CPSC 313: Computer Hardware and Operating Systems
Assignment #3, due Wednesday, March 15 2017 at 23:59

This assignment is all about caching. It consists of two theoretical questions and two programming questions. In one you are asked to complete the implementation of some code to simulate the management of a cache. This will be done in C. In the other you are to use your knowledge of cache behaviour to improve the performance of a correct, but not so fast, C program. For this assignment you can work with a partner if you like and if you do you will receive bonus marks similar to the previous assignments. Groups of three or more people are not permitted. The late penalty is 33 1/3% per day. No late assignments accepted after 2 days. Just like our previous assignments you will use Stash to get the starting code and to handin the assignment. As always your code must compile and run on the Unix undergraduate machines. If your code does not compile or run on these machines you will get 0 for the code and execution components of the assignment. Also, you are not allowed to use any functions in math.h for any of the questions. If you think you need to use something in math.h, then you are doing something wrong.

[10] 1. Suppose we have a system with the following properties:

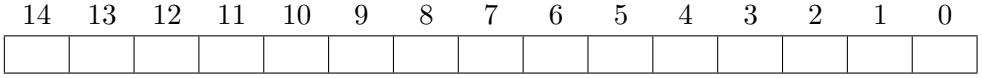
- The memory is byte addressable,
- Memory accesses are to `1-byte words` (not to 4-byte words).
- Addresses are 15-bits.
- The cache is four-way set associative ($E = 4$) with a 8-byte block size ($B = 8$) and eight sets ($S = 8$).

The contents of the cache are as follows, with all addresses, tags and values given in hexadecimal.

Set	Valid	Tag	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
0	0	032	43	F6	A8	88	5A	30	8D	31
	1	131	98	A2	E0	37	07	34	4A	40
	1	193	82	22	99	F3	1D	00	82	EF
	1	1A9	8E	C4	E6	C8	94	52	82	1E
1	0	193	82	22	99	F3	1D	00	82	EF
	0	04E	90	C6	CC	0A	C2	9B	7C	97
	1	063	8D	01	37	7B	E5	46	6C	F3
	0	191	79	21	6D	5D	98	97	9F	B1
2	1	1C5	0D	D3	F8	4D	5B	5B	54	70
	1	0FF	D7	2D	BD	01	AD	FB	7B	8E
	1	1BD	13	10	BA	69	8D	FB	5A	C2
	0	004	5F	12	C7	F9	92	4A	19	94
3	1	01A	FE	D6	A2	67	E9	6B	A7	C9
	1	058	EF	C1	66	36	92	0D	87	15
	1	17B	39	16	CF	70	80	1F	2E	28
	1	0D7	E0	D9	57	48	F7	28	EB	65
4	1	087	18	BC	D5	88	21	54	AE	E7
	1	1B5	4A	41	DC	25	A5	9B	59	C3
	0	00D	53	92	AF	26	01	3C	5D	1B
	1	002	32	86	08	5F	0C	A4	17	91
5	1	18B	8D	B3	8E	F8	E7	9D	CB	06
	1	103	A1	80	E6	C9	E0	E8	BB	01
	0	1E8	A3	ED	71	57	7C	1B	D3	14
	1	1B2	77	8A	F2	FD	A5	56	05	C6
6	1	10E	65	52	5F	3A	A5	5A	B9	45
	0	074	89	86	26	3E	81	44	05	5C
	0	0A3	96	A2	AA	B1	0B	6B	4C	C5
	1	1C3	41	14	1E	8C	EA	15	48	6A
7	0	1F7	C7	2E	99	3B	3E	E1	41	16
	1	036	FB	C2	A2	BA	9C	55	D7	41
	1	083	1F	6C	E5	C3	E1	69	B8	79
	1	031	EA	FD	6B	A3	36	C2	4C	F5

[2] (a) The following diagram shows the format of an address (one bit per box). Indicate the fields that would be used to determine the following:

- CO: the cache block offset.
- CI: the cache set index.
- CT: the cache tag.



[8] (b) For each of the following memory accesses, indicate the cache set that will be searched, the tag associated with the address, if it is a cache hit or miss **when carried out in sequence** as listed, and the value of a read if it can be inferred from the information in the cache. (If the value is not known indicate that the value is unknown) Justify your answers.

Operation	Address	Cache Set	Tag	Hit?	Value (or unknown)
Read	`0x02F8`				
Read	`0x35DD`				
Read	`0x7A2F`				
Read	`0x2055`				

[16] 2. One way to determine what a particular cache's miss rate will be is to run through a number of programs using a simulated cache, and take note of the number of cache hits and misses as a function of the total number of memory accesses. In this question, you will complete a C implementation of a simulated cache. Nine files are provided in the code directory. Five of them are relevant to this question and three are used in the final question. The files relevant to this question are:

- `cache.h`: a header file with some declarations.
- `cache.c`: the incomplete implementation of the cache.
- `cache-test.c`: a test program to help you debug your implementation.
- `cache-ref.o`: The object code from a reference implementation of the cache.c file. When you use the make command without arguments, this file gets linked to the object code from `cache-test.c` to produce the executable `cache-ref`. You can use this program to help determine if your implementation of the cache handling code is correct. **This object code file will only work on the undergraduate Linux machines.** We will not be providing versions that work with the plethora of systems you might be working with, however, if you are using a linux machine the file might work if you have all the right pieces.
- `Makefile`: typing `make` at the command prompt will recompile any of the .c files that have changed. and produce the executables: `cache-test`, `cache-ref`, and `timelife`. The latter is used in the last question. You can type make clean to clean things up.

The file `cache.c` contains the comment `/* TO BE COMPLETED BY THE STUDENT */` whereever you need to supply code. The first two functions where this occurs can be completed using a single line of code. The next three will need from 10 to 20 lines of code each (the exact number may of course vary depending on whether or not you place curly braces on separate lines, how you write the code, etc).

Complete the implementation of the cache in `cache.c`, and then run the program `cache-test` to verify that it works correctly. Here is the expected output:

```
Sum = 4326400
Miss rate =   0.2500
Sum = 4326400
Miss rate =   1.0000
Sum = 4326400
Miss rate =   0.5000
```

[12] 3. This problem tests your ability to predict the cache behaviour of C code. For each of the following caches, array sizes and functions, first determine the miss rate using your cache implementation from the previous question (or the reference solution contained

in the file `cache.o`), and then explain how you would have derived the miss rate given only the program code and the characteristics of the cache (that is, using pencil and paper only). You may need to make modifications to cache-test.c to get the information needed. Assume that we execute the code under the following conditions:

- `sizeof(int)` = 4
- The cache contains 128, sixteen-byte blocks.
- Arrays are stored in row-major order.
- The only memory accesses to be considered are to the entries of the array `a`.

In order to simplify your explanations, you can write them assuming that $a[0][0]$ ends up in cache set 0, even if this isn't actually the case when you run the program. Note that specific cache set numbers may in fact vary depending on things like the compiler or machine used. However the miss rate will remain the same.

Note: if you set the cache policies to `CACHE_TRACEPOLICY`, then the reference solution will output additional information about cache hits, misses, and which set was used in each case. You may find such information helpful. The scenarios you are to predict the cache behaviour for are:

[3] a. An array with 64 rows and 64 columns, a direct-mapped cache, and the function `sumA` from the program `cache-test` provided with question 2.

[3] b. An array with 64 rows and 64 columns, a direct-mapped cache, and the function `sumB` from the program `cache-test` provided with question 2.

[3] c. An array with 64 rows and 64 columns, a direct-mapped cache, and the function `sumC` from the program `cache-test` provided with question 2.

[3] d. An array with 68 rows and 68 columns, a direct-mapped cache, and the function `sumB` from the program `cache-test` provided with question 2.

[10] 4. As mentioned earlier, the code directory contains three files needed to complete this part of the assignment. In this part of the assignment you will be working with the game of life. The game of life consists of an NxN world where at each location a cell might exist. During an iteration of the game of life each location is examined and, based on certain rules, a determination is made as to whether a cell is born at that location, or if one is currently alive there, if it dies or continues to live. The decision about what happens is based on how many living cells are adjacent to the current cell. As a result the basic function is to examine a location and then count all the adjacent living cells.

The file, `timelife.c`, runs and times a *base_life*() function and the *life*() function, which is copy of the *base_life*() function you have to modify. The implementation of the *life* function is in a C source file named `life.c` and is the only file you are to modify.

Your task is to think about cache performance and use this knowledge to improve the performance of the function *life()* in `life.c`. In particular you will want think of things that can help maximize spatial and temporal locality.

You should start by running the unmodified version and then make modifications to try to improve performance. When taking measurements it is important to compare the implementations under as similar conditions as possible. In particular make sure that when comparing measurements you only compare measurements for the same machine. You

should also take several measurements, (say 10) and take their average. (The program provided prints most of this information for you.) Any proposed change must produce the proper output. Since different machines have different hardware configurations it is important to run all the tests on the same machine.

Think carefully about the memory access patterns and then make changes based on that. You are to keep a log in the file README.txt of each change you intend to make and the **rationale** for making that change. Then make the change and measure. As part of the log entry be sure to record an annotation/comment in life.c so that we can see the changes, (carefully follow the instructions in that file), the measurements you took, and their average. (Keep in mind to take multiple measurements, and to take the average. You may not always get a performance increase.) Give each log entry a number starting with 1.

The final part of the README.txt file is to a be table that summarizes the results. Each row is to correspond to a modification you made. However, the first row is to be the result of the unmodified version of *base_life()*. Subsequent rows are to be identified with the number that modification has in the log and the rows are to in the same order as their corresponding log entries. You should have a column for the average measured execution time for the modification and another column indicating the performance of the modified code relative to the unmodified version reported as a ratio of the new time to the original time. For example 0.53 means that the modification used 0.53 times the amount of time that the original did (i.e. it was 1.89 times faster) A number greater than 1 would indicate it was slower. You are not allowed to change compiler options to improve the program's performance. In addition your program must compile without warnings. You are not permitted to suppress warnings through compiler options or directives in the source files.

On remote.ugrad.cs.ubc.ca a modified program has been made that uses under 0.10 of the amount of time as the original (i.e 10 times faster) and we have seen solutions that are over 100 times faster. To get a top grade for the task of optimizing we would expect something approaching 20 times faster.

Your mark for this section will depend upon the changes you propose, the associated **rationale**, and the ultimate success of the changes. (ie. it is okay to make some changes that don't improve performance provided the rationale is sound and that those results are used to inform subsequent changes that actually do improve performance.) When considering changes you will want to consider strategies for improving both spatial and temporal locality. It is highly recommended that when proposing a change you indicate why that change would improve either temporal or spatial locality or both.

Deliverables

As with previous assignments we are using stash to manage the handin of the assignment. The updated files you need to make sure are part of your stash submission are:

1. A README.txt file containing the following information:
   - Your answers to questions 1, 3 and the log and table from 4
2. The updated files for the game of life.
3. The update `cache.c` file

4. The makefile in a working state so that we can compile and run your program as specified.

5. The filled in version of coverpage.txt. Follow the directions in the coverpage.txt file on how to fill-in and submit the file.

When you have pushed your final version of the assignment you need sign-in to stash to verify that you have committed and pushed the versions of the files you want marked and that they are visible in the master branch. You can also clone a new version of your repo just to make sure everything is as you expect. We will only look at the files in the master branch of the repo for marking purposes. If you make a request to mark a branch other than the master branch or pushed things incorrectly and want a different branch marked there will be a 15 percent penalty.