UNIVERSITÀ DEGLI STUDI DI SASSARI

UNISS
UNIVERSITÀ
DEGLI STUDI
DI SASSARI

Progetto di fine corso:

INSTALLAZIONE E TEST DI DUE SIMULATORI ARCHITETTURALI: Gem5 e MarssX86

Studente:	Matricola:
Daniele Picciau	50056771
Monica Manai	50057077
Diego Perazzona	50057111

ANNO ACCADEMICO 2025-2026

Indice

1 Installazione e setup dei simulatori per Windows e macOS

1.1 Premessa fondamentale: architetture e compatibilità

Prima di procedere, è essenziale capire che questi simulatori non sono normali applicazioni installabili con un semplice clic. Entrambi sono nati in ambito accademico per Linux, ma hanno esigenze molto diverse legate alla loro "età".

Questi due simulatori architetturali (Gem5 e MarssX86) nascono e vengono sviluppati nativamente per sistemi Unix-like.

- **macOS** è un sistema certificato Unix. "Parla la stessa lingua" di Linux, quindi può eseguire molti di questi programmi nativamente (direttamente dal terminale), con il solo aiuto di alcune librerie esterne.
- **Windows** ha un'architettura completamente diversa, per cui non può eseguire nativamente questi programmi. Per poterli eseguire è quindi necessario utilizzare delle soluzioni di **virtualizzazione o sottosistemi** per creare l'ambiente Unix necessario.

Bisogna anche considerare che, anche avendo un sistema Linux (o Unix), c'è il problema delle librerie software necessarie per compilare il codice (il "Time Gap").

- **Gem5** è un software **moderno e adattivo**, che viene continuamente sviluppato e aggiornato. Funziona senza problemi sui sistemi operativi moderni: su macOS lo si compila direttamente mentre su Windows si usa WSL con un'installazione Linux recente (Ubuntu 20.04/22.04).
- **MarssX86** è un progetto "legacy" (molto vecchio, fermo ad 2012 circa). Richiede compilatori (GCC vecchi) e librerie che sono state rimosse dai sistemi operativi moderni da anni.
  Nè macOS moderno ne WSL possono eseguirlo facilmente, dunque è necessario creare una **Macchina virtuale** (con VirtualBox) che simuli un computer del 2014 con installato **Ubuntu** 14.04.

1.2 Installazione e setup del simulatore Gem5

Per quanto riguarda l'installazione del simulatore Gem5, la soluzione varia leggermente a seconda del sistema operativo utilizzato.

1.2.1 Installazione e setup di Gem5 su Windows

Per installare il simulatore su Windows la soluzione consigliata è **WSL** (Windows Subsystem for Linux). Questo sottosistema permette di avere un terminale Ubuntu vero e proprio integrato in Windows. Dunque, l'installazione si compone di diversi passaggi:

- ***Passo 1*: attivazione di WSL**
  1. Aprire la **PowerShell** come amministratore;
  2. Digitare `wsl --install`, premere invio e attendere il compleatmento del download;
  3. Al termine, seguire le istruzioni a schermo per creare *username* e *password* Ubuntu.

  Una volta fatto ciò, la situazione iniziale nel terminale dovrebbe essere quella mostrata in figura 1, ed eseguendo il comando `pwd` sarà possibile vedere il proprio posizionamento all'interno del sottosistema WSL.

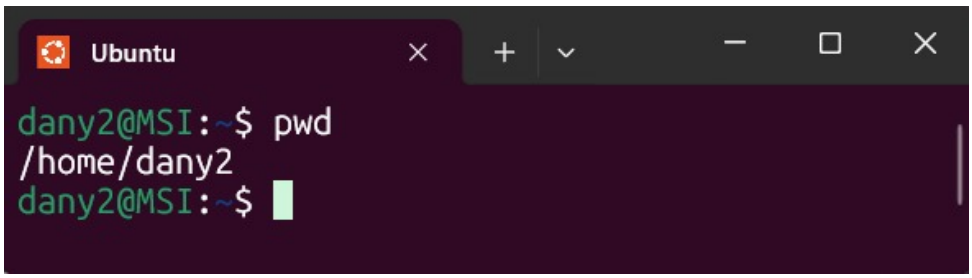


Figure 1: Schermata iniziale del sottosistema WSL

- ***Passo 2*: installazione delle dipendenze**
  Nel terminale di Ubuntu appena aperto incollare questi comandi uno alla volta:

```
//Aggiorna i pacchetti
sudo apt update && sudo apt upgrade -y
```

```
//Installa compilatori, Python e librerie necessarie a Gem5
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev
   libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-
   dev python3-dev python-is-python3 libboost-all-dev pkg-config libpng
   -dev libhdf5-dev libcapstone-dev -y
```

  Note: facendo copia-incolla dei comandi potrebbero verificarsi dei problemi dovuti ai "-" letti dal terminale di Ubuntu come caratteri speciali, in tal caso è consigliato riscriverli a mano.

- ***Passo 3*: scaricare e compilare Gem5**
  Prima di **scaricare** Gem5 è consigliato creare una cartella adatta a poterlo contenere. Quest'operazione viene svolta principalmente perché in futuro dovremo scaricare anche il simulatore architetturale **MarssX86**: così facendo avremo entrambi i simulatori in una cartella apposita.
  Senza spostarsi dalla posizione iniziale `/home/NomeUtente` si va a creare la cartella:

```
mkdir dev_local // Crea la cartella dev_local
cd dev_local // Ci si sposta dentro dev_local per scaricare gem5
```

  Una volta posizionati in `/home/utente/dev_local` è possibile utilizzare il comando `git clone` per scaricare il simulatore gem5 dal repository ufficiale

```
//Clonare il repository del progetto nel proprio workspace
git clone https://github.com/gem5/gem5.git
```

  Quando la clonazione sarà completata, all'interno di `dev_local` sarà presente un'altra cartella nominata "gem5" al cui interno saranno presenti tutti i file necessari per la compilazione del simulatore.

  Prima di far partire la compilazione è necessario fare alcune **premesse importanti**.
  Utilizzare un calcolatore con 16GB o meno di memoria RAM non basterà: Gem5 è enorme e usa template C++ complessi, ogni core che compila può occupare anche 2-3 GB di RAM. Inoltre, considerando che Windows utilizza parte di questa RAM, la situazione si complica ulteriormente.
  Una possibile soluzione a questo problema, nel caso in cui non si disponesse di un calcolatore

con la quantità di RAM necessaria, è quella di "truccare" WSL per dargli più memoria usando il disco fisso come se fosse RAM (operazione di Swap della memoria):

1. Aprire il blocco note di windows;
2. Incollare esattamente queste righe:

   ```
   [wsl2]
   memory=10GB
   swap=16GB
   ```

   In questo modo stiamo dando a WSL massimo 10GB di RAM vera, ma aggiungiamo 16GB di "memoria lenta" su disco per evitare che la compilazione vada in crash.
3. Salvare il file con nome `.wslconfig` (assicurandosi che non ci sia .txt alla fine!) nella propria cartella utente principale: `C:\Utenti\NomeUtente\.wslconfig`
4. Aprire la PowerShell (sempre in modalità amministratore) e spegnere WSL per applicare la modifica con il comando `wsl --shutdown`

Una volta fatte queste modifiche, è finalmente possibile far partire la compilazione: riaprendo il terminale Ubuntu e posizionandosi in `/home/NomeUtente/dev_local/gem5` deve essere scritto il seguente comando:

```
scons build/ARM/gem5.opt -j 1
```

Tramite di esso la compilazione viene fatta partire utilizzando un solo core della cpu. Nonostante l'utilizzo dello "Swap" con la memoria del disco, la compilazione può essere talmente pesante che utilizzando più di un core per la compilazione si rischierebbe di saturare la RAM mandando in crash la compilazione. È **assolutamente** "**normale**" che i tempi di compilazione varino da 1h a 2h in base alla cpu che si ha a disposizione.

Note: non appena la compilazione del simulatore ARM termina, per incominciare la compilazione del simulatore RISC-V basterà reinviare il comando `scons` quì sopra, sostituendo `ARM` con `RISCV`.

- ***Passo 4*: test di funzionamento**
  Per testare che il simulatore sia stato installato correttamente è possibile lanciare un comando di prova già incluso al suo interno.
  Sempre posizionati in `/home/NomeUtente/dev_local/gem5`, lanciare questo comando:

  ```
  ./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-
     progs/hello/bin/arm/linux/hello
  ```

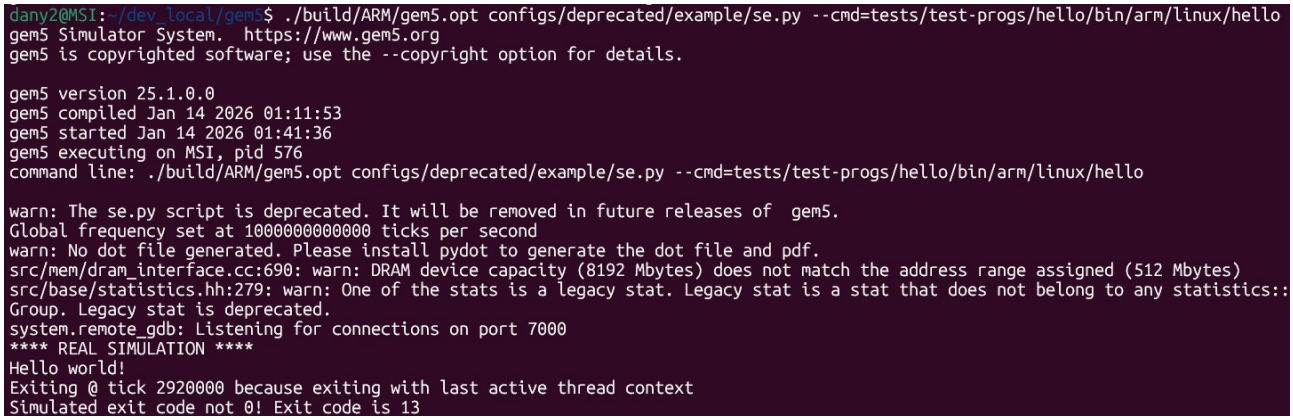


Figure 2: test di simulazione per Gem5

L'esecuzione di questo comando comporterà la creazione, all'interno della cartella "m5out" situata all'interno della cartella "gem5", di **due file fondamentali**:

- "***stats.txt***": Contiene tutte le statistiche (cicli, cache miss, istruzioni eseguite, ecc.). È il file di testo enorme che si dovrà analizzare dopo l'esecuzione vera e propria del workload;
- "***config.ini***": Contiene la descrizione dettagliata dell'hardware appena simulato (utile per verificare se è stata davvero usata la CPU che si voleva).

1.2.2 Installazione e setup di Gem5 su macOS

Sebbene sia teoricamente possibile compilare nativamente su macOS, la gestione delle dipendenze (in particolare Python e Protobuf) risulta spesso instabile a causa degli aggiornamenti di sistema Apple.

Per garantire la **massima riproducibilità** e stabilità, la soluzione raccomandata per macOS è l'utilizzo di un container **Docker** basato su Ubuntu 22.04. Come per WSL, questo approccio permette di avere un ambiente di sviluppo basato su Linux nativo.

- ***Passo 1*: preparazione del container**
  1. Scaricare e installare Docker desktop dal sito ufficiale `https://www.docker.com/`;
  2. È consigliato aprire il terminale posizionandosi nella cartella del proprio apple account come mostrato in figura 3;

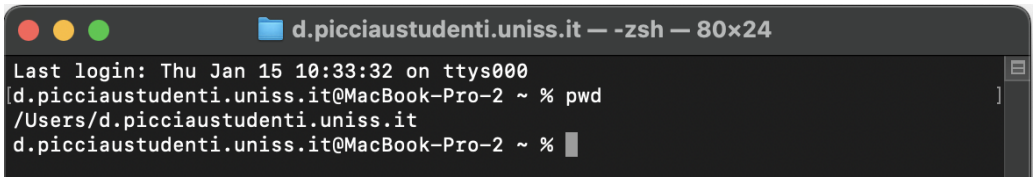


Figure 3: posizionamento iniziale per l'installazione di docker

  3. Creare il proprio workspace che conterrà il simulatore Gem5 e in futuro MarssX86.

```
mkdir dev_local
cd dev_local
```

- ***Passo 2*: installazione delle dipendenze**
  1. All'interno di `/Users/NomeUtente/dev_local`, installare tutte le dipendenze necessarie per la creazione del container di Ubuntu 22.04:

```
docker pull ghcr.io/gem5/ubuntu-24.04_all-dependencies:v25-1
```

  2. Una volta che il workspace è stato creato è possibile far partire il container di Ubuntu (sempre stando posizionati in `/Users/NomeUtente/dev_local`):

```
docker run --volume /Users/NomeUtente/dev_local/gem5:/gem5 --rm -it
   ghcr.io/gem5/ubuntu-24.04_all-dependencies:v24-0
```

     Il completamento di questo comando dovrebbe aprire una riga di comando di questo tipo, ad indicare che siamo all'interno del container Ubuntu.

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

- ***Passo 3*: scaricare e compilare Gem5**
  All'interno del container Ubuntu appena creato, è possibile **scaricare** il repository ufficiale tramite il comando `git clone`:

```
root@{...}:/#  git clone https://github.com/gem5/gem5.git
```

Verrà creata una cartella "gem5" al cui interno saranno al cui interno saranno presenti tutti i file necessari per la compilazione del simulatore.

Come specificato al "***Passo 3***" del capitolo 1.2.1 la compilazione richiede una grande quantità di RAM per cui è richiesto un calcolatore con 16 o più GB di memoria RAM (solo per la compilazione di Gem5!). Dopo alcuni test da noi effettuati è risultato che la compilazione su un calcolatore con 8GB di RAM potrebbe impiegare anche mezza giornata!
Proprio per questo motivo, anche disponendo di 16GB di memoria RAM è necessario andare nella GUI di Docker `impostazioni` $\rightarrow$ `resources` $\rightarrow$ `advanced`, e da qui cambiare le seguenti impostazioni:

- ***Memory limit***: impostiamo la capacità massima di memoria RAM che Docker può utilizzare. È consigliato impostarlo quasi al massimo, (ad esempio, con 16 GB di RAM a disposizione impostarlo a $14/15$) per permettere al sistema operativo (macOS) di "sopravvivere"
- ***Swap***: Invece, tramite questa impostazione, indichiamo quanto spazio di "memoria lenta" prendere dal disco nel caso in cui la RAM vada in saturazione a causa della pesantezza di compilazione.

Una volta che il repository è stato clonato ci si sposta all'interno della cartella "gem5" per far partire la **compilazione del simulatore**, utilizzando 1 o al massimo 2 core per evitare che la compilazione vada in crash a causa della saturazione della RAM a dispozione.

```
// Mi sposto all'interno della cartella gem5
root@{...}:/# cd gem5

// Compilazione ARM
root@{...}:/gem5# scons build/ARM/gem5.opt -j1
```

Come per Windows, anche la compilazione su macOS può impiegare da 1h a 2h, ed è "**normale**" vista la mole di file da scaricare.
Note: non appena la compilazione del simulatore ARM termina, per incominciare la compilazione del simulatore RISC-V basterà reinviare il comando `scons` quì sopra, sostituendo `ARM` con `RISCV`.

***Passo 4*: test di funzionamento**
Per testare che il simulatore sia stato installato correttamente è possibile lanciare un comando di prova già incluso al suo interno.
Sempre posizionati in `root@1ab1ef08f2f:/gem5#`, lanciare questo comando:

```
./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-
   progs/hello/bin/arm/linux/hello
```

L'esecuzione di questo comando comporterà la creazione, all'interno della cartella "m5out" situata all'interno della cartella "gem5", di **due file fondamentali**:

- "***stats.txt***": Contiene tutte le statistiche (cicli, cache miss, istruzioni eseguite, ecc.). È il file di testo enorme che si dovrà analizzare dopo l'esecuzione vera e propria del workload;
- "***config.ini***": Contiene la descrizione dettagliata dell'hardware appena simulato (utile per verificare se è stata davvero usata la CPU che si voleva).

1.3 Installazione e setup del simulatore MarssX86

Nonostante sia un processo abbastanza lungo e tedioso la scelta consigliata per effettuare l'installazione del simulatore, sia per Windows che per macOS, è utilizzare Docker.

1.3.1 Installazione e setup di MarssX86 su Windows

L'utilizzo di Docker è la scelta raccomandata per mantenere un flusso di lavoro unificato. Grazie all'integrazione con WSL 2, è possibile avviare l'ambiente "legacy" necessario per MarssX86 direttamente dallo stesso terminale utilizzato per Gem5 (come visto al passo 1 del capitolo 1.2.1), senza dover configurare macchine virtuali esterne pesanti.

- ***Passo 1*: Configurazione (WSL + Docker)**
  1. Scaricare e installare Docker desktop dal sito ufficiale `https://www.docker.com/`;
  2. Aprire docker desktop su Windows;
  3. Andare nelle impostazioni (icona dell'ingranaggio in alto a destra);
  4. Spostarsi nella sezione **resources** e poi in **WSL integration**;
  5. Assicurarsi di attivare la casella "Ubuntu" per abilitare l'integrazione;
  6. Cliccare su "*apply e restart*"

  A questo punto, aprendo il terminale WSL, è possibile verificarne il funzionamento tramite il comando `docker -version`.

- ***Passo 2*: preparazione del container**
  1. Nel terminale WSL spostarsi all'interno della cartella `/home/NomeUtente/dev_local` (come fatto al passo 3 del capitolo 1.2.1), e creare una cartella per il progetto Marss con al suo interno il file di definizione dell'ambiente:

     ```
     mkdir marss
     cd marss
     nano Dockerfile
     ```

     All'interno dell'editor che si aprirà dopo l'ultimo comando (`nano Dockerfile`), incollare il seguente contenuto che definisce un ambiente Ubuntu 18.04 con tutte le dipendenze necessarie.
     Nonostante MarssX86 sia stato scritto in un ambiente Ubuntu 14.04 (risalente al 2012), non è possibile replicare tale ambiente, poiché da molti anni i server hanno cancellato tutti i file. Dunque una possibile strategia è quella di usare un sistema operativo più recente (Ubuntu 18.04) i cui server funzionano perfettamente, ma installare manualmente i "vecchi" compilatori.

     ```
     # Usiamo Ubuntu 18.04 che ha server ancora attivi e stabili
     FROM ubuntu:18.04

     # Aggiorniamo i repository (funzionera' senza errori 404)
     RUN apt-get update

     # Installiamo i tool di base e specificamente GCC 4.8 / G++ 4.8
     # MarssX86 richiede queste versioni antiche per compilare
     RUN apt-get install -y build-essential git scons zlib1g-dev python2.7
         python-minimal g++ \
         vim libsdl1.2-dev gcc-4.8 g++-4.8
     ```

```
# Configuriamo il sistema per usare GCC 4.8 come default assoluto
# Questo "inganna" Marss facendogli credere di essere su un vecchio
   sistema
RUN update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8
   100 && \
    update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8
       100

# Cartella di lavoro
WORKDIR /root/marss

CMD ["/bin/bash"]
```

2. Rimanendo posizionati in `/home/NomeUtente/dev_local/marss`, lanciare il comando di build per l'ambiente Ubuntu. Verrà richiesta una password, inserire quella dell'account Ubuntu come fatto al "***Passo 1***" del capitolo 1.2.1.

```
sudo docker build --no-cache -t marss_env .
```

3. Infine, è possibile creare l'ambiente Ubuntu scelto. Servirà la password.

```
sudo docker run -it --name marss_container -v "$PWD":/root/marss
   marss_env
```

Nota importante: l'aggiunda del comando `-v "$PWD":/root/marss` crea un collegamento diretto tra la cartella in cui ci si trova (`/home/NomeUtente/dev_local/marss`) nel disco e la cartella di lavoro "virtuale" del container. Senza di esso non si vedrebbero le modifiche effettuate nel container, costringendo a copiare ogni file che si vuole utilizzare nel disco del computer, tramite comandi aggiuntivi. Ovviamente, il container rimane necessario per compilare/eseguire i file dell'ambiente Ubuntu scelto.

Una volta fatto ciò, si dovrebbe essere riusciti ad entrare nell'ambiente di simulazione, e a schermo dovrebbe vedersi questo:

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

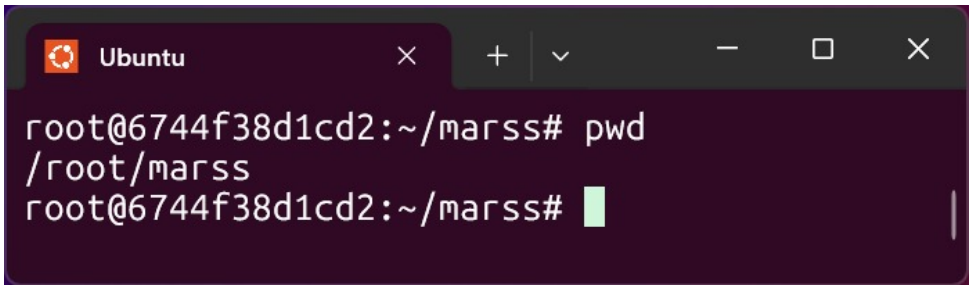


Figure 4: ambiente di simulazione del MarssX86

Inserendo subito il comando `"pwd"` si potra vedere come ci si trova in `/root/marss` come specificato nel Dockerfile.

4. Per **tornare alla bash di Ubuntu** basterà scrivere il comando `"exit"` da qualsiasi posizione, mentre per rientrare nell'ambiente già creato servirà un comando differente da lanciare (posizionati in `/home/NomeUtente/dev_local/marss`):

```
docker start -ai marss_container
```

Invece, per rimuovere il container serve prima sapere il container id. Nell'ambiente Ubuntu (quindi dopo avere fatto "exit") utilizzare il seguente comando:

```
docker ps
```

Subito dopo

```
docker rm <container id>
```

- ***Passo 3*: scaricare e compilare MarssX86**
  Dopo essere riusciti ad entrare dentro l'ambiente di simulazione, bisogna **scaricare il repository** del progetto ufficiale (posizionati in `/root/marss`):

```
root@{...}:~/marss# git clone https://github.com/avadhpatel/marss.git
```

Una volta completata la clonazione, all'interno di `/root/marss` verrà scaricata un ulteriore cartella "marss" contenente i file reltivi al repository.
Anche in questo caso la compilazione risulta abbastanza pesante a causa della quantità di RAM richiesta. Fortunatamente, Docker desktop su Windows eredita da WSL il file `.wslconfig` (***passo 3*** del capitolo 1.2.1) creato per fare in modo che la compilazione non vada in crash.

Prima di **procedere con la compilazione** ci sono alcune problematiche che vanno risolte, relative alle discrepanze tra l'utilizzo di comandi moderni e compilatori vecchi oppure causate da librerie di sicurezza moderne (già presenti nella versione 18.04 di Ubuntu che stiamo utilizzando) che bloccano la compilazione del file finale di compilazione. Queste problematiche sono sorte in fase di compilazione stessa, e avendole individuate, è necessario risolverle per garantire il funzionamento della compilazione.

1. Posizionarsi in `/root/marss/marss/ptlsim` e aprire il file "*SConstruct*" che rappresenta il "cuore" del simulatore PTLsim:

```
root@{...}:~/marss/marss/ptlsim# apt-get install -y nano
root@{...}:~/marss/marss/ptlsim# nano SConstruct
```

2. Una volta all'interno dell'editor di testo nano per il file "*SConstruct*" cercare la riga:

```
env.Append(CCFLAGS = ['-fdiagnostics-color=always'])
```

ed eliminarla o commentarla utilizzando un "#". Per aiutarsi è possibile entrare in modalità ricerca con la combinazione di comandi `ctrl + w`.

Subito sotto questa riga, ne dovrebbe essere presente un'altra di questo tipo:

```
env.Append(CCFLAGS = ' -std=gnu++11 ')
```

la quale deve essere modificata in questo modo:

```
env.Append(CCFLAGS = ' -std=gnu++11 -U_FORTIFY_SOURCE ')
```

In questo modo stiamo dicendo al compilatore di "*disabilitare i controlli di sicurezza moderni per il codice vecchio del repository*".
Premere `ctrl+o` per salvare le modifiche, invio, e `ctrl+x` per uscire dall'editor di testo.

3. Tornati nell'ambiente di simulazione bisogna effettuare lo stesso procedimento nel file "*SConstruct*" principale, quindi non quello presente in `/root/marss/marss/ptlsim` ma in `/root/marss/marss/`, quindi:

```
root@{...}:~/marss/marss# nano SConstruct
```

Una volta all'interno dell'editor di testo, bisogna scendere nel file fino a trovare una instruzione condizionale di questo tipo:

```
if int(pretty_printing) :
    base_env = Environment(
            CXXCOMSTR = compile_source_message,
            CREATECOMSTR = create_header_message,
            CCCOMSTR = compile_source_message,
            SHCCCOMSTR = compile_shared_source_message,
            SHCXXCOMSTR = compile_shared_source_message,
            ARCOMSTR = link_library_message,
            RANLIBCOMSTR = ranlib_library_message,
            SHLINKCOMSTR = link_shared_library_message,
            LINKCOMSTR = link_program_message,
            )
else:
    base_env = Environment()
```

Questo blocco serve a creare l'ambiente di compilazione (base_env). Bisogna intercettare quella variabile appena è stata creata e aggiungere il "*flag salvavita*" che disabilita i controlli di sicurezza troppo moderni. Dopo l'else bisogna aggiungere le seguenti righe:

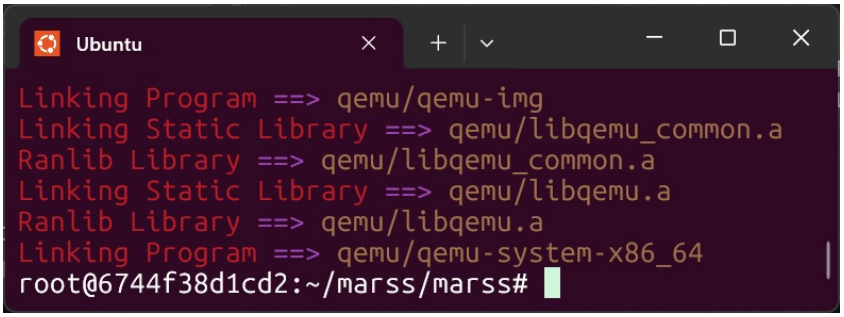
```
...
...

else:
    base_env = Environment()

base_env.Append(CFLAGS = ['-U_FORTIFY_SOURCE'])
base_env.Append(CCFLAGS = ['-U_FORTIFY_SOURCE'])
```

Fatto ciò, nuovamente `ctrl+o` per salvare le modifiche, invio, e poi `ctrl+x` per tornare nell'ambiente di simulazione.

Effettuate queste modifiche è finalmente possibile **compilare** tramite il comando:

```
root@{...}:~/marss/marss# scons -Q C=1 debug=0
```

Come risultato finale si dovrebbe visualizzare un comando di questo tipo:



Al termine della compilazione è possibile verificare la presenza dell'eseguibile tramite il seguente comando:

```
root@{...}:~/marss/marss# ./qemu/qemu-system-x86_64 --version
```

Se in risposta si riceve un qualcosa del tipo:

```
QEMU emulator version 0.14.1, Copyright (c) 2003-2008 Fabrice Bellard
```

allora il simulatore è pronto all'uso.

- ***Passo 4*: installazione immagine disco rigido**
  Dopo aver completato la fase di installazione del simulatore con l'ambiente Ubuntu, si passa alla fase di installazione del sistema operativo vero e proprio.
  A differenza di Gem5 che carica direttamente l'eseguibile, MarssX86 è un simulatore Full System (FS): simula un PC intero (BIOS, scheda madre, disco rigido). Quindi, per farlo funzionare, ci serve un'immagine del disco rigido con dentro un Sistema Operativo (Ubuntu vecchio).
  1. Si rende necessario quindi, preparare la cartella dei dischi. Posizionati in `root/marss/` per creare una cartella "*disks*":

     ```
     mkdir disks
     cd disks
     ```

  2. Una volta creata la cartella, spostarsi in `root/marss/marss/disks` e installare il comando "*wget*" per poter poi installare il sistema operativo da emulare tramite l'ambiente Ubuntu preparato precentemente.

     ```
     root@{...}:~/marss/marss/disks# apt-get install -y wget bzip2
     ```

     Installerà tutte le librerie necessarie al suo funzionamento, ci sarà da aspettare qualche secondo.
  3. A questo punto è tutto pronto per installare il sistema operativo.
     Il sistema operativo scelto è ottenibile al seguente link `https://people.debian.org/~aurel32/qemu/amd64/debian_squeeze_amd64_standard.qcow2`.
     Si tratta di *Debian Squeeze*, una distribuzione di Linux rilasciata nel febbraio del 2011.
     La scelta di questa distribuzione specifica è dettata da vincoli di compatibilità:
     - **Sincronizzazione Temporale**: MarssX86 è stato sviluppato intorno al 2011-2012, basandosi su una versione di QEMU (0.14) di quel periodo. I simulatori Full System simulano l'hardware di quell'epoca, e Debian Squeeze, essendo del 2011, "parla la stessa lingua" del simulatore.
     - **Leggerezza (no graphic)**: l'immagine scaricata è una versione "Standard/Server", priva di interfaccia grafica (niente finestre, niente mouse), e questo è cruciale poiché riduce il consumo di RAM e CPU simulata

     Dunque, per installare la distribuzione scelta si utilizza il seguente comando:

     ```
     root@{...}:~/marss/marss/disks# wget https://people.debian.org/~
        aurel32/qemu/amd64/debian_squeeze_amd64_standard.qcow2
     ```

  4. Nonostante l'immagine del disco sia idonea per MarssX86, è necessaria una versione precedente. L'immagine scaricata è in formato QCOW2 versione 3 (introdotto da QEMU 1.1 nel 2012), ma MarssX86 usa QEMU 0.14 (2011), che conosce solo la versione 2.

Per risolvere questo problema legato alla versione, è necessario effettuare un "downgrade" dell'header del file del disco virtuale per renderlo leggibile dal vecchio simulatore:

```
root@{...}:~/marss/marss/disks# apt-get update && apt-get install -y
   qemu-utils
```

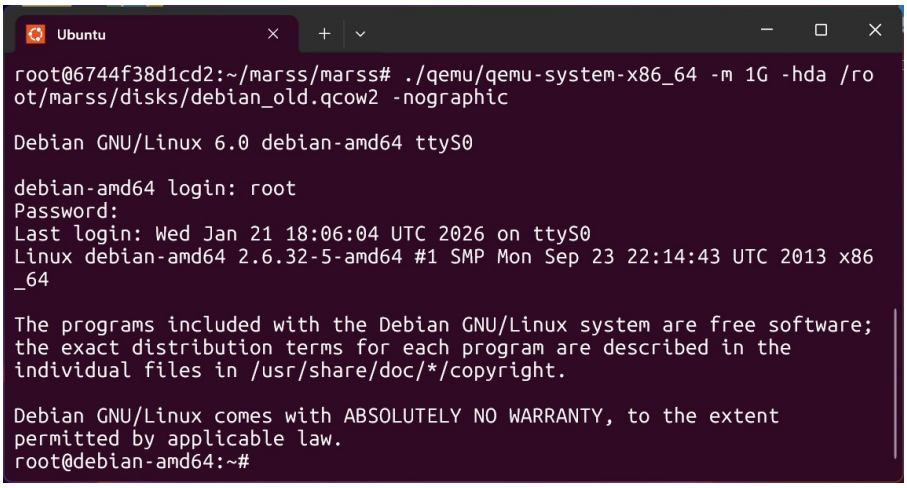
```
root@{...}:~/marss/marss/disks# qemu-img convert -f qcow2 -O qcow2 -o
    compat=0.10 debian_squeeze_amd64_standard.qcow2 debian_old.qcow2
```

`compat=0.10` forza il formato a essere leggibile dai simulatori del 2011, creando una nuova immagine del disco *debian_old.qcow2*.

5. Ora, per far partire a distribuzione Debian, basterà lanciare il seguente comando, posizionati in `/root/marss/marss`:

```
root@{...}:~/marss/marss# ./qemu/qemu-system-x86_64 -m 1G -hda /root/
   marss/disks/debian_old.qcow2 -nographic
```

Verrà richiesto login e password, le credenziali saranno "*root*" per entrambi i campi.



Ruolo nell'architettura della simulazione

La procedura si presenza abbastanza complessa, dunque, per chiarire il funzionamento complessivo, facciamo un recap:

1. **Host Fisico**: il PC Windows (server WSL) o il PC macOS.
2. **Container Docker**: l'ambiente che contiene le librerie necessarie per compilare ed eseguire il simulatore.
3. **Simulatore (MarssX86/QEMU)**: il programma che finge di essere un computer fisico (crea CPU, RAM e Disco finti).
4. **Guest OS (Debian Squeeze)**: Il sistema operativo che "vive" dentro il disco finto e che esegue effettivamente il codice C.

1.3.2 Installazione e setup di MarssX86 su macOS

Dal momento che, per simulare MarssX86 su Windows si sta utilizzando Docker, gran parte dei passaggi da effettuare su macOS rimangono identici.

Tuttavia, per impostazione predefinita, Docker su un dispositivo che monta un processore apple silicon crea un contenitore ARM64 (aarch64). MarssX86 è un simulatore più vecchio

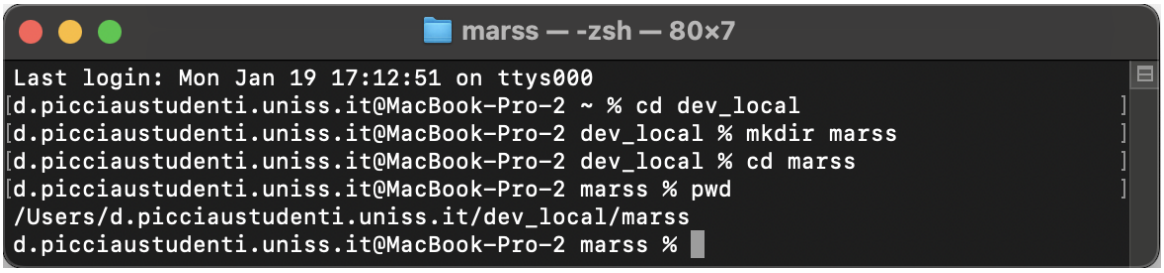
progettato esclusivamente per architetture x86_64 (Intel/AMD), dunque ne consegue che se si andasse ad effettuare il seguente script di build (SConstruct):

```
root@{...}:~/marss/marss# scons -Q C=1 debug=0
```

verrebbe rilevato che la CPU è basata su ARM e il processo verrebbe interrotto non sapendo come effettuare la compilarlazione.

A questo punto, facendo riferimento al capitolo 1.3.1 è necessario eliminare il "***passo 1***" poiché WSL non viene utilizzato, revisionare il "***passo 2***", modificando semplicemente i comandi di build e creazione dell'ambiente Ubuntu, e mantenere il "***passo 3***" invariato.

1. Come spiegato al "***passo 1***" del capitolo 1.2.2 scaricare Docker dal link fornito. Inoltre, è consigliato creare una cartella "*marss*" all'interno di "*dev_local*", per mantenere separati i flussi di lavoro tra i due simulatori.
   Per fare ciò è necessario essere posizionati in `/Users/NomeUtente/dev_local/marss`

2. All'interno della cartella "*marss*", come fatto per Windows, bisogna creare il "Dockerfile" all'interno del quale definire l'ambiente Ubuntu 18.04, dunque:

   ```
   nano Dockerfile
   ```

   e in seguito inserire le seguenti righe di codice:

   ```
   # Usiamo Ubuntu 18.04 che ha server ancora attivi e stabili
   FROM ubuntu:18.04

   # Aggiorniamo i repository (funzionera' senza errori 404)
   RUN apt-get update

   # Installiamo i tool di base e specificamente GCC 4.8 / G++ 4.8
   # MarssX86 richiede queste versioni antiche per compilare
   RUN apt-get install -y build-essential git scons zlib1g-dev python2.7
      python-minimal g++ \
       vim libsdl1.2-dev gcc-4.8 g++-4.8

   # Configuriamo il sistema per usare GCC 4.8 come default assoluto
   # Questo "inganna" Marss facendogli credere di essere su un vecchio
      sistema
   RUN update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 100
       && \
       update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 100

   # Cartella di lavoro
   WORKDIR /root/marss
   ```

```
CMD ["/bin/bash"]
```

3. Rimanendo sempre posizionati in `/Users/NomeUtente/dev_local/marss` lanciare il comando di build per l'ambiente Ubuntu.

   ```
   docker build --platform linux/amd64 --no-cache -t marss_env .
   ```

   Questo dice a Docker di fingere che sia una macchina Intel durante la costruzione.
   Note: questo passaggio richiederà più tempo rispetto a prima a causa dell'emulazione.
4. Subito dopo creare e far partire il container, specificando la piattaforma:

   ```
   docker run --platform linux/amd64 -it --name marss_container -v "$PWD
      ":/root/marss marss_env
   ```

   Una volta lanciato questo comando si dovrebbe essere riusciti ad entrare nel container e a terminale si dovrebbe visualizzare questo:

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

5. Dopo essere riusciti ad entrare all'interno del container, i seguenti passaggi sono tali e quali a quelli descritti dal "***passo 3***" in poi del capitolo 1.3.1.

2 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.

2.1 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;
}
```

2.2 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.

2.2.1 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.

2.2.2 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.

2.3 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.

2.4 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.

2.4.1 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.

2.5 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.

2.5.1 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
```

2.5.2 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.

2.5.3 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=O3CPU \
  --caches --l2cache \
  --l1d_size=128KB --l1i_size=128KB \
  --l2_size=2MB \
  --sys-clock=2GHz \
  --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=MinorCPU \
  --caches --l2cache \
  --l1d_size=128KB --l1i_size=128KB \
  --l2_size=2MB \
  --sys-clock=2GHz \
  --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=O3CPU \
  --caches --l2cache \
  --l1d_size=32KB --l1i_size=32KB \
  --l2_size=256KB \
  --sys-clock=2GHz \
  --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=2MB \
  --sys-clock=2GHz \
  --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=2MB \
  --sys-clock=2GHz \
  --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=2GHz \
  --cmd=./matrix_mul_riscv
```

2.6 Esecuzione del workload per X86 (MarssX86)

3 Analisi dei dati raccolti

I dati raccolti non verranno analizzati solo in valore assoluto, ma verranno **normalizzati rispetto alla propria baseline**. Questo approccio permette di rispondere a domande del tipo: "Quale architettura beneficia maggiormente di un aumento della cache?" o "Quale ISA risulta più efficiente a parità di risorse?". Il confronto finale avverrà dunque su due livelli:

1. **Intra-ISA**: Confronto tra le tre build della stessa architettura per valutarne la scalabilità.
2. **Inter-ISA**: Confronto tra i rapporti di performance delle tre architetture rispetto alle rispettive baseline, per determinare l'efficienza relativa dei set di istruzioni.