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-compilatori specifici per ARM, RISC-V e x86, come dettagliato nelle sezioni successive, ed è il seguente:

```c
#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** operà 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-compilate 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 oviare 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 funzionament 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 eseguite tre diverse configurazioni di simulazione, per un totale di 9 test. L'obiettivo è isolare l'impatto del modello di esecuzione della CPU e della gerarchia di memoria sulle prestazioni del workload.

Le varianti si basano su due modelli di processore fondamentali:
- ***In-Order***: un modello in cui le istruzioni vengono eseguite nell'esatto ordine in cui appaiono nel programma. È un'architettura più semplice ed efficiente dal punto di vista energetico, ma soggetta a "stalli" (blocchi) se un'istruzione deve attendere dati dalla memoria.
- ***Out-of-Order (OoO)***: un modello avanzato che permette alla CPU di eseguire le istruzioni non appena le loro dipendenze sono soddisfatte, anche se non in ordine sequenziale. Questo permette di "nascondere" le latenze della memoria, ma richiede una logica hardware molto più complessa (scheduler, registri di rinomina, ecc.).

Per ogni architettura sono state definite le seguenti tre configurazioni:
1. **Configurazione 1 (Baseline)**: Utilizza una CPU **Out-of-Order** ad alte prestazioni. La gerarchia di memoria prevede una cache L1 da 128 KB e una cache L2 da 2 MB. Rappresenta il target di massima potenza computazionale.
2. **Configurazione 2 (CPU Efficiency Test)**: Mantiene le stesse cache della baseline (L1 128 KB, L2 2 MB) ma sostituisce il modello di core con una CPU **In-Order**. Questo test serve a misurare quanto l'architettura tragga beneficio dall'esecuzione fuori ordine per il particolare workload di moltiplicazione matriciale.
3. **Configurazione 3 (Memory Constraint Test)**: Ritorna al modello di CPU **Out-of-Order** della baseline, ma riduce drasticamente le dimensioni delle cache (L1 a 32 KB e L2 a

256 KB). Questo permette di osservare quanto le prestazioni siano limitate dalla capacità della memoria (memory-bound) rispetto alla potenza pura di calcolo.

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-compilatori 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/dev_local/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-compilatore lanciare il seguente comando:

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

Infine, per compilare il file C si utilizza il seguente comando (ricordando `-static`):

```
aarch64-linux-gnu-gcc -static matrix_mul.c -o matrix_mul_arm
```

***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-compilatore lanciare il seguente comando:

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

Infine, per compilare il file C si utilizza il seguente comando (ricordando `-static`):

```
riscv64-linux-gnu-gcc -static matrix_mul.c -o matrix_mul_riscv
```

1.5.2 Installazione dei cross-compilatori 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 una volta entrati nell'ambiente docekr si è già un amministratore.

1.5.3 Comandi per la build del workload

Il comando per la build del workload (il file C) rimane invariato rispetto al calcolatore in cui viene utilizzato.
Ogni comando di build, all'interno di `gem5/m5out/NomeCartella`, creerà un file `.txt` che conterrà tutte le statistiche relative alla simulazione con le caratteristiche specificate.
Dunque, sia che ci si trovi in `root/gem5` per macOS o in `/home/NomeUtente/ dev_local/gem5` su un sistema Windows i comandi saranno i seguenti:

Le tre build per l'architettura ARM

```
// Comando di build per la configurazione 1 (baseline)
./build/ARM/gem5.opt --outdir=m5out/arm_baseline configs/deprecated/
   example/se.py \
  --cpu-type=ArmO3CPU \
  --caches --l2cache \
  --l1d_size=128kB --l1i_size=128kB \
  --l2_size=2MB \
  --sys-clock=2.688GHz \
  --cmd=./matrix_mul_arm
```

```
// Comando di build per la configurazione 2 (CPU efficiency test)
./build/ARM/gem5.opt --outdir=m5out/arm_var1_cpu configs/deprecated/
   example/se.py \
  --cpu-type=ArmMinorCPU \
  --caches --l2cache \
  --l1d_size=12kB --l1i_size=128kB \
  --l2_size=2MB \
  --sys-clock=2.688GHz \
  --cmd=./matrix_mul_arm
```

```
// Comando di build per la configurazione 3 (Memory costraint test)
./build/ARM/gem5.opt --outdir=m5out/arm_small_cache configs/deprecated/
   example/se.py \
  --cpu-type=ArmO3CPU \
  --caches --l2cache \
  --l1d_size=32kB --l1i_size=32kB \
  --l2_size=256kB \
  --sys-clock=2.688GHz \
  --cmd=./matrix_mul_arm
```

Le tre build per l'architettura RISC-V

```
// Comando di build per la configurazione 1 (baseline)
./build/RISCV/gem5.opt --outdir=m5out/riscv_baseline configs/deprecated/
   example/se.py \
  --cpu-type=O3CPU \
  --caches --l2cache \
  --l1d_size=128kB --l1i_size=128kB \
  --l2_size=256kB\
  --sys-clock=2.688GHz \
  --cmd=./matrix_mul_riscv
```

```
// Comando di build per la configurazione 2 (CPU efficiency test)
./build/RISCV/gem5.opt --outdir=m5out/riscv_var1_cpu configs/deprecated/
   example/se.py \
  --cpu-type=MinorCPU \
  --caches --l2cache \
  --l1d_size=128kB --l1i_size=128kB \
  --l2_size=256kB  \
  --sys-clock=2.688GHz \
  --cmd=./matrix_mul_riscv
```

```
// Comando di build per la configurazione 3 (Memory costraint test)
./build/RISCV/gem5.opt --outdir=m5out/riscv_small_cache configs/
   deprecated/example/se.py \
  --cpu-type=O3CPU \
  --caches --l2cache \
  --l1d_size=32kB --l1i_size=32kB \
  --l2_size=256kB \
  --sys-clock=2.688GHz \
  --cmd=./matrix_mul_riscv
```

***Nota*:** cambiare O3CPU con ArmO3CPU ha senso per una questione di rigore metodologico e stabilità della simulazione per i seguenti motivi:

- **Specificità del modello**: Mentre O3CPU è un'etichetta generica, ArmO3CPU forza il simulatore a usare parametri di pipeline (come la profondità degli stadi e la dimensione del Reorder Buffer) specificamente modellati sulle caratteristiche delle CPU ARM;
- **Versioni future**: Il file se.py è considerato "deprecated". Nelle versioni più recenti di Gem5, l'automazione che associa O3CPU ad ArmO3CPU potrebbe essere rimossa, rendendo il comando generico non funzionante.

Questa cosa non vale in RISC-V dove il binario "build/RISCV/gem5.opt" contiene solo l'implementazione RISC-V quindi O3CPU non può essere altro che un processore RISC-V.

1.6 Esecuzione del workload per X86 (MarssX86)

A causa dell'architettura di MarssX86, che si basa su una versione datata di QEMU, l'interazione standard con il simulatore presenta significative limitazioni operative per l'esecuzione di benchmark automatizzati.

Inizialmente, la procedura standard prevedeva un intervento manuale dell'utente: era necessario avviare il workload nel sistema guest e, simultaneamente, inviare un comando al monitor di QEMU (tramite una combinazione di tasti rapida) per attivare la registrazione delle statistiche (simconfig -run). Questo approccio presentava due criticità fondamentali:

- **Mancanza di Determinismo**: Il tempo di reazione umano nell'impartire il comando di avvio non è costante. Questo rendeva impossibile sincronizzare perfettamente l'inizio della registrazione con l'inizio del calcolo matematico;
- **Inquinamento dei Dati ("*Noise*")**: Per dare all'utente il tempo fisico di attivare il simulatore, era necessario inserire nel codice C delle funzioni di ritardo (come sleep()) o loop a vuoto prima del calcolo. Tuttavia, il simulatore, essendo cycle-accurate, registrava anche questi cicli di attesa (fino a 160 milioni di cicli inutili nei test preliminari), falsando drasticamente metriche chiave come l'IPC (Instructions Per Cycle) e il conteggio totale delle istruzioni.

Per ovviare a queste problematiche e garantire la rigorosità scientifica dei dati, è stato necessario implementare una metodologia basata sulla **Region of Interest (ROI)**.
Questa tecnica sposta il controllo della simulazione dall'utente direttamente al codice sorgente. Tramite l'inclusione della libreria specifica ptlcalls.h e l'uso di "hypercalls" (istruzioni speciali che comunicano direttamente con l'hardware simulato), è il benchmark stesso a ordinare al simulatore di accendersi (ptlcall_switch_to_sim) esattamente un istante prima dell'inizio dell'algoritmo di moltiplicazione e di spegnersi (ptlcall_switch_to_native) appena terminato.

In questo modo, abbiamo eliminato completamente l'errore umano e il rumore di fondo, ottenendo misurazioni che riflettono esclusivamente il carico di lavoro computazionale oggetto di studio.

1.6.1 Implementazione della metodologia ROI

Dal momento in cui sta venendo utilizzato docker per entrambi i sistemi macOS e Windows, la procedura da eseguire rimane invariata. Possiamo suddividere la procedura in alcuni passi:

- ***Passo 1*: ottenimento della libreria `ptlcall.h`**
  Subito dopo essere entrati nel container è necessario lanciare un comando per copiare il file `ptcalls.h` all'interno della cartella "disks".

```
root@{...}:~/marss# cp /root/marss/marss/ptlsim/tools/ptlcalls.h /root/
   marss/disks/
```

- ***Passo 2*: modifica del codice C**
  Il codice C rimarrà simile a quello utilizzato per l'esecuzione del workload con Gem5: verranno aggiunte solamente delle righe in più per poter aggiungere le librerie descritte prima. Per fare ciò, si utilizza il seguente comando:

```
root@{...}:~/marss# cd marss/disks
```

```
root@{...}:~/marss/marss/disks# cat > matrix_mul.c <<EOF
#include <stdio.h>
#include <stdlib.h>
#include "ptlcalls.h" // La libreria per il controllo ROI

#define N 64
```

```
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() {
    // Usiamo static invece di malloc per evitare crash nell'ambiente
       statico
    // La dimensione e' N*N * sizeof(double), gestita automaticamente
    static double A[N * N];
    static double B[N * N];
    static double C[N * N];

    printf("Inizializzazione dati (Veloce)...\n");

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

    printf("--- ATTIVO LA SIMULAZIONE ORA ---\n");
    printf("(Il computer rallentera' drasticamente per simulare i
       calcoli floating point)\n");

    // === START ROI ===
    // Qui inizia la misurazione precisa
    ptlcall_switch_to_sim();

    matrix_multiply(A, B, C, N);

    // === STOP ROI ===
    // Qui finisce la misurazione
    ptlcall_switch_to_native();

    printf("--- SIMULAZIONE TERMINATA ---\n");
    printf("Benchmark Completato! Risultato cella 0: %f\n", C[0]);

    return 0;
}
EOF
```

- ***Passo 3*: compilazione del file C**
  Per effettuare la compilazione di `matrix_mul.c`, eseguire i seguenti comandi uno alla volta:

```
root@{...}:~/marss/marss/disks# rm matrix_mul 2>/dev/null
```

  Prima di installare il compilatore per i file C è necessario installare alcune librerie:

```
root@{...}:~/marss/marss/disks# apt-get update
```

```
root@{...}:~/marss/marss/disks# apt-get install musl-tools musl-dev
```

  Non appena le librerie sono state installate, far partire il comando di compilazione:

```
root@{...}:~/marss/marss/disks# gcc -std=gnu99 -D_GNU_SOURCE -static -
   nostartfiles -nostdinc \
     -I . \
     -I /usr/include/x86_64-linux-musl \
     -L /usr/lib/x86_64-linux-musl \
     /usr/lib/x86_64-linux-musl/crt1.o \
     /usr/lib/x86_64-linux-musl/crti.o \
     matrix_mul.c \
     /usr/lib/x86_64-linux-musl/crtn.o \
    -lc -o matrix_mul
```

  Poiché l'ambiente virtualizzato (Guest) è isolato dal sistema ospitante (Host), è necessario creare un meccanismo di trasferimento per l'eseguibile compilato.
  Quindi, tramite l'utility genisoimage, il file binario del benchmark (file C) viene incapsulato in un'immagine disco standard (ISO 9660). Questa immagine viene successivamente montata come unità CD-ROM virtuale all'avvio di QEMU, permettendo al sistema operativo Debian di leggere e copiare il file al suo interno.

```
root@{...}:~/marss/marss/disks# rm trasferimento.iso
```

```
root@{...}:~/marss/marss/disks# genisoimage -o trasferimento.iso
   matrix_mul
```

- ***Passo 4*: configurazione delle CPU e delle Cache**
  L'ultimo passaggio prima di procedere con l'esecuzione del workload è leggermente macchinoso. Infatti con il simulatore MarssX86 non è possibile specificare le tipologie di CPU o di Cache direttamente nel comando di build del workload (come si è fatto con Gem5) ma è necessario configurare alcuni file.

  Scelta della CPU
  Posizionati in `/root/marss/marss`, si andrà a creare una cartella "*my_configs*" all'interno della quale si andranno a inserire dei file per specificare il tipo di CPU che si vuole utilizzare in base alla configurazione che si vuole buildare:
  - ***baseline.cfg***: viene utilizzato per la configurazione baseline. All'interno di esso è specificata una CPU di tipo out-of-order (`-machine single_core`);
  - ***atom_core.cfg***: viene utilizzato per la configurazione CPU efficency test. All'interno di esso è specificata una CPU in-order (`-machine atom_core`);

- ***small_cache.cfg***: viene utilizzato per la configurazione memory constraints test. All'interno di esso viene specificata la stessa CPU out-of-order della baseline poiché in questo caso ciò che cambia è la dimensione delle Cache.

Scelta delle Cache L1 ed L2

All'avvio di QEMU, le Cache L1 e L2 vengono preimpostate rispettivamente a 128KB e 2MB. Ciò significa che se si volessero utilizzare delle Cache più piccole, è necesario ricompilare il simulatore prima dell'avvio di QEMU.
Per fare ciò vengono utilizzati i seguenti comandi (posizionati in `root/marss/marss`)

```
root@{...}:~/marss/marss# sed -i 's/SIZE: 128K/SIZE: 32K/g' config/
   l1_cache.conf
root@{...}:~/marss/marss# sed -i 's/SIZE: 2M/SIZE: 256K/g' config/
   l2_cache.conf
```