

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN SCIENZE DELL'INFORMAZIONE

OTTIMIZZAZIONI MICROARCHITETTURALI PER L'HIGH PERFORMANCE COMPUTING

Relazione finale in
Biofisica delle Reti Neurali e loro applicazioni

Relatore
prof. Renato Campanini

Presentata da
Benini Luca

Co-relatore
dott. Matteo Roffilli

Sessione II
Anno Accademico 2003/2004

Per più di trent'anni, i microprocessori hanno raddoppiato la propria potenza ogni 18 o 24 mesi. Quel progresso continuerà per un'altra decina d'anni, o almeno così dicono i produttori di chip. Poi bisognerà trovare qualche nuova tecnologia per sostituire il semiconduttore in silicio.

Sfortunatamente le aziende che realizzano microprocessori e li usano per realizzare computer non possono vivere sempre alle spalle della Legge di Moore. Con il diminuire di dimensioni dei transistor (ce ne saranno un miliardo su un singolo chip nel giro di cinque anni), i chip diventano esponenzialmente più costosi da progettare, costruire e testare.

Circa il 60 per cento dei miglioramenti complessivi nei microprocessori è derivato da frequenze di clock maggiori permesse da transistor più piccoli e più veloci. Il restante 40 per cento è invece legato ad architetture di calcolo che consentono l'esecuzione di più di un'istruzione per ciclo di clock.

Tuttavia questo rapporto può essere migliorato attraverso un reale sfruttamento delle capacità vettoriali dei microprocessori moderni. Molto spesso queste opportunità ottimizzative vengono trascurate, in quanto il loro sfruttamento non risulta immediato.

In questo lavoro di tesi ci si propone di affrontare la scelta e l'implementazione di costrutti ottimizzati attraverso la metodologia nota con il nome di closed loop cycle concentrandoci su diversi livelli di ottimizzazione, che a fronte di una complessità via via crescente permetteranno di ottenere prestazioni sempre migliori.

Nella realizzazione di questa tesi ci concentreremo prevalentemente su alcune delle più interessanti tecnologie emergenti.

Tutto il codice che verrà implementato sarà totalmente compatibile sia con le attuali macchine a 32 bit, sia con le sempre più diffuse macchine con architettura AMD64.

Terremo in grande considerazione la necessità sempre più pressante di software multipiattaforma, effettuando anche il porting delle ottimizzazioni verso altri sistemi operativi.

La possibilità di lavorare in un importante progetto di ricerca, con pesanti requisiti computazionali, permetterà di ottenere misurazioni estremamente precise dell'effettivo speed up delle ottimizzazioni realizzate.

Durante la stesura della tesi valuteremo inoltre alcuni tra i più diffusi strumenti per la valutazione delle performance al fine di individuare quello che meglio si presta a valutare le ottimizzazioni realizzate.

Naturalmente anche la possibilità offertami dal prof. Campanini di lavorare a stretto contatto con il gruppo di ricerca di Biofisica delle Reti Neurali di Cesena rappresenterà certamente un'importante opportunità di crescita sia professionale che personale.

Indice

1	Potenzialità hardware	7
1.1	Applicazioni della potenza di calcolo allo sviluppo del software	8
1.2	Applicazioni della potenza di calcolo al Number Cruncing	10
1.3	Linguaggi e Ottimizzazioni	10
2	Metodologia delle ottimizzazioni	13
2.1	Workload	13
2.1.1	Misurabilità	14
2.1.2	Riproducibilità	14
2.1.3	Staticità	14
2.1.4	Rappresentatività	15
2.2	Closed loop cycle	15
2.2.1	Acquisizione dei dati	16
2.2.2	Analisi dei dati	18
2.2.3	Generazione di approcci alternativi	20
2.2.4	Implementazine delle alternative	21
2.2.5	Verifica dei risultati	21
3	Hardware Bottleneck	23
3.1	I limiti dei processori moderni	23
3.1.1	Le latenze della memoria e della cache di secondo livello	23
3.1.2	Le dipendenze dei dati	25
3.1.3	Diramazioni (salti condizionali)	25
3.2	Risposte hardware alle problematiche dei processori	26
3.2.1	Riduzione della latenza delle memorie	26
3.2.2	Dipendenze fra i dati	29
3.2.3	Predizione dei salti	30
4	Source Level Optimizations	31
4.1	Opportunità offerte dalle ottimizzazioni a livello di codice sorgente	31
4.1.1	Ottimizzazioni dei compilatori	31
4.1.2	Ottimizzazioni a livello di programmazione C	32
4.2	Applicazione delle ottimizzazioni	32
4.2.1	Architettura di riferimento	32
4.2.2	Sistema Operativo di Riferimento	33

4.3	<i>Compilatore di Riferimento</i>	34
4.4	<i>Prima Implementazione</i>	34
4.5	<i>Ottimizzazioni Applicate</i>	37
4.5.1	<i>Prodotto Scalare</i>	37
4.5.2	<i>Inverted For</i>	38
4.5.3	<i>Versione Finale</i>	38
4.5.4	<i>Eliminazione del chiamante</i>	39
4.5.5	<i>Eliminazione della serialità</i>	40
4.6	<i>Risultati</i>	43
5	<i>Ottimizzazioni Microarchitetturali</i>	45
5.1	<i>Analisi dei istruzioni SSE</i>	47
5.2	<i>Prima implementazione</i>	48
5.3	<i>Ottimizzazioni Applicate</i>	51
5.3.1	<i>Instruction Reordering</i>	51
5.3.2	<i>Memory Alignment</i>	53
5.3.3	<i>Versione Finale</i>	53
5.4	<i>Risultati</i>	55
6	<i>Progetto ATLAS</i>	57
6.1	<i>Introduzione ad ATLAS</i>	57
6.2	<i>ATLAS e BLAS</i>	58
6.3	<i>GEMM</i>	58
6.4	<i>Compilazione di ATLAS</i>	60
6.4.1	<i>Alcuni risultati della compilazione</i>	61
6.5	<i>ATLAS su Sistemi Windows</i>	61
6.6	<i>Implementazione</i>	63
6.7	<i>Risultati</i>	63
	<i>Conclusioni</i>	65
	<i>Bibliografia</i>	66

Indice dei listati

4.1	Versione legacy	35
4.2	Versione legacy (chiamante)	36
4.3	Versione finale	39
4.4	Versione ottimizzata priva del chiamante	40
4.5	La struttura utilizzata	40
4.6	Versione finale	41
5.1	Prodotto scalare con SSE	50
5.2	Versione non riordinata	52
5.3	Versione riordinata	52
5.4	Versione con memoria allineata a 16 byte	53
5.5	Versione finale del prodotto scalare	53
6.1	Versione utilizzante la libreria Atlas	63

Capitolo 1

Potenzialità hardware

Nel 1965 Gordon Moore¹ durante una conferenza sui chip di memoria presentò la ben nota osservazione, che ancora oggi porta il suo nome. In particolare egli fece notare che:

*“Ogni chip aveva una capacità circa doppia rispetto al suo predecessore e che ogni 18-24 mesi nasceva una nuova generazione di chip.”*²

Questa legge è applicabile con discreta precisione ai processori e può quindi essere considerata un buon strumento di pianificazione per le industrie microelettroniche, ed anche come previsione per le prestazioni dei calcolatori futuri.

Prendendo in considerazione l'affermazione di Moore, fino ad oggi dimostratasi estremamente precisa, risulta immediatamente evidente come la tecnologia dei semiconduttori, tecnologia fondamentale nella realizzazione di un calcolatore moderno, ha permesso una crescita vertiginosa delle potenzialità dei sistemi di calcolo.

Oggi con poche migliaia di euro è possibile acquistare un personal computer con migliori prestazioni, più memoria, sia principale che secondaria, di un computer acquistato negli anni '70 per un miliardo lire. Se consideriamo che *Gemini*, il computer realizzato dall'IBM su contratto con la NASA per la missione Mercury, impiegava 420 millisecondi per eseguire una moltiplicazione³, e che un calcolatore attuale nello stesso lasso di tempo è in grado di eseguire circa *500 milioni* di moltiplicazioni, sorge istintivamente chiedersi come siano utilizzate le capacità di calcolo dei computer attuali. Qual è il vantaggio di costruire computer sempre più veloci? Perché non potremmo utilizzare una macchina che impiega un mese o una settimana invece di un giorno o di un'ora? Per molti problemi si può anche fare così. Ma il fatto è che oggi stiamo giust'appunto cominciando a raggiungere la potenza di calcolo sufficiente per capire cosa succede in sistemi con migliaia o milioni di variabili; le macchine più veloci sono già in grado di prevedere ciò che può succedere. Si prenda, per esempio, i gas a effetto serra e le modalità con cui stanno modificando il clima globale, uno dei problemi per i quali è stato costruito Earth Simulator⁴. Con i computer abbastanza veloci da predire con precisione i cambiamenti climatici, possiamo conoscere con un certo grado di precisione quale percentuale di anidride carbonica nell'atmosfera determina lo scioglimento delle calotte polari. Analogamente, dato che Earth Simulator costruisce modelli del clima del pianeta a

¹<http://www.intel.com/pressroom/kits/bios/moore.htm>

²<http://www.intel.com/research/silicon/mooreslaw.htm>

³<http://www.hq.nasa.gov/office/pao/History/computers/Ch1-2.html>

⁴<http://www.es.jamstec.go.jp/esc/eng/ESC/index.html>

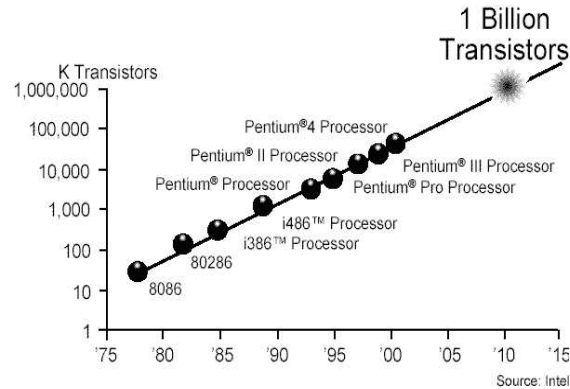


Figura 1.1: La crescita del numero di transistor

un livello incredibile di granularità, è in grado di eseguire simulazioni che tengano conto di fenomeni su scala locale come i temporali. Tali fenomeni possono riguardare aree estese solo 10 chilometri, contro i 30-50 chilometri di risoluzione delle griglie adottate normalmente per i modelli meteorologici. Prendiamo per esempio le difficoltà incontrate quando si cerca di capire e controllare un fenomeno come la fusione nucleare, la panacea di ogni nostro problema energetico. Come afferma Thomas Sterling, membro di facoltà presso il Center for Advanced Computing Research del Caltech⁵

“Potrebbero volerci dieci anni per arrivare a un singolo esperimento di fusione. Un computer più veloce potrebbe anticipare di decenni i tempi di questi esperimenti, consentendoci non solo di progettare reattori sicuri per alimentare il fabbisogno energetico del pianeta, ma anche di sapere come liberarci delle scorie”

1.1 Applicazioni della potenza di calcolo allo sviluppo del software

Le crescenti prestazioni dei sistemi di calcolo moderni hanno permesso lo sviluppo di nuovi strumenti implementativi estremamente più espressivi di quelli utilizzati 10 anni fa.

Sono stati sviluppati linguaggi ad alto livello che con i loro costrutti, sintassi e grammatica tendono ad avvicinarsi il più possibile al linguaggio ed al ragionamento umano, rendendo più rapido l'apprendimento del linguaggio medesimo, ed incrementando di molto la produttività del singolo programmatore.

Questi linguaggi vengono spesso distribuiti in dotazione ad ambienti RAD (Rapid Application Development)⁶ che attraverso interfacce del tipo punta-e-clicca, permettono ad una software house di ridurre drasticamente il *Time to Market*, di ridurre cioè il tempo impiegato da un prodotto, ad esempio un programma, per essere immesso sul mercato dopo la sua progett-

⁵<http://www.caltech.edu>

⁶<http://lib74123.usask.ca/scaa/rad/section2.htm>



Figura 1.2: Cray T3

tazione.

Questo approccio, volto alla semplificazione della fase di realizzazione del software, che ha permesso lo sviluppo di utilissimi strumenti software quali i garbage collector⁷, o le macchine virtuali⁸, tuttavia richiede necessariamente un consumo maggiore di risorse da parte del programma che utilizza queste infrastrutture, si opera quindi una riduzione dei tempi di sviluppo scaricando l'onere in termini di prestazioni sull'utente finale.

Tuttavia la crescente capacità di calcolo ha aperto nuove e, a nostra opinione, più interessanti possibilità.

Basti pensare che Cray⁹ ha annunciato il lancio di un computer high-end basato sui processori Opteron di AMD entro il 2005.

Il nuovo sistema sarà in grado di fornire prestazioni superiori all'Earth Simulator di NEC, l'attuale sistema più potente del pianeta. Il progetto di Cray, denominato Red Storm, sarà in grado di fornire un picco di performance di 40 teraflop, grazie all'adozione di 10000 CPU Opteron. Il sistema offrirà anche possibilità di espansione, permettendo di arrivare fino a ben 60 teraflop.

Il progetto fa parte dell'iniziativa Accelerated Strategic Computing del Dipartimento dell'Energia, che ha come scopo la simulazione di esplosioni nucleari mediante l'impiego di calcolatori ad elevata potenza. Il supercomputer sarà installato nei National Laboratories di Sandia ed avrà un costo di circa 90 milioni di dollari.

⁷http://en.wikipedia.org/wiki/Computer_memory_garbage_collection

⁸<http://java-virtual-machine.net/download.html>

⁹<http://www.cray.com>

1.2 Applicazioni della potenza di calcolo al Number Cruncing

Al momento in cui scrivo si trovano in commercio macchine *convenzionali*, dove per convenzionali si intende con un prezzo di mercato al di sotto della soglia psicologica dei 1000 euro, con una potenza di calcolo pari a circa 3-4 Gigafllops.

Queste macchine, che vengono utilizzate generalmente come calcolatori desktop, presentano delle capacità di calcolo di molti ordini di grandezza superiori a quelle dei calcolatori utilizzati dalla NASA nelle prime missioni spaziali; stabilire come sfruttare queste potenzialità è compito dell'utente, nella scelta del software, e degli sviluppatori nella realizzazione del medesimo. Dal punto di vista computazionale queste macchine permettono di affrontare problemi ritenuti prima inaffrontabili per le necessità di calcolo.

Si possono realizzare simulazioni più complesse, che tengono conto di un maggior numero di variabili, creando quindi modelli per il comportamento di un sistema, sia esso biologico o metereologico, più fedeli alla realtà permettendo quindi una maggiore comprensione dei problemi che si va ad affrontare.

Ad esempio una delle principali attività di ricerca nella moderna cosmologia è lo studio delle strutture su grande scala.

Per grande scala si intende una estensione delle nostre osservazioni su distanze molto più grandi della galassia in cui viviamo. Una adeguata unità di misura per queste distanze è il Megaparsec (Mpc), che corrisponde a circa tre milioni di anni-luce (lyr).

La nostra galassia ha una estensione nel piano di circa 90,000 lyr. Le galassie si aggregano in strutture sempre più grandi, dai sistemi binari ai gruppi, agli ammassi (circa un migliaio di galassie) fino ai superammassi.

Le scale di distanze coinvolte arrivano fino a circa 300 Mpc. Lo studio della distribuzione attuale di materia su queste scale è particolarmente importante perché si ritiene rifletta le condizioni iniziali che hanno dato luogo alle strutture osservate. Queste strutture si sono evolute nella loro forma attuale per effetto della interazione gravitazionale e dell'espansione dell'Universo.

L'evoluzione temporale di una di queste strutture può essere seguita per mezzo di una simulazione numerica. In queste simulazioni un modello al computer della distribuzione e delle interazioni simula la struttura reale.

Nei limiti di applicabilità del modello una tale simulazione è l'analogo numerico di una misura in laboratorio.

Risulta quindi evidente l'importanza di sfruttare al meglio la potenza di calcolo messa a disposizione dai nuovi microprocessori.

1.3 Linguaggi e Ottimizzazioni

Un miglior sfruttamento delle capacità di calcolo delle macchine a disposizione permette di affrontare ed eventualmente risolvere problemi più complessi, considerando ciò risulta quindi

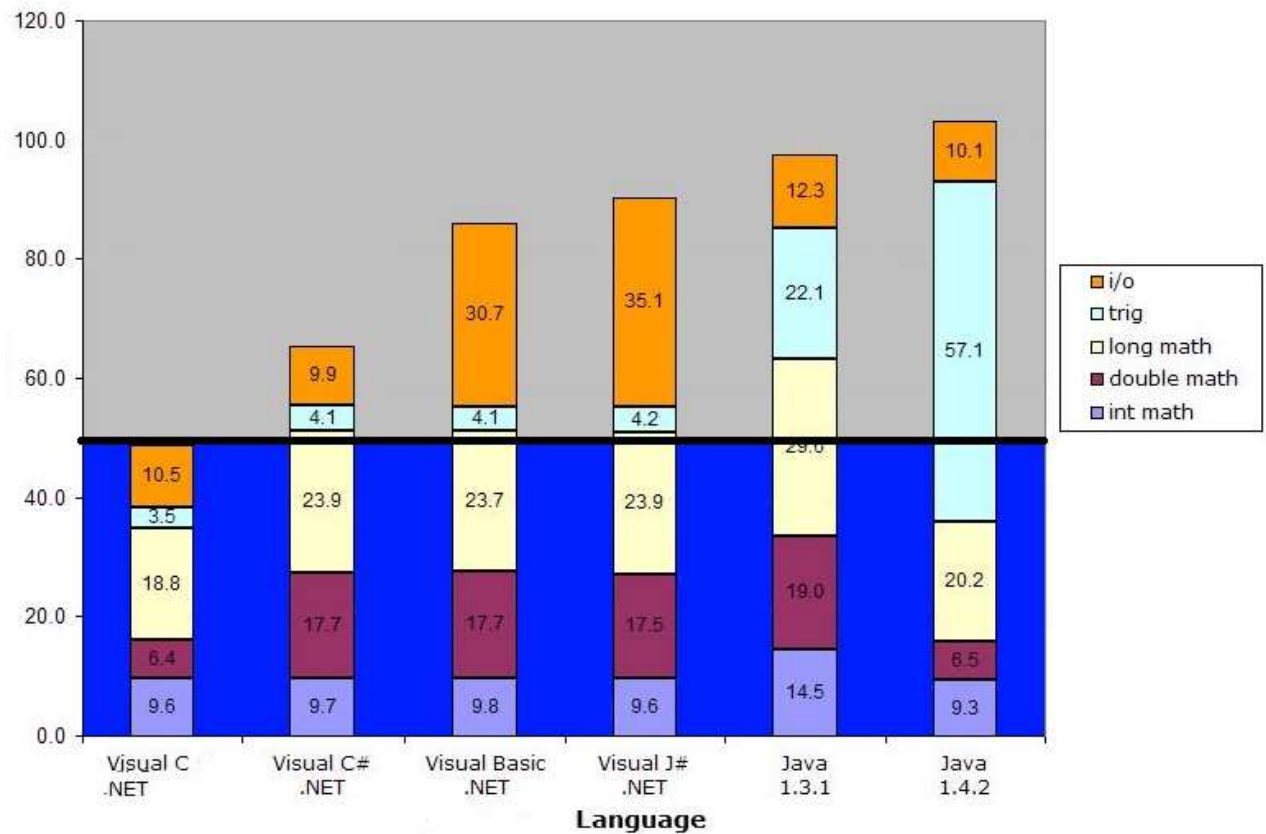


Figura 1.3: Performance test su alcuni linguaggi

fondamentale utilizzare al meglio l'hardware a disposizione.

Per rendere possibile questo sfruttamento si necessita di un linguaggio particolarmente performante, senza cioè tutti i moduli software presenti nei linguaggi moderni, che permetta uno stretto contatto con la macchina, al fine di garantire un alto controllo da parte dello sviluppatore sull'hardware sottostante.

Allo stato attuale il miglior linguaggio per questo scopo a disposizione della comunità mondiale è senza ombra di dubbio il linguaggio C¹⁰, che è anche quello che utilizzeremo nel nostro sviluppo. Inoltre un vantaggio indiscutibile del linguaggio C è la semplicità con cui esso si lega con il linguaggio che più di ogni altro è orientato al raggiungimento di alte performance: il linguaggio Assembler¹¹, che permette ad un programmatore con una buona conoscenza dell'hardware sottostante di sfruttare al meglio le caratteristiche della macchine, che non sempre vengono utilizzate dai compilatori.

Infatti allo stato attuale i *backend* per la generazione del codice all'interno dei compilatori presentano motori di ottimizzazione complessi, che non riescono quindi a stare al passo del-

¹⁰<http://cm.bell-labs.com/cm/cs/cbook/>

¹¹<http://webster.cs.ucr.edu/>

l'introduzione di nuove caratteristiche all'interno dei processori attuali.

Il linguaggio C presenta inoltre numerose caratteristiche. Qui di seguito ne elenchiamo alcune:

- **Dimensioni del codice ridotte**
Generalmente il codice sorgente di un programma in C ha un peso (in Kb) relativamente piccolo, in questo modo risulta molto agevole trasportare il codice da un PC ad un altro, anche usando un semplice floppy. Dimensioni dell'eseguibile ridotte - Anche una volta compilato, un programma in C, risulta molto piccolo e quindi di più facile diffusione; ovviamente un programma in C potrà essere eseguito solamente sul medesimo Sistema Operativo per cui è stato compilato.
- **Efficienza dei programmi**
Un programma scritto in C, proprio per la possibilità messa a disposizione dal linguaggio di gestire a fondo la memoria, e per le sue dimensioni ridotte, risulta particolarmente efficiente. Può essere compilato su una vasta gamma di computer. Ogni computer può differire dagli altri per almeno due cose, l'architettura ed il sistema operativo; ad esempio un computer con processore x86 e Windows ha delle istruzioni (intese come istruzioni del processore) ed una gestione della memoria diverse da uno Sparc con Linux, però un programma scritto in C può essere compilato su ambedue le macchine, data l'alta disponibilità di compilatori per diverse piattaforme. Certo non è portabile come Java, però il fatto di essere sulla scena da molti anni e la sua enorme diffusione ne fanno, di fatto, uno strumento altamente portabile.
- **È un linguaggio di alto livello**
Un linguaggio di programmazione viene definito di alto livello tanto più si avvicina alla terminologia umana, inversamente si dice che un linguaggio è di basso livello se il suo codice si avvicina al linguaggio macchina (quello formato da 0 ed 1); tipico esempio di linguaggio a basso livello è l'Assembler, mentre linguaggi ad alto livello sono, oltre al C, il C++, il Java e molti altri. La particolarità dei linguaggi ad alto livello è quella di avere una semplice sintassi in cui si usano parole della lingua inglese per descrivere comandi corrispondenti a decine di istruzioni in assembler o centinaia di istruzioni in linguaggio macchina.
- **Facilitazione per la gestione di attività di basso livello**
Il C è considerato il linguaggio di più basso livello tra i linguaggi di alto livello. Questo è dovuto al fatto che ha poche istruzioni, gestisce in maniera efficiente la memoria ed è possibile inserire direttamente all'interno di un file in C del codice Assembler.

Capitolo 2

Metodologia delle ottimizzazioni

Tutti i sistemi, indipendentemente dalla loro natura, sono soggetti alla valutazione delle prestazioni: nella fase di progettazione, assemblaggio, vendita, utilizzo.

In ognuna di queste fasi le prestazioni del sistema sono valutate da persone differenti con differenti scopi e punti di vista. L'obiettivo comune è, tuttavia, verificare che il sistema soddisfi certi requisiti di efficienza e che possa essere utilizzato per risolvere un determinato problema. L'applicazione delle ottimizzazioni ad un modulo software può essere per certi versi associato alla regolazione di determinati parametri in un sistema.

Data la complessità dei moduli software a cui si applicano le ottimizzazioni può rivelarsi fondamentale affrontare il problema seguendo una metodologia che permetta di valutare esattamente se e quando sono stati raggiunti i risultati desiderati, una metodologia che permetta quindi di ottenere una valutazione dei risultati libera dall'influenza di parametri esterni.

2.1 Workload

Il primo punto da prendere in considerazione nell'iniziare un'attività di ottimizzazione è la definizione di carico di lavoro tipico (*workload*) per l'applicazione di interesse.

Il workload può essere definito come l'input che fornito all'applicazione, meglio di ogni altro, sottopone il sistema ad un carico di lavoro assimilabile a quello che si registrerà in un'esecuzione tipica in un ambiente reale.

Un buon workload deve quindi presentare determinate caratteristiche affinché sia utilizzabile a fini ottimizzativi:

- deve essere *Misurabile*;
- deve essere *Riproducibile*;
- deve essere *Statico* durante le sessioni di ottimizzazione;
- deve essere *Rappresentativo* dell'input tipico;

2.1.1 Misurabilità

Un buon workload dovrebbe avere una metrica quantificabile che possa essere messa in relazione con le prestazioni dell'applicazione.

Ovviamente una simile metrica deve presentarsi come una metrica ben definita e consistente attraverso le varie misurazioni.

E' quindi evidente come per diversi workload, e per diverse applicazioni sarà necessario provvedere alla ricerca di metriche differenti, che meglio modellino l'utilizzo tipico del software.

Nel campo del calcolo scientifico, ad esempio, potrebbe essere scelta come metrica il numero di operazioni in virgola mobile che saranno eseguite. Questa quantità varierà inevitabilmente al variare dell'input che sarà fornito all'applicazione, tuttavia esso potrà essere messo in correlazione con le prestazioni misurate.

2.1.2 Riproducibilità

Il carico di lavoro deve essere necessariamente riproducibile e consistente, quindi ripetute esecuzioni della medesima applicazione con il medesimo workload devono produrre gli stessi risultati, o ad essi assimilabili, in termini sia di performance sia di output.

Nel caso di applicazioni che facciano uso intensivo della CPU sarà necessario che si tenga conto delle diverse politiche di gestione della cache nel sistema di interesse, in quanto la applicazioni che richiedono una grande capacità di calcolo sono particolarmente sensibili alle fluttuazioni introdotte da una gerarchia di memoria. Allo stesso modo nel caso si faccia pesantemente uso dell'input da un dispositivo di memoria secondaria bisognerà tener conto della possibilità che richieste che si andranno a generare in esecuzioni consecutive potrebbero essere soddisfatte, non attraverso l'effettiva lettura da disco, ma dalla lettura da memoria principale che a discrezione del sistema operativo può svolgere le funzioni di buffer per l'accesso a disco.

In entrambi i casi si dovranno prendere contromisure al fine di evitare l'ottenimento di misure non riproducibili; due strategie molto comuni sono:

- il riavvio, nel caso che interessi solo l'effettivo tempo della prima esecuzione;
- l'eliminazione della prima misura, nel caso che la misura di interesse siano le performance a regime;

Nel caso si cerchi di ottenere la misurazione a prescindere dalle memorie cache sarà sufficiente eseguire altri programmi tra due test successivi al fine di ottenerne lo svuotamento della medesima.

2.1.3 Staticità

Il workload deve rimanere stabile durante tutto il periodo durante il cui quale ci si occuperà delle prestazioni del software, al fine ottenere sia un calibramento adeguato degli strumenti di profiling, sia per ottenere misurazioni affidabili delle effettive performance del sistema.

Consideriamo ad esempio il workload per un applicazione di posta elettronica.

Per quanto visto fino ad ora il carico di lavoro deve essere statico e riproducibile, tuttavia una

applicazione di posta sperimenta il massimo carico circa verso le ore 8:00, quanto migliaia di utenti accedono al mail server in lettura, piuttosto che alle 13:00 quando l'attività principale consiste nella lettura e nell'invio di poche email.

In questo caso, un workload che copra il periodo dalle 8:00 alle 13:00 non è statico, quindi se l'obiettivo è l'ottimizzazione dell'invio delle email sarà necessario considerare il carico di lavoro generato verso le 13:00, d'altra parte se l'obiettivo è l'ottimizzazione della fase di accesso al sistema il workload dovrà rappresentare l'effettivo carico di lavoro che si genera verso le 8:00.

2.1.4 Rappresentatività

Ovviamente il carico di lavoro che si utilizzerà per valutare le performance dovrà impiegare attivamente le componenti del sistema che in un'esecuzione tipica vengono più sollecitate.

Nel caso di un web server un carico di lavoro che comprenda un qualsiasi numero di accesso alla medesima pagina non è chiaramente un buon carico di lavoro, in quanto non rappresenta un utilizzo tipico di un web server.

Al contrario dell'esempio precedente tuttavia, una simile considerazione può risultare errata nel caso in cui un'utilizzo tipico preveda per l'appunto un frequente accesso ad una sola pagina web, come ad esempio nel caso di un motore di ricerca.

Bisogna quindi generare delle condizioni di esercizio che rispecchino non solo gli eventi tipici di un'esecuzione, ma anche la successione degli eventi che si registra in un'ambiente reale, al fine di evitare che fenomeni di secondo piano, come ad esempio l'esistenza di una particolare ottimizzazione per un singolo caso, incida in maniera errata nella valutazione delle performance. Da quanto visto finora risulta chiaro il ruolo fondamentale che gioca l'identificazione di un'adeguato carico di lavoro nell'ottimizzazione.

2.2 Closed loop cycle

Stabilito un workload che rispetti le caratteristiche appena citate è necessario stabilire una metodologia per l'ottimizzazione da utilizzare durante il performance tuning.

Il *closed loop cycle* è una metodologia di ottimizzazione che deve essere applicato ad ogni stadio del processo di ottimizzazione. In uno scenario ottimizzativo tipico si andrà ad applicare più volte questo approccio al fine di ottenere risultati significativi.

Il closed loop cycle può essere suddiviso in 5 parti fortemente intercorrelate:

1. Acquisizione dei dati sulle prestazioni;
2. Analisi dei dati acquisiti ed identificazione delle problematiche;
3. Generazione di approcci alternativi;
4. Implementazione delle alternative prodotte;
5. Verifica dei risultati;

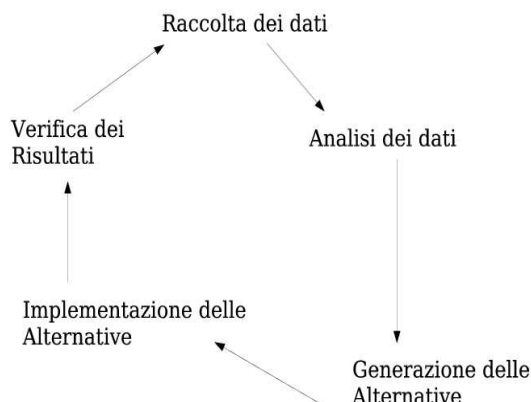


Figura 2.1: Le fasi del closed loop cycle

2.2.1 Acquisizione dei dati

Il primo punto per la risoluzione di un problema di prestazioni è raccogliere una misurazione di base, intesa come la misurazione delle prestazioni a fronte di una corretta esecuzione del modulo software sui cui si andranno ad applicare le ottimizzazioni.

Questa misura definirà lo standard con cui si andranno a verificare tutte le soluzioni implementative che possono offrire un miglioramento in termini prestazioni.

E' ovviamente indispensabile che questa misurazione sia scelta all'inizio della fase di ottimizzazione e che sia il più possibile accurata.

Naturalmente bisognerà dotarsi di strumenti che garantiscano una valutazione precisa e stabile delle effettive performance del sistema.

Al momento della redazione di questa tesi i due strumenti per la valutazione delle performance più utilizzati sono VTune¹ di Intel ed il progetto Open Source noto come Valgrind².

Intel VTUNE

VTune è un software appartenente alla categoria dei profiler a livello di sistema, è in grado sfruttando le funzionalità del sistema operativo e dell'hardware sottostante per valutare le prestazioni di un'applicazione escludendo dal computo l'overhead introdotto dal profiler stesso.

VTune utilizza gli interrupt del processore per la raccolta di dati campione (sampling), ot-

¹<http://www.intel.com>

²<http://valgrind.kde.org>

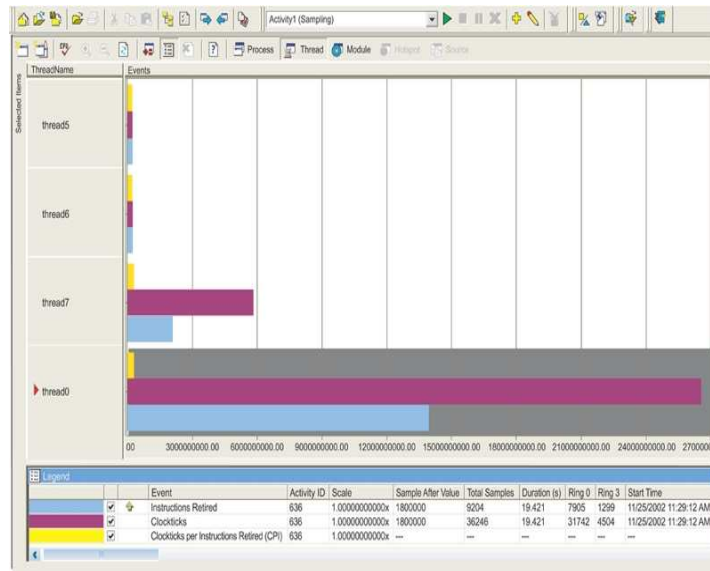
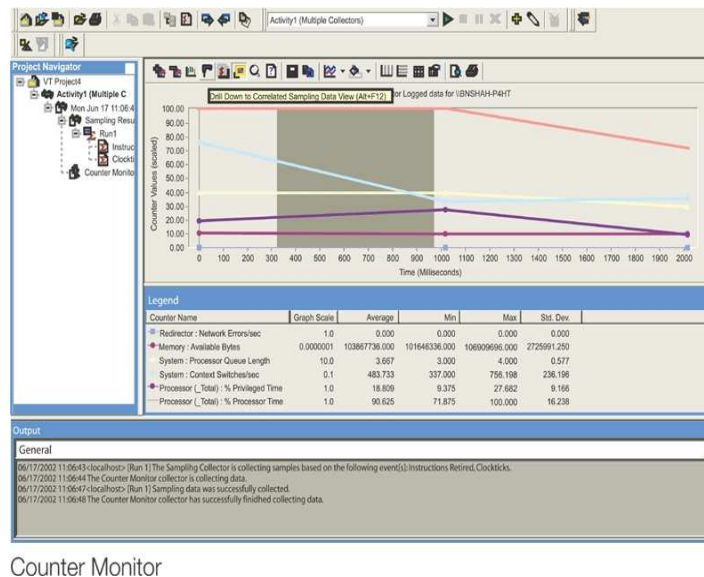


Figura 2.2: La fase di campionatura con Intel VTune

tenuti anche attraverso l'utilizzo di dati statistici. Le caratteristiche chiave di VTUNE sono le seguenti:

- **Campionatura per eventi**
Grazie a questa funzionalità, che consiste nel raccogliere i dati relativi al sistema è possibile ridurre la quantità di dati da analizzare senza trascurare i punti critici del sistema software. E' infatti possibili acquisire i dati presenti al raggiungimento di un certo numero di eventi, come ad esempio il numero di cache miss, o di context switch.
- **Grafico delle chiamate**
Attraverso lo studio dell'andamento dello stack, VTune realizza un grafico rappresentante le interconnessioni esistenti tra le varie funzioni che costituiscono il sistema. La possibilità di ottenere il tempo impiegato all'interno di ogni singola funzione permette di identificare immediatamente le funzioni che costituiscono il collo di bottiglia del software, al fine di concentrare su quest'ultimo la ricerca di ottimizzazioni.
- **Completa portabilità su sistemi Windows - Linux**
- **Counter monitor**
Con l'utilizzo di alcune tecnologie prese in prestito dai debugger, Intel VTUNE utilizza una serie di interrupt specifici al fine di raccogliere in maniera precisa tutta una serie di dati di particolare interesse nel campo delle ottimizzazioni, come la quantità di memoria utilizzata, il numero di accessi a disco, e tutta una serie di parametri a livello di microprocessore ottenuti anche con l'utilizzo di apposite funzionalità esposte dai microprocessori.



Counter Monitor

Figura 2.3: Alcuni counter monitor su Intel VTUNE

Valgrind

Valgrind è uno strumento estremamente flessibile per il debugging ed il profiling di eseguibili x86 su sistema operativo Linux. Lo strumento consiste di un nucleo che fornisce un'emulazione di una moderna CPU x86, su cui sono stati sviluppati della comunità open source una serie di plugins che forniscono funzionalità aggiuntive di alto livello.

Tra i plugins degno di nota è Cachegrind che rappresenta un strumento innovativo fornendo al programmatore la possibilità di ottenere una valutazione del comportamento della cache.

Per quanto sia un progetto recente Valgrind è stato da subito impiegato da numerosi progetti tra cui il NASA Mars Exploration Rover ³ ed il noto browser web Opera⁴.

Inoltre Valgrind fornisce anche un supporto per il debugging, in particolare orientato all'identificazione delle problematiche relative alla gestione della memoria dinamica.

2.2.2 Analisi dei dati

Il secondo passo nell'approccio del closed loop cycle è l'analisi dei dati raccolti al fine di identificare i punti critici nel software che generano le maggiori problematiche in termini di prestazioni.

Supponiamo che tra i dati raccolti sia possibile derivare un uso percentuale del processore pari ad un 25% con un alto numero di trasferimenti verso il disco, in un simile caso è immediato identificare che ci troviamo di fronte ad un'applicazione che presenta un collo di bottiglia nell'I/O.

In questo caso sarà essenziale stabilire come risolvere questa problematica.

³<http://marsrovers.jpl.nasa.gov/home/>

⁴<http://www.opera.com>

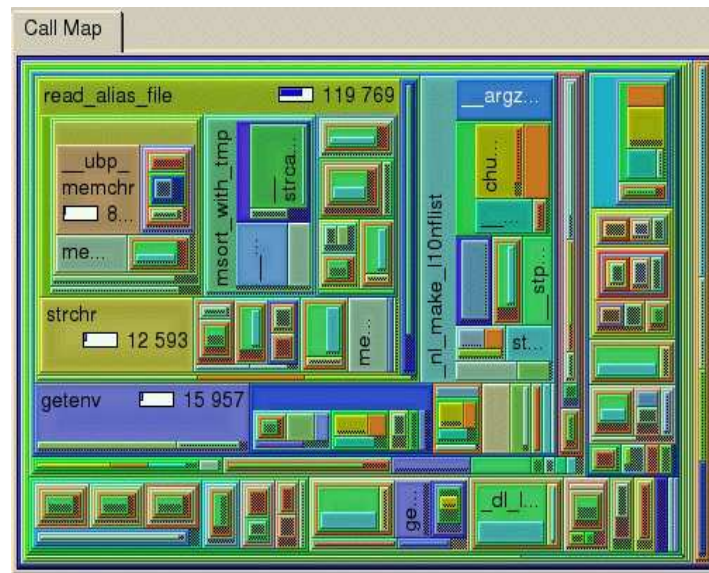


Figura 2.4: Il grafico delle chiamate in Valgrind

In generale un'applicazione potrebbe anche essere rallentata dall'insufficienza dell'hardware a disposizione, in particolare alcuni parametri sono indicativi dell'attuale stato del sistema in particolare:

- L'idle di CPU non deve essere costantemente a 0. (CPU bound)
- La coda di run-queue non deve superare 1 per lungo tempo. (CPU bound)
- Il tempo di System di CPU deve essere in genere minore del tempo User. (controllare la tipologia di attività)
- I virtual fault di paginazione dovrebbero essere limitati. (Memory bound)
- La principale attività di paginazione deve essere di page-in. (Memory bound)
- L'attività del page stealer, il cui compito è garantire la disponibilità di almeno una pagina libera in memoria principale, deve essere limitata. (Memory bound)
- Non deve essere presente swapping. (Memory bound, job mix)
- Le code sui dischi debbono essere vicine ad 1. (IO bound)
- I dischi debbono essere utilizzati in maniera bilanciata. (bilanciamento IO)
- I tempi di accesso ai dischi debbono essere vicini ai tempi medi di accesso dichiarati dal costruttore.

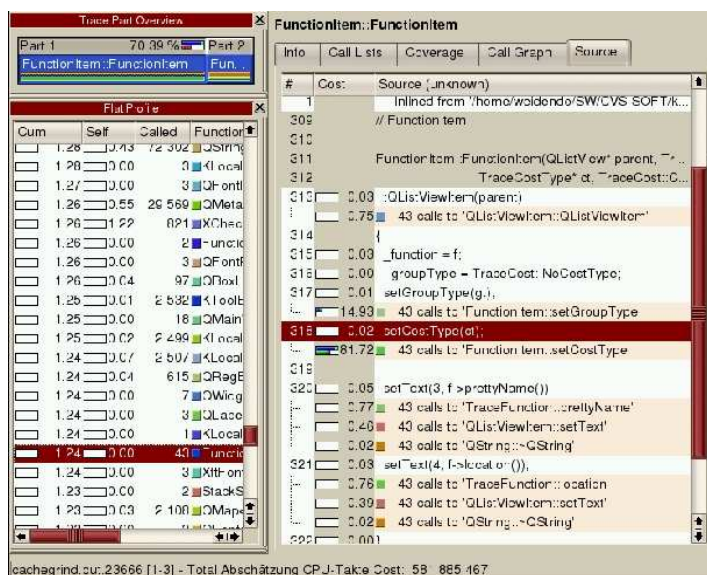


Figura 2.5: Valutazione del codice in Valgrind

Per i parametri che sono stati appena citati si può operare anche a livello di sistema, non operando necessariamente direttamente sul software; in particolare si può intervenire efficacemente su questi performance bug attraverso

- La riduzione delle dimensioni e della complessità del kernel;
- La creazione del corretto job mix da presentare alla macchina;
- La definizione corretta delle aree di paginazione e di swapping;
- Il bilanciamento dei carichi sui dischi;
- La diminuzione del seeking dei dischi;
- Il tuning sui parametri di sistema (paginazione, numero massimo degli utenti);

2.2.3 Generazione di approcci alternativi

Ovviamente molto spesso un approccio a livello di sistema, per quanto efficace non può riuscire a soddisfare le esigenze prestazionali che deve soddisfare il sistema.

Il passo immediatamente successivo è la ricerca di possibili alternative alle porzioni di codice che rappresentano il collo di bottiglia della nostra applicazione.

Nel caso precedente si potrebbe cercare un possibile approccio alternativo, sia in termini di eventuali miglioramenti hardware, sia in termini di riscrittura del codice di I/O al fine di implementare un livello di buffering adatto alle esigenze del software.

2.2.4 Implementazione delle alternative

Dopo aver scelto le possibili strade che possono offrire miglioramenti in termini di prestazioni, sarà necessario provvedere alla loro implementazione, dando particolare importanza a non implementare contemporaneamente più di un nuovo approccio per evitare che i miglioramenti forniti da un nuovo approccio siano negati dalle cattive prestazioni di un'altra alternativa.

Ovviamente si dovrà valutare le scelte implementative in un'ottica più vasta considerando come le scelte fatte andranno poi a ripercuotersi su tutta l'esecuzione del sistema.

2.2.5 Verifica dei risultati

Al termine dell'applicazione delle ottimizzazioni si dovrà procedere alla verifica dei risultati ottenuti, accertandosi in prima battuta della correttezza delle risposte fornite dal sistema.

Bisognerà in seguito verificare l'effettivo miglioramento delle performance, ed accertarsi che esso presenti un reale miglioramento in termini di prestazioni.

Nel caso i miglioramenti siano al di sotto delle aspettative o del tutto assenti si dovrà reiterare ripetutamente i vari stadi del processo di ottimizzazione, da quest'ultimo passo deriva per l'appunto il nome Closed Loop Cycle.

Capitolo 3

Hardware Bottleneck

La generazione attuale dei microprocessori ha prestazioni, rispetto ai predecessori, superiori di svariati ordini di grandezza. Questi radicali aumenti di performance, velocità e numero di transistor possono sembrare slegati ad un osservatore casuale.

In realtà, sebbene il progetto attuale dei microprocessori vari enormemente da caso a caso, possono essere individuate delle linee guida comuni. Ogni processore infatti effettua una generazione degli indirizzi, contiene unità logico-aritmetiche, possiede dei file dei registri, e ha un'interfaccia di sistema. Molti hanno una o più cache on-chip, un TLB (translation lookaside buffer), e praticamente tutte le architetture correnti possiedono unità floating point on-chip. Per effettuare queste funzioni di base sono state implementate differenti tecniche di progetto, che però devono risolvere gli stessi problemi. Pertanto in questa sede analizzeremo per primi i più comuni problemi affrontati da tutti i progettisti di CPU, per presentare alcune tecniche utilizzate per superare questi problemi.

3.1 I limiti dei processori moderni

3.1.1 Le latenze della memoria e della cache di secondo livello

I primi microprocessori effettuavano il fetch delle istruzioni direttamente dalla memoria; in questo modo, dopo aver inviato una richiesta di dati, il processore doveva attendere un tempo molto lungo prima che i dati arrivassero, impedendo così alla CPU di operare in modo efficiente alla velocità per la quale era stata progettata.

L'implementazione della cache secondaria off-chip ha aiutato ad alleviare questo problema. Una memoria cache è solitamente di dimensioni limitate (normalmente da 32 a 1024 Kbyte), e contiene un blocco di indirizzi di memoria comprendenti una piccola sezione della memoria principale. La memoria cache fornisce un accesso più rapido, e può spedire i dati al processore ad una frequenza maggiore della memoria principale.

I sistemi di memoria cache on-chip possono poi aumentare ancora le prestazioni, poiché permettono il completamento di un accesso in un singolo ciclo di clock. L'aumento di prestazioni fornito dalla cache di primo livello ha indotto i progettisti ad aumentarne sempre più le dimensioni causando in molte architetture un notevole aumento dello spazio dedicato al progetto

della cache; in molte implementazioni attuali la cache occupa più dell'80% della superficie del die.

Le performance raggiungono il valore massimo quando l'applicazione può essere eseguita totalmente dentro la cache. Tuttavia, quando l'applicazione, come spesso avviene, è troppo grande per stare nella cache, le performance diminuiscono in modo significativo. La figura mostra la relazione esistente fra le prestazioni e la dimensione di un'applicazione.

La cache di primo livello contiene un range di indirizzi che comprende un sottoinsieme di

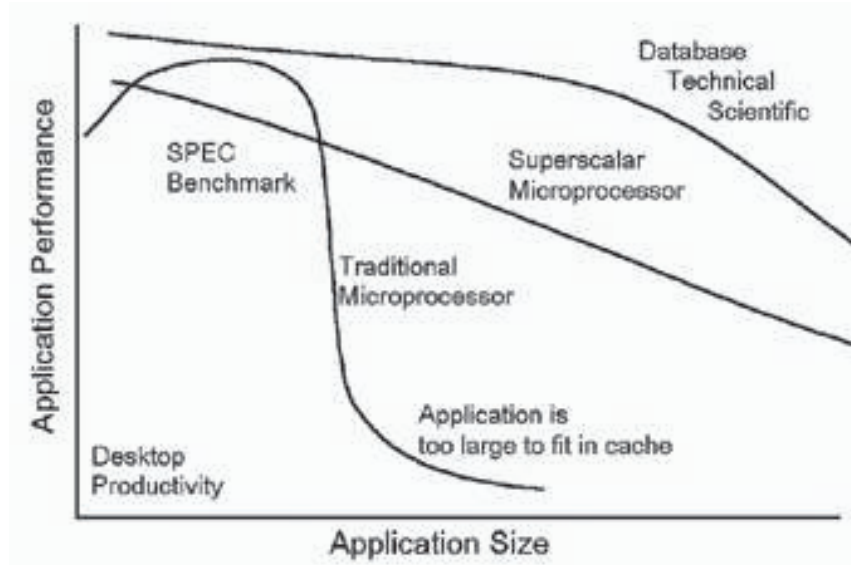


Figura 3.1: La relazione esistente fra le prestazioni e la dimensione di un'applicazione

quelli presenti nella cache secondaria, che a loro volta sono un sottoinsieme degli indirizzi presenti nella memoria principale. Se la cache on-chip ha il vantaggio di aumentare le prestazioni, la tecnologia attuale e il numero di transistori disponibili ne limitano la dimensione massima. Attualmente 64 Kbyte (32 per i dati e 32 per le istruzioni) rappresentano il limite della cache di primo livello, e la sua implementazione richiede molti milioni di transistor.

Questi fattori limitanti della cache on-chip aumentano l'importanza della cache di secondo livello, dove la dimensione della cache è limitata solo dal mercato in cui il prodotto sarà collocato. Il tempo di accesso di molti dei dispositivi RAM attualmente disponibili è piuttosto elevato rispetto al tempo di clock del processore, e ciò forza i progettisti a trovare soluzioni alternative. L'interleaving della cache è uno degli espedienti utilizzati, dato che permette la sovrapposizione delle richieste in memoria da parte del processore. Questa tecnica si può applicare sia alla memoria principale che alla cache; il più comune è l'interleaving a due o a quattro vie. Questo perché aumentando l'interleaving si riesce a nascondere gran parte del tempo di accesso, ma aumenta considerevolmente anche la complessità richiesta per supportarlo.

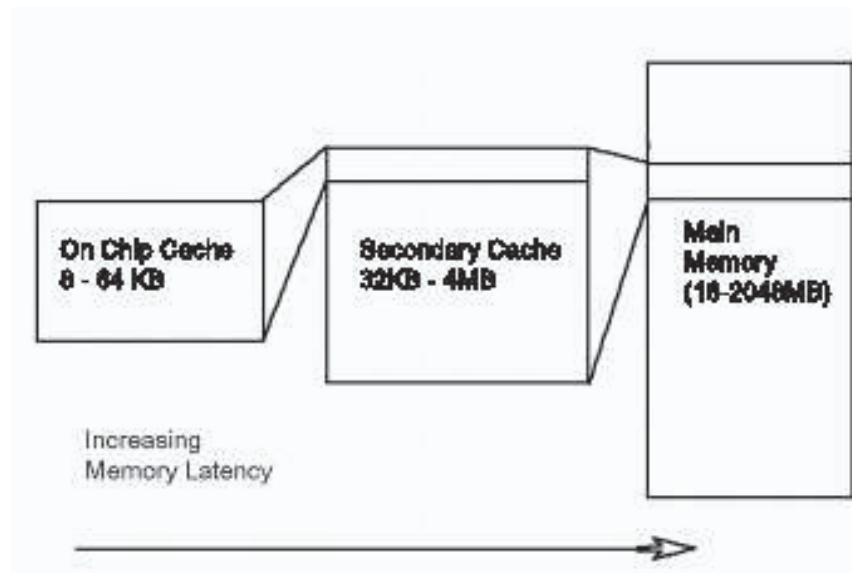


Figura 3.2: La figura mostra la relazione fra le cache in un tipico sistema

3.1.2 Le dipendenze dei dati

In un programma, le istruzioni vengono caricate dalla cache istruzioni, decodificate ed eseguite. Il dato corrispondente è spesso caricato da un registro, manipolato in un'ALU, e il risultato viene rimesso nello stesso registro o in un altro.

Se l'istruzione successiva della sequenza richiede il risultato dell'istruzione precedente per poter essere eseguita, avviene una dipendenza di dati. Per quelle istruzioni che richiedono molti cicli per essere completate, ci può essere un impatto notevole sulle prestazioni nel caso vi siano delle dipendenze. Alcune di queste possono essere eliminate semplicemente riarrangiando il programma in modo che il risultato di una data istruzione non venga utilizzato dalle prime istruzioni seguenti.

Un modo per alleviare il problema delle dipendenze dei dati è utilizzare l'esecuzione fuori ordine con ridenominazione dei registri.

3.1.3 Diramazioni (salti condizionali)

Tutti i programmi di computer contengono diramazioni (branch). Alcune sono non condizionali, cioè il flusso del programma viene interrotto non appena l'istruzione di branch viene eseguita; altre sono condizionali, cioè il branch viene eseguito solo se certe condizioni vengono soddisfatte. Le interruzioni del flusso del programma sono presenti in tutti i software, e l'hardware può solo cercare di adeguarsi ai branch nel modo più efficiente possibile.

Quando viene presa una diramazione, il nuovo indirizzo al quale il programma deve riprendere l'esecuzione può essere nella cache secondaria oppure no; a seconda di dove è situato il nuovo blocco di istruzioni la latenza aumenta o diminuisce. Poiché il tempo di accesso della memoria principale e della cache secondaria sono molto maggiori del tempo di accesso della cache on

chip, il branching spesso degrada le prestazioni del processore.

Questo problema è ancora più importante nelle macchine superscalari, dove ad ogni ciclo vengono eseguite più istruzioni. In un certo momento, infatti, a seconda della dimensione della pipeline, numerose istruzioni possono essere in vari stadi di esecuzione; quando viene presa una diramazione, non si conoscono il numero di cicli che saranno necessari per la sua esecuzione.

L'implementazione del branching è un importante problema architetturale. Per migliorare le prestazioni molte architetture attuali incorporano una circuiteria per la predizione delle diramazioni, la quale può essere implementata in vari modi.

3.2 Risposte hardware alle problematiche dei processori

3.2.1 Riduzione della latenza delle memorie

Interfaccia a larga banda con la cache secondaria

Un'interfaccia ideale cache secondaria-processore dovrebbe essere sempre in grado di ricevere una richiesta di dati dal processore e di soddisfare questa richiesta nel ciclo di clock successivo; ci si riferisce a questo comportamento come uno *zero wait state*. Per progettare una cache secondaria che sia in grado di raggiungere questo tipo di prestazioni, l'interfaccia deve essere studiata in modo da riuscire a trasmettere i dati sempre alla massima velocità possibile.

Il bus dati e il bus indirizzi per molti processori costituiscono l'interfaccia con l'intero sistema: il processore può accedere a qualsiasi tipo di dispositivo in ogni momento.

Quando avviene un cache miss, viene spedito sul bus un indirizzo, e si accede alla cache secondaria, trasferendo i dati richiesti alla cache on-chip.

Se avviene un cache miss in un bus di sistema condiviso, e il processore sta utilizzando il bus esterno per leggere o scrivere su qualche altro dispositivo, l'accesso alla cache secondaria deve attendere finché non si sono liberati i bus dati e indirizzi; ciò può richiedere molti cicli di clock, a seconda della periferica a cui si accede.

In un sistema con bus dedicato i bus dei dati, degli indirizzi e di controllo per la cache secondaria sono separati dai bus che si interfacciano con il resto del sistema. In questo modo gli accessi alla cache secondaria in caso di miss vengono garantiti sempre, qualsiasi cosa stia facendo il sistema.

Accesso a blocchi

Quando avviene un cache miss on-chip, esiste un numero di byte, solitamente programmabile, che viene trasferito sul bus ogni volta che si accede alla cache secondaria. Questo numero è la dimensione di una linea di cache; per le architetture attuali la dimensione classica è di 32 byte.

Il numero di accessi richiesto per effettuare il riempimento di una linea dipende dalla dimensione del bus di dati esterno del processore. Per esempio, un processore con bus dati a 64 bit

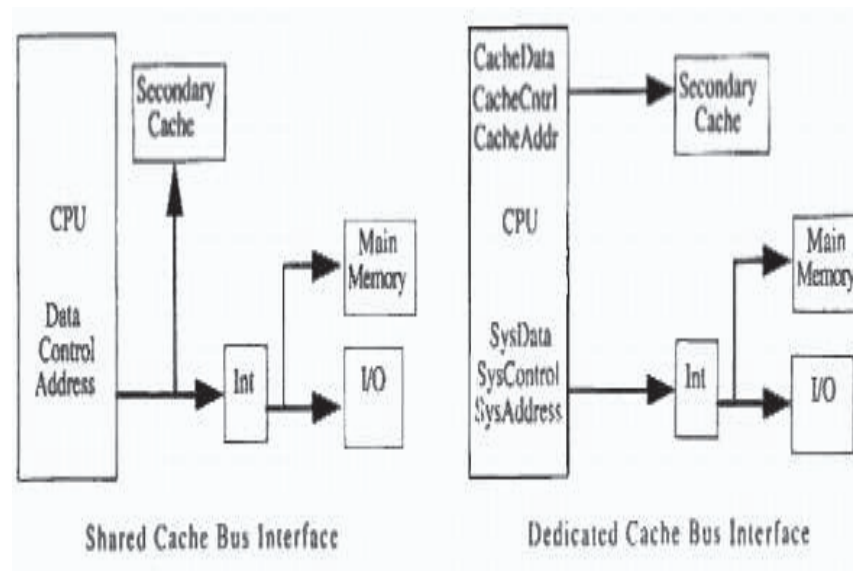


Figura 3.3: un diagramma a blocchi di un'interfaccia condivisa e di una separata

che si interfaccia con una memoria a 64 bit impiegherà quattro accessi alla cache secondaria per riempire una linea di cache da 32 byte. Per completare tutto ciò il processore deve generare quattro indirizzi separati e guidare ognuno sul bus indirizzi esterno tramite appropriati segnali di controllo.

Utilizzando l'accesso a blocchi il processore deve invece generare solo l'indirizzo iniziale della sequenza, mentre gli altri tre indirizzi vengono generati dalla logica di controllo della cache.

Interleaving

L'interleaving è una tecnica di progetto utilizzata per aumentare la larghezza di banda della memoria; tale tecnica può essere applicata sia alla cache secondaria che alla memoria principale.

Il sistema di memoria più semplice è quello con un solo banco di memoria. Quando si accede a tale banco, deve trascorrere un certo intervallo temporale prima di un secondo accesso; questo intervallo dipende sia dal progetto del sistema sia dalla velocità dei dispositivi di memoria utilizzati. Se sono presenti più banchi, allora l'accesso ai banchi può essere sovrapposto. L'abilità di sovrapporre tali accessi aiuta a nascondere le latenze della memoria e diventa sempre più importante al crescere della dimensione dei dati richiesti.

Una tipica memoria con interleaving consiste in banchi pari e dispari. Per esempio, il processore richiede dei dati ad un indirizzo pari, così il controller della memoria inizia un ciclo al banco pari. Una volta che l'indirizzo è stato raggiunto dalla logica di controllo della memoria, il processore può generare un nuovo indirizzo, il più delle volte nel ciclo di clock successivo. Se il nuovo indirizzo si riferisce ad un banco dispari, l'accesso alla memoria può iniziare immediatamente; in questo modo, non appena il banco pari ha completato l'operazione, il banco dispari è già pronto a fornire il dato. Pertanto più a lungo si riesce a minimizzare gli accessi

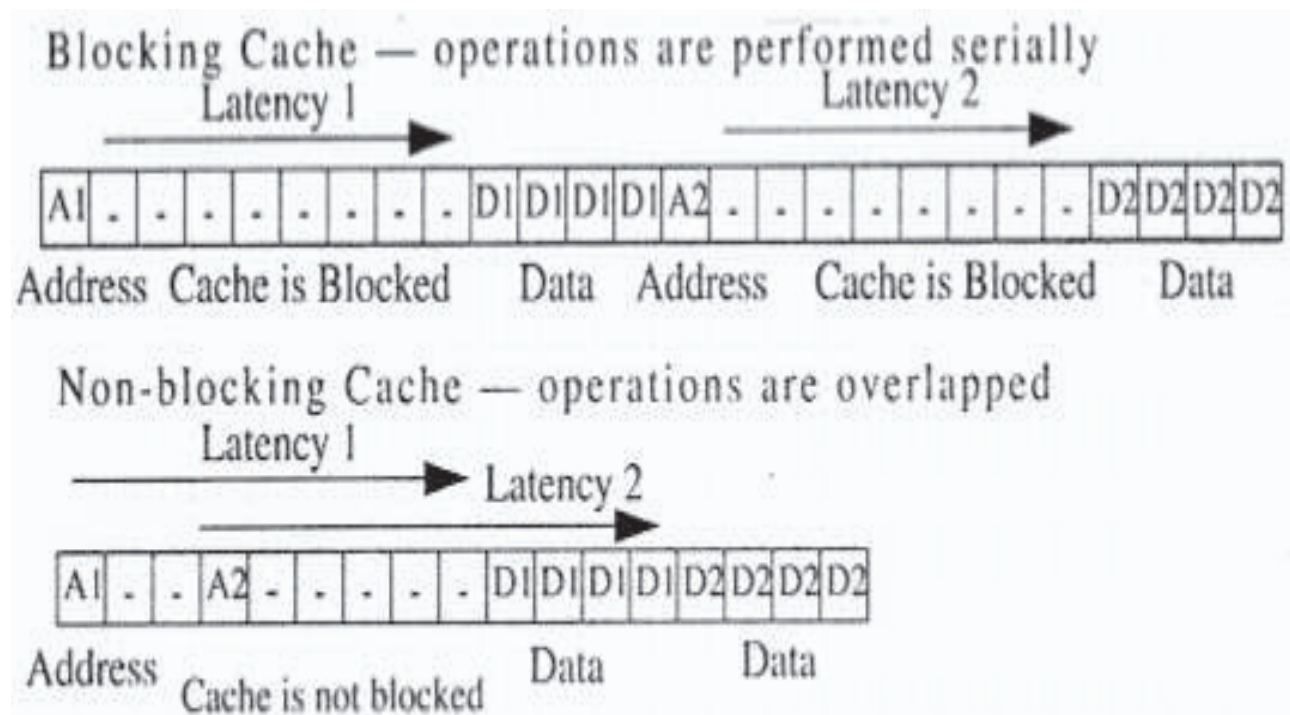


Figura 3.4: Un esempio di come le cache bloccanti e non bloccanti reagiscono a miss multipli

sequenziali, più ci si avvicina alle prestazioni di tipo zero wait state.

I più comuni sistemi di memoria con interleaving sono quelli a due e a quattro vie; il numero di banchi e la larghezza di banda di ciascuno sono spesso determinati dal processore.

Cache non bloccante

In una tipica implementazione il processore agisce sulla cache finché avviene un cache miss. A questo punto, trascorre un certo numero di cicli prima che i dati vengano riportati nella cache on-chip, permettendo la ripresa dell'esecuzione. Questo tipo di implementazione è detto bloccante, dato che non si può accedere alla cache finché non viene risolto il cache miss.

La cache di tipo non bloccante invece, permette accessi consecutivi anche in caso di cache miss. In questo caso, per aumentare le prestazioni globali del sistema, è cruciale localizzare il prima possibile i miss e effettuare i passi necessari per risolverli.

Prefetch

Il prefetching è una tecnica con cui il processore può richiedere un blocco di cache prima del momento in cui è effettivamente necessario. L'istruzione di prefetch deve essere integrata nel set di istruzioni, e deve esserci un appropriato hardware per eseguirla.

Per esempio, supponiamo che il compilatore stia avanzando in modo sequenziale attraverso

un segmento di codice. Il compilatore può fare l'ipotesi che questa sequenza continuerà oltre il range degli indirizzi disponibili nella cache di primo livello, e può richiedere un'istruzione di prefetch, la quale carica il blocco di istruzioni successivo nella cache di secondo livello. Quindi, quando il processore richiede la sequenza successiva, questa può essere eseguita ad una frequenza maggiore; se per qualche motivo tale blocco non è necessario, l'area nella cache secondaria viene semplicemente sovrascritta da altre istruzioni.

Il prefetching permette quindi al compilatore di anticipare la necessità di un dato blocco, e di piazzarlo il più possibile vicino alla CPU (località spaziale).

3.2.2 Dipendenze fra i dati

Ridenominazione dei registri

La ridenominazione dei registri distingue fra registri logici e registri fisici; i registri logici sono mappati dinamicamente nei registri fisici attraverso apposite tabelle che vengono aggiornate ogni volta che un'istruzione viene decodificata. Ogni nuovo risultato viene scritto in un registro fisico; tuttavia, il contenuto precedente di ogni registro logico viene salvato, e può essere recuperato nel caso l'istruzione debba essere abortita a causa di un'eccezione o di una previsione di salto non corretta.

Mentre il processore esegue le istruzioni, vengono generati moltissimi risultati temporanei, i quali sono immagazzinati in appositi registri. I valori temporanei diventano permanenti quando la corrispondente istruzione viene graduata, cioè quando tutte le istruzioni precedenti sono state completate con successo nell'ordine del programma.

Il programmatore è consapevole dell'esistenza dei soli registri logici, mentre l'implementazione dei registri fisici è nascosta.

La ridenominazione dei registri semplifica il controllo delle dipendenze fra i dati. In una macchina che può eseguire istruzioni fuori ordine, i numeri dei registri logici possono diventare ambigui, poiché ad uno stesso registro può essere assegnata una successione di valori diversi. Ma dato che i numeri dei registri fisici identificano in modo unico ogni risultato, il controllo delle dipendenze non risulta più ambiguo.

Esecuzione fuori ordine

In un tipico processore che esegue le istruzioni in ordine, ogni istruzione dipende dall'istruzione precedente che produce i suoi operandi, e l'esecuzione non può iniziare finché questi operandi non diventano validi. Se gli operandi richiesti per eseguire una data istruzione non sono validi, la pipeline stalla finché tali operandi non diventano disponibili. Poiché le istruzioni vengono eseguite rispettando l'ordine del programma, solitamente gli stalli ritardano tutte le istruzioni seguenti.

In una macchina superscalare *in-order*, dove vengono eseguite più istruzioni per ciclo, varie istruzioni consecutive possono iniziare simultaneamente l'esecuzione solo se tutti i loro corrispondenti operandi sono validi, altrimenti il processore va in stallo.

In una macchina superscalare *out-of-order*, ogni istruzione può iniziare la sua esecuzione non appena gli operandi necessari diventano disponibili, senza riguardo per la sequenza originaria.

L'hardware effettivamente riarrangia l'ordine delle istruzioni per tenere sempre occupate le varie unità di esecuzione. Questo processo viene chiamato *dynamic issuing*.

3.2.3 Predizione dei salti

BPU (Branch Prediction Unit)

Come già detto, le diramazioni interrompono il flusso della pipeline; pertanto, per minimizzare il numero di interruzioni, sono necessari degli schemi di branch prediction. Le diramazioni accadono frequentemente, in media ogni sei istruzioni; in un'architettura superscalare, dove vengono eseguite anche quattro istruzioni per ciclo, la predizione di tali diramazioni diventa importante.

Molti schemi di predizione utilizzano degli algoritmi che tengono traccia del comportamento delle diramazioni l'ultima volta che sono state eseguite. Per esempio, se il circuito che memorizza tali comportamenti mostra che la volta precedente un'istruzione ha preso la diramazione, allora si fa l'ipotesi che questa venga presa ancora. Un'implementazione hardware di questa assunzione significa che il programma invierà allo stesso indirizzo tutte le successive diramazioni. La pipeline ora contiene un'istruzione di salto condizionale e altre istruzioni successive ma in quel momento non si sa se tali istruzioni verranno eseguite; infatti se la diramazione non è stata predetta correttamente, le istruzioni nella pipeline devono essere abortite.

Molte architetture implementano un branch stack nel quale vengono salvati gli indirizzi alternativi. Se si prevede che la diramazione non sarà presa, viene salvato l'indirizzo dell'istruzione di branch; in caso contrario viene salvato l'indirizzo immediatamente seguente a tale istruzione.

Capitolo 4

Source Level Optimizations

I pionieri dell'informatica predissero che i programmatori avrebbero desiderato disporre di quantità illimitate di memoria veloce. La previsione si è avverata, e per esaudire questo desiderio, si è trovata una soluzione economica: una *gerarchia di memorie*, che sfrutta la località e il rapporto costo/prestazioni delle tecnologie di memoria. Il *principio di località* afferma che la maggior parte dei programmatori non accede nè al codice nè ai dati in modo lineare, questa osservazione è altresì espressa anche dalla cosiddetta legge del 80-20, che può essere esemplificata come “Il 20% del codice utilizza l'80 percento delle risorse”. Questa legge non si applica esclusivamente in campo informatico infatti per analogia in qualunque sistema sono pochi gli elementi rilevanti ai fini del comportamento del sistema.

4.1 Opportunità offerte dalle ottimizzazioni a livello di codice sorgente

Le ottimizzazioni a livello di linguaggio C possono fornire un buon incremento delle performance che in alcuni casi può anche risolvere le problematiche prestazionali individuate nel modulo software.

Ovviamente una simile possibilità è necessariamente legata alla tipologia del codice preso in considerazione. Sostanzialmente le opportunità offerte dal linguaggio C possono essere suddivise in due categorie

- Ottimizzazioni proprie dei singoli compilatori
- Ottimizzazioni implementate dai programmatori

4.1.1 Ottimizzazioni dei compilatori

I compilatori rappresentano un ottimo strumento per l'ottimizzazione, in particolare per progetti estremamente complessi, dove il punto critico in termini di prestazioni, varia al variare dell'input, e che quindi richiederebbe un lungo lavoro di ispezione del codice sorgente. A livello teorico un compilatore dovrebbe essere in grado di ottenere il miglior eseguibile possibile.

Tuttavia questa prospettiva è molto lontana dall'essere realizzabile, per una serie di fattori che riducono fortemente le capacità dei compilatori.

Consideriamo ad esempio la piattaforma x86, essa rappresenta un'architettura ben precisa e molto diffusa, sarebbe quindi un target ideale per un compilatore fortemente ottimizzante.

Tuttavia ogni generazione di processore fornisce caratteristiche nuove rispetto alle generazioni precedenti. Consideriamo ad esempio l'introduzione della tecnologia SSE3 presente nei processori Prescott in uscita in questi mesi, si tratta di una caratteristica che può fornire un miglioramento in termini di performance intorno al 400% per determinate operazioni.

Ad ogni modo, al momento della stesura di questa tesi, nessun compilatore è in grado di vettorizzare il codice per l'utilizzo di queste nuove funzionalità.

Naturalmente un compilatore è in grado di fornire ottimizzazioni molto importanti e di grande utilità in termini di prestazioni, come ad esempio una gestione ritardata dello stack. Con una simile ottimizzazione il compilatore evita di ripulire lo stack dopo ogni singola chiamata ad una funzione, se ciò non si rivela fondamentale per la corretta esecuzione del codice compilato.

4.1.2 Ottimizzazioni a livello di programmazione C

Naturalmente un programmatore esperto può fornire miglioramenti in termini prestazionali che superino di gran lunga le ottimizzazioni eseguibili da un compilatore. Un programmatore esperto può ad esempio suddividere il problema computazionale e distribuirlo tra le varie unità di calcolo in un sistema multiprocessore. Inoltre un attento lavoro di analisi del codice può rivelare la possibilità di risolvere a monte il problema, ad esempio riorganizzando il layout dei dati al fine di utilizzare al meglio le risorse della macchina.

Al contrario delle ottimizzazioni applicate dai compilatori queste ottimizzazioni richiedono un gruppo di lavoro specifico, con un'approfondita conoscenza delle possibilità offerte dalle varie architetture, e quindi rappresentano un costo maggiore rispetto alle ottimizzazioni in fase di compilazione, fornendo tuttavia risultati estremamente buoni.

4.2 Applicazione delle ottimizzazioni

4.2.1 Architettura di riferimento

Nell'applicazione delle nostre ottimizzazioni prenderemo come architettura di riferimento il processore AMD Opteron, basato sulla tecnologia Hammer, che amplia la lunga e ricca tradizione di AMD di offrire innovazioni che incontrino le necessità dei clienti. Questa tecnologia è studiata per salvaguardare gli investimenti che le aziende hanno compiuto nel software a 32bit, permettendo loro una transizione verso l'elaborazione a 64bit secondo le proprie necessità e tempistiche. Il processore AMD Opteron è studiato per offrire soluzioni dalle elevate prestazioni nelle più esigenti applicazioni per server e workstation nel segmento Enterprise. Le innovazioni chiave del processore AMD Opteron includono:

1. un controller di memoria integrato, che riduce i colli di bottiglia della memoria;
2. la tecnologia Hypertransport che aumenta le prestazioni globali, riducendo i colli di bottiglia del I/O, aumenta la banda passante e riduce la latenza.

La tecnologia HyperTransport rappresenta un nuovo collegamento point-to-point ad alta velocità, e ad elevate prestazioni, per l'interconnessione di circuiti integrati sulla scheda madre. Questa tecnologia consente di raggiungere velocità significativamente superiori a quelle di un bus PCI con un numero equivalente di piedini.

Sebbene sia principalmente utilizzata nel settore informatico e in quello delle telecomunicazioni, la tecnologia HyperTransport può offrire vantaggi in qualsiasi applicazione che richieda alta velocità, scalabilità e bassa latenza.

Inoltre a differenza dell'architettura IA64 (Itanium e seguenti) si sta diffondendo molto rapidamente, rappresentando allo stato attuale la fascia alta per quanto riguarda i calcolatori desktop¹ e visti i bassi consumi anche sui notebook².

4.2.2 Sistema Operativo di Riferimento

Come Sistema Operativo per l'architettura di riferimento abbiamo scelto Gentoo Linux.³ Gentoo Linux è una speciale distribuzione di Linux che può essere automaticamente ottimizzata e personalizzata per quasi ogni applicazione di cui si abbia bisogno. Performance estreme, configurabilità e una comunità di utenti e sviluppatori di prim'ordine sono tutte caratteristiche del mondo Gentoo.

Grazie ad una tecnologia chiamata Portage, che si richiama molto ai ports dei sistemi *BSD, Gentoo Linux può diventare un server sicuro e ideale, una workstation di sviluppo, un desktop professionale, un computer dedicato al gioco, una soluzione embedded o qualsiasi altra cosa desideriate. Vista la sua illimitata adattabilità, Gentoo Linux viene definita una metadistribuzione.

Cos'è il Portage?

Portage è il cuore di Gentoo Linux e fornisce molte funzioni chiave. Come prima cosa, Portage è il sistema di distribuzione del software per Gentoo Linux. Per avere l'ultima versione del software, basterà digitare il comando: `emerge sync`. Questo comando farà sì che Portage aggiorni il Portage tree locale attraverso Internet. Il Portage tree locale contiene una collezione completa di scripts che possono essere usati da Portage per creare e installare le ultime versioni dei pacchetti di Gentoo. Attualmente si trovano oltre 4000 pacchetti nel Portage tree, con aggiunta quasi quotidiana di nuovi pacchetti.

Portage tiene inoltre aggiornato il sistema. Digitando `emerge -u world` – un solo comando – si ha la sicurezza che tutti i pacchetti nel vostro sistema siano aggiornati automaticamente.

¹<http://www.amd.com/it-it/Processors/ProductInformation/0,,301189484,00.html>

²<http://www.amd.com/it-it/Processors/ProductInformation/0,,301181276,00.html>

³<http://www.gentoo.org>

4.3 Compilatore di Riferimento

Come compilatore per lo svolgimento dei test abbiamo scelto gcc 3.4.0 ⁴ che rappresenta lo stato dell'arte della compilazione su piattaforma x86-64. Attualmente sulla piattaforma x86-64 gli unici compilatori in grado di sfruttare realmente l'architettura sottostante sono i compilatori prodotti dalla Free Software Foundation ⁵ che presentano un buon supporto per l'architettura⁶.

Il famoso compilatore Intel non è in grado, e per scelta aziendale non si prevede che lo sarà mai, di generare codice oggetto per i microprocessori della famiglia AMD64, mentre il compilatore Microsoft per questa piattaforma è stato annunciato ⁷, ma non ancora disponibile. Inoltre Microsoft ha annunciato che esso sarà integrato nel nuovo framework .NET Whidbey che avrà come target di riferimento Windows XP 64bit (attualmente in stato di beta) e la nuova piattaforma Longhorn.

Rispetto alle precedenti versioni di gcc la versione che abbiamo scelto presenta numerosi vantaggi: di particolare interesse per il nostro compito sono stati i miglioramenti alle tecnologia ottimizzativa. Tra le principali feature ricordiamo:

- Un nuovo schema di ottimizzazioni per componenti, che a livelli di ottimizzazione aggressivi, riorganizza il codice all'interno del file oggetto per fornire un miglior allineamento dei dati e del codice in memoria;
- La rimozione delle variabili non utilizzate;
- Il tempi necessari per la compilazione sono stati drasticamente ridotti, riducendo quindi il tempo necessario per il testing delle varie implementazioni.

4.4 Prima Implementazione

La prima implementazione che presenteremo in questa tesi, sarà l'implementazione standard utilizzata attivamente nell'esecuzione del progetto CAD - Computer Aided Detection, in modo da avere a disposizione un metro di paragone funzionante ed utilizzato attivamente. Forniamo quindi il codice di questa implementazione, a cui ci riferiremo d'ora innanzi con il termine di *versione legacy*.

⁴<http://gcc.gnu.org/gcc-3.4>

⁵<http://www.fsf.org/>

⁶<http://gcc.gnu.org/>

⁷http://www.amd.com/us-en/Processors/ProductInformation/0,,3011887968927_78003,00.html

Listato 4.1: Versione legacy

```

double
classify_poly2_optimized (double coef_lin ,
                          double coef_const ,
                          double b ,
                          float *sv ,
                          double *alfa ,
                          unsigned int sv_num ,
                          float *example ,
                          unsigned int features_num)
{
    register double dist = 0.0;
    register double buffer = 0.0;
    unsigned int i;

    for (i = 0; i < sv_num; i++)
    {
        buffer =
            coef_lin * p_scalar (&sv[(i * features_num)] ,
                                example ,
                                features_num)
            + coef_const;
        dist += buffer * buffer * alfa[i];
    }

    dist -= b;

    return dist;
}

```

Non presentiamo integralmente il codice della funzione *p_scalar* vista la semplicità, tuttavia per completezza definiremo la funzione *p_scalar* come la funzione che accetta in input due puntatori ed un numero di scalari presenti nei vettori e restituisce il prodotto scalare tra i due vettori.

Inoltre la funzione presentata viene invocata, nella modalità visibile dal codice, dalla seguente funzione

Listato 4.2: Versione legacy (chiamante)

```

void
rhosvm_call (MODEL * model,           // rhosvm model header
             float *sv,               // support vector
             double *alfa,           // alphas
             unsigned long slave_start, // start mask included
             unsigned long slave_stop, // stop mask included
             float *examples_matrix,  // matrix of features vector
             float *results          // vector of results
        )
{
    unsigned long i, j;
    long features_num, sv_num;
    long row_width;
    float *c;
    float *alpha_f;

    /* set parameters */
    features_num = model->totwords;
    sv_num = model->sv_num;

    row_width = features_num;

    if ((model->kernel_parm.kernel_type == POLY)
        && (model->kernel_parm.poly_degree == 2))
    {
        for (i = slave_start; i <= slave_stop; i++)
        {
            results[i] =
                classify_poly2_optimized (model->kernel_parm.coef_lin,
                                         model->kernel_parm.coef_const, model->b,
                                         sv, alfa, sv_num,
                                         &(examples_matrix[i * row_width]),
                                         features_num);
        }
    }
}

```

Fin dalla prima osservazione si notano i tre punti chiave, dove si riscontrano alcune problemi prestazionali:

- *Il prodotto scalare* che rappresenta il fulcro computazione dell'intera modulo, per la grande quantità di calcoli in virgola mobile che è chiamato a svolgere;
- *L'esistenza di un chiamante* che non fornisce nient'altro che un supporto alla gestione del ciclo for;

- *L'uso seriale* dell'unità di calcolo, che non sfrutta nè il potente bus HyperTransport, nè i due microprocessori di cui è dotata la macchina;

4.5 Ottimizzazioni Applicate

4.5.1 Prodotto Scalare

Nel caso del prodotto scalare non ci possiamo certamente aspettare un forte miglioramento delle performance, vista la difficoltà con cui si può interagire con il microprocessore a livello di codice sorgente, tuttavia abbiamo applicato due importanti ottimizzazioni che fornisco maggiormente interessanti dal punto di vista computazionale.

Loop pipelining

La macchina tradizionale di von Neumann tipo SISD (Single Instruction Single Data) ha un solo flusso d'istruzioni (in pratica, un programma) eseguito da una CPU, ed una memoria che contiene i suoi dati. La prima istruzione è presa dalla memoria e poi eseguita. Poi è prelevata ed eseguita la seconda istruzione.

Tuttavia, anche entro questo modello sequenziale è possibile avere una quantità limitata di parallelismo, per esempio, prelevando una nuova istruzione e cominciando ad eseguirla prima che quella in esecuzione sia finita. In altre parole nella stessa CPU si eseguono contemporaneamente più parti di azioni diverse come in una catena di montaggio, allora basta costruire l'hardware con diverse unità funzionali.

Il parallelismo temporale è dato dalla suddivisione del ciclo di esecuzione di un'istruzione in più fasi, ciascuna delle quali è assegnata ad una differente unità funzionale.

Consideriamo una CPU ipotetica con cinque unità funzionali:

Durante il primo intervallo di tempo, l'istruzione è presa dalla memoria da P1. Nel secondo intervallo di tempo, l'istruzione è passata a P2 per essere analizzata, mentre P1 prende un'altra istruzione. In ognuno degli intervalli successivi, una nuova istruzione è prelevata da P1, mentre le altre istruzioni sono passate ad un'unità successiva lungo il percorso.

Questa organizzazione è chiamata macchina a pipeline, due concetti collegati sono:

- La superscalarità che consiste di più pipeline parallele cui smistare diversi tipi d'istruzione;
- Parallelismo spaziale in cui la CPU carica simultaneamente più istruzioni e le esegue in parallelo su un certo numero di unità di esecuzione specializzate identiche. Ciascuna delle unità di esecuzione parallela sfrutta a sua volta il parallelismo temporale utilizzando al suo interno una pipeline (pipeline integer e float).
- Il superpipelining che consiste nell'aumentare i passi in cui è suddiviso il trattamento dell'istruzione;

Se ciascun passo (intervallo di tempo) è di n nsec, ci vogliono $5n$ nsec per eseguire un'istruzione. Tuttavia a P5 arriva un'istruzione completa ogni n nsec, ottenendo un incremento di velocità di cinque volte. In condizioni ottimali, si esegue un'istruzione per ogni ciclo, invece di una

ogni cinque cicli, dando una velocità media di esecuzione di un'istruzione per ciclo minore. Sfortunatamente, alcuni studi hanno dimostrato che il 30% delle istruzioni sono salti e questo riduce l'efficacia della pipeline. Quando s'incontra una JMP, l'istruzione successiva da eseguire potrebbe essere quella seguente il salto oppure quella all'indirizzo del salto. Poiché l'unità di prelevamento istruzione non sa quale prendere, finché il salto è stato eseguito, si ferma fino all'esecuzione dello stesso. Di conseguenza la pipeline si svuota. Il salto ha effettivamente causato la perdita di quattro cicli, con una frequenza di un salto ogni tre istruzioni è chiaro che la caduta di prestazioni è sostanziale. Nel caso del nostro prodotto scalare abbiamo previsto di effettuare il calcolo su due componenti per volta al fine di ridurre il numero di salti e sfruttare al meglio le potenzialità dell'architettura.

4.5.2 Inverted For

L'Inverted For è un particolare accorgimento suggerito da tutti i costruttori di microprocessori basati su x86, che risulta utile anche nel campo delle macchine a 64 bit di AMD.

Esso consiste nella riscrittura dei cicli for o while al fine di far sì che la condizione di uscita abbia come termine di paragone lo zero.

Questo tipo di accorgimento, applicabile universalmente a tutti i cicli for, data la natura della modifica medesima, permette di poter stabilire il verificarsi della condizione di uscita senza la necessità di effettuare un confronto, sfruttando la peculiarità di molte istruzioni assembler x86 che settano determinati bit nel registro EFLAGS durante le operazioni.

Questa ottimizzazione, che permette un guadagno di prestazioni senza andare ad incidere sulla fase di sviluppo, è in oltre particolarmente avvantaggiato da ottimizzazioni microarchitetturelle messe a disposizione dai produttori di microprocessori.

4.5.3 Versione Finale

Dopo aver preso in considerazione le possibilità appena citate siamo giunti all'implementazione ed al confronto con la versione legacy:

Listato 4.3: Versione finale

<pre> inline double p_scalar (double *a, double *b, int nvec) { double result = 0.0; double tmp = 0.0; nvec -= 1; if (nvec & 0x1) { result += a[nvec] * b[nvec]; nvec -= 1; } for (; nvec >= 0; nvec -= 2) { result += a[nvec] * b[nvec]; tmp += a[nvec - 1] * b[nvec - 1]; } return result; } </pre>	<pre> inline double p_scalar (double *a, double *b, int nvec) { double result=0.0; int i; for (i=0;i<nvec;i++) result+=a[i]*b[i]; return result; } </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

4.5.4 Eliminazione del chiamante

C'è sempre un costo associato alle chiamate dei metodi: gli argomenti devono essere messi nello stack o archiviati nei Registri, l'introduzione e la parte finale del metodo devono venire eseguiti e così via. In alcuni casi è possibile evitare il costo di queste chiamate semplicemente spostando il corpo del metodo chiamato nel corpo del chiamante. È interessante osservare che le parole chiave *inline* e *_inline* di C non garantiscono che il compilatore renda un metodo inline).

Questo tipo di ottimizzazione fornisce generalmente buoni risultati, in particolare su una macchina con un grande numero di registri come gli AMD64, che permettono una drastica riduzione del numero di accessi allo stack.

Versione Finale

Questa ottimizzazione ha richiesto una più complessa riscrittura del codice fino alla seguente versione finale:

Listato 4.4: Versione ottimizzata priva del chiamante

```

void
caller (double *result , double *sv , double *alfa ,
        double *example , int sv_num , int fet , int len ,
        double coef_lin , double coef_const , double b)
{
    double buffer ;
    int i ;
    int y ;
    for (i = 0; i < len; i++)
    {
        for (y = 0; y < sv_num; y++)
        {
            buffer =
                coef_lin * p_scalar (&sv[(y * fet)] ,
                                     &example[i * len] , fet)
                + coef_const ;
            result[i] += buffer * buffer * alfa[y];
        }
        result[i] -= b;
    }
}

```

4.5.5 Eliminazione della serialità

Se si utilizzano due processori, è possibile fare in modo tale che uno dei due sia impegnato nell'esecuzione di alcuni calcoli (ad esempio, elaborazioni di dati di un foglio elettronico) mentre l'altro in altre operazioni (ad esempio, compressione di alcuni files) contemporaneamente: questo è un tipico esempio di multitasking, cioè di due o più applicazioni eseguite contemporaneamente in modo indipendente tra di loro.

Viceversa, se un particolare software è di tipo multithreading è possibile, utilizzando un secondo processore, fare in modo che alcuni thread vengano processati da una cpu, mentre altri thread dall'altro processore; così facendo, si ottiene una diminuzione dei tempi necessari per eseguire tutti i thread dell'applicazione, pertanto si hanno prestazioni computazionali superiori. In che misura le prestazioni aumenteranno? In base alla scalabilità del processo.

Si è richiesto quindi una riorganizzazione intesa del modulo software attraverso la realizzazione di una struttura utilizzata per raccogliere i dati necessari al calcolo.

Listato 4.5: La struttura utilizzata

```
typedef struct parm
{
    double *result;
    double *sv;
    double *alfa;
    double *example;
    double coef_lin;
    double coef_const;
    double b;
    int sv_num;
    int fet;
    int len;
    int s; //Flag per indicare pari o dispari
} parm;
```

I singoli thread si occuperanno di calcolare rispettivamente i risultati che appariranno in posizione pari o dispari del vettore result. Questo alternarsi dei due thread alla lettura dal vettore dei support vector permette di sfruttare al pieno delle potenzialità le grandi capacità del bus di memoria HyperTransport.

E' stata necessario inoltre un reimplementazione della funzione caller per permettere l'utilizzo di questa ottimizzazione.

Listato 4.6: Versione finale

```
void *
thcaller (void *p)
{
    double buffer;
    parm *tmp = (parm *) p;
    int i, y;
    for (i = tmp->s; i < tmp->len; i += 2)
    {
        for (y = 0; y < tmp->sv_num; y++)
        {
            buffer =
                tmp->coef_lin * p_scalar (&(tmp->sv[(y * tmp->fet)]),
                                          &(tmp->example[i * tmp->len]),
                                          tmp->fet)
                + tmp->coef_const;
            tmp->result[i] += buffer * buffer * tmp->alfa[y];
        }
        tmp->result[i] -= tmp->b;
    }
    return NULL;
}
```

```

void
caller (double *result, double *sv, double *alfa,
        double *example, int sv_num, int fet, int len,
        double coef_lin, double coef_const, double b)
{
    double buffer = 0.0;
    pthread_t t1, t2;
    parm p1, p2;
    if (len & 1)
    {
        int y, i = —len;
        for (y = 0; y < sv_num; y++)
        {
            buffer =
                coef_lin * p_scalar (&sv[(y * fet)],
                                     &example[i * len], fet)
                + coef_const;
            result[i] += buffer * buffer * alfa[y];

        }
        result[i] —= b;
    }
    p1.result = p2.result = result;
    p1.sv = p2.sv = sv;
    p1.alfa = p2.alfa = alfa;
    p1.example = p2.example = example;
    p1.coef_lin = p2.coef_lin = coef_lin;
    p1.coef_const = p2.coef_const = coef_const;
    p1.b = p2.b = b;
    p1.sv_num = p2.sv_num = sv_num;
    p1.fet = p2.fet = fet ;
    p1.len = p2.len = len;
    p1.s = 0;
    p2.s = 1;
    pthread_create (&t1, NULL, thcaller, &p1);
    pthread_create (&t2, NULL, thcaller, &p2);
    pthread_join (t1, NULL);
    pthread_join (t2, NULL);
}

```

4.6 Risultati

Abbiamo quindi confrontato la versione legacy e la versione finale che utilizza gli accorgimenti che abbiamo appena visto ottenendo i seguenti risultati:

Si può notare come per determinati input si verifica, come atteso, il fenomeno di cache trashing in maniera che va ad incidere in maniera estremamente significativa sulle prestazioni.

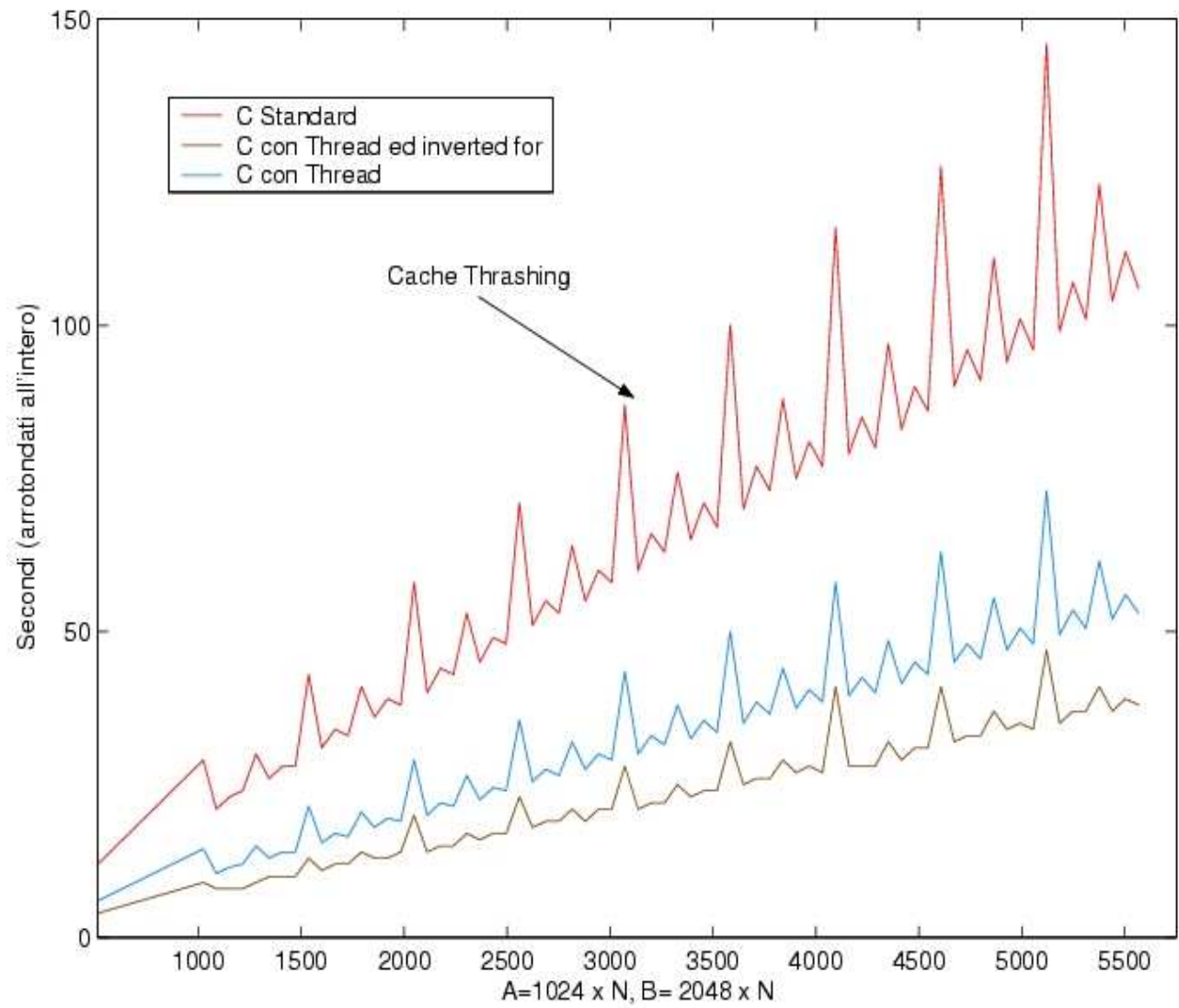


Figura 4.1: I risultati dell'implementazione

Capitolo 5

Ottimizzazioni Microarchitetturali

Le istruzioni MMX¹vevano in pratica fallito l'obiettivo di portare i benefici del paradigma SIMD di computazione nei comuni personal computer. E questo si era avuto per due motivazioni principali: lo scarso supporto fornito da Intel agli sviluppatori e la mancanza di istruzioni utilizzabili nell'emergente mondo della grafica 3D.

Le istruzioni MMX manipolano numeri interi e quindi non sono in grado di gestire le trasformazioni geometriche dei video giochi perchè in questo compito sono richieste operazioni floating point; inoltre l'utilità di un set di istruzioni capace di accelerare tali calcoli si è resa visibile in maniera pesante con l'affermazione di API dedicate alla gestione del 3D quali le Direct 3D, capaci di astrarre lo sviluppatore dal codice ottimizzato vero e proprio.

Questo fatto ha portato Intel (così come AMD con le sue 3DNow!) a sviluppare una estensione del set di istruzioni, in maniera analoga a quanto fatto con gli interi, anche per le operazioni in virgola mobile a singola precisione chiamata Internet SSE (Internet Streaming SIMD Extension) o più semplicemente SSE.

L'obiettivo proposto dai progettisti Intel era di raggiungere un incremento delle performance floating point tra il 70% e il 100%, ritenuto sufficiente a rendere percettibile la differenza e quindi competitivo il prodotto, al minor costo possibile in termini di incremento della complessità e aumento delle dimensioni del die.

Nel contempo si decise di estendere le applicazioni Multimedia introdotte con la tecnologia MMX (quale ad esempio codifiche in tempo reale di tipo MPEG-2) e di introdurre istruzioni per mascherare la latenza che deriva dalle notevoli dimensioni, in termini di memoria, dei dati implicati in applicazioni video. Il termine 'Streaming' del nome si riferisce appunto alla presenza di istruzioni che permettono il prefetch di dati simultaneamente all'elaborazione di altri già disponibili velocizzando il flusso (stream) dei dati in ingresso e in uscita del processore nascondendo nel tempo di esecuzione la latenza del fetch.

Una delle scelte basilari nella definizione di un'architettura SIMD consiste nel definire su quanti dati contemporaneamente si vuole operare in modo da raggrupparli in un vettore di dimensioni adeguate, che costituirà il nuovo tipo di dato cui faranno riferimento le istruzioni SIMD. Il team di sviluppatori di Intel ritenne che la computazione parallela di 4 floating-point a singola precisione (32 bit) e quindi di un data-type SSE da 128 bit consentisse un raddoppio complessivo delle performance senza aggiungere eccessiva complessità essendo ten-

¹<http://www.vlsilab.polito.it/thesis/alfonso/node259.html>

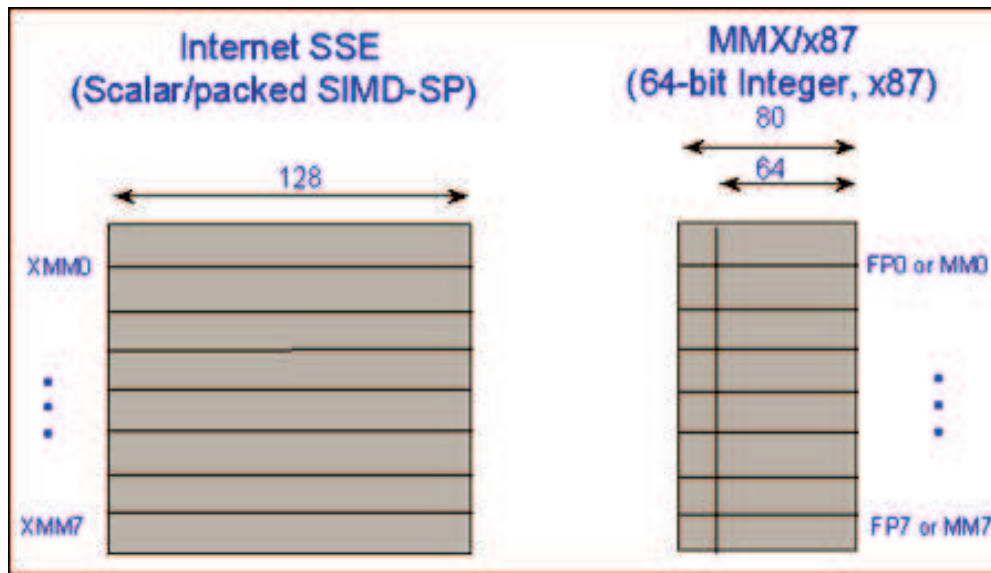


Figura 5.1: I registri SSE e i registri MMX

denzialmente ottenibile con un doppio ciclo della esistente architettura a 64 bit.

La scelta di operare su 2 floating-point non avrebbe consentito di ottenere paragonabili prestazioni mentre l'adozione di un datapath di 256 bit (8 FP da 32 bit) avrebbe determinato un impatto maggiore in termini di complessità. Mentre i 128 bit possono essere separati in 2 istruzioni da 64 bit che possono essere eseguite, come vedremo, in un ciclo, con 256 bit si sarebbe dovuto, per mantenere lo stesso throughput, raddoppiare la larghezza delle unità di esecuzione e quindi la banda di memoria per alimentarle.

Stabilito il datapath di 128 bit si poneva la domanda se implementare i registri a 128 bit nei registri MMX/x87 esistenti oppure definire un nuovo stato con registri appositi. La prima scelta, analoga a quella attuata con l'estensione alla tecnologia MMX, avrebbe comportato il vantaggio della piena compatibilità con il sistema operativo ma lo svantaggio di dover condividere i registri, già penalizzanti, della architettura IA-32.

La seconda scelta avrebbe comportato il problema di dover adattare i sistemi operativi, problema poco sentito da Intel data la sua forza contrattuale, ma avrebbe avuto il vantaggio di facilitare i programmatori e la possibilità di eseguire contemporaneamente istruzioni MMX, x87 o SIMD-FP. I progettisti Intel optarono per aggiungere un nuovo stato architetturale, per la prima volta dai tempi dell'aggiunta di quello x87 ai tempi del i386 nel 1985, con la definizione di 8 nuovi registri da 128 bit (chiamati registri XMM), cosa che non era stata fatta con l'introduzione delle istruzioni MMX che operavano sugli stesi registri fisici della Floatin Point Unit.

Benchè le istruzioni SSE abbiano introdotto un utile parallelismo delle istruzioni floating point permettendo incrementi di prestazioni in molti settori del processing multimediale, ci sono ancora una serie mancanze e di operazioni non supportate.

Non tutti i data-type sono supportati da tutte le istruzioni ad esempio mancano le multipli-

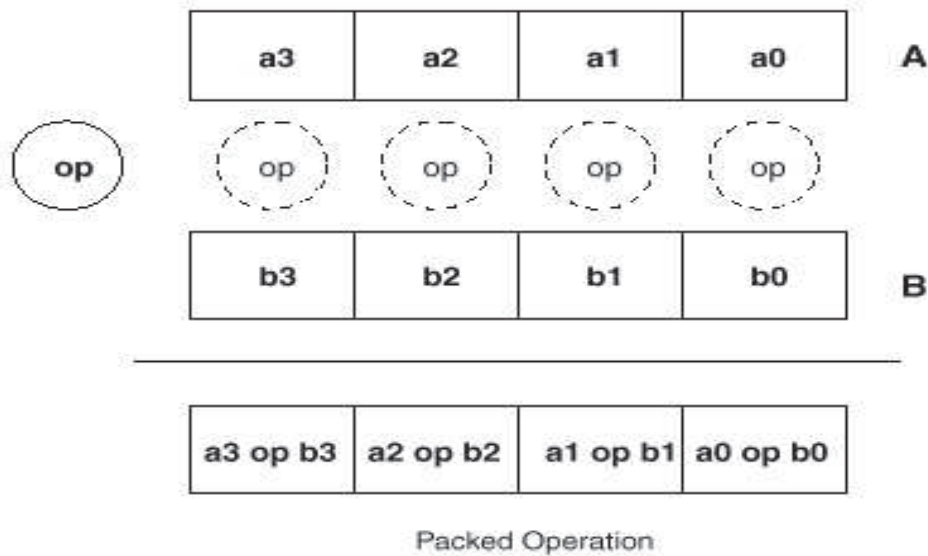


Figura 5.3: Un esempio di operazione vettoriale

- Streaming Memory: 8 istruzioni per il miglioramento della gestione della gerarchia di memoria;

Un'altra interessante suddivisione che si può operare è quella in base a come i dati che devono essere manipolati vengono gestiti. In quest'ottica facciamo riferimento a due categorie principali: istruzioni su dati packed, identificate dal suffisso 'PS' e istruzioni su dati scalari, identificate dal suffisso 'SS'. Si è scelto di definire esplicitamente operazioni scalari nel nuovo set di istruzioni e non di usare per le operazioni scalari solo le istruzioni x87 in quanto ciò avrebbe comportato problemi di compatibilità tra i risultati (calcolo in 32 bit per SIMD-FP contro gli 80 bit in x87) e costo in termini di tempo di esecuzione nel caso di utilizzo di istruzioni packed per operazioni scalari.

Le istruzioni packed operano su tutti e quattro gli elementi dei due operandi mentre le istruzioni Scalari operano solo sull'elemento 'least-significant' dei due operandi passando direttamente gli altri tre elementi del registro sorgente al registro destinazione.

5.2 Prima implementazione

Al fine di ottenere il miglior aumento delle prestazioni si è reso necessario utilizzare il medesimo tipo di dato float per tutta la fase computazione in quanto in un registro di 128 bit possono essere memorizzati 2 double (64 bit ciascuno) o fino a 4 float (32 bit).

Naturalmente si è resa necessaria una fase di analisi ulteriore per stabilire che la perdita di precisioni indotta dall'utilizzo di 32 bit anzichè 64 e dall'uso delle operazioni SSE anzichè dell'unità floating-point non incide in maniera rilevabile sull'intero progetto.

Si è deciso di utilizzare le istruzioni SSE anzichè le istruzioni 3DNow! sviluppate da AMD in

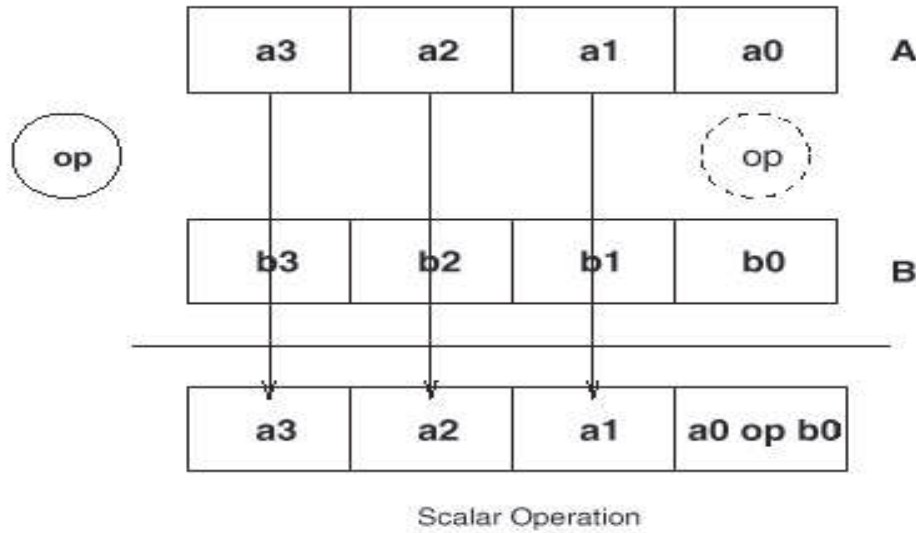


Figura 5.4: Un esempio di operazione scalare

quanto i registri sono fisicamente mappati all'interno dei registri utilizzati per l'unità floating-point (allo stesso modo delle istruzioni MMX), e quindi incorrendo nello svantaggio del basso numero di registri a disposizione per il compilatore (ad esempio l'entrata in modalità MMX/3DNow! preclude l'uso di istruzioni FPU x87) e nel vantaggio di non dover effettuare un cambio di stato per il passaggio da MMX a 3DNow!.

Con l'uscita delle SSE da parte di Intel e l'avvento dell'Athlon, sono cambiate le condizioni al contorno e la tecnologia 3DNow! così come era non risultava più al passo coi tempi, l'AMD ha quindi sviluppato le 3DNow! Enhanced. Le 3DNow! Enhanced hanno in parte recuperato lo svantaggio rispetto alle SSE grazie all'introduzione di interessanti istruzioni di pre-caching e streaming oltre ad utili istruzioni per accelerare la codifica e la decodifica dei filmati MPEG2 (lo standard alla base dei canali satellitari, del DVD e del DivX).

Tuttavia AMD, stanca di combattere con il problema del supporto, ha dovuto riconoscere in parte la superiorità delle SSE tanto da abbandonare in pratica lo sviluppo di ulteriori evoluzioni di 3DNow! da inserire il supporto alle SSE fin dal suo processore Athlon XP.

Tenendo conto dell'architettura della macchina utilizzata per lo sviluppo abbiamo deciso di concentrare lo sviluppo sulla funzione di prodotto scalare, che presenta le caratteristiche più promettenti in termini di miglioramenti delle prestazioni.

Il ciclo che sfrutterà la tecnologia SSE calcolerà il prodotto scalare di 16 float contemporaneamente, in quanto un numero minore avrebbe inciso in maniera negativa in termini di tempi impiegati per l'accesso alla memoria, mentre un numero maggiore di iterazioni avrebbe compromesso le prestazioni andando ad aumentare il numero di operazioni necessarie nel caso la lunghezza del vettore non sia multipla del fattore operazione del ciclo for, infatti l'implementazione che è stata sviluppata non richiede alcun tipo di padding all'interno dell'area di memorizzazione, migliorando quindi sia l'utilizzo della memoria sia la possibilità di riutilizzo

del codice che non richiede una particolare layout dei parametri forniti in input.

Listato 5.1: Prodotto scalare con SSE

```

inline float p_scalar( float *va, float *vb, int nvec)
{
    int ind,tmp;
    float b1=.0f,b2=.0f;
    float not1[4]={.0f,.0f,.0f,.0f};
    float not[4]={.0f,.0f,.0f,.0f};
    v4sf a,b,c,d,e;
    v4sf f,g,h;
    v4sf m,n;
    if((tmp=(nvec &0xF)))
    {
        for(ind=1;ind<=tmp;ind++)
        {
            not1[4]+=va[nvec-ind]*vb[nvec-ind];
        }
        nvec&=0xF;
    }
    m=__builtin_ia32_loadaps(not1);
    n=__builtin_ia32_loadaps(not);

    for(nvec--;nvec>0;nvec-=16,va+=16,vb+=16)
    {
        a=__builtin_ia32_loadups(va);
        c=__builtin_ia32_loadups(va+4);
        e=__builtin_ia32_loadups(va+8);
        g=__builtin_ia32_loadups(va+12);
        b=__builtin_ia32_loadups(vb);
        d=__builtin_ia32_loadups(vb+4);
        f=__builtin_ia32_loadups(vb+8);
        h=__builtin_ia32_loadups(vb+12);
        a=__builtin_ia32_mulps(a,b);
        c=__builtin_ia32_mulps(c,d);
        h=__builtin_ia32_mulps(h,g);
        e=__builtin_ia32_mulps(e,f);
        a=__builtin_ia32_addps(a,c);
        h=__builtin_ia32_addps(h,e);
        m=__builtin_ia32_addps(a,m);
        n=__builtin_ia32_addps(h,n);
    }
    m=__builtin_ia32_addps(m,n);
    n=m;
    n=__builtin_ia32_shufps(n,n,0x4E);
}

```

```

m=__builtin_ia32_addps(m,n);
n= __builtin_ia32_shufps (n, n, 0x11);

m=__builtin_ia32_addps(m,n);
__builtin_ia32_storess (&b2, m);

return b2;
}

```

Come si nota per l'implementazione di questa prima versione abbiamo scelto di utilizzare il linguaggio Assembler attraverso l'uso delle apposite istruzioni builtins fornite al programmatore dal compilatore, builtins che sono prerogativa di tutti i compilatore di alto livello (Gnu Compiler Collection, Visual Studio C/C++ Compiler, Intel C Compiler).

Queste istruzioni possono essere mappate in fase di compilazione in due modi:

- con Codice Nativo, nel caso che l'architettura di destinazione possieda all'interno della propria ISA² le istruzioni utilizzate.
- con appositi Template, nel caso che l'ISA dell'architettura di riferimento non possieda le istruzioni.

In questo modo il codice potrà essere utilizzato anche su architettura che non presentano le istruzioni SSE, in quanto sarà compito del compilatore scrivere il codice necessario, ovviamente affidarsi al lavoro del compilatore potrebbe far incorrere in gravi performance bugs, in quanto non necessariamente il codice generato attraverso template è il più performante.

Come si può notare dall'implementazione calcoliamo due prodotti scalare memorizzati nei segnaposto n ed m (sarà il compilatore a compile-time a scegliere i registri migliori da impiegare) lasciando al codice posto al di fuori del ciclo for il compito di effettuare la somma e l'operazione di shufps utilizzata per sommare tra loro i 4 valori contenuti all'interno del registro.

5.3 Ottimizzazioni Applicate

5.3.1 Instruction Reordering

Da un'attenta analisi dello schema di funzionamento di un Opteron si nota immediatamente l'esistenza di tre unità funzionali gestite dallo scheduler a 36 vie che si occupa di gestire e di effettuare l'eventuale riordino di tutte le istruzioni a virgola mobile.

La presenza di unità distinte per il calcolo delle somme e delle moltiplicazioni floating-point presenta numerosi vantaggi computazionali in quanto permette di effettuare questi tipi di operazioni in maniera contemporanea ed indipendente presentando quindi la possibilità di un ulteriore aumento delle prestazioni.

²<http://www.lithium.it/articolo0019p1.htm>

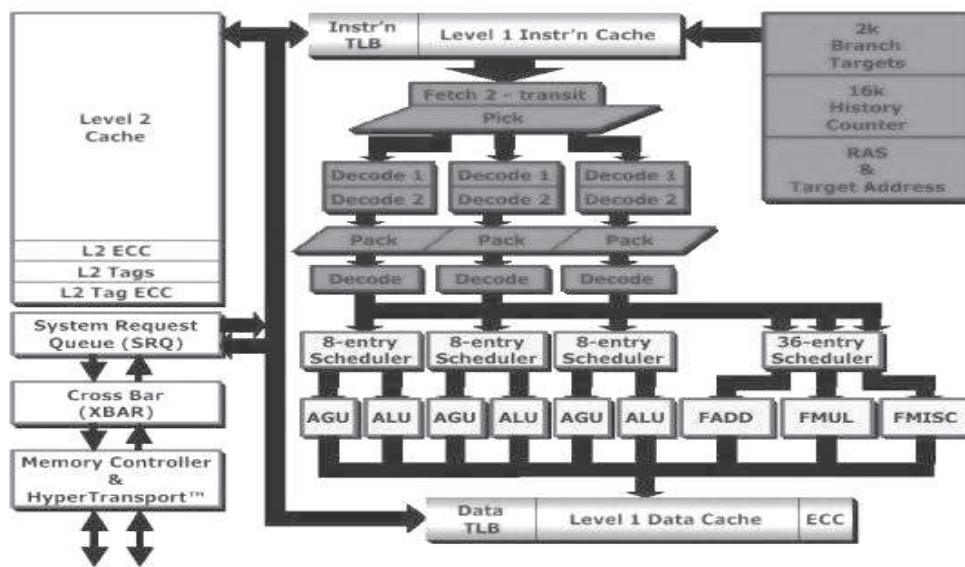


Figura 5.5: Schema di funzionamento di un Opteron

Naturalmente bisogna provvedere all'individuazione delle righe di codice dove un simili approccio risulti corretto e provvedere alla modifica. In particolare il codice

Listato 5.2: Versione non riordinata

```

1: a=__builtin_ia32_mulps(a,b);
2: c=__builtin_ia32_mulps(c,d);
3: h=__builtin_ia32_mulps(h,g);
4: e=__builtin_ia32_mulps(e,f);
5: a=__builtin_ia32_addps(a,c);
6: h=__builtin_ia32_addps(h,e);
7: m=__builtin_ia32_addps(a,m);
8: n=__builtin_ia32_addps(h,n);

```

è stato riscritto per poter utilizzare al meglio questo approccio, in particolare, dopo una serie di test si è scelto di utilizzare la seguente forma:

Listato 5.3: Versione riordinata

```

1: a=__builtin_ia32_mulps(a,b);
2: c=__builtin_ia32_mulps(c,d);
3: a=__builtin_ia32_addps(a,c);
4: h=__builtin_ia32_mulps(h,g);
5: e=__builtin_ia32_mulps(e,f);
6: h=__builtin_ia32_addps(h,e);
7: m=__builtin_ia32_addps(a,m);
8: n=__builtin_ia32_addps(h,n);

```

dove l'istruzione in riga 3 viene eseguita in contemporanea con l'istruzioni successiva (1 ciclo di clock di ritardo) riducendo quindi il tempo per iterazione in quanto impiega registri il cui valore è già calcolato al momento dell'esecuzione dell'istruzione.

5.3.2 Memory Alignment

Un altro fattore fondamentale nel campo delle prestazioni è l'accesso alla memoria.

La memoria è indirizzabile per singoli byte, tuttavia per migliorare le prestazioni dei sistemi di memorizzazione ad ogni singolo accesso in memoria si andrà a leggere tutti i dati che si trovano in una determinata linea di memoria.

Quindi se l'informazione di interesse è suddiviso in più linee di memoria, si renderà necessario un numero maggiore di letture in memoria, con conseguente perdita di cicli di clock nell'attesa che la memoria principale, che come è noto risulta molto più lenta dei microprocessori attuali, fornisca il dato. Naturalmente una simile situazione deve essere il più possibile evitata, anche se non sempre è possibile allineare i dati con successo.

Le istruzioni SSE, che come è noto, sono state progettate per ottenere la massima efficienza tengono conto della possibilità di effettuare accessi alla memoria in maniera allineata e mettono a disposizione un'istruzione apposita per l'accesso alla memoria *movaps*.

Per utilizzare questa istruzione è necessario che la memoria sia allineata 16 byte, mentre la malloc di sistema restituisce indirizzi allineati a 8 byte, per ovviare a questo problema si è scelto di richiedere al sistema un quantità di memoria leggermente maggiore del necessario e provvede a source code level all'allineamento del puntatore alla memoria attraverso un costrutto del tipo:

Listato 5.4: Versione con memoria allineata a 16 byte

```
void
array (size_t bytes , float **dst)
{
    *dst = malloc (bytes+16*4);
    *dst -= ((long int)dst&0xF); // Un indirizzo calcolato con modulo 16
    *dst += 0x10;
}
```

che fornisce il corretto allineamento.

5.3.3 Versione Finale

Dopo aver preso in considerazione le possibilità appena citate siamo giunti all'implementazione seguente:

Listato 5.5: Versione finale del prodotto scalare

```
inline float p_scalar( float *va, float *vb, int nvec)
{
    int ind, tmp;
    float b1=.0f, b2=.0f;
    float not1[4]={.0f, .0f, .0f, .0f};
```

```

float not[4]={.0f,.0f,.0f,.0f};
v4sf a,b,c,d,e;
v4sf f,g,h;
v4sf m,n;
if((tmp=(nvec &0xF))
    {
        for (ind=1;ind<=tmp;ind++)
            {
                not1[4]+=va[nvec-ind]*vb[nvec-ind];
            }
        nvec&=0xF;
    }
m=__builtin_ia32_loadaps(not1);
n=__builtin_ia32_loadaps(not);

for (nvec--;nvec>0;nvec-=16,va+=16,vb+=16)
    {
        a=__builtin_ia32_loadaps(va);
        e=__builtin_ia32_loadaps(va+8);
        b=__builtin_ia32_loadaps(vb);
        f=__builtin_ia32_loadaps(vb+8);
        c=__builtin_ia32_loadaps(va+4);
        d=__builtin_ia32_loadaps(vb+4);
        h=__builtin_ia32_loadaps(vb+12);
        g=__builtin_ia32_loadaps(va+12);
        a=__builtin_ia32_mulp(a,b);
        c=__builtin_ia32_mulp(c,d);
        h=__builtin_ia32_mulp(h,g);
        e=__builtin_ia32_mulp(e,f);
        a=__builtin_ia32_addps(a,c);
        h=__builtin_ia32_addps(h,e);
        m=__builtin_ia32_addps(a,m);
        n=__builtin_ia32_addps(h,n);
    }
m=__builtin_ia32_addps(m,n);
n=m;
n=__builtin_ia32_shufps(n,n,0x4E);
m=__builtin_ia32_addps(m,n);
n=__builtin_ia32_shufps(n,n,0x11);

m=__builtin_ia32_addps(m,n);
__builtin_ia32_storess(&b2,m);
return b2;
}

```


5.4 Risultati

Abbiamo quindi confrontato la versione antecedente le ottimizzazioni assembler e la versione finale che utilizza gli accorgimenti che abbiamo appena visto ottenendo i seguenti risultati:

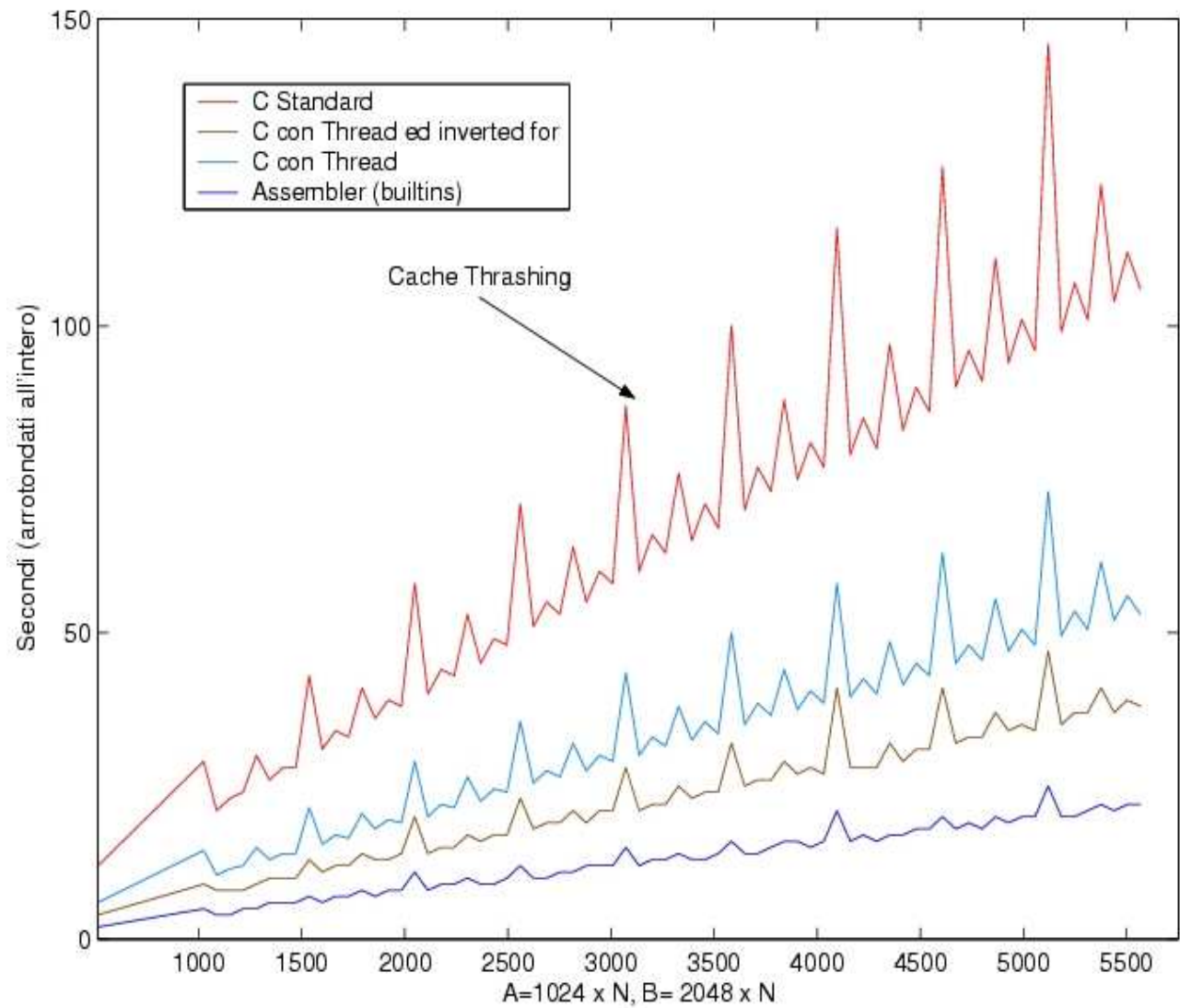


Figura 5.6: I risultati ottenuti con l'implementazione sopracitata

Capitolo 6

Progetto ATLAS

6.1 Introduzione ad ATLAS

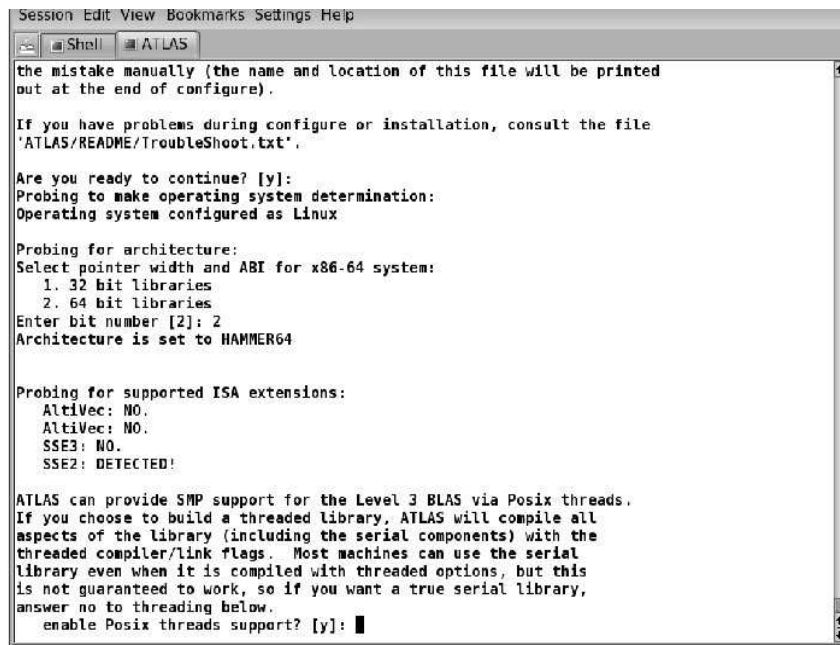
Il progetto ATLAS¹ (Automatically Tuned Linear Algebra Software) è un progetto di ricerca che si occupa dell'applicazione di tecniche empiriche con lo scopo di ottenere ottimizzazioni microarchitetturali portabili.

L'algebra lineare è ricca di operazioni che sono altamente ottimizzabili nel senso che un codice altamente ottimizzato può essere eseguito più velocemente di molti ordini di grandezza rispetto ad una funzione standard.

Tuttavia come abbiamo visto nei capitoli precedenti queste ottimizzazioni sono estremamente non portabili, addirittura avviene spesso che un'ottimizzazione particolarmente valida per una data architettura provochi un rallentamento quando impiegata su un'altra architettura. Il metodo tradizionale, che è stato applicato nelle precedenti implementazioni, prevede la scrittura di funzioni altamente ottimizzate per ogni singola architettura su cui si intenda sfruttare le ottimizzazioni. Ovviamente questo è un approccio particolarmente complesso, senza considerare le rapidissime evoluzioni hardware che introducono nuove possibili ottimizzazioni e nuovi possibili colli di bottiglia.

Nei primi mesi del 2000 è stata presentata quella che si presenta come soluzione a queste problematiche attraverso l'applicazione del paradigma AEOS (Automated Empirical Optimization of Software).

In un modulo software sviluppato secondo l'approccio AEOS, come ad esempio Atlas, il software fornisce svariate implementazioni delle operazioni richieste, alcune delle quali vengono generate in fase di compilazione. In seguito tra tutte le possibili implementazioni verranno scelte le più performanti per una data architettura in fase di compilazione attraverso la misurazione delle prestazioni di ogni singola implementazione.



```

Session Edit View Bookmarks Settings Help
[Shell] [ATLAS]
the mistake manually (the name and location of this file will be printed
out at the end of configure).

If you have problems during configure or installation, consult the file
'ATLAS/README/TroubleShoot.txt'.

Are you ready to continue? [y]:
Probing to make operating system determination:
Operating system configured as Linux

Probing for architecture:
Select pointer width and ABI for x86-64 system:
  1. 32 bit libraries
  2. 64 bit libraries
Enter bit number [2]: 2
Architecture is set to HAMMER64

Probing for supported ISA extensions:
  AltiVec: NO.
  AltiVec: NO.
  SSE3: NO.
  SSE2: DETECTED!

ATLAS can provide SMP support for the Level 3 BLAS via Posix threads.
If you choose to build a threaded library, ATLAS will compile all
aspects of the library (including the serial components) with the
threaded compiler/link flags. Most machines can use the serial
library even when it is compiled with threaded options, but this
is not guaranteed to work, so if you want a true serial library,
answer no to threading below.
enable Posix threads support? [y]: █

```

Figura 6.1: La fase di configurazione di ATLAS

6.2 ATLAS e BLAS

Atlas risulta essere in un'ultima analisi un'applicazione dell'approccio AEOS all fine di ottenere una versione estremamente ottimizzata ed estremamente portabile della ben nota API BLAS (Basic Linear Algebra Subroutine) suddivisa tradizionalmente in tre livelli ognuno dei quali implementa una categoria diversa di funzioni di algebra lineare utilizzabili individualmente, che saranno utilizzate per implementare alcune funzioni dei livelli superiori. In particolare un'implementazione di BLAS standard è così composta:

1. Livello 1: Implementa le funzioni tra vettori;
2. Livello 2: Implementa le funzioni tra matrice e vettore;
3. Livello 3: Implementa le funzioni tra matrici;

6.3 GEMM

Particolare attenzione è inoltre riservata all'implementazione della funzione GEMM (General Matrix to Matrix Multiply) che rappresenta il cuore per tutte le funzioni del livello 3. Gemm è una delle poche componenti software il cui codice sorgente non è generato in fase di compilazione, le cui differenze al variare delle architetture di destinazione sono contenute nei file sorgenti, o generate attraverso la parametrizzazione di determinate variabili all'interno della

¹Vedi <http://math-atlas.sourceforge.net/faq.html>.

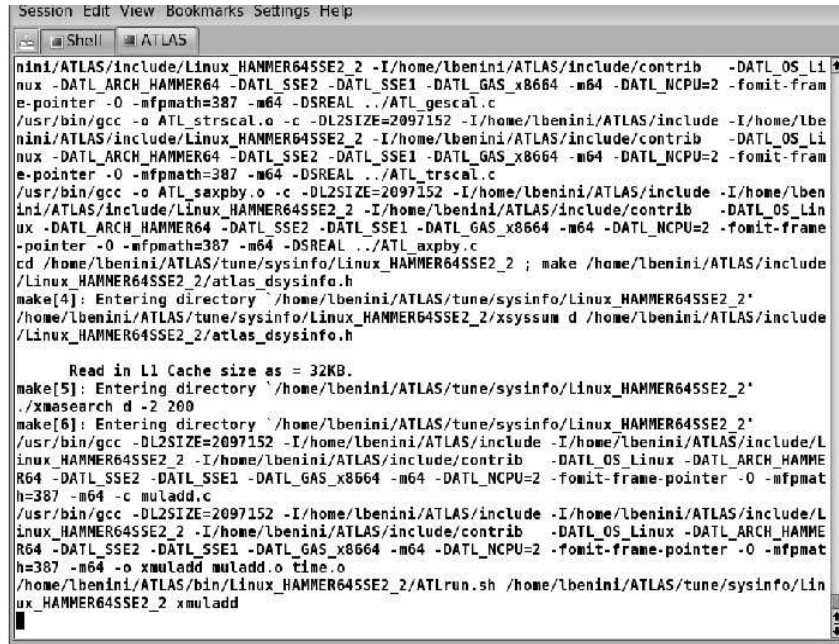


Figura 6.2: La fase di compilazione di ATLAS

medesima architettura.

Gemm si basa un particolare prodotto matriciale chiamato *L1 matmul* che al momento può essere considerato come una scatola nera che fornisce la possibilità di eseguire un prodotto matriciale estremamente ottimizzato tale per cui non è necessario prendere in considerazione la necessità della suddivisione della matrice in blocchi al fine di ottimizzare l'uso della memoria cache.

Quando un utente invoca la funzione `gemm` o una delle sue varianti a diversa precisione, ATLAS stabilisce se la dimensione del problema è sufficientemente grande da tollerare la copia delle matrici in input. Se le matrici sono sufficientemente grandi da tollerare questo overhead pari a $O(N^2)$ ATLAS provvederà ad effettuare la copia delle matrici riorganizzando i dati in maniera tale da effettuare un blocking ottimale per l'architettura sottostante, come stabilito in fase di compilazione.

Attraverso questa suddivisione le matrici in input sono già suddivise in blocchi contigui, in modo tale da minimizzare il fenomeno di cache trashing (il ripetersi di cache miss in seguito ad una erronea organizzazione dei dati in memoria), massimizzando l'uso della cache.

Questa copia permette ad ATLAS di effettuare anche un eventuale prodotto per un uno scalare α nel caso che quest'ultimo sia presente tra i parametri con valore diverso da 1.

Naturalmente se le matrici sono troppo piccole, il costo $O(N^2)$ risulta una componente dominante del costo computazione dell'algoritmo, che ricordiamo ha un costo computazione pari a $O(N^3)$. Per queste matrici ATLAS utilizzerà *L1 matmul* specificando come parametri di input non le matrici copiate, ma direttamente le matrici fornite dall'utente, in questo caso si avrà un costo aggiuntivo legato all'aritmetica dei puntatori necessaria per utilizzare le

matrici fornite dall'utente la cui allocazione in memoria è estremamente libera, basti pensare all'utilizzo di un eventuale padding.

In ATLAS la scelta della soglia oltre la quale l'utilizzo dell'operazione di copia produce sensibili miglioramenti in termini di prestazioni è considerato un parametro di AEOS, essendo un parametro che dipende fortemente dalla configurazione hardware specifica, basti pensare alle diverse frequenze delle memorie principali.

Per risolvere questo problema ATLAS confronta la velocità delle varie implementazioni di L1 matmul nel caso di operandi sottoposti o meno a copia al variare delle dimensioni e della forma delle matrici, su alcune architetture e per determinate configurazioni dei dati in input non si raggiungerà mai un punto critico oltre il quale la copia risulta conveniente.

Questo parametro, che viene stabilito in fase di compilazione, verrà poi utilizzato in fase di esecuzione per scegliere l'approccio più performante.

Un'altro componente AEOS particolarmente importante per gemm è la soglia oltre la quale è conveniente utilizzare una matrice temporanea per i risultati anziché memorizzarli direttamente nella destinazione fornita dall'utente.

L'utilizzo di una matrice temporanea presenta alcune vantaggi rispetto all'uso della matrice fornita in input:

- L'allineamento degli indirizzi può essere controllato al fine di garantire che la matrice temporanea inizi in una locazione di memoria allineata alla dimensione della riga della cache.
- I dati sono contigui, eliminando quindi la possibilità di una dispersione, in termini di locazioni di memoria, riducendo quindi la possibilità di incorrere in cache trashing.

D'altra parte l'utilizzo di una matrice temporanea implica necessariamente il costo computazionale legato alla copiatura dei dati dalla matrice temporanea alla matrice di destinazione fornita dall'utente.

Naturalmente l'approccio che utilizza la matrice temporanea è particolarmente indicato nel caso che gemm invochi molte volte l'operazione elementare L1 matmul, come nel caso precedente la soglia oltre la quale utilizzare una matrice temporanea è stabilita in fase di compilazione attraverso una serie di test su matrici di dimensioni variabili.

Rispetto all'implementazione fornita nel capitolo precedente l'utilizzo dell'implementazione con l'uso della libreria ATLAS permetterà un gestione quasi ottimale della memoria cache.

6.4 Compilazione di ATLAS

La compilazione di ATLAS è un processo complesso che cerca di identificare, spesso con successo, le caratteristiche fisiche della macchina sottostante utilizzando, dove possibile, codice C standard mentre per altri parametri, come ad esempio l'uso delle librerie pthread, è invece richiesto un input dall'utente.

Inoltre la compilazione può essere effettuata in due modalità:

- Default, in cui vengono eseguiti un numero limitato di test per la valutazione delle

prestazioni, scartando quelle ritenute dagli sviluppatori troppo lente per l'architettura, ed escludendo per determinati parametri i valori meno probabili.

- Full search in cui vengono prese in considerazione tutte le possibili implementazione e vengono effettuati tutti i test, anche su valori improbabili dei parametri. Ovviamente la compilazione Full search richiede un tempo estremamente più lungo rispetto alla compilazione Standard, fornendo però in genere un miglioramento delle prestazioni complessive di alcuni punti percentuali.

6.4.1 Alcuni risultati della compilazione

Durante la fase di compilazione vengono memorizzati i tempi e le performance delle funzioni scelte per essere inserite nella libreria. Al termine della compilazione viene prodotto un file di sommario che riassume le prestazioni raggiunte da ATLAS sul sistema. Ad esempio al termine della compilazione della versione 3.7.3 di ATLAS sull'Opteron abbiamo ottenuto, tra le altre informazioni, i seguenti dati:

STAGE 2-2-1 : BUILDING BLOCK MATMUL TUNE

```
The best matmul kernel was ATL_smm14x1x84_sse.c, NB=84
Performance: 4896.04MFLOPS (520.12 percent of apparent peak)
mmNN      : ma=1, lat=2, nb=28, mu=6, nu=1 ku=28, ff=0, if=6, nf=1
             Performance = 1789.06 (36.54 of copy matmul, 190.06 of peak)
mmNT      : ma=1, lat=2, nb=28, mu=6, nu=1 ku=28, ff=0, if=6, nf=1
             Performance = 1681.15 (34.34 of copy matmul, 178.59 of peak)
mmTN      : ma=1, lat=3, nb=28, mu=6, nu=1 ku=28, ff=0, if=6, nf=1
             Performance = 2049.93 (41.87 of copy matmul, 217.77 of peak)
mmTT      : ma=1, lat=2, nb=28, mu=6, nu=1 ku=28, ff=0, if=6, nf=1
             Performance = 1751.59 (35.78 of copy matmul, 186.08 of peak)
```

Da questo estratto del file *SUMMARY.LOG* si può osservare come la funzione scelta per il prodotto matriciale è la funzione contenuta nel file *ATL_smm14x1x84_sse.c* con l'impostazione del parametro di cache blocking ad un valore di 84.

Per quanto riguarda le performance del prodotto matriciale vero e proprio si può notare come l'utilizzo di atlas ho prodotto un forte incremento delle prestazioni in particolare rispetto al picco della versione in linguaggio C, in particolare il prodotto matriciale in cui la prima matrice viene fornita trasposta (mmTN) raggiunge un valore pari al 217.77% del picco della macchina.

6.5 ATLAS su Sistemi Windows

Il progetto ATLAS richiede che il sistema target per cui si vuole compilare ATLAS presenti le caratteristiche proprie dei sistemi operativi *Unix-like*, in particolare richiede in fase di compilazione alcuni degli strumenti più diffusi in quel tipo di sistemi operativi come i comandi per la creazione di link simbolici all'interno del filesystem.

Richiede inoltre alcune librerie per poter sfruttare al meglio le caratteristiche proprie dell'hardware sottostante, ad esempio per poter utilizzare al meglio un sistema multiprocessore è necessaria l'esistenza della libreria `pthread` che fornisce un'implementazione ai POSIX thread².

Per eseguire il porting di un sistema complesso come ATLAS non era certamente affrontabile effettuare il porting a livello di codice sorgente, in particolare considerando la complessità implementativa di alcune funzionalità richieste da ATLAS.

Abbiamo quindi deciso di operare il porting attraverso lo strumento Cygwin³.

Cygwin può essere visto come un ambiente Linux-like funzionante sotto Windows. Esso consiste di due parti distinte:

1. Una DLL che funge come strato di emulazione Linux, al fine di fornire sostanzialmente le funzionalità offerte, in un ambiente Linux reale, dalle chiamate di sistema;
2. Una collezione di tools che forniscono un *look and feel* tipico di un sistema Linux based.

Attraverso un'installazione completa del software Cygwin è stato possibile effettuare la ricompilazione di ATLAS fino ad ottenere la versione statica della libreria *libatlas.a*. Dopo un semplice ridenominazione del file in *libatlas.lib* è stato possibile effettuare il linking statico della libreria ATLAS attraverso i compilatori nativi per piattaforma Windows in particolare con il compilatore fornito con Visual Studio 6 e con Visual Studio .NET.

²<http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html>

³<http://www.cygwin.com>

6.6 Implementazione

Tenendo conto delle funzioni implementate nella libreria forniamo la seguente implementazione:

Listato 6.1: Versione utilizzando la libreria Atlas

```

void classify_poly2_atlas (float* a, int ra, int ca,
                          float* b, int rb, int cb,
                          float* c, float* result,
                          float* alfa, float coef_lin,
                          float coef_const, float threshold)
{
int i, j;
float *tmp;
/* allocate memory for intermediate temp matrix */
if ((tmp=(float *) malloc (sizeof(float)*ra*rb))==NULL)
{
    fprintf(stderr, "\nlibrhosvm: _error _malloc _in _classify_poly2_atlas");
    fflush(stderr);
    exit(EXIT_FAILURE);
}
(void) catlas_sset (ra*rb, 1, c, 1);
// call GEMM for single (float) values
(void) cblas_sgemm (CblasRowMajor,
                  CblasNoTrans,
                  CblasTrans, ra, rb, ca, coef_lin,
                  a, ca, b, cb, coef_const, c, rb);
/* duplicate memory */
(void) cblas_scopy (ra*rb, c, 1, tmp, 1);
for (i=0; i<rb; i++)
    (void) cblas_sscal (ra, alfa[i], c+i, rb);
for (i=0; i<ra; i++)
    result[i] = cblas_sdot (rb, c+i*rb, 1, tmp+i*rb, 1) - threshold;
/* free memory */
free(tmp);
}

```

6.7 Risultati

Verifichiamo ora le prestazioni dell'implementazione che utilizza le librerie ATLAS nelle medesime condizioni dei test precedenti.

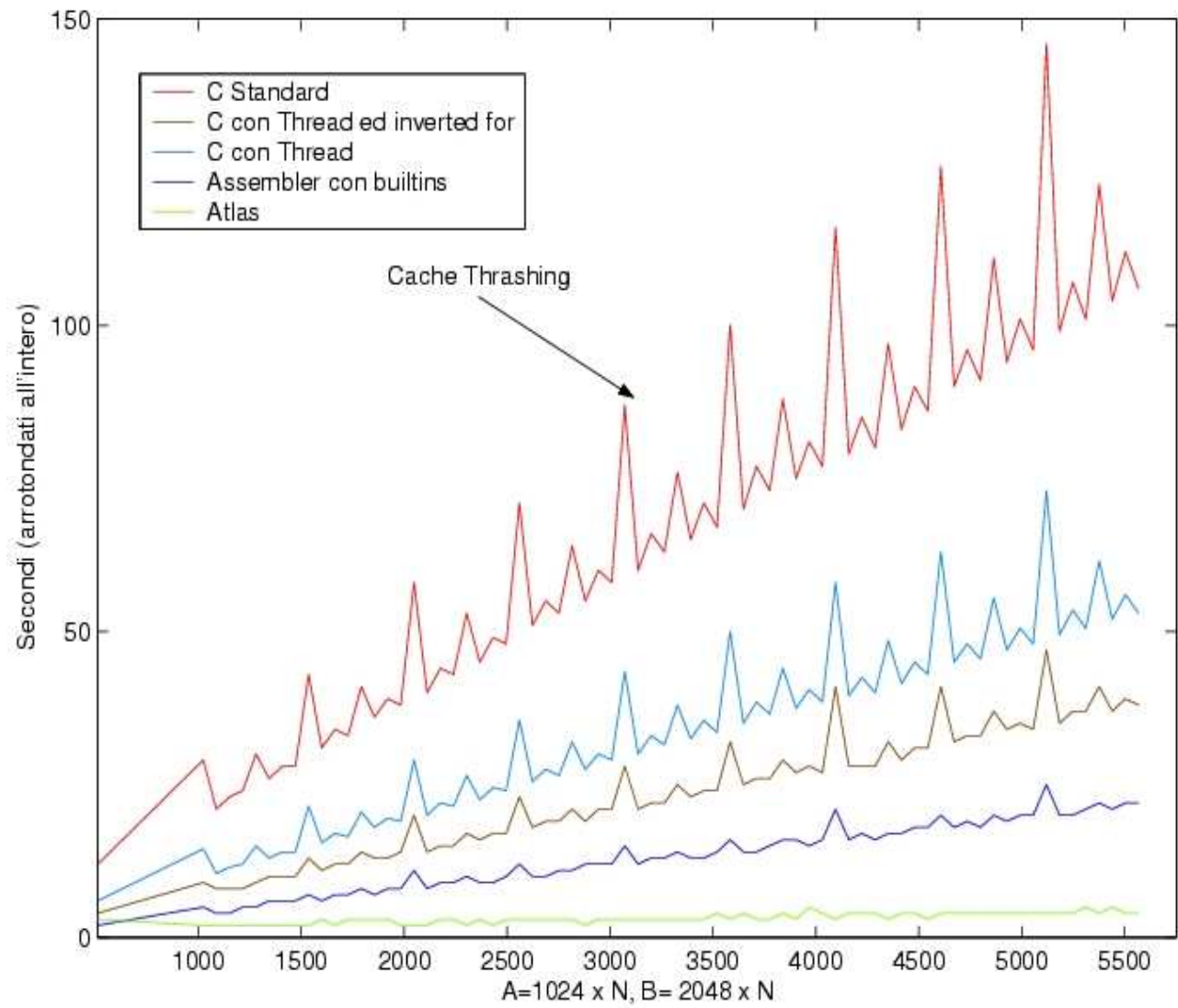


Figura 6.3: I risultati ottenuti con l'implementazione sopracitata

Conclusioni

La crescita della potenza di calcolo fornisce enormi potenzialità, permettendo di utilizzare macchine via via sempre più potenti per affrontare nuove problematiche.

Tuttavia questa crescita non dipende esclusivamente da un aumento delle velocità operazionali delle macchine, ma anche dall'introduzioni di nuove unità funzionali, come quelle preposte al calcolo vettoriale.

Per quanto forniscano la possibilità di raggiungere speed up enormi queste unità funzionali rischiedono un lavoro estremamente pesante per gli sviluppatori dei compilatori, che spesso non riescono a sfruttare al meglio le innovazioni tecnologiche, basti pensare alla tecnologia MMX, ormai vecchia di anni e tuttavia generalmente poco sfruttata dai compilatori.

In una simile ottica appare chiaro la necessità di continuare con la realizzazione di funzioni assembler estremamente ottimizzate codificate a mano, lasciando al programmatore l'onere di stabilire una corretta e performante vettorizzazione e gestione della cache.

In quest'ottica risulta innovativa l'idea avanzata da ATLAS che grazie al suo approccio AEOS offre la possibilità, al prezzo di una ristrutturazione del codice, di rendere portabili queste ottimizzazioni riducendo quindi enormemente il carico di lavoro da svolgere per ottenere una versione ottimizzata funzionante su più architetture.

Uno speciale ringraziamento va a tutti coloro che mi hanno aiutato nella stesura di questa tesi, in particolare al prof Campanini per avermi fornito un problema reale su cui applicare le ottimizzazioni, al dott. Roffilli ed al dott. Schiaratura per il continuo aiuto ed a C. Zoffolli per la grande competenza tecnica.

Bibliografia

- [1] David A. Patterson, John L. Hennessy, *“Struttura, Organizzazione e Progetto dei Calcolatori”*
- [2] William Stallings, *Architettura e Organizzazione dei Calcolatori”*
- [3] David A. Patterson, John L. Hennessy, *“Computer Architecture: A Quantitative Approach 3nd ed.”*
- [4] Randall Hyde *“Art of Assembly Language Programming”*
- [5] Steve Holmes *“C Programming”*
- [6] Nardo Giorgio *“Introduzione al L^AT_EX”*
- [7] A. Bazzani, A. Bevilacqua, D. Bollini, R. Campanini, D. Dongiovanni, E. Iampieri, N. Lanconelli, A. Riccardi, M. Roffilli, R. Tazzoli. *“A novel approach to mass detection in digital mammography based on Support Vector Machines”*, In Proc. of LXXXVIII National Congress of the Italian Physics Society, 2002.
- [8] R. Campanini, D. Dongiovanni, E. Iampieri, N. Lanconelli, M. Masotti, G. Palermo, A. Riccardi, M. Roffilli. *“A novel featureless approach to mass detection in digital mammograms based on Support Vector Machines”*, Physics in Medicine and Biology, volume 49, issue 6, 961-975, 2004.
- [9] O. Schiaratura: *“Progettazione ed implementazione di un sistema di calcolo ibrido multithread-multiprocesso per HPC: applicazione all’imaging medico”*, Tesi, Scienze dell’Informazione, Univ. Bologna, 2002-2003
- [10] M. Roffilli: *“SWAR - MMX, SSE, SSE2 - Programmazione Multipiattaforma”*, Talk, Linux Day, November 22 2002, Cesena, Italy.
- [11] R. Campanini, D. Dongiovanni, N. Lanconelli, G. Palermo, A. Riccardi, M. Roffilli: *“A Support Vector Machines Classifier based on Recursive Feature Elimination for Microarray Data in Breast Cancer Characterization.”*, First National Workshop on Bioinformatics, VIII National Congress of the Italian Association for Artificial Intelligence, September 10 2002, Siena, Italy.