CS 6290: High-Performance Computer Architecture

Project 3

This project is intended to help you understand cache coherence and performance of multi-core 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 PRJ3.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 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, except when we explicitly ask you to submit a zip file (in Part 3H). Each file we are asking for should be uploaded separately, and its name should be as specified in this assignment. You will lose up to 20 points for not following the file submission and naming guidelines. 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 as 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 Maulik Shah (enter None here if no partner) and his/her Canvas username here mshah357. Note that this means that you 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.

Part 1 [40 points]: Running a parallel application

In this part of Project 3 we will be using the LU benchmark. We will also be using a processor with more (sixteen) cores (cmp16-noc.conf). So, for example, to simulate 4-threaded execution you would use a command like this (note the absence of spaces between -n and 512, and between -p and 4):

~/sesc/sesc.opt -fAp4 -c ~/sesc/confs/cmp16-noc.conf -olu.out -elu.err lu.mipseb -n512 -p4

To complete this part of the project, run the lu application with 1, 4, and 16 threads. Then fill in the blanks, taking into account all the runs you were asked to do:

A) Submit the three simulation reports: **sesc_lu.mipseb.Ap1**, **sesc_lu.mipseb.Ap4**, and **sesc_lu.mipseb.Ap16**. You will not earn points for submitting these simulation reports, but you will lose 10 points for each missing simulation report.

B) Fill out the execution time, parallel speedup, and parallel efficiency with 4 and 16 threads. Enter Sim Time with precision of at least three decimals, and speedup and efficiency with precision of at least two decimals.

	SimTime (in ms)	Parallel Speedup	Parallel Efficiency
-p1	482.034 ms	1	1
-p4	161.167 ms	482.034/161.167=2.99089	2.99089 /4=0.7477225
-p16	81.842 ms	482.034/81.842=5.889812	5.889812/16=0.36811325

Note: Parallel speedup is the speedup of parallel execution over the single-thread execution with the same input size. Parallel efficiency is the parallel speedup divided by the number of threads used – ideally, the speedup would be equal to the number of threads used, so the efficiency would be 1. When computing the speedup and efficiency we cannot use IPC or Cycles that are reported for each processor, because these do not account for the cycles where that core was idle (e.g. because the thread was waiting for something to happen). So we need to use the "Sim Time" we get from report.pl because it accounts for all cycles that elapse between the start and completion of the entire benchmark.

C) Our results indicate that parallel efficiency changes as we use more cores. Why do you think this is happening?

When we go from p1, p4 to p16 we are parallelizing our work in better way to achieve a higher throughput thereby better speedup. We are not only using more cores (16) but we are using thread level parallelism to use the additional cores in most optimal way based on the LU benchmark code. Ideally the work

distribution should be such that overall work should be equally distributed. So for 16 core with 4 threads (only 4 cores are used) the overall time for execution should be p1Time/4 =482.034/4=120.508. In such case parallel efficiency would have been 1. However, what we see for core 0 is 161.167. Consequently, this impacts the parallel efficiency for Core 0. Why does Core 0 take more time? This is because LU benchmark instructions were not distributed equally or Core 0 does some extra work due to increase no of threads. Infact Core 0 executes a lot more instructions than Core1,2,3….

	nInst
0	179435381
1	125623386
2	131553434
3	131477788

This is possibly due to 2 reasons:-

1. **Code which is primarily serial portion has been assigned to Core 0. We cannot parallelize serial portion of the program**
2. **Core 0 thread is possibly also synchronizing results from different cores and collating it together. Such synchronization is extra overhead.**

This all results in parallel efficiency to not scale in proportion to number of threads.

Another aspect to note is that if number of threads goes to infinity then overall parallel efficiency tends to 0. This is case where we can parallelize every instruction using infinite threads.

D) When we use four threads (-p4) instead of one (-p1), the IPC achieved by Core 0 (the first processor listed) got slightly lower because

IPC drops for first core in p4 (IPC = 1.15) run vs p1(IPC1.18) run. Also, another observation is Miss Rate on core 0 in P4 run is 0.54 vs p1 run (0.52). Additionally, we see that Load/Store instructions executed on core 0 in p4 run (Load: 18.15% Store: 8.61%) is more than p1 run (Load: 18.10% Store: 8.56%) or any other core in p4 run.
All above stats point to fact that p4 runs core 0 does do extra work in comparison to other cores in p4 run. Why is that? There are 2 possible reasons for that.

1. **The first one is synchronization. There is extra effort on core 0 possibly in lieu of LU matrix operations distributions where matrix operations results are merged on core 0. Hence in order to achieve this it ends up locking and unlocking certain parts of implementation. Such memory operations (lock mutex, unlock mutex) mean more memory operations and consequently higher miss rates and eventually more memory latency resulting lower IPC for core 0. In multicore we can partition our work, but we also have overhead on coordination and synchronization, and communication overhead between cores. The thread level parallelism is also not able to leverage full potential of CPU as it many times waiting due to memory latency due to locking/unlocking code. Hence lower than expected IPC.**
2. **Second possible reason could simply be cache line invalidations are higher in core 0 in lieu of more load and store instructions. Hence higher coherence misses which are costly in terms of memory delays, and resulting lower than expected IPC**

E) Now look at the simulation reports for these simulations. Core 0 executes more than its fair share of all instructions because

In case p4, p16 runs Core0 shows more instructions executed than other cores. Additionally, as pointed out the Load/Store instructions show higher percentage of execution on Core 0. This ties into 2 possible reasons:-
1. **This ties into possible synchronization code and possibly LU benchmark code is written in such a way that we have locking and unlocking mutex for synchronizing results from multiple cores. This are extra memory operations/instructions thereby increasing overall instruction for this core in comparison to other cores in same run.**
2. **Additionally, this could simply be because of all serial portion of code being assigned to Core 0, which would mean extra work for Core 0. Hence more instructions executed by Core 0.**

Part 2 [10 points]: Cache miss behavior

In this part of Project 3, we will be focusing on the number of read misses in the DL1 (Data L1) cache of Core 0, using the same simulations that we already did for Part 1. In the report file generated by the simulator (sesc_lu.mipseb.something, not what you get from

report.pl), the number of cache read misses that occur in each DL1 cache (one per processor core) is reported in lines that begin with "P(0)_DL1:readMiss=".

F) The total number of read misses that occur in the DL1 cache of Core 0 is

Simulation	-p1	-p4	-p16
Core 0's DL1 read misses	760977	220878	78308

Your answers here should be integer numbers.

G) The number of these misses changes this way as we go from one to two to four, etc. threads because

When we go from 1 to 4 to 16 threads, we observe that there is downward trend in number of misses. The thread level parallelism attempts to reap performance benefit by finding independent work across multiple threads. Each of the threads are attempting to operate on independent code components of instructions, which means these instructions can be run parallelly and would not interfere with working of each other. This will result in better usage of cache's on each of the cores as threads may get less replacement or capacity misses. Why is that? When we have multiple cores, we are not only distributing the work across multiple cores but also make-believing that there is one cache via cache coherence protocols. Hence at one end we will have higher coherence misses as we go from p1 to p4 to p16 but, at the same time, we will achieve better spatial or temporal locality thereby lower capacity or replacement misses. Another difference between p1 to p4 is the cache size. In p1 only one core is used with 512 cache lines. However, in p4 we have 4 active cores, and since cache coherence protocol, attempts to make believe all caches as one cache hence mathematically it is much larger Cache (cache 0 : 512 Cache lines, cache 1= : 512 Cache lines …..) which behaves as one. This is another reason with apt spatial and temporal locality per core we will see drop a in capacity or replacement misses from p16 to p4 to p1.

Part 3 [50 points]: Identifying accesses to shared data

You task in this part of the project is to determine how many read misses in each core's DL1 cache are compulsory (readCompMiss), replacement (capacity or conflict, the counter should be called readReplMiss), and coherence misses (readCoheMiss), and separately also classify write misses (writeCompMiss, writeReplMiss, and writeCoheMiss). Note that this classification is similar to the one you did for Project 2, except that you now we are counting different categories of misses separately for reads (load instructions) and writes (store instructions), that we are placing conflict and capacity misses in the same (replacement) category, and that we are adding a category for coherence misses that we didn't have in Project 2. To simplify classification, we will **not** follow the exact definition of coherence misses ("those misses that would have been hits were it not for coherence

actions from other cores"). Instead, we will use a definition that allows much simpler implementation: a coherence miss is a miss that finds in the cache a line whose tag matches the block it wants, but that block has a coherence state that prevents such access. In the case of read misses, this means that the line was found in an "Invalid" coherence state. Note that this identification of coherence misses may not be trivial in the SESC simulator because of the way it handles tags during invalidation. If a miss is not a coherence miss, then you can classify it as either compulsory or replacement miss by checking if the block was ever in that cache. When checking whether the miss is a compulsory miss, be careful to track the "was previously in this cache" set of blocks for each cache separately.

H) Create a Changed.zip file with any simulator source code files that you have modified in Part 3 of the project, and submit this **Changed.zip** file together with your project. You will not earn points for submitting this file, but you will lose 50 points if it is missing or if it does not contain all the source code modifications. Then, with your changed simulator (that now counts compulsory, replacement, and coherence read misses), re-run the simulations from Part 1 and submit the resulting simulation report files as **sesc_lu.mipseb.Hp1**, **sesc_lu.mipseb.Hp4**, and **sesc_lu.mipseb.Hp16**. As in Part 1, you will not earn points for these submitted simulation reports, but you will lose 10 points for each simulation that is missing.

I) The number of all read misses, compulsory read misses, replacement read misses, and coherence read misses for the DL1 cache of Core 0 is:

	Core 0's DL1 readMiss	Core 0's DL1 compMiss	Core 0's DL1 replMiss	Core 0's DL1 coheMiss
-p1	760977	122	760855	0
-p4	220878	126	220629	123
-p16	78308	134	77861	313

Note: readMiss numbers here should be the same as those you had in Part 2.

J) The number of all write misses, compulsory write misses, replacement write misses, and coherence write misses for the DL1 cache of Core 0 is:

	Core 0's DL1 writeMiss	Core 0's DL1 compMiss	Core 0's DL1 replMiss	Core 0's DL1 coheMiss
-p1	33056	32981	75	0
-p4	33319	33018	80	221
-p16	33672	33156	78	438