**CPSC 261: Lab 5**

**Lab 5 - Cache memory performance**

* [Goals](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#goals)
* [Introduction](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#introduction)
* [Overview](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#overview)
* [The Details](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#details)
* [Provided Materials](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#provided)
* [Handing It In](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#handing)

**Goals**

The goals of this lab are to:

* Help you understand how cache memories work
* Understand how to improve the execution performance of a C program by thinking about the order in which accesses will be made and how that will impact the cache.

**Introduction**

In Lab 4 you worked to improve the performance of the issawtooth function using all the tricks you could think of. In this lab we are concentrating on cache memory: how it works, and how to take advantage of it to improve the performance of a C program.

**Overview**

This lab has two main parts:

* First you must complete a C program that simulates the behaviour of a memory cache given a sequence of accesses, in a manner similar to what we did in class or on the midterm practice problems. However, instead of doing these computations *by hand* you will be writing functions to do the computations.
* Next, you must take an existing simple C function and by using what you know about cache memories make it run as fast as possible.

**The Details**

* [Step 1](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#step1) Complete the tagFromAddress and and setindexFromAddress functions in simulate.c
* [Step 2](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#step2) Complete the updateLastUsedOrder and lruLineInSet functions in simulate.c
* [Step 3](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#step3) Complete the inSet and addToSet functions in simulate.c
* [Step 4](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab5.html#step4) Make closure run as fast as possible
* Step 1 - Complete the tagFromAddress and and setindexFromAddress functions in simulate.c

You have been given the skeleton of a simulation program in a source file named simulate.c. This program reads a sequence of memory addresses from its standard input and simulates the effectiveness of a cache characterized by a number of constants defined at the top of the file. The main part of the simulator is complete, but it depends on a number of functions that you need to implement.

Along with the simulator, you have been provided a program called generate.c that generates a sequence of memory addresses. The number, range, and distribution of the addresses are determined by the constants defined near the beginning of the file. The given settings are appropriate to get you started, but you may want to change them if you want to experiment with your cache simulator. The following constants are important:

HOWMANY

How many addresses should be generated? For initial debugging, this is set to 64. If you want to determine the steady state hit rate of your cache, you will probably want to make this bigger.

MAXADDR

What range of addresses are generated? Change the literal 8 in the definition of this constant to the number of address bits that you want to have in the addresses. 8 is appropriate for the default cache simulator settings.

HOT

This controls the shape of the distribution of addresses. If you define the constant HOT, then you get a distribution in which HOTPERCENTAGE of the addresses are within a smaller range (HOTFRACTION) of the addreses space, and the rest of the addresses are spread throughout the remaining portion of the address space. This is the default.

You can also choose instead to define the constants NORMAL or UNIFORM which give a different distribution of addresses.

The generate.c and simulate.c programs are set up so that you can run them like this:

./generate | ./simulate

The first two of these functions compute the tag and set index for a cache from a memory address:

long tagFromAddress(long address)

This function determines the tag of an address. It needs to remove from the address all of the bits that are used as offset into the block and as the set index and return all the remaining bits. You might want to use the C operators >> which right shifts the bits in an int or long value, and & which computes the bitwise *and* of two int or long values.

long setindexFromAddress(long address)

This function returns the set index of the set in the cache that might contain the block containing this memory address. It must be a long value in the range of 0 through NSETS - 1, where NSETS is the number of sets in the cache. Again the C operators >> which right shifts the bits in an int or long value, and & which computes the bitwise *and* of two int or long values will probably be useful.

At this initial stage, you probably want to add some printf calls to these functions to ensure that they are doing the right thing.

* Step 2 - Complete the updateLastUsedOrder and lruLineInSet functions in simulate.c

The next two functions are helpers that manage cache lines:

void updateLastUsedOrder(long setindex, long tag)

This function updates the lastUsedOrder fields of the lines in one set of the cache. It determines which line of the indicated set contains the given tag (it may assume that the tag is in the set), and updates the lastUsedOrder field of that line to 0 (to indicate that it is the most recently used line). All of the other valid lines in the set also get their lastUsedOrder fields updated so that the lastUsedOrder fields have the values 0 .. numberOfValidLines - 1.

Think about this a bit before you start writing code. It might be trickier than you think it is.

The skeleton function has a call to checkValidLastUsedOrder(setindex) at the end of it. This helper function ensures that the constraints on the lastUsedOrder field have been established by your function.

long lruLineInSet(long setindex)

This function returns the index of the line in the given set (the one associated with setindex) that has been used least recently. That is, its lastUsedOrder field is higher than every other. The only wrinkle with this function is that if any of the lines in the set are invalid (their valid field is 0), then this function should return the index of that line.

At this initial stage, you probably want to add some printf calls to these functions to ensure that they are doing the right thing.

* Step 3 - Complete the inSet and addToSet functions in simulate.c

Now, you only have two functions left to implement, one which determines if an access is a hit or a miss, and the other which updates the cache on a miss:

int inSet(long setindex, long tag)

If the indicated tag is in one of the cache lines in the set identified by setindex, then return the index of that cache line in the set. Otherwise, return -1;

long addToSet(long setindex, long tag)

This function updates the cache when an access results in a miss in the cache. It uses the function lruLineInSet() to determine which line to replace, and updates that line with the information from this access: the valid indicator and the tag. It also calls updateLastUsedOrder() to update the last used order of the lines in the set. It returns the index of the line that has been updated.

Once you have completed all of the functions, you can run the simulator to make sure that it works like it should. When you are satisfied that your simulator works, you should increase the value of the HOWMANY constant in generate.c (to say 64000) and run the simulator again (turning off VERBOSE first). If you are using the HOT option in generate.c, and you haven't tweaked any other parameters, then the hit rate of your cache should be a bit higher (say 5% or so) than the HOTPERCENTAGE constant in the generate.c program. The hit rate will be lower with a small number of accesses because of compulsory misses as the cache *warms up*.

* Step 4 - Make closure run as fast as possible

You have been given a timegraph.c file that runs and times a closure function. You have also been given an implementation of the closure function in a C source file named graph.c. You don't need to worry too much about what the closure function does (it computes the transitive closure of a directed graph stored as an adjacency matrix). There is a correct implementation in graph.c and also as base\_closure in timegraph.c.

Your task is to think about cache performance and use this knowledge to improve the performance of closure.

Keep track of each thing that you try and keep track of whether it increases the performance (or decreases it!) and by how much. Keep your log in your README.txt file. You should be able to decrease the time taken by the program by about 50% if you are clever enough.

Like with lab4, we will have a scoreboard where you can see where you rank among the class.

**Provided Materials**

You should use subversion to check out the lab5 directory from your repository, in the same way that you did for the previous labs.

Your lab5 directory should contain the following files:

* Makefile

The provided Makefile contains instructions for compiling and linking your programs. You should not need to, and should not, change the Makefile that we have provided for you.

* generate.c

A file that generates a random sequence of addresses to drive your cache simulator.

* simulate.c

A file contains the skeleton of the cache simulator. In this file you complete the necessary functions to make a correct cache simulator.

* graph.h

This file contains the size of the matrices that are manipulated by the closure function.

* timegraph.c

This program contains a main program that calls your closure function and times its execution.

* graph.c

This file contains the default implementation of the closure function.

You will need to create all of the other files.