CS152 Laboratory Exercise 2 (Version 1.2)

Professor: Krste Asanović
TAs: David Biancolin and Albert Magyar
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

February 22, 2019

Changelog

Version 1.2:

- Updated TLB miss penalty in Table 3
- Explains how to request custom AGFIs for Section 4.1.
- Minor grammatical fixes.

Version 1.1:

- Updated "Graded Items" (Section 1.1) to indicate you can share simulation results in the directed section. You must still do the analysis separately.
- Added a brief overview of `process_all.py`, in Section 3.4.3, to automatically generate result summaries from hpm_counter memory_stats.csv files.
- Added target-scripts to run all of spec and all of gapbs in `workloads/cs152-overlay`.

1 Introduction and goals

The goal of this laboratory assignment is to study processor memory hierarchy design by running realistic workloads on realistic RISC-V implementations. To do so, you'll be running a subset of the intspeed suite of SPEC2017, the latest revision of the industry standard benchmark suite for evaluating high-performance, general-purpose microprocessor implementations, and the Graph Algorithm Performance benchmark suite[1], a suite of portable, high-performance implementations of 6 important graph kernels. Instead of running these applications on a software microarchitecture simulator, you'll be using FireSim[2], which deploys FPGA-accelerated simulators on Amazon EC2 F1 instances. These simulators use detailed microprocessor models that have been transformed from silicon-ready RTL, generated by Rocket Chip SoC generator [3]. In addition, FireSim provides runtime-reconfigurable memory system timing models using another generator, FASED[4], which you'll use to simulate last-level caches and DRAM memory systems.

1.1 Graded Items

You will turn in a hard copy of your results to the instructor or TA, or a digital copy over bCourses. Please label each section of the results clearly. As before, each student will turn in a *complete* set of solutions for all of the closed ended problems. Since some of the experiments take a considerable amount of time to run, you may share simulation results with the members of your open-ended team, or with up to two other students, *but you must do the analysis independently.* In report, please indicate who ran the experiments associated with the results used for particular question.

As before, the open-ended section may be done in groups of one, two, or three; each group should submit a single report on *one* of the open-ended problems.

1. (Directed) Section 3.6: L1 Cache & TLB Parameter Sweep under SPEC CPU2017
2. (Directed) Section 3.7: Performance Modeling with Microarchitectural Events
3. (Directed) Section 3.8: Outer Memory System Study with GAP
4. (Open-ended) Section 4.1: Designing a Memory Hierarchy with a 5mm^2 Area Budget
5. (Open-ended) Section 4.2: Reverse Engineering of Memory Hierarchies
6. (Directed) Section 5: Feedback

Lab reports must be in *readable* English and not raw dumps of log-files. It is *highly* recommended that your lab report be typed. Charts, tables, and figures - when appropriate - are great ways to succinctly summarize your data.

2 Background

Since it's often confusing which machine we're talking about when speaking of computers simulating computers, throughout the lab we make a distinction between the *target*, the simulated machine, and the *host* the machine executing the simulation.

This section is lengthy as you'll be introduced to a number of new tools, target designs, target software, and benchmarks. Feel free to skip to the first section of the directed portion of the lab, which will walk you launching a FireSim simulator, running an application the target, and post-processing the data on the host.

2.1 Rocket Chip

Rocket Chip [3] is an open-source SoC generator suitable for both research and industrial purposes. Like the Sodor generators in Lab 1, Rocket Chip is a Chisel generator: a Scala program that, when executed, constructs a design based on some input parameterization to produce a single design *instance*. This is compiled down and emitted as Verilog. As such, Rocket Chip can generate a practically infinite space of instances, including many parameter sets that are impractical or suboptimal. In this lab, we will examine a few dozen points from this space, each with a different memory hierarchy, to explore the concepts described in class. All of the Rocket Chip instances you will use in this lab have three major components: processors, a cache hierarchy, and an outer memory system.

Rocket Chip derives its name from the processor microarchitecture it instantiates by default: Rocket, a 5-stage, in-order RISC-V processor (Figure 1). The instances you'll be using all implement

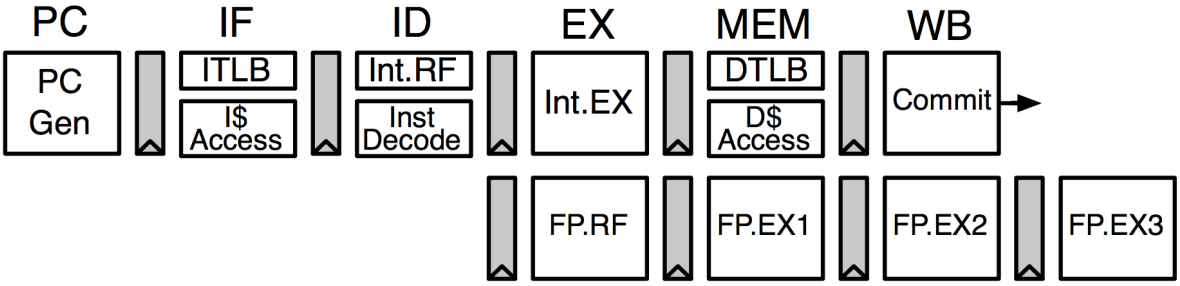

Figure 1: The Rocket Pipeline.

RV64IMAFDC, which refers to the 64-bit version of the RISC-V base ISA (RV64I), along with a set of useful extensions: M for integer multiply, A for atomic memory operations, F and D for single- and double-precision floating point, and C for 16-bit compressed representations of common instructions. Rocket Chip can generate multi-core instances, but in this lab you'll be using only uniprocessor instances.

A single-core Rocket Chip instance's cache hierarchy includes an L1 instruction cache, an L1 data cache, fully associative L1 I&D TLBs, and an optional, unified, direct-mapped L2 TLB. Each of these structures has a parameterizable size, associativity[1], and replacement policy, which is selected when Rocket Chip is generating the instance.

To talk to an outer memory system (generally DRAM), instances present an AXI4 master port. Since you will not be working with actual physical implementations of DRAM chips and controllers, a configurable, hardware memory model will be attached to this port, simulating the behaviors of a connected DRAM system and optionally a last-level cache (LLC).

2.2 FPGA-accelerated Simulation with FireSim

A Rocket Chip instance describes the large majority of the design of a complete SoC, one you could "tapeout" in silicon. Ideally, we'd use a software RTL simulator to do our microarchitecture performance studies, but these can only simulate a few thousand target machine cycles in each second of real wall clock time. Given that this makes the simulation take as long as if you ran the benchmark on a ˜1kHz computer, this is far too slow for long-running applications or booting an operating systems. In fact, this is about a million times slower than a silicon implementation of the real ˜1GHz SoC represented by that instance! Instead, we'll use an FPGA to simulate the instance by synthesizing the Verilog directly into the FPGA's LUT registers. Once you've paid the multi-hour FPGA compile time cost, these simulators can run at >100MHz – fast enough to boot Linux in seconds, and responsive enough to type into a simulated console in real time.

In this lab, we've done the painstaking work of precompiling all of your simulator FPGA configurations, or *bitstreams*, so you'll be able to switch between different instance simulators rapidly.

FireSim provides all the tools to build software for the simulated machine, generate Rocket Chip instances, compile simulator bitstreams, and batch out experiments to dozens of EC2 F1 instances. In this lab you'll be using a subset of the features to run a single simulation at a time, on a single EC2 F1 host.

[1] excluding the L2 TLB

2.3 FASED: Modeling Memory Systems In FireSim

Using FPGAs to do microprocessor simulation introduces a number of challenges. Given that the SoC would likely connect to DRAM through an off-chip interface, and given that the RTL of the design expresses only the actual SoC itself, there can be significant challenges in modeling the outer memory system accurately and deterministically. A real Rocket Chip-based SoC would instantiate a DRAM memory controller and drive real DRAM devices; given that these extra chips can't just be synthesized into FPGA fabric, we need to somehow reuse the DRAM subsystem attached to the FPGA, while simulating the DRAM subsystem we'd like to tape out. FireSim's memory timing model generator, FASED, does this by issuing memory requests to the FPGA's DRAM and applying latencies to the responses using a synthesizable hardware timing model. A FASED instance delays host memory responses that are too fast, and "pauses" the simulation if responses don't come back in time. In this way, a FASED instance can deterministically and faithfully model a DDR3 memory system using the FPGA's DDR4 memory system, or even simulate a single-cycle "magic" memory system, which is useful for doing asymptotic studies on memory system performance.

Since FASED instances are dedicated timing models not meant for silicon tapeouts, we've spent extra hardware to make them *runtime reconfigurable*. The timing parameters associated with a particular memory system configuration are loaded when you launch the simulator – this will allow you to change the simulated memory system without needing to recompile the FPGA's bitstream.

2.4 Target System Software

For this lab, you'll be running applications on a stripped-down Linux distribution; in particular, it uses a system configured by *buildroot*, an open-source tool that generates lightweight, embedded Linux environments. We'll provide a root filesystem image that includes everything needed to run the benchmarks for the the directed portion, as well as profiling tools you can reuse in the open-ended section.

2.5 The SPEC CPU2017 Benchmark Suite

The SPEC CPU2017 benchmark package (`https://www.spec.org/cpu2017/`) is the industry standard benchmark for evaluating general-purpose microprocessors. SPEC CPU2017 consists of four suites, but for this lab we will be studying a subset of the *intspeed* suite, which measures execution latency on integer code. You'll be running three benchmarks from the suite, with smaller *test* inputs [2] Brief descriptions of them, taken from `https://www.spec.org/cpu2017/Docs/benchmarks/`, follow. We given their approximate dynamic instruction counts in Table 1.

- `605.mcf_s` is a benchmark which is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C. The benchmark version uses almost exclusively integer arithmetic.
- `620.omnetpp_s` performs discrete event simulation of a large 10 gigabit Ethernet network. The simulation is based on the OMNeT++ discrete event simulation system ([1]www.omnetpp.org), a generic and open simulation framework.

[2]The reference inputs take a day to run on a typical rocket-based target in FireSim — per benchmark — so we batch out the suite to 11 FPGAs in parallel.

Benchmarks	Insns (Billions)
605.mcf	26
620.omnetpp	11
631.deepsjeng	50

Table 1: Dynamic instruction counts of Lab 2's SPEC201 intspeed sub-suite running its test inputs.

Benchmarks	Insns (Billions)
BFS	6.1
SSSP	38
CC	3.2

Table 2: Dynamic instruction counts of Lab 2's GAP sub-suite running with 2^{20} vertex Kronecker inputs.

- `631.deepsjeng_s` is based on Deep Sjeng WC2008, the 2008 World Computer Speed-Chess Champion. Deep Sjeng is a rewrite of the older Sjeng-Free program, focused on obtaining the highest possible playing strength.

2.6 Graph Algorithm Performance Benchmark Suite

The Graph Algorithm Performance Benchmark(GAPBS) Suite[1], developed by Scott Beamer, consists of a portable, high-performance implementations for 6 fundamental graph algoritms. As above, in this lab we'll be using smaller inputs (Kronecker graphs with 2^{20} vertices) as the reference inputs, real graphs of things like social networks, take too long to run. We provide three of the six:

- `Breadth-First Search (BFS)` - direction optimizing
- `Single-Source Shortest Paths (SSSP)` - delta stepping
- `Connected Components (CC)` - Afforest & Shiloach-Vishkin

Approximate dynamic instruction counts for each benchmark are given in Table 2.

3 Directed Portion (30%)

3.1 Connecting to your EC2 F1 Host

You'll be running on an EC2 F1 instance for the bulk of this lab. On the instructional machines, we provide scripts to manage your instance. SSH onto an instructional machine (`icluster{6-9}.eecs.berkeley.edu`). You'll need to install python's `boto3` package and source lab2's bashrc, like so:

```
inst$ pip install --user boto3
# Feel free to add this to your ~/.bash_profile
inst$ source ~cs152/sp19/cs152.lab2.bashrc
```

This will put four commands on your path:

- `f1-launch` spins up a fresh instance.
- `f1-stop` "pauses" your instance, all persistent state on EBS volumes (drives) is maintained, but you don't have to pay for th instance.
- `f1-start` restarts your previous stopped instance.
- `f1-terminate` blows away your instance and all of your state on attached drives.

You should only have to call `f1-launch` once at the start of the lab, after which you can just stop and start your instance at your leisure. If things get dire and you need a fresh machine (or if we need to deploy a fresh image), only then will you need to terminate and relaunch the instance.

Let's launch our F1 instance:

```
inst$ f1-launch
Launching an instance.
Waiting for instance initialization. This will take about a minute.
Instance ready.
    Instance ID: i-0f83b0bae99b6f43a
    Instance State: running
    Instance IP: 52.38.89.130
```

It will take a couple minutes to launch an instance, so you'll need to be a little patient the first time. *Note: your instances's IP will change every time you start and stop the instance, but the provided scripts will always give you the the up-to-date IP address.*

To SSH into your instance use the user `centos`, and the private key `~cs152/sp19/lab2/firesim.pem`. In lab2's bashrc, we've aliased `ssh` and `scp` to include this key.

```
inst$ ssh centos@<YOUR-INSTANCE-IP>
```

3.2 A Tour of FireSim's Configuration Files

Now you are on your F1 instance, move into the root of the firesim repository at `/firesim`, and set up your environment:

```
inst$ ssh centos@<YOUR-INSTANCE-IP>
centos$ cd ~/firesim
centos$ export FIRESIM_ROOT=$(pwd)
centos$ source sourceme-f1-manager.sh
```

We'll refer to the root of firesim as `$FIRESIM_ROOT` from now on. `sourceme-f1-manager.sh` will put a RISC-V toolchain and firesim's eponymous manager program, `firesim`. `cd` into `deploy/`. Here you'll find the base configuration files used by `firesim`. You'll be working in this directory and `$FIRESIM_ROOT/deploy/workloads` for much of the lab. Open `config_runtime.ini`.

```
centos$ cd deploy
centos$ ls
centos$ $EDITOR config_runtime.ini
```

This ini describes a simulation.

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=0
m4_16xlarges=0
f1_4xlarges=0
f1_2xlarges=1

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=no_net_config
no_net_num_nodes=1
linklatency=6405
switchinglatency=10
netbandwidth=200
# This specifies the instrumentation sampling rate (in target-clock cycles)
# on the FASED timing model. -1 = Disabled.
profileinterval=-1

# This selects a bitstream and runtime configuration file pair
defaulthwconfig=CS152BaseTConfig-LBPBaseConfig

[tracing]
enable=no
startcycle=0
endcycle=-1

[workload]
# This will let you spin up different root filesystems on your target
workloadname=linux-uniform.json
terminateoncompletion=no
```

Now open up `config_hwdb.ini`. Here you'll find a database of all the different target designs we've preconfigured for you. (The first entry is the name of the configuration, you'll use this string to specify your `hwconfig` in `config_runtime.ini`. Hardware database entries looks as follows:

```
# Rocket-Chip        FASED Memory Model
# Generator Config.  Generator Config.
# "TARGET_CONFIG"    "PLATFORM_CONFIG"
#    |               |
#    V               V
[CS152BaseTConfig-LLCDRAMBaseConfig] #  Name
agfi=agfi-0611c5eed29acfda8          #  Bitstream
deploytripletoverride=None           #  Ignore this
customruntimeconfig=None

[single-cycle]
agfi=agfi-01f762aab939519ce
deploytripletoverride=None
# Overrides default latency of latency-bandwidth pipe
# Custom runtime configurations must be in $FIRESIM_ROOT/sim/custom-runtime-configs
customruntimeconfig=LBPBaseConfig/single-cycle.conf
```

The name of a HWDB entry has two parts: a `TARGET_CONFIG`, which selects the SoC instance Rocket-Chip generates, and a `PLATFORM_CONFIG`, which selects memory-timing model instance FASED generates. For a given memory model instance, a space of different *runtime configurations* can be used. They are written to memory mapped registers in the simulator before simulation commences. To prevent you from mismatching a runtime-configuration with an incompatible instance we've put them in sub-directories according to their `PLATFORM_CONFIG`. `cd` to `$FIRESIM_ROOT/sim/custom-runtime-configs` and check out some of the provided examples:

```
    centos$ cd $FIRESIM_ROOT/sim/custom-runtime-configs/LLCDRAMBaseConfig
    centos$ $EDITOR 8W-64B-256K-LLC-FCFS-16G-4R-2133-OP.conf
+mm_relaxFunctionalModel=0
+mm_openPagePolicy=1
+mm_backendLatency=2
+mm_schedulerWindowSize=8
+mm_transactionQueueDepth=8
# DDR3 Memory Timings
+mm_dramTimings_tAL=0
+mm_dramTimings_tCAS=14
+mm_dramTimings_tCMD=1
...
+mm_dramTimings_tWTR=8
# DRAM organization & address assignment scheme
+mm_rankAddr_offset=16
+mm_rankAddr_mask=3
...
+mm_bankAddr_offset=13
+mm_bankAddr_mask=7
# Last-level cache size and organization
```

```
+mm_llc_wayBits=3
+mm_llc_setBits=9
+mm_llc_blockBits=6
```

The default hwdb in `config_runtime.ini` will simulate a single Rocket core, with 16KiB, 4-way, L1 I and D caches, with fully-associative L1 I/DTLBs with 32 entries, and a direct-mapped L2 TLB with 256 entries. The memory model can simulate a LLC with up to 4MiB in capacity (when configured with 8 ways and 128B lines), and is backed by a 16 GiB, quadruple-rank, DDR3 14-14-14 memory system.

3.3 Linux Boot and Hello World

To boot the default target we need to run two commands. If you haven't already, you should start a `tmux` or `screen` session.

```
centos$ firesim infrasetup # Programs FPGA, creates run dir @ ~/sim_slot_0
centos$ firesim runworkload &> /dev/null &  # Launches the simulation
```

Always call infrasetup before launching a simulator, as important simulator state is initialized during flashing. To attach to the simulator, use screen to attach to a screen instance named *.fsim0.

```
    centos$ screen -R
There are several suitable screens on:
    6587.fsim0      (Detached)  #  You want this one
    6308.virtual_jtag       (Detached)
    6126.hw_server  (Detached)
    Type "screen [-d] -r [pid.]tty.host" to resume one of them.
    centos$ screen -r 6587.fsim0

[    0.020000] disk [generic-blkdev] of loaded; 3800800 sectors
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 253:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 148K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK
```

If you've done this quick enough, you'll catch the target in the middle of Linux boot. You'll be presented with a prompt. Login with the username: `root` and password: `firesim`.

```
Welcome to Buildroot
buildroot login:
```

Congratulations! You've logged on a FireSim cycle-accurate simulation of a Rocket-Chip SoC. It may be running a minimalist buildroot distribution, but you should be able to interact with the target much like you would any other Linux box. For example:

```
    rocket$ uname -a
Linux buildroot 4.15.0-rc6-31587-gcae6324ee357 #1 SMP ...
    rocket$ cat /proc/cpuinfo
hart    : 0
isa     : rv64imafdc
mmu     : sv39
uarch   : sifive,rocket0
```

3.4 Instrumentation & Running a Workload

To collect core-side performance statistics from the simulator, we're going to run a user program, `hpm_counters`, that periodically polls the core's hardware performance monitor (HPM) counters every 0.1 target-seconds[3]. It should already be on your path (installed at `/usr/bin/hpm_counters` in the target's filesystem). To profile the execution of a program, you can invoke hpm_counters as a background process, run the desired program, and then terminate hpm_counters. For example:

```
rocket$ hpm_counters > stats.out &
rocket$ echo "Hello World"
rocket$ pkill hpm_counters
rocket$ cat stats.out
```

Produces a result like this:

```
##  Cycles = 24642332997
##  Instructions Retired = 3419229816
##  Time = 7700729
##  Loads = 679399395
##  Stores = 298182970
##  I$ miss = 6194625
##  D$ miss = 238974014
##  D$ release = 26570146
##  ITLB miss = 788193
##  DTLB miss = 11754357
##  L2 TLB miss = 11513147
##  Branches = 337933419
##  Branches Misprediction = 13819477
##  Load-use Interlock = 2028478246
##  I$ Blocked = 308655575
```

Finally, we've provided two shell scripts on your path, `start_counters` and `stop_counters`, which will dump the statistics file to the `/hpm_data`.

[3]Our targets run at "1GHz", so this corresponds to 100 million target-cycles

3.4.1 Running SPEC & GAPBS Benchmarks

We've put precompiled SPEC binaries at `/spec17-intspeed`, and provided a launch script, `run.sh`, that will spin up a benchmark with the right inputs. Passing `--counters`, will use hpm_counters to profile execution. Try running omnetpp (it will take a couple minutes, so feel free to interrupt it).

```
rocket$ cd /spec17-intspeed
rocket$ ./intspeed 620.omnetpp_s --counters
```

Standard out and error from the run can be found at `/output`. We can do the same with GAP, which is installed at `gapbs`. To run `bfs`:

```
rocket$ cd /gapbs
rocket$ ./gapbs.sh bfs-kron --counters
```

3.4.2 Powering Down A Simulator & Pulling Out Results

When you're done simulating, and ready to parse results, call `poweroff`. *Warning: If you want your results, don't try to rush it with `poweroff -f`, as firesim will need to re-mount the target's filesystem after it powers down.*

When a simulation terminates, FireSim collects all your desired outputs and puts them to `$FIRESIM_ROOT/deploy/results-workload/<TIME>-<workload-name>`. By default, after the target has terminated the Linux-uniform workload copies everything under `/output` and `/hpm_data` output of the target's filesystem[4]. There you'll also log of the UART (`uartlog`) and automatically collected memory-system statistics(`memory_stats.csv`). *NOTE: the values in memory stats.csv are 32 bit values, so they may roll over!*

3.4.3 Adding Custom Files & Automation

To carry you through the open ended section, you'll need to automate some of the drudgery of spinning up and tearing down simulations, and running workloads. This might include adding your own scripts and applications to the target's filesystem.

For this we provide two skeleton workloads, `cs152-automated.json` and `cs152-interactive.json` that will let you add arbitary files to the targets file system. The automated workload will also will run your desired script as part of init, before powering off the machine.

`cd` to `$FIRESIM_ROOT/deploy/workloads/` and look at `cs152-automated.json`.

```
centos$ cd $FIRESIM_ROOT/deploy/workloads
centos$ $EDITOR cs152-automated.json
```

```
{
  "common_bootbinary" : "bbl-vmlinux",
  "benchmark_name" : "cs152-automated",
  # The root of where you want to put your custom files
  "deliver_dir" : "/cs152/",
```

[4]You can always remount the filesystem yourself, you can find it at `~/sim_slot_0`

```
  # Files you want added to the target's filesystem, before simulation
  "common_files" : ["example-script.sh"],
  "common_args" : [""],
  # Add files you want removed from the target' filesystem here
  "common_outputs" : ["/hpm_data", "/output", "/cs152/hello.out"],
  "common_simulation_outputs" : ["uartlog", "memory_stats.csv"],
  "workloads" : [
    {
      "name": "cs152-automated-all",
      "files": [],
  # A shell command you want to run before poweroff
      "command": "cd /cs152 && ./example-script.sh > hello.out",
      "simulation_outputs": [],
      "outputs": []
    }
  ]
}
```

And then build it:

```
    centos$ make cs152-automated
mkdir -p cs152-automated
cp ../../sw/firesim-software/images/br-disk-bin cs152-automated/bbl-vmlinux
python gen-benchmark-rootfs.py -w cs152-automated.json -r \
        -b  ../../sw/firesim-software/images/br-disk.img \
        -s cs152-overlay

        Generating a Rootfs image for cs152-automated-all
        Copying base rootfs ../../sw/firesim-software/images/br-disk.img
            to cs152-automated/cs152-automated-all.ext2
        Copying src: cs152-overlay/example-script.sh to
            //cs152//example-script.sh in target filesystem.
        Creating init script with command:
            cd /cs152 && ./example-script.sh > hello.out
            Copying src: build/temp to /etc/init.d/S99run in target filesystem.
```

Now to run your new image on a simulator, update `config_runtime.ini` (or create a new one), and run the simulator as you did before:

```
    centos$ vim $FIRESIM_ROOT/deploy/config_runtime.ini

[workload]
workloadname=linux-uniform.json # -> change to "cs152-automated.json"

    centos$ firesim infrasetup
    centos$ firesim runworkload &> /dev/null &
```

If you attach you screen, you'll notice the target boots into init, runs the example script, and powers off before you're given the prompt. As before, you can find the results in an appropriately named directory in `$FIRESIM_ROOT/deploy/results-workload`.

In additional to the example script, we've also provided two additional target-scripts: `run-spec.sh` and `run-gapbs.sh`, in `$FIRESIM_ROOT/deploy/workloads/cs152-overlay`. These can be substituted for the example script to run all of the benchmarks of a single simulator.

Finally, we've provided a host-script, `process_all.py`, which will run by default on automated workloads, and will parse the memory-system and hpm-counter stats files and emit event summaries summaries that will resid in that workload's `results-workload` directory.

```
    # In the results-workload dir
    centos$ cd cs152-automated-all/post_processed
    centos$ ls
605.mcf_s0.csv
605.mcf_s0.summary
memory_stats_processed-605.mcf_s0.csv
memory_stats_processed-605.mcf_s0.summary

    centos$ cat 605.mcf_s0.summary
Total Cycles        : 133263388708
Total Instructions  : 25671816938
CPI                 :  5.191
D$ MPKI             : 89.549
I$ MPKI             :  0.173
D$ Miss %           : 22.999
ITLB MPKI           :  0.004
DTLB MPKI           : 32.716
L2 TLB MPKI         : 18.202
Branch Prediction % : 85.317

   centos$ cat memory_stats-605.mcf_s0.summary
Number of Reads Serviced     :  2.30E+09
Number of Writes Serviced    :  1.87E+08
AXI4 R Bus Utilization %     :  110.659
AXI4 W Bus Utilization %     :   9.002
LLC Summary:
    LLC Hit Rate (%)         :  40.589
    LLC MPKC                 :  11.108
    Miss : Writeback Ratio   :  13.522
Row Buffer Hit %             :  64.571
RANK 0 Summary :
    Row Buffer Hit %         :  65.423
    Cycles Idle %            :  21.814
RANK 1 Summary :
    Row Buffer Hit %         :  61.549
    Cycles Idle %            :  18.849
```

```
RANK 2 Summary :
    Row Buffer Hit %         :  65.163
    Cycles Idle %            :  21.656
RANK 3 Summary :
    Row Buffer Hit %         :  66.736
    Cycles Idle %            :  24.517
```

You can retroactively invoke `process_all.py` on a results-workload directory like so:

```
centos$ cd $FIRESIM_ROOT/deploy/workloads/cs152-automated/
centos$ ./process_all.py -b <path-to-your-results-workloads-dir>
```

3.5 Stopping Your Instance & Credits

Once you're done with your work on the EC2 f1 instance, logout and stop the instance.

```
centos$ exit
inst$ f1-stop
```

This will perserve the persistent state of your instance (any thing you wrote to a filesystem) so you'll be able to restart the instance and still have your environment setup and all of your results. Amazon does not charge for a stopped instance (only for the data you've have on persistent storage (EBS), which can be neglected).

For this lab, we have credit enough for 100 hours of f1.2xlarge up-time *per student*. We'll be tracking per-student expenditure, so once you spend above your limit your instance will be suspended, and you won't be able to start it again until the next lab.

With that out of the way, it's time to get to the graded content of the lab.

3.6 L1 Cache & TLB Parameter Sweep for SPEC CPU2017

In this section, you will collect memory system and core-side statistics for the SPEC2017 intspeed sub-suite. You'll run all three benchmarks on a set of different memory systems that sweep one independent parameter of: L1 cache size, L1 cache associativity, L1 TLB size, or lL2 TLB size.

First, assume the L1 cache access time is 1 cycle, the L2 cache access time is 25 cycles, 42 cycles for a DRAM row buffer hit, and 70 cycles for a DRAM row-buffer miss, for calculation of AMATs. Give an expression for calculating AMAT.

Now sweep across an appropriate space of rocket configurations, using the `BaseLLCDRAMConfig` memory model instance and it's default runtime-configuration. Use stats collected from the memory timing model, and from hpm_counters, to answer the following questions:

1. How does the L1 cache size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for the available L1 cache configurations by using a fixed associativity of 8, and fixed TLB capacities.

2. How does the L1 cache associativity affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for the provided L1 cache configurations assuming a fixed L1 I & D cache capacity of 8 KiB, and fixed L1 and L2 TLB capacities.

Microarchitectural events	Miss penalty (cycles)
L1 cache miss	25 (L2 cache access latency)
L2 cache miss	$2/3 \times 42 + 1/3 \times 70$ (DRAM access latency)
L1 TLB miss	2
L2 TLB miss	$3 \times (2/3 \times 1 + 1/3 \times 70)$
Branch condition mispredict	3
Target address mispredict	3

Table 3: Idealized Miss Penalties for Microarchitectural Events

3. How does the L1 TLB size size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for the provided L1 TLB configurations, fixing the L2 TLB capacity.

4. How does L2 TLB size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for the provided L2 TLB configurations (fixed L1 D$, I$ and L1 TLB sizes).

3.7 Performance Modeling with Microarchitectural Events

We can approximate CPI with the following equation:

$$CPI = CPI_{base} + \sum_{e \in \{events\}} MPI_e \times PENALTY_e$$

where MPI_e is misses per instruction for event e and $PENALTY_e$ is the miss penalty for event e. Table 3 shows the miss penalties for microarchitectural events.

Note that Rocket Chip supports three-level page tables (Section 4.3 in [5]). Thus, when there is an L2 TLB miss, the hardware page table walker (PTW) accesses a cache for page table entries (PTEs) first (let's assume its hit rate is 2/3) and the L2 cache and main memory if there is a miss in the cache. The PTW repeats it three times to access a leaf table to obtain the physical page number.

The Rocket core also predicts branch conditions as well as target addresses for control flow instructions. When these predictions are wrong, PC is redirected from the memory stage, and thus their penalties are 3 cycles in most cases.

Assuming $CPI_{base} = 1.3$ (why not 1?), predict CPIs for various cache parameters across benchmarks using the data from Section 3.6 and compare them against the actual CPIs from the simulations. How close are the predicted CPIs to the actual CPIs? What assumptions do you think contribute the most to the descrepancy? Explain, using collected results where possible.

3.8 Outer Memory System Study with GAPBS

In this question we'll be sweeping LLC and DRAM parameters while keeping the `TARGET_CONFIG` fixed, to study how the outer memory system affects performance of the provided GAPBS sub-suite. Start by collecting two baselines, one with a DRAM memory system but with no LLC, and one using a single-cycle memory system.

1. How does the L2 cache size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for the available L2 runtime-configurations assuming a 64-byte block

size, and 8 way set associativity. Constrast the results against the cacheless DRAM model, and the single-cycle outer memory system. How does DRAM row-buffer hit rate change as the cache capacity changes? When, if ever, would you consider using a closed page policy?

2. Repeat the experiment above, now with 128B cache lines. Report miss rates, MPKIs, AMATs (in cycles), and CPIs, and constrast the results to the 64B.

3. How does the L2 cache associativity affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for L2 associativities of 2, 4, and 8 ways using a fixed cache capacity of 512KiB, and 64B cache blocks. How does DRAM row-buffer hit rate change as the cache associativity changes?

4 Open-ended Portion (70%)

Pick *one* of the following questions. The open-ended portion is worth a large fraction of the grade of the lab, and the grade depends on how comprehensive the process to your conclusion is.

4.1 Designing a Custom Memory Hierarchy with a 5mm^2 Area Budget

This question strikes at the heart of job of a computer architect. We want to figure out how we can make the best of a 5mm^2 cache area budget for caches for our target application (you'll pick either the GAP or SPEC sub-suites we provided). You'll need to consider the effects of both cycle time and CPI on performance using the Iron Law. Using FireSim, you'll be able to measure CPI and explore different cache organizations. To explore area and cycle time, you will use CACTI [6], a tool that models physical cache implementations.

4.1.1 Calculating Cycle Time

For the purpose of calculating cycle time, we will assume the critical paths of both the core and outer memory system go caches and/or TLBs (this is not atypical). Therefore, the access times for the L1 caches and L1 TLB determine the cycle time of the core domain where they reside, along with the CPU. For this problem, we'll also assume that our L2 cache, L2 TLB, and DRAM subsystems reside in a separate clock domain, with the constraint that this domain runs at exactly half the frequency. We will use CACTI to estimate the access times and the areas for given cache configurations.

You can use CACTI on your computer, but we recommend using the `icluster` machines. First, clone the CACTI repository and compile CACTI:

```
inst$ git clone https://github.com/albert-magyar/cacti.git
inst$ cd cacti
inst$ make
```

In `cs152-sp19-configs`, there are `L1Icache.cfg`, `L1Dcache.cfg`, `L2cache.cfg`, `L1TLB.cfg`, `L2TLB.cfg`, which are configuration files representing the L1 instruction cache, L1 data cache, L2 cache, L1 TLB, and L2 TLB, respectively.

Parameters	Values
Associatively	1, 2, 4, 8
Number of Sets	Up to 4096 (power of 2)
Block Size	Up to 128 bytes (power of 2)

Table 4: Available LLC cache parameters

To evaluate the access time and area of a cache with the parameters specified by a given configuration file, run the following command from the top-level `cacti` directory:

```
inst$ make run CFG=cs152-sp19-configs/<configuration file>
```

The result will be saved in `cs152-sp19-outputs/`. Note that results are appended to the output file when you repeat it for the same configuration file.

Next, sweep the parameters in these files to represent the space of different cache configurations under consideration. Each time you edit a file, re-run `make run ...` to obtain the new results. You should start by picking configurations you can simulate on the provided AGFIs (Table 4 shows the space of available configurations for the L2 cache). If you'd like to evaluate a design that hasn't been provided, you may add it to the Google sheet provided on Piazza (@141). *You can only request L1 cache configurations that avoid aliasing.*

If you want to flush all output files, run:

```
inst$ make clean
```

4.1.2 Submission

Now that you can calculate cycle time, go ahead and explore the design space! Pick a design point that you feel represents a good tradeoff of cycle time and CPI, and make sure you size your L2 cache and TLB to take advantage of their decreased clock speed – more cycle time means you can fit a bigger cache while still closing timing!

Submit a report that explains how you arrived at your proposed design. You don't have enough time to do a brute force exploration of the design space, so explain what you tried, and what you didn't, and why. Leverage data you've collected where possible. For example, try plotting CPI, cycle time, or execution time over a design parameter to motivate why you didn't take that parameter further. Finally, explain how particular workloads in the benchmark influenced your design. If you could remove one, which would it be, and why?

Finally, please attach the configuration files from Section 4.1.1.

4.2 Validation and Reverse Engineering of Memory System Organizations

In this question, we'll try to infer parameters of a computer's memory system by running user code and measuring execution latency. This is useful for a number reasons:

- To help guide application optimizations when the microarchitecture host system is unknown or secret.

- To do performance validation of a memory system organization before tape out. Some of the most insidious bugs in computer system design are performance bugs, since applications still execute correctly only more slowly. We'd like to catch these bugs before committing a design to silicon, but without a performance model of the machine-under-test they may go undiscovered.
- Using the same principle as above, to help validate simulation models (read: FASED). FASED, like many simulation models splits its timing and functional model. This scheme makes it possible to build very large cache models with actually modeling data – but without the data it's very easy to write "correct" but fundamentally broken timing models.[5]

4.2.1 Getting Started and Easy Questions

To validate our memory hierarchy, we will use the `caches` benchmark in `ccbench` [7] developed by Christopher Celio[6] In `$FIRESIM_ROOT/deploy/workloads` we've defined an automated workload that run ccbench for you, and visualize the result. The make recipe will also build CCbench from source. To run CCbench, do as follows:

```
centos$ cd $FIRESIM_ROOT/deploy/workloads
centos$ make ccbench-cache-sweep
centos$ firesim infrasetup -c workloads/ccbench-cache-sweep.ini
centos$ firesim runworkload -c workloads/ccbench-cache-sweep.ini {&> /dev/null &}
```

This will take about four minutes. `ccbench` also provides a script to visualize the output file. The provided CCbench workload will run this script on the uartlog after powerdown to generate a plot of loop latency vs array size.

```
    centos$ cd ../results-workload/<TIMESTAMP>-ccbench-cache-sweep/ && ls
ccbench-all  outputplot.pdf # <- open pdf locally
    centos$ $EDITOR ccbench-all/uartlog
...
App:[caches],NumThreads:[0],AppSize:[1024],Time:[4.01151],
App:[caches],NumThreads:[0],AppSize:[2048],Time:[4.01097],
App:[caches],NumThreads:[0],AppSize:[4096],Time:[4.02701],
#           Size of array (4B words) ^           ^ Cycles per iteration
...
```

Open up the plot and answer the following questions:

- What is the L1 cache size?
- What is the L1 cache latency?
- What is the L2 cache size?
- What is the L2 cache latency?

[5] For example, in one bug I fixed, a write address was mistakenly being used for a read access to the tag-array of LLC model. In a real cache, this would return incorrect data but in the model it just manifested as small timing aberration.

[6] Chris is also the author of BOOM, the RISC-V Out-of-Order machine you'll study in lab3.

To proceed further, you'll need to understand the source code of the `caches` benchmark in ccbench, and the arguments the program accepts. Once your do, run another simulation with a modified init script to determine, L1 and L2 block sizes. Provide your plot (you may need to modify the plotting script).

4.2.2 Harder Questions

Chris did most of the work you in the last part. Now you'll modify Chris's code, or write your own to try to determine some more subtle parameters. We're interested in the following:

- L1 D Cache Replacement Policy
- L1 I Cache Size
- L1 I Cache Associativity
- L1 TLB Reach
- L2 TLB Reach
- L2 TLB Hit Latency
- DRAM Page Policy (Open or Closed)
- Aggregate (Ranks X Banks) DRAM Page Size
- Number of DRAM Ranks (you can assume there are 8 banks)
- DRAM CAS/RCD/RP latencies (you can safely assume they are the same)

Feel free to glean whatever information you can from the provided AGFIs and runtime configurations to write the most effective code possible. Even read the chisel code if necessary. When you're ready, try running your code on our three "Mystery Microarchitectures", provided at the bottom of config_hwdb.ini. These have hardwired memory-model timings and so don't use a runtime configuration. And don't read too much into Chisel compilation that occurs during infrasetup – we're using a dummy configuration to build the simulation driver.

4.2.3 Submission

For the three mystery microarchitectures, report the cache dimensions and latencies as indicated by the provided version of the CCbench cache benchmark. (Include the plots.) Then for each mystery microarchitecture, provide your best estimate for at least 5 (half) of the parameters – they don't have to be the same ones for each uarch[7]. For each of those parameters explain, referring to your code as necessary, how you measured it. If you think you cannot accurately guess 5 parameters, still provide your code and explain what you tried. A higher grade will be awarded to a measured negative result, over an ill-justified guess (that may be correct). If you have data or plots to show your code works on the known instances, provide it in your justification.

[7] Hint: some parameters will be more inferable on particular configurations

5 Feedback

To make FireSim a more effective teaching resource for future CS152 students, we'd really appreciate your feedback. Please fill out the survey form at `https://goo.gl/forms/dMdX64t7hECeBQ0z2`.

How many hours did the directed portion take you? How many hours did you spend on the open-ended portion? What did you learn? What would you change? Feel free to write as little or as much as you want.

6 Acknowledgments

We'd thank Amazon for their credit donation for this lab report. We'd also like to thank Donggyu Kim, who wrote the first iteration of an EC2 F1 based lab2 for CS152 in Spring 2018. This lab was inspired by the previous set of CS 152 labs written by Henry Cook, Yunsup Lee and Andrew Waterman, which targeted functional simulators such as Simics and Spike.

References

[1] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[2] S. Karandikar *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, (Piscataway, NJ, USA), pp. 29–42, IEEE Press, 2018.

[3] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, apr 2016.

[4] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanović, "Fased: Fpga-accelerated simulation and evaluation of dram," in *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*, FPGA '19, (New York, NY, USA), ACM, 2019.

[5] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10," May 2017.

[6] S. Wilton and N. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, may 1996.

[7] C. Celio, "The ccbench micro-benchmark collection (https://github.com/ucb-bar/ccbench/wiki)," 2010.