POLITECNICO DI MILANO

FACOLTA' DI INGEGNERIA DELL'INFORMAZIONE
Corso di laurea specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Memoizzazione: studio sperimentale sull'impatto dovuto alla variazione dei parametri dei componenti di memoria di un'architettura ARM.

Relatore: Prof: Ing. Chiara FRANCALANCI
Correlatore: Prof. Ing. Giovanni AGOSTA
Ing. Marco BESSI

Tesi di laurea di:
Marco ZAPPALA'
Matricola 755074

Anno Accademico 2011-2012

A Vita e Sergio,
il cui cuore
è sempre stato un porto sicuro
nel mare burrascoso della mia esistenza.

ABSTRACT

Regarding the green software research and development area, the following thesis aims to the estimation of the impacts induced by the variation of different parameters of an ARM architecture memory components applied and affecting a dynamic memoization methodology which intends to reduce the energy consumptions of computing intensive applications. This methodology has been developed at Politecnico di Milano.

This script describes the different experiments carried out, which have been realized through the implementation of the GEM5 simulator. The architectures have been simulated using different specific and mapped components in relation to some actual solutions available on the market for the servers inside a medium-high product range.

The results coming from the script clearly allow to state that NO substantial variations occur in the execution times performance for the selected benchmarks neither if the variation is applied to the numbers of hierarchical levels of cache memory, neither if the variation is applied to the different cache dimensions, neither if the variation is in the latencies of the memory components including the RAM.

Then, it can be also stated that considering the memory components and latency frequencies actually offered on the market, the availment of the technique described above, in relation to the run of the original benchmarks, is to be considered always advantageous given that enables execution time savings estimated in about one order of magnitude.

SOMMARIO

Nell'ambito di ricerca dello sviluppo di software energeticamente efficiente, si è cercato di valutare l'impatto che la variazione di diversi parametri dei componenti di memoria di un'architettura ARM possano avere su una metodologia che sfrutta il concetto di memoizzazione dinamica per ridurre il consumo d'energia di applicazioni computing intensive, sviluppata al Politecnico di Milano.
L'elaborato presenterà i diversi esperimenti, effettuati attraverso l'utilizzo del simulatore GEM5, in cui sono state simulate architetture con parametri dei diversi componenti mappati rispetto ad una ricerca sulle offerte attuali di mercato per i server di fascia medio-alta.
I risultati hanno permesso di affermare che NON ci sono variazioni sostanziali sui tempi di esecuzione dei benchmark scelti, né facendo variare il numero di livelli gerarchici di memorie cache, nè facendo variare le dimensioni delle varie cache e nemmeno variando le latenze dei componenti di memoria, compresa la RAM. È stato invece possibile affermare che, con gli attuali componenti di memoria e frequenze offerte sul mercato, l'utilizzo della tecnica di cui sopra, rispetto all'esecuzione dei benchmark originali, è sempre vantaggiosa e permette risparmi di tempo di almeno un ordine di grandezza inferiori.

INDICE

LISTA DELLE TABELLE

LISTA DEI GRAFICI

LISTA DELLE FIGURE

1. INTRODUZIONE

L'individuazione e lo sviluppo di metodologie, tecniche e tecnologie sostenibili per l'ambiente è un'attività ormai principe in tutti gli ambiti di ricerca tecnici, aldilà delle materie in oggetto. In questo contesto anche l'Information Technology (IT) vanta numerose branche di ricerca nell'ambito del cosiddetto “Green IT”, in cui si individua una categorizzazione in “Green Hardware” e “Green Software”.
Nel progetto di ricerca, sfociato anche nella definizione di questo elaborato, ci si è inseriti nella seconda categoria delle precedenti due citate. In particolare, come base di partenza, si è preso un lavoro di ricerca, in parte già svolto ed ancora in corso, portato avanti dal Dipartimento di Elettronica ed Informazione del Politecnico di Milano. Questo lavoro è sfociato ad ora nella definizione di un framework che attua una particolare tecnica, detta di “**memoizzazione**”, nell'ambito di sviluppo energeticamente efficiente del software. In questo ambito, si è andati a **valutare l'impatto che diverse gerarchie e strutture di memoria su architetture ARM possano avere su questa tecnica**.

Nel **secondo** capitolo sarà introdotto il concetto di “sviluppo sostenibile in ambito IT” nonché la differenziazione fra “green HW” e “green SW”. Partendo da valutazioni generiche su questi temi saranno, esposte le forti motivazioni che portano a studi in questi campi e possibili risvolti o ricerche attualmente in atto in entrambi le categorie.

Nell'ambito del “Green SW” saranno poi introdotti ed esemplificati, nel **terzo** capitolo, la metodologia basata sul concetto di memoizzazione ed il framework sopra citato, esponendo le valutazioni nonché le ipotesi di fondo che hanno portato alla definizione dello stesso, con enfasi su determinati argomenti che sono in relazione all'effettivo lavoro di ricerca oggetto di questa tesi. In questo capitolo sarà inoltre presentata l'architettura ARM e le motivazioni che hanno portato alla sua scelta, per le analisi svolte durante il lavoro di ricerca.

Il **quarto** ed il **quinto** capitolo riguardano proprio il progetto svolto. In particolare, il primo dei due, darà una panoramica della fase di “setup” del progetto e delle scelte progettuali iniziali fatte, con relative motivazioni. Saranno esposte le necessità presentatesi per raggiungere lo scopo del progetto e che hanno portato alla scelta dei diversi tool, oltre che all'individuazione sul mercato delle offerte proposte attualmente, in termini di componentistica delle memorie di server commerciali, che verranno presi come punto di riferimento per le successive simulazioni ed analisi.
Il secondo dei due si occuperà invece di esporre le effettive attività del progetto svolto che comprendono: una prima fase di adattamento del simulatore utilizzato a certe caratteristiche, necessarie per svolgere le simulazioni utili per il lavoro di ricerca, e il setup della macchina server su cui eseguire il simulatore; ed una seconda fase con il percorso dettagliato delle simulazioni effettuate, dei risultati ottenuti e delle relative valutazioni tecniche.

Il **sesto** e ultimo capitolo riguarderà invece le conclusioni del lavoro, le valutazioni anche in chiave economica ed i possibili sviluppi futuri di ricerca.

2. SOSTENIBILITÀ E GREEN IT

2.1 Sviluppo sostenibile

In un pianeta in continuo sviluppo ed espansione sotto tanti punti di vista, la limitatezza delle risorse ha reso sempre più forte ed attuale la necessità di tecniche e metodologie di sviluppo sostenibili per l'ambiente. In questo contesto è assolutamente non trascurabile l'evoluzione e l'espansione sempre più dirompente della tecnologia informatica, intendendo tutti quei dispositivi che sono stati, e continuano ad essere, motori di una nuova rivoluzione che ha cambiato anche in parte i modi di vivere.
L'informatica e le sue tecnologie, per quanto siano utili in molti ambienti per agevolare e velocizzare qualsivoglia tipo di attività o per soddisfare più o meno direttamente dei bisogni, hanno incrementato la necessità di energia elettrica richiesta per adempiere ai propri scopi. La pervasività degli apparati informatici ha però reso questa richiesta di energia non sostenibile o, perlomeno, non per molto ancora.

Alcune recenti ricerche informatiche vengono in sostegno di quanto appena detto. In particolare, per esempio, secondo i risultati raggiunti da GARTNER e FORRESTER (2007) sul totale delle emissioni globali di diossido di carbonio circa il 2% è dovuto all'IT [1]. Un valore spaventoso se si pensa che questo è paragonabile all'inquinamento prodotto da un'industria aeronautica.
Un altro dato significativo proviene dalle stime fatte nel 2006 dall'IDC (Intelligent Distributed Computing) secondo cui il costo sostenuto per raffreddare e alimentare un server corrisponde a più del 70% del costo sostenuto per l'acquisto dello stesso [2], il ché, detto in altri termini, significa che del TOTAL COST OF OWNERSHIP una grandissima parte è imputabile al consumo di energia del sistema.

FIG. 2.1: costo in $cent/KWh dell'energia per uso industriale.

Fonte: Key world energy statistics, IEA (2007).

Per altro l'inefficienza energetica sta diventando un limite davvero duro per i gestori della rete elettrica, i quali rischiano di non poter più sostenere determinati ritmi, infatti la domanda di fornitura d'energia sta crescendo di circa il 10% all'anno per quelle aziende che vantano data center dislocati in aree ad alta densità popolare. Si prospetta quindi un evidente grosso problema di scalabilità di questi sistemi informatici il che porterà le aziende ad implementare nuovi sistemi in nuove aree, con ulteriori impatti ambientali e di costo. Un cane che si morde la coda.

2.2 Green IT

Dai dati precedenti si giustifica e si avvalorano le attività di ricerca che, negli ultimi anni hanno portato ad un esplosione delle performance dei dispositivi (in senso generale) ed ad un approccio di progettazione degli stessi che non limitasse l'efficienza ma bensì portasse benefici dal punto di vista del minor consumo di potenza.

Queste attività rientrano nell'area del Green ICT, in cui è possibile individuare una suddivisione in ulteriori tre macroaree principali di studio:

- La gestione ecocompatibile del ciclo di vita dell'IT;
- L'efficienza energetica dell'IT;
- L'utilizzo dell'IT come strumento per una governance "green".

L'impatto negativo sull'inquinamento globale dovuto alle tecnologie IT inizia, ovviamente, dalla definizione e produzione delle stesse fino alla loro dismissione. L'intero ciclo di vita di un computer, dalla produzione allo smaltimento, apporta diversi problemi ambientali. La produzione consuma elettricità, prodotti chimici ed altre risorse, divenendo così causa di inquinamento che non è quindi unicamente riferibile al consumo di energia ma deriva, in parte, dall'utilizzo e dall'errato smaltimento delle sostanze tossiche utilizzate nei processi. Un fenomeno purtroppo molto diffuso e definito con l'acronimo: WEEE, Waste of Electric and Electronic Equipment.
Una gestione quindi attenta e rispettosa dell'ambiente, attuata attraverso politiche rigorose che, ad esempio, diano linee guida per ridurre l'utilizzo delle componenti inquinanti delle attrezzature IT, per l'ottimizzazione del packaging, per l'eco-etichettatura delle varie parti e per il ricondizionamento e recupero dei diversi componenti, è l'ambito di studio della prima macroarea.

Per quanto riguarda l'efficienza energetica, grazie agli sviluppi in ambito GREEN IT in cui, alla ricerca di miglioramenti, si è agito sia sulla progettazione che sulla gestione delle strutture e dei data center o ancora modificando la cultura aziendale e le pratiche di utilizzo dell'hardware a disposizione, si può apprezzare come negli ultimi anni il rapporto CONSUMO ENERGETICO/PRESTAZIONI sia diminuito dato che la crescita delle performance è stata più poderosa rispetto a quella dei consumi energetici necessari per averle.

L'ultima macroarea ha come oggetto la gestione e l'utilizzo delle tecnologie informatiche come strumento per la rilevazione e misurazione di determinati parametri utili ad evidenziare i consumi dei propri sistemi: energia elettrica, calore, o quantità di rifiuti tossici prodotti. Purtroppo l'attenzione a questo tema, da parte di chi in un'azienda si occupa di gestire le risorse IT, è spesso nulla o comunque poco incentivata.
Diversi studi hanno rilevato come grosse percentuali (vicine al 90% [3]) dei dipartimenti IT in UK non sia a conoscenza della quantità di CO_2 prodotto ed emesso nell'ambiente, e l'80% delle organizzazioni non è a conoscenza del dispendio energetico delle proprie attività. In

Italia, un'analoga ricerca prodotta dal DEI del Politecnico di Milano [4] ha individuato che ben l'89% degli addetti alla gestione IT, delle aziende campione, non è a conoscenza dei consumi energetici dei proprio sistemi. Diviene quindi impossibile cercare di migliorare qualcosa di cui non si è a conoscenza.

2.3 Possibili aree di ricerca

Analizzando il consumo di energia in un data center medio, come si può apprezzare dalla figura 1.2 è chiaro come il problema dell'efficienza 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.

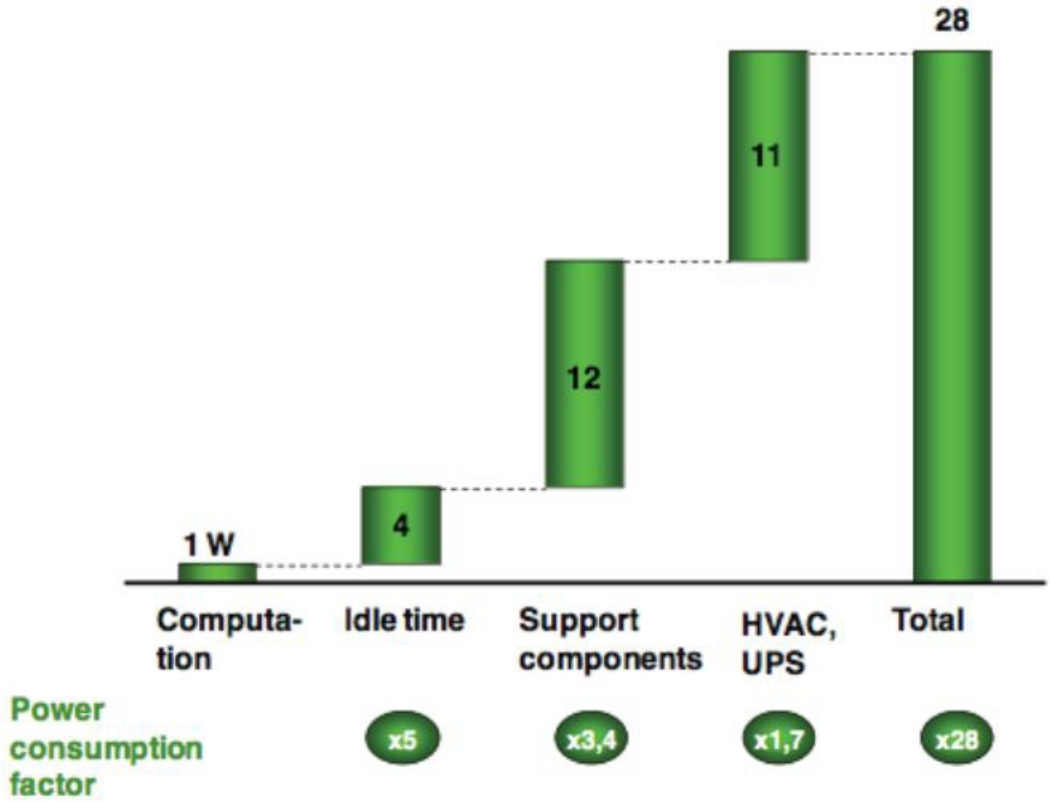


FIG. 2.2: Ripartizione percentual del consumo d'energia in una data center.

Fonte: IBM (2007).

Uno sforzo maggiore deve essere inoltre posto per ottimizzare l'efficienza energetica dei livelli più bassi del sistema, in quanto inefficienze negli stadi inferiori saranno propagate ed amplificate a quelli superiori. Basti pensare, per esempio, che ad un maggiore numero di computazioni effettuate dal processore ci sarà una conseguente maggiore richiesta di memoria ed un ulteriore bisogno di raffreddare il sistema.
In particolare, da una ricerca fatta dal MIT sul consumo richiesto per la COMMUTAZIONE di un bit in un sistema (quindi un livello davvero bassissimo di analisi), si è potuto arrivare a stimare il fattore medio di amplificazione del consumo, nella propagazione a livelli più alti, pari a 30.
Attualmente l'efficienza energetica di un sistema IT viene stimata al 50% dell'efficienza massima teorica ottenibile. In tabella 1.1 sono presentate alcune stime sull'efficienza dei vari livelli dei sistemi IT più comunemente utilizzati [5, 1].

LIVELLO	EFFICIENZA ENERGETICA STIMATA
Infrastruttura	50,00%
Sistema	40,00%
Server	60,00%
Microprocessore	0,00%
Software	20,00%
Rete	10,00%
Database	60,00%
Pratiche di utilizzo dell'IT	30,00%

TAB 2.1: Efficienza energetica stimata rispetto all'efficienza massima teorica degli attuali sistemi IT

Quali sono quindi i fattori su cui si è agito, si sta agendo e si potrà agire per rendere efficienti questi sistemi? Sono molti i fattori che contribuiscono all'inefficienza dei vari livelli IT e molte sono le leve sulle quali si può agire per ottenere dei miglioramenti.
Per esempio, a livello d'infrastruttura, sarebbe possibile agire sui sistemi di condizionamento per far si che regolino DINAMICAMENTE il raffreddamento dei data center dove e quando necessario in base ai diversi livelli di workload, ottenendo un grande risparmio dato che i sistemi di raffreddamento, insieme ai gruppi di continuità, rappresentano le più grosse cause di inefficienza. Un altro approccio potrebbe essere la suddivisione delle memorie cache in segmenti con alimentazione singola o l'utilizzo di nuovi device di storage, quali quelli a

STATO SOLIDO, in sostituzione degli HARD DISK classici.
Inoltre, da vari studi, si è individuato nei 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 ed oltre al loro limite architetturale, essi non sono neppure utilizzati al massimo delle potenzialità e, di conseguenza, diventano inefficienti. E 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 50% [18].
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 indirettamente collegati all'efficienza energetica.

2.4 GREEN SOFTWARE

La più grossa fetta di ricerche per un "Green IT" si è svolta sul lato hardware dei sistemi informatici ottenendo numerosi e lodevoli risultati. Questo grazie sicuramente agli enormi sviluppi che le ricerche in elettronica e microelettronica hanno portato. Di contro c'è sempre stata un'indifferenza di attenzione sull'impatto che il software, che guida i dispositivi, possa avere sui consumi. D'altro canto, è la stessa crescente disponibilità di hardware efficiente e a basso costo che induce i programmatori a non occuparsi dell'efficienza energetica del codice.
Il Green Software però, la disciplina che studia le modalità secondo cui il software influisce sui consumi energetici dell'IT e come ottimizzarle, è un argomento preso sempre più in considerazione negli sviluppi di progetti informatici ed il Politecnico sta impegnando una considerevole quantità di forze e risorse su questo ambito di ricerca. In particolare son state percorse diverse strade che hanno portato alla definizione di linee guida per lo sviluppo di software energeticamente efficiente, metriche e relativi tool. Gli studi hanno implicato ed implicano la valutazione di diverse scelte progettuali, sia architetturali che tecnologiche, per evidenziarne le qualità positive o negative in termini di consumi del software.
Da studi effettuati si è individuato come, a parità di architettura hardware e di task richiesti, il sistema operativo giochi un ruolo in termini di efficienza energetica del software, ruolo per altro non sempre prevedibile. Inoltre, sempre dallo stesso studio, si evince che la relazione tra tempo di esecuzione ed energia consumata non è sempre lineare [6]. Le osservazioni

presentate da questo lavoro dimostrano che il consumo energetico indotto dal software è significativo e che ci sono possibili ampi margini di risparmio potenzialmente derivabile dall'ottimizzazione delle applicazioni.
Sono stati individuati due tematiche di sviluppo e ricerca in questo ambito:

- LA DEFINIZIONE DI METRICHE PER LA VALUTAZIONE DELL'EFFICIENZA ENERGETICA DEL SOFTWARE;

- L'INDIVIDUAZIONE DI MODALITÀ PER MIGLIORARE L'EFFICIENZA ENERGETICA;

Innanzi tutto, occorre misurare l'efficienza energetica del software tramite opportune metriche. Si noti che il concetto di efficienza è diverso da quello di consumo, in quanto rapporta l'energia consumata alla quantità di lavoro che ha permesso di svolgere. Un insieme di metriche per l'efficienza energetica costituisce lo strumento essenziale per poter confrontare tra loro sistemi diversi e quindi fare benchmarking.
Esistono già diverse metriche consolidate per misurare l'efficienza energetica dell'IT a vari livelli. Il Power Usage Effectiveness (PUE), calcolato come la potenza entrante in un data center divisa per la potenza effettivamente assorbita dal carico IT, è comunemente utilizzato per valutare l'efficienza dei livelli infrastrutturali, includendo quindi sistemi di condizionamento e UPS (gruppi di continuità). Per valutare il bilanciamento dei carichi e il livello di virtualizzazione si possono adottare metriche come la percentuale media di utilizzo dei processori, sia fisici che virtuali. Vi sono poi diverse metriche utilizzabili a livello di server e di processore per valutare l'effi cienza energetica dell'hardware, per esempio W/tpm (Watt per transazioni al minuto), FLOP/Wh (operazioni floating point per Wh) oppure MIPS/W (Milioni di istruzioni al secondo per Watt) per gli ambienti mainframe. Tuttavia non vi sono ancora metriche consolidate e sufficientemente generali per misurare l'efficienza energetica del software.
È importante analizzare l'efficienza di un sistema informativo a tutti i suoi livelli, in quanto gli attori coinvolti possono essere molto diversi. Le difficoltà risiedono nella quantificazione del "lavoro" svolto da un'applicazione con una quantità unitaria di energia, e questo sposta il problema alla definizione di concetti di transazione o di carichi di lavoro standard, il più possibile generali, da far eseguire ad un'applicazione per effettuare le misure. Per alcune applicazioni la definizione di transazione è abbastanza intuitiva (per esempio, per i DBMS), mentre per altre, come i sistemi ERP, occorre considerare differenti tipologie di transazione e

quindi diverse metriche di efficienza. Le diverse transazioni corrispondono a differenti moduli e porzioni di codice dell'applicazione, quindi è sensato adottare metriche diverse, anche se è auspicabile giungere ad una metrica unica per ciascuna applicazione, derivata da una media degli indicatori rispetto alla frequenza di utilizzo. Nella definizione di queste metriche, inoltre, occorre ovviamente anche tenere presente i dati forniti come input e le dimensioni dei database sottostanti.

È evidente quindi come la definizione e il calcolo di queste metriche sia più complicata che per gli altri livelli infrastrutturali.

Fondamentali allora sarebbero i risultati che riguardano la definizione di metriche che permetterebbero di individuare effettivamente quale sia, ad un certo livello accettabile di dettaglio, l'impatto del software nei consumi di energia e di poter quindi indirizzare le fasi di progetto e sviluppo dei sistemi.

Ponendosi invece l'obiettivo di migliorare l'efficienza energetica, gli approcci perseguibili possono essere raggruppati in due grandi categorie:

- l'ottimizzazione del codice, in quanto un codice scritto meglio può avere un effetto sulle performance energetiche dell'applicazione. Quest'attività può tradursi sia in metodologie, linee guida e tool per lo sviluppo, che in linee guida a livello organizzativo e relative alle competenze e al percorso di formazione degli sviluppatori. Purtroppo non è un'attività sempre fattibile o non sempre conveniente dato che, ad esempio, non sarebbe pensabile realizzare l'analisi e l'ottimizzazione di tutto il codice del sistema informativo di una banca e, in ogni caso, lo sforzo necessario per questa operazione difficilmente sarebbe ripagato dai risparmi ottenuti.

- l'ottimizzazione a livello di sistema operativo, per esempio si potrebbe effettuare una scelta oculata dello stack, considerando l'impatto che l'interazione fra sistema operativo e applicazione ha sul consumo complessivo.

In questo contesto il Politecnico di Milano ha svolto diverse ricerche in particolare riguardanti rispettivamente la gestione green della memoria e del garbage collector e l'utilizzo della tabulazione dinamica per aumentare l'efficienza di determinate applicazioni. A tal proposito, interessante e produttiva è stata l'individuazione, per le applicazioni **computing intensitive**, della possibilità di diminuire l'energia consumata cercando di utilizzare meno componenti dei dispositivi adibiti al calcolo, quali i processori, a favore di altri componenti quali le memorie ed i dispositivi detti di "storage". La tecnica in questione è definita come MEMOIZZAZIONE e sarà oggetto di spiegazione del successivo capitolo.

3. Basi del progetto

Come evidenziato nel precedente capitolo, esiste una ormai sempre più grossa branca di ricerca interna al “green IT” che si occupa di efficienza energetica del software. Lo sviluppo, da parte del Politecnico di Milano, di un framework basato sul concetto di Memoizzazione ne è un esempio. In questo capitolo verranno proposti premesse e risultati del lavoro svolto dal Politecnico, che saranno presi come base introduttiva per la definizione dei principi e aspettative su cui si basa il lavoro da me svolto in quest'ambito.
Il progetto ha riguardato l'analisi e la valutazione dell'impatto che le variazioni in gerarchie, dimensionamento e latenza delle struttura di memoria, in un architettura ARM, possano avere sulla Memoizzazione. Il tutto utilizzando il simulatore GEM5, introdotto e spiegato successivamente.

3.1 Memoizzazione e Framework

Il termine memoizzazione è derivato dal termine “memo function" coniato da Donald Michie (1968) [7] riferendosi al processo con cui una funzione può ricordarsi del risultato di una computazione precedente. Nello specifico, la memoizzazione è una tecnica di programmazione dinamica 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. 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.
L'idea base, di questa tecnica, è quella di memorizzare una tabella di coppie <input,risultato>

e, per ogni invocazione della funzione, di controllare nella tabella se esiste gia la entry corrispondente. Prendendo spunto dal lavoro svolto dall'Ing. Bessi Marco, per spiegare il concetto in modo piu approfondito si valuti l'esempio della funzione, in pseudo-codice, "**int fibonacci(int n)**" dove "n" è la posizione del numero che vogliamo restituito all'interno della serie di Fibonacci. Un possibile algoritmo potrebbe essere il seguente:

```
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 e un risultato 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 e 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 l'esecuzione 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 salvare 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.

3.1.1 Funzione pura

La definizione di funzione pura può essere derivata dal concetto di funzione matematica, ossia un mapping uno a uno tra input e output [8]. In programmazione, un metodo è definicome come “puro” se soddisfa le seguenti 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.

In riferimento alle proprietà appena citate è opportuno andare a definire meglio cosa si intenda per “metodo libero da side-effect” e per “metodo deterministico”.

Un metodo è libero da side-effect se i soli oggetti che esso modifica sono creati come parte dell'esecuzione del metodo stesso, il che permette al metodo di creare e modificare nuovi oggetti ma non di modificare tutti i valori osservabili dall'esterno del metodo. In aggiunta, possiamo richiedere che un metodo puro non causi nessun side-effect fuori dall'ambiente del sistema, ad esempio non potràa 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 generera lo stesso risultato. Il requisito di determinismo per un metodo è analogo a quello delle funzioni matematiche. 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 sarebbe necessario 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 si trova 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 risposta definitiva alle domande sopra proposte perché per ogni problema che si deve affrontare sarà ottimale una scelta piuttosto che l'altra.

Detto ciò, una definizione di determinismo di un metodo, in programmazione, potrebbe essere: ”Il determinismo di una metodo e una proprietà parametrica: data la definizione di cosa si intende per equivalenza tra parametri, un metodo e deterministico se tutte le chiamate con argomenti equivalenti ritornano risultati indistinguibili tra di loro”.
Secondo questi principi e ragionamenti, è stato sviluppata quindi una metodologia ed un framework che si occupa di individuare le funzioni pure in un eseguibile e di applicare il tecnica di memoizzazione per poter trarre vantaggio dal punto di vista di efficienza energetica. Tenendo conto che la maggior parte delle applicazioni che utilizzano funzioni computing intensive sono scritte in linguaggi pre-compilati e già utilizzate e funzionanti, si è reso necessario individuare situazioni in cui è possibile intervenire senza modificare il codice sorgente dei suddetti eseguibili, applicando la tecnica di memoizzazione senza modificare le funzionalità del software. Nel panorama odierno, la maggior parte delle funzioni a cui lo studio ha fatto riferimento, sono quelle bancarie e scritte nel linguaggio procedurale COBOL. Una buona parte però, almeno negli sviluppi recenti, fa uso del linguaggio java ed è quindi JVM-based. Si è quindi scelto di andare ad applicare la tecnica di memoizzazione modificando il bytecode mantenendo intatte le funzionalità pregresse. Nel paragrafo successivo verrà introdotta una definizione più raffinata di funzioni pure, le scelte per l'attuazione della memoizzazione, il framework definito ed un sunto sui benchmark che verranno poi utilizzati per il lavoro di ricerca scopo di questa tesi.

3.1.2 Architettura del framework

Per lo scopo preposto, la definizione introdotta prima di “metodo puro” non è soddisfacente, dato che i nuovi oggetti creati devono essere differenti ad ogni invocazione. Una nuova definizione, più consona allo scopo, è quella che concerne le proprietà seguenti:

- La sua firma non include parametri di tipo oggetto, ad eccezione di: array (se il tipo base dello stesso è “primitivo”, parametro *THIS;*
- Il tipo di ritorno è “primitivo;
- Tutti i metodi invocati al suo interno devono a loro volta essere puri;
- Per tutte le istruzioni contenute nel suo corpo, deve valere:
  - l'istruzione non legge ne modifica variabili statiche;
  - l'istruzione non legge ne modifica variabili che non sono dichiarate all'interno delle funzione stessa;
  - l'istruzione non modifica valori di array.

È opportuno notare che, a differenza di molte definizioni di funzioni pure, in questa appena citata non ci sono prevenzioni sulla generazione od il “catch” di eccezioni, dato che queste si modificano il normale flusso di esecuzione del programma ma non influenzano risultati esterni o prevengono la generazione di valori di ritorno. Inoltre, è possibile allocare oggetti dentro la funzione se, e solo se, questi oggetti non sono “vivi” alla chiusura della stessa.

L'architettura (mostrata in fig. 3.1) e composta da quattro parti principali: una parte di analisi statica del bytecode dell'applicazione per la scoperta delle funzioni pure in esso contenute e la loro successiva modica, una parte che si occupa dell'esecuzione del meta-modello “Decision maker", una parte di analisi delle informazioni per reportistica ed infine una parte che si occupa del salvataggio delle tabelle dati in memoria su file XML.

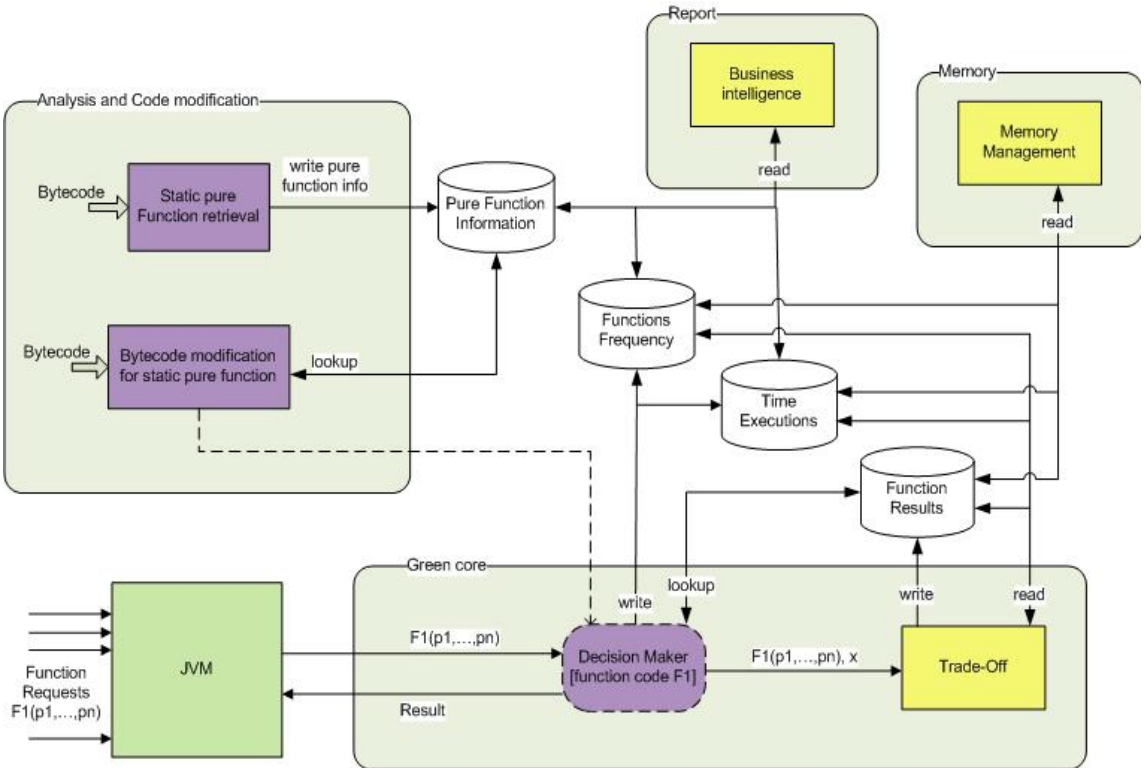


FIG.3.1: Architettura di riferimento del framework di applicazione della memoizzazione a software JVM-based [10]

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 e stato analizzato ed e stato deciso quali funzioni pure è utile tabulare, può essere eseguito il modulo “Bytecode modification" che si occuperà della modifica delle funzioni pure inserendo dentro di esse il codice necessario per l'esecuzione del meta-modulo “Decision maker".

Durante l'esecuzione dell'applicazione la JVM analizza ed esegue normalmente il bytecode. Nel caso di esecuzione di una funzione pura, la JVM non caricherà il codice originario della classe in cui si trova la funzione richiesta ma caricherà il codice che e stato modificato nella precedente parte dal modulo “Bytecode modification" (eseguito una-tantum per applicazione). Il metodo, prima dell'esecuzione normale del codice della funzione, si occuperà di effettuare una lookup sulla tabella relativa alla funzione in esecuzione con i parametri effettivi per cui e stata richiamata: se esiste nella tabella il risultato, relativo alla chiamata alla funzione per quei determinati input, la funzione ritorna direttamente questo valore, altrimenti continua la sua esecuzione normalmente e prima dell'istruzione return verrà chiamato il modulo “Trade off". Questo modulo si occuperà 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, che discuterò brevemente perchè meno relativa al mio lavoro di ricerca, si occuperà dell'analisi dei dati relativi all'esecuzioni (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 modiche per aumentarne ancora le potenzialità. La parte di gestione della memoria, infine, si occupa del salvataggio delle tabelle presenti in memoria su file XML.

Per un dettaglio approfondito dei moduli: “Pure function retrieval", “Bytecode modication", “Trade off”, “Business Intelligence” e il meta-modulo “Decision maker” si rimanda all'elaborato di tesi specialistica dell'Ing. Bessi Marco, ove vengono anche spiegate in dettaglio le interazioni software fra i vari moduli per una migliore comprensione del funzionamento dell'approccio scelto. Quel che interessa fare presente in questa sede è che i vari moduli, ed in particolare il “Bytecode modification”, si occupano di andare a ridefinire il codice bytecode delle funzioni individuate come pure dal “Pure function retrieval”, controllando sul database le informazioni che identificano queste funzioni pure e, per ognuna di queste, inserendo le istruzioni che permettano l'esecuzione del meta-modulo “Decision maker”. 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 che permettono l'esecuzione del meta-modulo “Decision maker". L'inserimento del meta-modulo, così come per l'analisi dell'applicazione avviene a livello di istruzioni bytecode e non richiederà quindi una compilazione ulteriore del codice prodotto. In realtà il modulo di “Decision maker" è un meta-modulo, ossia un modulo astratto che è l'integrazione dei moduli di “Trade off" e di “Memory management". Inoltre esso rappresenta le istruzioni bytecode che permettono il funzionamento e l'integrazione generale di tutti i moduli dell'approccio proposto. Il lavoro di “inserimento” di questo meta-modulo viene quindi effettivamente svolto dal “bytecode modification” che, nella pratica, inserisce due blocchi di istruzioni: un blocco per la Lookup sulla tabella relativa alla funzione in esame ed un blocco per la chiamata al modulo di “Trade off".

Nello specifico, avendo una funzione pura **returnType className.methodName(params)** implementata come in fig. 3.2 (in pseudo-codice), la relativa funzione modificata dopo l'inserimento del meta-modulo “Decision maker" risulta come lo pseudo-codice in fig. 3.3.

```
public class className {

    ReturnType methodName(Parameters p){
        ReturnType result = null;
        //ORIGINAL CODE {
            //effettua il calcolo di result
        //ORIGINAL CODE }
        return result;
    }

}
```

FIG.3.2: className.methodName PRIMA dell'inserimento del modulo “Bytecode modification". [10]

```
public class className {

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

        //BLOCCO di LOOKUP
        String methodId = "className;methodName;methodSignature";
        Object[] params = {p};
        double alpha = 0.5;
        TOParameters toparam = new TOParameters(params);
        Return ret = MemoryManagement.getInstance().lookupResult(methodId, toparam);
        if (ret != null) {
            MemoryManagement.getInstance().setupFrequency(methodId, toparam);
            return (ReturnType) ret.getpValue();
        }
        long compTimeStart = System.nanoTime();

        //ORIGINAL CODE {
            //effettua il calcolo di result
        //ORIGINAL CODE }

        //BLOCCO di TRADE OFF
        long finalTime = System.nanoTime();
        long priorityTemp = (finalTime - compTimeStart);
        ret = new Return(result);
        TOFunction tof = new TOFunction(signature, toparam, ret, alpha);
        tof.setupTime(priorityTemp);
        TradeOff.tradeOff(tof);
        MemoryManagement.getInstance().setupFrequency(signature, toparam);

        return result;
    }

}
```

FIG.3.2: className.methodName DOPO l'inserimento del modulo “Bytecode modification". [10]

Evidenziati in colori diversi, si denotano i blocchi di codice relativi a:

- verde - blocco di Lookup in cui viene effettuata la ricerca nella tabella della funzione per controllare se e già stato tabulato il risultato per i parametri di ingresso della funzione;

- blu - blocco di codice originario della funzione;

- giallo - blocco di Trade off che si occupa della gestione della tabella tramite le API fornite dal modulo di “Trade off”.

Il flusso di esecuzione della funzione, dopo l'inserimento del meta-modulo “Decision maker" risulta, come riportato in fig. 3.4. I blocchi del diagramma di flusso riprendono i colori usati in precedenza per evidenziare nuovamente come sono distribuite le operazioni all'interno della funzione.

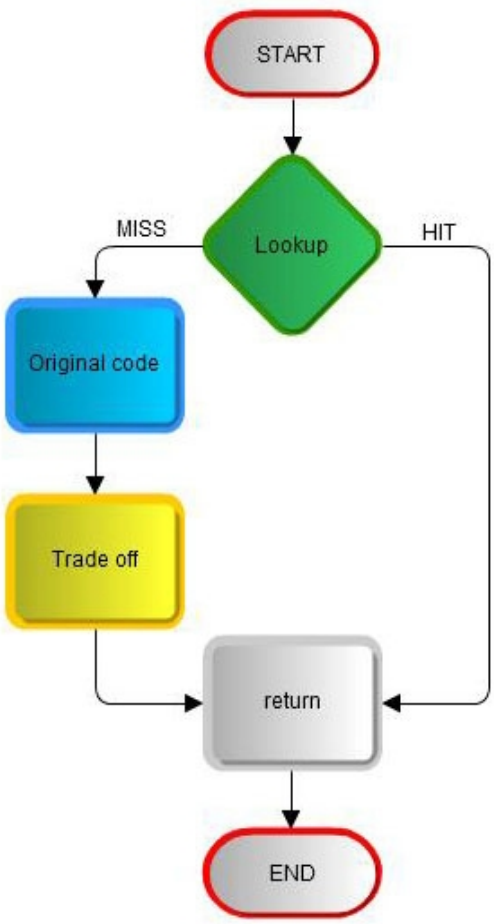


FIG.3.4: flusso di esecuzione del programma DOPO l'inserimento del meta-modulo “Decisione Maker".

Il blocco di Lookup si occupa di effettuare la ricerca sulla tabella della funzione per controllare se è presente il risultato pre-calcolato per gli input dell'attuale esecuzione della funzione. In particolare (in riferimento alla figura 3.3, blocco verde) il blocco effettua le seguenti operazioni primarie:

1. Crea la stringa “methodId” usata successivamente per effettuare la Lookup 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 Lookup;

3. Crea una variabile “Alpha” necessaria per il funzionamento successivo del blocco di Trade off;

4. Effettua una Lookup 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);

5. Se la Lookup fa un HIT:

    a) Aggiorna la frequenza di utilizzo (aumenta un contatore interno di 1) della funzione con id “methodId” chiamata con i parametri in esame;

    b) Il metodo ritorna il corretto valore prelevato dalla tabella per la funzione e i parametri con la quale e stata chiamata ottenuto dalla Lookup precedente;

6. Inizializza una variabile per il calcolo del tempo di esecuzione del codice originale della funzione.

Il codice viene modificato al suo interno solo per l'inserimento, prima di ogni return, di una chiamata al modulo di “Trade off". Il codice inserito effettua le seguenti operazioni (in riferimento alla figura 3.3, blocco giallo):

1. Calcola il tempo di esecuzione della funzione originale;

2. Crea l'oggetto Return con il risultato dato dal calcolo effettuato dalla funzione originale;

3. Crea l'oggetto TOFunction passandogli signature, parametri in input e risultato dell'esecuzione del codice originale della funzione;

4. Chiama il metodo TradeOff passandogli l'oggetto TOFunction appena creato;

5. Incrementa il contatore della frequenza delle chiamate per la funzione corrente chiamata con i parametri in esame.

Il ruolo principale di questo modulo e di gestire i dati interni al Memory Management in accordo a regole ben definite. Il modulo di “Trade off" è responsabile di fornire decisioni intelligenti riguardo la possibilità di registrazione d'informazioni sulle funzioni pure (parametri e risultati).
Inoltre il modulo mantiene un riferimento con priorità ad ogni TOFunction salvata in memoria permettendo varie scelte riguardo la gestione della memoria come: il salvataggio del risultato per la specifica funzione, la cancellazione di alcuni risultati per far spazio ad ulteriori salvataggi, ecc...
Il modulo “Trade off" lavora dopo l'esecuzione del codice originale, ossia quando la Lookup sulla tabella relativa al metodo eseguito restituisce una miss per gli argomenti passati alla funzione. Nello specifico, si occupa di decidere, in accordo con le proprie regole interne (per l'ottimizzazione della memoria libera rimanente e l'utilizzo ottimale della memoria occupata), se il nuovo risultato che è stato calcolato deve essere salvato nella tabella in memoria.

Dopo aver quindi introdotto la memoizzazione ed il framework sviluppato al Politecnico di Milano, il prossimo paragrafo darà una visione dei benchmark scelti per la validazione del tool appena spiegato. Questi benchmark sono gli stessi utilizzati nel mio lavoro di ricerca ed i risultati della validazione fatta per il framework di cui sopra saranno ripresi nel capitolo 5 come elemento di confronto per le analisi da me svolte.

3.1.3 Benchmark

Nella validazione del progetto, sono stati scelti come benchmark delle funzioni utilizzate in ambito finanziario. Queste sono state riprese, così come trovate in alcuni progetti open source sul web, senza alcuna modica. Le funzioni sono computational-intensive e quindi richiedono un grande uso del processore a causa di cicli di elevamenti a potenza ed altre operazioni matematiche. In particolare, fra le varie, sono stati individuati:

- **Xirr**, funzione che calcola lo “annualized internal rate” di ritorno di un flusso di cassa, ad un punto arbitrario nel tempo. Come evidenziato dalla fig. 3.5 il metodo XIRR prende in input due array, rispettivamente *double* e *int*, che rappresentano il flusso di cassa in termini di valore e data, ed un'ipotesi di soluzione usata per inizializzare l'algoritmo (un *double*). Notare che, dato che l'array è composto di tipi primitivi, non causa “impurità” con riferimento alla definizione di “funzione pura” data precedentemente. Lo stesso vale per gli altri due parametri. Inoltre le linee 2-4 dichiarano variabili di tipo primitivo e quindi non cambiano la purezza del metodo. L'analizzatore del bytecode, dopo aver verificato queste premesse, andrebbe a valutare i 2 cicli *FOR* analizzando gli accessi ai dati, sia in lettura che scrittura, verificando che non ci siano cambi di purezza del metodo che è infatti quel che succede, dato che il metodo scrive soltanto su variabili locali di tipi primitivi e che gli accessi agli array sono soltanto in lettura. Un'unica assunzione, comunque vera, è che la funzione “signAdrd()” sia pura.

```
public static double Xirr(double[] value, int
    [] days, double guess){
  int maxConvergenza = 50;
  double adrd = 0.05, xirr = guess, sum;
  char sign = '+';
  for (int i = 1; i <= maxConvergenza; i++) {
    sum = 0;
    for (int j=0; j<value.length; j++) {
      sum += value[j] / Math.pow(1 + xirr,
          days[j] / 365.0);
    }
    adrd = signAdrd();
    xirr = xirr + adrd;
    if(Math.abs(sum) < 0.0000001) { break; }
  }
  return xirr;
}
```

FIG.3.5: codice sorgente del metodo XIRR, uno dei benchmark.

- I**mpliedVolatility**, funzione che calcola la volatilità cui è soggetta un'opzione; è calcolata sulla base del modello di Black e 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 d'interesse[11] ;

- **BlackScholes,** 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 [12];

- **Serialization (Fourier)**, che calcola i primi N coefficienti di Fourier per la funzione:

$$f(x)=(x+1)^x$$

Senza scendere troppo nel dettaglio, nel lavoro svolto dal Politecnico, è stato individuato un modello delle performance [13] basato sul tempo di esecuzione delle funzioni. In particolare, la stima di effettività della memoizzazione è basata su: t**empo di esecuzione della funzione originale**, la **disponibilità di memoria**, la **precisione richiesta** e la **distribuzione dei parametri in ingresso**.
Per questo lavoro di tesi sono state mantenute coerenti, rispetto alle valutazioni fatte, le scelte in termini di quantità di input dati ai benchmark e modalità di definizione casuale degli stessi. Quest'ultimi sono stati creati sfruttando il tool di generazione casuale già prodotto dal politecnico che produce parametri input casuali con distribuzione Gaussiana e in base ad una data Media, Deviazione Standard e Precisione, come determinate ed esposte in [13]. In particolare, il numero di esecuzioni scelte per i singoli benchmark, e quindi il numero di input generati e dati in pasto, è pari a: **100.000** per **Implied Volatility** e **Fourier**, di **1.000.000** per **XIRR** e di **40.000.000** per **Black Scholes**.

Infine, per poter anche fare un confronto fra altri dati recuperati in fase di validazione del framework sopra discusso ed i dati invece raccolti in questo elaborato, si è scelto di fissare ad 1GB la memoria RAM disponibile per la memoizzazione.
Come anticipato precedentemente, il progetto si pone l'obiettivo di valutare su un'architettura ARM se i risultati di efficienza energetica, ottenuti tramite la memoizzazione, cambino e come lo facciano, nel prossimo paragrafo quindi si presenterà l'architettura ARM in breve.

3.2 Perchè ARM

Sebbene architetture come Intel o PowerPC abbiano nei decenni raggiunto una maturità, sofisticatezza e venerazione che sarebbe difficile da raggiungere per ARM che ha solo 22 anni di vita, quasi tutte le più grandi aziende produttrici di tecnologie informatica, vedi AMD, Apple, Dell, Dialog, Freescale, Fujitsu, HP, LSI, Microsoft, Motorola, MStar, Nvidia, Qualcomm, Samsung, STMicroelectronics e Texas Instruments, utilizzano core ARM per tutto; dagli smartphone ai tablet; alle basestation ai server.
È vero che lo “ecosystem” dei chip di supporto, i sottosistemi e il software della intel non hanno eguali nel settore e che essa affronta molti altri problemi del mondo reale piuttosto che solo la dimensione del “die” a basso consumo e di piccole dimensioni, il che fa di ARM un gioco da ragazzi per molti nuovi progetti. Vero è anche che, sebbene ARM abbia guadagnato grosse fette del mercato per le applicazioni embedded a basso consumo, deve ancora affrontare un lungo e tortuoso percorso per sfidare, da pari livello, i pluriennali “re dei microprocessori”, dato che la compatibilità è ancora un punto debole per ARM e che ci sono molti mercati verticali che rimangono inaccessibili, per i suoi processori, a causa delle loro standardizzazione.

Comunque la lotta “Intel versus ARM” rimane un rompicapo. Per esempio, lo spazio del cloud computing, in rapida crescita, utilizza la virtualizzazione per offrire agli utenti di dispositivi mobili l'accesso ad applicazioni eseguite sui server. In questo ambito Intel offre una soluzione “top-to-bottom”: Vtx, la quale assicura collegamenti fra dispositivi mobili basati su core x86 con cloud computer basati su processori Xeon. Dall'altra parte ARM sta ancora cercando di raggiungere estensioni di virtualizzazione che possono offrire integrazioni, fra dispositivi mobili e computer nel cloud, simili alla precedente.
Come afferma Sergis MUSHELL (analista ricercatore capo alla Gartner Inc.), Arm stà diventando sempre più matura, passando dai primi processori “light-weight” per dispositivi mobili a processori come l'A15 che possono competere sicuramente con gli ATOM (della intel) ma non sono ancora della stessa classe degli XEON (sempre intel).
Un altro risultato notevole è la penetrazione dei processori ARM nel mercato delle basestation wireless, andando ad invadere un mercato fortemente legato ad architetture MIPS. Infatti, sempre secondo Mushell, ARM sta ormai minacciando “pericolosamente” gli ecosystem MIPS, immettendosi con forza nel campo della basestation e del networking in generale, guadagnando grosse licenze come quella della Texas Instruments.
Comunque, sebbene sia vero che gli ecosystem ARM per i microprocessori embedded e per i microcontrollori stiano crescendo con frequenza rovente, ci sono delle applicazioni in cui i produttori hanno trovato più utile mantenere un'architettura proprietaria.

Nonostante i grandi sforzi che ancora questa giovane architettura debba evidentemente fare, non si può non prenderla seriamente in considerazione per eventuali sviluppi relativi a questo campo. Per altro la differenziazione chiave fra i chip ARM e quelli INTEL è che i primi richiedono livelli di potenza più bassi dei secondi e, inoltre, sono più economici, almeno nel mercato odierno. Questo è anche uno dei motivi per cui sono stati adotttati su quasi tutti i tipi di smartphone.
Si è scelto di impostare quindi il progetto di ricerca su una valutazione che abbia ARM come scelta di base architetturale, data la frequenza di adozione crescente da parte delle case produttrici. Del resto anche la Microsoft ha definito un nuovo sistema operativo, chiamato Windows RT, prodotto proprio per ARM.[14]

3.3 ARM overview

FIG.3.6: ARM logo

L'architettura ARM descrive una famiglia di processori per computer RISC-based (infatti originariamente l'acronimo ARM stava per “Advanced RISC Machine”, poi trasformato in “Acorn RISC Machine”[15]) progettatti e messi sul mercato dall'azienda Britannica “ARM HOLDINGS”. Il primo sviluppo risale al 1980 e, ad oggi, è l'istruction set architecture a 32 bit più usato in termini di quantità prodotta. Solo nel 2011 i produttori di chip basati su questa architettura riportano un ammontare di consegne di 7.9 miliardi che, nello specifico, rappresentano il 95% degli smartphone, 90% degli hard disk, 40% di televisori digitali e set-top boxes (decoder), 15% di microcontrollori e il 20% laptop[16].
L'utilizzo di un approccio RISC-based permette ai processori ARM di richiedere un valore significativamente inferiore di transistor rispetto a quelli richiesti comunemente dai computer odierni. I benefici di questo approccio sono quindi costi più bassi, minor produzione di calore e minor consumo di potenza; tutti tratti desiderabili per dispositivi leggeri, portatili e con funzionamento a batteria come gli smartphone ed i tablet pc.
Inoltre, la ridotta complessità ed il design più semplice permette alle compagnie di produrre

una sistema a basso consumo energetico su un singolo chip per sistemi embedded che incorpori memoria, interfacce, ecc....
Il primo esempio è stato il tablet della Apple: “NEWTON”, ma lo stesso approccio è stato poi utilizzato nei chip Apple A4 e A5 dell'iPad. Nel 2012 la Microsoft ha prodotto il suo nuovo tablet: “SURFACE” con tecnologia ARM e la AMD ha annunciato che, dal 2014, inizierà a produrre chip per server basati sull'architettura dei processori ARM a 64 bit[17]

3.3.1 Architecture overview

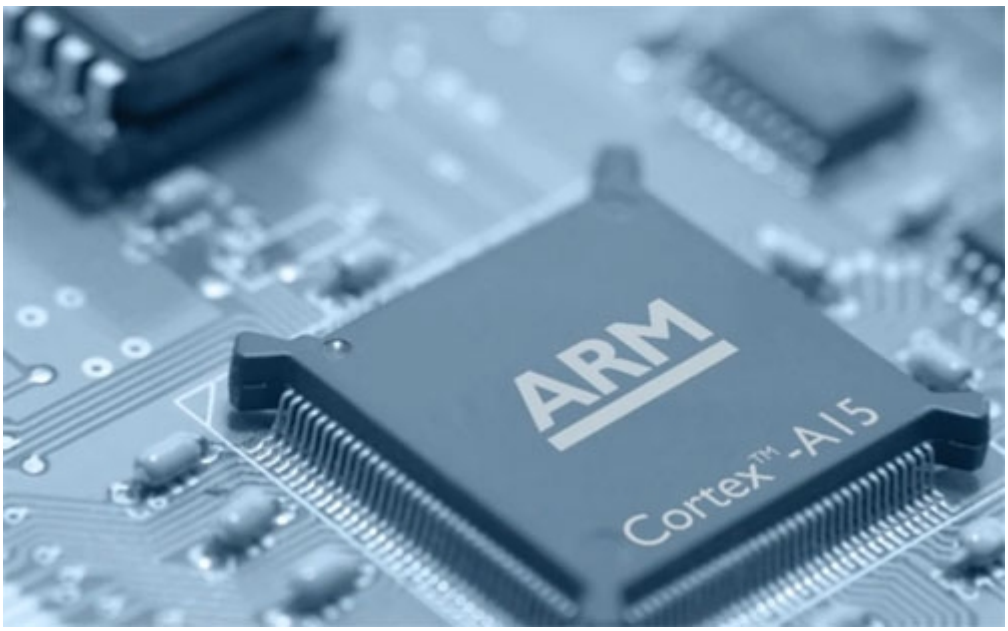


FIG.3.7: ARM CORTEX A15

L'architettura ARM definisce come un processore ARM deve operare. Questo include:

- Instruction Set;
- La configurazione del sistema;
- La gestione delle eccezioni;

- Il modello della memoria.

Ovviamente le nuove eventuali versioni possono modificare o aggiungere proprietà agli elementi di questa lista. Le eventuali modifiche o aggiunte saranno compatibili con le versioni precedenti, in modo da semplificare la migrazione dei vecchi processori alla nuove più recente architettura.
Ogni versione specifica un numero di proprietà di sistema e comportamenti così come definite dall'implementazione. Ad esempio:

- Quanti livelli di cache implementate, e le loro dimensioni;

- Tutte le funzionalità offerte dal Registro di Controllo Ausiliario;

Addizionalmente un'architettura può anche definire estensioni opzionali quali, per esempio:

- supporto hardware (VFP) “floating point” , introdotto in ARMv6;

- il supporto avanzato SIMD (NEON), introdotto in ARMv7.

Versioni precedenti dell'architettura ARM descrivono architetture comuni con differenti scelte di implementazione. Per esempio, un processore può essere implementato con un'Architettura con SISTEMA di MEMORIA VIRTUALE (VMSA, Virtual Memory System Architecture, basata su una MMU(Memory Management Uni), od un'Architettura con Sistema di Memoria Protetta(PMSA, Protected Memory System Architectur), basata su una MPU (Memory Protection Unit).

In una delle ultime versioni, ARMv7 nello specifico, sono stati introdotti i concetti di PROFILO ARCHITETTURALE per individuare determinate versioni dell'architettura utili a differenti tipi di processori per differenti segmenti del mercato. I profili definiti nell'ARMv7 sono:

- **A**: Il profilo “Application” definisce un'architettura del microprocessore basata su VMSA. Un'architettura adatta per sistemi ad alte prestazioni. Con questo profilo si può **eseguire tutte le funzionalità di un sistema operativo.** Inoltre supporta gli Instruction set ARM e Thumb;

- **R**: Il profilo “Real-time” definisce un'architettura del microprocessore basata su PMSA. Un'architettura adatta a sistemi che necessitano di “timing” deterministico e bassa latenza d'interrupt. Anche in questo caso si supportano gli instruction set ARM e Thumb;

- **M**: Il profilo “Microcontroller” fornisce processamenti di interrupt a bassa latenza che sono accessibili direttamente da un linguaggio di programmazione ad alto livello. Ha un modelo di gestione delle gestioni diverso rispetto agli altri due profili. Inoltre implementa una variante delle architetture PMSA e supporta soltanto una variante del Thumb come instruction set.

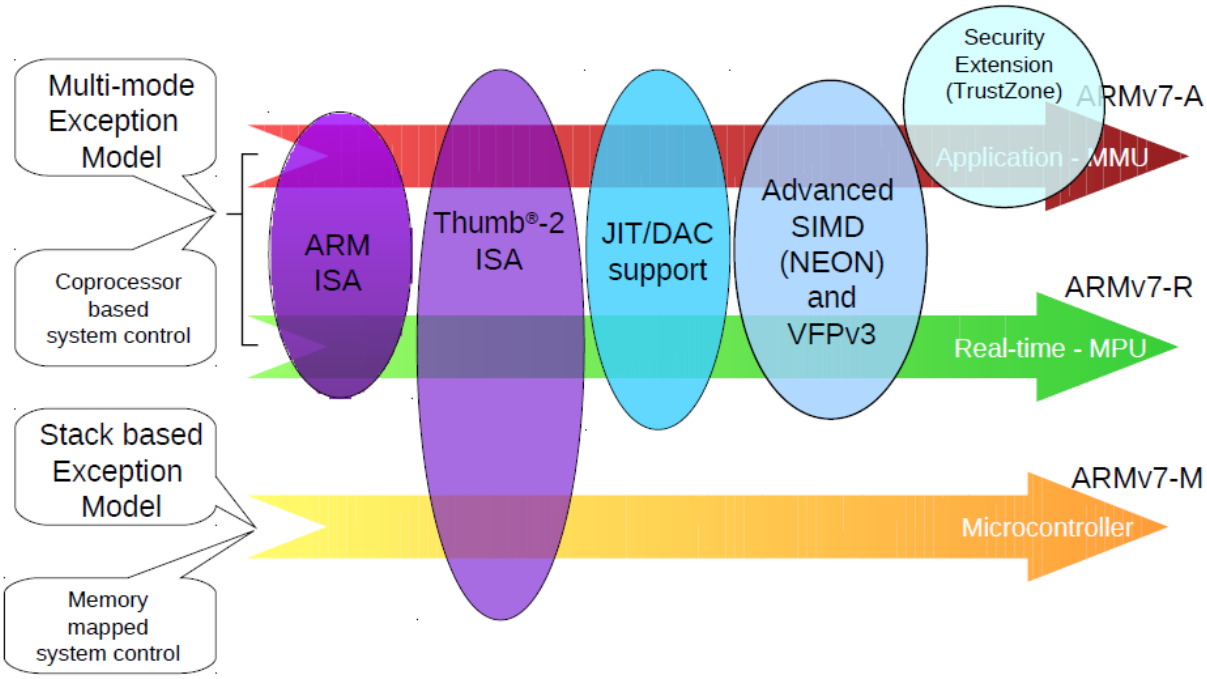


FIG.3.8: Overview di feature ed estensioni dei tre profili di microprocessori ARM.

Riguardo a questi 3 profili la fig. 3.7 definisce chiaramente quali feature chiavi ed estensioni sono state incluse nelle varie implementazioni.

Come detto precedentemente, l'architettura ARM è simile a quella RISC (Reduced Instruction Set Computer), infatti incorpora le seguenti proprietà architetturali tipiche proprio della seconda:

- "**Uniform register file load/store architecture**", dove l'attività di processamento dei dati opera solo sul contenuto dei registri, quindi non direttamente sul contenuto della memoria;

- "**Simple addressing modes**", con tutti gli indirizzi "load/store" determinati soltanto dai campi delle istruzioni e dal contenuto dei registri.

Miglioramenti rispetto alla RISC hanno permesso ai processori ARM di raggiungere un buon bilanciamento fra alte prestazioni, piccolo dimensionamento del codice, basso consumo di potenza e piccola area in silicone occupata.
Benchè l'ultima evoluzione architetturale dei processori ARM sia quella a 64 bit: ARMv8, come si vedrà più avanti, il software GEM5 permette la simulazione di processori a 32 bit ARMv7-A con estensione multi-processore. Nello specifico è incluso il supporto per Thumb®, Thumb-2, VFPv3 (32 double register variant) e NEON™.

Abbiamo quindi definito il framework di memoizzazione, le funzioni economiche che prenderò come strumento per le successive valutazioni e l'architettura di riferimento. Nel successivo capitolo verrà esposto il "setup" del progetto definendo le linee guida di evoluzione dello stesso e presentando gli strumenti utilizzati. [18]

4. Il progetto:

Strumenti utilizzati e fase di setup

L'obiettivo preposto è, come già accennato, quello di valutare l'impatto che diverse gerarchie di memoria (con diverse parametrizzazioni in termini di dimensioni e latenze dei singoli componenti), su un architettura ARM, possano avere sul meccanismo di MEMOIZZAZIONE. Dato che il framework spiegato nel precedente capitolo lavora sul bytecode di applicazioni Java, il primo problema da risolvere, o comunque la prima vera valutazione da fare, è stata la scelta di strumenti di emulazione/simulazione di architetture ARM e di una Java Virtual Machine (JVM).

I primi studi quindi hanno avuto l'obiettivo di capire se effettivamente esistesse una Virtual Machine Java per architetture ARM e, in caso di risposta positiva, come fosse possibile configurarla su un sistema operativo avviato tramite un simulatore.
L'approccio iniziale ha visto utilizzare un EMULATORE: QEMU, a cui è stato fatto eseguire una versione per ARM di DEBIAN LINUX. Su questo sistema simulato si sono valutate diverse versioni open source di Virtual Machine Java, la scelta è ricaduta sulla JAMvm. La configurazione però ha richiesto un lungo ed ostico periodo e dato che, in questo frangente, la Oracle ha reso disponibile una sua versione di JVM compatibile con ARM, si è optato per l'utilizzo di questa perchè sicuramente più supportata della precedente.
L'emulatore QEMU, benchè veloce ed utile allo scopo precedente, non è risultato però adatto agli obiettivi finali del progetto, dato che non rende possibile la modifica dei parametri architetturali.
Fra i vari SIMULATORI utilizzati in questo tipo di ricerche, uno open source dei più conosciuti e supportati da una comunità folta ed attiva è GEM5. Nel prossimo paragrafo sarà introdotto questo simulatore e, nello specifico, saranno individuate quali possibilità metta a disposizione in termini di variazione di parametri architetturali relativi alle memorie ed ai

componenti di comunicazione fra queste e il processore.
Successivamente, rispetto ad una ricerca di mercato su quali siano le offerte proposte attualmente in ambito server relativamente ai dimensionamenti ed alle gerarchie di memoria, sarà anche spiegato come è stato modificato il codice sorgente del simulatore per permettere di adattarlo alle esigenze di ricerca, così da avere una base completa per la valutazione effettiva delle simulazioni e dei relativi risultati che saranno oggetto del capitolo 5.

4.1 GEM5

Nel campo di ricerca delle Architetture dei Computer e delle Analisi di Potenze, i ricercatori usano un gran numero di tool software, fra cui: simulatori di architetture, generatori di statistiche e analizzatori di potenze.

Il processo di simulazione ed analisi è composto da alcuni precisi step:

- Definizione dell'architettura da simulare, in termini di struttura hardware e, ovviamente, di instruction set;
- Scelta dei Benchmarks da assegnare ai differenti processori della precedente architettura (o al singolo processore se in Multicore);
- Recupero delle statistiche di sistema usando il simulatore;
- Fornire le corrette statistiche di sistema come input all'analizzatore di potenze;
- Valutazione del consumo di potenze;

Rispetto alla lista appena riportata, sarebbe opportuno trovare un simulatore di architetture che permetta di svolgere almeno i primi tre punti ed un analizzatore di potenze multilivello che invece si occupi dei restanti ultimi due.
Per altro sarebbe utile anche un tool in grado di proiettare sotto forma di grafici i dati sui consumi di potenza definiti dall'analizzatore, così da rendere più semplice l'individuazione di informazioni utili allo scopo.

Fra i simulatori attualmente disponibili si è scelto di utilizzare GEM5.
L'infrastruttura di questo tool è l'integrazione dei migliori aspetti di altri due software: M5 e GEMS. In particolare, M5 fornisce un framework di simulazione ad alta configurabilità, ISA multiple e diversi modelli di CPU, mentre GEMS completa queste proprietà con un sistema di memoria dettagliato che include un supporto per protocolli di coerenza per cache multiple e diversi modelli di interconnessione dei componenti.
La versione corrente di GEM5 supporta le più importanti ISA commerciali (ARM, ALPHA, MIPS, Power, SPARC, e x86), e mette a disposizione il boot di sistemi operativi linux su 3 di queste (ARM, ALPHA, and x86).[19]

I ricercatori che si occupano delle architetture dei computer usano software di simulazione per prototipare e valutare le proprie idee. Per questo scopo necessitano di un framework che sia capace di valutare una vasta diversità di design architetturali e che supporti tante utility di facilitazione, tipiche dei Sistemi operativi, che si occupino di gestire operazioni I/O, di networking e che permettano di collaborare agevolmente con i colleghi.
Lo scopo principale del simulatore GEM5 è di essere un “community tool” focalizzato sulla modellizzazione architetturale. Tre aspetti chiave per questo scopo sono: la possibilità di modellizzazione flessibile che riesca a richiamare l'attenzione di un vasto range di utenti, la larga disponibilità e utilità della community di sviluppo, e l'alto livello di interazione fra gli sviluppatori per favorire la collaborazione.

GEM5 permette l'esecuzione di sistemi simulati in 2 modalità:

- **Syscall Emulation** (SE), in cui emula la maggior parte dei servizi offerti dal sistema operativo e quindi rende NON necessario occuparsi della gestione dei device e/o comunque avviare un sistema operativo dentro cui far eseguire i propri benchmark;

- **FullSystem Emulation** (FS), in cui vengono eseguite istruzioni sia user-level che kernel-level e modella un sistema completo, inclusi quindi il sistema operativo ed i device.

A questo proposito, nelle fasi iniziali di progetto, si è cercato di sfruttare la prima delle due modalità facendo eseguire la JVM in SE per poi assegnargli il benchmark da mandare in esecuzione. Molte funzionalità e servizi di sistema, necessari per queste operazioni, non sono ancora state implementate nell'attuale versione di GEM5 e quindi si è passati alle simulazioni in modalità FS.
Un ulteriore punto importante a favore di GEM5 è l'integrazione dello stesso con Python.

Nonostante l'85% del simulatore sia scritto in C++, Python si occupa di coprire tutti gli aspetti riguardanti le operazioni svolte dal simulatore. [19]

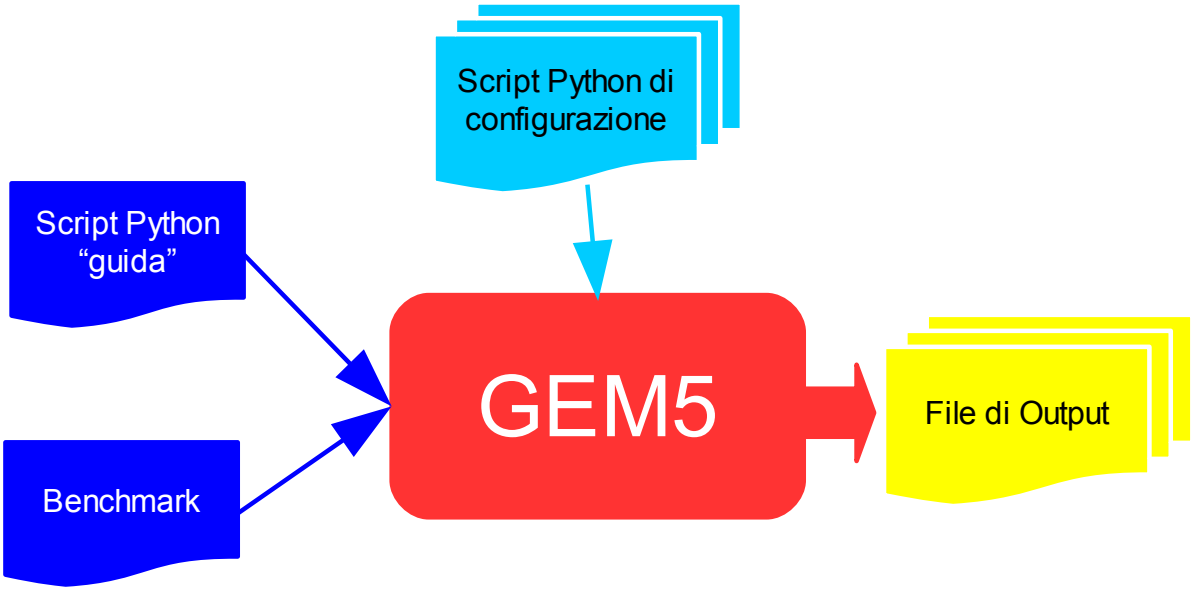


FIG. 4.1: Input e Output del simulatore GEM5

In particolare, come si vedrà in dettaglio nel paragrafo 4.1.2, tramite python saranno definiti e modificati i componenti architetturali hardware simulati e un ulteriore script "guida" delle diverse simulazioni effettuate.
In questa modalità, vedi figura 4.1, GEM5 prende in input il benchmark e gli script python definiti pocanzi e produce in output, fra i vari, un file di testo strutturato che riepiloga la configurazione hardware dell'architettura appena simulata ("config.ini") ed un secondo file di testo comprendente diverse statistiche calcolate relativamente all'esecuzione fatta ("stat.txt"). Due strumenti veramente utili in fase di ricerca e valutazione e che saranno oggetto di dettaglio del prossimo paragrafo.

4.1.1 Output del simulatore

Come si evince dalla figura 4.1, GEM5 produce come output diversi file, ovviamente riferiti alla simulazione avviata, che nel caso di esecuzione in modalità FS sono:

- **File di configurazione** (figura 4.2): sono due file, dalla funzionalità parallela, che danno una visione testuale strutturata della componentistica hardware dell'architettura simulata. I formati sono ".ini" e ".json". La struttura ricalca ciò che viene definito dal file guida di configurazione e simulazione e, come si evince dalla figura 4.2, nel formato ".ini" la struttura ad albero creata parte sempre da un oggetto "root" e, per questo come per le altre foglie dell'albero, la proprietà "children" rappresenta la lista delle sotto foglie ad esso collegate. Inoltre, per ogni componente foglia, vengono proposti vari parametri fra cui le porte utilizzate come metodo di connessione (dettaglio sul meccanismo "a porte" nel prossimo paragrafo).

```
[root]
type=Root
children=system

(....)

[system]
type=LinuxArmSystem
children=bridge cf0 cpu intrctrl iobus iocache membus physmem realview terminal vncserver
kernel=/home/gem5-stable-f75ee4849c40/vmlinux-emm-pcie/vmlinux-3.3-arm-vexpress-emm-pcie

(....)

[system.cpu]
type=AtomicSimpleCPU
children=dcache dtb icache interrupts itb tracer
clock=286
(....)
dcache_port=system.cpu.dcache.cpu_side
icache_port=system.cpu.icache.cpu_side

(....)
```

FIG. 4.2: file rappresentativo delle configurazione hardware del sistema simulato: "config.ini"

- **File di log**: un file di registro delle interazioni fatte col sistema (in input e output) via terminale (per maggiori dettagli su modalità di connessione e dialogo col sistema simulato si faccia riferimento al paragrafo A.2.3 dell'appendice A);

- **File di statistiche** (figura 4.3): è un file di testo che conterrà tutte le informazioni sulle statistiche calcolate in relazione ai componenti effettivamente definiti per l'architettura simulata e quindi in coerenza col i file di configurazione. È possibile effettuare operazioni di salvataggio delle statistiche in qualsiasi momento sfruttando i file di sistema ".rcs" (come si vedrà più avanti) e/o l'utility "M5TERM" messa a disposizione dal simulatore (ulteriori dettagli al paragrafo A.2.3 dell'appendice A). Come si evince dalla figura x, vengono riproposte le statistiche generali di simulazione, seguite (in figura v'è ovviamente solo un sunto) dalle statistiche del sistem. La definizione del parametro statistico di riferimento è collegato (visivamente col ".") al componente in esame ed al rispettivo "genitore" in modo ricorsivo, fino ad arrivare al componente "system" (quello immediatamento sotto il "root") seguendo fedelmente lo schema ad albero definito nel file di configurazione.

```
---------- Begin Simulation Statistics ----------
sim_seconds                                  0.086005   # Number of seconds simulated
sim_ticks                                 86004579804   # Number of ticks simulated
final_tick                               460268889652   # Number of ticks from beginning of simulation (re
sim_freq                                1000000000000   # Frequency of simulated ticks
host_inst_rate                               6042577   # Simulator instruction rate (inst/s)
host_op_rate                                 8078207   # Simulator op (including micro ops) rate (op/s)
host_tick_rate                             561774367   # Simulator tick rate (ticks/s)
host_mem_usage                               1463176   # Number of bytes of host memory used
host_seconds                                  153.09   # Real time elapsed on the host
sim_insts                                  925084699   # Number of instructions simulated
sim_ops                                   1236728809   # Number of ops (including micro ops) simulated
(...)
system.physmem.num_reads::total              6659721   # Number of read requests responded to by this mem
system.physmem.num_writes::total              967971   # Number of write requests responded to by this me
(...)
system.cpu.numCycles                       300630026   # number of cpu cycles simulated
system.cpu.numWorkItemsStarted                    0   # number of work items this cpu started
system.cpu.numWorkItemsCompleted                  0   # number of work items this cpu completed
system.cpu.committedInsts                  188354582   # Number of instructions committed
system.cpu.committedOps                    242613370   # Number of ops (including micro ops) committed
system.cpu.num_int_alu_accesses            217372675   # Number of integer alu accesses
(...)
system.cpu.icache.replacements               2997968   # number of replacements
system.cpu.icache.tagsinuse                511.998952   # Cycle average of tags in use
system.cpu.icache.total_refs                 187551962   # Total number of references to valid blocks.
(...)
system.cpu.dcache.replacements               2109365   # number of replacements
system.cpu.dcache.tagsinuse                511.999043   # Cycle average of tags in use
system.cpu.dcache.total_refs                  93734722   # Total number of references to valid blocks.

---------- End Simulation Statistics   ----------
```

FIG. 4.3 file riassuntivo delle statistiche di simulazione calcolate: "stat.txt"

4.1.2 Modello della memoria e parametri modificabili

Per lo scopo del progetto non è fondamentale valutare i diversi modelli di processore messi a disposizione dal simulatore, per questo motivo si soprassede, in questa sede, alla definizione degli stessi (per ulteriori informazioni sul tema fare riferimento al paragrafo A.2.1 dell' appendice A).

Per la spiegazione dei modelli di sistema di memorie previsti nel simulatore è opportuno specificare che ogni singolo componente dell'architettura simulata è inteso come un Object del sistema. In particolare, per le memorie, tutti gli oggetti ereditano da una classe generica: MemObject che fornisce metodi di base utili, fra cui l'accesso ai componenti necessari per definire le interconnessioni e le comunicazioni con gli altri elementi del sistema.

Il metodo di connessione generico degli oggetti simulati relativi alle memorie è quello "a porte". Queste sono utilizzate per interfacciare fra loro gli oggetti e permettono la definizione modulare dei sistemi ed il non dover preoccuparsi di creare una specifica interfaccia di comunicazione per ogni oggetto.

L'integrazione con Python, elogiata prima, permette agevolmente di connettere gli elementi assegnando le varie porte agli identificatori dei bus di collegamento e/o degli elementi cache

e simili. Inoltre, per i componenti che possono avere numerose porte tipo i bus, sono predisposti degli specifici “vector ports” in cui le connessione sono gestite in “append” per ordine di definizione dal codice.
Le porte sono di diverso tipo ed ereditano tutte da una classe Port generica. In particolare si individuano due tipi principali:

- **FunctionalPort**, che forniscono metodi semplici da utilizzare per scrittura e lettura di indirizzi fisici. Vengono utilizzate solamente per caricare dati nella memoria e/o aggiornare costanti prima che la simulazioni abbia inizio;

- **VirtualPort**, che forniscono gli stesso metodi della precedente ma l'indirizzo passato è virtuale ed è quindi richiesta e svolta una traduzione per determinare l'indirizzo fisico.

Esistono tre tipi di accesso supportati dagli oggetti “porta”:

- **Timing**: è il tipo di accesso più dettagliato e che riflette al miglior modo la temporizzazione realistica. Include la modellizzazione di code di ritardo e di contesa delle risorse;

- **Atomic**: è meno dettagliato del precedente a favore di una maggiore velocità. Sono accessi usati per il “fast forwarding” delle cache e restituiscono una valutazione approssimata del tempo di completamento di una richiesta non tendendo conto di eventuali code di ritardo ne di tempi relativi alla contesa di risorse. Questo tipo di accessi non può coesistere con i precedenti su una stessa architettura;

- **Functional**: similmente agli accessi “atomic”, quelli “functional” sono istantanei ma possono coesistere assieme ad accessi di tipo “timing”.Sono usati generalmente per caricare file binari, esaminare/cambiare variabili del sistema simulato e permettono ad un “debugger” remoto di poter connettersi al simulatore;

Il primo tipo di accesso, essendo più accurato e dettagliato, necessita a definizione di un flusso di controllo.

Stante i concetti di base generali appena esposti, GEM5 mette a disposizione due differenti modelli di sistema di memoria: **Classic**, ereditato dal sistema predisposto in M5; **Ruby**, basato invece sul modello di sistema di memoria di GEMS. I modelli sono complementari e offrono unici vantaggi e svantaggi.

Il primo fornisce un sistema veloce e facile da configurare a discapito dell'accuratezza e della flessibilità. Tutti gli oggetti di questo modello ereditano dalla generale classe “MemObjects” e sono connessi logicamente con un meccanismo “a porte”. Questo tipo di meccanismo supporta la connessione “point-to-point” tra MemObject connettendosi ai diversi bus di comunicazione.

I protocolli di coerenza delle cache sono mantenuti usando un protocollo di snooping astratto in cui le sonde vengono trasmesse in maniera istantanea lungo la gerarchie di memoria.
Usando queste metodologie il modello Classic presenta i seguenti vantaggi:

- **Fast Forwarding**: il modello classic supporta accessi alle porte di tipo “atomic” che, come detto prima, sono più veloci che dettagliate. Questa modalità operazionale è paricolarmente utile quando si ha la necessità di portarsi avanti (“fast forward”) e valutare solo determinate parti d'interesse dell'esecuzione;

- **Velocità**: Oltre agli accessi “atomic”, il modello Classic ha anche accessi di tipo “timing” relativamente veloci rispetto a quelli del modello Ruby;

- **Facilità di configurazione**: semplicemente modificando la configurazione scritta in Python, il Modello Classic permette di creare gerarchie di memoria arbitrarie. Il protocollo astratto di coerenza delle cache si estende automaticamente a qualsiasi tipo di gerarchia composta da cache, bus e cpu;

e, di contro, i seguenti svantaggi:

- **Cache Coherence Flexibility**: anche se questo modello permette la definizione di gerarchie arbitrarie di memoria collegando componenti cache, bus e cpu. Esso è ristretto dal protocollo di snooping astratto. La modifica di questo protocollo richiederebbe degli sforzi troppo elevati;

- **Accuracy**: Non modellando le transazioni di stato e la temporizzazione delle sonde, il modello classico non simula i protocolli di contesa delle risorse così accuratamente come invece fatto dal secondo modello.

Il modello Ruby, rispetto al precedente, sacrifica la velocità di simulazione ma fornisce un'infrastruttura flessibile capace di simulare con accuratezza una vasta gamma di sistemi. In particolare viene supportato un linguaggio specifico del dominio, chiamato SLICC (Specification Language for Implementing Cache Coherence) tramite il quale è possibile definire differenti tipi di protocolli di coerenza per le cache. Essenzialmente SLICC definisce le cache, la memoria, ed il Controller DMA (Direct Memory Access) come “macchine” di stato individuali per-memory-block che, insieme, formano il protocollo. Attraverso la definizione di controller logici tramite un linguaggio di alto livello, SLICC permette a differenti protocolli di incorporare stessi meccanismi di transizione di stato con un minimo sforzo da parte del programmatore.

Rispetto al modello classico, Ruby non sfrutta il meccanicmo “a porte” per la connessione degli oggetti interni del sistema di memoria. Le porte, in questo caso, connettono solo le cpu ed i device esterni al sistema di memoria tramite l'oggetto RubyPort. All'interno del sistema Ruby vengono poi connessi i vari oggetti tramite MessageBuffers. Questi ultimi sono simili alle porte nel senso che forniscono agli oggetti delle interfacce standard di comunicazione. In particolare però, rispetto alle porte, forniscono una coda che immagazzina i messaggi. Da questo coda i messaggi non possono essere inseriti e tolti nello stesso ciclo di clock simulato e quindi la comunicazione non è istantanea come nel caso Classic, questo implica che i MessageBuffers supportano solo un accesso di tipo timing e non quindi accessi functional o atomic.

Date le possibilità di cui sopra ,ecco quindi i vantaggi di questo modello:

- **Cache Coherence Flexibility**: Utilizzando SLICC, Ruby può implementare una grande varietà di protocolli di coerenza delle cache, che vanno da quelli “directory” a quello “snooping” e tanti ibridi che stanno in mezzo;

- **Accuracy**: Ruby, nel suo sistema di memoria, modella accuratamente sia la coerenza della cache che le proprietà relative alle reti. In particolare la metodologia di gestione degli eventi, basata sui messaggi SLICC, modellizza accurtamente le transizioni di stato temporali. Ed ancora, il modello di rete Garnet, integrato in Ruby, modelliza accuratamente le contese di rete ed il flusso di controllo;

e gli svantaggi:

- **Fast Forwarding**: Ruby non supporta accessi di tipo “atomic” e, quindi, non ha ragionevoli capacità di fast forwarding;

- **Velocità**: Comporato con il modello Classic, quello Ruby è relativamente lento. Questo fatto è evidente soprattutto quando si usa il modello di rete Garnet.

- **Facilità di configurazione**: Nonostante SLICC sia un tool davvero potente per modellare una vasta gamma di protocolli per la coerenza delle cache, i protocolli risultanti sono ottimizzati solo per una specifica configurazione gerarchica di cache. A causa di cio, risulta difficile estendere il protocollo definito ad un'altro livello di cache.

L'adottare un modello di tipo Ruby sarebbe stato più utile per il progetto. Purtroppo però, nell'attuale versione del simulatore, questo sistema non è implementato per architetture ARM quindi si è adottato un sistema di tipo Classic. [20]

Per quanto riguarda i parametri effettivamente definibili delle strutture di memoria, la tabella 4.1 riporta un sunto dei punti su cui è possibile attuare modifiche/scelte in relazione alle proprie necessità.

COMPONENTE	NUMERO	DIMENSIONE	LATENZA
Cache livello 1	illimitato	illimitata	Variabile (> 1ps)
Cache livello 2	illimitato	illimitata	Variabile (> 1ps)
Memoria RAM	1	Max 1024MB	Variabile (> 1ps)
Bus	illimitati	-	Variabile (> 1ps)

TAB. 4.1: Lista dei parametri che è possibile modificare tramite il simulatore e relativo range.

Le cache sono effettivamente definibili senza limiti in numero e dimensione ma quest'ultima deve essere una potenza del 2. La latenza viene espressa in “ns” e, per le cache, si intende come il tempo di hit di quel particolare livello.
I bus sono definibili anch'essi in maniera illimitata e associabili ai diversi componenti. Per quanto riguarda le memorie RAM invece c'è un grosso limite in numero e, soprattutto in dimensione, dato che non si può superare 1GB. Per altro la modifica della velocità di processamento d'informazioni della stessa non è agevolmente modificabile dato che si dovrebbe mutare il codice sorgente del simulatore e quindi “mettere” mano al codice C++ ed alle strutture SWIG di connessione fra questo e la parte scritta in Python.
Per il processore, non riportato in tabella 4.1, sono modificabili diverse proprietà ma, per il progetto, l'unica d'interesse è la velocità di computazione dello stesso. Quest'ultima è effettivamente modificabile senza alcun limite.

4.1.3 utilizzo del simulatore

Nei precedenti discorsi sono sicuramente state evidenziate le enormi possibilità che GEM5 metta a disposizione del ricercatore/sviluppatore. All'atto pratico, stante la possibilità di sviluppo e modifica del codice sorgente C++ e sottointendendo l'aver già costruito un'architettura di base ARM simulata su cui lavorare, (dettagli a riguardo in appendice A paragrafo A.2.2) l'utilizzo per la definizione di architetture da simulare, adattate ai propri scopi, e l'avvio delle simulazioni avvengono tramite la creazione/modifica di script python. In particolare per il primo task ,se necessario, si possono modificare script python comuni alle varie simulazioni che riguardano individualmente le cache (“Caches.py”), la configurazione delle stesse(“ConfigCache.py”), il Sistema operativo(SO.py), i processori (CPU2000.py), ecc.... L'avvio e le linee guida di simulazione, insieme alla definizione/variazione di determinati parametri dei diversi componenti, sono definiti in un ulteriore file script Python passato da linea di comando al simulatore in fase di avvio.

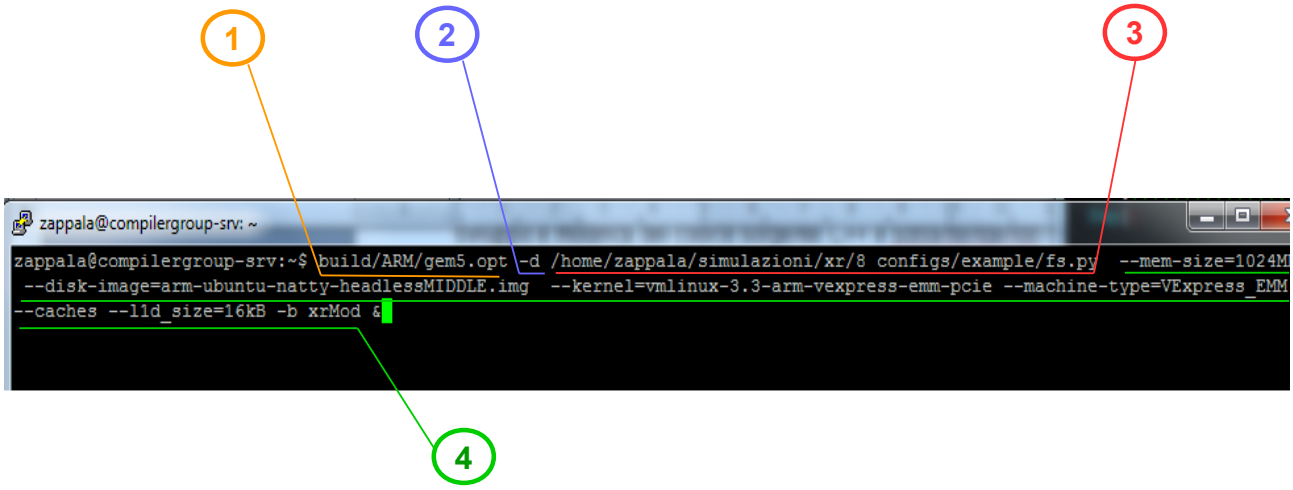


Fig.4.5: Esempio comando di avvio simulazione GEM5

La figura 4.5 mostra un esempio di stringa di avvio del sistema ARM in modalità FS. I particolare, per l'avvio, si scelgono:

1. L'eseguibile che si riferisce al sistema ARM precedentemente creato tramite (vedi appendice A paragrafo A.2.2 per dettaglio);

2. Determinati “switch” associati al comando del punto 1, fra cui, come si vede in questo esempio, la possibilità definire un path, diverso da quello di default, per il salvataggio dei file di output del sistema;

3. Il riferimento al file di guida e configurazione della simulazione;

4. Determinati “switch” relativi al comando del punto 3. Questi sono modificabili o meglio estendibili tramite il file del punto 3. Alcuni di quelli definiti di default sono utili allo scopo del progetto ed in particolare:

- “--caches”: di tipo booleano, se incluso nel comando d'avvio allora viene inserito nel sistema un primo livello di cache separati fra Instruction (“I”) e Data (“D”);

- “--l2cache”: anch'esso di tipo booleano, se incluso nel comando d'avvio viene inserito nel sistema un secondo livello di cache.

- “--l1d_size”: la dimensione in KB o MB del primo livello di cache di tipo D;

- “--l1i_size”: idem come il precedente ma per primo livello di cache di tipo I;

- “--l2_size”: sempre per la dimensione ma del secondo livello di cache;

- “--kernel”: indica il kernel con cui avviare il sistema;

- “--disk-image”: indica il path del file immagine col sistema operativo configurato;

- “--mem_size”: la dimensione della memoria RAM (massimo consentito: 1024 MB).

Dato che l'analisi del progetto ha riguardato architetture con anche 3 livelli di cache, si è resa necessaria la modifica del codice sorgente per raggiungere questo scopo. Le modifiche sono riportare nel paragrafo 4.3.
In modalità FS, è possibile sia avviare un intero sistema operativo Linux pre-configurato in un particolare file immagine, sia modificare/utilizzare dei file “.rcs” attraverso cui è possibile definire dei benchmark che vengano eseguiti dal simulatore all'avvio del sistema operativo simulato. In particolare si dovrebbe inserire il proprio benchmark, ed eventuali opzioni generiche, nella lista dei benchmark supportati (“Bernchmarks.py”) e definire un nuovo file “.rcs” ad-hoc in cui inserire le operazioni che il sistema deve svolgere automaticamente al suo avvio. Questo, come si vedrà successivamente, permette di poter avviare simulazioni mirate all'esecuzione di particolari programmi e poter, grazie ad un ulteriore strumento messo a disposizione da GEM5, eliminare dalle statistiche calcolate (quelle scritte nel file “stat.txt” a fine simulazione) degli overhead relativi a operazioni del sistema operativo non riguardanti l'esecuzione del benchmark stesso utilizzato.
Gli switch elencati precedentemente sono riferiti, come detto, al file di configurazione e guida della simulazione. Fra quelli invece inerenti l'eseguibile ririferimento dell'architettura (per intenderci: quelli da post-porre a “build/ARM/gem5.opt”) v'è uno molto utile: “-r” seguito da un

numero intero positivo.

Questo switch permette di avviare la simulazione da un checkpoint precedentemente definito. Questo meccanismo, messo a disposizione dal simulatore, permette, in qualsiasi istante della simulazione, di creare dei punti di ripristino (checkpoint) tramite comandi inviati con “m5term” (vedi paragrafo A.2.3 dell'appendice A e figura. 4.6).

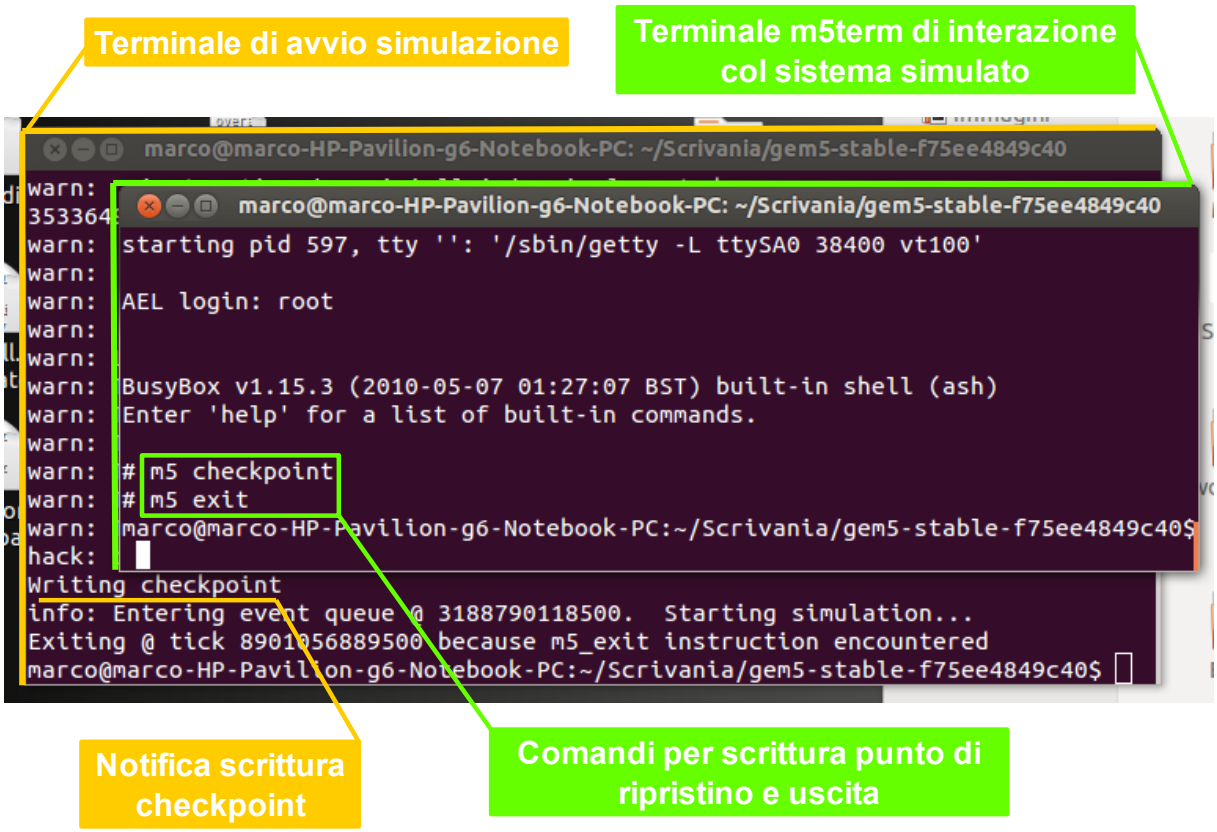


Fig.4.6: Esempio creazione chekpoint della simulazione tramite tool “m5term”

Il checkpoint fa un salvataggio della memoria RAM in quell'istante (non tiene conto di ciò che è nelle memorie cache) per permettere, in un secondo momento, di riavviare la simulazione del sistema da quell'istante evitando di perdere del tempo per il riavvio del sistema e/o comunque nei casi in cui si rende necessaria la suddivisione della simulazione in più

tranche. I checkpoint sono gestiti da GEM5 con una lista crescente a partire dal più recente, richiamabile in fase di avvio della simulazione appunto con lo switch “-r” ed un numero intero relativo alla posizione del checkpoint sulla lista appena citata. Notare che la lista è unica e non sono fatte distinzioni sull'immagine disco di riferimento della simulazione in cui è stato definito il checkpoint. Sta all'utente tenere un promemoria delle associazioni.

Ulteriori dettagli su modifiche ed effettive implementazioni saranno esposti nel paragrafo 4.3 e nel prossimo capitolo, in relazione alle necessità del progetto svolto.

4.2 Ricerca di mercato

Dopo aver individuato gli strumenti necessari, la JVM ed il simulatore, si è cercato di capire come impostare al meglio l'evoluzione del progetto e, nello specifico, la più corretta strutturazione e programmazione delle simulazioni da effettuare. Un primo passo è stato quello di individuare sul mercato una gamma di possibili macchine reali su cui basarsi per definire le diverse architetture simulate. La ricerca è stata fatta sul campo dei maggiori produttori di server, valutando i pacchetti da loro offerti. In particolare sono state valutate macchine vendute dalla IBM (PowerPC), dall'HP e dalla DELL. Inoltre è stata anche effettuata una selezione dei parametri di adattamento nominali rilasciata dalla Intel in relazione alle possibilità di interfacciamento dei loro processori XEON rispetto a diversi livelli, dimensioni e velocità di cache, memorie e altri dispositivi di storage e inteconnesione.

La tabella seguente mostra un sunto strutturato dei risultati della ricerca evidenziando i parametri interessanti anche relativamente alle effettive possibilità di customizzazione offerte da GEM5

MODELLO	CPU*	CACHE L1*	CACHE L2	CACHE L3	RAM
POWER770	POWER 7+ (**2.8** – **4.2**) GHz	-	**256** KB	**10** MB eDDR dedicata	**DDR3** 1066 Mhz (1 – 4) TB
POWER750	POWER 7 (**3.2**– **3.7**) GHz	-	**256** KB	**4** MB dedicata	**DDR3** RDIMM (8 – 512) GB
POWER775	POWER 7 (**3.8** – **4.0**) GHz	-	-	-	128 DIMM slot fino a 2 TB
POWER LINUX 7R2	POWER 7+ (**3.6** – **4.3**) GHz	-	**256** KB	**10** MB dedicata	**DDR3** RDIMM (32 – 512) GB
POWER710	POWER 7 (**3.0** – **3.7**) GHz	-	**256** KB	**4** MB dedicata	**DDR3** 1066 Mhz (4 – 64) GB
POWER730	POWER 7 (**3.0** – **3.7**) GHz	-	**256** KB	**4** MB dedicata	**DDR3** 1066 Mhz (4 – 128) GB
POWER755	POWER 7 (**2.2** - **3.6**) GHz	-	**256** KB	**4** MB dedicata	**DDR3** 1066 Mhz (128 – 256) GB
DELL 12th GEN. BLADE	INTEL XEON E5 (**1,5** – **3.0**) GHZ	**32** kB	-	**2,5** MB dedicata (fino a 10MB tot)	192GB 1333 MHz o 768GB a 1600MHz
DELL 12th GEN. RACK	“	**32** kB	-	**2,5** MB dedicata (fino a 20MB tot)	“
DELL 12th GEN. TOWER	“	**32** kB	-	**2,5** MB dedicata (fino a 20MB tot)	“
INTEL XEON	5600	**32** KB	Da **256** KB	Fino a **12** MB non dedicata	**DDR3** 800, 1066, 1333 MHz (1,2) GB
“	5500	**32** KB	Da **256** KB	Fino a **12** MB non dedicata	**DDR3** 800, 1066, 1333 MHz (1,2,4,32) GB
“	E3-1200	**32** KB	Da **256** KB	Fino a **8** MB non dedicata	**DDR3** 1066, 1333 MHz (1,2,4,32) GB
“	E7- 8800, 4800, 2088	**32** KB I **16**,**32** KB D*	Da **256** KB	Fino a **30** MB non dedicata	**DDR3** 800, 978, 1067 MHz fino a 32 GB

* tutte le cache di livello 1 sono dedicate per core.
** I = Instruction cache; D= Data cache.

TAB 4.2: Ricerca di mercato, tabella riassuntiva[21]

Dalla tabella 4.2 si evincono valori basilarmente piccoli per cache di primo e secondo livello, in particolare le cache di secondo livello sono generalmente molto piccole rispetto a quelle previste per architetture con soltanto due livelli di cache. La motivazione deriva dal fatto che

sono state adottate memorie cache di terzo livello di tipo “embedded DRAM” (eDRAM) che, almeno nominalmente, assicurano delle latenze inferiori ai 5ns, il che a sua volta permette di diminuire in dimensione la cache del livello appena sopra (L2).
Rispetto ai vincoli ed alle possibilità di scelta, messe a disposizione dal simulatore, e rispetto ai risultati della ricerca di mercato di cui sopra, in tabella x sono riassunti i parametri d'interesse che saranno cambiati in fase di simulazione ed i rispettivi range di variazione.

CACHE L1	Dimensione	16 - 32 kB
	Latenza	0.5 , 1 ns
CACHE L2	Dimensione	256 kB
	Latenza	2 , 10 ns
CACHE L3	Dimensione	2 , 4 , 8 MB
	Latenza	4 , 20 ns
RAM	Latenza	30 , 50 , 75 , 100 ns
CPU	Velocità	2, 3, 3.5, 4 GHz

TAB 4.3: parametri e variazioni oggetto di modifica per le simulazioni

Rispetto alla tabella 4.3, come si vedrà nel prossimo capitolo, sono state fatte ulteriori simulazioni aumentando il range di determinati parametri. Queste simulazioni si sono rese necessarie per valutare al meglio l'andamento in certi casi particolari e per poter trarre conclusioni con maggior certezza.

5. Il Progetto:
Simulazioni e risultati

Nel precedente capitolo sono state poste le basi che hanno portato a definire la struttura ed il processo di svolgimento delle simulazioni necessarie a raggiungere lo scopo del progetto. In particolare, in relazione alla tabella 4.2 sulle offerte del mercato attuale e alle possibilità di “customizzazione” messe a disposizione da GEM5, si è definito un “albero” di ricerche possibili con relative casistiche di scelta del percorso da seguire strada facendo.
I percorsi di analisi sono stati suddivisi per scopo e per funzione da utilizzare, in particolare si è cercato di valutare l'impatto sulla tecnica di memoizzazione in base alla variazione:

1. dei livelli di memoria cache;
2. delle dimensioni delle cache;
3. della frequenza del processore;
4. delle latenze dei diversi componenti di memoria;

questo percorso è stato seguito dapprima per tutte e quattro le funzioni economiche modificate con la tecnica di memoizzazione e, successivamente, anche per le 4 funzioni originali, in modo da poter avere un confronto anche fra le prime e le seconde.
La figura 5.1a è quindi da intendersi come percorso di simulazioni relativo ad una sola delle quattro funzioni. Percorso ripetuto per le successive sette, a meno della fase due (figura 5.1b) che è stata attuata solo per il più significativo benchmark “XIRR”.

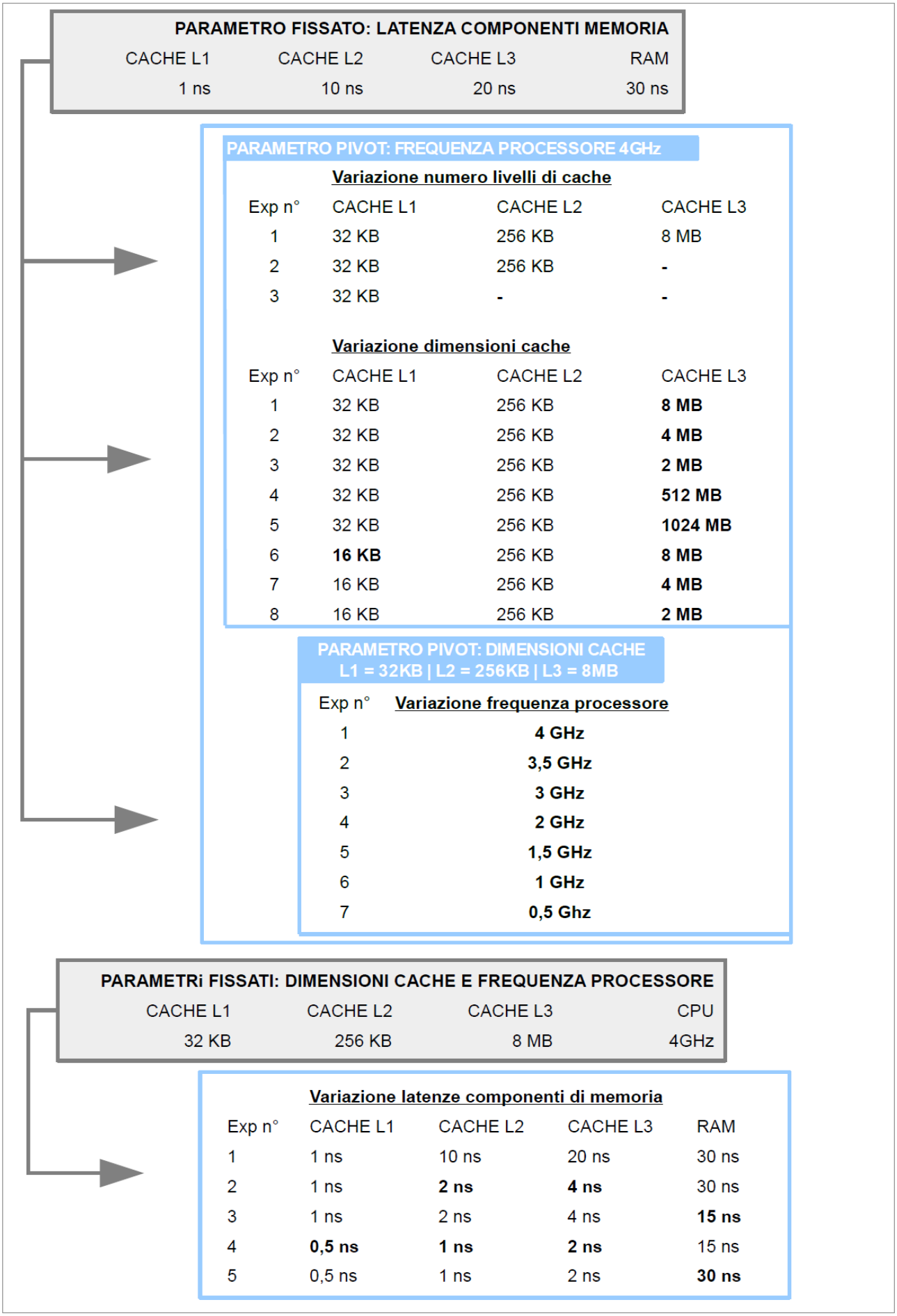


FIG 5.1a: Schema delle simulazioni parte 1

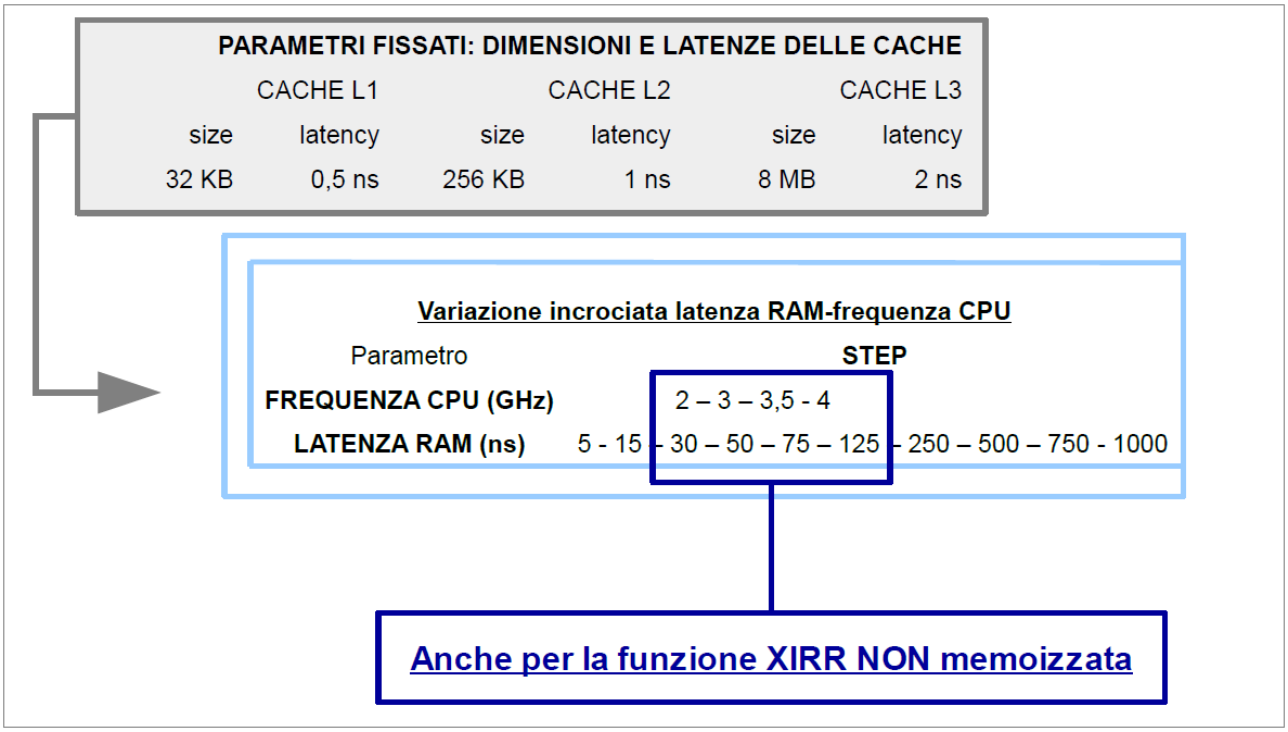


FIG 5.1b: Schema delle simulazioni parte 2

Nel prossimo paragrafo vengono definite le caratteristiche tecniche della macchina host utilizzata per i test e le modalità d'interfacciamento con la stessa.

Nel paragrafo successivo saranno invece mostrate dapprima le modifiche apportate al simulatore per adattarlo alle esigenze di ricerca e, successivamente, saranno evidenziate le simulazioni, in relazione all'albero di figura 5.1 (a e b), con relative valutazioni e scelte dei passi successivi da compiere, in base alle analisi man mano effettuate.

5.1 Specifiche tecniche

Le prime prove di utilizzo del simulatore scelto, sono state effettuate su un portatile HP con processore **QuadCore Intel i5** a **2.4 Ghz** (con tre core a 800 Mhz ed uno a 1600Mhz, secondo quanto recita il file '/proc/cpuinfo'), memoria RAM **DDR3 6 GB** con sistema operativo **Windows 7 Home Edition**. Questa configurazione implicava l'utilizzo di un virtualizzatore (VirtualBox, nello specifico) su cui far eseguire un versione di UBUNTU 12.04 a 64 bit che, a sua volta, faceva eseguire il simulatore e la relativa JVM Oracle per ARM.

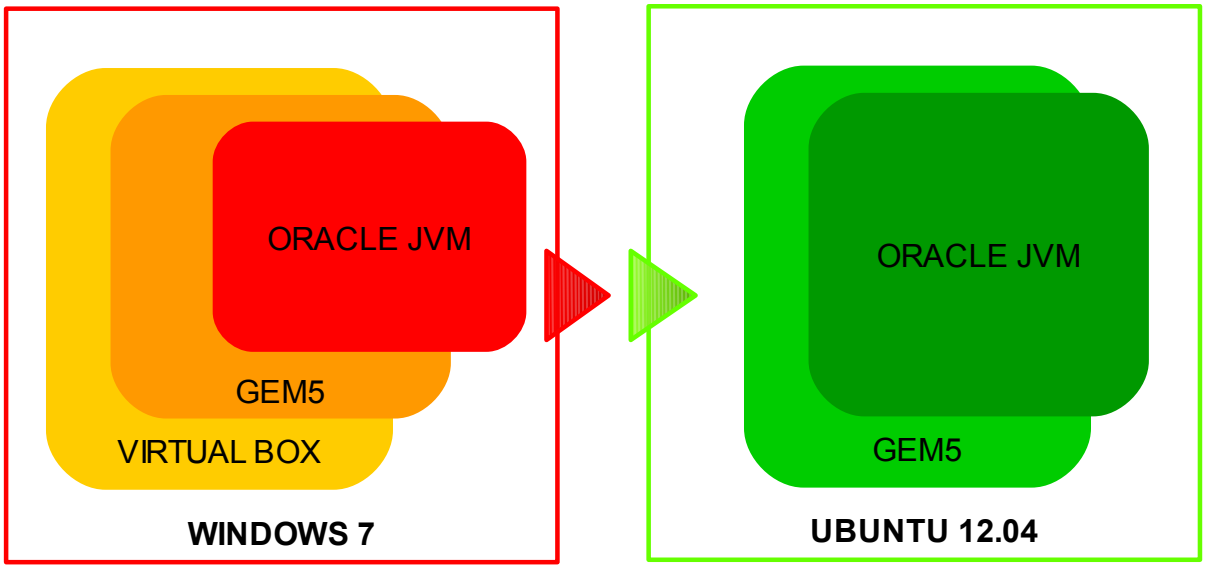


FIG 5.2: Scelta dei livelli di virtualizzazione

Questi 3 livelli di virtualizzazione (Virtualbox > Gem5 > Oracle JVM) oltre alla probabile non adeguata potenza del calcolatore richiedevano tempi di utilizzo veramente lunghi nonché un riscaldamento della macchina non indifferente. Anche eliminando uno dei 3 livelli, ovvero installando UBUNTU 12.04 sul disco rigido del portatile, evitando quindi l'utilizzo di VirtualBox, i tempi non sono risultati accettabili.

Per questo motivo, si è scelto di utilizzare, come sistema HOST del simulatore, uno dei server messi a disposizione dal DEI del Politecnico di Milano.

Il server in questione presenta invece le seguenti specifiche tecniche: **8 processori Intel Xeon CPU** a **2.0 Ghz** e **16GB** di **RAM** DDR3. Anch'esso con una versione di **UBUNTU 12.04 LTS** a **64 bit**. La comunicazione con il server è avvenuta attraverso protocollo SSH e grazie all'utilissimo tool: PuTTy(vedi appendice A paragrafo 4.4)

5.2 Simulazioni, risultati e valutazioni

L'evoluzione dello studio ha visto una lunga e onerosa fase di setup a causa anche dell'utilizzo di diversi software open-source privi di documentazione ed, in parte, non perfettamente adatti agli scopi del progetto. Dopo gli stadi di scelta e setup degli strumenti, accennati negli paragrafi e capitoli precedenti, si è dovuto modificare il simulatore, come succintamente evidenziato dal prossimo paragrafo, ed infine attuare le simulazioni per poter acquisire abbastanza dati da analizzare in merito agli scopi preposti.

5.2.1 Adattamento del simulatore

Dalla ricerca di mercato del paragrafo 4.2 si è evidenziata la necessità di definire architetture con un eventuale 3° livello di cache. Si è dovuto quindi modificare il simulatore andando a prevedere effettivamente un nuovo tipo di oggetto cache (che ereditasse, come gli altri, dalla classe MemObject) richiamabile in fase di definizione dell'architettura di simulazione. Ovviamente è stato anche necessario ridefinire il flusso logico di evoluzione della simulazione nonché le regole di definizione dell'architettura in termini di struttura d'interconnessione dei vari componenti di memoria, dei bus e della cpu. In particolare sono state fatte modifiche ai file python:

- "**ConfigCache.py**": che, richiamato dal file "guida" di simulazione, si occupa di definire gli oggetti cache, con eventuali sovrascritture dei valori di default parametrali, e l'interconessione delle stesse, in base all'architettura che si vuole definire, sfruttando determinati Bus e Porte preventivamente creati;

- "**Caches.py**": che riporta la definizione di classe dei diversi tipi di cache utilizzati dal precedente script;

Nel primo dei due è stata inserita la parte seguente di codice:

```
[...]

if options.num_l3caches == 1:
    system.l3 = L3Cache(size = options.l3_size, assoc =
                                options.l3_assoc,

block_size=options.cacheline_size)
    system.tol3bus = CoherentBus()
    system.l3.cpu_side = system.tol3bus.master
    system.l3.mem_side = system.membus.slave
    system.l2.mem_side = system.tol3bus.slave
else:
    system.l2.mem_side = system.membus.slave

[...]
```

che istanzia un oggetto cache di terzo livello e lo connette in cascata a quello di secondo livello se, entrambi, vengono richiesti dal file guida di configurazione e simulazione. Viceversa sarà componente cache di secondo livello a dover essere collegato al bus della memoria RAM.

Nel secondo è stato definita la classe di riferimento dell'oggetto cache di terzo livello istanziato nel “ConfigCache.py”:

```
[...]

class L3Cache(BaseCache):
     assoc = 8
     block_size = 64
     latency = '4ns'
     mshrs = 20
     tgts_per_mshr = 12


[...]
```

In questo stesso contesto sono stati adattati i valori di latenza delle cache alle esigenze, riferendosi alla ricerca di mercato del paragrafo 4.2. Notare che sarebbe stato anche possibile modificare le dimensioni dei componenti ma questo compito è stato affidato al file "guida" di simulazione che è stato modificato a dovere per permettergli di valutare, in base agli "switch" assegnati da linea di comando in fase di avvio della simulazione, quali livelli di cache definire e con che dimensione. A tal proposito sono state definite le seguenti istruzioni nel file di guida della simulazione:

```
    [...]

    if options.caches or options.l2cache
                    or (options.num_l3cache == 1):
        test_sys.iocache = IOCache(
addr_ranges=[test_sys.physmem.range])
        test_sys.iocache.cpu_side = test_sys.iobus.master
        test_sys.iocache.mem_side = test_sys.membus.slave
    else:

    [...]

    CacheConfig.config_cache(options, test_sys)

    [...]
```

A questo punto rimane da valutare la modifica effettuata per permette la variazione della latenza associata alla Memoria RAM. In questo caso è stato necessario andare a valutare la parte del codice sorgente scritta in C++. Da questa valutazione, allo stato attuale di implementazione del simulatore, si è appurato che l'oggetto associato alla memoria è istanziato con classe di riferimento "SimpleMemory" (che eredita dalla classe "AbstractMemory"). Attualmente è in corso una fase di sviluppo di una nuova versione di classe, per la memoria RAM, che permetta di definire, tramite il file guida, molti parametri utili. La classe "SimpleMemory" prevedeva invece un valore di latenza fisso e non modificabile, la modifica apportata per poterlo variare ha visto aggiungere un elemento ai parametri di dichiariazione della classe. Questo parametro funge da variabile per passare un valore scelto dall'utilizzatore del simulatore. Nella parte di codice Python è stato poi

modificato il file: “FSconfig.py”, che si occupa di istanziare il tipo di sistema in base all'architettura scelta:

```
[...]
def makeArmSystem(mem_mode, machine_type, mdesc = None,
                        bare_metal=False):
     assert machine_type
[...]
   self.physmem = SimpleMemory(latency='30ns', range =
                  AddrRange(self.realview.mem_start_addr,
                            size = mdesc.mem()),
                            conf_table_reported = True)
[...]
```

Il codice riportato sopra evidenzia le modifiche effettuate, ovvero l'aggiunta di un parametro in fase di chiamata del costruttore di classe SimpleMemory. Sarà poi il simulatore, a run-time, a “mappare” lo spazio di memoria dell'oggetto python in quello dell'oggetto c++ tramite il tool SWIG (vedi appendice A paragrafo A.3).

Con riferimento alle possibilità messe a disposizione, nel sul utilizzo, da GEM5 (vedi paragrafo x capitolo 4) inizialmente si è cercato di capire come sfruttarle al meglio cercando di ottenere dei valori delle statistiche simulazione col minor numero di overhead. L'utilizzo di una sola simulazione e di diversi checkpoint con l'evento di reset delle statistiche post ad un'istante precedente rispetto all'avvio della JVM e del benchmark, nonché la chisusura subito successiva della simulazione, è stata la prima strada percorsa. Le difficoltà però, in questo metodo, di eliminazione dell'overhead dato dall'avvio della JVM e delle tempistiche di inserimento da terminale dei comandi, piccole ma comunque esistenti, ha suggerito l'utilizzo dei file “.rcs” in cui andare ad inserire le operazioni che il sistema ha dovuto eseguire per poi concludere da solo la simulazione.

5.2.2 Simulazioni e commento dei risultati

La struttura di questo paragrafo procederà seguendo i 4 punti esposti ad inizio capitolo e, per ognuno, verrà ripetuta e valutata l'analisi per le 4 funzioni da usare come benchmark. I successivi paragrafi saranno quindi rispettivamente riferiti a:

- Mantenimento fisso dei valori di latenze e dimensioni delle memorie e della velocità del processore, facendo **variare il numero dei livelli di cache** da 1 a 3;

- Mantenimento fisso dei valori di latenze e livelli delle memorie e della velocità del processore, facendo **variare le dimensioni delle cache;**

- Mantenimento fisso dei valori relativi a dimensioni, livelli e latenze delle memorie, facendo invece **variare la frequenza del processore**;

- Mantenimento fisso di velocità del processore, dimensioni e livelli di memorie, facendo **variare le latenze delle memorie**

è bene notare che i parametri scelti nelle diverse simulazioni sono coerenti con quelli della ricerca di mercato effettuata (vedi paragrafo 4.3).
Come evidenziato nel capitolo tre (paragrafo 3.1.3), si può apprezzare un miglioramento dell'efficienza energetica del software, grazie alla memoizzazione, se il numero di ripetizioni, per singola esecuzione dei benchmark scelti, è ad un certo livello. Dato che le tempistiche di simulazione con quei valori spaziavano dalle 192 alle 480 ore (per singola simulazione) si è scelto di diminuirli di almeno un ordine di grandezza. Questa scelta non rende inefficaci le analisi proposte per tre motivi:

- I quattro punti relativi alle valutazioni da effettuare hanno come obiettivo l'analisi dell'impatto delle memorie sulla memoizzazione;

- Sul calcolo dei tempi di esecuzione, rispetto alla diminuzione delle ripetizioni, risulta una perdita di accuratezza di circa il 10% considerata accettabile dato che, in valore assoluto, si parla di differenze dell'ordine dei secondi ;

- Tutti i dati raccolti sono stati decurtati, ove necessario, dell'overhead dato dall'avvio della macchina virtuale che si occupa di eseguire i benchmark (overhead per altro <0.1% del tempo di esecuzione totale);

5.2.2.1 Variazione numero dei livelli di cache

Per capire l'impatto della variazione di livelli gerarchici delle memorie si sono effettuati tre esperimenti, ripetuti per le diverse funzioni economiche, in cui far diminuire progressivamente il numero di livelli di cache da 3 ad 1. Gli altri parametri architetturali, evidenziati in tabellla 5.1 sono stati mantenuti costanti.

Exp. n°	**Cache L1***		**Cache L2**		**Cache L3**		**RAM**	**CPU**
	size(KB)	Latency(ns)	size(KB)	Latency(ns)	size(MB)	Latency(ns)	Latency(ns)	Speed(Ghz)
1	32	1	256	10	8	20	30	4
2	32	1	256	10	-	-	30	4
3	32	1	-	-	-	-	30	4

*al primo livello le cache sono due da 32KB distinte per Istruzioni e Dati

TAB.5.1: variazione parametri architetturali degli esperimenti relativi alla variazione del numero dei livelli gerarchici di cache

Partendo dal benchmark "**Implied Volatility**", per cui le ripetizioni effettuate sono state di 100 mila per singola esecuzione, fra i vari risultati si sono ottenuti i seguenti tempi di esecuzione del sistema simulato (stimati da gem5):

TEXEC_1=93.89 secondi
TEXEC_2=94.34 secondi
TEXEC_3=91.31 secondi

ove TEXEC_X è il tempo di esecuzione reale, stimato dal sistema simulato, per la simulazione numero X.
Si può vedere come effettivamente non ci sia una differenza notevole nelle 3 esecuzioni. Curiosa la relazione d'ordine dei tempi di esecuzione che è la seguente:

TEXEC_3 < TEXEC_1 < TEXEC_2

la spiegazione dipende dal fatto che la somma delle latenze dei 3 livelli di cache è molto simile alla latenza assegnata alla memoria RAM e che il numero di richieste soddisfatte dalla cache di primo livello è molto superiore rispetto alle richieste effettuate sugli altri due livelli.
Investigando più a fondo, fra le varie statistiche restituite dal simulatore, si evidenziano altri dati riassunti nella tabella 5.2:

Exp.	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
n°	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~24.89*10^9	0.1184%	~109*10^6	23.81%	~26*10^6	7.22%
2	~24.89*10^9	0.1143%	~139*10^6	18.72%	-	-
3	~24.89*10^9	0.1118%	-	-	-	-

TAB.5.2: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti relativi alla variazione dei livelli gerarchici di cache per il benchmark "Implied volatility"

Essendo gli input dati in ingresso uguali per tutte e 3 le simulazioni, effettivamente c'è coerenza fra accessi totali in lettura e Miss Rate nella cache di 1° livello.
Da questi dati si può effettivamente notare come quasi tutte le richieste alle cache vengano soddisfatte da quella di primo livello e che le richieste effettuate sugli altri due livelli sono in numero nettamente inferiore rispetto alle richieste totali. La relazione d'ordine sui tempi di esecuzione deriva appunto dai tempi di latenza che, per come sono stati definiti, implicano un peggioramento dei tempi di risposta quando si hanno livelli di cache superiori al primo. Questo fatto non va a intaccare la valutazione che ci si era preposti di fare, ovvero quella sull'impatto della variazione dei livelli di cache, data l'enorme differenza di richieste soddisfatte fra il primo e gli altri livelli di cache. Non c'è un impatto significativo.

Passiamo adesso alla valutazione del benchmarck "**Black Scholes**". Per questo benchmark, il numero di ripetizioni effettuate per singola esecuzione è di 40 milioni. Valutiamo i tempi di esecuzione risultanti, che sono:

TEXEC_1=245.68 secondi

TEXEC_2=242.20 secondi

TEXEC_3=237.68 secondi

in questo caso si possono apprezzare differenze, non troppo significative, nelle tre esecuzioni. Se si vanno a suddividere i singoli tempi di esecuzione sopra riportati in due parti, quella effettivamente dovuta alla computazione da parte del processore e quella dovuta alle latenze delle memorie e dei componenti di connessione, si individuano tempi di computazione praticamente uguali e pari a circa 45,59 secondi (ci sono differenze dalla terza cifra decimale) mentre i tempi dovuti ai ritardi delle memorie sono:

TEXEC1_LATENZA=200.10 secondi;

TEXEC2_LATENZA=196.62 secondi;

TEXEC3_LATENZA=192.09 secondi;

La differenza sta quindi in questa parte del tempo di esecuzione totale. Curiosa anche la relazione d'ordine del tipo:

TEXEC3_LATENZA < TEXEC2_LATENZA < TEXEC1_LATENZA

Per lo stesso discorso di prima, la spiegazione potrebbe dipendere dal fatto che la somma delle latenze dei 3 livelli di cache è molto simile alla latenza assegnata alla memoria RAM, con le conseguenze precedentemente individuate. Investigando più a fondo, fra le varie statistiche restituite dal simulatore, si evidenziano i parametri riassunti nella tabella 5.3.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~35.85*10^9	0.7286%	~2.56*10^9	4.53%	~116*10^6	14.58%
2	~35.85*10^9	0.6816%	~2.81*10^9	4.16%	-	-
3	~35.85*10^9	0.5389%	-	-	-	-

TAB.5.3: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti relativi alla variazione dei livelli gerarchici di cache per il benchmark “black scholes”

Dalla tabella si evince la coerenza nei dati relativi alla cache di primo livello nei tre esperimenti. I tempi di esecuzione, in relazione d'ordine di cui sopra, dipendono quindi molto probabilmente dagli input dati in ingresso nel simulatore. Comunque la variazione è minima e quindi, anche in questo caso, non si evidenziano variazioni significative.

Per il benchmark "**Serialization**" sono state effettuate 100 mila ripetizioni per singola simulazione e i dati di relativi ai tempi di esecuzione sono i seguenti:

TEXEC_1=241.11 secondi
TEXEC_2=241.27 secondi
TEXEC_3=241.49 secondi

Come nei casi precedenti non si hanno variazioni significative sui tempi di esecuzione. In questo caso la relazione d'ordine rispetta quello che ci si potrebbe aspettare ed è infatti:

TEXEC_1 < TEXEC_2 < TEXEC_3

Non sembra necessario una valutazione più approfondita ma, anche per continuità, fra le varie statistiche restituite dal simulatore si evidenziano i parametri riassunti nella tabella 5.4:

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~66.17*10^9	0.0488%	~1.20*10^9	2.13%	~25.63*10^6	6.68%
2	~66.18*10^9	0.0440%	~1.15*10^9	2.44%	-	-
3	~66.17*10^9	0.0455%	-	-	-	-

TAB.5.4: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache, degli esperimenti relativi alla variazione dei livelli gerarchici di cache, per il benchmark "Serialization"

Da cui si evince la coerenza dei valori per la cache di primo livello e di nuovo si nota come questa risponda positivamente a quasi tutte le richieste fatte, passando ai livelli inferiori un numero molto minore di richieste. Si può affermare che non ci sono variazioni nette anche per questo benchmark.

Per l'ultimo dei quattro benchmark: “**XIRR**”, per cui sono state effettuate 500 mila ripetizioni per singola esecuzione, il simulatore ha riportato tempi di esecuzione pari a:

TEXEC_1=811.29 secondi
TEXEC_2=812.63 secondi
TEXEC_3=813.21 secondi

Anche per quest'ultimo benchmark non si notano variazioni significative sui tempi di esecuzione con diversi livelli di cache. Si ha inoltre la stessa relaziona d'ordine, sui tempi d'esecuzione, ottenuta col precedente benchmark:

TEXEC_1< TEXEC_2< TEXEC_3

per continuità, sono riportati in tabella 5.5 i dati su latenze e accessi totali alle memorie:

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~196.04*10^9	0.1747%	~1.53*10^9	14.67%	~225*10^6	10.38%
2	~196.05*10^9	0.1406%	~1.27*10^9	18.17%	-	-
3	~196.04*10^9	0.1706%	-	-	-	-

TAB.5.5: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti relativi alla variazione dei livelli gerarchici di cache, per il benchmark “XIRR”

Come ci si poteva aspettare, anche in questo caso c'è coerenza fra i dati relativi alla cache dati di primo livello. Nuovamente, il totale delle ricerche dato in cache che non hanno risposta positiva, che quindi passano alla memoria RAM, è di ordine molto inferiore rispetto alle richieste soddisfatte dalla cache uno. In base al numero di ripetizioni dell'esperimento ed alla dimensione degli input richiesti dal benchmark XIRR, si è potuto appurare che le dimensioni cache non sono sufficienti per contenere i dati memoizzati riutilizzabili dalla funzione. Questo implica che praticamente tutte le ricerche di dati memoizzati devono necessariamente essere trovati in memoria RAM.
Concludendo, si è quindi visto come per ogni benchmark non ci sia stata alcuna significativa

variazione, sui tempi di esecuzione, dovuta al cambiamento del numero dei livelli di cache nella gerarchia. Questa valutazione è riferita ai valori delle cache attualmente sul mercato dei server, quindi è possibile affermare che, allo stato attuale delle offerte, non c'è impatto sulla tecnica di memoizzazione variando il parametro fin qui considerato. Rispetto al problema di cui sopra si andrà a valutare più a fondo (paragrafo 5.2.2.4), fra le analisi dell'impatto dovuto al cambiamento delle latenze delle memorie, quella relativa alla RAM.

5.2.2.2 Variazione dimensioni delle cache

Sempre utilizzando i quattro benchmark dell'analisi precedente, in questo caso gli esperimenti effettuati sono quelli riassunti nella tabella 5.6.

Exp. n°	**Cache L1**		**Cache L2**		**Cache L3**		**RAM**	**CPU**
	size(KB)	Latency(ns)	size(KB)	Latency(ns)	size(MB)	Latency(ns)	Latency(ns)	Speed(Ghz)
1	32	1	256	10	**8**	20	30	4
2	32	1	256	10	**4**	20	30	4
3	32	1	256	10	**2**	20	30	4
4	32	1	256	10	**512**	20	30	4
5	32	1	256	10	**1024**	20	30	4
6	**16**	1	256	10	**8**	20	30	4
7	**16**	1	256	10	**4**	20	30	4
8	**16**	1	256	10	**2**	20	30	4

TAB.5.6: variazione parametri architetturali degli esperimenti relativi alla variazione delle dimensioni delle cache

Come si può apprezzare, si è scelto di fissare tutti i parametri architetturali facendo variare dapprima la dimensione della cache di livello tre e, successivamente, quella di livello uno. Il range di variazione è quello evidenziato nel paragrafo 4.3 e relativo alla ricerca di mercato effettuata. Le valutazioni dei casi 4 e 5 sono state fatte perchè, come si vedrà proseguendo, non sono state trovate differenze significative variando la dimensione della cache di terzo livello nel range scelto.

Il numero di ripetizioni per singola esecuzione di ogni benchmark sono le stesse della precedente analisi.

Per “**Implied volatility**” si sono ottenuti i tempi di esecuzione evidenziati in figura 52.

Exp n°	Tempo esecuzione
1	93,89 s
2	93,96 s
3	94,00 s
4	93,81s
5	93,81 s
6	102,37 s
7	102,93 s
8	102,48. s

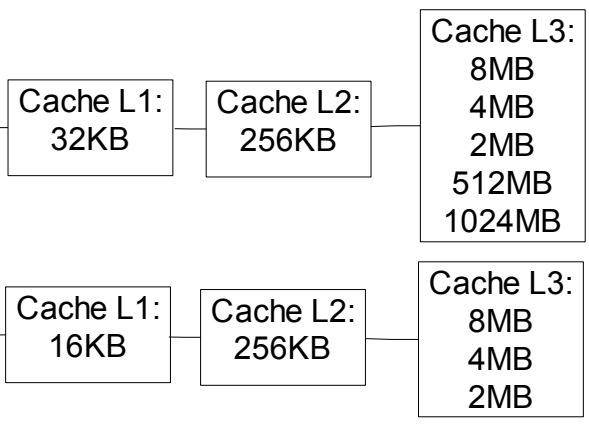


FIG.5.3: Tempi di esecuzione per gli esperimenti relativi alla variazione della dimensione della cache per il benchmark “Implied volatility”

Dai primi tre dati si evince come non ci siano variazioni sostanziali, nei tempi di esecuzione, facendo variare la dimensione della cache di terzo livello nel range assegnato. Aldilà del fatto che il tempo di esecuzione nel terzo caso risulti leggermente maggiore rispetto agli altri due, questo risultato è un po' fuorviante. Cercando di analizzare più a fondo i dati statistici restituiti dal simulatore, si evidenziano quelli in tabella 5.7.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~24.89*10^9	0.1184%	~109*10^6	23.81%	~26.0*10^6	7.22%
2	~24.90*10^9	0.1134%	~108*10^6	24.02%	~25.9*10^6	10.01%
3	~24.89*10^9	0.1187%	~109*10^6	23.81%	~26.0*10^6	15.83%

TAB.5.7: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 1,2 e 3, relativi alla variazione delle dimensioni delle cache, per il benchmark "Implied volatility"

Come ci si potrebbe aspettare, i dati per le cache dei primi due livelli e quelli di accesso in lettura per la cache di livello 2 e 3 sono coerenti (e praticamente uguali). Inoltre il miss rate della cache di livello 3 è crescente rispetto alla diminuzione della dimensione della stessa. In particolare, al raddoppiamento della cache da 2MB a 4MB si ha una diminuzione del miss rate di circa il 37%, e nel raddoppio ulteriore da 4MB a 8MB il miss rate diminuisce di circa il 28%.
I tempi di esecuzione rimangono praticamente uguali nei tre casi perchè la quantità di accessi fatti alla memoria RAM, a causa dei miss nei 3 livelli di cache, rimangono molto bassi rispetto alle richieste positive fatte sulla cache di primo livello. Infatti i valori del miss rate sul terzo livello, benché crescenti sui tre casi, sono comunque su un numero di accessi di ben 3 ordini di grandezza inferiore rispetto alle richieste, con esito positivo, fatte sulla cache di primo livello. Per evidenziare questo risultato e rafforzare la mia valutazione, sono stati effettuati i test 4 e 5 in cui, se l'analisi appena fatta fosse veritiera, si troverebbero dei miss rate sul terzo livello di cache bassissimi (quasi nulli) ma i tempi di esecuzione degli esperimenti praticamente invariati rispetto ai precedenti. I tempi di esecuzione sono effettivamente invarianti (come si vede dalla figura 5.3).

Per quanto riguarda gli accessi in lettura totali ed i miss rate (riportati in tabella 5.8) si nota che sia per i parametri dei primi due livelli che per le letture totali sul terzo livello, non ci sono modifiche. Cambia invece il miss rate del terzo livello di cache che, coerentemente con l'analisi precedente, scende quasi a zero ma questo non modifica significativamente il tempo di esecuzione.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
4	~24.90*10^9	0.1188%	~109*10^6	23.67%	~25.9*10^6	0.5335%
5	~24.90*10^9	0.1188%	~109*10^6	23.67%	~25.9*10^6	0.5018%

TAB.5.8: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 4 e 5, relativi alla variazione delle dimensioni delle cache, per il benchmark "Implied volatility"

Si può quindi affermare che, con questo benchmark, non ci sono variazioni di tempistica dovuti alla modifica delle dimensioni del terzo livello di cache, rispetto a valori dei componenti attualmente sul mercato, tenendo fissi gli altri parametri.

L'analisi successiva riguarda la variazione invece della dimensione della cache di primo livello per poter confrontare i risultati con quelli ottenuti variando la cache di terzo livello. I tempi di esecuzione sono quelli individuati nella figura 5.3 e, come si può apprezzare, ci sono state variazioni, seppur non enormi, ottenute dalla diminuzione di dimensione della cache L1 da 32KB a 16KB. Questo dipende molto dal fatto che, come si faceva notare precedentemente, il numero di accessi alle cache sono quasi tutti soddisfatti da quella di primo livello e l'ordine di grandezza degli accessi alle altre cache è molto inferiore rispetto a quello della cache L1. Quindi si ha ovviamente un aumento più evidente del tempo di esecuzione dovuta alla diminuzione delle dimensione di quest'ultima. I risultati relativi a numero di accessi e miss rate sono quelli riportati in tabella 5.9.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
6	~24.89*10^9	0.5121%	~499*10^6	5.28%	~26.4*10^6	7.15%
7	~24.90*10^9	0.5135%	~501*10^6	5.28%	~26.5*10^6	10.10%
8	~24.89*10^9	0.5126%	~500*10^6	5.28%	~26.4*10^6	15.85%

TAB.5.9: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 6,7 e 8 relativi alla variazione delle dimensioni delle cache, per il benchmark "Implied volatility"

I risultati sono coerenti a quel che ci si potrebbe aspettare, infatti si ha un aumento del miss rate sulla cache di primo livello con un relativo "scaricamento" delle richieste sugli altri due

livelli. Inoltre, sul livello 3, il rapporto di diminuzione del miss rate rispetto all'aumento della dimensione di cache rimane quello precedente. Non si hanno variazioni significative sul tempo di esecuzione perchè le richieste non soddisfatte dal livello uno vengono soddisfatte al livello 2 e sono in rapporto 5 a 1000.
Si può quindi affermare che anche la variazione delle dimensioni della cache di primo e terzo livello, seguendo la componentistica attualmente offerta sul mercato, non ha impatti significativi sulla memoizzazione.

Valutiamo adesso il benchmark "**Black Scholes**". Anche in questo caso i tempi di esecuzione, per gli esperimenti dall'1 al 3, sono praticamente uguali e pari a circa 246 secondi, mentre per gli esperimenti 4 e 5 si hanno tempi uguali e pari a circa 236.4 secondi. Per le simulazioni 6, 7 e 8 si hanno invece tempi di esecuzione uguali e pari a circa 295.5.secondi.
Con la stessa logica di prosecuzione, si analizzano i dati riportati in tabella 5.10

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~35.85*10^9	0.7286%	~2.56*10^9	4.53%	~116*10^6	14.58%
2	~35.85*10^9	0.6510%	~2.55*10^9	4.50%	~115*10^6	19.56%
3	~35.85*10^9	0.7287%	~2.56*10^9	4.60%	~118*10^6	34.22%

TAB.5.10: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 1,2 e 3 relativi alla variazione delle dimensioni delle cache, per il benchmark "Black Scholes"

Da cui si evidenzia che i dati per le cache di primo livello e quelli di accesso in lettura per la cache di livello 2 e 3 sono coerenti (e praticamente uguali). Inoltre il miss rate della cache di livello 3 e crescente rispetto al raddoppiamento della cache. Da 2MB a 4MB si ha una diminuzione del miss rate di circa il 43%, e nel raddoppio ulteriore da 4MB a 8MB il miss rate diminuisce di circa il 25%.
I tempi di esecuzione rimangono praticamente uguali nei tre casi per le stesse motivazioni del caso precedente. Anche in questo caso sono stati effettuati i test 4 e 5 in cui, se l'analisi appena fatta fosse veritiera, si troverebbero dei miss rate sul terzo livello di cache bassissimi (quasi nulli) ma i tempi di esecuzione degli esperimenti praticamente invariati rispetto ai

precedenti. I tempi di esecuzione degli esperimenti 4 e 5, rispetto a quelli dei primi tre, hanno una variazione inferiore al 3.9%, quindi non abbastanza significativa. Comunque è una variazione percentuale maggiore rispetto a quella ottenibile, per lo stesso confronto, nel caso del benchmark precedente, infatti il rapporto fra richieste fatte alla cache di livello 3 e quelle alla cache di livello 1 maggiore, per questo benchmark rispetto al precedente.

Inoltre gli accessi in lettura alle cache ed i miss rate sono esposti in tabella 5.11.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
4	~35.85*10^9	0.7287%	~2.54*10^9	4.25%	~108*10^6	1.5394%
5	~35.85*10^9	0.7287%	~2.54*10^9	4.25%	~108*10^6	1.5149%

TAB.5.11: *dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 4 e 5 relativi alla variazione delle dimensioni delle cache, per il benchmark "Black Scholes"*

Sia per i parametri dei primi due livelli che per le letture totali sul terzo livello, non abbiamo modifiche. Cambia invece il miss rate del terzo livello di cache che, coerentemente con l'analisi precedente, scende quasi a zero ma questo non modifica significativamente il tempo di esecuzione.
Si può quindi affermare che, anche con questo benchmark, non ci sono variazioni di tempistica dovuti alla modifica delle dimensioni del terzo livello di cache, rispetto a valori dei componenti attualmente sul mercato, tenendo fissi gli altri parametri.

Rimangono ora da commentare i risultati dell'esecuzioni 6, 7 e 8. Come consueto, la tabella 5.12 riporta i valori di accessi in lettura e miss rate per questi esperimenti.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
6	~35.85*10^9	1.6108%	~4.87*10^9	2.33%	~113*10^6	15.08%
7	~35.85*10^9	1.6108%	~4.87*10^9	2.31%	~112*10^6	20.52%
8	~35.85*10^9	1.6109%	~4.89*10^9	2.33%	~114*10^6	35.71%

TAB.5.12: *dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 6, 7 e 8 relativi alla variazione delle dimensioni delle cache, per il benchmark "Black Scholes"*

Ovviamente le proporzioni di incremento del miss rate, al diminuire della dimensione della cache, rimangono quelle di prima. Inoltre si nota un valore più che raddoppiato del miss rate sulla cache di primo livello in tutti e 3 i casi, dovuto appunto al dimezzamento della dimensione.
La tempistica di esecuzione del benchmark varia questa volta di circa il 10% dato che aumenta consistentemente il miss rate sul primo livello di cache (che fino ad ora aveva soddisfatto richieste in numero molto superiore alle richieste passate ai successivi livelli). Come fatto però notare nel paragrafo precedente, praticamente tutti i dati di input del benchmark devono necessariamente essere ripescati in memoria RAM perchè le dimensioni delle cache non sono abbastanza grandi da contenerli. Questo implica che quasi tutte le richieste soddisfatte dalle cache sono riferibili al resto delle necessità di computazione. Anche in questo caso quindi non ci sono variazioni notevoli di sorta rispetto alla tecnica di memoizzazione.

Per il benchmark "**Serialization**", sono stati calcolati tempi di esecuzione uguali e pari a circa **241** secondi, per gli esperimenti dall'1 al 5, mentre pari a circa **280** secondi per gli altri 3 esperimenti.

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~66.17*10^9	0.0488%	~1.20*10^9	2.13%	~25.63*10^6	6.68%
2	~66.17*10^9	0.0569%	~1.20*10^9	2.13%	~25.58*10^6	8.75%
3	~66.17*10^9	0.0531%	~1.20*10^9	2.13%	~25.62*10^6	14.21%

TAB.5.13: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 1,2 e 3 relativi alla variazione delle dimensioni delle cache, per il benchmark "Serialization"

Con la stessa logica di prosecuzione, si analizzano i dati riportati in tabella 5.13 da cui si evidenzia, di nuovo, che i dati per le cache di primo livello e quelli di accesso in lettura per la cache di livello 2 e 3 sono coerenti (e praticamente uguali). Nel passaggio, della dimensione della cache di livello 3, da 2MB a 4MB si ha una diminuzione del miss rate di circa il 38%, e nel raddoppio ulteriore da 4MB a 8MB il miss rate diminuisce di circa il 24%.
Per motivazioni analoghe alle precedenti sono stati effettuati i test 4 e 5 in cui, per rafforzare

la valutazione sull'uguaglianza dei tempi di esecuzione dei vari esperimenti. Effettivamente, con cache di terzo livello enormi, i tempi di esecuzione rimangono invariati mentre i miss rate sono ~0 (tabella 5.14).

Exp.	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
n°	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
4	~66.17*10^9	0.0488%	~1.20*10^9	2.11%	~25.36*10^6	0.4946%
5	~66.17*10^9	0.0488%	~1.20*10^9	2.11%	~25.36*10^6	0.4550%

TAB.5.14: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 4 e 5 relativi alla variazione delle dimensioni delle cache, per il benchmark "Serialization"

Sia per i parametri dei primi due livelli che per le letture totali sul terzo livello, non abbiamo modifiche. Cambia invece il miss rate del terzo livello di cache che, coerentemente con l'analisi precedente, scende quasi a zero ma questo non modifica significativamente il tempo di esecuzione.
Si può quindi affermare che, anche con questo benchmark, non ci sono variazioni di tempistica dovuti alla modifica delle dimensioni del terzo livello di cache, rispetto a valori dei componenti attualmente sul mercato, tenendo fissi gli altri parametri.

Rimangono ora da commentare i risultati dell'esecuzioni 6, 7 e 8. Come consueto, la tabella 5.15 riporta i valori di accessi in lettura e miss rate per questi esperimenti.

Exp.	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
n°	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
6	~66.17*10^9	0.0826%	~1.20*10^9	1.31%	~25.94*10^6	6.61%
7	~66.17*10^9	0.0897%	~1.20*10^9	1.31%	~25.98*10^6	8.68%
8	~66.17*10^9	0.0827%	~1.20*10^9	1.31%	~25.93*10^6	14.26%

TAB.5.15: *dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 6, 7 e 8 relativi alla variazione delle dimensioni delle cache, per il benchmark "Serialization"*

Ovviamente le proporzioni di incremento del miss rate, al diminuire della dimensione della cache, rimangono quelle di prima. Inoltre si nota un valore più che raddoppiato del miss rate sulla cache di primo livello in tutti e 3 i casi, dovuto appunto al dimezzamento della dimensione cache di primo livello.
La tempistica di esecuzione del benchmark varia anche in questo caso ma di un 14%, rispetto ai casi con cache di primo livello pari a 32KB, per motivazioni analoghe al caso precedente. Anche in questo caso quindi non ci sono variazioni notevoli di sorta per la tecnica di memoizzazione.

Valutiamo adesso il benchmark "**XIRR**", per cui i tempi di esecuzione sono uguali e pari a circa **166,19** secondi, per gli esperimenti 1, 2 e 3, a circa **162,7** secondi per gli esperimenti 4 e 5 ed, infine, pari a circa **174** secondi per le restanti tre simulazioni.
Con la stessa logica di prosecuzione, si analizzano i dati riportati in tabella5.16

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1*	~39.65*10^9	0.2123%	~336*10^6	19.11%	~64.2*10^6	9.06%
2	~39.72*10^9	0.2179%	~336*10^6	19.19%	~64.6*10^6	12.62%
3	~39.72*10^9	0.2179%	~336*10^6	19.54%	~65.7*10^6	22.48%

*L'esperimento, come negli altri casi, è lo stesso numero 1 del paragrafo 5.2.2.1 per XIRR. Le esecuzioni, per questo benchmark, erano lì 500000, adesso sono 100000 per motivazioni già spiegate. A parte la perdita di accuratezza del tempo di esecuzione, valutata prima, il fattore 5 di divisione è rispettato perfettamente per tutti gli altri parametri. I dati sono stati normalizzati

TAB.5.16: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 1,2 e 3 relativi alla variazione delle dimensioni delle cache, per il benchmark "XIRR"

Nulla da dire sui primi due livelli, mentre sulla cache di terzo livello si possono apprezzare le variazioni sul miss rate dovute alla diminuzione della dimensione. In particolare, nel passaggio da 2 a 4 MB si apprezza una diminuzione del miss rate pari a circa il 44%, mentre da 4 a 8 MB è del 28% circa.
Per motivazioni analoghe alle precedenti sono stati effettuati i test 4 e 5, per rafforzare la valutazione sull'uguaglianza dei tempi di esecuzione dei vari esperimenti. Effettivamente, con cache di terzo livello enormi, i tempi di esecuzione sono invariati, mentre i miss rate sono quasi nulli (tabella 5.17).

Exp.	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
n°	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
4	~39.72*10^9	0.2038%	~309.3*10^6	19.59%	~60.60*10^6	0.8972%
5	~39.72*10^9	0.2038%	~309.3*10^9	19.59%	~60.60*10^6	0.8683%

TAB.5.17: *dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 4 e 5 relativi alla variazione delle dimensioni delle cache, per il benchmark "XIRR"*

Sia per i parametri dei primi due livelli che per le letture totali sul terzo livello, non abbiamo modifiche. Cambia invece il miss rate del terzo livello di cache che, coerentemente con l'analisi precedente, scende quasi a zero ma questo non modifica significativamente il tempo di esecuzione.
Si può quindi affermare che, anche con questo benchmark, non ci sono variazioni di tempistica dovuti alla modifica delle dimensioni del terzo livello di cache, rispetto a valori dei componenti attualmente sul mercato, tenendo fissi gli altri parametri.
Rimangono ora da commentare i risultati dell'esecuzioni 6, 7 e 8. Come consueto, la tabella 5.18 riporta i valori di accessi in lettura e miss rate per questi esperimenti.

Exp.	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
n°	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
6	~39.73*10^9	0.3177%	~696.7*10^6	9.02%	~62.77*10^6	9.52%
7	~39.72*10^9	0.3825%	~696.3*10^6	9.05%	~63.05*10^6	13.08%
8	~39.72*10^9	0.3824%	~696.8*10^6	9.16%	~63.09*10^6	23.56%

TAB.5.18: dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti 6, 7 e 8 relativi alla variazione delle dimensioni delle cache, per il benchmark "XIRR"

Ovviamente le proporzioni di incremento del miss rate, al diminuire della dimensione della cache, rimangono quelle di prima. Inoltre si nota un valore più che raddoppiato del miss rate sulla cache di primo livello in tutti e 3 i casi, dovuto appunto al dimezzamento della dimensione.
La tempistica di esecuzione del benchmark varia di appena un 4,5% in questo caso. Valore non sufficiente affinchè i tempi di esecuzione presentino variazioni degne di nota. Anche in questo caso quindi non ci sono impatti notevoli di sorta sulla tecnica di memoizzazione.

5.2.2.3 Variazione frequenza del processore

Come per le precedenti analisi, saranno utilizzati i 4 benchmark di cui finora si è usufruito. In questo caso si farà variare soltanto la velocità di computazione del processore. In particolare si avranno invariati, per tutti e 4 gli esperimenti, i valori della tabella 5.19.

	Cache L1	**Cache L2**	**Cache L3**	**RAM**
latency	1ns	10ns	20ns	30ns
size	32kB	256kB	8MB	1024MB

TAB.5.19: variazione parametri architetturali degli esperimenti relativi alla variazione della frequenza del processore

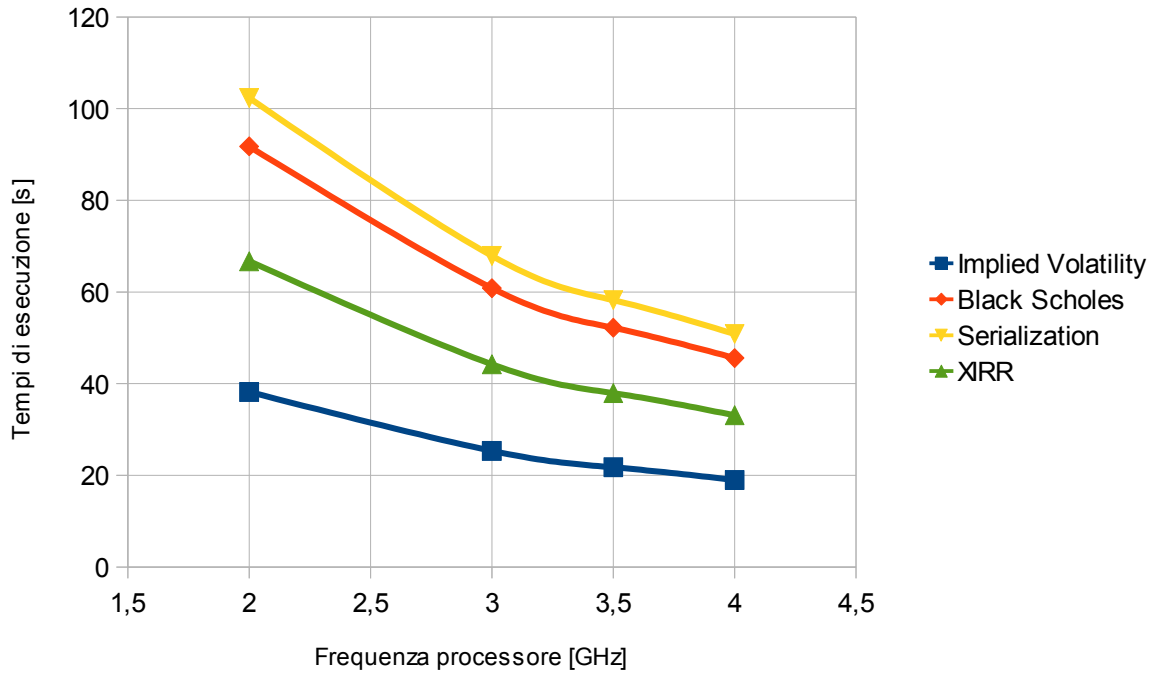
Mentre si farà variare, dal caso 1 al caso 4, la frequenza del processore rispettivamente a: **4GHz**, **3.5GHz**, **3GHz** e **2GHz.**

Exp. n°	**Implied Volatility**	**Black Scholes**	**Serialization**	**XIRR**
1	93,89 s	245,68 s	254,11 s	166,19 s
2	96,86 s	254,04 s	263,86 s	171,90 s
3	101,83 s	266,47 s	274,95 s	177,80 s
4	114,02 s	296,16 s	305,71 s	201,49 s

TAB. 5.20 Tempi di esecuzione per gli esperimenti relativi alla variazione della frequenza del processore per tutti i benchmark

I tempi di esecuzione ottenuti, riportati in tabella 5.20, sono quelli totali e cioè comprensivi del tempo di computazione e del tempo dovuto ai ritardi di soddisfacimento delle richieste fatte ai componenti di memoria. Facendo variare la frequenza del processore però, essendo uguali in numero gli input dati ai rispettivi benchmark, la parte variante non è quella dovuta ai ritardi che sono pari a circa: 75s per Implied Volatility, 203 secondi per Black Scholes, 205 secondi per Serialization e 134 secondi per XIRR, bensì quella dovuta alla computazione.

Il grafico 5.1 riporta l'interpolazione lineare dei risultati ottenuti per la parte di sola computazione dei tempi di esecuzione, nei quattro esperimenti, di tutti e quattro i benchmark.



GRAF. 5.1: Grafico dell'andamento dei tempi di esecuzione dei 4 benchmark per gli esperimenti relativi alla variazione della frequenza del processore.

I tempi di esecuzione quindi peggiorano notevolmente con la diminuzione della velocità del processore. Nell'ordine si nota un peggioramento più repentino per “Serialization”, seguito da “Black Scholes”, “Xirr” e “Implied Volatility”. Inoltre, all'aumentare della frequenza del processore, la differenza percentuale nei tempi di esecuzione, fra i vari benchmark, tende ad annullarsi. Questo porta a pensare che sopra un certo livello di velocità del processore non converrebbe più investire, se si sceglie di attuare la tecnica di memoizzazione nelle funzioni utilizzate.

Un confronto interessante sarebbe quello fra i tempi di esecuzione, con stessi parametri architetturali, delle funzioni originali (che avranno un andamento decrescente pressochè lineare) e di quelle memoizzate, alla ricerca di intersezioni per valutare un range (almeno minimo) entro cui conviene effettivamente l'utilizzo della tecnica di MEMOIZZAZIONE.

A tal proposito, avendo individuato, nel range di variazione della velocità della CPU appena usato, un enorme dislivello fra i tempi di esecuzione delle funzioni originali, rispetto a quello memoizzate, sono state effettuate altre simulazioni. Queste ultime, alla ricerca del valore di velocità del processore minimo da cui conviene utilizzare la tecnica di memoizzazione, è proseguita con una sequenza di dicotomica fra 0 e 2 Ghz. I particolare sono state effettuate delle simulazioni uguali, in termini di parametri architetturali relativi alle memorie, facendo però variare la velocità del processore a **500MHz**, **1GHZ, 1.5GHZ**. Gli esperimenti sono stati ripetuti solo per il benchmark XIRR, sia in versione memoizzata che in versione originale ed il numero di ripetizioni per singola simulazione è stato diminuito a 10 mila, per motivazioni già anticipate e riesposte nel prossimo paragrafo.
Come riportato dalla tabella 5.21, per valori di velocità del processore ben al di sotto di quelli attualmente disponibili sul mercato e “prossimi” a zero, conviene sempre attuare la tecnica di memoizzazione per il benchmark in esame.

Frequenza CPU [GHz]	**0,5**	**1**	**1,5**
Tempo esecuzione XIRR originale [s]	874,32	513,5	403,18
Tempo esecuzione XIRR memoizzato [s]	52,45	34,44	28,52

TAB.5.21: confronto tempi di esecuzione dei benchmark originali e di quelli con memoizzazione facendo variare la velocità del processore a 0.5 Ghz, 1 Ghz e 1.5 GHz

Le analisi alla ricerca di un higher bound entro cui conviene utilizzare la tecnica di memoizzazione non sono state effettuate in questo contesto per mancanza di tempo e perchè comunque, con i valori di velocità dei processori attualmente sul mercato, la tecnica di memoizzazione risulta sempre vantaggiosa.
I dati riportati in tabella 5.20 si traducono in un enorme risparmio energetico, secondo le motivazioni evidenziate nel capitolo 2.

5.2.2.4 Variazione latenze dei componenti di memoria

Questa ultima analisi cercherà di individuare eventuali variazioni sui tempi di esecuzione delle simulazioni, dei benchmark utilizzati finora, date dalla modifica delle latenze dei componenti di memoria delle architetture. In particolare sono stati mantenuti costanti i parametri di dimensionamento delle cache e della velocità di processamento della CPU variando i ritardi di risposta di cache e RAM. I parametri architetturali delle varie simulazioni sono riassunti nelle tabelle 5.22 e 5.23.

Exp. n°	**Cache L1**	**Cache L2**	**Cache L3**	**RAM**
	Latency(ns)	Latency(ns)	Latency(ns)	Latency(ns)
1	1	**10**	**20**	30
2	1	**2**	**4**	30
3	1	2	4	**15**
4	**0,5**	**1**	**2**	15
5	0,5	1	2	**30**

TAB. 5.22: parametri architetturali degli esperimenti relativi alla variazione delle frequenze dei componenti di memoria (parte1)

Exp. n°	**Cache L1**	**Cache L2**	**Cache L3**	**CPU**
	size(KB)	size(KB)	size(MB)	Speed(Ghz)
1,2,3,4,5	32	256	8	4

TAB. 5.23: parametri architetturali degli esperimenti relativi alla variazione delle frequenze dei componenti di memoria (parte2)

Dalle tabelle si individua a scelta di far variare inizialmente la latenza delle cache di 2° e 3° livello, rapportandole a quelle attuali del mercato, soprattutto per la cache di terzo livello per cui si stanno adottando memorie eDRAM che vantano tempi nominali di latenza inferiori ai 4ns. La riduzione fatta è di un fattore 5 per entrambi i livelli di cache sopra citati. Successivamente si è cercato di aumentare la differenza fra la somma delle latenze delle cache e il valore di latenza della RAM. Per fare ciò, si sono mantenuti i nuovi valori di latenza delle cache di secondo e terzo livello e si è posto a 15ns la latenza della memoria RAM, attestando a più del doppio la differenza di cui sopra. Si è infine proseguito all'ulteriore

aumento di questa differenza, prima dimezzando i valori delle latenze delle cache e, successivamente, raddoppiando quella della memoria RAM.
Come per le precedenti analisi, gli esperimenti sono stati ripetuti per tutti e 4 i benchmark.

Partendo dalla valutazione di "**Implied Volatility**", sono stati individuati i valori di tempo di esecuzione riportati in tabella 5.24.

Exp n°	Tempo di esecuzione
1*	10,32 s
2	10,28 s
3	10.22 s
4	8,24 s
5	8,22 s

*È la stessa simulazione utilizzata nell'analisi 5.2.2.2, in cui il numero di ripetizioni era fissato a 100000, adesso pari a 10000. I valori sono normalizzati

TAB. 5.24: Tempi di esecuzione per gli esperimenti relativi alla variazione delle latenze dei componenti di memoria per ili benchmark "Implied volatility"

Individuata e considerata accettabile la perdita di accuratezza alla diminuzione del numero di ripetizioni del benchmark per singola esecuzione, si è avuta la necessità di ridurre i tempi di esecuzione del simulatore. Volendo passare a tempi medi più bassi, per implied volatility si è scesi ad un numero di ripetizioni pari a 10000.
Considerando una perdita di accuratezza media di circa il 10%, si può valutare come i tempi di esecuzione siano rimasti praticamente invariati almeno nei primi 3 casi. Valutando adesso il numero accessi alle diverse cache ed i miss rate (ripotati in tabella 5.25) si evidenzia come,

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~2.70*10^9	0.5138%	~36.52*10^6	34.51%	~12.60*10^6	12.71%
2	~2.69*10^9	0.4980%	~37.22*10^6	33.79%	~12.58*10^6	12.67%
3	~2.69*10^9	0.5080%	~36.72*10^6	34.26%	~12.58*10^6	12.67%
4	~2.70*10^9	0.5292%	~37.23*10^6	33.86%	~12.61*10^6	12.70%
5	~2.69*10^9	0.5162%	~39.47*10^6	31.89%	~12.59*10^6	12.69%

TAB. 5.25: Dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti relativi alla variazione delle latenze dei componenti di memoria per il benchmark “Implied volatility”

coerentemente con i tempi di esecuzione, rimanendo uguali i parametri relativi alle dimensioni ed al numero gerarchico delle cache, abbiamo quasi uguaglianza fra i valori riportati in tabella 5.25. I tempi di esecuzione però risultano diminuire negli esperimenti 4 e 5, dove si è dimezzata la latenza della cache di primo livello, il che è giusto data l'importanza, in termini di numero, degli accessi con risposta positiva fatti a quel livello, rispetto agli altri. Come si è fatto notare già dal paragrafo 5.2.2.1, con molta probabilità, quasi tutti i valori di input necessari ai benchmark sono “sovradimensionati” rispetto alle dimensioni delle cache. Fermo restando che i risultati raggiunti finora sono corretti, sarebbe opportuno valutare più in profondità l'impatto della variazione di latenza della sola memoria RAM, facendo magari variare anche la velocità del processore nel range definito nel paragrafo precedente. Saranno adesso esposti brevemente i risultati delle precedenti 5 sperimentazioni relativi agli altri tre benchmark, successivamente si passerà alle valutazioni fatte facendo variare solo la latenza della RAM e la velocità del processore.

Per il benchmark “**Blach Scholes**”, sono stati individuati i valori di tempo di esecuzione riportati in tabella 5.26.

Exp n°	Tempo esecuzione
1*	21,65 s
2	21,13 s
3	21,68 s
4	17,66 s
5	17,15 s

*È la stessa simulazione utilizzata nell'analisi 5.2.2.2, in cui il numero di ripetizioni era fissato a 40000000, adesso pari a 4000000. I valori sono normalizzati.

TAB. 5.26: Tempi di esecuzione per gli esperimenti relativi alla variazione delle latenze dei componenti di memoria per ili benchmark “Black Scholes”

Anche in questo caso, volendo passare a tempi medi di esecuzione del simulatore più bassi, si è scesi ad un numero di ripetizioni pari a 4 milioni. Anche per questo benchmark i tempi di esecuzione nei primi tre casi risultano analoghi, dato che, come evidenziato già prima, le latenze delle cache di livello 2 e 3 e della RAM non hanno un grande impatto perchè quasi tutte le richieste effettuate ai componenti di memoria sono soddisfatte dal livello 1. Si riportano, per continuità, anche i valori del numero accessi alle diverse cache ed i miss rate (ripotati in tabella 5.27).

Exp. n°	**Cache L1 (DATI)**		**Cache L2**		**Cache L3**	
	Letture Totali	Miss Rate	Letture Totali	Miss Rate	Letture Totali	Miss Rate
1	~4.15*10^9	1.1138%	~301*10^6	9.30%	~27.99*10^6	12.71%
2	~4.19*10^9	1.1971%	~360*10^6	9.53%	~34.27*10^6	11.99%
3	~4.20*10^9	1.1768%	~331*10^6	10.15%	~33.59*10^6	12.23%
4	~4.20*10^9	1.1145%	~324*10^6	10.34%	~33.51*10^6	12.20%
5	~4.19*10^9	1.3211%	~366*10^6	9.12%	~33.4*10^6	12.31%

TAB. 5.27: Dati relativi a numero accessi in lettura e frequenza di miss per i tre livelli di cache degli esperimenti relativi alla variazione delle latenze dei componenti di memoria per il benchmark “Black Scholes”

Su questi dati, analoghe analisi e conclusioni possono essere tratte rispetto al benchmark precedente dato che c'è effettiva coerenza fra i diversi esperimenti in merito a quei parametri.

Per **Serialization** si sono ottenuti i tempi di esecuzione di tabella x. In questo caso il numero di ripetizioni della funzione per singola esecuzione è diminuito a 100 mila.

Exp n°	Tempo esecuzione
1*	26,51 s
2	26,57 s
3	26,52 s
4	21,33 s
5	21,34 s

TAB. 5.28: Tempi di esecuzione per gli esperimenti relativi alla variazione delle latenze dei componenti di memoria per ili benchmark "Serialization"

I tempi di esecuzione nei primi tre casi risultano analoghi, dato che, come evidenziato già prima, le latenze delle cache di livello 2 e 3 e della RAM non hanno un grande impatto perchè quasi tutte le richieste effettuate ai componenti di memoria sono soddisfatte dal livello 1. I tempi di esecuzione però risultano diminuire negli esperimenti 4 e 5, dove si è dimezzata la latenza della cache di primo livello. Anche in questo caso quindi non si individuano variazioni notevoli di sorta.

Per l'ultimo benchmark: **XIRR**, invece si sono ottenuti tempi di esecuzione simili e pari a circa **17,6** secondi, nei primi 3 esperimenti. Tempi che diminuiscono a circa **14,37** secondi per quelli successivi.

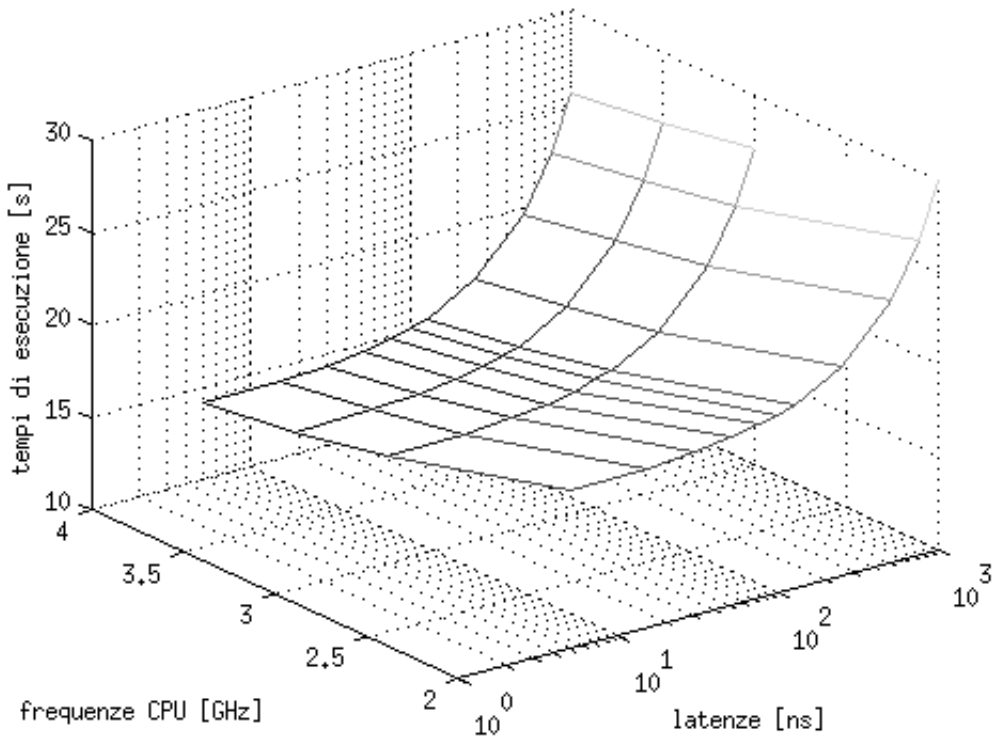
In conclusione, con questi valori e range di variazione delle latenze, non ci sono impatti notevoli sulla tecnica di memoizzazione.

Come anticipato prima, l'ultima analisi riguarda un più approfondito studio sull'impatto dato dalla variazione della latenza della memoria RAM incrociando le variazioni della stessa con la variazione della frequenza del processore simulato.
In particolare, si è esteso e frattagliato molto di più il range di variazione della latenza per il componente in esame e gli esperimenti effettuati sono stati quelli individuati dalla tabella 5.29 (TEMPI RISULTANTI ESPRESSI IN SECONDI).

GHz/ns	**5**	**15**	**30**	**50**	**75**	**100**	**125**	**250**	**500**	**750**	**1000**
4	14,13	14,2	14,37	14,6	14,88	15,19	15,45	16,88	19,69	22,59	25,51
3,5	14,89	15,02	15,15	15,4	15,69	16,01	16,24	17,69	20,53	23,4	26,24
3	15,84	15,94	16,11	16,31	16,63	16,96	17,16	18,64	21,52	24,34	27,18
2	18,64	18,64	18,86	19,11	19,44	19,62	19,96	21,35	24,24	27,03	29,95

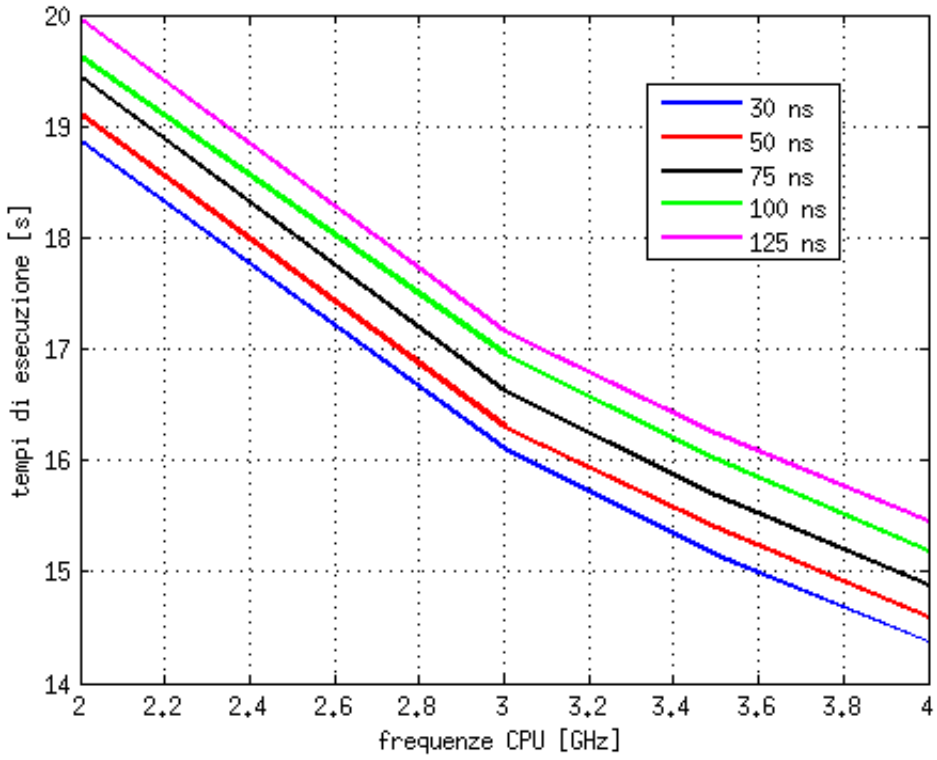
TAB. 5.29: Tempi di esecuzione per gli esperimenti relativi alla variazione della latenza di memoria RAM incrociati con la variazione delle frequenza dell processore, per ili benchmark “XIRR”

Alla variazione della latenza è stato fatto corrispondere anche una variazione della frequenza del processore, sempre nel range individuato dalla ricerca di mercato: **4GHz**, **3,5Ghz**, **3GHz**, **2GHz**. L'andamento crescente dei tempi di esecuzione richiesti sia ll'aumentare della latenza che all'aumenta della velocità del processore, è più apprezzabile sul grafico 5.2.



GRAF. 5.2: Tempi di esecuzione per gli esperimenti relativi alla variazione della latenza di memoria RAM incrociati con la variazione delle frequenza del processore, per il benchmark "XIRR"

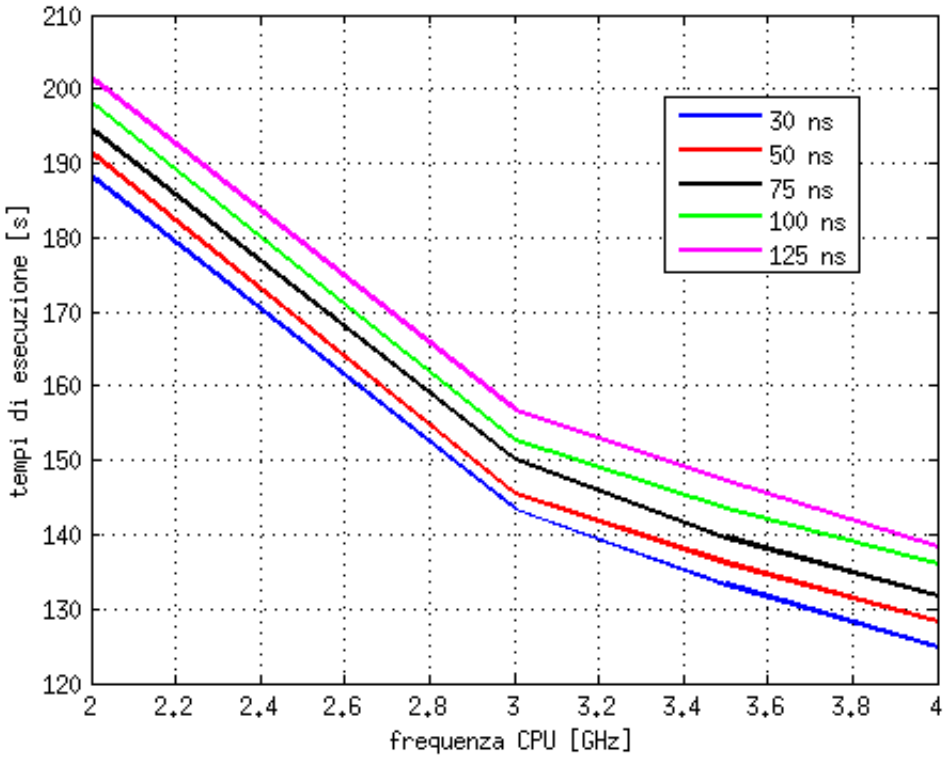
Dato il range e gli step di variazione della latenza scelti, la parte crescente dei tempi di esecuzione all'aumentare delle latenze diventa più ripida, più velocemente, da 100ns in avanti. Interessante è soprattutto la fascia di variazione che va dai 30 ns ai 125 ns, che mappano le condizioni realistiche del mercato, e per cui il grafico 5.3 mostra più in dettaglio l'andamento dei tempi di esecuzione al variare dei processori.



GRAF. 5.3: Tempi di esecuzione per un range significativo di variazione della latenza di memoria RAM al variare della frequenza del processore, per il benchmark "XIRR"

Com si può notare dal grafico, i tempi di esecuzione subiscono variazioni veramente poco notevoli (è in un range di 1 s). In particolare, la pendenza di discesa dei tempi, all'aumentare della frequenza del processore, tende a diminuire anche perchè ovviamente tenderà a 0 all'infinito.

Inoltre, il grafico 5.4, mostra le stesse esecuzioni però ripetute per il benchmark XIRR originale (non memoizzato).



GRAF. 5.4: Tempi di esecuzione per un range significativo di variazione della latenza di memoria RAM al variare della frequenza del processore, per il benchmark "XIRR ORIGINALE"

Dal confronto dei due è facile individuare la differenza media di circa un ordine di grandezza per i tempi di esecuzione, a discapito delle funzioni non memoizzate, soprattutto per valori di frequenze del processore più basse. Questo implica un enorme risparmio in termini si di tempo ma anche, e soprattutto, in termini di consumi energetici per tutte le valutazioni riportate nella tesi in merito ai concetti di green software e memoizzazione. È possibile quindi affermare che non ci siano enormi variazioni, o comunque significative, nell'utilizzo della tecnica di memoizzazione variando le latenze dei componenti di memoria. È altresì possibile affermare con certezza che, con gli attuali componenti presenti sul mercato, per un'architettura ARM, è sempre conveniente sfruttare la tecnica di memoizzazione per le funzioni computing intesive, in modo da raggiungere risparmi di potenza consumata e di tempi di esecuzioni utili all'uomo ed anche all'ambiente.

6. Conclusioni e sviluppi futuri

Nell'ambito dello sviluppo di software energeticamente efficiente sono molti gli sforzi fatti ed altrettanti quelli da fare, così come le possibili vie da percorrere. Questo progetto di tesi, che si colloca nell'ambito appena citato, si è posto l'obiettivo di valutare come l'utilizzo della tecnica di **memoizzazione**, secondo il modello sviluppato al DEI del Politecnico di Milano, possa essere affetto dalla variazione dei parametri dei componenti di memoria in un'architettura ARM. L'architettura scelta è molto importante data la pervasività che la stessa sta raggiungendo in molti settori informatici, da quello dei server ai dispotivi embedded.
Prendendo come base minima, per i range dei parametri da far variare, quelli riferibili alle offerte attuali sul mercato dei server fra i più grossi colossi di produzione e distribuzione degli stessi, la serie di esperimenti, sotto forma di simulazioni, ha permesso di affermare che:

1. NON ci sono effetti notevoli di sorta dati dalla variazione del numero dei livelli gerarchici di memorie cache;
2. NON ci sono effetti notevoli di sorta dati dalla variazione della dimensione delle cache sui vari livelli;
3. Ci sono effetti dovuti alla variazione della frequenza del processore;
4. NON ci sono variazioni notevoli dovute alla variazione delle latenze dei vari componenti di memoria, soprattutto della memoria RAM;
5. La tecnica di memoizzazione permette enormi risparmi in termini di tempo d'esecuzione, e quindi di energia, rispetto all'esecuzione degli stessi benchmark, in forma non memoizzata, anche su architetture di tipo ARM.

Il **primo** di questi cinque punti quindi porta ad affermare che la definizione o l'acquisto di architetture con livelli gerarchici di memoria cache varianti da uno a tre non restituisce effettive differenze se viene utilizzata la tecnica di memoizzazione per quelle funzioni

computing intensive, per esempio, enormemente usate in campo bancario. È da notare che il dettaglio delle statistiche, calcolate e restituite dal simulatore, ha fatto intuire come, fissati il numero e tipo di input dei vari benchmark, le dimensioni delle cache non siano in gran parte sufficienti a contenere le informazioni necessarie alla tecnica di memoizzazione per il corretto funzionamento senza la necessità di accesso alla memoria RAM.

Per il **secondo**, dei cinque punti di cui sopra, alcuni specifici esperimenti hanno confermato quanto appena accennato. In particolare quelli in cui si è posta una cache di grandezza "esagerata" per cui infatti il miss rate sulla stessa si è quasi azzerato, azzerando di conseguenza la necessità di accesso alla memoria RAM, per le richieste dei dati memoizzati, fatte dal processore. Nello specifico i tempi di esecuzione sono rimasti praticamente invariati perchè il fattore dominante di questo veniva fissato dalle cache di primo livello, in quanto le richieste ad esse poste venivano soddisfatte quasi totalmente ed erano in numero molto superiore (di 3 ordini di grandezza in media) rispetto a quelle fatte ai successivi due livelli. Queste richieste però erano in gran parte dovute ad operazioni "accessorie" non imputabili alla parte delle funzioni prese come benchmark su cui ha effetto la memoizzazione.

Il **terzo** punto ha permesso di dimostrare come l'utilizzo di processori con frequenze che stanno in un range da 0,5 Ghz a 4 Ghz, quindi partendo da un valore ampiamente al di sotto delle offerte attuali di mercato fino ai top di gamma disponibili ad oggi, faccia variare i tempi di esecuzione delle funzioni, ma questo dipende molto dagli effettivi input dati in ingresso ovvero, in altre parole, da quanti di questi permettano alla tecnica di memoizzazione di svolgere efficacemente il proprio scopo, diminuendo il tempo dovuto alla sola computazione effettiva.

Il **quarto** punto ha permesso di affermare come le uniche variazioni, comunque non molto significative, sono state riscontrate con il dimezzamento della latenza delle cache di livello uno. Questo perchè, come già fatto notare, il numero di accessi alle cache di primo livello sono quasi tutti soddisfatti e sono in valore molto superiore rispetto alle richieste passate ai successivi due livelli.

Dato che per quasi tutti i dati memoizzati necessari il processore ha dovuto fare riferimento all'ultimo livello gerarchico di memoria veloce (la RAM), il **quinto** punto ha evidenziato che, alla variazione più dettagliata della latenza della stessa non sono sorte variazioni elevate sui tempi di esecuzione del benchmark prescelto, questo implica una poca significatività della variazione dei ritardi di memoria RAM, nei range dei componenti attualmente disponibili sul mercato, rispetto alla tecnica di memoizzazione. Inoltre questo punto ha permesso un

confronto diretto fra l'esecuzione delle funzioni memoizzate e le stesse in forma originale. L'enorme risparmio di tempo e quindi di potenza consumata, secondo le motivazioni evidenziate nell'elaborato, induce quindi ad affermare che l'utilizzo della tecnica di memoizzazione per questo tipo di funzioni è sempre efficace.

Tutte le architetture simulate hanno previsto modelli di CPU con esecuzione In-Order, sarebbe utile quindi sviluppare simulazioni analoghe anche con modelli di esecuzione Out-Of-Order. Durante il progetto svolto sono state provate alcune simulazioni di questo tipo per le quali però GEM5 non ha restituito alcun calcolo statistico a causa di probabili errori nel codice sorgente C++. L'analisi di questo avrebbe richiesto un tempo non indifferente che però non è stato disponibile per questo progetto. Sarebbe quindi sicuramente uno dei possiibli grossi sviluppi futuri.
Sarebbero molte le possibili strade successive da intraprendere ed, in particolare, si potrebbe riuscire a valutare le stesse casistiche simulando però architetture con un numero di processori maggiore di uno. Questo comporterebbe anche la valutazione di diversi protocolli di coerenza delle cache, già in parte definiti da GEM5, in base alle scelte architetturali.
Un altro passo, non ancora possibile con le attuali feature messe a disposizione del simulatore, potrebbe essere quello di estendere la dimensione della memoria RAM adibita alla tecnica di memoizzazione. L'attuale kernel e tipo di macchina messi a disposizione da GEM5 limita la dimensione della RAM a 1GB.
Come evidenziato nell'elaborato, esiste un nuovo modello di sistema di memoria, denominato Ruby, molto più dettagliato di quello utilizzato per il progetto in esame. Purtroppo questo modello non è ancora disponibile per le architetture ARM ma i tempi di sviluppo sono abbastanza veloci per questo simulatore, quindi un altro task futuro possibile sarebbe quello di valutare alcune casistiche sfruttando questo nuovo modello di memoria.

Sono quindi tante le possibili strade da percorrere in questo interessante ramo di ricerca che, speranzosamente, potrà essere uno dei “pezzi del puzzle” che porteranno ad uno sviluppo sostenibile, ed un utilizzo più attento all'ambiente, delle tecnologie IT.

APPENDICE A

A.1 QEMU

QEMU è un software open source di virtualizzazione ed emulazione di diverse architetture hardware.

FIG. A.1: Logo di QEMUlator

In particolare, quando usato come EMULATORE, può eseguire interi Sistemi Operativi e programmi fatti per una determinata macchina, con relativa architettura, su altre macchine differenti ("host machine"). Nel far questo è in grado di raggiungere buone prestazioni usando una traduzione dinamica delle operazioni. Quando usato come VIRTUALIZZATORE, riesce invece a raggiungere prestazioni vicine a quelle native eseguendo il codice direttamente sulla CPU della macchina host. QEMU supporta inoltra la virtualizzazione quando esegue le proprie operazioni sotto lo "xen hypervisor" o quando usa il KVM (Kernel Virtualization Module) in Linux. In particolare, in quest'ultimo caso, può virtualizzare sistemi: x86, ospiti S390, PowerPC server ed embedded.

La versatilità e la vasta gamma di sistemi host su cui è possibile utilizzarlo (Linux, Microsoft Windows e qualche versione UNIX) nonché quella dei microprocessori dei sistemi target, rendono questo software uno strumento molto potente. Infatti può essere davvero utile per provare diversi sistemi operativi, testare software ed eseguire applicazioni che non sarebbe possibile utilizzare sulla piattaforma nativa del proprio calcolatore.

Nell'utilizzo fatto per il progetto, è stato necessario sfruttare le funzionalità di networking messe a disposizione. In particolare, per la comunicazione fra il sistema host e quello simulato, è stata modificata la configurazione di base dell'interfaccia di rete ethernet del primo (tramite apposite linee guida generiche dispnibili sulla documentazione di QEMU) in modo da definire un “bridge” fra questa e l'ìnterfaccia di rete virtualizzata. Questo meccanismo ha permesso di scambiare documenti e file con il Sistema operativo DEBIAN virtualizzato su architettura ARM.
Nello specifico sono stati scambiati i file necessari alla configurazione e della classpath Java (versione 0.98) e della JAMVM che si è poi cercato di avviare sul sistema. Numerosi sono stati i problemi derivanti dall'infinità serie di eccezioni richieste in fase di configuazione e “make” (con riferimento al meccanismo “make” di shell linux) richiesti sia per il classpath che per la virtual machine. Per questo motivo e per altri, già evidenziati nel capitolo 4, si è scelto infine di utilizzare la JVM ORACLE per architetture ARM già preconfigurata.[22]

A.2 GEM5: ulteriori dettagli

Nel capitolo 4 è stato introdotto il simulatore GEM5, ed il dettaglio dello stesso, in termini di modelli di memorie e gestione “a porte” delle connessioni. Di seguito invece saranno presentate i possibili modelli di CPU implementati ed, infine, il processo di installazione e creazione di una macchina ARM da simulare in modalità Full System.

A.2.1 Modelli delle CPU

Il simulatore, allo stato attuale, prevede una categorizzazione fra **SimpleCPU** e **O3CPU**. Queste due categorie raccolgono delle proprietà generali successivamente specializzate in diverse sottocategorizzazioni.

Il SimpleCPU è un modello puramente funzionale che implica un'esecuzione di tipo "**InOrder**". Per esso sono definite due sottocategorie: "AtomicSimpleCPU" e "TimingSimpleCPU". Dal punto di vista tecnico e di programmazione, entrambi ereditano dalla classe BaseSimpleCPU che prevede:

- Gestisce le statistiche comuni a tutti i modelli di tipo SimpleCPU;

- Definisce funzioni per il checking di interrupt, per il settaggio e la gestione di richieste di fetch, per la gestione del setup di pre-esecuzione, per la gestione delle azioni post-esecuzione e per l'avanzamento del program counter (PC) alla prossima istruzione.

Non possono essere direttamente istanziati oggetti di questa classe e quindi si deve necessariamente usare una delle due categorie sopra citate: "AtomicSimpleCPU" o "TimingsimpleCPU".
L' "AtomicSimpleCPU" è la versione che usa accessi alla memoria di tipo "atomico" (vedi capitolo 4 paragrafo 4.1.2). Usa la latenza stimata dall'accesso "atomic" per stimare a sua volta il tempo di accesso cache totale. Eredita dalla BaseSimpleCPU ed in più implementa funzioni di lettura e scrittura della memoria e definisce la porta che è effettivamente usata per agganciarsi alla memoria e connette la CPU alle cache.
La "TimingSimpleCPU" è invece la versione che usa accessi alla memoria di tipo "timing" (vedi capitolo 4 paragrafo 4.1.2). Rispetto alla precedente, va in "stallo" in accesso alla cache e aspetta la risposta del sustema di memoria prima di proseguire con l'esecuzione. Come detto eredita da BaseSimpleCPU ed implementa le funzioni aggiuntive del modello "AtomicSimpleCPU". In più, definisce le funzioni necessarie per gestire le risposte date dalle memorie alle richieste di accesso.
Il modello di tipo O3CPU è stato definito nella release v2.0. Rispetto ale "SimpleCPU" implementa un'esecuzione di tipo "**OutOfOrder**", quindi ha una gestione diversa delle pipeline e gode di altre funzionalità (per esempio i "branch predictor"). Nelle simulazioni del progetto sono state avviate delle macchine con modelli CPU di tipo "TimingSimpleCPU", per cui in questa sede verrà tralasciata la descrizione del funzionamento dettagliato dell'esecuzione OutOfOrder concentrandosi su un'analisi dei meccanismi implementati per un'esecuzione InOrder.

L'implementazione della pipeline in esecuzione InOrder è stata fatta in maniera “astratta” riferendosi a come dovrebbe essere e comportarsi una pipeline in un qualsiasi modello di CPU. In generale, dai vari stage di una pipeline, ci si dovrebbe aspettare:

- La gestione d esecuzione di operazioni di tipo “Decode” e “Execute”;

- Gestione dell'invio di un'istruzione al successivo stage se le operazioni sono terminate con successo e se il buffer dello stage successivo ha spazio per l'istruzione in questione. Viceversa, il mantenimento dell'istruzione nel buffer d'istruzione della pipeline.

Scendendo sul tecnico, la classe "PipelineStage" si occup dell'effetiva gestione di cui al punto due del precedente elenco ma non si occupa del primo. Più specificatamente, non ci sono degli stage della pipeline definiti come “Decode” o “Execute”, piuttosto questa classe permette alle istruzioni di definire quale task devono svolgere in un particolare stage a cui sono. Questo meccanismo permette di modellare pipeline arbitrariamente lunghe e quindi un possibile sviluppo facile per un'eventuale necessità di variaziona dei modelli di base del simulatore.
Per quanto riguardo la gestione delle comunicazioni fra i vari stage, si presentano I seguenti scenari:

- L'invio di un'istruzione al prossimo stage;

- L'invio di una segnale di “stallo” all'indietro al precedente stage;

- L'invio di un cosiddetto “squash” (messaggio di “branch misprediction”) all'indietro al precedente stage.

Le comunicazioni “in avanti” sono le più semplici e riguardano i casi in cui un'istruzione ha finito tutti I task necessari per un determinato stage e verrà piazzata (del PipelineStage) in una coda che è in lettura per il prossimo stage. Tecnicamente, le informazioni sul collegamento fra gli stage sono prese dalla stracut “InterStageComm” trovabile nel file sorgente “comm.hh”.
Le comunicazioni “all'indietro”, che sono quelle riguardandi messagli di “stallo” e di “squash” e vengono gestite secondo i concetti di “wire”. Quando una CPU e definitia, ogni “wire” è parametrizzato con una certa latenza che governa quanti “tick” una stage deve aspettare per recuperare le informazioni da quel “wire”.

La definizione base di una pipeline nel modello InOrder è quella a 5-stage che sono: Instruczion Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), Register Write Back (WB). [23]

A.2.2 Build & Run

La definizione di un' architettura, per effettuare successivamente le diverse simulazioni utili agli scopi, passa dal “build” di una macchina base in relazione all'ISA a cui si vuole fare riferimento. Il sistema di definizione appena citato è basato su Scons, un software open-source implementato in Python. Il file principale Scons è chiamato Sconstruct e si trova ala radice dell'albero dei sorgenti, inoltre altri Sconscript si possono trovare l'uno l'albero, tipicamente vicini ai file a cui sono associati..
Gli obiettivi di costruzione sono nella forma del tipo:

scons build/ARM/gem5.opt

ove per build si può scegliere una cartella piazzata ovunque nel file system ma che abbia questo nome. La parte “ARM” è riferita al tipo di ISA scelto, e permette al costruttore scons di scegliere determinate caratteristiche che controllano le funzionalità del simulatore, tipo appunto l'ISA usta, che tipi di CPU includere, che protocollo di coerenza Ruby, ecc..
Il “gem5.opt” è il nome del binario gem5 da costruire, quello che poi sarà richiamato in fare di “running”, che specifica l'insieme di flag di compilazione usati.

Le possibili scelte sono:

- “gem5.debug” che non prevede ottimizzazione. Questo implica che le variabili non saranno ottimizzate, ed il flusso di controllo non avrà comportamenti anomali o che possano soprendere. Tutto ciò rende facile lavorare con tool tipo gdb, al costo di una elevata lentezza. Sarebbe da preferire questa scelta quando serve usare appunto tool come gdb, valgrind e non si vogliono “oscuramenti” di dettagli.

- “gem5.opt” prevede invece l'ottimizzazione e comporende alcune funzionalità di debug quali, ad esempio, “assert” e “Dprintf”. In questo caso si ha un buon bilanciamneto fra velocità di simulazione e possibilità di capire cosa stia accadendo in caso di comportamenti anomali. È la versione migliore nella maggior parte delle circostanze.
-
- “gem5.fast” prevede anche questa l'ottimizzazione ma non ha alcun tipo di funzionalità di debug. È la versione più performante in termini di velocità al discapito di uno sforzo elevato per capire cosa non funzioni correttamente in caso di comportamenti inaspettati. Ovviamente è la versione raccomandata solo se si è realmente confidenti che tutto lavorerà perfettamente e quindi si vuole raggiungere il picco di prestazioni da parte del simulatore.

- “gem5.prof” è simile al precedente ma inoltre include alcuni strumenti che gli permettono di essere usato con tool di profilazione tipo “gprof”. Non è una versione molto usata ma è utile quando si vuole identificare l'area di GEM5 che dovrebbe essere modificata per migliorare le prestazioni.

- “gem5.perf” è una versione complementare alla precedente ed include strumenti per usare tool di profilazione “google-pprof”. Probabilmente sostistuirà completamente il precedente per tutti i sistemi basati su sistema operativo Linux.

La fase di Run viene richiamata come esemplificato nel capitolo 4 paragrafo 4.1.2 [24]

A.2.3 Interazione col simulatore: m5term

Dopo aver fatto il “build” delle componenti di base della macchina d'interesse, si può avviare il test di determinati benchmark o il sistema operativo full system definito nell'immagine disco preconfigurata (esistono versioni messe a disposizioni sul sito di GEM5). Come si evince dalla figura A.2, l'avvio di una simulazione fa si che il sistema apra subito le porte 5900 e 3456

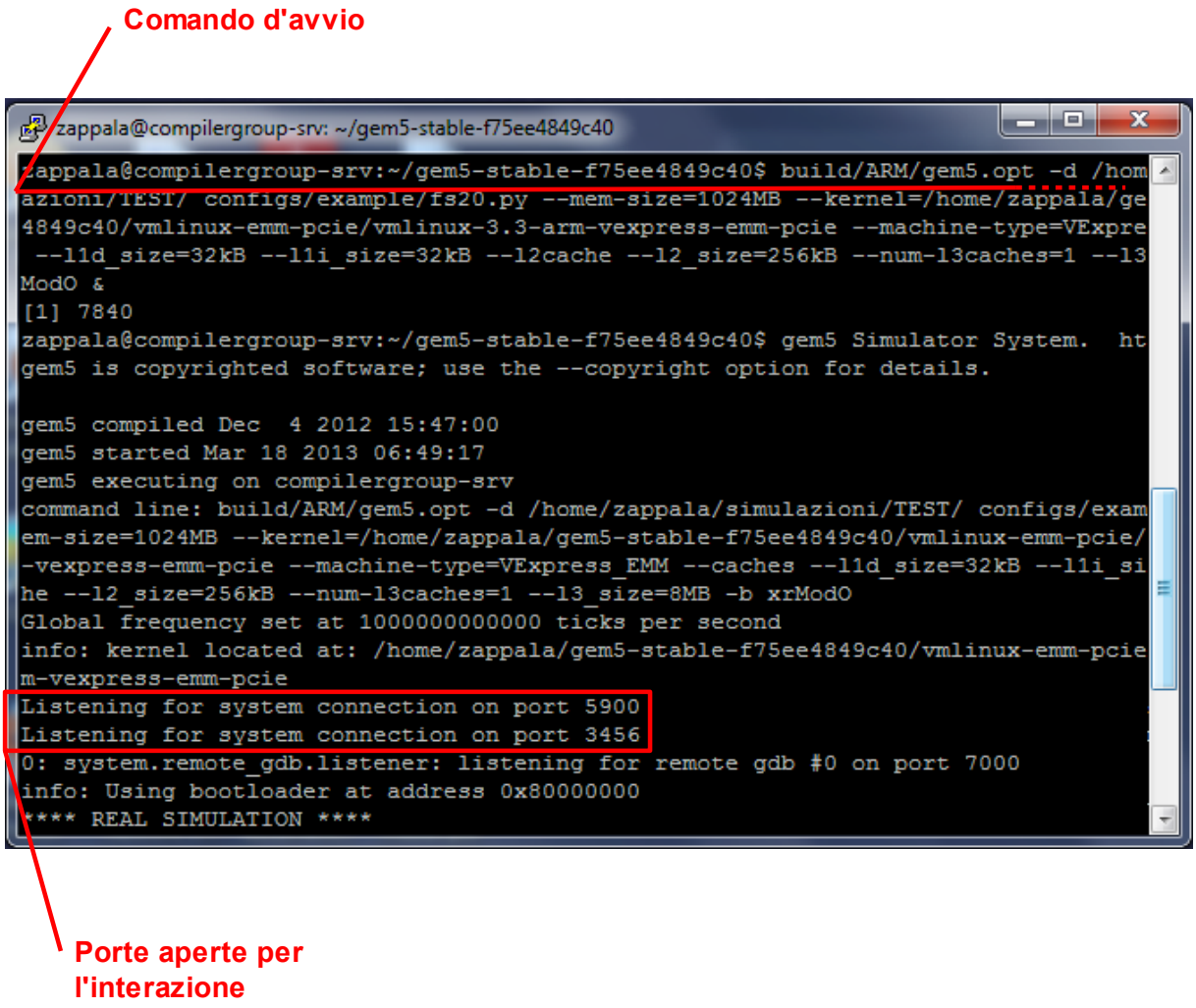


FIG. A.2: Avvio simulazione GEM5 da terminale

per permettere all'utente di comunicarvi. Se sono state avviate altre simulazioni della stessa macchina, automaticamente la seconda porta aumenta di valore in base alla disponibilità effettiva. La connessione sulla prima porta avviene tramite client VNC, mentre sulla seconda tramite protocollo Telnet. In entrambi i casi però l'interazione risulta lenta e le risposte del sistema arrivano con un quantomento “fastidioso” ritardo. GEM5 mette a disposizione un'utility molto migliore del telnet per la comunicazione col sistema simulato: m5Term.

Fornita con il pacchetto contenente il simulatore, questa utility permette di collegarsi al sistema da terminale nel modo individuato in figura A.3.

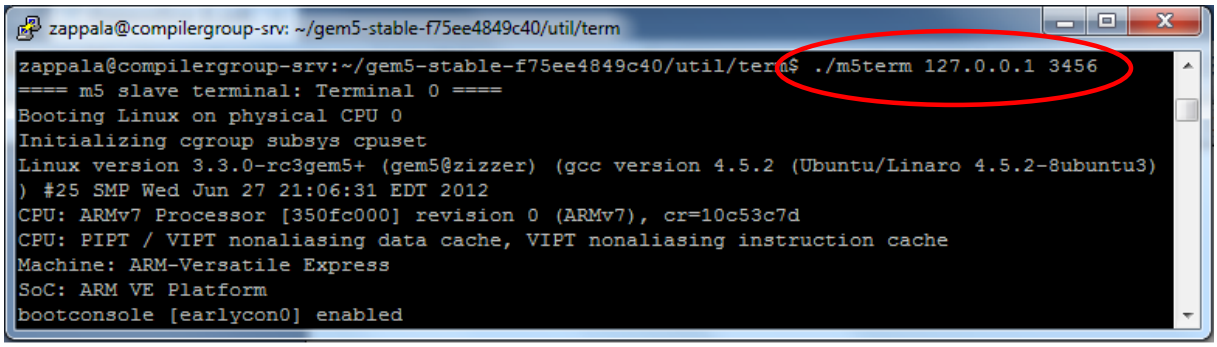


FIG. A.3: Avvio dialogo col sistema simulato tramite "m5term"

Se collegati con m5Term, è possibile, dalla console shell del sistema simulato, utilizzare determinate istruzioni utili in fase di simulazione (vedi: checkpoint, dump e reset statistiche, chiusura simulazione, ecc...).

A.3 SWIG

FIG. A.4: Logo del tool SWIG

SWIG è un tool di sviluppo softweare che connette programmi scritti in C e C++ con una vasta gamma di linguaggi di programmazione di alto livello. È usato con linguaggi di scrpting molto comuni fra cui Perl, PHP, Python, TCL and Ruby. Inoltre è possibile connettersi a linguaggi non di scripting tipo C#, D, Go languag, Java e Android, Lua, Modula 3, OCAML,

Octave, R, e LISP più comuni (CLISP, Allegro CL, GFFI, UFFI). Sono supportati inoltre un numero consistente di schemi di implementazione compilati ed interpretati (Guile, MzScheme/Racket, Chicken).
SWIG è maggiormente utilizzato per creare ambienti di programmazione compilati o interpretati, interfacce utente, e fornisce tool per testare e prototipare software C/C++
Tipicamente è utilizzato per effettuare il “parsing” di interfacce C/C++ e generare “glue code” richiesto per I linguaggi sopra citati, i modo da permettergli di chiamare oggetti, metodi e funzioni, scritte in C/C++.
Con questo tool è inoltre possibile esportare l'albero di “parsing” in forma di file XML e di “s-expression” LISP. Da notare che è un software libero ed il codice da esso generato è compatibile con lo sviluppo di progetti commerciali e non.

Andando più nello specifico, ecco alcune delle funzionalità e possibilità di utilizzo messe a disposizione da questo tool.

- È possibile creare programmi C/C++ più “potenti”. Utilizzando SWIG si può sostituire la funzione main() di un progrmama C con un interprete di script da cui è possibile controllare l'applicazione. Questo aggiunta permette un maggiore livello di flessibilità e rende il programma “programmabile”, ovvero permette all'utente ed allo sviluppatore di modificare il comportamento del programma senza dover modificare il codice di basso livello ma bensì utilizzando l'interfaccia di scripting. Questo fatto porta ovviamente un considerevole numero di benefici che risaltano subito all'occhio di chi ha dimestichezza con la progrmamazione.

- Prototipizzazione e debug rapido permettono ai programmi C e C++ di essere piazzati in un ambiente di scripting che può essere usato per effetuare dei test. Per esempio, è possibile testare una librearie con una collezione di scropt o usare l'interprete di script com un'interfaccia per il debug delle applicazioni. Inoltre, dato che SWIG permette di non dover fare riferimento e modifiche diretta al codice di basso livello C/C++, esso può essere usato anche se il prodotto finare non fa riferimento a concetti di scripting.

- System Integration. I linguaggi di scripting lavorano bene per controllollare ed “incollare” insieme dei componenti software “loosely-coupled”. Con SWIG, differenti programmi C/C++ possono essere trasformati in moduli di estensione di linguaggi di scripting. Questi moduli posso successivamente essere combinati assieme per creare nuovi ed interessanti applicazioni.

SWIG è spesso comparato a compilatori di linguaggi di definizione d'interfacce (Interface Definition Language, IDL) simili a quelli trovabili in sistemi CORBA o COM. Sebbene ci siano alcune similarità, il punto centrale di SWIG è di far in modo che non sia necessario aggiungere un “layer” extra di specifica IDL alle proprie applicazioni. Inoltre è molto di più di un tool di sviluppo e prototipizzazione di applicazioni e, per le motivazioni, si rimanda alle ampie documentazioni reperibili dal sito ufficiale.:

In conclusione è giusto notare che sebbene SWIG sia comparato occasionalmente altri e più specializzati tool di costruzione d'estenzioni attraverso linguaggi di scripting (per sempio, Perl XS, Python bgen, ecc...), il “cliente” principale è il programmatore C/C++ che vuole aggiungere componenti di linguaggi di scripting alla propria applicazione. Per questo motivo, SQIG tende ad avere focus leggermente differenti rispetto a tool progettati per costruire piccoli moduli per uso diffuso in applicazioni distribuite di linguaggio di scripting. [25]

A.4 PuTTY

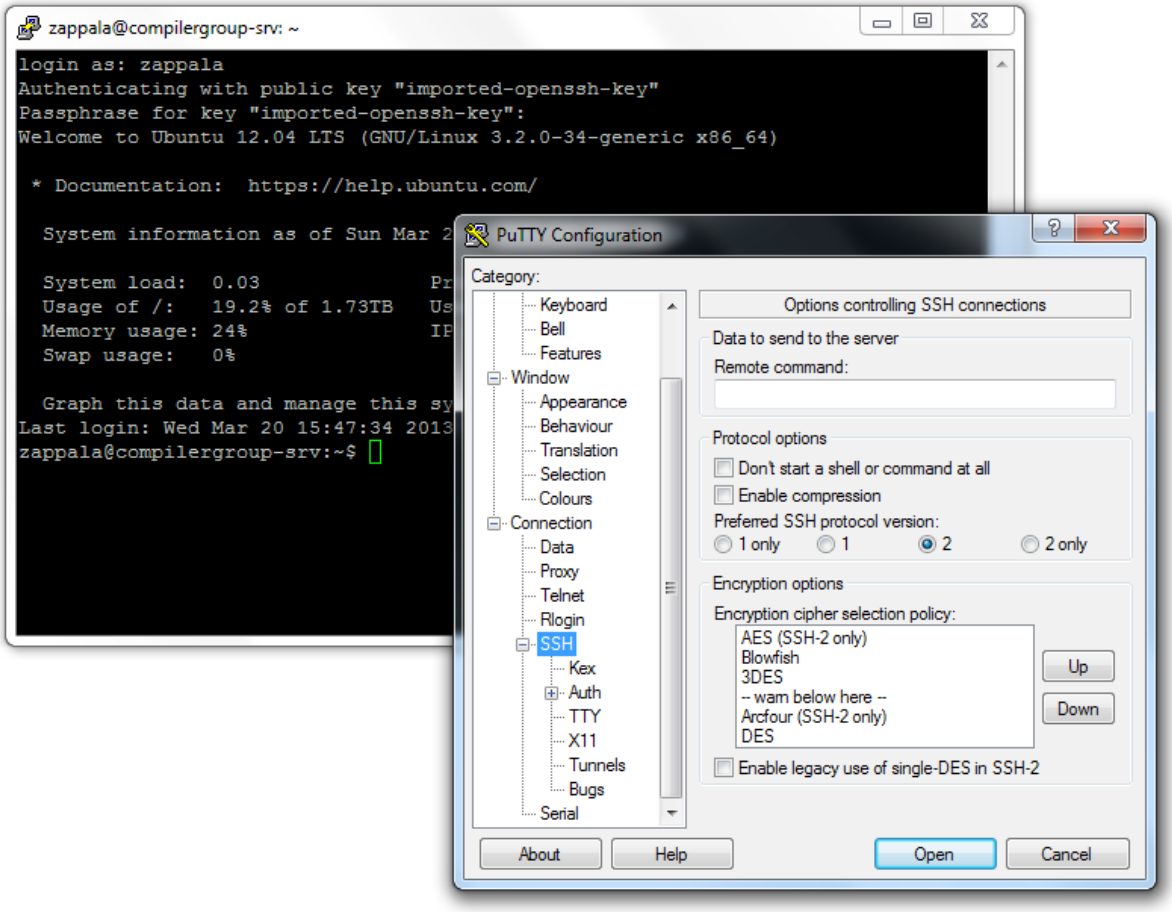


FIG. A.5: finestra di accesso e configurazione e finestra di dialogo Telnet col server "compilergroup@dei.polimi.it" messe a disposizione da PuTTY

PuTTY è un client SSH, Telnet ed rlogin combinato con un emulatore di terminale.
È una suite di software libero, originariamente disponibile solo per sistemi Microsoft Windows, ma in seguito anche per vari sistemi Unix e Unix-like. Esistono anche adattamenti non ufficiali per altri sistemi, ad esempio Symbian.

In relatà è una suite di tool comprendenti i seguenti:

- PuTTY - Il vero e proprio client SSH, Telnet ed rlogin che include anche l'emulatore di terminale, con possibilità di salvare i server e le preferenze;

- PSCP - Un client testuale per SCP (Secure CoPy, ovvero un comando di shell linux utile per trasferire file in/da remoto con la sicurezza di una connessione SSH), utilizzato durante il progetto di tesi per scambiare i file d'immagine disco ed altre componenti utili al simulatore GEM5;

- PSFTP - Un client testuale per SFTP (Secure File Transfer Protocol);

- PuTTYtel - Un client solo per Telnet;

- PuTTYgen - Un programma per la generazione di chiavi RSA e Digital Signature Algorithm (DSA);

- Pageant - Un programma che mantiene in memoria delle chiavi RSA e Digital Signature Algorithm, mettendole a disposizione degli altri componenti (ed anche a WinSCP) per l'autenticazione con SSH, in modo da far avvenire automaticamente la scelta del tipo di idetificazione nelle comunicazioni con i sistemi remoti.[26]

BIBLIOGRAFIA

[1] T. Restorick. AN INECIENT TRUTH. Global Action Plan Report 2007.

[2] Planer I. BUSINESS INTELLIGENCE METHODOLOGIES APPLIED TO GREEN IT.

[3] vedi [1].

[4] C. Francalanci e E. Capra. GREEN IT. SFIDE E OPPORTUNITÀ. Mondo Digitale, pagine 36-42, 2008.

[5] S. Murugesan. HARNESSING GREEN IT: PRINCIPLES AND PRACTICES. IT professional, 24-33, 2008.

[6] G.Agosta e E. Capra. GREEN SOFTWARE ANCHE Le APPLICAZIONI CONSUMANO ENERGIA. Mondo digitale, pagine 9-24, 2011

[7] D. Michie. MEMO FUNCTIONS AND MACHINE LEARNING. Nature, pgine 19-22, 1968.

[8] Matthew Finifter, Adrian Mettler, Naveen Sastry e David Wagner. VERIABLE FUNCTIONAL PURITY IN JAVA. In CCS '08: Proceedings of the 15th ACM conference on Computer and communications security, pagine 161-174, 2008.

[10] M. Bessi, “UNA METODOLOGIA BASATA SULLA MEMOIZZAZIONE DINAMICA PER L'EFFICIENZA ENERGETICA DEL SOFTWARE APPLICATIVO”, Politecnico di Milano, a.a. 2009-2010

[11] S. Mayhew. IMPLIED VOLATILITY. Financial Analysts Journal, pagine 8-20, 1995.

[12] F. Black and M. Scholes. THE PRICING OF OPTIONS AND CORPORATE LIABILITIES. The journal of political economy, pagine 637-654, 1973.

[13] G. Agosta, Marco Bessi, Eugenio Capra, Chiara Francalanci. AUTOMATIC MEMOIZATION FOR ENERGY EFFICIENCY IN FINANCIAL APPLICATIONS. Dipartimento di Elettronica e Informazione, Politecnico di Milano.

[14] R.C. Johnson. HAS ARM WON THE PROCESSOR WARS? HARDLY! Anno 2012 (http://www.eetimes.com/electronics-news/4376703/ARM-has-not-won-the-processor-wars-yet)

[15] Acorn Archimedes Promotion from 1987.

[16] R. Murray. 32 BIT OPERATION. Anno 2004 (http://www.heyrick.co.uk/assembler/32bit.html)

[17] BBC News. AMD IN CHIP TIE-UP WITH UK'S ARM. 30 ottobre 2012 (http://www.bbc.co.uk/news/technology-20137041)

[18] ARM DEVELOPER GUIDES AND ARTICLES. Anno 2013 (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0001a/CHDEFAGB.html)

[19] M. Chiappone e M. Zappalà, " Power Analysis: Definition of a Python parser in order to integrate McPAT and GEM5", Politecnico di Milano, anno 2012

[20] GEM5: GENERAL MEMORY SYSTEM, http://www.m5sim.org/General_Memory_System ultima consultazione: 03 Marzo 2013
GEM5: CLASSIC MEMORY SYSTEM, http://www.m5sim.org/Classic_Memory_System ultima consultazione: 03 Marzo 2013
GEM5: MEMORY SYSTEM, http://www.m5sim.org/Memory_System ultima consultazione: 03 Marzo 2013
GEM5: RUBY, http://www.m5sim.org/Ruby ultima consultazione: 03 Marzo 2013

[21] IBM System and Technology, SERVER IBM POWER 770 – SCHEDA TECNICA, 2012
IBM System and Technology, SERVER IBM POWER 750 – SCHEDA TECNICA, 2011
IBM System and Technology, IBM POWER 755 SERVER – SCHEDA TECNICA, 2011
IBM System and Technology, IBM POWER 775 – DATA SHEET, 2011
IBM System and Technology, IBM POWERLINUX 7R2 SERVER – DATA SHEET, 2013
IBM ITSO, IBM POWER 710 and 730 – Technical overview and introduction, 2010

DELL, MEMORY FOR DELL POWEREDGE 12th GENERATION SERVERS, 2012

INTEL, Intel XEON PROCESSOR 5600 SERIES – DATASHEET VOL 1 & 2, 2011
INTEL, Intel XEON PROCESSOR E3-1200 v2 – DATASHEET VOL 1 & 2, 2012
INTEL, Intel XEON PROCESSOR E7-8800/4800/2800 – DATASHEET VOL 1 & 2, 2011

[22] http://wiki.qemu.org/

[23] GEM5: CPU MODELS, http://www.m5sim.org/CPU_Models ultima consultazione: 18 Marzo 2013

[24] GEM5: BUILD SYSTEM, http://www.m5sim.org/Build_System ultima consultazione: 18 Marzo 2013

[25] http://www.swig.org/

[26] http://it.wikipedia.org/wiki/PuTTY