Welcome for discussion :)

## Question 1

(not examinable this year)

2a.

CPU-bound: the microarchitectural profiling shows a significant front-end bound and back-end bound. In the former case, the pipeline stalls since the issue-width of the CPU is not sufficient to make full use of its microarchitectural resources (e.g. functional units). In the latter case, it could potentially also be due to overwhelmed cache misses, which is not CPU-bound and should be distinguished by checking the number of misses of caches. If the misses are not significant, and the code contains a considerable amount of heavy instruction like MULT and DIV, they can be considered CPU-bound.

memory bandwidth bound & memory latency bound: If bandwidth-bound, the memory bus is fully utilised. Both bandwidth and latency bound would show significant cache misses in the profiling. They should be distinguished by further analysing the source code. Guided by the profiling tool, the hot section of the program can be found. The size and pattern of the data structure being accessed need to be figured out. If the size is larger than the caches (L3), it is likely to be bandwidth bound. The access pattern needs to be looked at to ensure the code is not iteratively retrieving the same region of data separated by a large period of access. And the latter is considered latency bound, especially when the access size is not large.

b. i.

The comparison time, which involves the time to access the i-th and j-th elements in the array, is initially low. Because they can be fit into the cache, providing quick access. The curve goes down slightly for the first half. It can be due to cache warming, where the first few accesses can take longer since the data is still in memory and need to be loaded. Those compulsory misses would increase the average comparison time more significantly since the total number of elements accessed is small at the beginning. After a certain threshold, the access size exceeds the cache capacity, causing the access latency onto a higher step. The latency time keeps growing later on, because accessing more array elements may require more cache lines to be loaded from the memory.

ii.

⊙⊕

s\_trav(R.n=inputSize, R.w=sizeof(i), u=1) ⊕ s\_trav(R.n=inputSize, R.w=sizeof(j), u=1)

However, this model is not accurate. Since the input[i]/input[j] is each accessed for inputCount^2 times but remains at the region of size inputCount, which cannot be accurately captured by the algebra.

**Not sure about above, here’s an alternative:**

This experiment cannot be perfectly modelled using the pattern access algebra covered in the lectures. For our input[j] we are performing a sequential traversal repeated inputCount times. Modelling this as inputCount sequential traversals (using the ⊕ operator) would be inaccurate as we would not account for the performance benefit of having these values in our cache, since our access patterns do not identify accesses to the same array. Furthermore, modelling the input[i] access in our inner for loop is difficult, as it does not match any of the access patterns (it's a repeated identical access). As such, we can assume that this value would be kept in a CPU register, meaning we would only access it once. By unrolling the outer for loop, our closes approximation would be inputCount+1 s\_trav(R.w=1, u=1, R.n=inputCount) accesses performed sequentially (⊕), inputCount s\_trav for the input[j] accesses and 1 s\_trav for the input[i] access.

iii.

If the number of duplicates increases, the condition in the code is more likely to be true, which leads to more accurate prediction of the branch predictor as it goes beyond 50%. However, the overall performance will not be improved much, because the program is considered memory-bound, where the time to retrieve the elements dominate and is compulsory in every iteration.

c. i.

Green for garbage collection, Blue for manual memory management, Orange for reference counter.

Manual memory management has no performance overhead (if used corrected and efficiently). Data structures are just destructed at the right time. It should be considered to have the highest performance. But it is initially sub-optimal in the graph, maybe because in code snippet the destructions are not placed at the best position. Reference counter works efficiently as well, except that it requires extra space for the counters. It is implemented by the compiler so that it tends to have better tuning of when to destruct. However, after a certain threshold of vector size, the counters may not fit into the cache. So it is overtaken by the manual one later. Garbage collection requires extra program to detect and remove the unused objects while maintaining certain data structures for the meta-data. It should be considered the slowest.

**Alternative (thoughts?):**

Assumption: In all 3 cases, we assume that the memory is freed once we leave the scope of our benchmark (i.e. the de-allocation time is included in the benchmark, despite there still being a reference to the vector).

The green line is likely to correspond to the std::shared\_ptr approach with reference counting. This is because for each shared pointer, additional memory must be allocated to track the reference count, and this memory must also be freed once the shared pointers are de-allocated. This additional memory-allocation overhead makes them 2x slower than manual memory management. (https://www.modernescpp.com/index.php/memory-and-performance-overhead-of-smart-pointer).

The yellow line probably corresponds to the boehm garbage collection implementation. Whilst there is overhead in running traces, the way the memory is de-allocated once we leave scope is likely to be more efficient than the naive for-loop used in the manual memory management approach. However, as the size of the number of objects increases, so does the cost of running a trace. This cost is eventually greater than the more efficient de-allocation, which causes it to eventually perform worse than the manual memory managed approach.

This leaves the blue line as manual memory management.

ii.

Yes, when it exceeds the size of the caches, reference of newly allocated items would be allowed to be written into the array only after the previous ones written back to the memory, in order to make spaces in the cache.

**Alternative cont.**

The size of the allocation would impact the relative performance of the benchmarks. Allocating more memory is likely to be more expensive as `new` uses a sub-allocator, and it is more likely that this sub-allocator can provide a small block directly than a large block. However, this would affect all 3 of our benchmarks as they all use `new`. Taking our manual memory allocation (blue) as our baseline, the garbage collection approach would perform worse relatively as the cost of running a trace will become more expensive, (assuming objects need to be loaded into our cache when running a trace - I'm not certain how this works). With reference counting, the performance relative to our baseline may improve slightly, as the size of the counter we are allocating to reference counts will become smaller as we increase the size of the allocation, and so it's cost relative to the allocation for the object may become less significant. However, it is likely that the overhead of the second allocation (for the reference counter) is more significant than the cost of allocating a larger object for quite a while, although asymptotically we'd expect its performance to dominate, which should bring the performance of reference counting in line with manual memory management.

d.

Opportunities:

1. The software systems built upon can have a better view of what components work quickly while others are slower, guiding it to perform merely certain optimisation rather than trying to address all possible cases. Overhead is usually introduced when performing unnecessary optimisation.

2. Different models of CPU will have different combination of SLAs regarding various aspects (e.g. some will have stronger guarantees on performing vectorisation operation, while others are claimed to be better at control instructions). It will be easier for professional users to choose the one best suiting their needs.

Challenges:

1. The performance (e.g. throughput, latency) is measured under certain type and amount of workload. When defining SLAs, they either need to be only valid under the pre-defined workload or may be failed to be obeyed by the hardware in the exposure of other workloads. It makes defining such SLAs challenging.

2. If the SLA is decided such that they can mostly be met, the performance metrics it guarantees may not be acceptable. Because in order to not be voilating it at the majority of time, those number should be set such that all measurements are taken in the worst cases.

3. The architectures of the CPUs are so complex (for general-purpose) that the underlying components may not be assigned corresponding individual SLAs thoroughly.