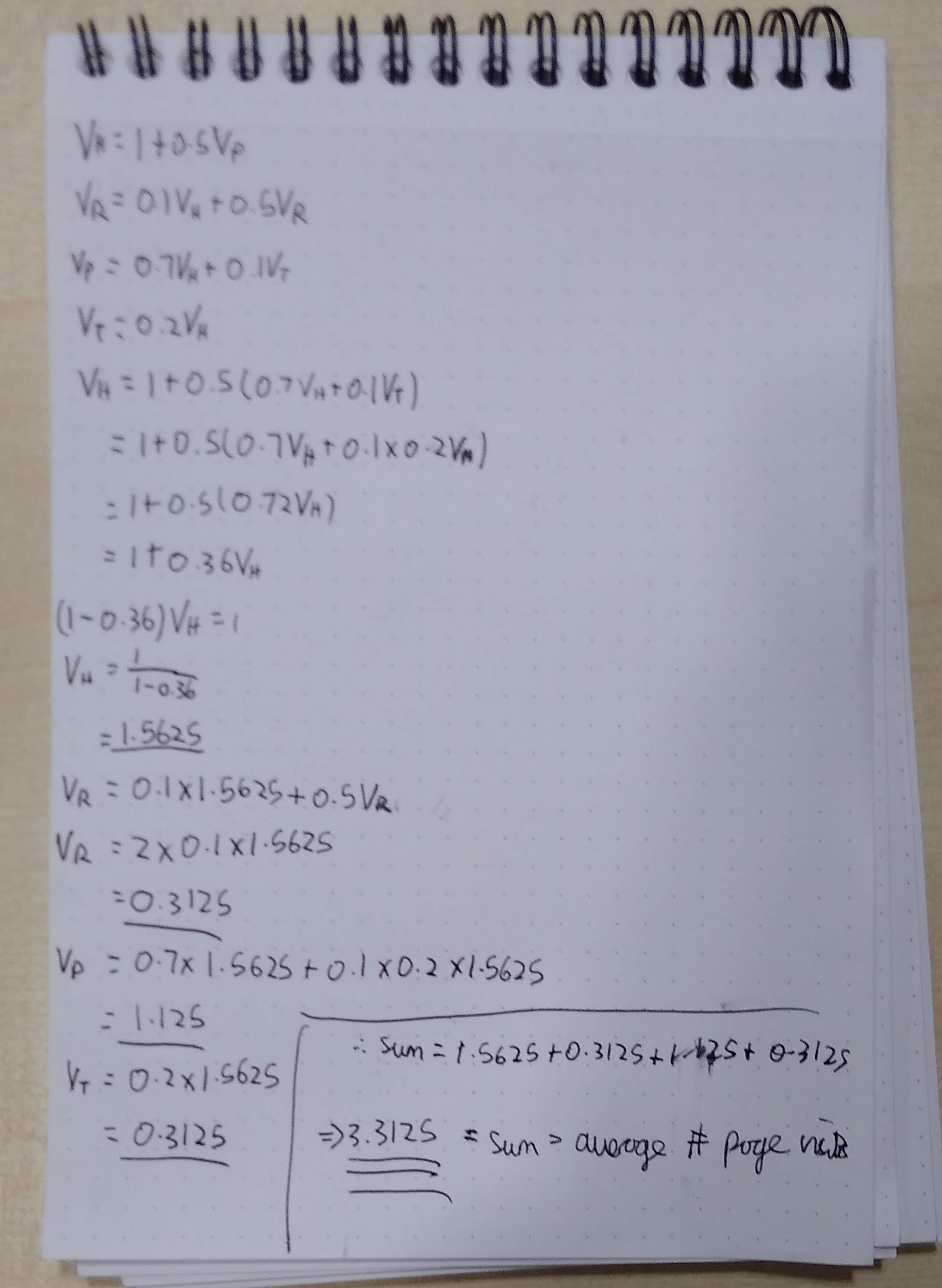
USUAL DISCLAIMER THAT I HAVE NO IDEA WHAT I AM DOING. MY ANSWERS LIKELY CONTAIN SUBSTANTIAL ERRORS.THESE ANSWERS COME WITHOUT WARRANTY FOR ANY PURPOSE ETC...

1. 1. 1. False. Benchmarks can be specific to a specific system. For example if I know that I will be running my program on the specific hardware in question the benchmark need not generalize to other systems. If however I want to optimize a program being run on many different systems, then the benchmark should generalize to those many systems.
      2. **False.** Benchmarks should be representative of what you are trying to optimize. If your optimized code/system will not use lots of vendor specific features, then they should not be used in the benchmark. Of course if the program/system being optimized does use a number of vendor specific features, then it can make sense to use those features in the benchmark. From slides: “benchmarks should avoid to systematically favour systems from specific companies.”
      3. True. Assuming the log data has some way of identifying individual users/sessions(maybe IP Address/session id?), as well as logging of requests to the server, then it makes sense to use the proportions of users navigating to another page from a previous page as the values in the UBG. Considerations when parsing log files:
         * Several users may share the same IP (e.g. users behind a proxy).
         * A user may navigate with two or more open browsers.
         * A user may wait several minutes between sending requests. A threshold needs to be defined for when a session terminates.

We can use a clustering algorithm to do this.

* + 1. False. For example, some users may navigate from login to home to sign out, because they were previously logged in, however only users who have previously logged in could do this. The UBG does not allow for distinguishing between users who have previously logged in.

Solution with answer = 3.3125



* + 1. Very not sure about this one, seems to easy:

First, we take the mean session length and use that to get the session service time for the pages. In other words the answer from above times 1.0 ms. The home page requires an additional 3.2 ms(2 calls to the database each 1.6ms) We need to add this in. We can find how many times on average the homepage is visited using similar reasoning as above. Then we do 1/time per session, to get the maximum throughput.

Potential answer following the above method:

1.5625 \* 4.2 \* 10^-3 + (3.3125 - 1.5625) \* 10^-3 = 8.3125 \* 10^-3 s

Therefore throughput = floor(1/(8.3125 \* 10^-3)) = 120 requests per second

**2021 – NEW**

What we need to do is take the formula for finding the utilisation of a resource, set it equal to 1, and determine what request rate will maximise that resource usage.

Utilisation = Sum over service classes (Demand \* Request rate)

Let’s start with the web server

1 = lambda\_h \* dh + lambda\_r \* dr + lambda\_p \* dp + lambda\_t \* dt

All pages induce the same demand on the web server (1.0ms).

Additionally, lambda for a class = overall requests \* visit odds for class

1 = lambda \* .001 \* (vh + vr + vp + vt)

1 = lambda \* 0.0033125

Lambda = 301.887

Maximum request rate for web server = 301.887 requests/sec

Now database

Only the home page induces a demand on the database (3.2ms)

1 = lambda \* .0032 \* vh

1 = lambda \* .005

Lambda = 200

Maximum request rate for database = 200 requests/sec

Therefore, the database will bottleneck before the web server so the overall maximum session throughput is 200 requests/sec.

* + 1. I would find all the paths of length n, and then multiply the transition probabilities for those paths, and add up all the path’s multiplied probabilities. There may be some way you’re actually supposed to do this but idk what that is.

**Way from the slides:** p is a vector with probabilities for being in a given state. Then p0 = (1,0,0,0,0,0), the probability of being at each state after 0 steps. Then pn = p0Pn where Pn is the transition matrix to the nth power i.e the probabilities of being at each state after n steps. So we want to check pn at the last index, which corresponds to the probability of being at state X.

* 1. Was this covered? I didn’t go to all the lectures? Seems like this is a case study or something

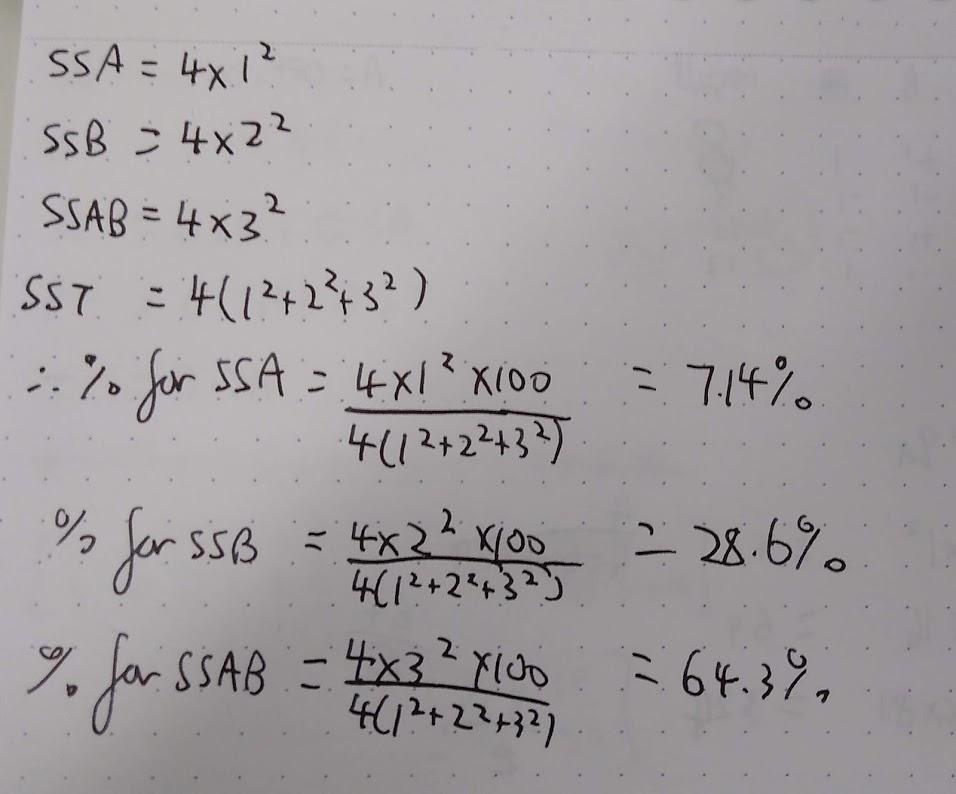
1. * 1. Sign table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I | A | B | AB | Y |
| 1 | -1 | -1 | 1 | 12 |
| 1 | -1 | 1 | -1 | 10 |
| 1 | 1 | -1 | -1 | 8 |
| 1 | 1 | 1 | 1 | 18 |
| 12 | 1 | 2 | 3 |  |

Qualitatively, we can see that there is a lot of interaction between the factors A and B, so the factor AB in the table is the largest

* + 1. SST = 56  
       SSA = 4 7.19%  
       SSB = 16 28.57%  
       SSAB = 36 64.29%  
         
       Factor AB accounts for 64.29% of variance.

Alternative answer (I think there were some mathematical errors above):



* + 1. Sign table is just the basic 23 sign table with BC replaced by D. Using D=BC we get C = DB, B = DC and ABC = AD, which gives us a resolution 3 design.

More confoundings:

AD = ABC

CD = B

BD = C

ABD = AC

ACD = AB

ABCD = A

I = BCD

* 1. max. 4l1+7l2+l3 st. 10l1+5l2+9l3 <= 1  
      4l1+7l2+l3 <= 1  
      3l1+0l2+12l3 <= 1  
      0l1+10l2+0l3 <= 1

L1, l2, l3 >= 0

The optimal value is 1 iff the resource B can be saturated for some combination of arrival rates l.

* + 1. We set a rule for our system like the following: If the current throughput is >= 80% of the maximum throughput over a period of 90 seconds, then begin the scale up/out process. If the current throughput is < 40% of the maximum throughput over a period of 90 seconds, begin the scale down process. While scaling up, apply throttling until the scale-out is completed.

In general

if V > V\_up for T\_up seconds then

scale out by adding N\_up nodes

do nothing for S\_up seconds

else if V < V\_dn for T\_dn seconds then

scale down by removing N\_dn nodes

do nothing for S\_dn seconds

end if

* + 1. Just apply the formulas: K1=V phi => phi = 0.4

V = sigma2 / (1 - phi2) => sigma = sqrt(1.68)

u = c / (1 - phi) => c = 0.6

1. 1. False sharing occurs when two different cores are both writing to a single cache line, but not to the same value. This causes contention over that cache line which translates into lots of intercore communication, in order to keep the cache coherent. This is a problem because it slows down the program even though the cores don’t share any data. There was a piazza question about this see there for more info.
   2. 1. Snippet 1: s\_tra on A and B in parallel, specifically sizeof(int) , inputSize time. Along with writes to C.
      2. Snippet 2: s\_tra on A and B in parallel, specifically sizeof(Point) , inputSize time, with u = 2\*sizeof(int). Along with writes to C.
      3. Snippet 3: s\_tra on A and B in parallel, specifically sizeof(Point) , inputSize time, with u = 2\*sizeof(int). Along with writes to C, and complicated access patterns to the stack.

mathsy answers:

i. S\_trav(r.w = 1, u = 1, r.n = inputSize) ⊙ S\_trav(r.w = 1, u = 1, r.n = inputSize) ⊙ S\_trav(r.w = 1, u = 1, r.n = inputSize)

ii. S\_trav(r.w = 16, u = 3, r.n = inputSize)

iii. Same as i I guess, ew? + jump pain of glory

* 1. Snippet 1 is similar to a dot product but with an extra branch. Snippet 2 is similar to 1, but the memory layout is much less dense, so we could expect it to be more memory bound than snippet 1. Snippet 3 is best described as a recursive version of 1. We would expect more overhead than 1. (though irl I imagine the compiler would optimize to the same thing.) Profile 1 best matches Snippet 2, since it is more frequently memory bound. Profile 3 best matches the recursive snippet 3 , while Profile 2 matches snippet 1.

^ to add to the above, profile 3 is snippet 3 due to recursive calls = jumps => front-end bound, which is worst in profile 3.

* 1. JITed code can make observations about what it is executing, for example a JVM could observe that a particular reference is never null and avoid performing null checks on that reference. This means that the program is only partially evaluated but in practice has the same behaviour. One disadvantage of this approach is that the unexpected case still needs to be handled. In the case of avoiding a null check the unexpected case will get raised as a pagefault, which then has to be handled. This is likely very expensive. Another disadvantage of JIT’ing is that the cost of compiling is incurred repeatedly and during the execution of the program. This makes the program initially slower than a precompiled program.

When you compile code normally, the compiler has access to the entire source code, and is able to make certain optimisations as a result. The JiT compiler does not know how far to look ahead in order to apply the optimisation without having too much of a cost.

1. 1. Perturbation is when the act of profiling a program causes the program to no longer perform as it would when it is not being profiled. Since we care about performance of the program when it is not being profiled the changed performance and characteristics of the program can be problematic. For example a profiler could insert small pauses into a program during which it collects information about the current stack. In doing this it accidentally evicts cache entries(since querying the stack reads in new entries). This then causes a performance degradation that would not otherwise have happened.

How to reduce:

* + - * Avoid manual instrumentation
      * Make the performance measurements less frequently
      * Tell the compiler to not use the cache for the measurement code
      * If the code you’re benchmarking is single-threaded, run the benchmarking code in a separate thread
  1. The top level if will be taken less and less as s increases, until s reaches 200, at which point it will no longer be taken. When s is less than 0 it will always be taken. The if within the top level if is taken more and more as s increases, with the same boundary conditions. The if within the else is also taken more and more as s increases. The innermost if statements are slower on average than the outermost statements because a is sorted and is therefore much easier for the branch predictor to predict. As s increases I expect the overall runtime to go down, since every iteration contains at least one difficult to predict branch, but a decreasing number of additions. Predication could improve the performance of this loop by getting rid of the difficult to predict branches. Improvements: if check on s before doing any looping - can avoid having branches at all in loop when s < 0 or s > 200. Would SoAs help for 0 < s < 200 because we usually access more than one of a, b, c, d, e?  
     Can make branch free (Can make both nested ifs branch-free but since a is sorted we only need to refactor inner loop):  
     count += (b[i] < s) \* c[i];
  2. Input2 is sorted because in the first graph once the data being accessed in input1 no longer fits in cache the number of cache misses does not continue climbing. This indicates that input1 is being accessed in a linear manner such that there are a limited number of cache misses. In the second graph the cache misses continue to increase after the modulus part of the input2 no longer fits in the cache, which indicates access patterns are random, resulting in ever increasing cache misses.
  3. One advantage of reusing floating point registers is that it can eliminate the need for transferring some data into the SIMD registers, since the desired data is already in the register. Another advantage is that the reduced number of registers allows for a faster CPU, with less wasted silicon. One disadvantage of this is that the programmer/compiler needs to save/restore register values if they are mixing floating point operations with SIMD operations. This could be rather expensive and make some vectorization slower than the traditional implementation.