ECE 452: Computer Organization and Design
Spring 2022
Homework 6: Architectural Exploration with Sniper

Assigned: 12 Apr 2022
Due: 19 Apr 2022

Instructions:

- Please submit your assignment solutions (no code needs to be attached) via Canvas in a word or pdf file. • Some questions might not have a clearly correct or wrong answer. In such cases, grading is based on your arguments and reasoning for arriving at a solution.
- Directly copying commands from this file onto the terminal may result in some unexpected characters in the pasted command. It is always a good idea to type the command out if copying does not produce the expected behavior.

Introduction

Sniper is a next generation parallel, high-speed and accurate x86 simulator. This multi-core simulator is based on the interval core model and the Graphite simulation infrastructure, allowing for fast and accurate simulation and for trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures. The Sniper simulator allows one to perform timing simulations for both multi-program workloads (multiple applications running on different cores) and multi-threaded, shared-memory applications (one multi-threaded application running on multiple cores) with 10 to 100+ cores, at a high speed when compared to existing simulators.

This assignment will provide you with hands on experience working on Sniper to perform architectural exploration for processor/memory subsystems. The following pages will first go through the process of helping you to set up the environment, before presenting you with problems to solve.

Q1 [100 points]. In this problem you will explore the cache hierarchy in an x86 processor using Sniper. Consider an x86 multiprocessor subsystem of 4 in-order cores running at 2 GHz, with L1 data cache size of 4 KB and associativity of 1, L1 instruction cache size of 4 KB and associativity of 1, and unified L2 cache size of 32 KB with associativity of 4. (See *small.cfg* in attachment.) The following command simulates a 4-core system (-n 4) running the *volrend* benchmark with small input size (-i small) using architectural parameters specified in *small.cfg* (-c small):

```
> cd ~/workspace/sniper_assignment/sniper-7.2/benchmarks
>./run-sniper -p splash2-radix -i small -n 4 -c small
```

After simulation ends, you should find the performance summary in *sim.out* under the

current folder. `> gedit sim.out &`

(a) Suppose you are given a budget of 512KB to distribute between L1 instruction and data caches, and a maximum total L2 cache size budget of 4MB. This budget must be shared among the 4 cores. Assume that the minimum L1
instruction and data cache size is 4KB each. The L1 budget should be equally shared among the 4 cores, but the sizes of the L1 instruction and L1 data caches do not have to be equal in a core. Similarly, the L2 budget should be equally shared among the 4 cores. Note that cache sizes are always a power of 2 and obviously each core has a private L1 instruction and data cache. Assume also that L2 caches are private to each core (the 'shared_cores = 1' parameter in the *small.cfg* file under [perf_model/l2_cache] section indicates that fact; thus the default size of 16KB for each L2 cache essentially results in a total L2 footprint of 4 * 16KB in our 4 core system). Assume the minimum L2 cache size to be 16KB per core. You are also allowed to increase maximum L1 associativity to 4 and L2 associativity to 8. Assume that doubling the size increases both data access time and tags access time by 2 cycle for any cache (i.e., if the data_access_time = 4 and tags_access_time = 1 cycle for a cache_size = 4KB, then increasing cache size to 8KB increases data access time to 6 cycles and tag access time to 3 cycles, and so on. This change should be made in the .cfg file). Modify *small.cfg* and simulate to find the cache configuration for *radix* that gives the best performance results (i.e., smallest execution time in the *sim.out* file) while respecting the constraints on associativity and size for caches. Present the cache size values and associativity values of your best configuration and improvement in execution time over the base case (i.e., default values in *small.cfg*).

The configuration that I found gave the shortest execution time was Through using a
64KB L1 I-cache associativity 2, 64Kb L1 D-cache associativity 4, 1MB L2 cache associativity 8.
With this configuration I was able to achieve an execution time of 58.25, however, I want to note that when I reran this test in order to confirm I had gotten correct results since this was the lowest execution time by a 4 second period I was unable to produce the same results. The next lowest execution time I was able to achieve was 64.79 sec, indicating that this result may have been just a good case where cache misses were not happening as often. Considering this i believe that the true best configuration is
64KB L1 I-cache associativity 4, 64Kb L1 D-cache associativity 4, 1MB L2 cache associativity 8.
This configuration had a minimum execution time of 62.97. More importantly when I reran the simulations once more this configuration was consistently executing around the same execution time of 63-64 seconds. Showing that it would consistently do better than the previously mentioned configuration.

(b) Repeat the analysis for the *lu.cont* benchmark with small input size, using the command below. Are the best configurations for both benchmarks the same?

```
> ./run-sniper -p splash2-lu.cont -i small -n 4 -c small
```

Interestingly enough i found that the same configuration
64KB L1 I-cache associativity 4, 64Kb L1 D-cache associativity 4, 1MB L2 cache associativity 8.
Had the best execution time of 394.59, this was the lowest time that it was able to reach but when i ran this test multiple times to make sure that I had gotten correct results but when this was done I found that the execution time would fall between 404-410. On one instance of running this test I was able to achieve a total execution time of 383.12 But if we consider the absolute lowest execution time to be the best configuration then it would be
64KB L1 I-cache associativity 4, 64Kb L1 D-cache associativity 4, 1MB L2 cache associativity 4.
With this configuration I was able to reach an execution time of 378.90. But as with the previous tests we ran i was unable to achieve this level of performance when i reran the simulation again. On the following runs the execution time would land between 410-421 showing that this may have just been another case in which we got lucky and there were far fewer cache misses.

Q2 [150 points]. In this problem you will compare simulation results between in-order processor (*small.cfg*) and out of-order processor (*big.cfg*), capture power data, and explore how varying parallelism impacts applications.

The following command simulates a system using out-of-order cores with McPAT power

model enabled. `> ./run-sniper -p splash2-fft -i small -n 4 -c big`

`--power`

After simulation, you should find total power summary displayed in the terminal and performance summary saved in *sim.out* under your current folder.

(a) For the *fft* benchmark, create a table to compare simulation results (average power, energy, IPC, and execution time), between the in-order configuration and the out-of-order configuration. Note that for parallel execution, execution time for the program is the maximum out of the execution times on each of the cores. Use the same sized L1 and L2 caches (and latencies) for the in-order configuration (*small.cfg*) as those found in the out-of-order configuration file (*big.cfg*). What conclusions can you draw from these results?

	in-order	out-of-order
Average power	3.23	3.73
energy	0.10	0.11
IPC	0.52	2.28
EXEC TIME	189.50	201.68

From running this test a couple of times for both the in-order and out-of-order configuration I found that the in-order configuration was faster than the out-of-order configuration in all the times that I ran both configurations back to back. Though there are a few things that I think are important to understand in the difference between the two configurations. We can see that they both consume about the same amount of energy, however when we look into the average power consumption from the cache itself we can see that the in-order configuration achieved a lower average power consumption. We are also able to see that even though the two finish in a relatively close time from one another we see that having a higher IPC does not mean that the processor will run faster. Sometimes there are more important things than just optimizing the IPC as in this case it lead us to have a higher average power consumption and a higher execution time.

(b) For the *fft* benchmark running with the in-order configuration, with each core having 4KB L1 I-cache, 4KB L1 D cache, and 32KB L2-cache, plot the execution time for the following misprediction penalities: 4,8,12, and 14 along with the type of branch predictor as one-bit and pentium-m. Populate the tables shown below with your results (in seconds) and explain the trend that you observe.

One-bit branch predictor

Mispredict penalty	Execution Time
4	195.77
8	195.89
12	195.29
14	200.36

Pentium_m branch predictor

Mispredict penalty	Execution Time
4	199.57
8	200.09
12	198.05
14	198.59

The first thing that I noticed from running these tests was that the one bit predictor was actually faster than the pentium_m brand predictor in almost all of the cases. The second thing that I noticed was that when I ran the test for the one bit predictor the execution time was steady until I reached the penalty of 14. I reran this test in order to make sure that I didn't just get a bad run, however, that wasn't the case and the lowest execution time for a penalty of 14 was 200.36. The next thing that I noticed was that regardless of the mispredict penalty the execution time of the pentium remained around in the range of 198-200. Once again I reran the tests to ensure that the results were accurate and reproducible but the initial tests were correct as the execution time kept falling in that range of 198-200.

The conclusions that I can draw from this is that In the case of the pentium_m branch predictor the mispredict penalty does not truly affect the execution time by very much and therefore if we had a large mispredict penalty we would want to use the pentium_m predictor. In the case that we have a low penalty it is beneficial to be using the one bit branch predictor.

(c) For the *fft* benchmark running with the in-order configuration, with each core having 4KB L1 I-cache, 4KB L1 D cache, and 32KB L2-cache plot the execution time as the number of cores is increased from 4, 8, 16, 32, and 64 (i.e., increasing TLP). Explain the trend that you observe.

# of Cores	Execution Time
4	198.11
8	155.97
16	120.87
32	136.58
64	224.20

We can see that amdahl's law does not truly hold in this scenario. If it were going to hold we would find that the execution time would keep decreasing until it eventually plateaus in this case it could have been an 100s. Instead the trend that we see is that once there are more cores than 16 the execution time actually increases. From this we can see that adding more cores and increasing TLP doesn't always translate to better performance. We can see that once we add enough cores that it actually does worse than only having 4 cores to the processor. Out of curiosity I also ran a test with 128 cores to see how poorly this would react to having too many cores. This interestingly only took 305.35 seconds to complete running showing that adding to many cores is bad and there is a sweetspot