Georgia Institute of Technology

School of Computer Science

CS 4290/6290, ECE 4100/6100: Fall 2014 (Prof Conte)

Project 1: Cache design

Version 1.0

Due: Friday, October 3rd, 2014 at **11:55pm**

Rules

This is the first project for the course --- here are some rules:

1. Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy.

2. It is acceptable for you to compare your results, and only your results, with other students to help debug your program. It is not acceptable to collaborate either on the code development or on the final experiments.

3. You should do all your work in the C or C++ programming language, and should be written according to the C99 or C++98 standards, using only the standard libraries.

4. Unfortunately experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered. *It is your responsibility to check the website often and download new versions of this project description as they become available*.

5. A Makefile with the frontend will be given to you; you will only need to fill in the empty functions and any additional subroutines you will be using. You will also need to fill in the statistics structure that will be used to output the results.

Project Description:

Cache memories are hard to understand! One way to understand them is to actually build a cache. We do not have time to do this in this class. However, writing a simulator for a cache can also make caches easier to understand. So in this project, you will design a parametric cache simulator and use it to design data caches well suited to the SPEC benchmarks.

Specification of simulator:

Cache simulation core capabilities:

a. The simulator should model a caching system with 2^C bytes of data storage, having 2^B-byte, and with sets of 2^S blocks per set (note that S=0 is a direct-mapped cache, and $S = C - B$ is a fully associative cache).

b. The memory addresses are 64-bit addresses.

c. The cache is byte-addressable.

d. The cache implements the write-back, write-allocate (WBWA) policy. There is an additional *dirty bit* for each tag in the tag store.

e. The cache implements least-recently-used (LRU) as the replacement policy. The LRU cache chooses the least recently used block for replacement.

f. There is 1 valid bit per block of storage overhead required. The valid bits are set to 0 when the simulation begins.

g. The following are used to calculate AAT, hit time, and miss penalty. (HT means Hit Time, MR means Miss Rate, and MP means Miss Penalty):

  - AAT = HT + MR * MP
  - HT = 2 + 0.2*S
  - MP = 200

h. The cache has a victim cache (described below) that can hold *V* blocks. Vary *V* in the range [0, 4].

i. There is a prefetcher (described below) that will prefetch *K* blocks on a miss. Vary *K* in the range [0, 4].

j. In general, (*C, B, S, V, K*) completely specifies the caching system.

Cache simulation optimizations (see Experiments Section):

a) The cache performs a strided prefetch algorithm to reduce compulsory misses. The prefetcher works as the following when a miss occurs:

  - Define the function Block_Addr(X) = address of X with offsets bits of X to zero.

- On a miss to address X, calculate d = Block_Addr(X) - Last_Miss_Block_Addr. Then set Last_Miss_Block_Addr = Block_Addr(X).

- If d == Pending_Stride (see below), then prefetch the next K blocks. This means that we bring blocks with block addresses X + *i* * Pending_Stride, for i = 1..(*K*) from memory to the cache. *K* is given as a simulation parameter.

- Note that prefetch does not contribute to AAT. Any prefetch are *not* included in the calculation of the miss rate. (However, prefetched blocks can pollute the cache by causing the eviction of useful blocks.)

- A prefetched block becomes an LRU (least recently used) block in its set. (Hint: you can make the timestamp of the prefetched block equal to min(timestamps of all other blocks in the set) – 1.)

- Finally, regardless of whether d matches Pending_Stride or not, set Pending_Stride = d.

- Vary *K* in the range [0, 4].

b) There is a Victim Cache

- The VC is fully associative. It can hold *V* blocks, and the replacement policy is FIFO.

- When there is a cache miss, first check the VC. If the block is present, replace the LRU block in the set (the "victim") with the VC block. If the block is not present, fetch it from memory. Place the victim block in the VC. Evict the oldest block from the VC (the VC is FIFO replaced).

- When a block is evicted from the cache, it is written to the VC. Evicted blocks from the VC are written back, if dirty, to memory.

- Vary *V* in the range [0, 4]

Explanation of Provided Framework

We are providing you with a framework to build the cache simulator. You must fill in the following functions in the framework:

```
void setup_cache(uint64_t c, uint64_t s, uint64_t b, uint64_t v,
uint32_t k);
```

Subroutine for initializing the cache. You many add and initialize any global or heap variables as needed.

```
void cache_access(char type, uint64_t arg, cache_stats_t* p_stats);
```

Subroutine that simulates the cache one trace event at a time. Type can be either READ or WRITE, which is each defined in cachesim.hpp. A READ event is a memory load operation of 1 byte to the address specified in arg. A WRITE event is a memory store operation of 1 byte to the address specified in arg.

```
void complete_cache(cache_stats_t *p_stats);
```

Subroutine for cleaning up memory and calculating overall system statistics such as miss rate or average access time.

***Statistics (output)*: The simulator must output the following statistics after completion of the run:**

a. number of accesses to the cache

b. number of reads

c. number of read misses for the cache

d. number of read misses that also miss in the VC

e. number of writes

f. number of write misses for the cache

g. number of write misses that also miss in the VC

h. total number of misses

i. number of write backs

j. total number of misses in the VC

k. number of prefetched blocks

l. number of useful prefetches = cache accesses that hit on a prefetched block. Only count this for the first-time hit to any particular prefetched block.

m. total number of *bytes* transferred to (writeback) or from (miss repair or prefetch) memory

n. the average access time (AAT) of the overall cache system

The output of these variables should be handled by the driver code, and you only need to fill in the structure cache_statistics_t, defined in cachesim.hpp.

Experiments:

First validate your simulator (see the section on *validation requirement* below).

THE EXPERIMENTS ARE AS IMPORTANT AS HAVING A WORKING SIMULATOR!

For each trace in the trace directory, design a cache subject to the following goals:

1. You have a total budget of 48KB respectively for the cache system's storage (including all tag storage, valid bits, cache data storage, VC storage, VC tag storage, but exclude bits used for LRU and storage used for the prefetch registers Last_Miss_Block_Addr and Pending_Stride)

2. The cache should have the lowest possible AAT

You may vary any parameter (*C, B, S, V, K*).

Validation Requirement

Four sample simulation outputs will be provided on T-Square by the TAs. You must run your simulator and debug it until it matches 100% all the statistics in the validation outputs posted on the website. You are required to hand in this validated output with your project (see grading).

Comparison of Optimization Techniques

You need to decide whether your system should use option A (prefetcher) or option B (victim cache). Suppose you only have the budget to implement one!

You should also consider the area and energy overheads of each of these optimization techniques. We will provide a framework for estimating energy consumption. Stay tuned. Explain why you chose one technique or another, what assumptions you made in making your decision, and what the insights are about how each optimization works.

What to hand in via T-Square:

1. Output from your simulation showing it matches 100% all the statistics in the validation outputs posted on the website for each validation run

2. A document with the design results of the experiments for each trace file, with a *persuasive* argument of the choices that were made. (An argument may be as simple as an explanation of the search procedure used to find the designs and a statement about why the procedure is complete.) This argument should include output from runs of your program. (*There are multiple answers for each trace file, so I will know which students have "collaborated" inappropriately!*)

3. The commented source code for the simulator program itself.

4. Remember that your code must compile and run on a current variant of Linux (i.e., Debian, Red Hat, Ubuntu) running on an x86 architecture (i.e., Intel or AMD).

5. **Note that late projects will not be accepted.**

Grading:

0% You do not hand in anything by the deadline
+50% Your simulator doesn't run, does not work, but you hand in significant commented code
+20% Your simulator matches the validation outputs (5% each)
+20% You ran all experiments and found a cache for each trace
+10% The project hand in is award quality. For example, you justified each cache with graphs or tables and a persuasive argument

Hints

There is no data store (or values) modeled in your simulator, just a tag store.

Implementation Details

This section describes the design choices that I made when implementing the project. There are many edge cases, so please make sure your implementation matches this exactly.

Dirty Bit Logic

- Suppose there is a write miss. The new block is first brought to the cache, and then is written to. So the newly inserted block should be marked as dirty. You may find evidence that this is not a "realistic" implementation, but it is the simple assumption that I made for the project.
- Any write access to a block that is not dirty causes it to become dirty. This is regardless of whether the block was in the main cache or victim cache.
- A write back happens when a dirty block is evicted from the VC to main memory (or from L1 if there is no VC).
- Make sure to copy the dirty status when moving blocks between L1 and VC!

Victim Cache Replacement Logic

- After an L1 cache miss, probe the victim cache. If there is a hit in the VC, replace the VC block that you hit on with the LRU block in the L1 cache. Insert the block from VC as the Most Recently Used block in the L1.
- If there is a miss in both L1 and VC, then evict the LRU from the set in L1 and insert it in the VC using FIFO replacement policy.

Prefetching Details

- Initial prefetch values (last miss and pending stride) should be zero.
- Order of operations:
    - First probe the L1.
    - If there is a miss then probe the VC.
    - Next, perform the L1 insertion (with a block from VC if there is a hit, or from main memory if a miss).
    - After the insertion, check the prefetcher if there was an L1 miss. Even if there was a VC hit, still check the prefetch! But be sure to do it after you insert the missing block in L1!
- Prefetch Replacement Policy:
    - When inserting a prefetched block in to L1, it must be in the least-recent-used position (it must have the lowest time stamp).
    - If a prefetched block already exists in the L1, do not touch the LRU position.
    - If a prefetched block is not in the L1, but it is in the VC, then swap the L1 LRU with the block in VC as you would normally. This way, the same block should not be both in L1 and VC. You should still insert the block as LRU in L1.
- Other Details:
    - A prefetch request should not generate additional prefetch requests. Only check the prefetcher for a "regular" access (an access coming from the trace).
    - A prefetch request never affects hit or miss statistics directly.
    - A "useful prefetch" is only counted once. A useful prefetch happens when you hit on a prefetched block either in L1 or in VC.

Detailed Statistic Description

- Accesses: How many times is my cache accessed by a load instruction in the trace? Do not count prefetches.
- Reads: How many instructions from the trace are reads?
- Read Misses: How many read instructions miss in L1?
- Read Misses Combined: How many read instructions miss in both L1 and VC?
- Writes: How many write instructions are there in the trace?
- Write Misses: How many write instructions miss in L1?
- Write Misses Combined: How many write instructions miss in both L1 and VC?
- Misses: Total Misses in L1
- Writebacks: Number of blocks transferred out of all caches and back to main memory.
- Victim Cache Misses: Total misses in the victim cache.
- Prefetched blocks: How many blocks are brought in by the prefetcher? Remember that the prefetcher is dumb and does not check whether a block exists in the cache before bringing it from memory.
- Useful Prefetches: Of all the prefetched blocks, how many were actually used?
    - Clarification: It doesn't matter if the prefetched block was already in the L1 or VC. Keep it simple. If you prefetch a block, and then use it before it is written back to memory, count 1 useful prefetch.

- Bytes Transferred: How many bytes moved between main memory and the cache? Count all prefetches, all misses which require bringing a block from memory, and all writebacks.
- Hit Time: Calculated as shown above. Clearly, it should be of float or double type.
- Miss Penalty: Calculated as shown above.
- Miss Rate: Misses/Access in the L1
- AAT: Calculate as shown above, but remember that accesses which miss in L1 but hit in VC do not pay the miss penalty.