CS/ECE 752: Advanced Computer Architecture I
Fall 2020

Professor Matthew D. Sinclair

Midterm Exam 2 SOLUTION
Wednesday, November 18th, 2020
Weight: 15%

TAKE-HOME EXAM

NAME: _____

DO NOT OPEN EXAM UNTIL TOLD TO DO SO!

Read over the entire exam before beginning. You should verify that your exam includes all of the problems listed in the table below.

This exam is intended to take 2 hours, although since it is a take-home exam you may take longer if you so choose. The exam will be **due at 2:15 PM Central on Wednesday, November 18th, no exceptions**. You should submit your solution to Canvas, and we will be uploading these solutions to Gradescope to grade. You are expected to turn in either a) a scanned, handwritten PDF of the exam with your solutions or b) a typed PDF of the exam with your solutions. If possible, please do not change where the questions are and answer in the provided space, as this will make it easier for Gradescope to identify where the answers are.

Regardless of which option you choose, we will be checking for academic misconduct (including checking online tutoring services like Chegg), and you are expected to take the exam yourself and write your own answers. Any issues will be handled according to the UW Academic Misconduct policy.

Problem	Possible Points	Points
Problem 1	5	
Problem 2	25	
Problem 3	15	
Problem 4	20	
Problem 5	25	
Total	95	

Problem 1 [5 points]

In class we talked about how IPC is a poor metric for determining how good a synchronization scheme is. What metric(s) should we use instead? To receive credit, you must justify your answer.

Solution:

There are several possible answers for this. The simplest one is to use runtime – since runtime provides a direct metric for how long the synchronization took (i.e., if there are many failed lock acquires, then the runtime should be longer since this is on the critical path). However, runtime only presents an overall picture – it doesn't tell us anything about what happened when the program was being run, just how long it took to run. Thus, we likely want additional metrics beyond runtime. For example, we may consider having special counters that track how many atomics occur, and even how many of them hit or miss in the caches. This would require adding additional counters into the system (or using the existing L1/L2/etc. hit/miss counters as proxies, with some noise for non-synchronization accesses) but would give us insight into how exactly the synchronization algorithm is behaving under the hood.

Problem 2 [25 points]

Making memory accesses fast is seen as very important for high-performance CPUs. However, a wide variety of memory system optimizations such as sector caches and skewed caches have been proposed over the years.

You are the lead architect for a company that builds high performance microprocessors and dealing with memory accesses is becoming increasingly important as the data set sizes for the applications your company cares about become larger. Assume that your company specializes in producing out-of-order, superscalar uniprocessors (i.e., not multi-core processors), and that currently your design uses a single level of caches with 8-way set associativity. Your co-worker Bob argues that your next processor should redesign the caches to use sectors, while your co-worker Carol argues that your next processor should redesign the cache to be a skewed cache.

Part A [5 points]

What arguments might Bob be making?

Solution:

Sector caches are a more straightforward addition to the existing hierarchy than a skewed cache, as a sector cache (usually) improves performance at the cost of only a few bits per line (e.g., an additional valid bit and dirty bit per sector per cache line). Moreover, sector caches do a good job of reducing wasted fetch bandwidth from memory (in this case, since there are no additional levels of cache, this might be particularly important). Since memories are usually much wider than caches, this is a significant win, and if there is limited locality in the applications, would significantly reduce the time spent fetching this data and avoid wastefully bringing it in (if you add too many sectors and the transfer/bus width exceeds the sector size, then this could be a problem though). Sector caches can also reduce miss time, since only the sector with the requested word(s) needs to be brought in before responding to the request (Note: if you assumed critical word first support, then this would be less of a benefit/potentially no benefit at all). If the access pattern is sparse, power may also be a benefit with sector caches, similar to the reduction in wasted bandwidth.

Part B [5 points]

What arguments might Carol be making?

Solution:

Skewed caches are especially important when applications suffer from conflict misses (but do not suffer from capacity misses). This allows the skewed hashing function to "utilize" empty space in the cache (i.e., space not being used by other sets in a traditional, non-skewed hashing function) to reduce the number of conflicts for certain "hot" sets. If the applications the company cares contain a significant component of such accesses, this can be extremely useful – especially since there are no additional levels of cache and going to memory over (and over and over …) due to conflict misses will likely throttle the performance of the system. It also reduces "pressure" to move to a higher level of associativity (assuming capacity misses aren't the problem) since the skewed nature provides some of the benefits of increased associativity without all of the overheads that come with it.

Part C [15 points]

You are responsible for making the final decision. Assuming that your company only has enough time to implement one of these optimizations, which one should your company pursue? If you feel neither optimization is clearly better, be sure to explain what sort of additional information would be needed and how you would use this information to choose one over the other. Justify your answer.

Solution:

Crucially, the problem did not state what kind of applications your company is focused on. Information from this that might be useful includes how much reuse is common, and what sort of misses (e.g., capacity, conflict) are occurring. It may also be worth knowing what the hit latency is and how big the cache as is, is. Many of you made different assumptions about what this would say, which affected your conclusions (you were graded accordingly).

Absent this information though, it seems like sector caches are likely to be more useful. Since the cache is already 8-way set associative, the effects of conflict misses are likely already accounted for to a reasonable degree. Adding in a skewed cache requires designing more complex hashing functions and potentially redesigning the SRAM blocks to accommodate this. The win from doing this seems smaller since we already account for conflict misses to a reasonable degree with 8-way set associativity.

In comparison, sector caches require relatively small changes to the existing infrastructure and, especially if the applications have a sparse access pattern or large cache lines, would reduce overfetching. As mentioned above, this can be significant in the existing design, since we have no additional levels of cache and memories are usually wider than caches. Moreover, if time is short, the complexity of adding sector caches is also less. As long as we size our sectors to be "big enough", then we shouldn't have huge problems with less prefetching either, because a given sector will bring in enough of the other words on a line without fetching all of them wastefully (sidenote: this is why NVIDIA's L2 cache has 4 sectors). Several of you also mentioned that having a 16-way associativity cache on the critical path for memory accesses may be too expensive. This is true, depending on if you view the memory accesses all trying to be done in a very small number of cycles, or if the single level of cache is intended to be used for bandwidth filtering instead.

(If you said neither was clearly better, you should have talked about access patterns, cache size, etc. similar to mentioned above)

(If you said skewed better, you should have justified the applications having > 8-way conflicts, among other things)

Problem 3 [15 points]

Part A [10 points]

After helping with the decision in Problem 2, Bob decided to leave your company and now works for *ACMG* (a company that makes GPUs). Since his area of "expertise" is designing CPUs, he is asked to estimate the performance improvement that the companies GPUs can provide over CPUs. Given that the company designs GPUs with 128 CUs, the competing CPU has 4 CPU cores and 4-wide vector extensions, and that the CPU clock frequency is 1.5X faster than the GPU clock frequency, Bob makes the following estimate (ignoring the efforts of rewriting code, and assuming the code has sufficient parallelism to keep the GPU and CPU resources busy):

```
CPU – 4 cores    GPU – 128 CUs:              128/4 = 32X

CPU has SIMD extensions (4-wide):            32/4 = 8X

CPU clock is 1.5X faster than GPU clock:     8/1.5 < 6X
```

Therefore, Bob estimates that the GPU should provide no more than a 6X speedup over the competitor's CPU. His new employers find this surprising, because recent literature has shown that GPUs can provide 100X improvements over CPUs. However, you check Bob's math, and it appears that he is right. Nevertheless, it is true that some applications report much higher speedups. Provide two reasons why some applications report higher speedup on GPUs than 6X – what other hardware factors could impact application performance and cause the benefits to exceed what Bob calculated? To receive credit, you again must justify your answer.

Solution:

There are several possible answers. Some include:

- GPUs have special memories such as scratchpad's (NVIDIA's shared memory, AMD's LDS) and texture caches (OpenCL: images/samplers) that CPUs do not have. These special memories can be highly efficient and thus increase performance significantly beyond what a CPU can provide.
- GPUs have larger register files, which potentially allow more threads to be operating simultaneously (whereas CPUs may suffer from WAR/WAW hazards due to limited registers).
- GPUs use higher bandwidth memories like GDDR and HBM (although some CPUs are starting to use HBM). These memories can service more requests simultaneously, which improves GPU performance by allowing more overlap of memory requests (MLP).

The key idea, though, is that the GPU is using some feature that the CPU doesn't have or is smaller on the CPU.

Unrelated to the above: if you said that programmers did not tune their CPU code as well as their GPU code, and thus their performance gains were overstated, I would also give you full credit despite the question asking for hardware solutions. Because this happened frequently.

Part B [5 points]

The [Wall HPCA '19] paper claims that the current focus of the architecture community on specialization is myopic. Do you agree with the author's claim? If so, why? If not, why not? To receive credit, you must justify your answer.

Solution:

As with several other questions, there are many possible answers to this. Some answers include:

- Applications are not static. In recent years, we have seen a tremendous explosion in fields like ML, AR/VR, and Big Data (among others) that are frequently creating new, compute/memory-hungry algorithms that push the bounds of what is possible on modern hardware (and even modern accelerators). Thus, even if the community rapidly congeals around the "ideal" hardware design for a given application, since algorithms continue to evolve at a rapid pace, further innovation will be needed to design efficient hardware for these algorithms. This is especially true for the mobile and edge communities, where the number of accelerators (and the number of uses of smartphones and edge devices) is rapidly expanding.
- Once transistor scaling ceases, additional innovation is arguably the only way for systems to continue improving performance. There is no better motivator than necessity. And, as we've already seen with accelerators becoming more commonplace, what were previously seen as wild and crazy ideas that would rapidly be overrun by improvements in technology scaling are no longer so wild and crazy! Chiplets are another great example, although we did not focus on this in this class and thus I wouldn't expect you to say this.
- Technology may also improve (although we did not focus on this in class), with quantum computing and carbon nanotubes (among others) representing potentially interesting, long-term ideas.
- With the proliferation of accelerators, arguably programmability becomes the only thing that matters – and this may lead to more companies (like NVIDIA and ARM) focusing on hardware-software co-design and finding ways to improve both in tandem. However, this does run the risk of eventually petering out of the hardware may run into trouble.
- If you answered in the negative, you should focus on the issues the paper highlighted.

Problem 4 [20 points]

Part A [10 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 *)malloc((arrSize + offset) * sizeof(int));
```

```
        int * arrB = (int *)malloc(arrSize * sizeof(int));
        int * arrC = (int *)malloc(arrSize * sizeof(int));
        …
        for (int i = 1; i < arrSize; ++i)
        {
              arrB[i] = arrC[arrA[i-1]] + arrA[(i + rand()) % offset];
        }
        …
}
```

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. 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.

Alternative:

The problem did not specify the number of banks/DIMMs. If you stated that you assumed that the three arrays would map to different banks/DIMMs and thus could be accessed in parallel, I gave you full credit if you stated than an open page policy made 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.

Part B [10 points]

Most modern processors run multiple threads on a core, either with simultaneous multi-threading (SMT) or fine-grained multi-threading (FGMT). However, different companies have opted for different strategies in terms of how many threads per core they use. AMD and Intel tend to use two threads per core, while other companies such as Oracle up to eight threads per cores (e.g., in their Sparc T5 design). What sort of factors may have led these different companies to make different decisions about how many threads per core they support? To receive credit, you must justify your answer.

Hint: Since this is a take-home exam, you are welcome to consult the "read" [Feehrer '13] paper that discusses the Sparc T5 design. However, I believe the question can be answered without a detailed reading of this paper by thinking about the potential tradeoffs of 2 vs. 8 threads and what that may imply about the design.

Solution:

My old mentor Mike O'Connor always told me: "having 2 of something in a chip is easy, more than 2 is where the hard stuff comes in" … definitely applies here too. Some of the key reasons include:

- Oracle/Sun pipelines for their processors are significantly simpler (sometimes even being in-order), whereas Intel and AMD processors are usually much more complicated, out-of-order, superscalar processors. In general, the ROBs, instruction queues, RSs, wakeup logic, etc. in these devices are much more power-hungry than the Oracle/Sun ones. By designing simpler, less power-hungry processors, Sun/Oracle can instead use more threads, somewhat similar to GPUs, to focus on exploiting parallelism between threads (TLP) and in memory (MLP), instead of at the instruction level (ILP).
    - It is worth noting though that the T5 is a dual-issue, out-of-order processor. Thus, it is still exploiting some ILP, just not to the same extremes as the AMD and Intel designs tend to, which allows them to have a simpler, somewhat lower power, design.
- Since the cores are also simpler, Oracle could potentially crank up the frequency of their pipeline (although the T5 uses 3.6 GHz, which is similar to AMD and Intel – thus I would consider this a secondary reason).
- As a company focused primarily on applications like databases, the workloads Oracle are designed for also tend to have more parallelism (e.g., multiple queries running simultaneously) than the more general purpose workloads AMD and Intel need to support as well. Thus, while AMD and Intel often see diminishing returns after 2 threads per core, since Oracle is focused on applications that have more parallelism, they have more things to run in parallel (in the average case). This also

dovetails nicely with the design of their processors, which are more focused on exploiting TLP and MLP than AMD/Intel.

Problem 5 [25 points]

Consider a memory hierarchy with the following components and properties:

- An L1 data cache with 64 KB capacity, 64-byte blocks, direct-mapped placement, a write-no-allocate policy, and physical tags. An L1 hit stalls the in-order pipeline for 2 cycles (the load-use delay).
- A two-entry victim cache (i.e., victim buffer) between the L1 and L2 with fully-associative placement, LRU replacement. When a request hits in the victim cache, that entry should be removed from the victim cache (to be put back into the L1) and the evicted L1 cache entry should be the MRU victim cache entry. A reference that misses in the L1 cache but hits in the victim cache stalls a total of 4 cycles.
- An L2 unified cache with 16 MB capacity, 256-byte blocks, 8-way set associativity, LRU replacement policy, a writeback policy, and physical tags. A reference that misses in both the L1 cache and victim cache, but hits in the L2 cache stalls for a total of 20 cycles.
- A main memory system with 32 GB capacity. A reference that misses in all caches and is satisfied by the main memory stalls a total of 300 cycles.
- Physical addresses are 32 bits, and the smallest addressable unit is one byte.
- Virtual addresses are 32 bits and pages are 64 KB.
- Address translation is performed in parallel with the L1 cache via a 128-entry, fully-associative TLB.

Part A [6 points]

How many bits are required to implement each of the components described above? Be sure to identify and include the state needed to implement the various policies. Complete the table below. Show your work in the space below the table.

Table 1:

Cache	Bits per Block	Blocks per set	Bits per set	Total bits in cache
L1 Data Cache	512	1	529	541,696
Victim Cache	512	2	530	1,059
L2 Cache	2048	8	16,491	135,094,272

Solution:

(Note: if you did bits/block and bits/set as the bits from the address instead, I still gave you credit)

See above. Additionally, the work:

L1:

First, we need to determine the number of bits for the tag, index, and offset given our system hierarchy. For now,

Offset:

64B lines at the L1, $\log_2(64) = 6$, so 6 bits are needed for the offset

Index:

Since our L1 cache is 64 KB and each line is 64 B:

$$\frac{64\ KB}{64\ B/index} = \frac{2^{16}}{2^{6}} = 2^{10}\ indices$$

Given this, we need $\log_2(2^{10}) = 10$ bits for the index.

Tag:

32 – 10 – 6 = 16 bits for the tag.

Now that we have this, we can start calculating the information for the L1 data cache for Table 1:

Since our L1 data cache is direct mapped, there is only 1 way, and each set only has 1 location (block) to place the data in. This means that the *number of blocks per set* must be 1.

Next, each data block has 64 B (512 bits). This means the *bits per block* is 512.

Additionally, since we have a write-no-allocate policy for the L1 cache, we don't need to worry about having a dirty bit per cache line. As a result, in terms of metadata, all we need are the tag bits and the valid bit. As calculated above, each block will have 16 bits of tag; the valid bit adds 1 additional bit. Thus, there are 17 bits of metadata per cache line. Given this, we can calculate the number of *bits per set*:

$$64\ B * \frac{8\ bit}{B} + (16\ bits + 1\ bit) = 512 + 17 = 529\ bits$$

Finally, we can calculate the total bits in the cache. Recall there is only 1 block per set, and we know from our earlier calculations there are 2^{10} sets (indices):

$$529\frac{bits}{set} * 2^{10} sets = 541{,}696$$

Victim:

The victim cache has the same cache line size as the L1, so we can reuse some of the calculations from the previous part. For example, the numbers of bits per block remains 512.

However, the blocks per set changes, since the victim cache is fully associative. Thus, there is only 1 set, with both locations in the victim cache in it. Thus, the number of *blocks per set* is 2.

The number of bits per set also changes slightly, because now we need to track which entry is the LRU location (in the L1 we didn't need to track this since it's direct mapped and thus the entry at an index is always the thing to be evicted). Since there are only 2 entries, we need 1 bit per set for this. Thus, the number of bits per set is 530 instead of 529. Note that this LRU bit is per set, while the other metadata is per way in the set. This affects our total bits in cache calculation.

Finally, in terms of total bits in the cache, we multiply 529 x 2 since there are 2 ways in the set and 1 LRU bit shared across both of those ways (and only 1 set): 529 * 2 + 1 = 1059.

L2:

Since the L2 has a different line size and associativity, we need to redo our calculations:

Offset:

256B lines at the L2, $\log_2(256) = 8$, so 8 bits are needed for the offset

Index:

Since our L2 cache is 16 MB and each line is 64 B:

$$\frac{16\ MB}{256\ B/index} = \frac{2^{24}}{2^8} = 2^{16}\ indices$$

This gives us the total number of indices across all 8 ways, to determine the number of entries per way:

$$\frac{2^{16}\ indices}{8\ ways} = \frac{2^{16}}{2^3} = 2^{13}\frac{indices}{way}$$

Given this, we need $\log_2(2^{13}) = 13$ bits for the index.

Tag:

32 – 13 – 8 = 11 bits for the tag.

Now we can do our calculations for Table 1:

The number of blocks per set is 8, because there are 8 places a data block could be placed for a given set index.

The bits per block are 256 B * 8 bits/B = 2048 bits

In terms of metadata, we need 11 bits for the tags, 1 valid bit, and 1 dirty bit (since it's writeback) per way in the set, and the entire set needs 3 bits to track LRU information (note: to efficiently implement LRU, you may want a counter per way in the set, indicating when each set was last accessed – if you answered this way instead and made it clear you were assuming this, I still gave you credit; my scheme assumes we a priori we know which one was accessed last). Thus, we have:

$$= \left(256\ B * \frac{8\ bits}{B} + (11\ bits + 1\ bit + 1\ bit)\right) * 8 + 3\ bits$$

$$= (2048 + 13) * 8 + 3 = 16491\frac{bits}{set}$$

Finally, in terms of total bits in the cache:

$$= \left(\left(256\ B * 8\frac{\frac{bits}{way}}{B} + \left(11\frac{bits}{way} + 1\frac{bit}{way} + 1\frac{bit}{way}\right)\right) * 8\frac{ways}{set} + 3\frac{bits}{set}\right) * 2^{13}\ sets$$

$$= 16491\frac{bits}{set} * 2^{13} sets = 135{,}094{,}272\ bits$$

Part B [19 points]

Consider only the L1 data cache and victim cache for this part of the problem. Assume that all entries in both caches are initially invalid. For the memory reference stream below, determine which references hit or miss at each level (accessed from top to bottom). Assume that the victim cache is only accessed on L1 misses and all accesses that miss in the victim cache will hit in the L2 cache. Assume byte addressing and all accesses are 32-bit loads.

Complete the tables below. In Table 2, indicate in which cache each reference hits. In Table 3, indicate the total number of hits, misses, and stall cycles for each cache.

Hint: Calculate the L1 index bits for each access in the middle column. Then track which entries are added or removed from the victim cache.

Table 2:

Physical Address	L1 Index	Hits in?	Victim Cache Contents (MRU, LRU)
0x12340280	10 (0xa)	L2	Invalid, Invalid
0x12340294	10 (0xa)	L1	Invalid, Invalid
0x567802C4	11 (0xb)	L2	Invalid, Invalid
0x5678030C	12 (0xc)	L2	Invalid, Invalid
0x56780280	10 (0xa)	L2	0x12340280, Invalid
0x34560288	10 (0xa)	L2	0x56780280, 0x12340280
0x56780294	10 (0xa)	VC	0x34560280, 0x12340280
0x34560304	12 (0xc)	L2	0x56780300, 0x34560280
0xABCD02C0	11 (0xb)	L2	0x567802C0, 0x56780300
0x3456028C	10 (0xa)	L2	0x56780290, 0x567802C0
0x56780294	10 (0xa)	VC	0x34560280, 0x567802C0
0xABCD02C4	11 (0xb)	L1	0x34560280, 0x567802C0
0x567802C4	11 (0xb)	VC	0xABCD02C0, 0x34560280

Table 3:

Cache	# of Hits	# of Misses	Hit Latency	Total Stall Cycles
L1 Data Cache	2	11	3	4 = 2 * 2
Victim Cache	3	8	5	12 = 3 * 4
L2 Cache	8		20	160 = 8 * 20
All				176

Solution:

See above. For stall cycles, note that the hit latency for L1 and victim cache is + 1 cycle.