CS425: Computer Systems Architecture

Programming Assignment 2

Assignment: December 14, 2011

Due: January 9, 2012 – 23:59:59

Instructions: Solve the programming assignment and deliver the sources and a short report via e-mail to Vassilis Papaefstathiou (papaef@csd.uoc.gr). Use the subject: **HY425 – Programming Assignment 2**

Start Early!

Cache Simulation

The purpose of this assignment is to familiarize you with the details of caches. You have to implement caches yourself and measure their quantitative properties. You will also see the impact of alternative design choices in the miss rate and IPC. For the simulation of caches you will use our custom simulator that is based on the PIN dynamic binary instrumentation tool (www.pintool.org).

Simulator

You can get the simulator from the home directory of the course:

/home/lessons1/hy425/HW/2011f/PA/PA2_CacheSimulation

The directory contains: (i) the instrumentation tool (CacheSimulation.cpp), (ii) a generic cache model (CacheModel.H), (iii) a cache controller that performs the appropriate steps to implement the cache functionality (CacheController.H), (iv) a configuration class that keeps the current settings of the cache(s) (CacheConfiguraton.H), (v) a profiler class that keeps statistics and implements an extremely simple timing model (CacheProfiler.H) and (vi) a directory with 7 benchmarks (4 integer and 3 floating point) to exercise the caches and measure their performance.

Copy the simulator in your home directory and study carefully the sources to familiarize with the simulator. To compile the simulator just type: `make`. To run a benchmark with the simulator (e.g. fft) use the following command:

```
make SIM_ARGS="-l1a 4" SIM_APP="./benchmarks/6.fft/fft" run
```

SIM_ARGS sets the command line arguments that you can pass to the simulator (check the CacheSimulation.cpp KNOBs to see the available switches) and SIM_APP sets the application/benchmark that will run on simulator.

The *timing model* of the simulator assumes an in-order processor that executes every instruction in 1 clock cycle, it has a perfect instruction cache, a perfect branch predictor and a main memory with infinite bandwidth, but with a latency of 100 clock cycles. All the above assumptions allow us remove the implications of other components and focus on the data caches alone. An L1 hit costs 1 clock cycle while an L1 miss costs 101 clock cycles (fetch data from main memory). *Note:* simulating each of the given benchmarks takes about 5 minutes to complete (they contain a few billion instructions) and you will have to collect several data points so start early!

Measure L1 Miss Rate and IPC

Your first task is to explore the effects of cache-block size and associativity in an L1 data cache. Assume that your L1 data cache budget is 32 KB. Run experiments with varying block sizes and set-associativity for all the given benchmarks, draw miss-rate and IPC graphs similar to those presented in class and find the configuration that gives the highest average IPC. Explore the following block sizes: (i) 32 bytes, (ii) 64 bytes and (iii) 128 bytes and the following associativities: (i) direct-mapped, (ii) 2-way and (iii) 4-way. *Hint:* prepare scripts and use the command line switches, you can run the experiments overnight!

Implement and Measure an L2 Cache

Implement in the simulator a *strictly-inclusive* write-back L2 cache, i.e. every cache-block that is present in L1 should always be present in L2 and if by chance you need to evict a cache-block from L2 then you should also evict it from L1 if present. The provided CacheModel is generic enough to be used for L2, so you need only to modify the CacheController, however you are free to change the code at will. Add command line switches (knobs) to parameterize the L2, implement the appropriate functions for configuration and profiling of the L2 and modify the timing model to take into account the L2 cache.

Run experiments with varying L2 cache sizes and associativity for all the given benchmarks draw *global* miss-rate and IPC graphs similar to those presented in class and find the L2 configuration that gives the highest average IPC. Explore the following L2 cache sizes: (i) 128KB, (ii) 256KB and (iii) 512KB and the following associativities: (i) 4-way, (ii) 8-way and (iii) 16-way. For your measurements assume that the L1 configuration is the one you found before, the cache-block size of the L2 cache is the same with L1 and that the L2 access time is 10 clock cycles.

Implement and Measure next-*k*-line Prefetchers

Use the best performing L1 and L2 cache configurations that you found before, and implement a next-*k*-line prefetcher only for the L2 cache. Next-line prefetchers are extremely simple hardware schemes that attempt to exploit spatial locality beyond cache-block boundaries. Typically, when a cache experiences a cache miss for line A, then the next-*k*-line prefetcher fetches the next *k* sequential cache-blocks, if not already present in the cache; *k* is known as the prefetch degree.

You have to implement the following two prefetching policies: (i) *always-prefetch* and (ii) *prefetch-on-miss*. The *always-prefetch* scheme tries to fetch the next-*k* blocks even on hits, while the *prefetch-on-miss* prefetches only on misses. Run experiments with varying prefetch degrees for all the given benchmarks, draw *global* miss-rate and IPC graphs similar to those presented in class and find the prefetcher configuration that gives the highest average IPC. Explore the following prefetch degrees: (i) 1, (ii) 2 and (iii) 4. Assume that prefetches complete instantly.

An important implication of cache prefetching is the need for higher memory bandwidth. Based on the memory traffic facility of the simulator, find the prefetching configuration that best balances between IPC and memory traffic. *Hint:* calculate the metric *Memory Traffic / IPC.*