CS 6290: High-Performance Computer Architecture

Project 2

This project is intended to help you understand caches and performance of out-of-order processors. As with previous projects, for this project you will need VirtualBox and our project virtual machine. Just like in previous projects, you will put your answers in the reddish boxes in this Word document, and then submit it in Canvas (but this time the submitted file name should be PRJ2.docx).

In each answer box, you **must first provide your answer to the actual question** (e.g. a number). You can then use square brackets to provide any explanations that the question is not asking for but that you feel might help us grade your answer. E.g. answer 9.7102 may be entered as **9.7102 [Because 9.71+0.0002 is 9.7102].** For questions that are asking "why" and/or "explain", the correct answer is one that concisely states the cause for what the question is describing, and also states what evidence you have for that. Guesswork, even when entirely correct, will only yield up to 50% of the points on such questions.

Additional files to upload are specified in each part of this document. **Do not archive** (zip, rar, or anything else) the files when you submit them – each file should be uploaded separately, with the file name specified in this assignment. You will lose up to 20 points for not following the file submission and naming guidelines, and if any files are missing **you will lose all the points for answers that are in any way related to a missing file** (yes, this means an automatic zero score if the PRJ2.docx file is missing). Furthermore, if it is not VERY clear which submitted file matches which requested file, we will treat the submission as missing that file. The same is true if you submit multiple files that appear to match the same requested file (e.g. several files with the same name). In short, if there is any ambiguity about which submitted file(s) should be used for grading, the grading will be done as if those ambiguous files were not submitted at all.

Most numerical answers should have **at least two decimals** of precision. Speedups should be computed to **at least 4 decimals** of precision, using the number of cycles, not the IPC (the IPC reported by report.pl is rounded to only two decimals). You lose points if you round to fewer decimals than required, or if you truncate digits instead of correctly rounding (e.g. a speedup of 3.141592 rounded to four decimals is 3.1416, not 3.1415).

This project can be done either individually or in groups of two students. If doing this project as a two-student group, you can do the simulations and programming work together, but each student is responsible for his/her own project report, and each student will be graded based solely on what that student submits. Finally, **no collaboration with other students or anyone else is allowed**. If you do have a partner you **have to** provide his/her name here Luis Alberto Guiffarro (enter None here if no partner) and his/her Canvas username here lguifarro3@gatech.edu. Note that this means that you cannot change partners once you begin working on the project, i.e. if you do any work with a partner you cannot "drop" your partner and submit the project as your own (or start working with someone else) because the collaboration you already had with your (original) partner then becomes unauthorized collaboration.

In this project we will be using the FMM benchmark with 256 particles and single-core execution, and we will continue to use the cmp4-noc.conf configuration file. Remember to first **restore the cmp4-noc.conf file to its default contents**. Also, if your branch predictor changes from Project 1 can result in changing the results of the simulation even when using the default (Hybrid) predictor, you need to restore the original code of the simulator. Essentially, you need to undo all the changes made in Project 0 and Project 1. Then you can run the simulation:

```
cd ~/sesc/apps/Splash2/fmm
```

If the fmm.mipseb file is not already present in this directory, then build it:

```
make
```

and run the first simulation we will need for this project **exactly** like this (this should be one line where all '-' characters are the normal "minus" character, the line has a single space between -ofmm.out and -efmm.err, a single space character between fmm.mipseb and -p, a single space between -p and 1, and no spaces, tabs, or anything else after that):

```
~/sesc/sesc.opt -f Default -c ~/sesc/confs/cmp4-noc.conf
-iInput/input.256 -ofmm.out -efmm.err fmm.mipseb -p 1
```

In this command line the -c, -o, and -e simulator options should already be familiar. The -f option tells the simulator to save the report file as **sesc_fmm.mipseb.Default** instead of using a random string at the end. This way you can produce report files with the names you need for your Project 2 submission, without having to rename them.

As with every simulation, you should verify that the simulated execution has not crashed (or terminated too soon) due to misspelling of the command line. After a correct simulation, the fmm.err file should be empty, and fmm.out should begin with "Creating a two cluster, non uniform distribution for 256 particles" and have a "Total time for steps 3 to 5 : 0" line at the end. Having this does not guarantee that **everything** is OK, but it at least means that FMM did not exit prematurely.

Part 1 [35 points]: Performance impact of caches

In this project, we will be modifying the data caches of the simulated processor, so let's take a closer look at the [DMemory] section of the configuration file. It says that the structure the processor gets data from is of type "smpcache" (it's a cache that can work in a multiprocessor, as we will see Project 3), which can store 32KBytes of data (size parameter), is 4-way set associative (assoc parameter), has a 64-byte block/line size (bsize parameter uses cacheLineSize, which is set to 64 earlier in the configuration file), is a write-back cache (writePolicy), uses LRU replacement policy, and has two ports with port occupancy of 1 cycle (so it can handle two accesses every cycle), has a 1-cycle hit time, and takes 1 cycles to detect a miss. If there is a miss, the processor keeps track of it using the DMSHR (data miss handling registers) structure, which is described in the [DMSHR] section as a 64-entry structure where each entry can keep track of a miss to an entire 64-byte block. On a miss, the L1 cache requests data from the core's local slice of the L2 cache, or from the on-chip router that connects it to the L2 slices of other cores. Note that in this project we will still be using only one core (Core 0) so it gets to use the entire L2 cache (all four slices). Looking at the [L2Slice] section, we see that each slice can store 1

megabyte of data (so the total L2 cache size is 4MB), that it is a 16-way set associative cache with a 64-byte block size, write-back policy, LRU replacement, 2 ports, 12-cycle hit time, that it needs 12 cycles to detect a miss, and that it uses a 64-entry MSHR to keep track of misses. When there is a miss, it is handed off to a local on-chip router, which uses the on-chip network (NOC) to deliver the message to a memory controller. It in turn uses the off-chip processor-memory bus to access the main memory, which is modeled in this configuration as an infinite cache with a 200-cycle hit delay. Recall that a real off-chip main memory has a similar delay but has much more complicated behavior, so this simplification is there mostly to avoid specifying the myriad main-memory parameters.

Now let's change some cache parameters and see how they affect performance. Before we make any changes to the cmp4-noc.conf file, we should **save the original** so we can restore the default configuration later. In general, you should **be very careful about ensuring that you have the correct configuration**. The values for one thing (e.g. L1 cache) can affect what happens in other things (e.g. L2 cache), so you should be able to restore the default parameters when needed.

A) Run the fmm benchmark with the default configuration and submit the report from this simulation as **sesc_fmm.mipseb.Default**

B) Change the L1 cache size to 2kB (leave all other conf parameters unchanged, and make sure to save the original cmp4-noc.conf), run that simulation (using `-f SmallL1` this time in the simulation command line), and submit the report as **sesc_fmm.mipseb.SmallL1**

C) With a 2kB L1 cache, the miss rate in the L1 cache is **12.54** percent, and with a 32kB L1 cache the miss rate is **1.03** percent. The overall speedup achieved by replacing a 2KB L1 cache with a 32KB cache is **23250048/15849769= 1.4669013**.

D) In Part 1B we have seen that the *simulated* execution is faster when we change the L1 cache size from 2kB to 32kB. But if we look at the time it takes to do the simulation itself ("Exe Time" from report.pl), we find that *simulation* takes less time, too, although in both simulations the simulated processor executes almost exactly the same number of instructions. You may think that it takes less time to simulate fewer cycles, even when the same overall work gets done over fewer cycles, but actually the simulator takes less time because it **does do less work** when we have a larger L1 cache. ***What is the simulator's work that is eliminated?***

Simulator Perspective (for this specific question)

2KB takes (Exe Speed)105.745 KIPS (kilo instructions per second) while 32KB does 169.382 KIPS. If we look at the number of instructions then they are same, so the end target is same in terms of total no of instructions to be executed for the processor. For 2KB cache there is high miss rate in comparison to 32KB. What is the simulated processor doing in the period when there is miss? The simulator is basically waiting or practically inducing processor clock cycles where it is waiting for missDelay counter to complete. Hence events per processor clock cycles reduce thereby impacting the over all

Exe Time/Exe Speed (KIPS). In 32 KB Cache the data is readily available as miss rate is low. Hence simulator attempts to execute *more events every clock cycle*, thereby improving overall Exe Time.

Processor Perspective (more for understanding)

The processor operates better when cache size is increased from 2KB to 32KB because now the cost of reading from memory is reduced as there are more hits or a lower miss rate. The objective of cache in first place was to reduce the round trip time of getting information from main memory. With cache of 2KB (Smaller) vs 32KB (larger/default), there are capacity restrictions due to which miss rate is higher 12.54 for 2KB vs 1.03 for 32KB. This also means that each time there is miss there is an associated penalty due to misses. In 2KB, there is poorer miss rate in lieu of lesser capacity, thereby resulting in higher penalty when multiplied by miss rate, in contrast to 32KB. Hence 32KB eliminates the work related to cache misses which involves going to L2 Cache then to memory resulting in more cycles spent on overall simulation. L2 Cache has missDelay and hitDelay of 12 which means each time L1Cache has miss, it will result in 12 clock cycles awaiting for memory read (or write) to execute as penalty.

This can also be explained from the equation
L1 Miss Penalty = L2 Hit Time + L2MissRate * L2MissPenalty ….(1)

So L1 AMAT = L1HitTime + L1MissRate * L1MissPenalty (1)….(2)

Since higher hierarchy caches recursively depend on lower ones for computation of AMAT as evident from above 2 equations, hence with 2KB which has high Miss rate results in poorer AMAT in comparison to 32KB.

E) Now let's compare the default 32kB 4-way set-associative L1 cache with one that has the same size but is direct-mapped. You already ran a simulation with the default (set-associative) configuration, so you only need to run a simulation for the direct-mapped L1 cache. Submit the simulation report for this run as **sesc_fmm.mipseb.DMapL1**

F) The miss rate with the direct-mapped L1 cache is **1.61** percent, the miss rate with the 4-way set-associative L1 cache is **1.03** percent, and the overall speedup achieved by changing from direct-mapped (1-way set-associative) to 4-way set-associative L1 cache is **16235997/15849769 =1.024368**

.

G) Now let's restore the default configuration (32kB, 4-way set associative L1 cache) and change the L1 cache latency to 4 cycles (change both hitDelay and missDelay

to 4) and then to 7 cycles. Submit the reports for these simulations as **sesc_fmm.mipseb.4CycL1** and **sesc_fmm.mipseb.7CycL1**.

H) The speedup of improving L1 latency from 7 to 4 cycles is **17819095/16768175=1.06267348**, the speedup of improving the L1 latency from 4 to 1 cycle is **16768175/15849769= 1.0579444**, and the speedup of improving the L1 latency from 7 to 1 cycles is **17819095/15849769=1.1242495**.

I) The change in L1 latency from 7 cycles to 1 cycle represents is a 7X improvement, and the loads and stores represent about 30% of all instructions. So a naïve application of Amdahl's law tells us to expect a speedup of 1.35. Explain **why** the actual speedup is very different from that:

If simply apply the naïve Amdahl law then the overall computation operates on 7 times improvement on 30% Load/Store instructions.
If we classify our overall instruction type into categories – Memory and Non-memory instructions, then following is the breakup

So overall speedup is as follows:-

Overall Speedup = 1 / ((1-X) + X/7) = 7/(7-7X+X)= 7/(7-6X)) …..(1)
= 1 / ((1-0.3) + 0.3/7) = 1.346 ~ 1.35
Here X is all mem operations so accounts for 30%

However this is not true in case of current Load and Store instructions because the 7 times improvement is *only* operated on L1 Caches. There are also L2 Caches that are part of overall all memory operations. In above scenario Memory instructions constitute 30% of improvements over L1 and L2 caches combined not L1 caches alone! So our overall memory access needs to be further broken down as follows for more concrete picture:-

Say of these 30%, X` are L1 Memory access operations, Y` are L2 memory access operations and Z` is Main Memory access operations. So in reality X`+Y`+Z`=30.
Hence naturally we cannot use 30% instruction improvement over improvement on L1 alone as L2($lineFill) and MM both have their individual contribution to overall Load and Store instructions, and that has not changed in current scenario.

So now if we apply Amdahl's law to above equation, the improvement due to L1 Cache, on reducing missDelay and hitDelay to 1 from 7 is as follows:-

Overall SpeedUp(actual) = 7 / (7-6X`) …..(2)
Where X` is actual L1 Cache operations.

Now if we compare (1) and (2) then X > X` as X = 0.3 and X` is less than 0.3, in lieu of load/store operations include L2 Cache operations, which implies following for denominators in equations (1) and (2)

7-6X < 7-6X` as X > X`
This implies that (2) has larger denominator and consequently a lower speedup than (1)

Part 2 [20 points]: Changing the simulated cache

The cache implementation in the simulator can only model LRU replacement policy – note that a RANDOM policy can be specified in the configuration file but the code that models the replacement policy will still implement LRU even when RANDOM is specified. Now we will explore what happens when we actually change the cache's replacement policy. We will implement the NXLRU (Next to Least Recently Used). While LRU replaces the block that is the first in LRU order (i.e. the least recently used block) in the cache set, NXLRU should replace the block that is the second in LRU order in the set, i.e. the second-least-recently-used block in the set.

To implement NXLRU, we need to modify the code of the simulator. The source file which implements the 'smpcache' (used for our L1 cache) is in SMPCache.h and SMCache.cpp in the sesc/src/libcmp/ directory. For much of the "basic" cache behavior, the SMPCache uses code in sesc/src/libsuc/CacheCore.h (and CacheCore.cpp). There are separate classes for CacheDM (for direct mapped caches) and CacheAssoc (for set-associative caches). Since direct-mapped caches do not have a replacement policy (they must replace the one line where the new block must go), we will be looking at the CacheAssoc class. First we must add "NXLRU" as an option that can be specified in the conf file and selected when a CacheAssoc object is constructed. Probably a good approach is to look for "LRU" in the code to see how this is done for LRU (and RANDOM), and then add NXLRU. Then we must actually implement this policy. The function that actually implements the cache's replacement policy is the findLine2Replace method of the CacheAssoc class in CacheCore.cpp. The parameter supplied to this method is the new address that needs a line in the cache. Note that this method does not only implement the **replacement** policy because an actual replacement (replace one valid line with another) may not be needed. For example, when addr is already in the cache (a cache hit), this method returns the line that contains addr. When the set where addr belongs contains non-valid lines, one of those non-valid lines is used – a valid block may have a cache hit in the future, while a non-valid line cannot, so we should only replace a valid line if the set has no non-valid lines. Finally, when the set contains "locked" lines they are skipped, so the actual "LRU" policy implemented by findLine2Replace is "From the set where addr belongs, return the line that contains addr if there is such a line, otherwise return the invalid line that was accessed most recently if there are any invalid lines, otherwise return the least recently used line among

the lines that are not locked". Even that is not the complete specification because findLine2Replace must consider what should happen when all lines are valid and locked – in that case it returns 0 unless ignoreLocked is true, in which case it returns the least recently used line chosen among all the (valid and locked) lines in the set.

Our NXLRU policy should treat hits and invalid lines just like the existing LRU policy, but when there is no hit and no invalid lines to return, the NXLRU policy should find the second-least-recently-used line among the non-locked lines. However, if only one non-locked line exists in the set, that line must be returned, and if all lines are valid and locked the second-least-recently-used one in the set should be returned.

Note that you **have to** add the NXLRU policy as an option in the configuration file, i.e. it is not OK to just change the existing LRU (or RANDOM) code to actually follow the NXLRU policy. Changing the behavior of existing policies will change the behavior of all cache-like structures in the processor, including TLBs. We will want to change the replacement policy only in L1 caches and leave behavior of TLBs, L2 caches, etc. unchanged!

Make the changes needed to implement the NXLRU replacement policy and then:

J) Run a simulation with a 2kB L1 cache, using NXLRU policy, and with all other settings at their default values. Submit the simulation report for this as **sesc_fmm.mipseb.L1NXLRU**. You already ran a simulation with the same L1 cache that uses LRU (in Part 1A). With a 2kB L1 cache, the LRU policy gave us a hit rate of 100-12.54=87.46 percent, while NXLRU gives us 100-15.85=84.15 percent. The number of blocks that are fetched (read) by the L1 cache from the L2 cache changes from 408433 with LRU to 532898 with NXLRU, and the speedup of using LRU instead of NXLRU is 24751952/23250048 =1.0645978.

Note: Because report.pl does not provide summary statistics on the L2 cache, you will have to directly examine the report file generated by SESC. This file begins with a copy of the configuration that was used, then reports how many events of each kind were observed in each part of the processor. Events in the DL1 cache of processor zero (the one running the application) are reported in lines that start with "P(0)_DL1:". Events in the L2 cache are reported separately for each of the four slices. In the report file, the number of blocks requested by the L1 cache from the L2 cache is reported as lineFill (these become entire-block reads from the L2 cache), and the number of write-backs the L1 wants to do to the L2 is reported as writeBack (these become entire-block writes to the L2 cache).

Part 3 [45 points]: Classifying misses in the L1 cache

Now we will change the simulator to identify what kind of L1 cache miss we are having each time there is a miss – compulsory, conflict, or capacity. Recall that a miss is a

compulsory miss if it would occur in an infinite-sized cache, i.e. if the block has never been in the cache before. A capacity miss is a non-compulsory miss that would occur even in a fully associative LRU cache that has the same block size and overall capacity, and a conflict miss is a miss that is neither a compulsory nor a capacity miss. The L1 cache in the simulator counts read and write misses in separate counters (which appear in the simulation report as readMiss and writeMiss number for each cache, e.g. there is line in the report for "P(0)_DL1:readMiss=something" in the report file. Now you need to have additional counters, which should appear in the simulation report file as **compMiss**, **capMiss**, and **confMiss** counters (these three values should add up to the readMiss+writeMiss value). Each of the new counters should count both read and write misses of that kind. It is OK to **also** have counters that count compulsory, capacity, and conflict misses separately for reads and writes, or to do this classification of misses for other caches. But report files that do not have the overall (reads+writes) items for P(0)_DL1:compMiss, P(0)_DL1:capMiss, and P(0)_DL1:comfMiss will not be graded.

K) Attach the **CacheCore.h**, **CacheCore.cpp**, **SMPCache.cpp**, and **SMPCache.h** files that contain your modifications for both Part 2 (NXLRU) and Part 3 (classification of misses). You should not modify (or add/delete) any other files.

L) With your new miss-classification code in the simulator, you should run a simulation with the default configuration (32kB 4-way set-associative LRU L1, cache), with an 2kB 4-way set-associative LRU L1 cache, with a direct-mapped 32kB L1 cache, and with a 32kB 4-way set-associative NXLRU L1 cache. Submit the four simulation reports as **sesc_fmm.mipseb.DefLRU**, **sesc_fmm.mipseb.SmallLRU**, **sesc_fmm.mispeb.DefDM**, and **sesc_fmm.mispeb.DefNXLRU**.

M) Fill the following table. The first row of fill-in fields is the total number of L1 misses for each configuration, the second is the percentage of all L1 misses for that configuration that are compulsory misses, the third is the percentage of L1 misses that are conflict misses, and the fourth is the percentage of L1 misses that are capacity misses. For example, if a simulation ended up with a total of 1000 L1 misses, and if they include 300 compulsory, 500 conflict, and 200 capacity misses, then the column for that configuration would have the following numbers (from top to bottom): 1000, 30,50,20.

	32kB SA LRU	**2kB SA LRU**	**32kB DM**	**32kB SA NXLRU**
Total # of misses	29722	408433	46840	40128
% Compulsory	11.94065002	0.868930767	7.576857387	8.844198565

% Conflict	15.31189018	25.18919872	52.61528608	31.03070175
% Capacity	72.74745979	73.94187051	39.80785653	60.12509968

N) Now we will consider the 32kB 4-way set-associative LRU cache as the baseline and compute, for each of the other three configurations, the percentage increase of various kinds of misses relative to that baseline. For example, if the 32kB 4-way SA LRU configuration has resulted in 200 compulsory misses and the 1kB 4-way SA LRU configuration has resulted in 190 compulsory misses, then we should put -5 in the compulsory-misses entry in the 2kB SA LRU column: the change is -10 compulsory misses (190-200), and -10 is 5% of the baseline's 200 misses (-10/200 = -0.05).

% Change in	**2kB SA LRU**	**32kB DM**	**32kB SA NXLRU**
Total # of Misses	1274.177377	57.59370164	35.01110289
# of Compulsory Misses	0	0	0
# of Conflict Misses	2160.624039	441.5293342	173.6101956
# of Capacity Misses	1296.739432	-13.76375913	11.58542226

Some helpful notes for Part 3:

In the simulator, there are two kinds of misses – normal misses and "half-misses". A half-miss occurs when the processors loads/stores a data item, the corresponding block is not present in the cache, but a cache miss is already in progress for that block. Note that a half-miss would not occur if the cache was handling one access at a time – the block would be brought to the cache as a result of a "normal" miss, and the access that was a half-miss would be a hit. To avoid confusion, in Part 3 of this project, you should only be concerned with the "normal" misses, i.e. you should treat half-misses as cache hits.

Another potentially confusing consideration is that it is possible to have an access that is a hit in a set-associative cache but it a miss in the fully associative cache that we are modeling to determine if a cache miss is a conflict miss. This can occur, for example, when a cache block B is not accessed for a long time, but the other blocks that map to the same cache set are also not accessed for a long time. In a fully-associative cache, block B would eventually be replaced because we need room in the cache for other blocks, but in the set-associative cache we need room in other sets so block B stays in the cache. When B is eventually

accessed, we get a hit in the set-associative cache and a miss in the fully-associative cache. For Part 3 of this project, you can simply ignore this problem – your task is to only look at misses that do occur in the actual cache and determine whether these misses would be hits in infinite-size and/or fully-associative caches, so accesses that are hits in the set-associative cache can be ignored.