ECE459 Final Submission

Daniel Robson
20725956
dlrobson,

April 20, 2022

1 Short Answer

1.1 Benchmarking

The first run is likely to be unrepresentative of the actual performance because the cache hasn't stored any references used by the program. It can be referred to as "cold cache" in that scenario. Aspects of the program may store "warm cache" during the first few iterations, which can store references to disk storage locations.

1.2 Queueing theory

$$\begin{aligned} T_q &= \frac{s}{1-\rho} \\ &= \frac{90}{1-\frac{1}{150}} \\ &= 90.60 \end{aligned}$$

Increasing the rate of cooking the toast will improve T_q. Suppose we had a service time of 45s, this would result in T_q to halve.

$$\begin{aligned} T_q &= \frac{s}{1-\rho} \\ &= \frac{45}{1-\frac{1}{150}} \\ &= 45.30 \end{aligned}$$

Alternatively, you could also add more toasters, complicating the equation.

1.3 Reduced-resource computation

This is an acceptable strategy where there is significant computation that is required to calculate something, however a result is needed quickly. Google search is an example, where they may try to tailor their search to your previous searches. However, the website needs to be snappy, so instead they will just show the default search given your region.

1.4 Queueing theory

Yes, poisson distribution is okay here. Since Black Friday is a very infrequent event, it falls under the category of very small probability, while having a large number of sources.

1.5 Compiler optimizations

Dead store optimization can be useful, since the memory can now be freed for future use. This would be great for programs running on hardware with limited memory, that benefit in a reduced memory footprint.

1.6 Overhead

Two possible reasons for this is that the printing to console has to be done in series, resulting in randomness in the locking of stdout. Another reason is the magnitude of data being printed: since more data is being printed, there is a larger variance of printing. The program is slow either way since it needs to lock the printing mutex, and it still needs to execute the print even though it's /dev/null.

1.7 Inter-thread communication

For GPU CUDA programming, shared memory is preferred to reduce the large number of messages that would need to be relayed. Speed is important for GPU programming, which is a benefit of shared memory.

1.8 Bottlenecks

One possible reason for the general-purpose memory allocator to be the bottleneck is because they are often centralized and only support one thread allocating/deallocating at a time. This would result in the allocation being wrapped around a mutex. A solution is to preallocate the space beforehand for the individual threads.

1.9 AWS

AWS is considerably faster, AWS fees are cheaper than using it on my own, or AWS down-time.

1.10 Rust/ECE/UW

2 Self-Optimizing Payment Processing

We're going to try and figure out what the rate we're limited by is. Assume that the current limit time is defined by t. To find this, we'll do a binary search of a really short time 0s, and t. This will be done by calling try_to_pay sequentially with a time delay as specified. The binary search will be broken up as (0, 1) -¿ (0, 0.5), (0.5, 1.0) -¿ (0, 0.25), (0.25, 0.5), (0.5, 0.75), (0.75, 1.0), etc. Try it at each time limit multiple times to ensure the the limit time is consistent.

If the limit time is found to be inconsistent, then another method to find the optimal time length will be used. This will be based on the success rate and the interval time, where their product will be maximized.

Once we have the limit time, we'll divide each invoice into it's own thread. The try_to_pay will be protected by a mutex, and when the mutex is acquired, try_to_pay will be called, and a sleep for the found limit time. Then the mutex is released, and another thread will pick it up. This way, the try_to_pay will be called at it's maximum frequency.