2020 Past Paper Solutions

1. a. In BOOM, the execution time attributable to a branch whose takenness is correctly-predicted may vary. What does it depend on? You should be able to identify at least three factors.

   It depends on the speed of loading the Branch Target Buffer and Return Address Stack entry into fetch PC, the tag comparator speed when mataching current PC with the PC tags field in BTB, and the access speed of instruction cache. It also depends on the branch type (jmp, ret, jal)

   b. What is the "dominating branch" in a Fetch Packet?

   The dominating branch refers to the first branch that is predicted taken in a Fetch Packet. The branches before it are not taken, thus it will not affect the execution flow. Since this branch is the first to be predicted as taken, it will change the execution flow so that the instructions following it will predict not to be executed, including possible branch instructions.

   c. Consider this Java method, which adds up the area of all the emojis in a list. Suppose there are many kinds of Emoji, so the area method has many implementations.

   i) How do you think the branch target for the call to area is predicted in the Boom design?

   The BTB in the Boom design provides a target field which will provide the predicted target address of the function call. When first calling `em.area()` , it will store the runtime function call address into the BTB. When it is invoked the second time, the NLP predicts that it will jump to the address previously stored in the BTB, thus Fetch PC will be set to this address on the next cycle and fetch instructions from that location.

   ii) Under what circumstances do you think branch execution would be slow for this code fragment?

   If the type of emoji in the `emojis` list varies a lot, especially when the types interleave (e.g. 0th element is Type 1 emoji, 1st element is Type 2, 2nd is Type 1 or 3) so that adjacent emojis have different types. In this case, branch misprediction will happen frequently or even happen in every single loop iteration since each time the actual function location is different than the predicted location.

   iii) What techniques could help with this? Try to identify at least two ideas.

   We can mitigate the situations describedin ii) either using a software approach or hardware approach.

   Using a software approach, we can group the emojis of the same type together, e.g. order the `emojis` list according to their types before passing into totalArea, or implemen `totalArea` several times, each for a different type of emoji

   Using a hardware approach, we can mark the `em.area()` call as polymorphic and try to buffer a list of function call locations of different emoji types, so that later when encountering the same type of emoji, it can different use the address of `area()` of the same type in the buffer.

   d. According to page 2, footnote 2, BOOM currently decodes each RISCV instruction to exactly one UOP:

   i) Some RISC processors do decode RISC instructions to more than one UOP. Suggest an example where this might be a good idea.

For example, in the store instruction, address calculation and translation can be performed separately while loading the cache line and attempting to store the data. In this case, it is a good idea to issue two uops to fetch address first and then prepare the cacheline and data. Basically any instruction that can be broken down into several parallel operations can be implemented using uops to become more efficient.

ii) In some processor architectures, instructions can be decoded to yield no UOP at all. When might this happen?

When there is instruction such as `nop` that acts like a bubble in the pipeline, e.g. it does nothing but just delay the pipeline by one cycle. In this case, nothing will be executed and thus no UOP will be generated.

iii) Suggest an example where the decoder might combine two adjacent macro-instructions to yield a single UOP?

For instance, when executing

```
mov x1, x0
mov x2, x1
```

the two instructions can be executed using one single uop such that the content of register `x0` is copied to both `x1` and `x2` in the same cycle. (this might not be a google example because we are not sure if uop would be capable of holding three registers, so we are under the assumption that the uops are capable of doing this)

Another possible example would be bit manipulation: when using two bit manipulation instructions, the decoder might optionally merge the into one, e.g. masking the first half of a number followed by masking the second half of a number

2. a. What limits the number of uncommitted, speculated branches that can be issued? How is this affected when a Fetch Packet contains more than one branch instruction?

   The number of uncommitted speculated branches are limited by whether there is an instruction after the branch that will produce a side-effect such as a store to write to memory. The effect of this instruction should be disabled since all the executions at this point are speculative: the real outcome of the branch is not known, thus it is unknown whether this side-effect instruction(s) will take place. If there are no side-effect instructions such as store, then we can speculate any number of branches as the pipeline allows. (This is to say that it is bounded by the number of pipeline stages before write-back and commit) This is also limited by the number of empty spaces left in ROB.

   When a Fetch Packet contains more than one branch instruction, we should stop speculation before the second branch since the first branch outcome is unknown. (Actually not sure about this part this is just some gibberish)

   b. Why does the ROB track the oldest excepting instruction? What happens, during out-of-order exectuion, when a second exception occurs before another excepting instruction is committed.

   The ROB needs to track the oldest excepting instruction in order to determine whether and when it will be commited (finished execution and marked as not busy). Since it is the oldest excepting instruction, it could potentially be the first instruction that causes an exception. If it is not commited (i.e. a branch misprediction might clear the ROB queue), then we don't need to throw an exception.

When a second exception occurs, we wait for the first exception to be committed and first exception to be thrown. Once the first exception is committed, the exception handler will be executed, which will flush the ROB entries, including the second exception. Thus the second exception will not be thrown/will not occur.

c. Identify and explain two different events that can lead to a physical register being freed for reallocation.

Branch misprediction, exception, context switch

d. When a load instruction is executed, an access to the data cache is initiated (line 883):

i) Why might that access be killed?

Because if there is a previous store that updates the data requested by this load, then we should not access the old value in the memory because is it outdated. Instead, we should forward the value produced from the newest store relevant to this load

ii) The store mask records which stores a load might depend upon (line 878). What happens to a load instruction if one of the stores in the store mask does not yet have a valid address?

The load will be pending and stay in the LSU queue until the address is resolved and then check if it is matching with the address of the load

e. In BOOM, the TLB translates virtual addresses to physical addresses as soon as they are generated (Figure 1.23). A common alternative approach is to perform address translation in parallel with L1 cache access - a virtually-indexed, physically tagged cache design.

i) What is the main performance advantage of the VIPT approach?

VIPT is dolours parallelism by allowing address translation and tag matching to perform simultaneously. If there is a cache miss, we can then directly use the translated address to index L2 cache

ii) With VIPT, store-to-load forwarding could be done by comparing virtual addresses, before translation. What correctness problem arises?

During context switch, the LSU might not be able to finish execution after the new process/thread is installed, thus it has the hazard of executing load/store from the last process.

3. a. Why do you think BOOM's D-cache supports hit-under-miss wile the I-cache does not (line 149) - why is hit-under-miss more important for data-cache accesses?

For I-cache, the instructions are likely to either execute sequentially or fetched based on branch predictor, which is likely to predict correctly (since most branches' takenness are bimodally distributed and branch predictor is designed based on this). This means that I-cache will likely to get a miss. In comparison, data access does not follow a certain pattern and hence it's likely to get more misses than I-cache, thus useful to have hit-under-miss in order to parallelize execution and miss penalty.

b. BOOM's implementation of the RISCV memory model is described in Section 1.14.3 of the document.

i) Why do you think the RISCV WMO memory model requires loads from the same address to be ordered? (line 896)

Just like the same paragraph has said, the cache coherence probe event might snoop the core's memory. If the loads are not ordered, then the changes to the cache coherency state (e.g. shared, dirty, dirty shared) might not be consistent to other cores. (Not entirely sure about this one)

ii) Why does it matter, in deciding whether to replay the load, whether a cache coherence probe event occurs? Explain your answer carefully.

If cache coherence probe event occurs, it will observe the state of that cache line and determine if invalidation needs to be performed. If loads are not ordered, then it might change the coherency state differently and thus will provide the wrong states when determining how to deal with this cache line. (e.g. when other cores writes to the same cache line, between two loads, then its relative ordering matters because the last-to-execute load will be invalidated and it will fetch from the core that holds the newest copy) (again, not entirely sure about this)

iii) Input/output devices are controlled and used by reading and writing device registers that are mapped into the processor's memoroy address space. Such memory regions are marked in the TLB as "not-cacheable". How do you think consistency of loads from not-cacheable addresses is maintained?

Those loads need to be strictly ordered (i.e. all other loads need to wait before the device "not-cacheable" loads finish) because reading to a device memory might generate side-effects to the device (e.g. device thinks that you performed two reads and change its internal register state to record the number of reads)

c. BOOM sends loads to the data cache as early as possible - so cache accesses include speculative loads (line 923):

i) Explain how allocations due to speculative loads can be exploited via a side-channel attack to circumvent language-based security

Since loads could be speculative, we can invoked the victim code so that it reads sensitive/target addresses before the language-based security finds out that these read requests are out-of-range. We can first cause the victim to run, then evicts lines of interest, then run the victim again. Any variation in execution time indicates that the line of interest was accessed by the victim.

ii) Explain how replacements due to speculative loads can be exploited via a side-channel attack to circumvent language-based security

Alternatively, we can first fill the cache with attacker's data, then run the victim. We then time the access of each cache line to see if any are evicted; if so, the victim must have touched an address that maps to the same set.

iii) Suggest how BOOM's L1 data cache might be modified to avoid such vulnerabilities

L1 data cache can check if the address/tag is within the valid range or not, so that if it is invalid, it should abort the load immediately and throw an exception.

4. This question partly concerns the architecture of graphics processor, such as the large discrete GPUs made by NVIDIA and AMD. We consider, for the sake of discussion, a GPU with 16 cores (NVIDIA calls these "SMs"), each running 32 hardware threads (NVIDIA calls these "warps"). Each thread ("warp") executes 32-wide SIMD instructions, supporting an SIMT programming model with one software thread per vector lane.

a. Such a GPU has a fixed-size but partitionable register file:

i) What limits the number of software threads that can actually run on a core (“SM”) at the same time?

The number of register used by each thread, e.g. if each thread uses a large number of registers then the regsiter file cannot be partitioned into many sections to support execution of many threads.

ii) Why are programmers advised to try to achieve high “occupacy”?

Because higher occupancy means more threads within the possible resources (e.g. register file and execution unit), thus better latency hiding effect for memory accesses since during a slow cache miss or memory request another thread can run in order to utilize the processor more.

b. Here we address the use of 512-bit SIMD vector instructions (such as Intel’s AVX512) on a conventional CPU. Consider this function which adds two vectors:

i) What correctness concern might be identified in the compiler, leading this loop not to execute with vector instructions?

If the value of N is less than 64. (512 / 8 = 64)

ii) Describe two further reasons the compiler might generate multiple copies or versions of this loop body?

If N is not divisible by 64, or if the array A, B and C are not aligned with 64-bytes. Or even worse, if A, B and C regions overlaps.

iii) Do you think the stores to array C will result in cache allocations? Offer an argument for, and also an argument against.

Stores to array C will result in cache allocation because it might be an entirely new array and when there is another thread reading C, it is convenient to store it in the cache to provide the data.

Stores to array C may NOT result in cache allocation because we can have a special hardware that buffer/accumulate all the SIMD calculations and directly forward the result to main memory. It also might be the case that C is not referenced later in the program (e.g. memcopy function may not cache the copied data due to the same reason: those data might not be referred to later in the program)

c. Consider this function (a CUDA kernel), which adds two vectors. It is to be executed as an SIMT thread:

i) How is predication used in execution of this kernel?

Predication can be used to control the execution of each lane (GPU threads). Before a GPU thread executes, it will set the predicate register according to the value of B[i] > 0. It will then execute C[i] = A[i] + B[i] for the lanes where the predicate register says true, and C[i] = A[i] for those that are false. Specifically, it will set C[i] = A[i] for all i and only add B[i] for those lanes that are true.

ii) Do the loads in this kernel successfully exploit spatial locality?

The loads in this kernel successfully exploit spatial locality because near-by GPU threads will use adjacent elements of A and B

iii) How could branch incoherence be avoided?

If all the GPU threads in a warp goes the same way, we can achieve better performance. This can be achieved by waiting for all GPU threads to finish execution before branching off to another place. e.g. adjacent threads run in locksteps