# CS/ECE 752
# Fall 2024
# Homework 7 SOLUTION
# Due 11 AM Central Time on Saturday, November 9th, 2024

## NAME: _____

You should do this assignment on your own (i.e., including separate from Chegg, CourseHero, ChatGPT, Gemmini, or any other similar tools), although you are encouraged to talk with classmates electronically or on Piazza about any issues you may have encountered. The standard late assignment policy applies: you may submit up to 24 hours late without penalty.

*If at all possible, please retain the format of the provided PDF/Word document. I am going to grade this homework using Gradescope, which works best when answers are in set places.*

## What to Hand In

To submit your assignment, please type up your answers to the following questions and submit **one PDF named HW7-<netID>.pdf** on Canvas. If you prefer writing your answers by hand, that is fine, but please scan your solutions and submit them on Canvas. **Please make sure to put your name in the top-right of the first page of the PDF. And please have the answer to each question start on a new page to simplify grading (like prior homeworks).** Note that just because each question starts on a new page, it does not mean you need to write a whole page of answer – the page breaks are there for simplifying grading.

## Total Points: 55

## *Problem 1 [10 points]*

**Part A [5 points]**

In class we discussed how perfect/infinite hardware does a much better job of exploiting ILP than "real" hardware. Give two reasons why this is the case. To receive credit, you much justify your answer.

**Solution**:

(there are multiple answers, each will be graded on how relevant the answers are)

Since real hardware cannot have infinitely wide/large hardware units, it is unable to exploit as much ILP as ideal hardware. Two of the main culprits are branch prediction and memory latency.

Memory latency is a problem because real systems cannot always return memory accesses in a single cycle due to L1 cache misses. If the request also misses in subsequent levels of the memory hierarchy, this penalty only grows. If the TLB misses (and any subsequent levels of the TLB miss), this further adds to the delay in returning a memory request. Given that this can take hundreds of cycles (ignoring requests

that go to disk) in modern systems, this means that we need many independent instructions just to cover up a single memory request. This is challenging because we often have dependencies on these accesses, e.g., we often have RAW dependencies on memory loads. When the memory accesses take 1 cycle, this isn't a huge overhead. But when they take many cycles, it is a big overhead because those instructions can't be executed either (unless we do value prediction or some similar techniques). Add in the fact that memory accesses are 20-30% of all dynamic instructions and this overall leads to a large inability to exploit ILP.

Branch prediction also causes problems. Although modern branch predictors have > 99% accuracy, because modern processors have deep pipelines and because branches makeup ~20% of dynamic instructions, a miss will have a large penalty because we need to flush all of the instructions we fetched after the mispredicted branch. This means that, again, it is difficult for us to reach the theoretical limits of ILP because of mispredicted branches.

Other issues include window size and issue width. These are intimately related with the above issues. For example, since other components such as the ROB are of limited size (usually < 300 entries in modern processors), it is difficult to find enough independent instructions that are executing concurrently to hide cache misses.

**Part B [5 points]**

Why is hardware usually able to exploit more ILP in floating-point programs than integer programs? To receive credit, you must justify your answer.

**Solution**:

Floating-point programs usually have lots of arithmetic instructions that are independent from one another and can more easily be overlapped (and also used to hide memory accesses, although in many cases the arithmetic is dependent on the memory accesses – unrolling may help a lot here too) compared to some integer programs. Additionally, and perhaps most importantly, floating-point programs (in part because they are computation-heavy) have fewer branches than integer programs. This means that mispredicted branches occur less frequently.

## *Problem 2 [20 points]*

**Part A [12 points]**

You were recently hired at a big computer architecture firm, MBI! As your first task at your new job, you are assigned to optimize code where the following loop is the most important piece of the program:

```
for (i = 0; i < 127; ++i)
     for (j = 0; j < 31; ++j)
          A[i][j] = B[i] * C[j];
```

To improve performance, you decide to add software prefetching support to prefetch the references you expect to miss in the cache. Assume that a read prefetch is expressed as *PREFETCH(&variable)* and it fetches the entire cache line in which the variable resides in shared mode. A read-exclusive prefetch operation, which is expressed as *PREFETCH_EX(&variable)*, fetches the line in exclusive mode (i.e., so you can write it). Moreover, make the following assumptions about your system:

- Cache block size: 16 B.
- The cache is large enough to avoid any conflict or capacity misses.
- Each element of arrays A, B, and C is 4 B.
- Each iteration takes 16 cycles.
- Cache miss latency is 64 cycles; cache hit latency is 1 cycle.
- Size of array A is 128x32, B is 128, and C is 32.
- A[i][j] and A[i][j+1] are contiguous in memory, and so are A[i][31] and A[i+1][0].
- If necessary, you may assume A, B, and C are allocated such that they will never be on the same cache line (i.e., you may assume A[127][31] and B[0] are on different cache lines).
- You may assume the cost of adding prefetch instructions is minimal (i.e., each iteration still takes 16 cycles).

You may also consider using loop unrolling if you think it would be beneficial.

Rewrite the code to include as many prefetch operations as you believe are necessary to hide cache misses. Beneath the updated code, or inline with the code using comments, explain why you are adding each prefetch (i.e., what is each prefetch attempting to do).

**Solution:**

Given the assumptions stated in the program, we know that there are 4 memory locations per cache line (16 B/cache line / 4B/memory location = 4 memory locations/cache line). This means that A has 128*32/4 = 1024 cache lines in the cache, B has 128/4 = 32 cache lines, and C has 32/4 = 8 cache lines (sidenote: this means we didn't need to worry about A, B, and C being on the same cache line, because they always align perfectly with the cache line size).

Moreover, it also means that we need to prefetch several loop iterations ahead, because each loop iteration takes 16 cycles and cache misses take 64 cycles. Thus, our loop iteration is not going to take enough cycles to hide the latency of the next miss. By Little's Law, if we want the data for our next loop iteration to be available, then we would need to fetch is 64 / 16 = 4 loop iterations early. Note that this means, for the first couple loop iterations, we won't be able to hide the latency effectively – because we would need to wait 64 cycles for each of them to return. This influences how much prefetching we do for the first 4 j loop iterations – below the solution focuses on the steady state, but comments on the options for non-steady state iterations.

```
/*
   Very first loop iteration will always miss given the
   above assumptions, but we can hide some of the miss
   latency.
*/
PREFETCH(&B[0]);
PREFETCH(&C[0]);
PREFETCH_EX(&A[0][0]);
```

```
for (i = 0; i < 127; ++i)
     /*
        B repeats across outer loop iterations because it is
        indexed by i.  So we need to put its prefetch here so
        it works for i > 0 (really i > 4 since 4
        elements/cache line).  Could be done in an i % 4 ==
        0 if but will just be in MSHR otherwise.

     Note: also could check to make sure I is in bounds here:
     if (i + 4 < 127)
     */
     PREFETCH(&B[i+4]);
     /*
        Prevent misses to A in subsequent outer loop
        Iteration.
     */
     PREFETCH_EX(&A[i+1][0]);

     /*
        Given the system parameters, need to execute the next
        two prefetches every 4 j iterations.  Thus do some
        loop unrolling to help with this (unroll by 4 because
        1 loop iteration takes 16 cycles – unrolling by 4
        means the loop iteration now takes 64 cycles, enough
        to hide a cache miss/enough to prefetch just in
        time!).

        Note: this solution assumes j <= 31 and j < 31 behave
        the same – it may prefetch slightly more, but this is
        ok because of the size of our arrays and cache.
     */
     for (j = 0; j < 31; j += 4)
          // get C's for next j loop iteration
          // Note: technically only need to do this if i = 0
          // and if (j + 4 < 31)
          PREFETCH(&C[j+4]);
          /*
             A's second dimension spans multiple cache lines
             so prefetch ahead to get next j loop iteration's
             memory locations.

             Note: technically should check to make sure j is
             in bounds – e.g., if (j + 4 < 31)
          */
          PREFETCH_EX(&A[i][j+4]);
          /*
             First time through loops, these accesses miss,
```

```
            But are partially hidden by above prefetches;
            the code will stall until these misses are
            serviced.
        */
        A[i][j] = B[i] * C[j];
        /*
            Loop unrolling now causes all subsequent
            accesses within an iteration to hit if the first
            access hits, and next inner loop iteration also
            hits because of prefetching.
        */
        A[i][j+1] = B[i] * C[j+1];
        A[i][j+2] = B[i] * C[j+2];
        A[i][j+3] = B[i] * C[j+3];
    // next outer loop (i+1) iteration hits due to
    // prefetching of A and B (C already in cache).
```

**Note**: I did not take off points for doing (i % 4 == 0) or similar checks. These reduce the number of prefetches, and thus are slightly better in terms of prefetches, but also add branches that may lead to flushing and thus hurt performance. So there is a tradeoff here.

**Part B [4 points]**

What problem does prefetching attempt to tackle? Can prefetching hurt performance? To receive credit, you must justify your answer.

**Solution:**

Prefetching attempts to reduce cache miss latency (by speculatively/pre-emptively bringing in data to the L1/L2 caches that we believe we'll access soon). However, it can hurt performance in some situations. For example, prefetching may evict some useful data that would have otherwise stayed in the cache and (potentially) hit later. Even if we assume the cache is infinitely large, it can cause increased bus/interconnect traffic and additional cache misses (due to invalidation or coherence state changes) in a multiprocessor.

**Part C [4 points]**

Is prefetching more effective for single-issue, statically scheduled processors or multi-issue, dynamically scheduled processors?

**Solution:**

Prefetching is more effective for single-issue, statically scheduled processors. In comparison, multi-issue, dynamically scheduled processors have additional/alternate methods of hiding cache miss latency. For example, they can use out-of-order execution to identify additional, independent computation to perform while memory accesses are pending. In comparison, single-issue, statically

scheduled processors must stall any time a memory request misses in the L1 cache. As a result, prefetching is much more important.

## *Problem 3 [10 points]*

In a virtually indexed, physically tagged cache, the cache set to search is selected using only bits of the virtual address, so virtual-to-physical address translation can proceed in parallel with reading tags for comparison.

In the simplest design, the associativity of the cache is large enough so that the cache index and offset bits together fit entirely into the page offset bits. However, page size remains relatively fixed with architectures while cache size grows with semiconductor technology so this may require a high associativity.

### Part A [2 points]

A cache with a high associativity requires examining many ways on each access. This takes more time (even if comparisons are done in parallel) and uses more energy on each cache access, so we might want to keep associativity smaller. If a processor has 4KB pages and a 64KB level 1 cache, what is the minimum associativity required to use the simple virtually-indexed, physically tagged optimization?

**Solution:**

The total number of index+offset bits needed to address a direct mapped 64KB cache is lg 2^16 = 16. Only 12 of these fit within the offset bits for a 4KB page. Therefore, 4 are not in the page offset. If the cache associativity is 2^4, then the number of bits for index+offset will be reduced by 4, thus getting all the bits within the page offset. Therefore, we need an associativity of 16.

### Part B [2 Points]

Suppose a cache simply forms a longer index using a few of the least significant bits from the virtual page number. Describe a page table and access pattern where this cache will return incorrect data.

**Solution:**

If multiple virtual pages are mapped to the same physical page, say virtual pages 0 and 1 map to physical page 0, then two virtual addresses can map to the same physical address, but have different indices. If a program writes the value 12 to one address, then 25 to the other, and finally reads from the first address it will return 25 from the cache, but should see 12.

### Part C [6 points]

Does the MIPS R10000 have problems with synonyms or homonyms (or both)? If so, how does it deal with them? If not, why not? To receive credit you must justify your answer. *Hint*: to answer this question, you may need to visit/revisit the MIPS R10K paper [Yeager '96] which was a "read" paper from earlier in the semester.

**Solution:**

Yes, the MIPS R10K can have problems, especially with synonyms. This happens because the MIPS R10K chooses to have a larger L1 cache at the cost of potentially having synonyms for two

of the bits (bits 13:12) in the address. Thus, it is possible for synonyms to happen because we cannot do a proper translation between them.

The key paragraph to understanding how the MIPS R10K deals with this problem is:

> "Both primary caches use a virtual address index and a physical-address tag. To minimize latency, the processor can access each primary cache concurrently with address translation in its TLB. Because each cache way contains 16 Kbytes (four times the minimum virtual page size), two of the virtual index bits (13:12) might not equal bits in the physical address tag. This technique simplifies the cache design. It works well, as long as the program uses consistent virtual indexes to reference the same page. The processor stores these two virtual address bits as part of the secondary-cache tag. The secondary-cache controller detects any violations and ensures that the primary caches retain only a single copy of each cache line."

Essentially, the MIPS R10K solves it by a) assuming weird things won't happen in the L1 caches, so synonyms are unlikely ("It works well, as long as the program uses consistent virtual indexes to reference the same page."). Then, in the unlikely case when it does happen, the L2 cache controller will detect that a violation is happening and prevent the L1 cache from having both copies in the cache at the same time.

As it comes to homonyms, since the R10K is a VIPT, homonyms shouldn't be a problem since we have physical tags and thus if 2 PAs map to the same VA, they will have distinct locations but their tags wouldn't match when we do the tag comparison (i.e., even if they are both in the cache, and we choose the wrong block because of the 2 bits that differ between the VA and PA, the tag wouldn't match).

## Problem 4 [15 points]

You are the lead DRAM architect at a company that builds accelerators. Your company is designing a new accelerator and needs you to determine what DRAM page policy (open vs. closed) the accelerator should use. The memory access pattern for the new accelerator is similar to this snippet:

```
int main(int argc, char * argv[])
{
      int arrSize = atoi(argv[1]);
      int offset = atoi(argv[2]);
      …
      int * arrA = (int *)calloc((arrSize + offset), sizeof(int));
      int * arrB = (int *)calloc(arrSize, sizeof(int));
      int * arrC = (int *)calloc((arrSize*2 + offset), sizeof(int));
      …
      for (int i = 1; i < arrSize; ++i)
      {
            arrB[i] = arrC[arrA[i-1]] + arrA[(i + rand()) % offset];
      }
      …
}
```

### Part A [10 points]

Assume that your system has no caches, and thus requests are sent directly from the processor to the memory controller. Moreover, assume that arrSize is large enough that the arrays cannot be in the row

buffer at the same time, and that your memory controller can only handle 1 pending memory request at a time, so it cannot reorder memory requests. Finally, assume that for part A the main memory only has 1 bank and 1 DIMM. Which policy is likely to perform better? Again, to receive credit you must justify your answer.

**Solution**:

Given that the problem statement says that there are no caches and only 1 request can be pending at a time, this ultimately comes down to a question of how much reuse we can get from the row buffer. If we can get a lot of reuse, then an open page policy makes sense. If we can't get enough reuse from this, then a closed page policy is better. The access pattern is going to be:

LD arrA[(i + rand()) % offset]
LD arrA[i-1]
LD arrC[arrA[i-1]]
ST arrB[i]

(NOTE: You could potentially do LD arrA[(i + rand()) % offset] after LD arrC[arrA[i-1]], if you assume the compiler generates code this way. This is not typical in what I've seen compilers do, but if you stated it in your answer, I gave you credit.)

Given this, there is some reuse of arrA – specifically i-1 and i + rand() % offset will be accessed in the same loop iteration. However, we don't know what the values of rand and offset will be – it might be the case that overall offset will allow reuse if we use an open page policy (i.e., when i-1 and i+rand() % offset are in the row buffer at the same time) and it might not (i.e., when i-1 and i+rand() % offset are not in the row buffer at the same time). Basically, open page will help for arrA when the two accesses are near one another, and will not help when they are far from one another.

For arrB, the accesses are purely streaming. Since we only have one bank and one DIMM, there will not be any reuse of arrB possible. And since we can't overlap/reorder memory requests, or bring them into the cache, the A, C, B access pattern means if we were to use an open page policy, we would immediately have to close the page every time because a different array is being accessed.

For arrC, the level of indirection caused by arrA is essentially a red herring – since we can't reorder/overlap memory requests, there wouldn't be any reuse of arrC anyways. Thus, even if arrC didn't have the indirection, we'd still want to use a closed page policy (similar to arrB).

Thus, we have two arrays that prefer a closed page policy (arrB, arrC) and one array that may prefer an open page policy (arrA) depending on the access pattern and amount of reuse possible with the delta from rand(). Since arrA is accessed 50% of the time and arrB/arrC are collectively accessed 50% of the time, it is a reasonable option to consider open page. But, since we don't know how frequently arrA will see reuse, a closed page policy seems more prudent.

### Part B [5 points]

Now assume your memory has multiple banks/DIMMs. Which page policy will you select now, and why?

**Solution**:

If the memory has multiple banks/DIMMs, then the arrays could potentially be mapped to different banks/DIMMs and thus could be accessed in parallel. If this happens, then an open page policy makes more sense. In this system, you can have a row open for all three arrays, and thus you can hit in the row buffer for neighboring array elements in the same array (i.e., there is spatial locality). The only potential downside would be for arrA, which may not prefer an open page if rand() makes it likely you access locations in arrA that are far apart and thus not in the row buffer together.

If you said that arrSize was too big to fit in a row buffer anyways and thus closing may still happen frequently, I still gave you credit.