Murat Tökez

gem5 RISCV full-system Simulation

Abbreviations

PLIC: Platform Level Interrupt Controller
CLINT: Core Local Interruptor
RTC: Real Time Clock
DTB: Device Tree Blob
PIO: Programmed Input/Output
PMA: Physical Memory Attribute
mmu: Memory Management Unit

Introduction

This document focuses on how to perform a full system simulation on the gem5 simulator, how to start a RISC-V virtual machine and how to run CoreMark-PRO binaries on the machine.

1. Project Designs

The following file hierarchy is used and recommended for this project.

```
-riscv-fs/
   |__gem5/                  #gem5 source code
   |
   |__gem5-fs-scripts/       #gem5 system scripts to run the full-system simulation
   |
   |__riscv/
         |__build/           #to be used in simulation (bbl bootloader, linux kernel, disk image)
         |__bin/             #riscv benchmark binaries (coreMarkPro was used in this project)
         |__src/
            |__linux/        #linux kernel
            |__riscv-pk/     #RISC-V proxy kernel (bbl bootloader)
            |__riscv64-sample #UCanLinux
```

2. Tool and Compiler installations

2.1 RISC-V GNU Toolchain

RISC-V GNU Toolchain is used to compile RISC-V source code. First of all, the necessary packages should be installed before installation. It should be noted that this project is working on Ubuntu 22.04 version. If you are using a different version, you may need to do different installations.

$ sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev

The process of pulling and installing the github source code can then be done as follows.

$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

$ cd riscv-gnu-toolchain (or whatever folder you installed it in and under whatever name)

$ ./configure --prefix=/opt/riscv --enable-multilib

$ make linux -j$(nproc)

Path must be exported after the installation is complete. To make this permanent, .bashrc can be updated as follows.
$ cd $HOME

$ sudo vim .bashrc

The following lines are added to the last lines of .bashrc.

export RISCV=/opt/riscv
export PATH=/opt/riscv/bin:$PATH

Then .bashrc is sourced and the GNU Toolchain is ready to be used.

$ source .bashrc

2.2 gem5

Pre-installation packages are installed.

$ sudo apt install build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev python3-dev libboost-all-dev pkg-config

Installation is done as follows.

$ cd riscv-fs
$ git clone https://gem5.googlesource.com/public/gem5

$ cd gem5
$ scons build/RISCV/gem5.opt -j$(nproc)

3. Linux System

3.1 Linux Kernel

In this chapter we will build the Linux system. First, let's download the Linux kernel source code and get our kernel binary. Linux 6.3.1 is used in this project. It is recommended to use the latest and stable one.

https://www.kernel.org/
and download the latest release linux kernel. Navigate to the relevant directory where the compressed files are downloaded.

$ tar -xf linux-6.3.6.tar.xz -C $HOME/riscv-fs/riscv/src/linux

$ cd src/linux/linux-6.3.6

With the defconfig argument, the sample config file that comes with the source code and is located under linux/arch/riscv/configs is copied under linux/ directory as .config with the following make defconfig command.

$ make ARCH=riscv defconfig

Then the kernel's config information can be examined or updated with the menuconfig argument.

$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- menuconfig

The kernel is compiled with the following all entry.

$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- all -j$(nproc)

If the build goes smoothly, both the kernel and kernel modules will be generated. The new kernel will be generated with the name vmlinux just below the source code. Let's move this kernel to our build directory.

$ cp vmlinux $HOME/riscv-fs/riscv/build

3.2 Berkeley Bootloader (bbl)

When a machine is turned on, the programs that run before the operating system are called bootloaders. The main purpose of the bootloader is to load the Linux kernel. Initially U-boot was considered as a bootloader, but this was abandoned when more riscv project resources were available using riscv-pk.

Below are the steps to download and install the riscv-pk project.

$ cd $HOME/riscv-fs/riscv/src
$ git clone https://github.com/riscv/riscv-pk.git

$ cd riscv-pk
$ mkdir build
$ cd build

**$ ../configure --host=riscv64-unknown-linux-gnu **
--with-payload=$HOME/riscv-fs/riscv/build/vmlinux --prefix=$RISCV

$ make -j$(nproc)
$ make install

With the --with-payload option in the .configure command, bbl and the Linux kernel are combined to create the bbl image. The bbl image can be viewed as bbl + Linux kernel.

The generated bbl file is copied to the build directory.

$ cp bbl $HOME/riscv-fs/riscv/build

3.3 Root File System

The file system to which the kernel connects immediately after booting is called the root file system. The root file system forms the basic hierarchical structure on which all other files and directories in the computer system are located. The operating system and all user-accessible files are located in subdirectories within the root file system.

Usually in UNIX-like operating systems, the root file system is called '/' (root directory) and all other files and directories are organized hierarchically under this root directory. For example, the '/home' directory contains users' home directories, the '/usr' directory contains applications and utilities on the system.

In this application we will use **busybox** as the root filesystem and the skeleton root filesystem from Nazım Koç's ucanlinux repository. BusyBox provides a set of basic UNIX commands, such as ls, cp, mv, grep, find, tar, gzip, chmod.

First, let's download and install busybox. Here again, make sure to install the latest and stable version of busybox. At the time of writing this document, the latest version was 1.36.

```
$ cd $HOME/riscv-fs/riscv/src
$ git clone git://busybox.net/busybox.git

$ cd busybox
$ git checkout 1_36_stable

$ make menuconfig

$ make CROSS_COMPILE=riscv64-unknown-linux-gnu- all -j$(nproc)
$ make CROSS_COMPILE=riscv64-unknown-linux-gnu- install
```

Now let's download the ucanlinux repository.

```
$ cd $HOME/riscv-fs/riscv/src
$ git clone https://github.com/UCanLinux/riscv64-sample.git
```

In order not to corrupt the skeleton file system, a copy will be made under RootFS. Our example root file system will now be RootFS. The skeleton file system is copied under RootFS as follows.

```
$ cd $HOME/riscv-fs/riscv/src
$ mkdir RootFS
$ cd RootFS
$ cp -a ../riscv64-sample/skeleton/*  .
```

/bin, /sbin, /usr/bin and /usr/sbin directories are copied from busybox to RootFS as follows.

```
$ cd $HOME/riscv-fs/riscv/src/RootFS
$ cp -a $HOME/riscv-fs/riscv/src/busybox/_install/*  .
```

Kernel modules must be installed under RootFS/lib/modules.

$ cd $HOME/riscv-fs/riscv/src/linux/linux-6.3.6

**$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- **
INSTALL_MOD_PATH=$HOME/riscv-fs/riscv/src/RootFS modules_install

Then the libraries in the toolchain should be copied to /lib.

$ cd $HOME/riscv-fs/riscv/src/RootFS
$ cp -a /opt/riscv/sysroot/lib .

Now let's set up our other empty directories in our Linux root file system.

$ cd $HOME/riscv-fs/riscv/src/RootFS
$ mkdir dev home mnt proc sys tmp var

$ cd etc/network
$ mkdir if-down.d if-post-down.d if-pre-up.d if-up.d

Also the m5 library of gem5 should be built and added to the filesystem.

$ cd $HOME/riscv-fs/gem5/util/m5
$ scons riscv.CROSS_COMPILE=riscv64-unknown-linux-gnu- build/riscv/out/m5
$ cp build/riscv/out/m5 $HOME/riscv-fs/riscv/src/RootFS/sbin

The root file system is now ready.

3.4 Creating a Disk Image

In this section, our root file system will be mounted in a disk image and we will also mount the benchmarks in this disk image to make it usable in gem5.

First, a 500Mb free disk is created.

$ cd $HOME/riscv-fs/riscv/out
$ dd if=/dev/zero of=riscv_disk bs=1M count=500

Set up the file system

$ mkfs.ext2 -L riscv-rootfs riscv_disk

There is now a file system in riscv_disk. To access this filesystem, mount it as follows and copy all the information in RootFS to riscv_disk.

$ cd $HOME/riscv-fs/riscv/out
$ sudo mkdir /mnt/rootfs

$ sudo mount riscv_disk /mnt/rootfs

$ sudo cp -a $HOME/riscv-fs/riscv/src/RootFS/* /mnt/rootfs

$ sudo chown -R -h root:root /mnt/rootfs

$ sudo umount /mnt/rootfs

The disk image is ready to be used on gem5. But there is no benchmark binary in it at the moment. Any file, binary is put into our file system as follows.

$ sudo mkdir -p /mnt/old_disk
$ sudo mkdir -p /mnt/new_disk

$ sudo mount riscv_disk /mnt/old_disk
$ sudo mount riscv_coreMarkPro_disk /mnt/new_disk
$ sudo cp -a /mnt/old_disk/* /mnt/new_disk

Copy the benchmark binaries you want to use in this step to /mnt/new_disk. In this project coreMarkPro binaries are used. For more detailed information and to obtain RISC-V coreMarkPro binaries:

https://github.com/eembc/coremark-pro

https://github.com/ccelio/coremarkpro-util-make-riscv

$ sudo cp -a $HOME/riscv-fs/riscv/bin/* /mnt/new_disk

$ sudo umount /mnt/old_disk
$ sudo umount /mnt/new_disk

Now our disk image named riscv_coreMarkPro_disk with our benchmarks is ready in the /out directory.

4. Gem5 full system simulation

The kernel and disk image we will use in the full system simulation are now ready. This section will explain how to write a riscv full system script.

System Creation
In the script, a system class called RiscvSystem is created. This class inherits from the gem5 System class, which models a RISC-V system and defines how the basic components (e.g. cpu, cache, memory controller) are built and connected to each other. As a platform
HiFive() selected PLIC, CLINT, UART, VirtIOMMIO classes taken from here.

CPU and Cache Creation
The 'createCPU' and 'createCacheHierarchy' functions create the processors and cache hierarchy in the system. Each processor is assigned a separate L1 cache (both data and instruction cache). These caches are connected to the L2 and L3 caches via an xbar (crossbar switch), which allows multiple components to transfer data simultaneously. There should also be an 'mmuCache' in each core. An MMU controls how and where the processor uses memory, usually managing the conversion from virtual to physical memory. The MMU converts virtual memory addresses to physical memory addresses, which is necessary for an operating system to be able to run multiple processes simultaneously. mmuCache allows these conversions to be done quickly and efficiently.

Memory Controller and Interrupts
The 'createMemoryControllers' function creates the memory controllers in the system. The memory controller manages data transfers between the CPU and memory. "The function 'setupInterrupts' creates the interrupt controllers of the processors. The interrupt controller manages system interrupts and determines which processes are prioritized by the CPU.

Devices and I/O
The 'initDevices' function creates and configures the I/O devices in the system. This includes a platform-level interrupt controller (PLIC), a kernel-level interrupt controller (CLINT), a real-time clock (RTC), Peripheral Component Interconnect (PCI), UART, PMA (Physical Memory Attribute) checker and a block device (disk). These devices communicate with memory and other components via membus and iobus.

PLIC is used to handle interrupts from all sources on a processor. It determines which interrupts a processor needs to handle at a given time and prioritizes them. PLIC connects global interrupt sources to the interrupt target, the kernel. PLIC consists of "PLIC core" and "Interrupt gateways". There is one interrupt gateway for each interrupt source. A global interrupt sends one of its sources to the interrupt gateways. The interrupt gateways process the interrupt signal from each source and send a single interrupt request to the PLIC core. The PLIC core contains interrupt enable (IE) bits to enable individual interrupt sources in the PLIC. The PLIC core has pending interrupt bits to indicate that an interrupt is waiting to be processed. In addition, the PLIC core performs interrupt prioritization/arbitration. Each interrupt source is assigned a separate priority. The PLIC core latches the interrupt request to the Interrupt Pending bits (IP). When the pending interrupt priority reaches a certain threshold per destination, the PLIC core transmits an interrupt to the interrupt destination with an interrupt notification. The PLIC Claim/Complete register holds the highest priority interrupt waiting to be processed.

CLINTis a local interrupt controller used by a processor core in the RISC-V architecture. It is usually connected to a core's internal clock (timer) and software interrupts. Timer interrupts are automatically triggered at specific time intervals and pause the processor to perform a specific operation. Software

interrupts are usually triggered by software when a processor needs to perform a specific software service.

RTC is a hardware component used in a computer to monitor real time. It continues to track the correct time even when the system is turned off. The RTC is often used to perform time-related functions, such as setting timestamps for files or waking up the computer at a programmed time.

PCI is a standard for communicating data and devices between a computer's motherboard and additional hardware. The Programmed Input/Output (PIO) mode of the platform's PCI host must be connected to iobus (self.platform.pci_host.pio = self.iobus.mem_side_ports). PIO is a data transfer mode where the processor controls the data transfer. In this mode, data moves directly and actively between the processor and system memory.

PMA checker checks various physical memory characteristics. It makes sure that the processor is operating in accordance with valid memory access rules. For example, if a processor tried to write to a memory region that a PMA has marked as read-only, the PMA Checker stops this operation. In summary, PMA Checker's job is to monitor memory accesses and check that accesses to specific memory regions conform to the specifications defined for those regions. This helps with memory protection and prevents the processor from accidentally performing an unauthorized operation on memory.

Creating a Device Tree Blob (DTB)
This function creates a system-specific Device Tree Blob (DTB). The DTB is used in Linux to identify the hardware components of the system.

Kernel and Disk İmage
The prepared kernel and disk image will be given to the script as parameters on the terminal screen. To define the kernel, the necessary operations are done in the script.

```
self.workload.object_file = kernel
```

in the 'initDevices' function to define the disk image

```
image = CowDiskImage(child=RawDiskImage( read_only=True), read_only=False)
image.child.image_file = disk
```

defined.

5. Benchmark environment and testing of systems

CoreMark-pro was used in this project, but any benchmark binary can be used for testing. As mentioned in 3.4, the desired benchmark binaries can be thrown into the disk image and then run in the full-system as desired. In this project, 4 different cases and 9 coreMark-pro binaries were run in each case. The binaries were not executed manually but with a single bash script. The bash script was written on the host system side and then copied into the disk image just like the coreMark-pro binaries and then the script was run on the guest system side. It should be noted here that the poweroff

command is added to the end of the bash script. If this is not done, when the binaries finish running, the guest system will run to no avail and different results may occur for different cases.

6. running gem5 scripts

Now that everything is ready, the full-system simulation can be started. The scripts are run with the following command.

$ cd $HOME/riscv-fs/gem5

$ build/RISCV/gem5.opt --debug-flags=Clint -d m5out/multicore_fs/1L2_256kb_1L3_2mb_fs ../multicore_fs/1L2_256kb_1L3_2mb/run.py --kernel=$HOME/riscv-fs/riscv/build/bbl --disk=$HOME/riscv-fs/riscv/build/riscv_coremarkpro_disk --num_cpus=4

Here **1L2_256kb_1L3_2mb** is the gem5 python script. **-kernel, -disk and -num_cpus** are terminal options given by the python argparse library mentioned in chapter 4.