## Question 1

a

i)

Search HBIR: Quick experiments aimed at determining the maximum sustainable request rate for the system

RT curve building: transaction rates increased linearly, then kept constant to measure transaction response times until we find the maximum ops.

Validation: Establish the validity of the experiment, for example, check for failed transactions

Profiling: Monitoring data is collected and analysed

Reporting: Results are visualised (i.e. as a web page)

false

ii) A bottleneck switch is when, in a distributed system, there are many potential bottlenecks and while processing a transaction, the bottleneck may switch between different resources very rapidly. Bottleneck switches can make it difficult to diagnose performance issues, since the switching may happen at a rate below the resolution of your profiling tools and it can be difficult to diagnose bottleneck switching without looking for it.

b

i)

V(E) = 1

V(C) = 0.2 + V(P) \* 0.1

V(P) = 0.6 + V(H) \* 0.6 + V(C) \* 0.1

V(H) = 0.2 + V(P) \* 0.1 + V(H) \* 0.3

Solve system

V(C) = 0.288

V(H) = 0.411

V(P) = 0.875

Average session length: 1.574

ii)

Total resource utilisation = sum of all classes (arrival rate \* demand)

Overall: L = 300 requests/sec

LP = 262.5

LH = 123.3

LC = 86.4

Front server demands are 1.0ms for all classes

Database server demand for the history page is 4.5ms

Front server utilisation = 0.4722 (47.2%)

Database server utilisation = 0.5549 (55.5%)

iii)

What you would need to do is calculate the state probability matrix for n visits, which is calculated iteratively from the probability matrix for 0 visits (with all zeroes except for the entry probability). Calculate that matrix for n visits and then observe the probability that the visitor is in state X, which is equal to the probability that the visit terminated in n visits or less. Then, look at the probability that the visitor is in state X for n - 1 visits and subtract that from the previous result to get the probability that the session terminated in exactly n visits.

C)

Resource C can be a bottleneck as request rates increase. Regardless of which request class has their rate increase, each induces a larger demand on resource C than any other resource so resource C would be the first resource to bottleneck the overall request rate.

^In this case C is the slowest resource for all classes, so it will always be the bottleneck irrespectively of the request mix. As the demands at the other resources are strictly smaller than the demand on C on all classes, the resource utilization will also be smaller due to the utilization law.

## Question 2

a)

i)

y = q0 + qa \* xa + qb \* xb + qab \* xa/b

q0: 10.5

qa: 2.5

qb: 3.5

qab: 1.5

Shows that A and B have a similar effect and the effect of their interaction is also similar but slightly smaller. Explains why the most efficient configuration is with both options as off.

ii)

SST qa: 25 (30.1%)

SST qb: 49 (59.0%)

SST qab: 9 (10.8%)

SST: 83 (100%)

b)

i)

Trend component: Long-term trend (deterministic)

Seasonal component: Periodic changes (deterministic)

Random component: Irregular component (stochastic)

You'd apply an autoregressive model to forecast the random component of a metric. An AR(1) model would require two parameters, the c fixed offset value and the phi coefficient for how much the previous value of the model is weighted. A component of noise is also added.

ii)

E[a] = 1

Var[a] = 3

K1 = 0.9

1 = c / (1 - phi)

3 = sigma^2 / (1 - phi^2)

0.9 = 3 \* phi

c = 0.7

phi = 0.3

sigma^2 = 2.73

iii)

Transfer function for closed loop systems:

C(z) S(z) / 1 + C(z) S(z)

Top = N(z) = 2z / (z - 0.5)^2

Bottom = D(z) = 1 + (2z / (z - 0.5)^2)

Only pole of D(z) = -0.5

lambda = 0.5

Settling time = -4 / ln lambda = approximately 5.771 sec

Steady-state gain = H(1) = 8 / 9

## Question 3

a)

Tracing: Collecting a complete log of all the states the program enters during execution

Profiling: A representation of how many resources a program uses during certain parts of execution

Tracing and profiling are both ways to allow programmers to gain an insight into how their program performs. Tracing, however, introduces far more perturbation due to the volume of events requried to create a complete trace and is used when a complete picture of what the program is doing is required. Profiling is much more targeted and is usually used to gain an overall idea of where the performance hotspots are since it produces much less perturbation by limiting the information gathered to just timing how much time is spent in each section of the program.

b) Uniformly distributed random data between 0 and 100 with a mean of 50. No pattern, 50/50 success fail.

ii)

does not factor into the answer since the input pattern needed to trigger mispredictions forces the predictor into the "weakest" states where a branch predictor would change its prediction after the next misprediction.

C)

I)

Sequential traversal over input1

Size of input 1 \* log (treeSize) repetitive random accesses

Model fails to capture temporal aspects of execution. The subsequent sequential accesses to input1 only occur after the random access pattern occurs and there's no way of capturing the probability of eviction and invoking a new random access.

^ attempting to write this using the equations

S\_trav(R.w = 1 , u = 1, R.n = input1Size) ⊙ rr\_acc(R.w = 1, u = 1, R.n = input2Size, r = input1Size \* log(input2Size))

Ii)

e.g. data hazard bc multiple accesses to random regions of input2

e.g. bad speculation bc many if-else branches in the code

Verify by running the code and checking for the characteristics in the Intel decision tree for your examples (e.g. bad speculation: micro-ops issued but not retiring)

Iii) [trying to describe the optimisation loop]

first, profile the code and check for optimisation opportunities. (give example of an opportunity and what you would do.) Then profile the code again and check if the optimisation is effective. Repeat until the desired level of performance is achieved or no further optimisations are feasible.

See the \*2, cry, change it to << 1 to make the ALU happy.

3d) Maintainable, Flexible, Fast

## Question 4

a see slides

B)

Structural hazard, Control hazard, Data hazard, 4th one unknown?

F1 exposes structural hazard, both threads are competing for the ALU to perform arithmetic operations

Possible other answer: F1 exposes false sharing. Assuming each thread is on a different core, both are writing to the `sums` array, both elements of which should be on the same cache line as the array is small. Hence this would lead to write contention for that cache line and many needless cache line invalidations.

F2 exposes control hazard from branch misprediction, main for-loop contains a branch whose takenness can be unpredictable depending on the order of elements in `input`

F3 exposes data hazard, each loop iteration accesses a new struct which may fill an entire L1 cache line depending on the system parameters, hence causing a cache miss on every loop iteration

F4 exposes control hazard from function calls (frontend bound), as it contains a recursive call and hence would execute a `jmp` instruction to the top of its function body many times, creating many pipeline bubbles

C)

I) memory (latency) bound. Almost 10x speedup for multi-threaded case hints that single-threaded case may have been stalled on waiting for data to be fetched into the cache. Multi-threading allows other threads to continue doing useful work when a thread is stalled on IO. Profile also shows that the code is around 50% memory bound in both cases.technically you do

Ii)

video compression app (need to compress multiple frames, video is often very large and hence frame data are mostly stored on disk, frames may be compressed independently)

computer games (need to load multiple independent resources from disk e.g. for graphics)

Theoretical example: dynamic programming with a space-restricted array (or hash map). That is memory bound because if the hash map is full, eviction will be needed and recomputation will be needed. Also, perhaps LRU\_cache() in Python does the same thing (basically automates memoisation)?

D)

Good: low level control of data structures like in C, programmer can tune app behaviour to a high level compared to languages that take a one-size-fits-all approach that works for most cases but hence doesn’t achieve the maximum possible performance (e.g. auto garbage collection in Java)

Good: able to use compiler intrinsics, gives programmer fine control over actual compiled code and hence ability to make very targeted optimisations depending on the nature of the app.

Bad: Many easy-to-misuse footguns, template [substitution failure is not an error](https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error)

Bad: Lots of baggage from years of history, and no simple way to do some things (eg parse a string into an int, or split a string) compared to modern alternatives such as D and Rust.

Bad: lack of support for thread-safe data structures in standard library, concurrency is becoming more important as we reach the end of Moore’s law, support for such structures would make it easier to write high-performance code that uses concurrency (rather than reimplementing such structures yourself)