Introduction and goals

The goal of this laboratory assignment is to conduct memory hierarchy experiments by running realistic workloads on RocketChip [1]. To enable RTL-based simulation using FPGAs, we have already generated performance simulators by automatically transforming RocketChip [2, 3]. We will run these performance simulators in Amazon EC2 F1 instances (https://aws.amazon.com/ec2/instance-types/f1) to collect RocketChip’s memory system stats for a subset of the SEPC2006int benchmark suite (https://www.spec.org/cpu2006/). You will also make architectural design decisions based on the results.

The lab has two sections, a directed portion and an open-ended portion. Everyone will perform the directed portion the same way, and grades will be assigned based on correctness. The open-ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task or the arguments you provide in support of your ideas.

For both the directed portion and the open-ended portion of this lab, you must work in a group of two (not three). You are also encouraged to discuss solutions to the lab assignments with other groups, but must run through the lab by yourselves and turn in a hard copy of your own lab report in the beginning of the lecture on March 5th.

You are only required to do one of the open-ended assignments. These assignments are generally starting points or suggestions. Alternatively, you can propose and complete your own open-ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the instructor or the TAs.

1.1 Graded Items

You will turn in a hard copy of your results to the instructor or TA. Please label each section of the results clearly. The directed items need to be turned in for evaluation. You only need to turn in one of the problems found in the open-ended portion.

1. (Directed) Section 3.5: Analysis on Cache Statistics
2. (Directed) Section 3.6: Performance Modeling with Microarchitectural Events
3. (Open-ended) Coming soon
4. (Directed) Feedback on this lab
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

### 2.1 Rocket Chip

RocketChip [1] is an open-source SoC generator suitable for research and industrial purposes. Rather than being a single instance of an SoC design, RocketChip is a hardware design generator, capable of producing many design instances from a single piece of Chisel [4] source code. Multiple industry products as well as silicon prototypes are manufactured using RocketChip. A RocketChip instance generally consists of three major components: processors, a cache hierarchy, and an uncore.

Rocket Chip instantiates an in-order processor, Rocket, by default, but also supports various core implementations. Rocket is a 5-stage in-order processor (Figure 1) that implements the RISC-V ISA [5, 6]. Its cache hierarchy includes L1 instruction caches, L1 blocking data caches, and fully associative L1 TLBs and a direct-mapped L2 TLB with configurable sizes, associativities, and replacement policies. In this lab, we provide pre-built FPGA images for various configurations. For now, RocketChip does not provide an L2 cache implementation yet, so we adopt an abstract L2 cache model instead [3]. This cache model is runtime configurable, so we do not need different FPGA images for different L2 cache parameters.

### 2.2 The SPEC CPU2006 Benchmark Suite

The SPEC CPU2006 benchmark suite (https://www.spec.org/cpu2006/) was widely used to evaluate real systems as well as design ideas in computer architecture research. This benchmark suite includes real-world application which execute *trillions of instructions* with their reference inputs. In this lab, we will only use a subset of benchmarks with their test inputs as follows:

- **400.perlbench**: Cut-down version of Perl v5.8.7, the popular scripting language.
- **401.bzip2**: Modified Julian Seward’s bzip2 version 1.0.3, where all compression and decompression happens entirely in memory.
• **403.gcc**: C language compiler gcc version 3.2, which generates code for an AMD Opteron processor.

• **429.mcf**: Combinatorial optimization for single-depot vehicle scheduling in public mass transportation.

> SPEC CPU2006 will retire soon. Why not SPEC CPU2017? (Un)fortunately, SPEC CPU2017 is more realistic, requiring longer execution times. We just wanted to save time and money. However, we selected benchmarks that also exist in SPEC CPU2017.

### 2.3 FPGA-based Performance Simulation with Amazon EC2 F1 Instances

We evaluated various pipelines for very small benchmarks using software-based performance simulators for Lab1. However, software-based simulation does not provide sufficient performance to evaluate complex hardware designs for realistic software applications.

Instead, any RTL designs can be directly mapped and emulated in the FPGA at speed. However, we need more accurate and runtime configurable timing models for memory systems and devices that will be implemented as an abstract RTL model or a software model. For this reason, RocketChip RTL implementations are automatically transformed and instrumented to generate performance simulators running in the FPGA [2, 3], enabling efficient communications between FPGA and software.

There are increasing interests in using FPGAs for application-specific accelerators. As a result, cloud service providers decided to offer FPGA cloud instances such as Amazon EC2 F1 instances. We use these services to quickly evaluate real-world hardware designs for real-world software applications.

We pre-built FPGA-based performance simulators for various cache parameters and provide them as Amazon FPGA images (AFIs) that can be loaded into any F1 instances. We also provide the Amazon machine image (AMI) that contains necessary software binaries and scripts to run simulations.

### 3 Directed Portion (30%)

#### 3.1 Launching an F1 Instance

We will launch an F1 instance to start this lab. First, log in to an instructional machine (icluster{6-9}.eecs.berkeley.edu). Then, to enable commands to control your F1 instances, run:

```bash
# You can add this line in ~/.bash_profile
inst$ source ~/cs152/sp18/cs152.lab2.bashrc
```

Next, let’s launch an F1 instance:

```bash
inst$ launch-f1 | tee <file name>
wait until running
running now!
Instance ID: i-07ab2ee7a526ccacb
```
IP Address : 34.227.49.244
Keep Instance ID & IP Address
Please shut the instance down in 8 hours.
wait for initialization
initializing
... 
initializing
ok
It’s time to SSH into your instance

It will take for a while (~3 min) for initialization, so please be patient. Also, note that you will need the instance id and the IP address across this lab. Most importantly, each student is allowed to launch single instance at a time as many times as possible, but the total instance-hours should not exceed 40 hours. (Therefore, 80 hours are allowed for each group in total.) If you violate any rule, you will get a severely penalty. (50 % with the first violation and 100 % with the second violation. Therefore, you don’t have to do this lab if you violated the rules twice.)

3.2 Linux Boot and Hello World

Now, let’s SSH into the F1 instance:

   inst$ ssh centos@<IP Address>

Once you SSH into your F1 instance, move into cs152-lab2:

   $ cd cs152-lab2

This directory contains necessary files to conduct this lab. First of all, we should load an AFI image:

   # 16KiB L1$, L1 TLB reach = 128KiB, L2 TLB reach = 4MiB
   $ ./load-fpga.sh agfi-0aa6f0423f0ff7843

This will load a RocketChip simulator for 16KiB L1 I/D caches, fully-associative L1 I/DTLBs with 32 entries, direct-mapped L2 TLB with 1024 entries.

We provide you a make file command to conveniently run RISC-V binary images:

   # Default argument values
   # L1_SIZE ?= 16KB (design parameter)
   # L1_TLB_REACH ?= 128KB (design parameter)
   # L2_TLB_REACH ?= 4MB (design parameter)
   # L2_WAY_BITS ?= 2 (2^2 = 4 ways, runtime parameter)
   # L2_SET_BITS ?= 12 (2^12 = 4096 sets, runtime parameter)
   # L2_BLOCK_BITS ?= 6 (2^6 = 64 Bytes, runtime parameter)
   $ make BIN=<RISCV binary image>

This command creates a directory named output/<cache parameters>, runs the image in the simulation, pipes stdout and stderr to output/<cache parameters>/<RISCV binary image>.{out, err}, and dumps the cache (and branch) statistics to output/<cache parameters>/<RISCV binary image>.
image>.stat. (Read Makefile for more information.) This will be useful when you run simulations for various cache configurations.

Now, let’s boot Linux and print hello world in the simulator. Run:

$ make BIN=images/bblvmlinux-hello

You can see linux boot messages and Hello CS152! in the screen. It also prints performance counter values, which are also saved in outputs/16KiB-128KiB-4MiB-2-12-6/bblvmlinux-hello.stat:

```plaintext
## cycles = 19024725
## instret = 8995954
## loads = 674618
## stores = 430790
## L1 I$ misses = 95952
## L1 D$ misses = 68632
## L2$ misses = 20942
## ITLB misses = 1674
## DTLB misses = 2450
## L2 TLB misses = 2765
## branches = 390234
## branch mispredicts = 87680 // Mispredicts for branch directions (taken / not taken?)
## target mispredicts = 71764 // Mispredicts for control flow target addresses
```

Can you compute the CPI, the miss rates and the misses per kilo instructions (MPKIs) of miss events for bblvmlinux-hello?

**Warning:** make claen will delete all generated output files. So, ask yourself carefully if you really want to delete all the files before you run into a disaster.

### 3.3 Cache Parameter Sweep for SPEC CPU2006

You will collect cache and branch stats for SPEC CPU2006 by running images/bblvmlinux-\{400.perlbench, 403.gcc, 429.mcf\}. Note that these benchmarks run on top of Linux where TLBs play an important role for system performance.

To automate simulation runs for various cache configurations, we will run a script, sweep.py. This script loads AFI images for each cache configuration and runs simulations for each benchmark sequentially. Table 1 shows AFIs for different cache configurations. Also, for agfi-09d2f1c836f0ad278 (L1 $ size = 32 KiB, L1 TLB reach = 128 KiB, L2 TLB reach = 4 MiB) in Table 1, we apply L2 cache parameters in Table 2. Otherwise, we only have 1MiB 8-way L2 cache with 128 byte lines (the second parameter in Table 2). Please take a look and figure out how this script runs simulations using the makefile command.

Once you understand what is going to happen, it is time to run the script to sweep cache parameters:

```
# To prevent simulations gone while you are out
$ tmux
# Also, good to measure how long it took
$ time ./sweep.py
```
Table 1: AFIs for various cache configurations

<table>
<thead>
<tr>
<th>Size</th>
<th>Associativity</th>
<th>Sets</th>
<th>Block Size (Bytes)</th>
<th>L1 TLB reach</th>
<th>L2 TLB reach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1MiB</td>
<td>8</td>
<td>512</td>
<td>256</td>
<td>128 KiB</td>
<td>4MiB</td>
</tr>
<tr>
<td>1MiB</td>
<td>8</td>
<td>1024</td>
<td>128</td>
<td>2MiB</td>
<td></td>
</tr>
<tr>
<td>1MiB</td>
<td>8</td>
<td>2048</td>
<td>64</td>
<td>64 KiB</td>
<td></td>
</tr>
<tr>
<td>1MiB</td>
<td>8</td>
<td>4096</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1MiB</td>
<td>2</td>
<td>4096</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1MiB</td>
<td>4</td>
<td>2048</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512KiB</td>
<td>8</td>
<td>512</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2MiB</td>
<td>8</td>
<td>2048</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4MiB</td>
<td>8</td>
<td>4096</td>
<td>128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: L2 cache configurations simulated with agfi-09d2f1c836f0ad278 in Table 1

It will run for a long time (~7 hours). You can hang out while the machine is working, but make sure you come back on time. (Otherwise, you will blow credits out and have Donggyu go bankrupt!) If you want to return to the screen in the middle, SSH into F1 instance again and run:

$ tmux a

### 3.4 Terminating Your Instance

Once, you finish the experiments. Copy the output files to the instructional machine as follows:

inst$ scp -r centos@<IP Address>:~/cs152-lab2/outputs ./

You will have the whole output files in the current directory. Next, shut down your instance with the following command in an instructional machine:

inst$ terminate-f1 <Instance Id>

Once again, you should terminate an instance on time to avoid a penalty.

### 3.5 Analysis on Cache Statistics

Now, it’s time to analyze the output files of long simulations. We provide you a script to convert all output files into CSV files for each benchmark. Run:
cd outputs

cp ~cs152/sp18/lab2/analyze.py ./

python analyze.py

Using the CSV files, answer the following questions. Assume L1 cache penalty is 23 cycles and L2 cache penalty is 100 cycles. You may want to modify the script for each question.

1. How does the L1 cache size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L1 cache sizes by fixing other parameters.

2. How does the L2 cache size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L2 cache cache sizes by fixing other parameters.

3. How does the L2 cache associativity affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L2 cache associativities by fixing other parameters.

4. How does the L2 cache block size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L2 cache block sizes by fixing other parameters.

5. L1 TLBs cannot be as large as you want because it is fully-associative. How important their reaches are if there is no L2 TLB?

6. Do you think a direct-mapped L2 TLB is crucial for system performance? Present the evidence.

### 3.6 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 \(e\).

(You can easily see why we computed MPKIs in Section 3.5). Table 3 shows the miss penalties for microarchitectural events.

Note that RocketChip supports three-level page table (Section 4.3 in [6]). 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 PTE 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 the target address for control flow instructions. When these predictions are wrong, PC is redirected from the memory stage, and thus the miss penalty is 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 3.3 and compare them against the actual CPIs from the simulations. How close are the predicted CPIs to the actual CPIs? What information do you want to know for more accurate performance predictions?
<table>
<thead>
<tr>
<th>Microarchitectural events</th>
<th>Miss penalty (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache miss</td>
<td>23 (L2 cache access latency)</td>
</tr>
<tr>
<td>L2 cache miss</td>
<td>100 (DRAM access latency)</td>
</tr>
<tr>
<td>L1 TLB miss</td>
<td>2</td>
</tr>
<tr>
<td>L2 TLB miss</td>
<td>$3 \times (2/3 \times 1 + 1/3 \times 123)$</td>
</tr>
<tr>
<td>Branch condition mispredict</td>
<td>3</td>
</tr>
<tr>
<td>Target address mispredict</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Miss Penalties for microarchitectural events

4 Open-ended Portion (70%)

Stay tuned. We will let you know through Piazza when it’s ready.

References


