

# 1 Esecuzione di un workload tramite i simulatori

L'obiettivo di questo progetto è analizzare e confrontare le prestazioni di tre diverse architetture (ISA): ARM, RISC-V e x86, utilizzando un workload computazionale standardizzato. Per poter eseguire questo confronto fra le differenti architetture è imperativo stabilire una metodologia che ne garantisca l'equità.

## 1.1 Definizione di "piccolo workload"

Le istruzioni del progetto richiedono l'esecuzione di un "piccolo workload" con tracing e statistiche. Per garantire la riproducibilità e la confrontabilità tra ARM, RISC-V e x86, il workload non deve dipendere da librerie di sistema complesse che potrebbero variare tra le architetture.

Il candidato ideale per questo workload è un algoritmo di moltiplicazione di matrici (Matrix Multiplication) denso. Questo kernel computazionale offre diversi vantaggi analitici:

- **Prevedibilità:** l'accesso alla memoria è regolare, permettendo di analizzare l'efficacia delle gerarchie di cache.
- **Intensità Computazionale:** stressa le unità funzionali della CPU e la logica di pipelining, evidenziando le differenze tra l'approccio RISC (molte istruzioni semplici) e CISC (istruzioni complesse con accessi in memoria impliciti).
- **Portabilità:** può essere scritto in C standard (ANSI C) e compilato staticamente per tutte e tre le architetture senza modifiche al codice sorgente.

Il codice sorgente C proposto per l'esperimento dovrà essere compilato in tre binari distinti utilizzando cross-compiler specifici per ARM, RISC-V e x86, come dettagliato nelle sezioni successive, ed è il seguente:

```
#include <stdio.h>
#include <stdlib.h>

#define N 64 // Dimensione ridotta per simulazione rapida

void matrix_multiply(double *A, double *B, double *C, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            double sum = 0.0;
            for (int k = 0; k < size; k++) {
                sum += A[i * size + k] * B[k * size + j];
            }
            C[i * size + j] = sum;
        }
    }
}

int main() {
    size_t bytes = N * N * sizeof(double);
    double *A = (double *)malloc(bytes);
    double *B = (double *)malloc(bytes);
```

```

double *C = (double *)malloc(bytes);

// Inizializzazione deterministica
for (int i = 0; i < N * N; i++) {
    A[i] = 1.0;
    B[i] = 2.0;
}

printf("Inizio Benchmark Moltiplicazione Matrici %dx%d\n", N, N);
matrix_multiply(A, B, C, N);
printf("Fine Benchmark\n");

return 0;
}

```

## 1.2 Problema metodologico

Una delle sfide principali di questo confronto risiede nella natura diversa degli ambienti di simulazione disponibili:

- **Gem5** offre la modalità **Systemcall Emulation (SE)** la quale non simula un vero computer con un disco rigido e un sistema operativo completo. Permette di simulare l'esecuzione di binari user-space evitando di modellare i dispositivi o un OS, emulando la maggior parte dei servizi a livello di sistema;
- **MarssX86** opererà in modalità **Full System (FS)**, simulando un'intera macchina fisica, incluso il sistema operativo guest, i device e gli interrupt.

### 1.2.1 Il caso Gem5 in modalità (SE)

Nella compilazione standard (dinamica), l'eseguibile generato è incompleto: esso contiene riferimenti a funzioni esterne (come printf o malloc presenti nella glibc) ma non il loro codice macchina. Al momento dell'avvio, prima ancora che venga eseguita la funzione main(), il sistema operativo invoca il Dynamic Linker, il quale deve:

- Esplorare il filesystem alla ricerca delle librerie condivise (.so) richieste;
- Caricarle nella memoria virtuale del processo;
- Risolvere i simboli, collegando le chiamate del programma agli indirizzi di memoria delle librerie appena caricate;

Proprio per questo motivo, questo **meccanismo risulta critico** in modalità SE, poiché:

- **Assenza di Filesystem Virtuale Completo:** Gem5 in modalità SE non dispone di un filesystem guest nativo. Quando il Dynamic Linker (che è un programma a sé stante) cerca le librerie in percorsi standard (es. /lib/ o /usr/lib/), tali percorsi potrebbero non esistere nell'ambiente simulato o non coincidere con i percorsi delle librerie cross-compileate presenti sulla macchina host.
- **Complessità delle Syscall:** Il Dynamic Linker fa un uso intensivo di chiamate di sistema per mappare file in memoria (open, mmap). Sebbene Gem5 emuli le syscall comuni (come read()), la gestione complessa del caricamento dinamico può portare a errori di path lookup o incompatibilità di ABI (Application Binary Interface) tra host e guest.

Per ovviare a queste problematiche e garantire la riproducibilità degli esperimenti si rende dunque necessario imporre dei vincoli specifici sulla fase di compilazione e linking del workload che viene compilato staticamente utilizzando il flag `-static`.

Con il linking statico, tutte le dipendenze necessarie (inclusa la `libc`) vengono incorporate fisicamente all'interno del file binario al momento della compilazione. Ciò rende l'eseguibile autosufficiente:

- Elimina la necessità del Dynamic Linker in fase di avvio;
- Rimuove la dipendenza da librerie esterne durante la simulazione;
- Permette a Gem5 di caricare il binario in memoria ed eseguire direttamente il codice utente, intercettando ed emulando puntualmente le sole System Call esplicite generate dal programma (es. output su terminale), garantendo stabilità e portabilità tra diversi ambienti host.

### 1.2.2 Il caso MarssX86 (FS)

Nonostante il simulatore MarssX86 sia adatto a far girare un vero sistema operativo, con disco virtuale, un kernel Linux vero e delle librerie installate dentro l'immagine del disco (quindi in modalità Full System), rimane un problema di **compatibilità delle versioni**, dovuto principalmente al fatto che si tratta di un simulatore datato.

L'ambiente docker che viene utilizzato fornisce Ubuntu 18.04, uscito nel 2018, e utilizza la libreria `glibc` versione 2.27. Invece, le immagini del disco che si devono utilizzare per il corretto funzionamento su MarssX86 sono solitamente basate su ubuntu 10.04 o 12.04 (relative agli anni 2010 – 2012) e utilizzano una libreria `glibc` versione 2.15 o inferiore.

Purtroppo, come specificato al "**passo 2**" del capitolo 1.3.1, tali versioni dell'ambiente ubuntu sono state eliminate dai server.

La libreria standard di C non garantisce una *forward compatibility* (compatibilità in avanti):

- Un programma compilato su un sistema vecchio gira su uno nuovo (Backward Compatibility);
- Un programma compilato su un sistema nuovo (il Docker con 18.04) NON gira su uno vecchio (MarssX86).

Poiché l'ambiente di compilazione (Host) dispone di librerie di sistema (`libc`) molto più recenti rispetto all'ambiente simulato (Guest OS nell'immagine disco), un linking dinamico causerebbe errori di runtime dovuti al version mismatch delle librerie condivise.

Una possibile soluzione è utilizzare anche in questo caso il linking statico, che disaccoppia il benchmark dalle librerie del sistema guest. Pertanto, l'unico modo per riuscire a far partire il compilatore MarssX86 è utilizzare l'opzione `-static` rendendo il programma indipendente dalla "vecchiaia" del sistema operativo simulato dentro MarssX86.

## 1.3 Strategia adottata

Per garantire un confronto il più equo possibile nonostante queste differenze strutturali, è stata adottata la seguente metodologia:

1. **Isolamento Microarchitetturale (Gem5):** Per le architetture ARM e RISC-V su Gem5, abbiamo scelto la modalità **SE**. Questa scelta ci permette di misurare le prestazioni pure

della CPU e della gerarchia di memoria (cache), eliminando il 'rumore' introdotto dai servizi del kernel e dallo scheduling dei processi;

2. **Realismo Operativo (MarssX86):** Per l'architettura x86 su MarssX86, operiamo in un contesto FS. Siamo consapevoli che i risultati includeranno l'overhead del sistema operativo (context switch, gestione TLB software);
3. **Mitigazione tramite Workload e Compilazione:**
  - Abbiamo scelto un workload CPU-bound (moltiplicazione matrici) e non I/O-bound. Questo minimizza le chiamate di sistema, rendendo le prestazioni dipendenti quasi esclusivamente dalla potenza di calcolo della CPU e dall'efficienza delle cache, riducendo il divario tra modalità SE e FS;
  - Abbiamo utilizzato la compilazione statica (-static) per tutti i binari. Questo garantisce che su Gem5 (SE) il simulatore possa emulare le syscall senza dipendenze esterne, e su MarssX86 (FS) evita conflitti di librerie (ABI mismatch) tra la macchina host e il sistema guest simulato.

## 1.4 Metodologia e Configurazione degli Esperimenti

Una volta stabilita la natura del workload e le modalità di simulazione, la fase sperimentale prevede l'esecuzione di una matrice di test strutturata per isolare l'impatto delle diverse configurazioni microarchitetturali.

### 1.4.1 Matrice dei test e definizione delle Baseline

Per ciascuna delle tre architetture analizzate (ARM, RISC-V e x86), verranno effettuate tre diverse build (configurazioni di simulazione), per un totale di 9 simulazioni distinte. L'obiettivo è osservare come il medesimo workload risponda alla variazione di parametri chiave (come la frequenza di clock, la dimensione delle cache o il modello di CPU).

Per rendere il confronto significativo, all'interno di ogni gruppo architettonico viene definita una **Baseline**:

- **Configurazione Baseline:** Rappresenta il punto di riferimento con parametri standard (es. configurazione di default del simulatore).
- **Configurazione 2 e 3:** Rappresentano varianti in cui viene modificato un parametro specifico (ad esempio, raddoppio della cache L2 o differente modello di CPU).

## 1.5 Esecuzione del workload per ARM e RISC-V (Gem5)

Per poter eseguire il workload scelto sulle architetture target (ARM e RISC-V), è prima necessario affrontare il problema della compatibilità binaria. Poiché le macchine utilizzate per lo sviluppo (Host) dispongono solitamente di un set di istruzioni x86\_64 o ARM64 (Apple Silicon), i binari prodotti da un compilatore standard non sarebbero comprensibili dai simulatori configurati per architetture diverse.

In particolare, anche qualora si utilizzasse un processore Apple Silicon (già basato su ARM), sarebbe comunque necessaria la cross-compilazione (o più precisamente una compilazione specifica per il Guest) principalmente per due motivi:

- **ABI (Application Binary Interface):** il binario "nativo" di un Mac è progettato per macOS (formato Mach-O), mentre Gem5 richiede un binario in formato Linux ELF;
- **Librerie e System Call:** macOS utilizza librerie di sistema Apple, mentre il simulatore Gem5 emula l'interfaccia delle chiamate di sistema Linux.

### 1.5.1 Installazione dei cross-compiler su Windows (x86\_64)

Dunque, per prima cosa dobbiamo assicurarci di installare i "binari" giusti per la cross-compilazione. Nel sottosistema Ubuntu WSL, posizionati in `/home/NomeUtente/gem5`, utilizzare i seguenti comandi, uno alla volta:

```
sudo apt-get update
```

Per l'architettura ARM:

```
sudo apt-get -y install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
```

Per verificare la corretta installazione del cross-compiler lanciare il seguente comando:

```
aarch64-linux-gnu-gcc --version
```

Per l'architettura RISC-V:

```
sudo apt-get -y install gcc-riscv64-linux-gnu g++-riscv64-linux-gnu
```

Per verificare la corretta installazione del cross-compiler lanciare il seguente comando:

```
riscv64-linux-gnu-gcc --version
```

### 1.5.2 Installazione dei cross-compiler su macOS (ARM)

I comandi rimangono identici a quelli utilizzati su x86\_64 tranne che per qualche piccolo accorgimento. Nel capitolo 1.2.2 è stato utilizzato docker per l'installazione del simulatore. In questo caso, i comandi andranno utilizzati dalla seguente posizione `/root/gem5`:

```
root@{...}:/gem5#
```

Inoltre gli stessi devono essere utilizzati senza il comando "sudo" davanti poiché su docker, una volta entrati nell'ambiente, si è già un amministratore.

### 1.5.3 Comandi per la build del workload

## 1.6 Esecuzione del workload per X86 (MarssX86)