Programming Project - Total Points 100

Cache Memory and Program Execution Speed

In this project your team will design a *C* program to measure the effect of cache memory on program execution speed. *Cache* is a layer of the memory hierarchy that stands between main memory and the CPU. Understanding how to write a program to take advantage of cache is important in optimizing performance. A memory reference that can be satisfied from cache could be 50 times faster than one that must go to main memory.

Semiconductor memory comes in various speeds, and faster memory costs more. A (not recent) advertisement lists 32KB of 12 ns memory for $12.75. For $11.50, the same ad offers 256KB of 70 ns memory. If you built an 8 Meg memory for your PC entirely with the faster memory, you'd spend about $3000 just on the memory, whereas 8 MB of the slower memory would cost only $368. For some applications, you may want to spend more money. For example, in designing a "supercomputer", which is designed to get the biggest bang regardless of cost, you would choose the more expensive memory solution.

Caching cleverly combines fast and slow memory to achieve advantages of each. Cache exists in nearly all computers built today, from PCs to large compute/storage servers. The clever cache organization is as follows: Main memory consists of cheaper, slower chips. We use the fast, expensive memory (i.e. cache) to store copies of (selected) recently used values and their addresses in a cache. A cache controller is interposed between the CPU and the two different memory technologies. When the CPU stores a value, a copy of the value and the address is saved in the fast memory. When the CPU asks for the contents of an address, the cache controller checks to see if it has that address stored in fast memory, and if so, it can return the value more quickly than if the value is only stored in the slow memory.

If 90 out of 100 requests can be satisfied by reference to the cache, and a cache reference takes 12 ns, and the remaining 10 requests take 80 ns to complete, the average time for a memory request will be 0.90×12 ns + 0.10×80 ns = 18.8 ns. Of course, because we now have to store the address as well as the value, we'll lose big if we have to have 7MB (+28MB of address) of expensive fast memory in order to reach a 90% hit rate.

It turns out that most programs exhibit good *temporal locality*, that is, they tend to refer to the same words of memory over and over, either in code loops, or as data accesses. So a PC cache of 128K is big enough to achieve a 90% hit rate for most programs. In order to avoid having to store another 512KB of addresses, several different schemes are used; one frequent scheme is to organize adjacent bytes into cache blocks that can share the same address. For example, if the cache is organized into 128-byte cache blocks, each of the 128 bytes in a block will have the same high-order bits, and so we need store only 1/128 as many addresses. For our example, we could get away with storing only 1024 addresses.

Most current operating systems have no direct role in the implementation of caching; the hardware tries to hide the details from the software, and the trade-offs change rapidly. But if you were attempting to maximize the performance of your computer, you might want to take cache properties into account. Note that if you are running a program on our hypothetical PC (mentioned above) that never hits cache, your program can run 70/12 times longer than a program which is carefully tuned so that it always hits cache.

Basic Requirements

The computer system you are using has a cache system. In this programming project, your team will seek to discover **through experimentation** what your cache and main memory parameters are. You will be required to answer the following questions:

1. How big is a cache block?
2. How big is the cache?
3. How long does a reference to main memory take to complete?
4. How long does a reference that can be satisfied from cache take to complete?

The basic idea is to compare the execution time of two loops, one of which tries to force reading a new value from main memory at each iteration. Because we are talking only a few ns difference between the two read times, you'll have to execute a lot of reads in order to be able to measure the difference. **Note**: the answers to the four questions can be easily be obtained on the web, however this assignment requires you to ascertain these values **experimentally** through careful timings of specific access patterns using arrays.
The experimental program to solve this problem would be more accurate (timing-wise) if it were written in assembly language. I don't expect you to do that. However, you can determine what machine language instructions occur in the high-level loops you are timing. On UNIX systems, one way to do this is with the -S switch to the C compiler. If the compiler does not provide such a switch you can use a debugger to step through (or examine) the assembly code.
Your final submitted program won't be very long, although you may write and discard several versions "along the way", because they may not work. This is a difficult assignment. So, for any partial credit, give clear explanations of how you propose to perform the measurements, and what you think may have worked or gone wrong. Put this explanation in a file called, "README.txt", along with your answers to the five questions listed above.

Modern computer systems make your effort even more challenging by having several layers of cache. For example most current microprocessors have a significant amount of cache on the chip with the CPU (L1 cache), and also have one or more "off-chip" caches that are significantly faster than main memory (L2 and L3 caches), but slower than L1 cache.

Project Files

1. **.h and .c files**: No .h or .c files are provided. Your team must define the header files and implementation files (including all the standard libraries) needed to fully implement the basic requirements outlined in above. **Only one main()** method should be defined. HINT: The code necessary to answer the four questions is naturally modularized into four functions.
2. **Makefile**: No Makefile is provided. Your team must create one Makefile that can compile your program into a working executable.

Collaboration and Plagiarism

This is an **team assignment**. Collaboration **is only permitted** between team members that are in the same team. Plagiarism will not be tolerated. Submitted solutions that are very similar (determined by the instructor) will be given a grade of zero. Please do your own work, and everything will be OK.

Submission

Create a compressed tarball, i.e. *tar.gz*, that includes the README.txt file, the Makefile and all the .h and .c files needed to compile and execute your program. The name of the compressed tarball **must only be the last name of one team member**. For example, *ritchie.tar.gz* would be correct if the original co-developer of UNIX (Dennis Ritchie) submitted the assignment. Only assignments submitted in the correct format will be accepted (no exceptions). **Only one team member** will submit the compressed tarball to the Dropbox setup on OAKS. Your team may resubmit the compressed tarball as many times as you like, Dropbox will only keep the newest submission.

To be fair to everyone, late assignments will not be accepted. Exceptions will only be made for extenuating circumstances, i.e. death in the family, health related problems, etc. You will be given a week to complete this assignment. Poor time management is not excuse. Please do not email assignment after the due date, it will not be accepted. Please feel free to setup an appointment to discuss the assigned coding problem. I will be more than happy to listen to your approach and make suggestions. However, I cannot tell you how to code the solution. Additionally, code debugging is your job. You may use the debugger (gdb) or print statements to help understand why your solution is not working correctly, your choice.

Grading Rubric

For this assignment the grading rubric is provided in the table shown below.

Program Compiles	10 points
Program Runs with no errors	10 points
Makefile	10 points
README.txt	20 points
function implementation for size of the cache block	13 points
function implementation for size of the cache	13 points
function implementation for time for main memory access	12 points
function implementation for time for cache memory access	12 points

In particular, the assignment will be graded as follows, if the submitted solution

- does not compile: 0 of 100 points
- compiles but does not run: 10 of 100 points
- compiles and runs with errors: 15 of 100 points
- compiles and runs without errors: 20 of 100 points
- size of the cache block: 13 points
- size of the cache: 13 points
- time for main memory access: 12 points
- time for cache memory access: 12 points