esimo metodo e tutti gli altri dati necessari per i futuri riassegnamenti, sono salvati nel database.

4.3.2 Fase 2: Aggiornamento della dimensione delle zone di memoria

Dalla fase precedente si ottiene una prima configurazione della memoria, basata su dei parametri noti a priori che però non consentono di prevedere il comportamento delle funzioni a run time. Per questo motivo si rende necessaria una riallocazione della memoria a intervalli regolari. Viene definito a tale scopo un altro algoritmo che utilizza, come quello descritto nella Sezione 4.3.1, il guadagno G. Inoltre nel calcolare il valore dell' *hit rate* non si utilizza più un'approssimazione per mezzo della *error function* (Formula 4.8), bensì vengono utilizzati #hit e #miss, che sono stati precedentemente salvati durante l'esecuzione dei metodi. L' i-esimo α quindi sarà dato dalla seguente formula:

$$\alpha = \frac{\#hit}{\#hit + \#miss} \tag{4.15}$$

L'algoritmo utilizzato è il seguente:

In questa seconda fase la riallocazione della memoria viene svolta dal secondo algoritmo a partire dalla configurazione risultante in seguito all'esecuzione dell'Algoritmo 1 , cioè con dei valori $S_{m,i}$ maggiori di zero e tali per cui la loro somma sia pari alla memoria massima fisicamente disponibile (S_M). Ogni volta che il blocco di *Trade off* viene richiamato, in base alla scadenza temporale precedentemente fissata, viene scelta la funzione con il guadagno pesato maggiore ovvero quella per cui la memoizzazione risulta più vantaggiosa. A tale funzione viene così assegnata una porzione ϵ di memoria in più rispetto alla porzione $S_{m,i}$ che aveva prima. Di conseguenza, essendo la dimensione della memoria fissa, la porzione dedicata alle altre funzioni viene diminuita uniformemente della quantità $\epsilon/(\#F-1)$. La quantità $\epsilon/(\#F-1)$ non è altro che l'incremento ϵ aggiunto alla funzione con gudagno

Algorithm 2 Algoritmo 2

Inputs : F insieme di funzioni pure $f_i(x_{i,1}, ..., x_{i,n})$;

Parameters : S_M memoria massima, $S_{m,i}$ memoria allocata per f_i, percentuali p_i di memoria concessa per ogni funzione f_i, $\#hit_i$, $\#miss_i$ numero di hit e miss per la funzione i-esima.

1: **Begin**
2: prendo la f_i con $G_i/(S_{m,i}$ massimo e la pongo uguale a f_{max}
3: aumento il valore $S_{m,i}$ della funzione f_{max} di un quantitativo ϵ
4: diminuisco il valore $S_{m,i}$ delle funzioni diverse da f_{max} di un quantitativo $\epsilon/(\#F - 1)$
5: $p_i = S_{m,i}/S_M$, per ogni i
6: **End**

pesato maggiore diviso per il numero di funzioni che presentano invece un guadagno pesato minore $(\#F - 1)$.

Capitolo 5

Implementazione

Il blocco di *Trade off* è una componente di un più ampio progetto organizzato in sei package principali: *block* che contiene le classi `TradeOffBlock` e `LookUpBlock` che permettono di accedere alle funzionalità di *Trade off* e *Look up*; *logic* al cui interno è presente la classe `TradeOff` che implementa le funzionalità illustrate nel capitolo precedentemente e di cui parleremo nel dettaglio nel seguente capitolo; *memory* che contiene le classi imputate al controllo del salvataggio dei dati su databese, XML e in memoria RAM; *pureFunction* che contiene le classi necessarie per la gestione delle funzioni pure, ossia le classi che permettono l'esecuzione dei moduli di *Pure function retrieval* e di *Bytecode modification*; *utils* contiene le classi interfunzionali di supporto per la realizzazione degli altri package.

Per la spiegazione approfondita del package *memory* si rimanda alle Tesi dei colleghi Katsumi F, Maica Reis I. [4] e Planer I. [8], per i package *pureFunction* e *utils* invece si rimanda alla Tesi del collega Marco Bessi [9], mentre il package *logic* viene illustrato nel capitolo di seguito.

Verranno inoltre analizzate alcune classi appartenenti ad altri package che sono stati modificati per poter essere compatibili con il blocco di *Trade off*.

5.1 Il package *logic*

Il package *logic* è composto da due classi:

- `MemoryAssignmentGUI`, interfaccia grafica che permette di inserire tutte le informazioni relative alle funzioni e di fare una divisione iniziale della memoria disponibile;
- `TradeOff`, viene utilizzato a run time per ripartire la memoria tra le funzioni.

5.1.1 MemoryAssignmentGUI

Prima dell'effettiva esecuzione delle funzioni viene eseguita l'interfaccia `MemoryAssignmentGUI` che prende in ingresso tutte le informazioni che caratterizzano la singola funzione, dal tempo di esecuzione alla sua frequenza, e anche informazioni utili per la memoizzazione come la memoria massima allocabile e il tempo di *Trade off*.

Tutti questi dati sono il risultato di analisi pregresse, infatti non sarebbe possibile sapere a priori la frequenza con cui verrà eseguito il metodo.

Successivamente, i dati che hanno subito delle modifiche in seguito alla fase di run time, vengono aggiornati al fine di rendere la metodologia del *Trade off* sensibile alle variazioni di utilizzo da parte del sistema.

Per esempio, se in un primo momento si era stimato che una data funzione sarebbe stata richiamata poche volte, quindi con una bassa frequenza, il *Trade off* tende a concedergli una minore area di memoria, a parità di guadagno rispetto alle altre funzioni. In seguito, se durante l'esecuzione delle funzioni, avviene che tale funzione che in un primo momento veniva chiamata con una bassa frequenza, viene richiamata parecchie volte, il *Trade off*, quando viene attivato, valuta il beneficio apportato da questa funzione in relazione al dato aggiornato di frequenza e incrementa perciò l'area di memoria che le aveva destinato in precedenza.

Per quanto riguarda il funzionamento della classe `MemoryAssignmentGUI` possiamo dire che appena viene eseguita si avvia la schermata mostrata in Figura 5.1 in cui l'utente deve procedere all'inserimento dei seguenti dati riguardanti la memoria: quantità massima di memoria allocabile, tempo che il *Trade off* impiega a riallocarla, frequenza di chiamata del blocco di *Trade off*.

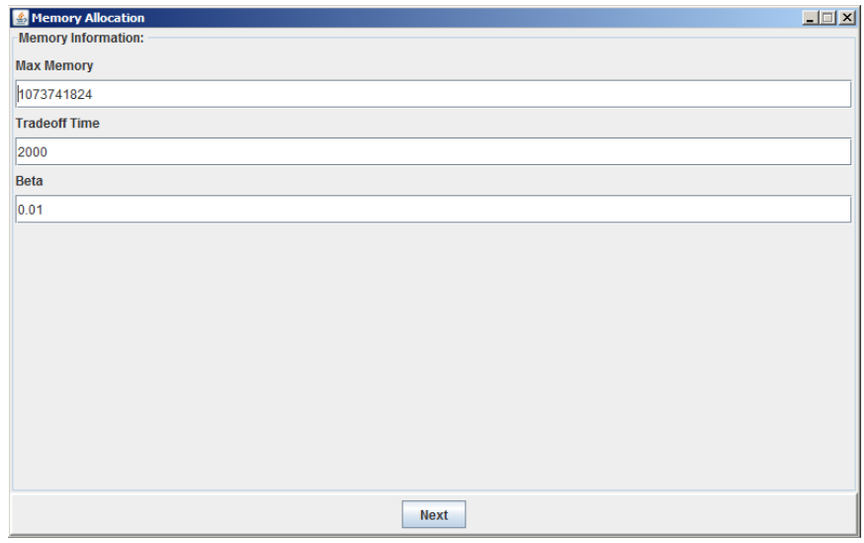

Figura 5.1: Informazioni sulla memoria

In seguito la classe `MemoryAssignmentGUI` si collega al database per ricavare il nome delle funzioni pure; nel database saranno contenuti inoltre tutti gli inserimenti dei dati relativi alle funzioni. In Figura 5.2 viene presentato il diagramma entità relazione del database, che rispetto a quello presentato nel Capitolo 3 Sezione 3.3.1 presenta una tabella in più, utilizzata appunto per contenere dei dati di supporto relativi alle funzioni.

Infine, dopo questa prima fase di data entry, avviene l'esecuzione del primo algoritmo descritto nel Capitolo 4 (Listing 1).

Nel seguito è riportato il codice del metodo che implementa tale algoritmo:

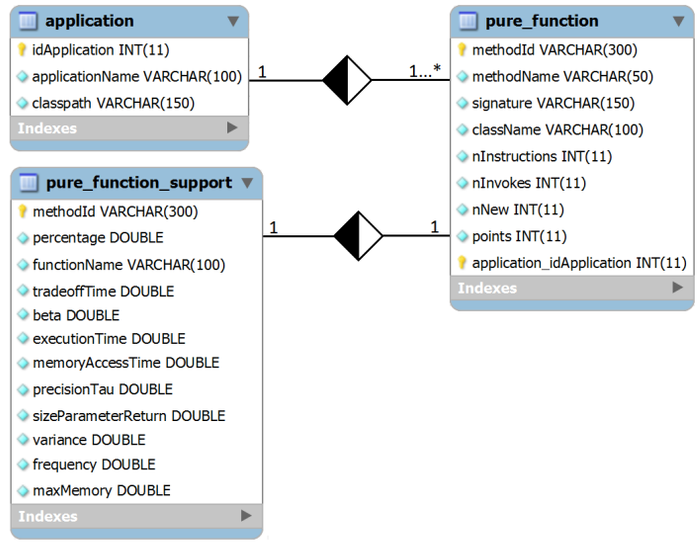

Figura 5.2: Tabelle del DB per la memorizzazione delle informazioni delle funzioni pure.

Listing 5.1: Metodo assignMemory per la riallocazione della memoria prima dell'esecuzione delle funzioni.

```
220 public static void assignMemory(HashMap<String,
221                 TOFunction> mMemory) throws MathException{
222
223    double memoryAllocated = 0;
224    double max = 0;
225    TOFunction tofMax = null;
226    TOFunction tofTemp;
227
228    Collection cValues = mMemory.values();
229    Iterator<TOFunction> tofIterator = cValues.iterator();
230    double maxMemory = tofIterator.next().getMaxMemory();
231    tofIterator = cValues.iterator();
232
233    while(memoryAllocated < maxMemory){
234
235       while(tofIterator.hasNext()){
```

```
236
237         tofTemp = tofIterator.next();
238
239         if(tofTemp.weightedGain() > max){
240               max = tofTemp.weightedGain();
241               tofMax = tofTemp;
242               }
243            }
244         max = 0;
245         tofMax.updateMemoryAllocated();
246
247         memoryAllocated += tofMax.getSizeParameterReturn();
248         tofIterator = cValues.iterator();
249         }
250      }
```

Il metodo prende in ingresso un hashmap `mMemory` contenente un oggetto `TOFunction` per ogni tipologia di funzione. Al suo interno vengono utilizzati diverse variabili:

- `memoryAllocated`, inizialmente posto a zero serve come condizione di uscita dal primo `while`, ed è la sommatoria delle zone di memoria allocata durante il metodo,
- `maxMemory`, indica il valore massimo della memoria disponibile,
- `max`, è una variabile utilizzata per scegliere la funzione con il valore di guadagno pesato massimo a ogni iterazione del primo `while`,
- `tofMax`, è una variabile di tipo `TOFunction` e serve per selezionare la funzione con il guadagno pesato maggiore in modo tale da poterle aggiungere successivamente parte di memoria,
- `tofTemp`, anch'essa variabile di tipo `TOFunction` che viene utilizzata per scorrere le funzioni presenti in `mMemory`.

Successivamente alla definizione delle variabili appena descritte e di altre di supporto è presente un ciclo `while` esterno che viene eseguito finchè la sommatoria delle memorie allocate risulta inferiore al valore di memoria massima disponibile. Al suo interno viene valutato il guadagno apportato da ogni funzione per mezzo di un altro ciclo `while` che scorre le suddette funzioni. La valutazione avviene come se a ogni funzione venisse assegnato lo spazio per un singolo inserimento, che corrisponde alla dimensione della coppia parametri di ingresso e risultato che varia per ogni funzione. Selezionata la funzione con il guadagno pesato maggiore, scelta per mezzo della variabile `max` e assegnata alla variabile `tofMax`, se ne aggiorna il valore della memoria allocata per mezzo del metodo `updateMemoryAllocated` presente nella classe `TOFunction`.

Il guadagno pesato viene ricavato per mezzo del metodo `weightedGain`, appartenente alla classe `TOFunction`, dentro il ciclo `while` interno. Infine viene incrementato il valore della variabile `memoryAllocated` di un quantitativo uguale alla dimensione della coppia ingresso-risultato della funzione assegnata a `tofMax`.

Al termine dell'esecuzione del metodo appena descritto otteniamo come risultato l'iniziale suddivisione della memoria disponibile per le varie funzioni presenti nel database, in cui viene anche salvata questa configurazione della memoria affinchè possa essere richiamata durante la fase di run time.

5.1.2 TradeOff

All'interno della classe `TradeOff` sono presenti due funzionalità:

1. viene riallocata la memoria tra le funzioni in esecuzione al fine di massimizzare il guadagno,

2. vengono gestiti gli inserimenti.

Ogni qual volta si deve salvare in memoria una coppia ingresso-risultato si valuta per mezzo del metodo `verifySpace`, presente all'interno della classe `MemoryManagement`, che ogni funzione non superi il limite massimo di memoria concessa loro in un' allocazione di memoria precedente.

Il *Trade off* agisce a intervalli predefiniti per riallocare la memoria a seguito di cambiamenti nella frequenza o nella percentuale di hit delle singole funzioni.

Viene quindi mostrata l'implementazione del secondo algoritmo esposto nel Capitolo 4 (Listing 2) per mezzo del seguente metodo:

Listing 5.2: Metodo assignMemoryRunTime per la riallocazione della memoria durante l'esecuzione delle funzioni.

```
164 private static void assignMemoryRunTime(HashMap<String,
165             TOFunction> mMemory) throws MathException {
166    ArrayList variations = new ArrayList();
167    ArrayList functions = new ArrayList();
168    double max;
169    int maxIndex = 0;
170    double percent;
171    int ok;
172
173    TOFunction tofTemp;
174
175    Collection cValues = mMemory.values();
176    Iterator<TOFunction> tofIterator = cValues.iterator();
177    tofIterator = cValues.iterator();
178
179    while(tofIterator.hasNext()){
180       tofTemp = tofIterator.next();
181       functions.add(tofTemp.getFunctionName());
182       double before = tofTemp.getWeightedRo();
183       double after = tofTemp.weightedGainRunTime();
184       variations.add( after - before );
```

```
185         }
186
187     percent = new Double(functions.size()-1) / 100;
188
189     max = Double.parseDouble(""+variations.get(0));
190     for(int i = 0;i < functions.size();i++){
191         if(Double.parseDouble(""+variations.get(i)) > max){
192             max = Double.parseDouble(""+variations.get(i));
193             maxIndex = i;
194             }
195         }
196
197     ok = 0;
198     tofIterator = cValues.iterator();
199     while(tofIterator.hasNext()){
200         tofTemp = tofIterator.next();
201         if(tofTemp.getFunctionName().equals(
202     functions.get(maxIndex)) &&
203     tofTemp.getMemoryAllocated() + tofTemp.getMaxMemory() *
204     percent < tofTemp.getMaxMemory() ){
205             ok++;
206         }else if(!tofTemp.getFunctionName().equals(
207     functions.get(maxIndex)) &&
208     tofTemp.getMemoryAllocated() - tofTemp.getMaxMemory() *
209     (percent / (functions.size()-1)) > 0){
210             ok++;
211         }
212     }
213
214     tofIterator = cValues.iterator();
215     while(tofIterator.hasNext()){
216         tofTemp = tofIterator.next();
217         if(tofTemp.getFunctionName().equals(
```

```
218     functions.get(maxIndex)) && ok == functions.size() ){
219           tofTemp.setMemoryAllocated(
220     tofTemp.getMemoryAllocated() + tofTemp.getMaxMemory() *
221     percent);
222        }else if(ok == functions.size()){
223           tofTemp.setMemoryAllocated(
224     tofTemp.getMemoryAllocated() - tofTemp.getMaxMemory() *
225     (percent / (functions.size()-1)) );
226        }
227     }
228
229 }
```

Il metodo ha le seguenti variabili principali:

- `variations`, array che contiene le differenze dei guadagni pesati di tutte le funzioni, del caso corrente rispetto a una configurazione precedente,
- `functions`, array contenente i nomi delle funzioni,
- `max`, variabile utilizzata per scegliere la funzione con il valore di guadagno pesato massimo,
- `tofTemp`, variabile di tipo `TOFunction` che viene utilizzata per scorrere le funzioni presenti in `mMemory`,
- `percentage`, rappresenta la quantità ϵ aggiunta alla funzione con la variazione di guadagno pesato massima,
- `ok`, serve per controllare che non vengano mai violati i vincoli di memoria concessa a una singola funzione compresi tra il limite minimo di 0 e il limite massimo corrispondente alla dimensione di memoria fisica disponibile.

Durante il primo ciclo `while` vengono popolati i due array `variations` e `functions` e viene definito il valore di `percent` in base al numero di funzioni presenti. Subito dopo è presente un ciclo `for` che seleziona la funzione con la variazione di guadagno pesato maggiore, mentre i due cicli `while` successivi servono rispettivamente a controllare che la modifica della configurazione della memoria possa aver luogo e nel caso si possa fare effettuare la modifica.

Il metodo `assignMemoryRunTime` viene richiamato a intervalli prefissati, e per fare ciò è stata modificata la classe `MemoryManagement` che controlla in fase di *Look up* se mandare in esecuzione o meno il *Trade off*. Una volta in esecuzione il blocco di *Trade off* aggiorna le frequenze e modifica la configurazione della memoria, secondo il metodo appena descritto, basandosi sia sulle frequenze di esecusione che sulle percentuali di hit ottenute dalle singole funzioni.

5.2 Il package *memory*

A seguito della fase appena descritta è possibile che una funzione abbia a disposizione meno memoria di quanta ne avesse in una configurazione precedente, il che porta alla necessità di dover eliminare le coppie ingresso-risultato in eccesso. Tale compito è svolto dal metodo `removeTOPs`, appartenente alla classe `TOFunction`, che richiamando il metodo `findEntryToRemove` cancella gli inserimenti meno frequenti. Il metodo `findEntryToRemove` si trova all'interno della classe `MemoizationHashSFArr` nel package *memory*. La classe `MemoizationHashSFArr` consiste nella ridefinizione del tipo `HashMap` di Java e le sue caratteristiche vengono illustrate nella sezione seguente.

5.2.1 findEntryToRemove

Il metodo `findEntryToRemove` ha come obiettivo la cancellazione dei valori richiamati con bassa frequenza ed è così implementato:

Listing 5.3: Metodo findEntryToRemove utilizzato per cancellare le coppie parametri di ingresso e risultato richiamate poco frequentemente.

```
97  public void findEntryToRemove(int quantity){
98
99  do{
100
101  int quantityDeleted = 0;
102  long[] meanFrequency = new
103  long[DEFAULT_ARRAY_CAPACITY];
104
105  for(int i=0;i<this.hashed_elements.length;i++){
106     if(hashed_elements[i].sfarr_elements[0]!=null){
107        for(int j=hashed_elements[i].sfarr_size-1; j>=0 &&
108             j>=(hashed_elements[i].sfarr_size-1-quantity)
109             ;j--){
110
111           meanFrequency[i] += this.hashed_elements[i].
112                   sfarr_elements[j].frequency;
113        }
114        if(this.hashed_elements[i].sfarr_size >= quantity){
115           meanFrequency[i] = meanFrequency[i]/quantity;
116        }else{
117           meanFrequency[i] = meanFrequency[i] /
118                     this.hashed_elements[i].sfarr_size;
119        }
120     }
121  }
122
123  long tempMin = 0;
124  int index = 0;
125  int k;
126
127  for(k=0; k < meanFrequency.length;k++){
```

```
128     if(meanFrequency[k]!=0 && tempMin==0){
129         tempMin = meanFrequency[k];
130         index = k;
131     }
132     if(meanFrequency[k] < tempMin && meanFrequency[k]>0 &&
133                                         tempMin > 0){
134         tempMin = meanFrequency[k];
135         index = k;
136     }
137  }
138
139  for(int h=hashed_elements[index].sfarr_size-1; h>=0 &&
140    h>=(hashed_elements[index].sfarr_size-1-quantity);h--){
141     Return ret = this.hashed_elements[index].remove(
142         this.hashed_elements[index].sfarr_elements[h].hash,
143         this.hashed_elements[index].sfarr_elements[h].key);
144     if(ret!=null)
145         --hashed_size;
146     quantityDeleted++;
147  }
148  quantity = quantity-quantityDeleted;
149  }while(quantity > 0);
150 }
```

Come viene esemplificato in Figura 5.3 il generico oggetto di tipo `MemoizationHashSFArr` è composto da un numero fisso di posizioni, da 0 a 8191. A ognuna di queste posizioni è associato un array dove vengono salvati i dati in caso di collisione a seguito di un inserimento nell'hashmap.

Gli array corrispondenti a ogni posizione hanno lunghezza iniziale di 512; se è necessario l'inserimento di un numero maggiore di elementi la loro dimensione viene fatta raddoppiare. Viceversa se il numero di elementi presenti diminuisce a seguito di cancellazioni e scende sotto la metà della dimensione

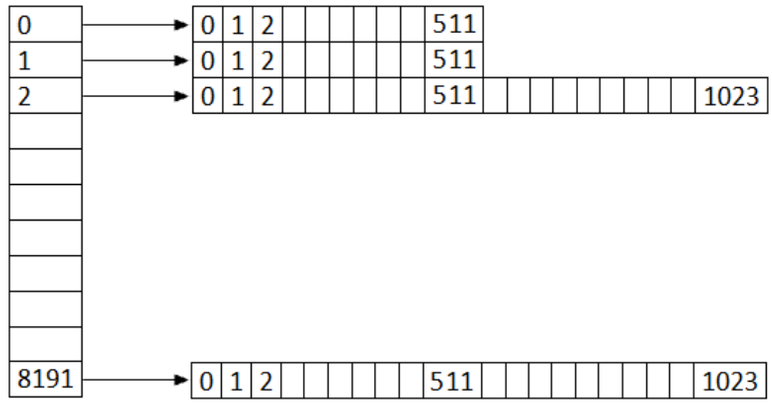

Figura 5.3: MemoizzazionHashSFArr

dell'array, quest'ultima viene dimezzata.

Gli elementi presenti in ognuno di questi array sono ordinati in base alla frequenza con cui vengono richiamati.

Il metodo `findEntryToRemove` ha le seguenti variabili principali:

- `quantity`, numero di elementi da cancellare,
- `quantityDeleted`, serve per sapere quanti elementi sono stati cancellati fino a quel momento all'interno del ciclo `do-while`, fino a che non si arriva a cancellarne esattamente `quantity`,
- `meanFrequency`, è un array contenente le frequenze medie di ogni array presente nell'oggetto di tipo `MemoizationHashSFArr`,
- `tempMin`, viene utilizzata per trovare l'array con frequenza media minore.

Il metodo è composto da un ciclo `do-while` che continua a cancellare valori dall'hashmap finchè non ne sono stati cancellati un numero pari a `quantity`.

All'interno del ciclo `do-while` sono presenti tre cicli for:

1. il primo serve per calcolare a salvare in `meanFrequency` il valore delle frequenze medie di ogni array dell'hashmap,
2. il secondo, facendo uso della variabile `tempMin`, trova il valore minimo di frequenza media nell'array `meanFrequency`,
3. infine il terzo ciclo cancella i valori dall'array selezionato nel ciclo precedente a partire dagli ultimi valori, posto che gli elementi di ogni array sono ordinati per frequenza.

Il senso del ciclo `do-while` è che una volta trovato l'array che presenta la frequenza media minore non è detto che tale array abbia un numero di elementi pari al valore di `quantity` da eliminare. Si procede quindi a cancellare i valori presenti nell'array selezionato; il numero di valori cancellati è memorizzato all'interno della variabile `quantityDeleted` e alla fine del ciclo viene aggiornato il valore della variabile `quantity`:

```
185  quantity = quantity-quantityDeleted;
```

In una successiva iterazione del ciclo `do-while` viene selezionato un nuovo array con frequenza media minima in cui trovare i restanti valori da cancellare. Il ciclo è eseguito fino a quando la variabile `quantity`, attraverso successivi aggiornamenti, non diventa pari a 0.

Capitolo 6

Validazione empirica

In questo capitolo vengono riportati i risultati ottenuti da questo lavoro di Tesi, in particolare riguardo alla modifica del blocco di *Trade off* all'interno della metodologia della memoizzazione e al suo impatto sul consumo energetico rispetto alla versione precedentemente implementata (Capitolo 3).

Nella Sezione 6.1 sarà illustrata la tecnica per la misurazione dei consumi energetici del sistema test. Invece nella Sezione 6.2 verranno illustrati i risultati dati dall'introduzione del blocco di *Trade off*.

6.1 Misurazione dei consumi energetici

La crescita della ricerca nel Green IT rende necessaria l'esecuzione di accurate misurazioni per poter monitorare, studiare ed analizzare i consumi energetici al fine di trovare tecniche e metodi per ridurli efficacemente. Un sistema informatico è un'architettura molto complessa, strutturata su più livelli. Per questo motivo i consumi energetici sono distribuiti su tutto il sistema complessivo, dall'hardware impiegato (CPU, schede madri, hard disk), al software eseguito (sistema operativo, applicazioni utente). Occorre quindi considerare i dispendi energetici di ogni singolo componente del sistema.

Esistono principalmente due soluzioni per effettuare le misurazioni dei consumi, basate sull'utilizzo della documentazione rilasciata dai produttori dell'hardware analizzato, o impiegando strumenti ad-hoc che mediante sensori integrati monitorano il fabbisogno energetico. Nel nostro caso abbiamo fatto uso di misurazioni ad-hoc. Oggi molte apparecchiature informatiche sono fornite di sensori hardware built-in che forniscono dati al sistema operativo, come ad esempio le percentuali d'utilizzo dei componenti o la loro temperatura, però questi sensori non forniscono informazioni sufficienti sui consumi energetici di cui quest'analisi necessita.

Per rendere possibile un confronto del sistema prima e dopo la modifica apportata dal framework, è stata misurata l'energia elettrica consumata dal sistema. Per eseguire queste rilevazioni esistono in commercio dei prodotti come gli *energy meter* che misurano la quantità di corrente assorbita da un carico elettrico oppure i *current clamp* che misurano la differenza di potenziale a seguito del passaggio di corrente d'intensità nota. Tuttavia questi strumenti non erano adatti al raggiungimento degli obiettivi prefissati, in quanto non erano in grado di registrare i valori letti nel tempo in modo da poterli poi acquisire ed elaborare con un computer. Per ovviare a queste problematiche è stato usato uno strumento costruito da Formenti e Gallazzi nel corso del loro lavoro di tesi [5].

6.1.1 Strumenti di misurazione

Lo strumento realizzato permette di monitorare il consumo di tutto il sistema o dei singoli componenti, di registrare i valori istantanei nel tempo e di acquisirli per una successiva elaborazione. Il kit sperimentale di misura si compone di tre parti:

1. *System board*: una board per il monitoraggio della potenza assorbita dal sistema e dai singoli componenti hardware. In particolare è possibile misurare i consumi di scheda madre, processore e disco fisso;

2. *DAQ Board*: è una scheda di acquisizione dati che acquisisce i segnali provenienti dalla board per il monitoraggio, li elabora e li trasmette ad un computer. La DAQ Board è in grado di gestire sia segnali analogici che digitali, permette l'acquisizione di più segnali contemporaneamente e il trasferimento tramite collegamento USB al PC. La board utilizzata è la NI USB-6210 della National Instruments;

3. *Software per l'acquisizione e l'analisi dei dati*: il software creato è basato su Laboratory Virtual Instrumentation Engineering Workbench, un ambiente di programmazione visuale della National Instruments. Tale software permette di acquisire, monitorare e visualizzare i grafici dei segnali provenienti dalla DAQ Board. Calcola inoltre i valori medi ed i valori integrali della potenza misurata, fornendo in tempo reale i consumi energetici dell'hardware analizzato.

Lo strumento che è stato utilizzato durante tutti i test è la *System Board* (Figura 6.1) che permette di misurare il consumo di potenza di tutto il sistema poichè si pone a monte dell'alimentatore del computer.

La Sytem Board è stata collegata a una DAQ Board (Data Acquisition Board) che ha il compito di acquisire i segnali inviati dalla System Board, di elaborarli e inviarli al computer. Per acquisire i dati è stato utilizzato un DAQ della National Instruments il modello NI USB-6210 DAQ (Figura 6.2) le cui specifiche tecniche sono riportate in Tabella 6.1.

Infine per la visualizzazione e il salvataggio dei dati è stato utilizzato un software (Figura 6.3), anch'esso sviluppato da Formenti e Gallazzi [5], basato su LabVIEW, un ambiente di programmazione visuale della National Instruments. Questo software è stato eseguito da un laptop collegato via USB alla *DAQ Board* e ovviamente alimentato tramite una presa diversa da quelle della *System board*.

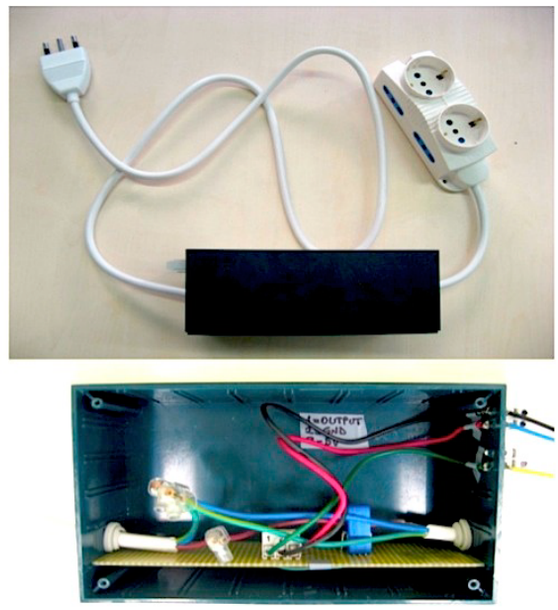

Figura 6.1: System Board per la misurazione dei consumi energetici.

6.1.2 Oggetto della misura

Per un corretto svolgimento delle misurazioni, è stato necessario definire le condizioni al contorno entro le quali eseguire i test, al fine di rendere i risultati tra loro comparabili. Grande importanza è assunta dalla riproducibilità dei test poiché è stato verificato che la ripetizione dello stesso test in momenti diversi, anche sullo stesso sistema, ha prodotto risultati differenti. È stato quindi evidente che mantenere sul sistema la stessa configurazione hardware

Figura 6.2: DAQ Board NI USB-6210 della National Instruments.

Canali Analogici d'ingresso	16
Sample Rate	250 kS/s
Risoluzione	16 bit
Range di Massimo Voltaggio	-10V /+10 V
Range di precisione	2,69 mV
Range di Minimo Voltaggio	-200 /-200 mV
Range di precisione	0,088 mV
Memoria Scheda	2095 samples
Canali Digitali I/O	4 DI / 4 DO
Massimo Input Range	0 / 5,25V
Massimo Output Range	0 / 3,8V
Connettori	Morsetti a vite

Tabella 6.1: Specifiche tecniche della scheda NI USB-6210 DAQ.

e software, è condizione necessaria ma non sufficiente per ottenere sistemi energeticamente equivalenti. Infatti, il consumo del sistema in idle, misurato in momenti differenti, può subire delle variazioni. Si è stabilita quindi una metodologia di esecuzione dei rilevamenti che prevede la misurazione dell'idle del sistema prima e dopo il test, in modo tale da controllare se tale valore cambi di poco.

Notata l'importanza e la variabilità dell'idle, prima di eseguire il test, è stato effettuato il monitoraggio del sistema in attesa che si portasse in una condizione di equilibrio con un idle pressoché costante. Questo approccio ha permesso di rendere confrontabili i delta di consumo tra valore istantaneo durante il test e il consumo in idle, anche al di fuori della stessa sessione, purché rieseguiti su un sistema quasi equivalente. Proprio per la variabilità del consumo in idle, invece, i valori assoluti dei consumi non sono direttamente confrontabili al di fuori della stessa sessione, anche se eseguiti sullo stesso sistema. Non è realisticamente possibile mantenere un sistema hard-

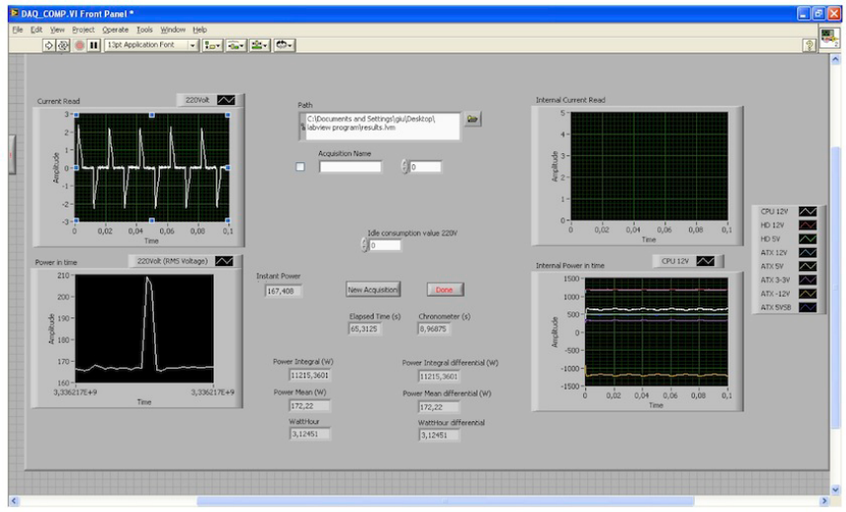

Figura 6.3: Interfaccia del tool di misurazione basato su LabView.

ware e software perfettamente equivalente nel tempo; il normale utilizzo del sistema, anche senza l'installazione di nuovo software, porta inevitabilmente a variazioni nello stesso. I valori riportati in questo lavoro sono stati ottenuti mediando i risultati derivati da diversi test secondo la metodologia appena descritta.

Le misure sono state effettuate interamente sulla stessa macchina, un IBM eServer x3500, che presenta le seguenti caratteristiche:

- 2 Processori Intel Xeon Quad-core x5450 @ 3.00 GHz (12 MB L2 cache), FSB 1,333 MHz,
- Memoria di sistema 17GB PC2-5300 DDR2 SDRAM,
- Sitema operativo Windows 2007 Server Enterprise.

Per dare un'indicazione dell'ordine di grandezza dei consumi complessivi del sistema server utilizzato durante le fasi di test, è qui riportato in Figura 6.4 uno degli andamenti di consumo in idle della macchina. In questo modo è possibile valutare l'incidenza percentuale dei consumi dei test e valutare il risparmio ottenibile.

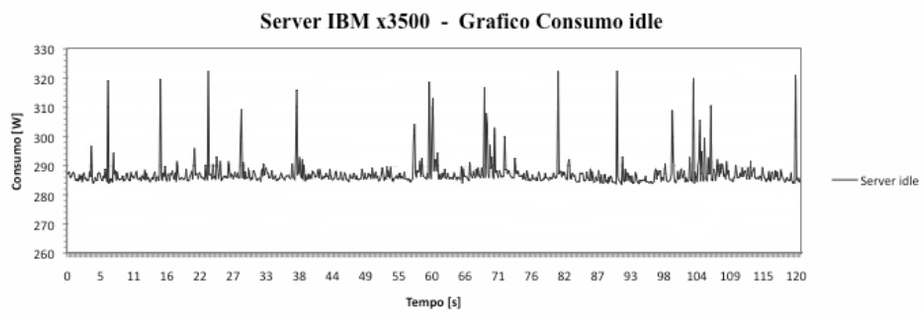

Figura 6.4: Consumo idle della macchina Server IBM x3500 del Politecnico di Milano.

Il server IBM x3500 è una macchina pensata per operare 24 ore su 24, sette giorni su sette. Dai risultati mostrati in figura, ricaviamo che mantiene un consumo idle medio pari a 287W. Considerando la macchina in esecuzione per un anno, sempre accesa e senza carichi di lavoro, scopriamo che assorbe un'energia maggiore di 9050 Mega Joule, che corrispondono ad un consumo superiore a 2500kWh annui. Il dispendio energetico così elevato è imputabile alla presenza degli otto core di processore, dei sei slot DIMM di memoria RAM, e in particolare per la presenza di una doppia unità d'alimentazione. Tale ridondanza, necessaria per aumentare la sicurezza e la stabilità del sistema, richiede un maggior dispendio energetico, che porta il consumo idle a livelli molto più elevati rispetto ad un sistema desktop.

6.2 Validazione del blocco di Trade off

Dopo aver introdotto le caratteristiche del server e quelle del kit per effettuare le misure, nella Sezione 6.2.1 verranno presentate le funzioni utilizzate nei test e verrà motivata la loro scelta, nella Sezione 6.2.2 verranno spiegati nel dettaglio i test svolti e infine nella Sezione 6.2.3 ne verranno analizzati i risultati.

6.2.1 Scelta delle funzioni

Come visto nel Capitolo 3 non tutte le funzioni possono essere memoizzate con successo. Alcune potrebbero non essere sufficientemente *computation intensive* e la memoizzazione si tradurrebbe in un rallentamento nella loro esecuzione. Per *computation intensive* si intende che il tempo di esecuzione T_e non si discosta di molto dal tempo di accesso alla memoria T_m, proprio perchè la funzione nella sua versione originale non occupa la CPU per molto tempo. Quindi per filtrare le funzioni tali per cui la memoizzazione sia vantaggiosa si valuta il rapporto T_m/T_e che, come visto nel Capitolo 4 Formula 4.6, rappresenta il limite minimo dell'efficacia η. Si prenderanno quindi quelle funzioni che presentano un basso valore di T_m/T_e. Le funzioni prese in considerazione per effettuare i test sono le seguenti:

- *Implied Volatility*, funzione che calcola la volatilità cui è soggetta un'opzione; è calcolata sulla base del modello di Black and Scholes, che la stima sulla base del prezzo corrente di un'opzione, del prezzo dello strumento sottostante, del prezzo d'esercizio, della durata residua e del livello dei tassi di interesse [11],
- *Black Scholes*, funzione che calcola l'espressione per il prezzo di non arbitraggio di un'opzione call (put) di tipo europeo, ottenuta sulla base del modello di Black-Merton-Scholes [3],
- *XIRR*, funzione che calcola il valore annuo dell' *Internal Rate of Return*(IRR) di un *cash flow* in un momento arbitrario,
- *Fourier*, funzione che computa i primi N coefficienti di Fourier per la funzione $f(x) = (x + 1)^x$.

Le funzioni Implied Volatility e Black Scholes sono largamente usate in applicazioni finaziarie per stimare prezzi e volatilità di stock options[1]. Allo

[1] Le stock option sono opzioni call europee o americane che danno il diritto di acquistare azioni di una società ad un determinato prezzo d'esercizio (detto strike). Non esistono per

stesso modo la funzione XIRR è inclusa in svariate applicazioni bancarie per valutare investimenti o piani di credito. Per cui il potenziale impatto della tecnica della memoizzazione su queste funzioni è molto alto.

6.2.2 Descrizione delle misure

Scelte le funzioni memoizzabili l'obiettivo è analizzare e confrontare i consumi energetici durante la loro esecuzione nei seguenti casi:

1. Esecuzione funzioni originali,
2. Esecuzione funzioni modificate con l'aggiunta della memoizzazione,
3. Esecuzione funzioni modificate con l'aggiunta della memoizzazione e del blocco di *Trade off*.

Le misure sono state effettuate sul server descritto nella Sezione 6.1.2 in ambiente di test. I dati sono stati usati per verificare un'effettiva riduzione dei consumi nella configurazione con il *Trade off* rispetto alle altre. Il server di cui ci siamo serviti è dotato di una memoria pari a 17 GB, mentre per i test è stato utilizzato 1 GB. I dati sono stati ottenuti tramite la misurazione del consumo energetico del server, durante lo svolgimento dei test di stress, al variare della frequenza di chiamata delle funzioni, della media dei parametri di ingresso e della loro varianza.

Nel secondo e nel terzo caso il testing segue tre fasi distinte:

1. Creazione e salvataggio degli ingressi delle funzioni,
2. Esecuzione funzioni per il riempimento della memoria,
3. Esecuzione funzioni con diversi valori di frequenza, media e varianza.

Mentre nel primo caso, visto che sono state eseguite le funzioni originali, non è necessario salvare alcun dato in memoria, perciò la fase due è assente. Nel seguito verranno analizzate nel dettaglio le diverse fasi.

tutte le società per azioni, ma solo per quelle quotate.

Fase 1: Creazione e salvataggio degli ingressi delle funzioni

Nella prima fase vengono generati gli ingressi delle funzioni secondo una distribuzione gaussiana. Questi ingressi sono stati salvati in dei file in modo tale da rendere i tre tipi di test visti sopra paragonabili, perchè eseguiti sullo stesso set di dati di ingresso, con caratteristiche di frequenza, media e varianza identiche. Nel detteglio, per creare i dati d'ingresso, si fa uso della classe `ServerGreenC` che offre anche delle funzionalità per la connettività: il laptop, su cui viene eseguita la classe `ClientGreenC`, dopo essersi collegato al server, che invece esegue la classe `ServerGreenC`, gli invia dei numeri. La classe `ServerGreenC`, attraverso il costrutto `switch-case`, interpreta questi numeri per compiere determinate operazioni. Per quanto riguarda la creazione dei dati ci sono sei `case` che creano altrettanti file, di cui uno conta 40 milioni di ingressi che verrano poi utilizzati nella Fase 2 e i restanti cinque, ognuno dei quali con 1 milione di ingressi, sono relativi alla Fase 3; le caratteristiche di questi ultimi cinque file sono specificate nella Tabella 6.2.

Fase 2: Esecuzione funzioni per riempimento della memoria

Alla Fase 1 appena descritta segue la fase di riempimento della memoria durante la quale vengono eseguite le funzioni con gli ingressi precedentemente salvati. Ricordiamo che questa fase viene eseguita solo nel secondo e terzo caso, rispettivamente con memoizzazione e con memoizzazione con il *Trade off*. Durante questa fase viene letto il file con i 40 milioni di ingressi creato nella fase precedente e parallelamente vengono effettuate le misure per mezzo del laptop collegato via USB alla *DAQ Board* che esegue il software per l'acquisizione dei dati.

In questa fase le funzioni vengono chiamate tutte con la stessa frequenza e il numero di chiamate di funzione è molto elevato, pari a 10 milioni per ogni funzione, proprio perchè il nostro obiettivo è riempire la memoria.

Fourier Series	Frequenza 25%		Frequenza 70%		Frequenza 10%		Frequenza 10%		Frequenza 10%	
	media	varianza	media	varianza	media	varianza	media	varianza	media	varianza
row	10	1	10	0,7	10	0,4	10	0,8	10	1
x1	1	0,1	2	0,14	1	0,04	3	0,24	4	0,4
x0	2	0,2	3	0,21	2	0,08	4	0,32	5	0,5
nstep	1000	100	1200	84	1100	44	1400	112	1500	150
Implied Volatility	Frequenza 25%		Frequenza 10%		Frequenza 70%		Frequenza 10%		Frequenza 10%	
	media	varianza	media	varianza	media	varianza	media	varianza	media	varianza
S	45	3,6	40	2	50	2	60	6	45	4,05
T	50	4	45	2,25	55	2,2	57	5,7	50	4,5
X	1	0,08	0,5	0,025	2	0,08	3	0,3	1,5	0,135
r	0,1	0,008	0,05	0,0025	0,3	0,012	0,5	0,05	0,4	0,036
price	derived		derived		derived		derived		derived	
Black Scholes	Frequenza 25%		Frequenza 10%		Frequenza 10%		Frequenza 70%		Frequenza 10%	
	media	varianza	media	varianza	media	varianza	media	varianza	media	varianza
CallPutFlag	0	0	0	0	0	0	0	0	0	0
S	45	2,25	45	3,15	60	5,4	50	5	40	2
T	50	2,5	50	3,5	57	5,13	55	5,5	45	2,25
X	1	0,05	1,5	0,105	3	0,27	2	0,2	0,5	0,025
r	0,1	0,005	0,4	0,028	0,5	0,045	0,3	0,03	0,05	0,0025
v	derived		derived		derived		derived		derived	
XIRR	Frequenza 25%		Frequenza 10%		Frequenza 10%		Frequenza 10%		Frequenza 70%	
	media	varianza	media	varianza	media	varianza	media	varianza	media	varianza
value	derived*		derived*		derived*		derived*		derived*	
days	fixed		fixed		fixed		fixed		fixed	
guess	0,1		0,1		0,1		0,1		0,1	
*importoCashFlowMean	1500	150	1700	85	2000	120	1800	144	2200	220
addebitoIniziale	-30000	3000	-35000	1750	-40000	2400	-38000	3040	-43000	4300

Tabella 6.2: Descrizione frequenza, media e varianza dei parametri di ingresso delle funzioni nella Fase 3.

Fase 3: Esecuzione funzioni con diversi valori di frequenza, media e varianza

Infine nella terza fase vengono eseguite le funzioni con le caratteristiche espresse in Tabella 6.2 e come nella fase precedente ne vengono acquisiti i dati tramite laptop. La Tabella 6.2 va letta nel seguente modo: sulla prima colonna ci sono i nomi delle funzioni con sotto i nomi dei loro parametri di ingresso, mentre nelle colonne successive è presente la frequenza di esecuzione di ogni funzione espressa in termini percentuali rispetto al numero totale di chiamate e i valore di media e varianza dei loro parametri di ingresso. Ogni colonna è un test. Le colonne successive alla prima infatti vanno considerate come cinque test differenti durante i quali vengono richiamate le funzioni in base alla percentuale di frequenza che presentano su un totale di 1 milione di possibili chiamate. Quindi, per fare un esempio, nella terza colonna si hanno 100.000 esecuzioni della funzione Fourier, 700.000 di Implied Volatility, 100.000 di Black Scholes e 100.000 dello XIRR, con valore di media e varianza dei parametri di ingresso espressi all'interno della colonna stessa e per un totale di 1 milione di chiamate di funzione.

I dati acquisiti in questa fase servono per valutare il comportamento del blocco di *Trade off* in seguito all'esecuzione delle funzioni con diversi valori di frequenza, media e varianza.

6.2.3 Risultati delle misure

In questa sezione vengono analizzati i risultati ottenuti in seguito all'introduzione del blocco di *Trade off* all'interno della metodologia della memoizzazione. Per fare questo si sono messi a confronto i dati ottenuti dalle misure descritte nella Sezione 6.2.2 nei tre casi trattati: funzioni originali, funzioni con memoizzazione e funzioni con memoizzazione e blocco di *Trade off*.

Come descritto nella sezione precedente nei due casi di memoizzazione e di memoizzazione con *Trade off* questi test sono stati eseguiti in seguito a una

fase di riempimento della memoria, con 40 milioni di chiamate equamente distribuite tra le funzioni, che porta a due configurazioni diverse mostrate in Figura 6.5 e 6.6.

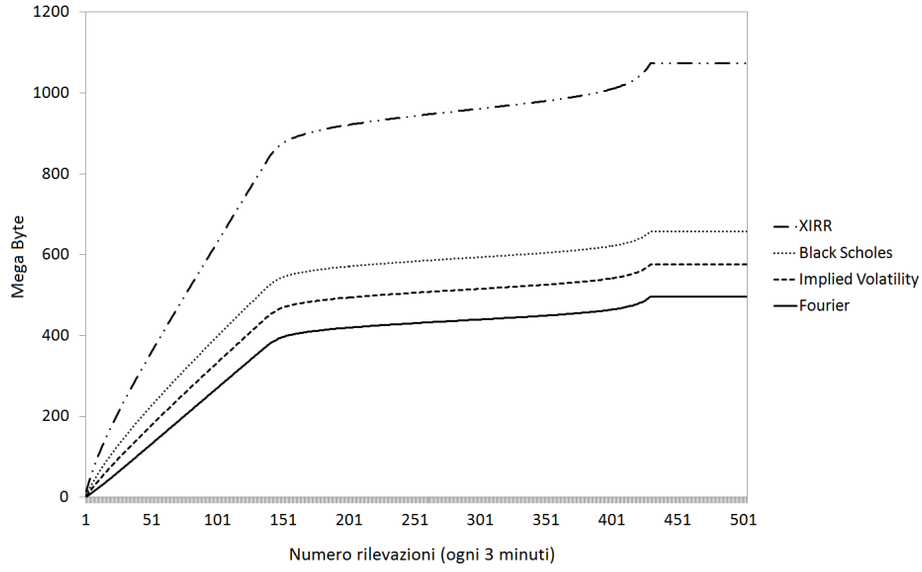

Figura 6.5: Allocazione della memoria in caso di sola memoizzazione.

In questa fase di riempimento si valuta l'impatto del blocco di *Trade off* sull'iniziale allocazione della memoria. Nel caso di sola memoizzazione (Figura 6.5) i dati relativi alle coppie parametri di ingresso risultato vengono salvati finchè la memoria diponibile per tutte le funzioni risulta piena.

Invece nella Figura 6.6 viene mostrata l'allocazione della memoria nel caso di memoizzazione con *Trade off* attivo. In questa si vede che non si riesce a raggiungere il tetto massimo di memoria, imposto come si è detto a 1 GB. Questo avviene perchè durante le chiamate del blocco di *Trade off* viene riallocata la memoria, in base al guadagno pesato delle funzioni, applicando cioè l'Algoritmo 2 descritto nel Capitolo 4. Per cui potrebbe succedere che ad una data funzione spetti meno memoria di quella precedentemente occupata e quindi si procede alla cancellazione dei suoi inserimenti, per mezzo del metodo analizzato nel Capitolo 5 nella Sezione 5.2.1.

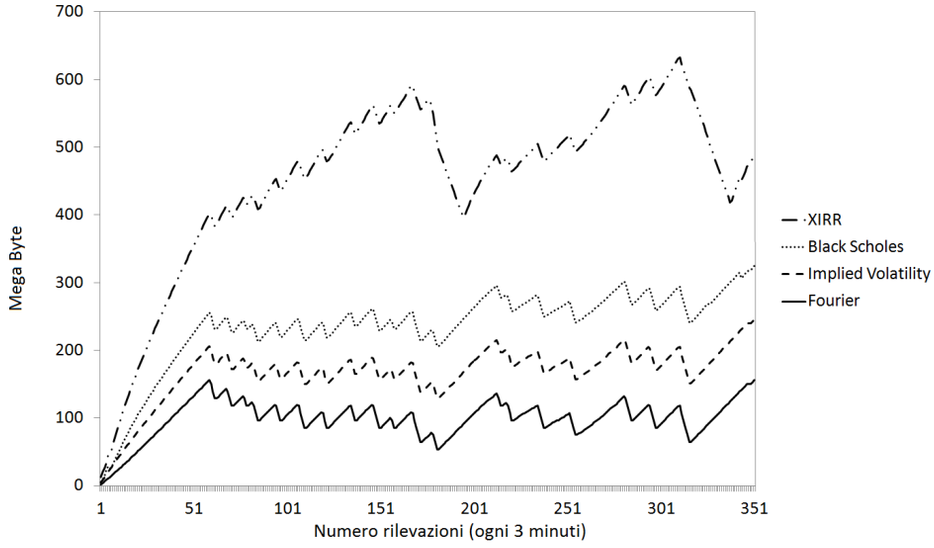

Figura 6.6: Allocazione della memoria in caso di memoizzazione con Trade off.

Si ottiene, per mezzo dell'utilizzo del *Trade off*, un'allocazione della memoria migliore dettata dalle caratteristiche dei parametri di ingresso e dalla frequenza di chiamata delle funzioni. Per valutare questo miglioramento si veda la Figura 6.7 che mostra il valore del guadagno totale delle due configurazioni nel tempo.

In seguito a questa fase di riempimento della memoria vengono effettuate le misure del consumo di potenza mostrate nelle figure da 6.8 a 6.12.

I risultati in queste figure sono stati ottenuti eseguendo 1 milione di chiamate di funzione con i valori della varianza dei parametri di ingresso visti in Tabella 6.2 e per ognuno dei seguenti valori di frequenza di chiamata:

- Test Fourier 25% Implied Volatility 25% Black Scholes 25% XIRR 25% (Figura 6.8)
- Test Fourier 70% Implied Volatility 10% Black Scholes 10% XIRR 10% (Figura 6.9)
- Test Fourier 10% Implied Volatility 70% Black Scholes 10% XIRR 10% (Figura 6.10)

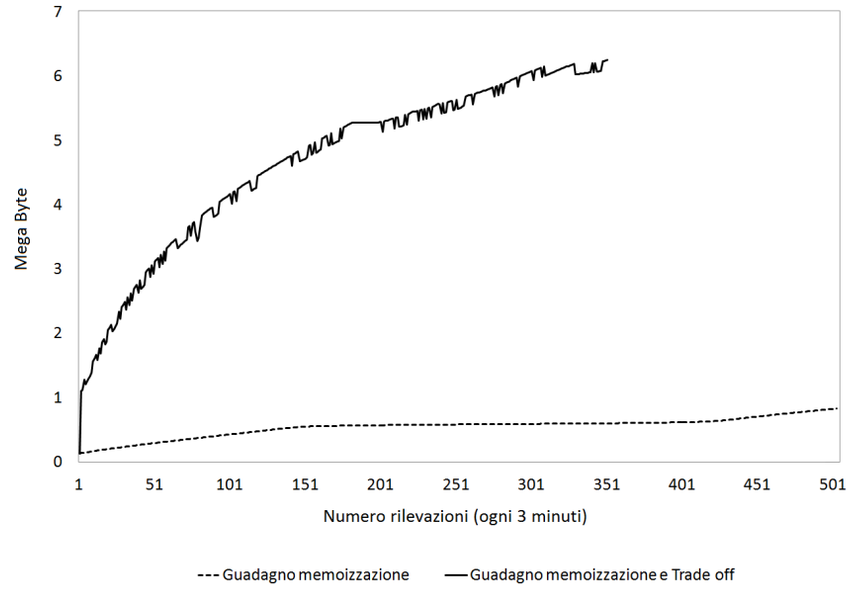

Figura 6.7: Guadagno totale nel tempo con memoizzazione e con memoizzazione con Trade off.

- Test Fourier 10% Implied Volatility 10% Black Scholes 70% XIRR 10% (Figura 6.11)

- Test Fourier 10% Implied Volatility 10% Black Scholes 10% XIRR 70% (Figura 6.12)

Come si può vedere nelle figure i grafici rappresentano l'andamento nel tempo della potenza, espressa in milliWatt (mW), nei tre casi descritti in alto a destra nella legenda.

Dai test effettuati si ottiene come risultato che l'introduzione del *Trade off* porta ad una riduzione sia dei tempi di esecuzione che dei consumi energetici. Tale miglioramento si verifica sia rispetto al caso in cui si usino le funzioni originali sia nel caso si applichi la sola memoizzazione.

Vengono inoltre presentati i risultati schematizzati nelle Tabelle 6.3 e 6.4 dove viene mostrata l'energia consumata e il tempo impiegato per effettuare i test nei diversi casi. Viene messo a confronto il risparmio percentuale prima

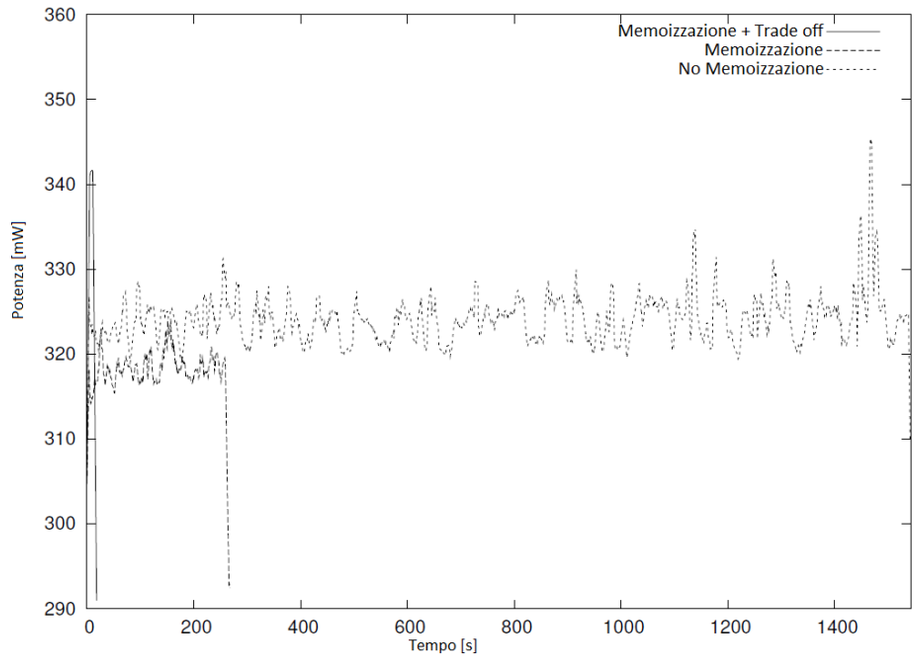

Figura 6.8: Test Fourier 25% Implied Volatility 25% Black Scholes 25% XIRR 25%.

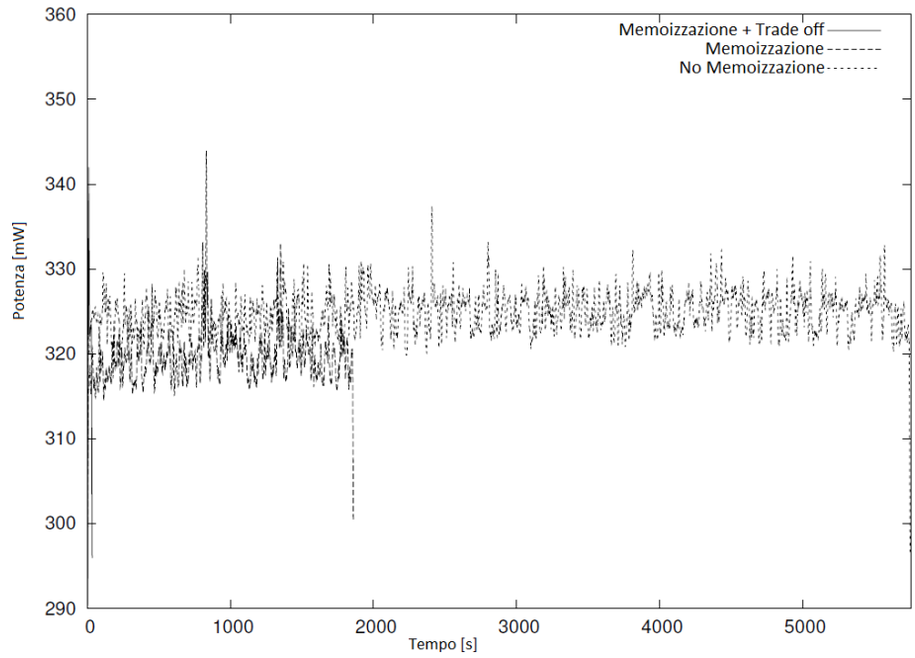

Figura 6.9: Test Fourier 70% Implied Volatility 10% Black Scholes 10% XIRR 10%.

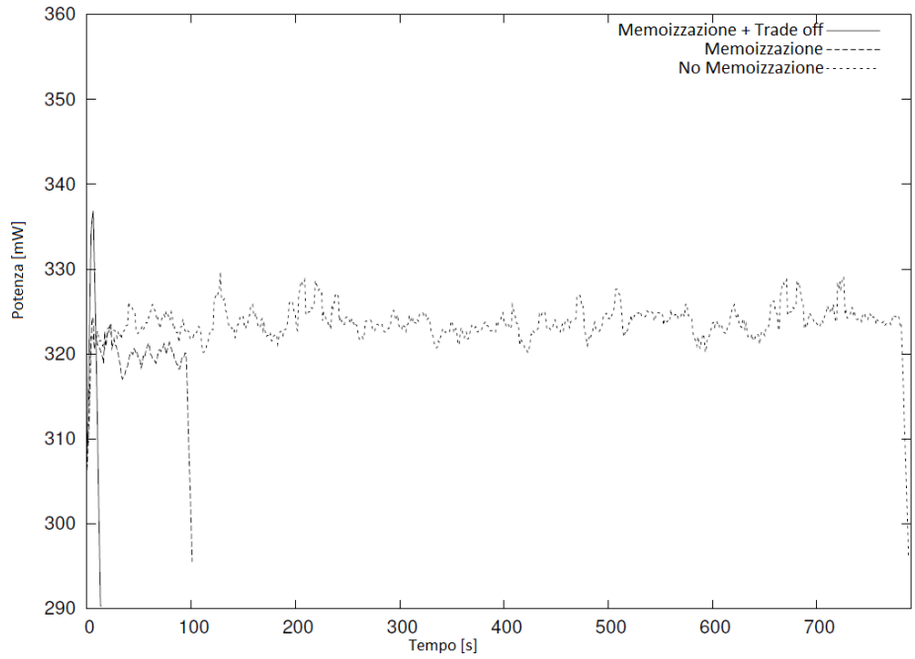

Figura 6.10: Test Fourier 10% Implied Volatility 70% Black Scholes 10% XIRR 10%.

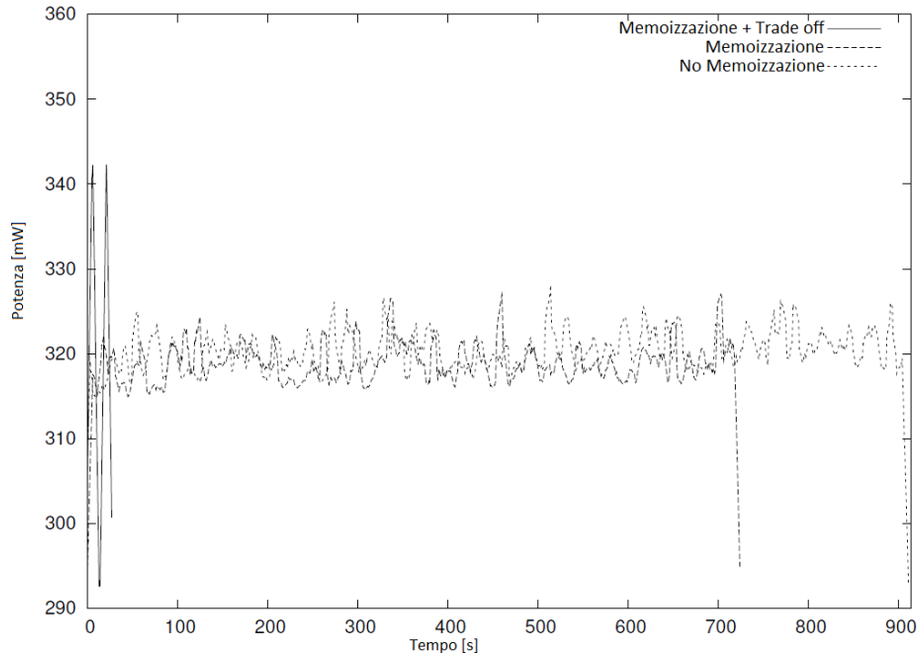

Figura 6.11: Test Fourier 10% Implied Volatility 10% Black Scholes 70% XIRR 10%.

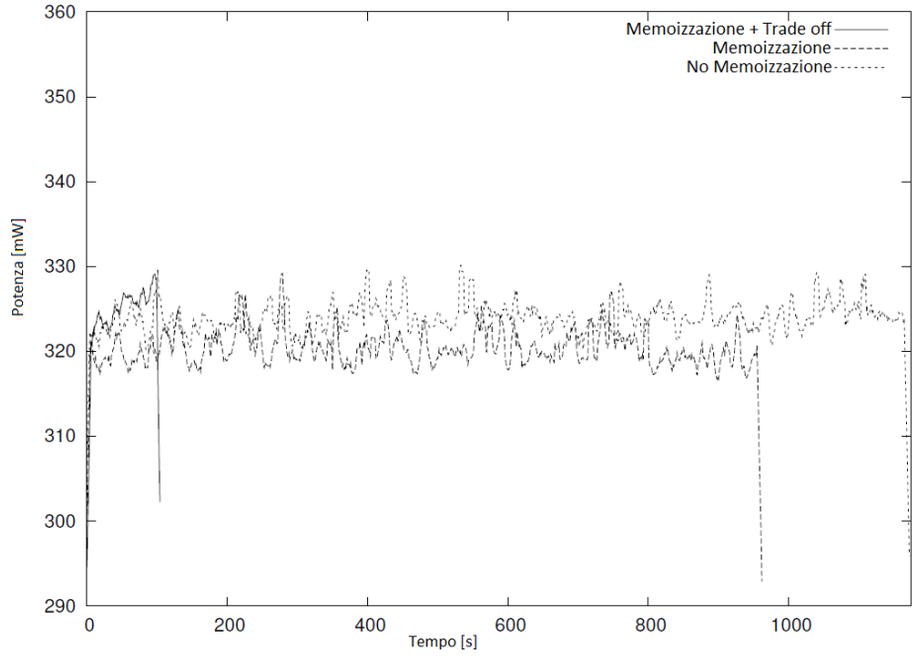

Figura 6.12: Test Fourier 10% Implied Volatility 10% Black Scholes 10% XIRR 70%.

del caso as-is, con le funzioni originali, rispetto all'utilizzo della memoizzazione e poi il caso della memoizzazione con *Trade off* rispetto agli altri due casi. Come possiamo vedere, il risparmio energetico medio tra il caso as-is e il caso con memoizzazione con *Trade off* è del 96.8%, con un picco del 98.79%. Invece, il risparmio medio sul tempo di esecuzione è del 97%, con un picco del 99%.

Quindi l'utilizzo della memoizzazione con in aggiunta il blocco di *Trade off* presentato in questa Tesi porta ad un notevole risparmio energetico. Anche applicando la sola memoizzazione si può notare un risparmio rispetto al caso in cui le funzioni vengano eseguite nella loro versione originale. Ma la differenza sostanziale tra la sola memoizzazione e la memoizzazione con il *Trade off* sta nel modo in cui è stato costruito il test al fine di poter rappresentare una situazione reale.

In riferimento alla Figura 6.5 possiamo vedere che durante le Fase 2

Tests F IV BS X (%)	Energia as-is (J)	Energia con memoizzazione (J)	Risparmio as-is vs memo (%)	Energia con memoizzazione + trade-off (J)	Risparmio memo vs memo+TO (%)	Risparmio totale as-is vs memo+TO (%)
25 25 25 25	47.556,58	5.539,93	88,35%	574,97	89,62%	98,79%
70 10 10 10	141.296,21	49.921,74	64,67%	1.345,75	97,30%	98,79%
10 70 10 10	19.859,22	2.639,37	86,71%	410,80	84,44%	97,93%
10 10 70 10	25.536,81	19.827,62	22,36%	418,14	97,89%	98,36%
10 10 10 70	31.626,52	24.135,51	23,69%	3.175,51	86,84%	89,96%

Tabella 6.3: Energia consumata durante i test presi in cosiderazione.

Tests F IV BS X (%)	Tempo di esecuzione as-is (s)	Tempo di esecuzione con memoizzazione (s)	Risparmio as-is vs memo (%)	Tempo di esecuzione con memoizzazione + trade-off (s)	Risparmio memo vs memo+TO (%)	Risparmio as-is vs memo+TO (%)
25 25 25 25	1.659,63	216,74	86,94%	18,37	91,52%	98,89%
70 10 10 10	5.125,10	1,840,10	64,10%	51,30	97,21%	99,00%
10 70 10 10	781,64	103,35	86,78%	17,66	82,91%	97,74%
10 10 70 10	890,14	746,80	16,10%	15,45	97,93%	98,26%
10 10 10 70	1.130,42	933,55	17,42%	100,83	89,20%	91,08%

Tabella 6.4: Tempi impiegato per effettuare i test.

la memoria è stata riempita totalmente, situazione facilmente riscontrabile nella realtà se il volume di chiamate di funzione è elevato. In questo caso però inizia la Fase 3 nell'impossibilità di salvare ulteriori dati, utili per poter applicare la memoizzazione con tutti i benefici connessi. Infatti, i dati creati nella Fase 1 necessari per svolgere i test nella Fase 3, sono potenzialmente differenti da quelli generati sempre nella Fase 1 per riempire la memoria nella Fase 2, proprio perchè creati con una distribuzione gaussiana ma con valori di media e varianza dei parametri di ingresso diversi. Questa situazione è dovuta alle politiche di utilizzo della memoria statiche adottate in assenza di *Trade off* che portano, in fase di riempimento della memoria, al salvataggio di qualsiasi tipo di coppia parametri di ingresso e risultato, anche se tale coppia non viene chiamata con un frequenza elevata.

Al contrario, utilizzando il *Trade off* (Figura 6.6), è possibile gestire la memoria in maniera più flessibile, in modo tale che le funzioni che in un dato momento apportano maggiore guadagno possano avere maggiore spazio in memoria. Inoltre, essendo il blocco richiamato più volte nel tempo, è possibile aggiornare l'allocazione della memoria in relazione alle variazioni di frequenza di chiamata delle funzioni e in relazione alla varianza dei loro parametri di ingresso. Viceversa, quelle che apportano meno guadagno, devono subire una penalizzazione in termini di riduzione della memoria a loro concessa. Nel caso queste funzioni abbiano occupato in precedenza un quantitativo di memoria maggiore rispetto a quella correntemente concessa, si procede alla cancellazione di alcuni loro inserimenti richiamati con bassa frequenza. Come conseguenza, riscontrabile in Figura 6.6, si ha una maggiore area di memoria disponibile per il salvataggio di dati che in futuro potrebbero avere un'alta frequenza di chiamata.

Nel test di memoizzazione con *Trade off* non si è arrivati a riempire tutta la memoria disponibile. Però, nel caso in cui si arrivasse alla saturazione della memoria, per mezzo del *Trade off* quest'ultima sarebbe gestita comunque

in maniera migliore rispetto alla gestione statica vista nel caso di utilizzo della sola memoizzazione, dove una volta riempita la memoria in una data configurazione, la situazione rimane immutata nel tempo. Invece, grazie al blocco sviluppato in questo lavoro di Tesi, l'occupazione dello spazio in memoria è variabile nel tempo e, quindi, anche la situazione di memoria satura non risulta definitiva ma è soggetta a cambiamenti nel tempo, in relazione al guadagno apportato da ciascuna funzione.

Capitolo 7

Conclusioni e sviluppi futuri

Questo lavoro di Tesi trova posto all'interno di un filone di ricerca orientato alla riduzione del consumo energetico, per mezzo della modifica, tramite la tecnica della memoizzazione, di software già creato.

L'architettura, basata sulla tecnica di memoizzazione, trova posto all'interno del Green IT e in particolare del Green Software e rende possibile l'ottimizzazione del software eseguito in tempo reale, senza la necessità di analizzare precedentemente il codice, che per altro potrebbe essere di terze parti e quindi non disponibile.

La tecnica di memoizzazione consiste nel salvataggio in memoria dei risultati delle funzioni, in modo tale da riutilizzarne i valori in futuro, in caso di una chiamata di funzione con gli stessi parametri di ingresso. Applicando questa tecnica la CPU non deve rifare calcoli già eseguiti in passato, perchè verrà demandato alla memoria il compito di ricavare i risultati.

Nel dettaglio questo lavoro di Tesi si è incentrato sulla definizione di un modello per l'allocazione dinamica della memoria, in base al quale viene successivamente implementato il blocco di *Trade off* posto all'interno dell'architettura di memoizzazione. Lo scopo del modello è la riduzione del consumo energetico per mezzo della tabulazione dei risultati delle funzioni

pure precedentemente selezionate secondo la tecnica definita in [9]. Il modello risulta necessario per la presenza di più funzioni che occupano la memoria in maniera concorrente.

Durante questo lavoro di Tesi sono stati implementati dei metodi necessari per il corretto funzionamento dell'architettura complessiva. All'interno della struttura `MemoizationHashSFArr` è stato creato un metodo per la cancellazione delle coppie parametri di ingresso e risultato. Questo viene richiamato quando, in seguito ad una riallocazione della memoria effettuata a opera del *Trede off*, una funzione presenta un'occupazione della memoria maggiore di quella correntemente concessa. É stato creato anche un metodo `sizeOF` , non presente in Java, utile in fase di contrattazione della memoria perchè permette di conoscere le dimensioni di due oggetti contenenti uno i parametri di ingresso e l'altro il risultato. La conoscenza di tale dimensione è necessaria per il corretto funzionamento del blocco di *Trade off* in quanto ogni funzione è diversa e così anche l'occupazione in memoria di un suo singolo salvataggio. Entrambi i metodi sono risultati funzionanti e in fase di testing non hanno presentato side effect.

L'introduzione del blocco ha portato dei miglioramenti sia in termini di tempo che in termini di risparmio energetico.

La gestione della memoria prima dell'introduzione di questo blocco era di tipo statico, dove si salvavano coppie parametri di ingresso e risultato solo finchè c'era spazio in memoria e una volta arrivati a saturazione la ripartizione della memoria tra le funzioni rimaneva fissa. Grazie al *Trade off* sviluppato in questo lavoro la gestione della memoria è diventata dinamica, cioè sensibile alle variazioni di frequenza di chiamata e varianza dei parametri di ingresso delle funzioni. Infatti, quest'ultimo, viene richiamato prima dell'esecuzione delle funzioni e, successivamente, in fase di run time, a intervalli regolari. Il suo scopo è quello di suddividere la memoria disponibile tra le funzioni pure utilizzando come metrica di giudizio il guadagno. In seguito

all'introduzione del blocco di *Trade off* si è ottenuto un risparmio energetico medio, rispetto all'esecuzione delle funzioni originali, del 96.8%, con un picco del 98.79%. Invece, il risparmio medio sul tempo di esecuzione è stato del 97%, con un picco del 99%.

7.1 Sviluppi futuri

Il lavoro di Tesi fin qui discusso pone le basi per future implementazioni. Potrebbe essere implementato un sistema di riconoscimento di funzioni pure a run time invece di una singola analisi prima dell'esecuzione della metodologia di memoizzazione. Come conseguenza di questa modifica sarebbe necessario aggiornare il blocco di *Trade off* affinchè possa gestire in modo corretto gli inserimenti in memoria della nuova funzione. Infatti questa nuova funzione inizialmente presenterà un valore di frequenza basso, perchè sarà richiamata per la prima volta, ed essendo il blocco di *Trade off* basato sulla metrica del guadagno pesato con la frequenza, la funzione risulterebbe penalizzata perchè non le sarebbe mai concesso spazio. Una possibile estensione all'architettura di memoizzazione potrebbe essere uno studio sui Service Level Agreement (SLA) riguardanti la precisione del risultato, con una conseguente valutazione del consumo energetico ai vari livello di servizio. Al fine di ottenere una più ampia conoscenza del comportamento della metodologia della memoizzazione, potrebbero essere condotti studi in merito al suo comportamento al variare dell'hardware preso in considerazione. In particolare sarebbe interessante osservarne il comportamento al variare del tipo di memoria cache utilizzata dal processore, sia in termini di livelli (L1 o L2), sia in termini di tipologia (fully associative, direct mapped o set associative). Un ulteriore sviluppo, di ben più grande impatto, sarebbe il riutilizzo dei moduli di*Pure function retrieval* e di *Bytecode modification* per la progettazione di una "JVM green". Questa JVM dovrebbe poter permettere la ricerca delle funzioni pure, la gestione delle stesse in memoria tramite la memoizzazione

e l'ottimizzazione del consumo energetico in modo automatico senza alcuna pre-analisi su un insieme di applicazioni adatte allo scopo e precedentemente selezionate.

Bibliografia

[1] Alberio A. e Brianza B. Un'analisi esplorativa sull'ottimizzazione del consumo energetico del software. Master's thesis, Politecnico di Milano, 2008-2009.

[2] C. Francalanci e S. Gallazzi E. Capra, G. Formenti. The impact of mis software on it energy consumption. In *18th European Conference on Information Systems*, 2010.

[3] F. Black e M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, 81(3):637–654, 1973.

[4] Katsumi F. e Maica Reis I. A software caching metodology for power optimization in data centers. Master's thesis, Politecnico di Milano, 2009-2010.

[5] G. Formenti e S. Gallazzi. A methodology to evaluate empirically software energy consumption and its impact on total cost of ownership. Master's thesis, Politecnico di Milano, 2007-2008.

[6] C. Francalanci and E. Capra. Green it: Sfide e opportunitá. *Mondo Digitale*, 4:36–42, 2008.

[7] Eugenio Capra Giovanni Agosta. Green software: anche le applicazioni consumano energia. *Mondo Digitale*, 1:16, 2011.

[8] Planer I. Business intelligence methodologies applied to green it. Master's thesis, Politecnico di Milano, 2009-2010.

[9] Bessi M. Una metodologia basata sulla memoizzazione dinamica per l'efficienza energetica del software applicativo. Master's thesis, Politecnico di Milano, 2009-2010.

[10] Naveen Sastry e David Wagner Matthew Finifter, Adrian Mettler. Verifiable functional purity in java. In *In CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174, 2008.

[11] S. Mayhew. Implied volatility. *Financial Analysts Journal*, 51(4):8–20, 1995.

[12] D. Michie. Memo functions and machine learning. *Nature*, 218(1):19–22, 1968.

[13] S. Murugesan. Harnessing green it: Principles and practices. *IT professional*, 10(1):24–33, 2008.

[14] T. Restorick. An inefficient truth. Technical report, Global Action Plan Report, 2007.

[15] M. Schoeberl. Jop: A java optimized processor. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 346–359, 2003.

Appendice A

Definizione del metodo sizeOf

Nel Capitolo 4 è stata presentata la logica del modulo di *Trade off* e si è visto quanto sia importante la gestione della memoria disponibile. Negli algoritmi 1 e 2 presentati nello stesso capitolo viene utilizzato il parametro $S_{d,i}$ che indica le dimensioni della singola coppia parametri di ingresso e risultato per l'i-esima funzione. Tale valore è necessario per gestire la memoria, evitando di eccedere le dimensioni massime concesse per l'esecuzione delle funzioni o quelle massime imposte dell'hardware. Infatti, moltiplicando la singola dimensione per il numero di ingressi si ottiene il valore $S_{m,i}$, poi sommando quest'ultimo per ogni funzione i si ottiene l'occupazione della memoria fino a quel momento.

I parametri in ingresso alla funzione vengono salvati all'interno di un oggetto di tipo `TOParameters`, mentre il risultato restituito dalla funzione viene salvato in un oggetto di tipo `Return`. Il numero dei parametri di ingresso alla funzione è variabile, come anche il loro tipo. Volendo trovare il valore di $S_{d,i}$ per l'i-esima funzione, si devono trovare le due dimensioni degli oggetti, `TOParameters` e di `Return`, e sommarli.

Il linguaggio di programmazione utilizzato all'interno di questo lavoro di Tesi è Java. Java è un linguaggio di programmazione orientato agli oggetti,

derivato dallo Smalltalk e creato da ingegneri di Sun Microsystems. I programmi scritti in linguaggio Java sono destinati all'esecuzione sulla piattaforma Java, ovvero saranno lanciati su una Java Virtual Machine e, a tempo di esecuzione, avranno accesso alle API della libreria standard. La memoria in Java non viene gestita dai programmatori bensì dal Garbage Collector (GC), descritto nella Sezione A.1. Non potendo gestire direttamente la memoria, e in particolare non essendoci un metodo *sizeof* come nel C, non è possibile ricavare le informazioni necessarie per calcolare $S_{d,i}$; sono stati quindi creati due metodi per svolgere questo compito che saranno analizzati nella Sezione A.2.

A.1 Garbage Collector

Garbage Collection è il nome dato al meccanismo che tiene traccia delle locazioni di memoria utilizzate da un programma e si occupa di renderle nuovamente disponibili non appena possibile. In linguaggi come Java che, a differenza di altri linguaggi (vedi il C o il C++), non danno la possibilità al programmatore di accedere in maniera diretta a una qualsiasi locazione di memoria, tale meccanismo assume un ruolo strategico in termini di efficienza, prestazioni, ottimizzazione della memoria. In Java non esistono puntatori del tipo di quelli del C/C++; un puntatore è un mero riferimento a un oggetto allocato dinamicamente su uno heap. Java libera il programmatore dall'incombenza di dover cancellare gli oggetti precedentemente creati. Infatti sarà il Garbage Collector a preoccuparsi di tutto; sarà lui a controllare costantemente quanti riferimenti vi sono relativi a un oggetto; sarà lui a liberare lo spazio occupato dall'oggetto quando esso non sarà più accessibile né necessario. Tutto ciò si traduce sicuramente in una maggiore semplicità di programmazione, infatti tutti gli eventuali errori dipendenti da una gestione manuale della memoria vengono eliminati in partenza. Come contro però si ha a volte un notevole peggioramento delle prestazioni. Il Garbage

Collector è una speciale routine, un vero e proprio thread a bassa priorità che viene eseguito parallelamente al programma principale, da considerare attentamente in termini di tempo di utilizzo della CPU, specialmente se si hanno dei vincoli di tipo real time.

Il Garbage Collector ideale dovrebbe riuscire ad allocare l'esatta quantità di memoria richiesta, al momento opportuno, senza sprechi. Dovrebbe essere in grado di rendere immediatamente disponibile al programma la memoria occupata da un oggetto non più necessario all'applicazione, ovvero irragiungibile da essa. Il tutto, ovviamente, senza interferire con il programma principale in termini utilizzo della CPU.

Volendo passare da un Garbage Collector ideale ad uno un po' più realistico, quest'ultimo dovrà essere in grado di distinguere gli oggetti *live*, ovvero quelli potenzialmente ancora raggiungibili dall'applicazione, da quelli *garbage*, ovvero quelli non più raggiungibili e quindi collezionabili. Dovrà essere, inoltre, compito del Garbage Collector capire se e quando liberare l'eventuale spazio occupato da oggetti ormai inutili presenti in memoria, rendendolo nuovamente disponibile all'applicazione. Potremmo essere, infatti, nel caso di una macchina con una tale quantità di memoria disponibile da rendere inutile l'azione del Garbage Collector (potrebbe essere più conveniente sprecare un po' più di memoria, a vantaggio di maggiori prestazioni in termini di tempo di CPU assegnato all'esecuzione del thread principale). A questo proposito, sorge la necessità di introdurre uno dei concetti fondamentali in ambito di Garbage Collection. Si parla, allora, di conservatism per indicare l'atteggiamento che, in generale, un valido Garbage Collector dovrebbe possedere in merito alla determinazione degli oggetti garbage. Meglio mantenere, dunque, in memoria un oggetto dalla liveness dubbia, piuttosto che collezionarlo, ovvero eliminarlo dalla memoria, portando l'intero sistema in uno stato di precaria stabilità. Il grado di cautela di un GC differisce da implementazione ad implementazione e influisce pesantemente sulle perfo-

mance del GC stesso. Il Garbage Collector sarà da considerare, almeno in questi termini, una sorta di daemon, non gestito dal programmatore, difficilmente controllabile, sempre presente in background, in delle modalità che poco si conciliano con delle specifiche di tipo *real time*. Quindi se in un primo momento si era pensato di utilizzare il Garbage Collector per ricavare la dimensione di un determinato oggetto, in seguito si è abbandonata questa scelta proprio per l'impossibilità di controllare la sua esecuzione. Si è optato invece per la soluzione delineata nella sezione successivo.

A.2 Metodo sizeOf

Non essendo presente in Java un metodo equivalente alla `sizeof` del C o del C++, sono stati creati due metodi all'interno della classe *TOFunction* con il compito di ricavare le dimensioni degli oggetti del tipo TOParameters e Return necessari per calcolare $S_{d,i}$.

Listing A.1: Metodo sizeOf per il calcolo della dimensione di un oggetto di tipo TOParameters contenente i parametri di ingresso della funzione.

```
312
313     public static int sizeOf(TOParameters top){
314       int size = 8;
315       if(top.toStringClass()!=null){
316         String s = top.toStringClass();
317         s = s.replace("Byte", "8+1");
318             s = s.replace("Short", "8+2");
319             s = s.replace("Char", "8+2");
320             s = s.replace("Integer", "8+4");
321             s = s.replace("Float", "8+4");
322             s = s.replace("Long","8+8");
323             s = s.replace("Double","8+8");
324             s = s.replace("Array", "4");
325             String ss[]=s.split("\\+");
```

```
326             for(int i=0;i<ss.length;i++)
327               size+=Integer.parseInt(ss[i]);
328       }
329       return size;
330     }
```

Il codice riportato nel Listato A.1 viene utilizzato per determinare le dimensioni di un oggetto di tipo `TOParameters`. Il calcolo della dimensione dell'oggetto avviene sommando le singole dimensioni dei parametri di ingresso alla funzione, in base al loro tipo. Vanno inoltre considerate anche le dimensioni aggiuntive dovute alla creazione degli oggetti e degli array.

La Tabella A.1 riporta la quantità di memoria necessaria per il salvataggio dei tipi primitivi, degli oggetti e degli array.

Tipo	Dimensioni (byte)
byte	1
short	2
char	2
int	4
float	4
long	8
double	8
Object	8

Costrutto	Dimensioni (byte)
Array	4

Tabella A.1: Quantità di memoria necessaria per il salvataggio dei tipi primitivi, degli oggetti e degli array.

É da notare che ogni singolo ingresso presente nella variabile `top` viene inglobato dentro un oggetto. Nel considerare per esempio l'occupazione in

memoria di una variabile di tipo int si deve sommare alla dimensione dell'intero, ovvero 4 byte, la dimensione dell'oggetto che lo contiene, 8 byte; quindi un oggetto `Integer` ha un'occupazione di 12 byte.

Nel dettaglio il metodo `sizeOf` inizializza la variabile `size`, valore che alla fine viene restituito, a 8, in riferimento all'oggetto top, successivamente richiama il metodo `toStringClass()` di `top` per avere informazioni sui tipi dei parametri al suo interno. Il metodo `toStringClass()` restituisce una stringa composta dai nomi dei tipi dei parametri divisi con il separatore "+". Per fare un esempio del suo funzionamento, posto che i parametri contenuti in `top` siano un `Double`, un `Integer` e un `Array` contenente tre `Float` si ha la seguente stringa: `Double+Integer+Array+Float+Float+Float`. Noti i tipi dei parametri di ingresso basta sommarne il valore e restituirlo al programma chiamante.

La stessa logica viene applicata al metodo presentato in Listato A.2, con la sola differenza che si ha un solo dato all'interno dell'oggetto di tipo `Return`.

Listing A.2: Metodo sizeOf per il calcolo della dimensione di un oggetto di tipo Return contenente il valore restituito in uscita dalla funzione.

```
332 public static int sizeOf(Return ret){
333       int size = 8;
334       if(ret.getType().toString().replace("class
335           java.lang.", "").equals("Byte")){
336         return size + 8 + 1;
337       }else if(ret.getType().toString().replace("class
338           java.lang.", "").equals("Short")){
339         return size + 8 + 2;
340       }else if(ret.getType().toString().replace("class
341           java.lang.", "").equals("Char")){
342         return size + 8 + 2;
343       }else if(ret.getType().toString().replace("class
344           java.lang.", "").equals("Integer")){
345         return size + 8 + 4;
```

```
346       }else if(ret.getType().toString().replace("class
347           java.lang.", "").equals("Float")){
348         return size + 8 + 4;
349       }else if(ret.getType().toString().replace("class
350           java.lang.", "").equals("Long")){
351         return size + 8 + 8;
352       }else if(ret.getType().toString().replace("class
353           java.lang.", "").equals("Double")){
354         return size + 8 + 8;
355       }else
356         return size;
357     }
```