Assignment 6: The Power of Caches

Due by: November 21, 2015 before 10:00 pm
Collaboration: Individuals or Registered Pairs (see Piazza). It is mandatory for every student to register on Piazza.
Grading: Packaging 10%, Style 10%, Design 10%, Correctness/Functionality 70% (where applicable)

Overview

This assignment is all about caches and their impact on performance. You'll do two things for this assignment: First you'll experiment with two simple C programs to gain first-hand experience with the impact of caches on real machines. Second you'll write a cache simulator that can be used to study and compare the effectiveness of various cache configurations.

If you have any questions or concerns about this assignment, please contact us **right away**. It may not look like much at first, but getting the details of caches right is actually a little harder than getting the details of branch predictors right...

Problem 1: Cache Experiments (20%) – data files are included.

For this problem you will first perform various measurements on the following very simple C program; it's not really important whether you know C or not; you can (and should!) take a look at file ‘matrix.c’ . (There's also ‘timeit.c’; you'll need below and a Makefile for building both.)

```
#include <stdlib.h>

#define SIZE 4096

int matrix[SIZE][SIZE];

int main(void)
{
   int i, j;
   for (i = 0; i < SIZE; i++) {
      for (j = 0; j < SIZE; j++) {
         matrix[i][j] = 107;
      }
   }
   return EXIT_SUCCESS;
}
```

Just in case this code confuses you, here's a short summary: Never mind the line with #include it's just something we need to do. The #define line introduces the constant SIZE which is used on the next line to define a two-dimensional array of integers. Then we get to the main function where a C program starts running. We use two nested for loops to run over the matrix cell-by-cell, and we set each cell to the value 107. That's it, that's all the program does; the final line just ends the program properly.

You are going to compile and run this program in various ways to experiment with the impact of caches on performance. **We assume that you will do this problem on a Linux system!** First compile the program thusly (or using the Makefile we gave you):

```
gcc matrix.c -O0 -o matrix
```

The -O0 option disables all optimizations so that we can study the performance using a simple-minded compiler first. Next run the program thusly:

```
time ./matrix
```

This will tell you how long it took to execute the code on your machine; you may want to repeat this experiment a few times to make sure that you got accurate results. Next run the program thusly:

```
valgrind --tool=cachegrind --branch-sim=yes ./matrix
```

The valgrind tool will collect detailed information about the cache performance (and the performance of the branch predictor) so you can see how well those things work. (You may have to install the valgrind tool using the package manager of whatever Linux distribution you are using.)

Now for the big experiment: Change the body of the loop from matrix[i][j] = 107 to matrix[j][i] = 107 instead! Recompile the program as before and re-run it. You should get **very** different performance now, and you should be able to clearly tell why (especially after re-running valgrind).

Write up your observations so far in your README file and feel free to speculate why exactly the performance and cache behavior of the two versions of this program are so very different. In addition you should do two more things:

- Experiment with different SIZE constants to see if the performance of the program (especially its cache performance) changes significantly. For instance, is there a size of the matrix for which it **doesn't** matter in which order we apply the indices? Why or why not?
- Experiment with different optimization levels. See if using -O1 or -O2 or -O3 or -O4 produce programs that perform better, for either index order.

Give a summary of your observations in your README file and try to draw some conclusions about both the effect of matrix size and the effect of compiler optimizations on cache performance for this program.

Important: If you find the results you're getting in VirtualBox strange, it may be a good idea to repeat your experiments on a machine that runs Linux natively. Sometimes VirtualBox has secondary effects on things like cache performance, and since that's exactly what you're trying to measure VirtualBox may be unsuitable for this kind of work.

Hint: We also provided a second program that tries to evaluate the cache performance differently by looking at the time it takes to perform a single read or write. You may be able to use that program to your advantage. Play with it, maybe you get another insight that you can use to write something smart into your README file.

Problem 2: Cache Simulation (80%)

For this problem you will design and implement a **cache simulator** that can be used to study and compare the effectiveness of various cache configurations. Your simulator will read a **memory access trace** from a given file, simulate what a cache based on certain parameters would do in response to these memory access patterns, and finally produce some summary statistics. Let's start with the file format of the memory access traces:

```
s 0x1fffff50 1
l 0x1fffff58 1
l 0x1fffff88 6
l 0x1fffff90 2
l 0x1fffff98 2
l 0x200000e0 2
l 0x200000e8 2
l 0x200000f0 2
l 0x200000f8 2
l 0x30031f10 3
s 0x3004d960 0
s 0x3004d968 1
s 0x3004caa0 1
s 0x3004d970 1
s 0x3004d980 6
l 0x30000008 1
l 0x1fffff58 4
l 0x3004d978 4
l 0x1fffff68 4
l 0x1fffff68 2
s 0x3004d980 9
l 0x30000008 1
```

As you can see, each memory access performed by a program is recorded on a separate line. There are three "fields" separated by white space. The first field is either "l" or "s" depending on whether the processor is "loading" from or "storing" to memory. The second field is a 32-bit memory address given in hexadecimal; the "0x" at the beginning means "the following is hexadecimal" and is not itself part of the address. You can **ignore** the third field for this assignment. (This format as well as some of the traces you'll use to test your simulator come from a similar programming assignment by Steven Swanson at UC San Diego; thanks Steven!) Here are two example traces: ‘gcc.trace’ and ‘swim.trace’

Your cache simulator will be configured with the following cache design parameters which are given as command-line arguments (see below):

- number of sets in the cache (a positive power-of-2)
- number of blocks in each set (a positive power-of-2)
- number of bytes in each block (a positive power-of-2, at least 4)
- write-allocate (1) or not (0)
- write-through (1) or write-back (0)
- least-recently-used (1) or FIFO (0) evictions

Note that certain combinations of design parameters account for direct-mapped, set-associative, and fully associative caches: A cache with n sets of 1 block each is essentially direct-mapped, a cache with 1 set of n blocks is essentially fully associative, and a cache with n sets of m blocks is essentially m-way set-associative. The smallest cache you need to be able to simulate has 1 set with 1 block with 4 bytes; this cache can only remember a single 4-byte memory reference and nothing else; it can therefore only be beneficial if consecutive memory references in a trace go to the same address. **You should probably use this tiny cache for basic sanity testing.**

A brief reminder about the other three parameters. The **write-allocate** parameter determines what happens for a **cache miss** during a **store**: if true, then a store brings the relevant memory block into the cache before it proceeds; if false, a cache miss during a store does not modify the cache at all; obviously this parameter interacts with the next one. The **write-through** parameter determines whether a store **always** writes to memory **immediately** or not: if true, then a store writes to the cache as well as to memory; if false, then a store writes to the cache only and marks the block dirty; if the block is evicted later, it has to be written back to memory before being replaced. **It doesn't make sense to combine no-write-allocate with write-back because we wouldn't be able to actually write to the cache for the store!** The last parameter is only relevant for associative caches: in direct-mapped caches there is never a choice which block to evict to make room for a new one! The **least-recently-used** policy picks the block that has not been **accessed** the longest for eviction; the **FIFO** policy picks the block that has **been in the cache** the longest for eviction.

Your cache simulator should assume that loads/stores from/to the cache take **one** processor cycle; loads/stores from/to memory take **100** processor cycles for **each** 4-byte quantity that is transferred. There are plenty of things about caches in real processors that you do **not** have to simulate, for example write buffers or smart ways to fill cache blocks; implementing all the options above correctly is already somewhat challenging, so we'll leave it at that.

Depending on your choice of implementation language, we expect to be able to run your simulator as follows:

- C/C++: ./csim 256 4 16 1 0 1 gcc.trace
- Java: java CacheSimulator 256 4 16 1 0 1 gcc.trace
- Python: python csim.py 256 4 16 1 0 1 gcc.trace

Each of these commands would simulate a cache with 256 sets of 4 blocks each (aka a 4-way set associative cache), with each block containing 16 bytes of memory; the cache performs write-allocate but no write-through (so it does write-back instead), and it evicts the least-recently-used block if it has to. (As an aside, note that this cache has a total size of 16384 bytes (16 kB) if we ignore the space needed for tags and other meta-information.)

After the simulation is complete, your cache simulator is expected to print the following summary information in **exactly** the format given below:

```
Total loads: 318197
Total stores: 197486
Load hits: 314798
Load misses: 3399
Store hits: 188250
Store misses: 9236
Total cycles: 9344483
```

Note that there may be a bug in these results. You should probably compare amongst yourselves by posting test cases on Piazza and discussing them...

Hint: Please note that your simulation is only concerned with hits and misses, at no point to you need the **actual** data that's stored in the cache; that's the reason why the trace files do not contain that information in the first place.

Hint: Use the data structures provided by the standard library or language of your choice to avoid having to write lots and lots of supporting code; plain C is probably the worst choice for implementing this project quickly because the C standard library lacks sophisticated data structures.

Hint: Don't try to implement all the options right away, start by writing a simulator that can only run direct-mapped caches with write-through and no-write-allocate. Once you have that working, extend step-by-step to make the other design parameters work. Also, sanity-check your simulator frequently with simple hand-crafted traces for which you can actually still derive manually what the behavior should be.

Deliverables

Please turn in a gzip compressed tarball of your assignment; the filename should be HW1_<your_name>.tar.gz with name replaced by the first part of the email address you used to register on Piazza (so I would use HW6_kmemon1.tar.gz).

Pairs submit one archive together, not two archives separately! Include a README file that briefly explains what your programs do and contains any other notes you want us to check out before grading; your answers to **written** problems should be in this file as well.

Grading

For reference, here is a short explanation of the grading criteria; some of the criteria don't apply to all problems, and not all of the criteria are used on all assignments. **Packaging** refers to the proper organization of the stuff you hand in, following the guidelines for Deliverables above. **Style** refers to either to programming style if we are talking about a programming problem (things like consistent indentation, appropriate identifiers, useful comments, generally simple, clean, readable code), or to the clarity and readability of your solution for a written problem. **Design** refers to proper modularization and the proper choice of algorithms and data structures for a programming problem, and the proper choice of abstractions for all problems. **Functionality** refers to your programs being able to do what they should according to the specification given above; if the specification is ambiguous and you had to make a certain assumption, defend that assumption in your README file.

If your programs cannot be built you will get no points whatsoever for programming problems. If your programs fail miserably even once, i.e. terminate with an exception of any kind or dump core, we will take off 10%; do proper error checking and handling! Finally, make sure to include your name and email address in *every* file you turn in (well, if there *is* a way to include that information, otherwise you can leave it out)!

Acknowledgement

This assignment was originally designed by Peter Fröhlich