CS/ECE 752: Advanced Computer Architecture I
Fall 2021

Professor Matthew D. Sinclair

Midterm Exam 2 PRACTICE
WILL NOT TURN IN

NAME: ______________________________

DO NOT OPEN EXAM UNTIL TOLD TO DO SO!

Problem 1 [10 points]

Part A [4 points]

What are two ways/reasons that companies like VMWare impacted the success of the cloud? To receive credit, you must justify your answers.

Solution:

(there are several possible answers, each will be graded depending on how reasonable the answer is)

One big reason VMWare helped enable the success of the cloud is they demonstrated how computers could be commoditized and shared across multiple different users. Especially for small companies and startups that can't afford their own servers, this removed the need for them to purchase their own servers and also provided them with an essentially unlimited number of hardware resources to run on and scale their resources.

Another reason is that they provided some isolation and security (although not perfect) from multiple processes running on the same machine. This is important, especially when we consider that companies are running sensitive information (i.e., either sensitive information about the customer or sensitive information about their produces/employees/etc.).

Part B [6 points]

What are three downsides to SMT? To receive credit, you must justify why the downsides are problems.

Solution:

(there are several possible answers)

SMT necessitates adding two stages to the pipeline, in order to handle the fact that the register accesses take too long to keep the prior clock frequency without this change. This change increases the lifetime of registers, which increases register pressure in the system.

SMT also (potentially) sacrifices some single-threaded performance, because it may switch to fetching instructions from a different thread while the initial thread still has instructions it can issue. In practice, these overheads are usually small (~2%), and SMT does a better job of getting all of the threads to complete sooner by overlapping them, but if single-threaded performance is important, then SMT may not be the best choice.

SMT also increases power, because it requires replicating numerous hardware resources. This is necessary because SMT allows instructions from multiple threads to be executing simultaneously (unlike, say CGMT), which necessitates additional bookkeeping. Moreover, things like the register file need to be duplicated because each thread needs its own register file to access (similar for the map table). All of this extra duplication can increase (dynamic) power consumption, and likely also increases design complexity.

Other possible answers include cache interference.

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.

Grading:

2.5 points for reason
2.5 points for justification

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)

Future note: basically everyone said it depends, so maybe need to make question more specific in future.

Problem 3 [14 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 [4 points]

Some processors do not perform load permission checks in parallel with the load, while others do allow the checks to be done in parallel with the load. Qualitatively compare and contrast the performance and security impacts of this design decision for accesses, to privileged data, in light of the recent Spectre and Meltdown disclosures.

Solution:

In terms of performance, doing the load permission check in parallel with the load improves performance, because it does not require the new page to be brought in before we can do the memory access. In the common case (e.g., we are accessing user space data), the permissions will show that it was safe for us to access the data and thus we improved performance by overlapping the access with the permission check.

Moreover, it was previously assumed that this overlapping was safe because in the uncommon case (e.g., accessing privileged data), the load was only speculative and thus wouldn't be committed – when the permission checks came back and showed that we weren't allowed to access this data, we would squash the instruction. However, Meltdown showed that this was, in fact, not a safe assumption because the speculative instructions would modify cache state. The cache state could then be used in a side channel attack to glean information (e.g., values) about the privileged data.

Thus, bypassing the page table protection checks represents a security risk, although it does improve performance (potentially significantly, since these accesses are very long latency and may require page table walks, etc.).

Note that this design decision does not affect the susceptibility to Spectre, because Spectre instead (usually) relies on the branch predictor being trained to bring certain data into the cache (so even if the permission checks were correct, the data would still be brought into the cache. I did not take any points off if you did not mention this or distinguish them, since the question did not explicitly ask you to.

Problem 4 [15 points]

Assume we have the following code snippet we are trying to optimize for:

```
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 = 0; i < arrSize; ++i)
        {
                for (int j = 0; j < arrSize; ++j)
                {
                        arrB[i] = arrC[j] + arrA[(j + rand()) % (arrSize + offset)];
                }
        }
        …
}
```

Assume that we have 1 level of direct-mapped, writeback caches, which can only handle 1 outstanding memory request at a time.

Part A [10 points]

Assume that arrSize is large enough that we need to evict some entries. We talked about several cache replacement policies in class, such as LRU, NMRU, and FIFO (among others). Qualitatively, which of these three schemes will perform best for the above code snippet and why? If you believe they will instead perform the same, explain why.

Solution:

Since the memory access pattern is streaming, there is no opportunity for reuse (besides that from spatial locality) for arrC in the inner loop – unless the cache is big enough to hold all locations of arrB, arrC, and arrA we access in the inner loop. In this case, there is potential reuse across inner loop invocations, since arrC is accessing the same locations every time in the inner loop. Owever, since the prompt since arrSize is large enough that evictions will be required, this is unlikely to happen. Thus, arrB and arrC are unlikely to provide significant reuse – arrC will only get spatial locality in the inner loop since it's access pattern is sequential and monotonically increasing, whereas arrB will only get temporal locality, and no spatial locality since it will/may be evicted across invocations of the inner loop (and the same location is accessed arrSize times within the inner loop). However, arrA can potentially get some reuse, because the rand() accesses may cause us to access the same locations multiple times within the inner loop.

Thus, our replacement policy would want to focus on how much potential reuse we could get from arrA. I would expect that LRU would be best at this, because it would be able to determine which locations, if any, in arrA are being accessed multiple times and keep them around longer than NMRU or FIFO. When comparing NMRU and FIFO, I expect FIFO to be worst because it will only be basing replacement on when arrA's index was accessed initially, as opposed to how frequently rand() may be reusing it. However, there could of course be pathological inputs where this is not the case.

Part B [5 points]

Can Belady's algorithm improve on the performance of the best replacement policy? If so, why? If not, why not?

Solution:

Of course this will depend on the access pattern, but I expect Belady's will help over LRU (from previous response). Because it will be able to identify what rand() will be accessing soon in arrA, not just what it may have reused recently in the past.

Problem 5 [15 points]

The 3 C's model classifies misses as compulsory, conflict, or capacity. Later Norm Jouppi added a 4th C for coherence misses. Complete the following table for a system with a 16B direct-mapped cache, 4B blocks, write back caches, and nibble notation. Assume that S stands for Shared, M for Modified, and I for Invalid. Note that the Event and Outcome columns are intentionally offset. For the Outcome column, label each event as either a compulsory, conflict, capacity, coherence miss, nothing, upgrade, downgrade, or invalidation. For the Cache Contents, you should list the state and address for each location every cycle. You can ignore the data values.

Cache Contents (State:Address)			
S: 0000	M: 0010	S: 0020	S: 0030
S: 0000	M: 0010	S: 0020	**M**: 0030
S: 0000	M: 0010	S: 0020	**S**: 0030
S: 0000	M: 0010	**S**: 0020	S: 0030

S: 0000	M: 0010	S: 0020	**I**: 0030
M: 0000	M: 0010	S: 0020	I: 0030
M: 0000	M: 0010	S: 0020	**S**: 3030
M: 0000	**M**: 0010	S: 0020	S: 3030

Event	**Outcome**
Wr: 0030	(S $\rightarrow$ M) Upgrade Miss
BusRd: 0030	M $\rightarrow$ S Downgrade
BusRd: 2020	Nothing
BusWr: 0030	S $\rightarrow$ I Invalidation
Wr: 0000	S $\rightarrow$ M Upgrade (Miss)
Rd: 3030	Compulsory Miss
BusRd: 2010	Nothing

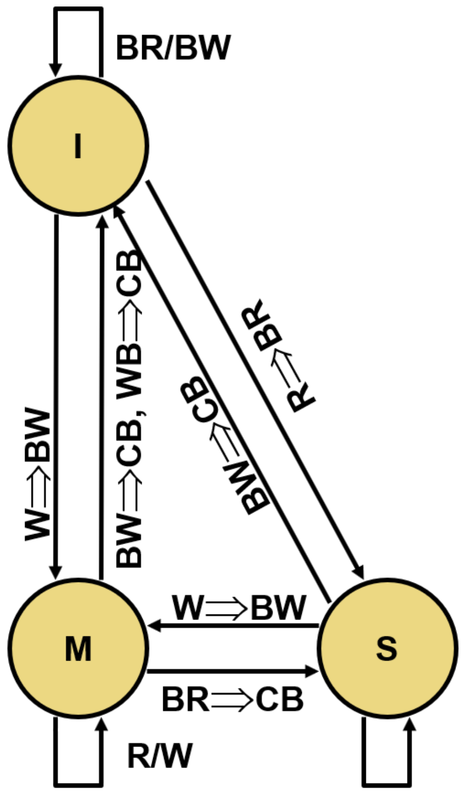
You can make the following assumptions:

- 16B direct-mapped cache, 4B blocks, and write back caches
- Bits [1:0] are used for offset, bits [3:2] are used for indexing, and bits [7:4] are used for the tag
- You can assume the cache is PIPT and the translation has already been done for you
- We are using nibble notation
- The data values can be ignored

Solution:

See above

Problem 5 (Continued)


BR/BW
I
W⟹BW
BW⟹CB, WB⟹CB
BW⟹CB
R⟹BR
W⟹BW
M
S
BR⟹CB
R/W