CS 211: Computer Architecture, Fall 2012
Programming Assignment 4: Caching
Due: Dec 12, 2012. 11:55pm

1 Introduction

Your task in this homework is to measure the size of the processor's cache and cache block.

2 Basic Properties of Cache

Before caches were invented, the scenario was simple. The CPU was connected directly to the main memory where it could fetch instructions and data from. Problems arose when due to technological development, processor speed increased faster than main memory access times could be increased in an economical way. This resulted in machines with the processor waiting for the memory most of the time just to get data to process. Faster memory chips could be constructed but since they were more expensive than conventional main memory, creating memories as large as the main memory was not economical. That's when the idea of caches appeared.

The idea was to insert a second level of memory hierarchy, so that instead of the processor being directly connected to the main memory, the processor would get its data from the faster but smaller cache memory containing copies of part of the main memory. Whenever the processor needs some datum, the cache is first searched if that datum is available. If so, it is fetched from the cache without consulting the main memory. We call this case a hit. If the desired datum is not available in the cache, a block of memory containing that datum is copied from the main memory to the cache, and the request of the processor is granted from the cache. This case is called a miss, and time required to copy the block into the cache is called the miss penalty time. The size of the block copied is the cache block size, a fixed power of two. The cache contains several (also a power of two) of such cache blocks (also called cache lines) that make up the cache. So how could we measure all these?

3 Measuring System Cache Parameters

3.1 Main idea

Implement C programs to determine for a computer system using capacity misses:

Cache block size

Cache size

The main idea is to control the hits and misses in the cache. This way we know how many hits and misses we had, so several time measurements with different number of hits and misses can give us results we could compare and deduce the desired values. But how can we control hits and misses?

We might try creating a program where all memory accesses are misses. Knowing how caches work, this is not hard. We set up a large array in main memory and access elements of it some fixed distance apart. If that distance is greater than or equal to the cache block size, all accesses will be misses. Why is that? Let's say a cache line has 64 bytes and we access every 64th bytes of our array. Then all accesses will be misses, since only 64 bytes are copied to the cache at a time, so two bytes at least 64 bytes apart will not be copied at the same. But how can we figure out the cache line size?

With our program we should be able to try various distances, thereby guessing the cache line size. As long as our guess is above the block size, all our accesses will be misses, we will just have fewer of them. As soon as we go below the actual cache line size, we will have hits as well, since more than one of the memory locations we access will be copied into the cache at once. Suppose, for example, that the cache block size is 64 bytes and our guess is 32 bytes. Now, when we access a byte and it is a miss, a block of 64 bytes is read, so the next byte we will be accessing is also copied into the cache at the same time. Thus our next memory access will be a hit. So we should make our program compute the average time for one access, and supply guesses in decreasing order, starting at a large number, say 512 bytes, halving the guess every time. What we will see is that the average time will be constant for guesses greater than or equal to the cache line size, and decrease for guesses below that limit.

3.2 Solution Details

So here is finally some detailed explanation of the solution approach. The program declares the large array, using the sbrk() function. Sbrk changes the data segment size by the specified increment. Your program should include the relevant header files so that the compiler will know about various standard library functions we are using, eg.:

```
#include  <stdio.h>
#include <sys/time.h>
#include <unistd.h>
```

Our main() function starts out by declaring a variable:

```
register *a;
```

We try to put all the variables we will use during the memory accesses into registers since the only memory accesses we want are to the array. If local variables would reside in memory, accessing them could have an effect on our measurements.

Next thing is to allocate our array, an array of around 4 million integers, that is 16 MB on 32-bit machines.

```
a=sbrk(4*1024*4096+8192);
a=(int  *)((((int)a>>13)<<13)+8192);
```

When a program is started, some memory is allocated to it. The function sbrk() increases this memory by the given amount. After the function call, "a" will point to the beginning of this huge array. We adjust a so that it is page aligned. What is a page? and what is aligned? and why do we need that at all?

3.2.1 Paged Memory

As multiprogramming entered the operating system scene, suddenly main memory became too small. Programs would have liked to have the complete memory for themselves, but then only one program could have been run. The idea came about to move unused portions of programs to disk, thereby allowing other programs to reside in memory at the same time. This moving to disk is called swapping. Since moving single bytes to disk is inefficient, blocks of memory called pages are moved to disk at the same time. Pages are always of size of a power of two for efficiency, and are nowadays usually 4096 or 8192 bytes. When some program tries to access a memory location that is actually on the disk, then the whole page containing that location is moved into memory so that the program can continue.

3.3 Common Problems

If you get different times from the program measuring the cache size, try reducing the array size until you get more consistent results. The problem is that the time to execute the loop may take too long, so that other processes might be scheduled to run in the middle of the loop. If the array is big this may happen multiple times. This gives you results that vary a lot if the machine is heavily loaded.

To measure the cache block size, run the program given you multiple times with each cache block size guess, average the running times, and look at big differences in execution time for different guesses. If the times vary a lot, try reducing the array size as mentioned in the previous paragraph.

3.4 Some Guidelines for Taking Good Measurements

When you are measuring some parameters of a system where there are several other programs running there are more than just the constant time overheads imposed by your program! There is a lot of non determinism involved, for example you might be swapped out of memory and restarted later, there might be external events like interrupts which would cause unnecessary delay by stopping your program midway through execution.

Some of the above can be avoided or their effect greatly reduced by taking the following very simple measures:

Avoid page faults by getting the whole array in memory. The way to do it is to touch every page present in the array that you allocate so that it is in memory when you start taking measurements.

Do not print out to the standard output or any file the results because that is a very expensive operation and can take a very long time causing you to swap out.

Be careful accessing the array so that you don't go out of bounds and cause a segmentation fault!

The array size and the iterations should be reasonable. If the array size is very large then you stand a greater chance of getting swapped out and if it is very small then there is a very big component of noise that comes into play. The best way to find out the optimum size is to experiment so that you get consistent measurements. You may need to experiment with different array sizes to get the cache size.

Do not use operations like multiplication in the program because that takes a very long time on the modern pipelined machines as compared to other operations like addition and subtraction.

Do not measure individual memory accesses. Since the time taken to measure the access is much more than the time taken to access, it is not a very good thing to do. Besides there is also the issue of the least count of the timer being greater than the time that it takes to access the memory. The way to do it is to jump through the array in some reasonable number of iterations and take the average time taken to access the locations as the measure.

Do not write to the array that you have allocated, but read from it. That is because you do not know whether the cache is write-back or write-through.

To take accurate measurements, do a differential measurement, that is suppose you run the program for n1 and n2 iterations with the same jump size.

T1 = Constant overhead + n1 * Time for access
T2 = Constant overhead + n2 * Time for access

Therefore,

T1 - T2 = (n1 - n2) * Time for access
Time for access = T1 - T2 / (n1 - n2)

Now this removes the constant overhead under very simplifying assumptions but still it does provide a more accurate measurement.

Here is a code fragment that will get you started:

```
struct timeval start;
struct timeval end;
int times, i, dummy, timeTaken;

gettimeofday(&start,NULL);
for (times=0; times<SOME_LARGE_NUMBER; times++){
 for (i=0; i<ARRAY_SIZE; i=i+SOME_ACCESS_SIZE){
      dummy=a[i];
 }
}
gettimeofday(&end,NULL);

/*timeTaken is the time for execution in micro seconds (usec)*/
timeTaken = (end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec
*1000000 + start.tv_usec);
printf("The time taken is: %ld",timeTaken);
```

Note: Make sure you normalize your timeTaken variable here, as it is not the time taken for accessing the array once, but the time for accessing SOME_LARGE_NUMBER of times.

3.5 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named pa4.tar that can be extracted using the command:

```
tar -xf pa4.tar
```

Your tar file must contain:

cacheblock.c: The C program that measures the cache block size.

cachesize.c: The C program that measures the cache size.

Readme.pdf: The file describing the program logic for all the three above C programs. This file should have your name at the top. Additionally, it should contain the traces of the program, the inputs and outputs along with an explanation of the results. It should list your conclusions about the cache and cache block sizes, and justifications for those conclusions. Graphs illustrating access time versus access size would be a great way to do this.

Makefile: You must have the following rules in your Makefile:
cacheblock for compiling cacheblock.c, generates an executable cacheblock.

cachesize for compiling cachesize.c, generates an executable cachesize.

Any additional files that you need to run your code.

**We will compile and test your programs on the iLab machines so you should make sure that your programs compile and run correctly on these machines. If your program will not compile and run on those machines, you will receive a zero for the assignment.

**You may develop in any environment, however for the sake of consistency and ease of checking of results, your submitted execution, and analysis of cache size should be on the iLab machines; eg we are interested in your program determining the size of the cache and cache block for the iLab machines, not your individual PC!

4 Grading Guidelines

4.1 Functionality

You should make sure that we can build your program by just running make. If

not, you will receive a grade of zero for the assignment.

You should test your code as thoroughly as you can.

Be careful to follow all instructions. If something doesn't seem right, ask.

4.2 Coding Style

Having said the above about functionality, it is also important that you write "good" code. Thus, part of your grade will depend on the quality of your code. Here are some guidelines for what we consider to be good:

Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.

Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.

You use variable names should have some meaning (rather than cryptic names like i).

Further, you should observe the following protocols to make it easier for us to look at your code:

Define prototypes for all functions.

Place all prototype, typedef, and struct definitions in header (.h) files.

Error and warning messages should be printed to stderr