Murat Tökez

gem5 RISCV full-system Simulation

Kısaltmalar

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

Giriş

Bu döküman, gem5 simülatörü üzerinde tam sistem simülasyonunun nasıl yapılacağı, RISC-V sanal bir makinenin nasıl başlatılacağı ve makine üzerinde CoreMark-PRO binary dosyalarının nasıl çalıştırılacağı konularına odaklanmaktadır.

1. Proje dizini

Bu projede aşağıdaki dosya hiyerarşisi kullanılmıştır ve kullanılması önerilmektedir.

```
-riscv-fs/
   |__gem5/                    #gem5 kaynak kodu
   |
   |__gem5-fs-scripts/         #full-sytem simülasyonunun koşturulacağı gem5 sistem scriptleri
   |
   |__riscv/
         |__build/             #simülasyonda kullanılacak(bbl bootloader, linux kernel, disk image)
         |__bin/               #riscv benchmark binary'leri (bu projede coreMarkPro kullanıldı)
         |__src/
            |__linux/          #linux kernel'ı
            |__riscv-pk/       #RISC-V proxy kernel (bbl bootloader)
            |__riscv64-sample  #UCanLinux
```

2. Tool ve Compiler kurulumları

2.1 RISC-V GNU Toolchain

RISC-V kaynak kodlarını derlemek RISC-V GNU Toolchain kullanılır. Öncelikle kurulum öncesi gerekli olacak paketler kurulmalıdır. Bu projede Ubuntu 22.04 sürümü üzerinde çalışıldığı unutulmamalıdır. Farklı bir sürüm kullanıyorsanız farklı kurulumlar yapmanız gerekebilir.

$ 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

Daha sonra github kaynak kodlarını çekme ve kurma işlemi aşağıdaki şekilde yapılabilir.

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

$ cd riscv-gnu-toolchain (veya hangi klasöre hangi isimle kurduysanız)

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

$ make linux -j$(nproc)

Kurulum tamamlandıktan sonra Path export edilmelidir. Bunu kalıcı hale getirmek için .bashrc şu şekilde güncellenebilir.
$ cd $HOME

$ sudo vim .bashrc

.bashrc'nin son satırlarına şu satırlar eklenir.

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

Daha sonra .bashrc source'lanır ve GNU Toolchain kullanılmaya hazırdır.

$ source .bashrc

2.2 gem5

Kurulum öncesi paketler kurulur.

$ 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

Kurulum şu şekilde yapılır.

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

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

3. Linux Sistemi

3.1 Linux Kernel

Bu bölümde Linux sistemini inşa edeceğiz. İlk olarak Linux kernel (çekirdek) kaynak kodunu indirelim ve kernel binary'imizi elde edelim. Bu projede linux 6.3.1 kullanılmıştır. En güncel ve stable durumda olan kullanılması önerilir.

https://www.kernel.org/
sitesine gidin ve latest release olan güncel linux kernel'i indirin. Sıkıştırılmış dosyaların indirildiği ilgili dizine gidin.

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

$ cd src/linux/linux-6.3.6

defconfig argümanıyla, kaynak kodu ile gelen ve linux/arch/riscv/configs altında bulunan örnek config dosyası, aşağıdaki make defconfig komutu ile, linux/ dizini altına .config adı ile kopyalanır.

$ make ARCH=riscv defconfig

Daha sonra menuconfig argümanı ile çekirdeğin config bilgileri incelenebilir veya güncellenebilir.

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

Çekirdek, aşağıdaki all girişi ile derlenir.

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

Derleme sorunsuz biterse hem ¸cekirdek hem de çekirdek modülleri üretilecektir. Yeni çekirdek, kaynak kodunun hemen altında vmlinux adı ile üretilmiş olacaktır. Bu çekirdeği build dizinimize taşıyalım.

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

3.2 Berkeley Bootloader (bbl)

Bir makine açıldığında, işletim sistemi öncesinde çalışan programlara önyükleyici denir. Önyükleyicinin temel amacı, Linux çekirdeğini yüklemektir. Ön yükleyici olarak ilk önce U-boot kullanılması düşünülmüş fakat riscv-pk ile yapılmış daha fazla riscv proje kaynağı bulunulunca bu tercih edilmiştir.

riscv-pk projesinin indirip kurulma aşamaları aşağıdadır.

$ 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

.configure komutundaki --with-payload seçenği sayesinde bbl ve Linux çekirdeği bir araya getirilerek bbl imajı oluştururlur. Bbl imajına, bbl + Linux kernel gözü ile bakılabilir.

Oluşturulan bbl dosyası build dizinine kopyalanır.

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

3.3 Root File System

Çekirdeğin açılışından hemen sonra bağlandığı dosya sistemine kök dosya sistemi (root file system) denir. Kök dosya sistemi, bilgisayar sistemindeki tüm diğer dosya ve dizinlerin üzerinde bulunduğu temel hiyerarşik yapıyı oluşturur. İşletim sistemi ve kullanıcı tarafından erişilebilen tüm dosyalar, kök dosya sistemi içindeki alt dizinlerde yer alır.

Genellikle UNIX benzeri işletim sistemlerinde, kök dosya sistemi '/' (kök dizini) olarak adlandırılır ve diğer tüm dosya ve dizinler bu kök dizini altında hiyerarşik olarak düzenlenir. Örneğin, '/home' dizini kullanıcıların ev dizinlerini içerir, '/usr' dizini sistemdeki uygulama ve yardımcı programlarını içerir.

Bu uygulamada kök dosya sistemi olarak **busybox'tan** ve Nazım Koç hocamızın ucanlinux repository'sinden aldığımız iskelet kök dosya sisteminden yararlanacağız. BusyBox, temel UNIX komutlarından oluşan bir dizi uygulama sunar, örneğin ls, cp, mv, grep, find, tar, gzip, chmod gibi.

Öncelikle busybox'ı indirip kuralım. Burada yine busybox'ın en güncel ve stable sürümünü kurmaya dikkat ediniz. Bu döküman yazılırken en güncel sürüm 1.36 idi.

```
$ 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
```

Şimdi ucanlinux repository'sini indirelim.

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

İskelet dosya sistemini bozmamak için RootFS altında bir kopyası alınacaktır. Örnek kök dosya sistemimiz artık RootFS olacaktır. İskelet dosya sistemi, RootFS altına aşağıdaki gibi kopyalanır.

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

/bin, /sbin, /usr/bin ve /usr/sbin dizinleri aşağıdaki gibi busybox'tan RootFS altına kopyalanır.

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

Kernel'da bulunan modüller RootFS/lib/modules altına kurulmalıdır.

$ 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

Daha sonra toolchain içinde bulunan kütüphaneler /lib altına kopyalanmalıdır.

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

Şimdi linux kök dosya sistemimizdeki diğer boş dizinlerimizi kuralım.

$ 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

Ayrıca gem5'ın m5 library'si de build edilip dosya sistemine eklenmelidir.

$ 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

Artık kök dosya sistemi hazırdır.

3.4 Disk İmajı oluşturma

Bu bölümde kök dosya sistemimiz bir disk imajı içine mount edilecek ayrıca benchmark'ları da bu disk imajına mount edip gem5'da kullanılabilecek hale getilmek amaçlanmaktadır.

Öncelikle 500Mb'lık boş bir disk yaratılır.

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

Dosya sistemini kur

$ mkfs.ext2 -L riscv-rootfs riscv_disk

riscv_disk içinde artık bir dosya sistemi vardır. Bu dosya sistemine erişmek için aşağıdaki gibi mount yapılır ve RootFS içindeki bütün bilgiler riscv_disk içine kopyalanır.

$ 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

Disk imajı gem5 üzerinde kullanılmak için hazırdır. Fakat içerisinde herhangi bir benchmark binary'si şuanda yoktur. Dosya sistemimizin içine herhangi bir dosya, binary şu şekilde koyulur.

$ 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

! Bu adımda kullanmak istediğiniz benchmark binary'lerini /mnt/new_disk e kopyalayınız. Bu projede coreMarkPro binary'leri kullanılmıştır. Daha detaylı bilgi ve RISC-V coreMarkPro binary'lerini elde etmek için:

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

Artık içinde benchmarklarımız da bulunan riscv_coreMarkPro_disk isimli disk imajımız /out dizini içinde hazırdır.

4. Gem5 full sistem simülasyonu

Full sistem simülasyonunda kullanacağımız kernel ve disk imajı artık hazırdır. Bu bölümde bir riscv full sistem script'i nasıl yazılır ona değinilecektir.

Sistem Oluşturma
Scriptte, RiscvSystem adında bir sistem class'ı oluşturulur. Bu class, bir RISC-V sistemini modelleyen ve temel bileşenlerin (örneğin, cpu, cache, memory controller) nasıl oluşturulacağını ve birbirlerine nasıl bağlanacağını tanımlayan bir gem5 System class'ından miras alır. Platform olarak HiFive() seçilmiş PLIC, CLINT, UART, VirtIOMMIO sınıfları buradan alınmıştır.

CPU ve Cache Oluşturma
‘createCPU’ ve ‘createCacheHierarchy’ fonksiyonları, sistemdeki işlemcileri ve önbellek hiyerarşisini oluşturur. Her bir işlemci’ye ayrı bir L1 önbellek (hem data hem de instruction önbelleği) atanır. Bu önbellekler, L2 ve L3 önbelleklerle bir xbar (crossbar switch) aracılığıyla bağlanır, bu da çoklu bileşenlerin aynı anda veri aktarabilmesini sağlar. Ayrıca her bir core içerisinde ‘mmuCache’ olmalıdır. Bir MMU, işlemcinin hafızayı nasıl ve nerede kullanacağını denetler ve genellikle sanal hafızadan fiziksel hafızaya dönüştürme işlemlerini yönetir. MMU, sanal hafıza adreslerini fiziksel hafıza adreslerine dönüştürür, bu da bir işletim sisteminin birden fazla işlemi aynı anda çalıştırabilmesi için gereklidir. mmuCache, bu dönüşümleri hızlı ve verimli bir şekilde yapmayı sağlar.

Memory Controller ve İnterrupt’lar
‘createMemoryControllers’ fonksiyonu, sistemdeki bellek kontrolcülerini (memory controller) oluşturur. Bellek kontrolcüsü, CPU ve bellek arasındaki veri transferlerini yönetir. “setupInterrupts” fonksiyonu, işlemcilerin interrupt controller’larını oluşturur. Interrupt Controller, sistem interruptlarını yönetir ve işlemciye hangi işlemlerin öncelikli olduğunu belirler.

Devices ve I/O
‘initDevices’ fonksiyonu, sistemdeki I/O aygıtlarını oluşturur ve yapılandırır. Bu, platform düzeyinde bir kesme denetleyicisi (PLIC), çekirdek düzeyinde bir interrupt denetleyicisi (CLINT), gerçek zamanlı bir saat (RTC), Peripheral Component Interconnect (PCI), UART, PMA (Physical Memory Attribute) checker ve bir blok aygıtı (disk) içerir. Bu aygıtlar, membus ve iobus üzerinden bellekle ve diğer bileşenlerle iletişime geçer.

PLIC, bir işlemci üzerindeki tüm kaynaklardan gelen interruptları işlemek için kullanılır. Bir işlemcinin belirli bir zaman diliminde hangi interruptları işlemesi gerektiğini belirler ve bu işlemleri öncelik sırasına göre düzenler. PLIC, global interrupt kaynaklarını, interruptları target yani çekirdeğe bağlar. PLIC, "PLIC çekirdeği" ve "Interrupt gateways"den oluşur. Her bir interrupt kaynağı için bir adet interrupt gateways bulunur. Global interrupt, kaynaklarından birini interrupt gateways’e gönderir. Interrupt gateways, her bir kaynaktan gelen interrupt sinyalini işler ve tek bir interrupt isteğini PLIC çekirdeğine gönderir. PLIC çekirdeği, PLIC'deki bireysel interrupt kaynaklarını etkinleştirmek için interrupt enable (IE) bitlerini içerir. PLIC çekirdeği, işlenmeyi bekleyen bir interrupt olduğunu bildirmek için bekleyen interrupt bitlerine sahiptir. Ayrıca, PLIC çekirdeği kesme önceliklendirme/arbitrajını gerçekleştirir. Her interrupt kaynağına ayrı bir öncelik atanır. PLIC çekirdeği, kesme isteğini Interrupt Pending bitlerine (IP) latch’ler. Bekleyen interrupt önceliği hedef başına belirli bir eşiğe ulaştığında, PLIC çekirdeği bir interrupt bildirimiyle interrupt hedefine bir interrupt iletimi yapar. PLIC Claim/Complete register’ı, işlenmeyi bekleyen en yüksek öncelikli kesmeyi tutar.

CLINT, RISC-V mimarisindeki bir işlemci çekirdeği tarafından kullanılan yerel bir interrupt denetleyicisidir. Genellikle bir çekirdeğin iç saatine (timer) ve yazılım kesmelerine (software interrupts) bağlanır. Timer interrupts, belirli zaman aralıklarında otomatik olarak tetiklenir ve belirli bir işlemi gerçekleştirmek için işlemciyi duraklatır. Yazılım kesmeleri ise genellikle bir işlemcinin belirli bir yazılım hizmetini gerçekleştirmesi gerektiğinde yazılım tarafından tetiklenir.

RTC, bilgisayarda gerçek zamanı izlemek için kullanılan bir donanım bileşenidir. Sistem kapalıyken bile doğru zamanı izlemeye devam eder. RTC genellikle zamanla ilişkili işlevlerin gerçekleştirilmesi,

dosyaların zaman damgalarının ayarlanması veya bilgisayarın programlanmış bir zamanda uyanması gibi işlemler için kullanılır.

PCI, bir bilgisayarın ana kartı ve ek donanımlar arasında veri ve cihazlarla iletişim kurmak için kullanılan bir standarttır. Platformun PCI host'unun Programmed Input/Output (PIO) modu iobus'a bağlanmalıdır (self.platform.pci_host.pio = self.iobus.mem_side_ports). **PIO** ise, işlemcinin veri transferini kontrol ettiği bir veri transfer modudur. Bu modda, veri, işlemci ve sistem belleği arasında doğrudan ve aktif olarak hareket eder.

PMA checker, çeşitli fiziksel hafıza özelliklerini kontrol eder. İşlemcinin geçerli hafıza erişim kurallarına uygun olarak çalıştığından emin olur. Örneğin, bir işlemci, bir PMA'nın yalnızca okunabilir olarak işaretlediği bir hafıza bölgesine yazmayı denediyse, PMA Checker bu işlemi durdurur. Özetle PMA Checker'ın görevi, hafıza erişimlerini izlemek ve belirli hafıza bölgelerine yapılan erişimlerin, bu bölgeler için tanımlanmış özelliklere uygun olup olmadığını denetlemektir. Bu, hafıza korumasına yardımcı olur ve işlemcinin yanlışlıkla hafıza üzerinde izin verilmeyen bir işlem gerçekleştirmesini önler.

Device Tree Blob (DTB) Oluşturma
Bu fonksiyon, sisteme özgü bir Cihaz Ağacı Blob'unu (Device Tree Blob - DTB) oluşturur. DTB, Linux'te sistemin donanım bileşenlerini tanımlamak için kullanılır.

Kernel ve Disk İmage
Hazırlanan kernel ve disk image script'e terminal ekranında parametre olarak verilecektir. Scriptte bunun için gerekli işlemler yapılmış kernel'ı tanımlamak için

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

disk image'i tanımlamak için 'initDevices' fonksiyonu içerisinde

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

tanımlanmıştır.

5. Benchmark ortamı ve sistemlerin testi

Bu projede CoreMark-pro kullanılmıştır fakat istenilen benchmark binary'si test edilmek üzere kullanılabilir. 3.4'te bahsedildiği gibi istenilen benchmark binary'leri disk imajının içine atıldıktan sonra full-system'in içinde istenildiği gibi çalıştırılabilir. Bu projede 4 farklı case ve her case içerisinde 9 adet coreMark-pro binary'si çalıştırılmıştır. Binary'leri tek tek elle değil tek bir bash script ile çalıştırılmıştır. Bash script, host system tarafında yazılmış daha sonra aynı coreMark-pro binary'lerine yapıldığı gibi disk imajının içine kopyalanmış daha sonra guest system tarafında script çalıştırılmıştır. Burada dikkat edilmesi gereken bash script'in sonuna **poweroff** komutunun eklenmesidir. Bu yapılmazsa binary'lerin çalışması bittiğinde guest system boşuna çalışacak ve farklı case'ler için farklı sonuçlar doğabilecektir.

6. gem5 script’lerinin koşturulması

Her şey hazır olduğuna göre artık full-system simülasyonu başlatılabilir. Yazılan scriptler aşağıdaki komutla çalıştırılır.

$ 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

Burada **1L2_256kb_1L3_2mb** yazılan gem5 python script’idir. **–kernel, –disk ve –num_cpus,** bölüm 4’de de bahsedilen python argparse kütüphanesi ile verilen terminal opsiyonlarıdır.