CS 250 Spring 2017 Homework 09 SOLUTION
GTAs: Please enter scores in Blackboard by April 15, 2017

Maximum score = 15

1. **[3 pts.]** If a cache is big enough for the 90/10 Rule to apply, and the cache is 10 times faster than main memory, what does Amdahl's Law predict for the overall speedup from using this cache?
   Answer: The 90/10 rule says 90% of the time the program will be executing in a way that exhibits locality, which, given that this cache is "big enough" for the rule to apply, means we can expect to benefit from this cache 90% of the time. Therefore,
   speedup = 1 / ((1 – 0.9) + (0.9/10)) = 1 / 0.19 = 5.3 times.
   Note: Typically, we will not know if a cache is "big enough" without performing simulation of the cache operating on the instruction execution trace of a program or an entire software workload of interest.

2. **[3 pts.]** How many direct-mapped cache organizations are possible given 32-bit byte addressing for programmers, word-addressed physical memory, 4-byte words, and a 17-bit tag?
   Answer: There are 14 organizations possible.
   Direct-mapped means that the cache has one place to map a given block. With 32-bit byte addresses and 17-bit tags, there are 32-17 = 15 bits to allocate between the two other fields in the address, the index and offset fields. Word addressing of physical memory means that the minimum cache block size is one word, so the minimum byte offset field must chose among the four bytes of a single word, meaning the minimum is 2 bits. So one extreme allocation of the index and offset bit fields is 13-bit index combined with 2-bit offset. The other extreme is 0-bit index (just one cache block that everything maps into) and 15-bit offset. Thus, there are a total of 14 organizations, index bits $\in$ {13, 12, …, 1, 0}.
   The extreme design choices are the following. First, a 0-bit index field and 15-bit offset field yielding a single-block cache design holding 2^{15} = 32,768 bytes = 8,192 32-bit words that requires a $2^{15}x2^{3}$-bit interface to main memory. That interface requires 262,144 wires! No way in today's technology. Second, and likely not of best performance due to lack of parallelism in accessing memory, a 13-bit index for 8,192 cache blocks each holding but a single word. Typically, simulation will show that designs at the extremes of a design space perform more poorly than designs somewhat away from the extreme, but not always. Experiments via simulation are quite valuable.

3. The i-cache (instruction cache) for a 32-bit RISC processor is direct-mapped with an offset field of 4 bits. Assume the cache is cold. Soon, the processor will fetch its first instruction at address 0x00400000.
   a. What is the best assumption about the size(s) of instructions for this processor? Why is it almost certain that the processor is designed this way?
      Answer: Instructions have a fixed size of 32 bits. This design allows for a simple, fast circuit to compute the default next instruction. Fetching instructions is among the most common activities of a processor (executing instructions is the other), so urged on by Amdahl's Law, making fetching fast will be a priority for knowledgeable

processor designers.

b. **[1 pt.]** How many 32-bit instructions can a block in this i-cache (instruction cache) hold?
Answer: Given the 4-bit offset, we know that the cache block is 16 bytes in size. Therefore, each block holds 4 instructions.

c. **[2 pts., 1 pt. for Miss, Hit, Hit, Hit pattern and 1pt. for Miss is of type Compulsory or, equivalently, of type First Reference or of type Cold Cache]** Assume that the program loaded into memory starting at address 0x00400000 is pure straight line code. This means that this program is one, giant basic block of code, who knows how long, that fits within the available memory. Describe the hit and miss behavior of this i-cache (instruction cache) as the processor fetches the first four instructions of the program. State the type(s) of miss(es) that occur. Carefully consider how the address of the first instruction affects the number of hits and misses.
Answer: One compulsory miss (also called a cold-start miss or a first-reference miss) followed by three hits.

The first instruction fetch at 0x00400000 will be a compulsory miss because the very first time a block is accessed it cannot be in the cache. The address of the first fetch, 0x00400000, has a 4-bit offset field containing the bit string 0000. This means that the 16-byte block starting at 0x00400000 and ending at 0x0040000F that will be copied into the cache from the lower memory level will start with the first byte of the first instruction of the program and end with the last byte of the fourth instruction of this straight line code. This means that the second fetch at address 0x00400004 must hit in the i-cache, as will the third and fourth instruction fetches at addresses 0x00400008 and 0x0040000C, respectively.

d. **[1 pt.]** Now assume that the program was loaded into memory starting at address 0x00400008. What is the hit/miss behavior and miss type(s) for the first four instructions? What happens afterwards?
Answer: Two cold-cache misses and two hits for the first four instructions, followed by one cold-cache miss and three hits for each successive four instructions.

The first instruction fetch is again a cold miss. However, the block brought in to the cache now has the first instruction starting less favorably at offset = 8 instead of offset = 0. This means that the first instruction occupies bytes 8. 9, 10, and 11 of the block, so the second instruction in the program occupies bytes 12, 13, 14, and 15. The second instruction fetch will be a hit, but will also be the last instruction to be found in this cache block as the processor fetches in a straight line. Note that the first 8 bytes of this block do not contain instructions of this program; they are a waste of space in the i-cache.

The third instruction fetch is at address 0x00400010 (0x00400008 + 8 = 0x00400010) and causes a second cold-cache miss. The fourth program instruction is contained in this block and its fetch will be a hit.

After the faux pas of starting the program at address 0x00400008, which prevented getting three hits after the first cold-cache miss, the program will have steady state behavior the same as when the program started at address 0x00400000. Aligning

programs on cache block boundaries costs nothing and yields a cute little benefit in performance, so knowledgeable OS designers should always choose executable starting addresses so as to obtain this small gift.

e. **[1 pt.]** Expressed as a percentage, what is the overall hit rate of the straight line program? Remember, the program is many instructions long.
Answer: Straight line code will miss on the first instruction in each block and then hit on the remaining three instructions for every block. For a long program of many instructions the special case of the final cache block not necessarily being entirely full of 4 instructions, will not significantly affect the overall (average) hit rate. Thus, the hit rate is 75%.

f. **[2 pts.]** Now the program contains more than just straight line code. It has been re-written to have straight line code plus the occasional if-then-else statement. The program looks like this around an if-then-else statement (based Figure 9.2 in our textbook).

High level program	Assembly code			Basic Block
statement ;		instrs. for statement	; straight line (SL) instructions	1
if (cond.) {		instr. for condition	; 1 instruction evaluates condition	1
		branch to else	; EXIT the SL basic block	1
then_part		instrs. for then_part	; section of SL, a basic block	2
		…	; remaining then_part SL instrs.	2
} else {		branch to done	; EXIT the SL "then" basic block	2
else_part	else:	instrs. for else_part	; TARGET, else_part basic block	3
}		…	; remaining else_part SL instrs.	3
next_stmt ;	done:	instrs. for next_stmt	; TARGET, fall-through from else_part continues SL	3

Assume that the address of any given assembly instruction (machine instruction) in the assembly code is equally likely to correspond to any one of the 4 possible word offsets (w_offset) within an i-cache block. This means that the execution trace of the program can exit straight line code when executing any instruction word in a cache block.

Just for cache blocks containing a unconditional branch instruction, what is the average hit rate expressed as a percentage?

For simplicity in analyzing the situation for this question, assume (1) that any cache block contains at most one branch instruction (just one chance for the program execution trace to exit the block early) and (2) if a cache block contains a branch instruction, then it does not also contain a target instruction (a way back for the program execution trace to get back into the block after exiting because of the branch).

Answer: Average hit rate for i-cache blocks containing the unconditional branch to "done:" is 60%.

Analysis of the unconditional branch to "done:".

With the simplifying assumptions, a cache block containing an unconditional branch can exit the block early, that is, exit before fetching the instruction at the third word offset within the block, at word offsets of 0, 1, and 2, but the branch target will be outside of the block containing the unconditional branch. Exiting at word offset 3 is the same as exiting a cache block because of executing straight line code. The hit rate analysis for code containing the unconditional branch, then, has 3 new cases and one old case. They are:

Branch instruction at word offset (w_offset) = 0 yields one cold-cache (compulsory) miss for the block and no follow-on hits because the branch causes the program execution trace to exit this block, and by the simplifying assumptions above, the trace cannot return to this block. Hit rate = 0 of 1 cache access = 0%.

Branch at w_offset = 1 yields 1 miss, 1 hit, then exit. Hit rate = 1 of 2 = 50%.

Branch at w_offset = 2 gives 1 miss, 2 hits, then exit. Hit rate = 2 of 3 = 66%.

Branch at w_offset = 3 gives 1 miss and 3 hits. Hit rate = 3 of 4 = 75%.

The average hit rate for blocks with branches is 1+2+3 = 6 total hits among 1+2+3+4 = 10 total accesses from the collection of 4 equally probable cases, thus, average hit rate = 6/10 = 60%.

g. **[2 pts.]** In part (f), the inclusion of if-then-else in the original program, which was only straight line code, worsened the hit rate for the i-cache. Now imagine adding loops to the program of part (f). Will the hit rate further worsen, stay the same, or improve? Using the program instruction execution trace concept, explain the reasoning behind your prediction of the effect of loops on hit rate.
Answer: Loops means that the instruction execution trace will repeat some instructions in the i-cache. If the cache block holding these instructions is still in the cache when subsequent execution of a given instruction occurs, then there will be no miss when accessing that instruction. Because Hit_rate = 1 – Miss_rate = 1 – (Number of instruction accesses that are misses) / (Number of instruction accesses) a reduction in the number of instruction accesses that are misses must increase Hit_rate. The boundary condition for this scenario is when the cache is never holding an instruction when the loop needs it again (possible, but unlikely in practice). For this boundary case no cache misses will be avoided, hence, the Hit_rate will be unchanged due to the addition of loop program structures.

h. The i-cache of this question has been designed with 4-word blocks. Think of i-cache operation in the event of a cache miss as an opportunity to perform instruction execution trace prediction. As compared to an i-cache design alternative where the block holds only one word, a 4-word-block design embodies its designer's vote for which type of execution trace prediction: no prediction, predict default_next, predict not default_next? Pick your answer and then explain why it is the correct choice.
Answer: Predict default_next. Default-next here refers to the default_next_instruction. For straight line code default_next is always chosen for the next instruction. By bringing 4 consecutive instruction words into the cache each time there is a miss, instead of bringing just 1 word, the cache designers are anticipating program locality that would soon access the instructions immediately

following the instruction that just now caused a cache miss.

A cache design where the block holds only one instruction word is a design that has no bias, no prediction, as to which instruction in the program will be the next instruction in the execution trace. We could say that a one-word-per-block i-cache design is one that makes no prediction about the path of instruction execution.

Implementing predict not default_next is difficult to imagine. First, a fetch address more specific than "any instruction in the memory space other than the default_next instruction" would be needed. Given such, what would come after that? My imagination is insufficient to this task.

i. Cache misses can be categorized into three groups: compulsory miss, capacity miss, and conflict miss. In question part (g) program loops were introduced. Which miss type(s) can occur in the program with loops that are not possible for the program with only straight line code and if-then-else constructs?
Answer: Capacity misses and conflict misses can happen when loops are added to the program.

If-then-else constructs are straight line code, just with a skip (either the then or the else part is skipped), and straight line code just keeps on fetching. In either case, the only miss type that occurs is compulsory. This is because each new instruction fetch is at a new, higher memory address than any fetch before, so every time a miss occurs it is because the program execution trace is making its first ever reference to a block.

When a program has a loop, then it can miss in two new ways. First, a miss can occur because the cache is not large enough to hold all the blocks referenced so far by the entire program, so that a loop-caused second reference to a block could miss (capacity miss). If the cache did have sufficient capacity, then once a block was brought into the cache, it could reside there until the program ended, never generating another miss.

Second, a miss due to looping can occur because two instructions of a (large) loop that are in two distinct blocks of memory are, nonetheless mapped to the same cache block (conflict miss). Before, with only straight line or if-then-else code, a conflict miss between two blocks would be called a compulsory miss because the "conflict" only happens once and only when the second block is being referenced for the first time. The first reference miss would happen no matter how large the cache, so that is the name chosen to type that miss.

4. A faster replacement algorithm for an associate cache improves which term or factor of the average memory access time equation?
Answer: Replacement time is a component of the total time of the Miss_penalty factor in the Miss_rate x Miss_penalty term of the equation. The speed of the page replacement algorithm has no effect on the Miss-rate factor, only the quality of the algorithm affects the miss rate. Higher quality means improving the choice of page to replace. Further, the speed of the replacement algorithm does not affect the Hit_time term in the equation, because the replacement algorithm is not invoked in the event of a hit.