Homework 4 - CIS501 Fall 2011

Instructor: Prof. Milo Martin
Due: Noon, Tuesday, November 8th (at the start of class)

Overview

In this assignment you will explore the effectiveness of caches and the impact of different cache configurations. Your task will be use the program trace format used in the previous assignments (see Trace Format document) to evaluate the impact of cache size, associativity, block size, etc.

To do this, you'll write a program that reads in the trace and simulates different cache configurations. You're welcome to write the program in whatever language you like; although we are going to ask for a printout of your source code, to end result of this assignment are the experimental results (and not the program source code).

Processing the Trace

In this assignment we want to focus only on load and store micro-operations (and ignore all the other micro-ops). These micro-ops are easily identified as they have a "L" or "S" in the load/store column.

To help you debug your cache simulator, we've provided some annotated prediction results for entire 1K trace for two cache configurations (see below). Comparing your results to these annotated outputs should help ensure your code is working properly.

Question 1 - Cache Tag/Index/Offset Calculations

As talked about in class, a cache has three primary configuration parameters:

- **S**: Cache size (how much data the cache holds)
- **B**: Block size (the granularity of the data)
- **A**: Associativity (number of blocks with the same index that can be in the cache at the same time).

These three parameters the number of sets and ways in the cache. These parameters also specify how a memory reference address is split into its three components: block offset bits, index bits, and tag bits.

Assuming 32-bit address, calculate the following values for a two-way set associative 32KB cache with 64B blocks (that is: S=32KB, B=64B, and A=2):

a. The number of bits in the block offset
b. The number of "sets" in the cache
c. The total number of block frames in the cache
d. The number of index bits
e. The number of tag bits

Question 2 - Generalizing the Calculations

Assuming 32-bit address, write formulas in terms of *S*, *B*, and *A* for calculating the following quantities. You may find using *log(S)*, *log(B)*, and *log(A)* in the formulas easier than *S*, *B*, and *A* directly. For this assignment, you may assume *S*, *B*, and *A* are all powers of 2. Write the formulas for:

a. The number of bits in the block offset
b. The number of "sets" in the cache
c. The total number of block frames in the cache
d. The number of index bits
e. The number of tag bits

Hint: you may find it helpful to first work out the numbers for a direct-mapped cache (that is, consider just *S* and *B*), and then extend it to set-associative caches.

Question 3 - Extracting Bits from Addresses

When writing your cache simulator, you'll find it useful to have function (or methods) to extract the tag bits and index bits from an address. Given the four integer variables: *address* (address of the access), *num_tag_bits* (number of tag bits), *num_index_bits* (number of index bits), "*num_offset_bits* (number of offset bits), use bit manipulation operations to write C/C++/Java code to:

a. Calculate the *tag* of the address (an integer value):

```
tag = <your code here>;
```

b. Calculate the *index* of the address (an integer value):

```
index = <your code here>;
```

IMPORTANT: Do not convert the integer values to character strings (which is slow and unnecessarily verbose). Using integer bit manipulation instructions (shifts, bitwise and, or, not, etc.) can be used to to easily and efficiently accomplish such bit extraction operations. Hint: to generate an integer value with all bits set to one, you can use `~0` (bit-wise not of zero).

Question 4 - Miss Rate vs Cache Size

The first experiment is to gauge the impact of cache size on cache miss rate. Write a program to simulate a direct-mapped cache parametrized by *S* and *B*. You may actually find it more convenient to configure your cache in terms of *log(S)* and *log(B)* instead of *S* and *B*.

For this first direct-mapped cache simulations, the contents of the cache can be represented by a single array of values: the tag array. To avoid needing to track a separate valid bit, just initialize all the tag values to zero and assume that all blocks are valid and remain valid throughout the simulation. As we're not actually manipulating the actual data values (as a real processor would) but just calculating hit/miss for memory reference, there is no need to track the contents of the data array (in fact, the trace doesn't even have the information you would need to do this).

For now, handle loads and stores in the same fashion.

Use your cache simulator to produce cache miss rates for varying cache sizes. Generate the data for caches from 256 bytes (2^8) to 4MB (2^{22}). Configure the block size to 64 bytes. Generate a line plot of this data. On the y-axis, plot the "cache miss rate (percent of all memory references)". The smaller the miss rate, the better. On the x-axis, plot the log of the cache size (basically, the log of the cache sizes), in essence giving us a log scale on the x-axis.

a. How large must the cache be for the miss rate to be less than 10%? How large to be less than 5%?

b. Today's processors generally have 32KB or 64KB first-level data caches. By what ratio does increasing the cache size from 32KB to 64KB reduce the miss rate? (2.0 would be halving the miss rate; 1.0 would be no change in miss rate; less than 1.0 would be an increase in misses).

Question 5 - Set-Associative Caches

Modify your simulator to support two-way set associative caches. Each set in the cache will now need two "tag" entries (one for each way), and a single "LRU" (least recently used) bit used for cache replacement. Whenever you make a memory reference to a block, update the LRU bit to point to the other entry in the way. When replacing a block, replace the LRU entry (and don't forget to update the LRU bit to point to the other entry).

Generate miss rate data for the same block size and caches sizes as in the previous question, but simulate two-way set-associative caches. Plot this result with the original data collected for a direct-mapped cache (two lines on the graph: one for direct-mapped caches and one for the two-way set-associative cache).

a. How large must the set-associative cache be for the miss rate to be less than 10%? How large to be less than 5%?

b. How large must the direct-mapped cache be before it equals or exceeds the performance of the 16KB two-way set-associative cache (ignore any noise in which the direct-mapped cache might slightly outperform the set-associative cache for some configurations)?

c. This graph shows that the performance gap between set-associative and direct-mapped caches narrows as the cache size grows very large. Explain what likely causes this narrowing.

As the set-associative cache performs better than the direct-mapped cache, all subsequent experiments use the two-way set associative cache.

Question 6 - Traffic of Write-Back vs Write-Through Caches

Modify your simulator to calculate the total number of bytes transfered. The first source of traffic is the data transfers to fill cache blocks. This quantity is easy to calculate: it is just the number of misses multiplied by the block size. The other source of traffic is write traffic (either writeback or writethrough). In this question, you'll calculate the traffic for both write-back and write-through caches.

- **Write-through**: For the write-through cache, the traffic is the average number of bytes written by each store multiplied by the number of stores. The trace doesn't contain how many bytes each store writes, but for a similar trace we used in the past it was approximately 4 bytes per write, so use that in your calculations. You'll need to modify your simulator to count the number of stores.
- **Write-back**: For the write-back cache system, the traffic is the number of dirty evictions multiplied by the block size. Because at most one eviction happens per miss, this writeback traffic can at most double the overall traffic (in the case in which all evicted blocks were dirty). To calculate the number of dirty evictions, modify your simulator by adding a "dirty" bit for each block in the cache. Set the dirty bit whenever a store operation writes to the block. This dirty bit is *not* per word or per byte; it is one dirty bit for the entire cache block. (As this is a two-way set-associative cache, you'll need a dirty bit for each of the frames in the set; be sure to set the correct dirty bit.)

In either case, use a write-allocate policy. As such, the number of misses between write-through and write-back is the same under our assumptions (no write buffers and such), only the traffic is different.

For a two-way set-associative cache with 64-byte blocks, generate a graph plotting the same cache sizes as the previous question (on the x-axis) versus total data traffic per memory operation (total number of bytes transferred divide by the total number of memory references). Plot two lines: (1) the write-through cache (miss fill traffic + write-through traffic) and (2) the write-back cache (miss fill traffic + write-back of dirty blocks).

a. At what cache size do the two write policies generate approximately the same amount of traffic?

b. Why does the difference between the two schemes diverge at *large* cache sizes?

c. Why does the difference between the two schemes diverge at *small* cache sizes?

As the write-back cache uses less traffic for reasonable cache sizes, all subsequent experiments use write-back caches.

Question 7 - Cache Block Size

Now let's explore the impact of different cache block sizes. Modify your simulator to support various block sizes. Use the simulator to calculate the miss rate for a 32KB, two-way set-associative, write-back cache with several different block sizes: 8B, 16B, 32B, 64B, 128B, 256B, and 512B blocks. Create two graphs from this data:

- **Miss Rate Graph**. Plot the data on graph with miss rate on the y-axis, log of the cache **block** size on the x-axis, and the miss rate plotted as a line.
- **Traffic Graph**. Plot the data on graph with bytes per memory reference on the y-axis, log of the cache **block** size on the x-axis, and the miss rate plotted as a line.

Answer the following questions:

a. What is the block size with the lowest **miss rate**?

b. What is the block size with the lowest *traffic* (bytes per memory reference)?

c. What are the two (or three) sources of additional traffic as the block size grows? Explain why each component grows.

d. Given that current processors typically use, say, 64B blocks, which metric (miss rate or traffic) are today's caches designed to minimize?

Question 8 - Way Prediction

One way to reduce the hit latency penalty of a set-associative cache is with way prediction. Way prediction allows the data array and tag array to be accessed in parallel. If the predicted way was correct (determined by a tag match), no penalty occurs. If the prediction was incorrect, additional cycles are needed to find the data.

A way predictor is a table that guesses which "way" a given address should access. For our two-way set-associative cache, this is just a single bit per entry (way "zero" or way "one"). The table uses *n* bits of the data block address to index into the table that contains 2^n entries. The "offset" bits of the address are not used, so the *n* bits are the lower order bits from the combination of tag and index bits.

The way predictor is trained in two cases. First, whenever a block is inserted into the cache because of a miss, set the corresponding entry in the way predictor to record in which way of the cache the block was placed. Second, whenever a way mis-prediction occurs, update the way predictor to reflect the way of the block that caused the mis-prediction.

Simulate a way predictor for a two-way set-associative cache with 64B blocks and capacity of 32KB. Simulate way predictors sizes ranging from one entry to 2^{16} entries. Record how many way mis-predictions occur by categorizing each access as either a normal hit, a way mis-predicted hit, or a normal cache miss. If the block isn't in the cache, it isn't consider a mis-prediction; it is considered a cache miss.

Calculate the "percentage of cache hits that are way mis-predicted" (the lower the better). Plot a graph of this mis-prediction rate (y-axis) versus way predictor size (x-axis).

a. How accurate is just a single-entry way predictor? Was it more or less accurate than you anticipated? Give two reasons that explains its accuracy.

b. How large must the predictor be for the number of mis-predictions + cache misses to be **smaller** than the miss rate of a direct-mapped cache with the same capacity and block size? Basically, we're asking you to determine the point at which the two-way set associative cache that uses a way predictor is almost certainly better than a direct-mapped cache.

c. At the predictor size calculated above, approximately what is the percentage overhead in number of bits that the way predictor adds to the overall cache? (You can ignore the cache's tags if you wish to simplify the calculation.)

Test Cases

We have also provided you with a sample trace *output* run on "1K" shorter version of the trace. The output files have the following format:

```
[Set 0: {Way 0:000000, C} {Way 1:000000, C} LRU: 0] [Set 1: {Way 0:000000, C} {Way 1:000000, C} LRU: 0]  | bfede200 S miss clean
[Set 0: {Way 0:ffb788, D} {Way 1:000000, C} LRU: 1] [Set 1: {Way 0:000000, C} {Way 1:000000, C} LRU: 0]  | bfede200 S  hit    --
[Set 0: {Way 0:ffb788, D} {Way 1:000000, D} LRU: 0] [Set 1: {Way 0:000000, C} {Way 1:000000, C} LRU: 0]  | bfede200 S  hit    --
```

The contents of each set are specified, these are the contents of each way (way number: tag, state, where C=clean, D=dirty) and the LRU way for that set. Each line says what the contents of the cache were before a particular cache access and then lists the address being accessed, the access type (S for store, L for load), the outcome (hit or miss) and if the state of the LRU way being evicted (clean or dirty on a miss, none on a hit). Note: the address in the trace above is the "block address". That is, it is the address from the trace with the offset bits set to zero.

The two output traces are:

- cache-128-assoc-1-block-32.trace.output: A direct-mapped cache with 32B blocks and a total cache size of 128 bytes.
- cache-256-assoc-2-block-64.trace.output: A two-way set-associative cache 64B blocks and a total cache size of 256 bytes.

What to Turn In

- **Short answers**. Turn in your answers to the above questions. I strongly urge you to type up your answers.
- **Graphs**. Turn in printouts of the following graphs. One per sheet of paper (to make them big enough to read). Label the axises and give each line a descriptive names in a legend.
  - Graph A [Questions #4 and #5]: Miss rate for various cache sizes. Two lines: direct-mapped and two-way set associative.
  - Graph B [Question #6]: Traffic vs cache size. Two lines: write-through and write-back caches.
  - Graph C [Question #7]: Impact of cache block size on miss rate. One line: miss rate for various block sizes.
  - Graph D [Question #7]: Impact of cache block size on traffic. One line: bytes of traffic per memory operation for various block sizes.
  - Graph E [Question #8]: Impact of way prediction. One line: way mis-prediction rate (percent incorrect)
- **Source Code**. Print out your final source code for your cache simulator.

Hints & Comments

- **Automation of data collection**. Put a little bit of thought into how you're going to collect the various data for each question. If you're running your program multiple times by hand and then copying and pasting the individual results into Excel, you're doing something wrong (or at least super inefficiently). Your code should be able to perform a range of simulations in an automated fashion (either by building it into the main program of using scripts to call it with different parameters). You program should be able to split out data files that can be imported into Excel (or whatever) to make graphs with minimal tedium.
- **Code reuse**. In the end, the total amount of code needed to perform these simulations isn't actually that much. Don't go crazy in making the code super modular, but conversely there is significant potential for code reuse that shouldn't be ignored.
- **Runtime**. As with previously assignments, running through the trace should take too long if implemented efficiently. Even if you run through the entire trace to generate each data point, it shouldn't take more than few minutes to generate all the data you need for this assignment. If your code is taking much longer, you're probably not doing something right (ask me or the TAs about it).

  The most common culprit for greatly increased runtime in the past have been: (1) using string objects rather than using bit manipulation operations (shift, bit-wise and, bit-wise or, etc.) for extracting the index and tag bits, and (2) allocating lots of heap objects as part of processing each line in the trace.
- **Computer resources**. If you wish to write the code by ssh'ing into a Linux box, you can log into minus.seas.upenn.edu. These machines are multi-core multi-Ghz 64-bit chips, so they are quite fast. However, if your jobs run for too long (say 10 or 20 minutes) the system may kill off the job (to make sure that users don't use these machines for very long running jobs). Of course, you're also welcome to write and run the code on your personal computers or the lab machines.

Addendum

- None, yet.