Your Name:  

SID Number:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
[ This page left for π ]

3.141592653589793238462643383279502884197169399375105820974945
Question #1: Prediction

1a) Why does branch prediction work?

*Machine code tends to exhibit regularity (e.g. branch instructions that run for loops) that can be taken advantage of in the processor. Machine code is an inefficient representation of programs and thus it exhibits all the signs of a weak encode (such as predictability).*

1b) What is the aliasing problem for branch predictors? Is aliasing always bad?

*Branch tables have limited sizes (essentially a cache of all history of all branches). Thus multiple branches map onto the same entry. Since tags are not stored to identify these conflicts and clear out the old entries, aliasing occurs.*

*No, some aliasing can be beneficial if aliased branches exhibit similar behaviour (can improve cold start behaviour).*

1c) How does the branch target buffer (BTB) help modern branch predictors? (*hint: we want to be able to remove all branch delay slots):*

*Even if branch prediction predicts successfully that a branch will be taken, the branch address needs to be calculated and sent to the I$. Instead of stalling on all taken branches, the branch targets can be “cached” in the BTB so that the new PC is immediately available and so no stalling / branch delay slots are needed (except in the case where the calculated new PC mismatches with the new PC from the BTB).*

1d) Draw the hardware for a gshare branch predictor. Why does this type of predictor generally perform better than an equivalent GAg branch predictor?

*Aliasing is reduced due to the xor hash, and since there tends to be more bad aliasing than good aliasing, the predictor does better.*
1e) For correct performance, out-of-order processors must detect and resolve RAW, WAR, and WAW hazards between loads and stores to memory. Describe the hardware support for detecting these hazards and provide a short, pseudo-code description of the questions that must be asked before a load or store is released to the memory system. Assume no dependence speculation for the moment (conservative removal of hazards).

The hardware support that is needed is a load-store buffer with one entry for each load and store in flight stored in program order which contains the address of the data location and a pointer to its ROB entry or functional unit.

LOAD:
Add entry to table.
Wait for address to be computed.
If a store that is older in the table has an uncomputed address, wait.
If a store that is older in the table has the same address, just get that data value (youngest older store value, might have to check coherence tags), exit.
Go to memory.

STORE:
Add entry to table.
Wait for address to be computed.
If a store that is older in the table has an uncomputed address, wait.
If any store that is older in the table has the same address, wait until that store has gone to the memory or write buffer.
If any younger loads are waiting on uncomputed store addresses, and this is the only uncomputed store address younger than that load, wake it up.
Go to memory.

If) “Naive dependence speculation” assumes that loads and stores are not dependent on each other, if their addresses are unknown. How does this change your algorithm above? On average, is this a better idea than being exact (as in 1e)? Why or why not?

Remove the stalls on unresolved addressed in the load case, check for address conflicts when the load returns or retires.

Better because most load store pairs are not dependent and because of the long latency of memory operations, you have cycles of execution to waste during which address conflicts are not known.
1g) What is memory dependence prediction and why does it help to improve performance in an modern processor with-out-of-order execution? (hint: compare with naive dependence speculation)?

Predict that a given load/store will cause and address conflict with another load/store (see store sets paper). Helps to improve performance because it reduces mis-speculations and thus we spend fewer cycles in undoing mis-speculated work.

1h) What hardware might be used to detect and predict repeating patterns of four or less data values (e.g. something like: 4, 5, 6, 2, 4, 5, 6, 2, etc...)? Assume that these are 32-bit values.

Keep a history of accesses. One way to predict: use an associative table which maps triples of values to a small number of pairs of (predicted values and counter). Since a triple of values is 32x3 bits, this must be an associative hardware table.

To predict, take a set of 3 values and look it up in the table to find the most frequent follow on (with largest counter). To update, take four values. Look up by the first 3.

1) if the triple exists, and the 4th value is one of the entries, increment its counter.
2) if the triple exists, and the 4th value doesn’t exist, throw out the value with the lowest counter, and enter the new value with a count of 0.
3) if the triple doesn’t exist, you need to throw out a complete row of the table to enter the new triple. One heuristic might be to toss out the entry that has the smallest maximum counter value...

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4,5,6</td>
<td>2</td>
<td>count</td>
<td>3</td>
<td>count</td>
</tr>
<tr>
<td>4,5,9</td>
<td>0</td>
<td>count</td>
<td>2</td>
<td>count</td>
</tr>
<tr>
<td>5,6,2</td>
<td>4</td>
<td>count</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1i) How might data prediction be used in a modern processor? Under what circumstances might this improve performance (try to be specific, i.e. don’t just say “when it is correct”)?

It can be used to speculate on the results of long latency instructions (such as loads, FP ops) so that execution can continue.

If non-blocking caches are used and a reorder buffer is available, the load values of $-miss reads can be speculated on and then speculative execution can follow. Overall performance will be improved if (correct_spec_rate * cycles_saved) > (miss_spec_rate * cycle_cost_of_miss_spec). Loading of initial/empty array values and repetitive FP calculations can be predicted more easily.
Problem #2: Error Correction and RAID

The error correction coding process can be viewed as a transformation from one space of bits (the unencoded data) to another (the coded data).

2a) In the \((n,k)\) notation for an error correction code, \(k\) is the unencoded data width in bits and \(n\) is the encoded width. Which is larger, \(n\) or \(k\)? Why must it be this way?

“\(n\)” is bigger. We are adding redundancy, there must be more bits after coding than before. Alternatively, if \(n<k\), then we would be losing information during coding (some pairs of input words would be mapped to the same output word!)

2b) Define the minimum Hamming distance, \(d_{\text{min}}\), of an error correction code:

The minimum distance is the minimum number of bits difference between any two code words. For linear codes, this is also the minimum number of ones in a non-zero code word.

2c) For a code with \(d_{\text{min}}\) minimum distance, what is the formula for the maximum number of errors, \(E_{\text{detect}}\), that can be detected?

\[
E_{\text{detect}} = \frac{d_{\text{min}} - 1}{2}
\]

2d) For a code with \(d_{\text{min}}\) minimum distance, what is the formula for the maximum number of errors, \(E_{\text{correct}}\), that can be corrected? (We called this “t” in class)?

\[
E_{\text{correct}} = \left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor
\]

2e) Can you simultaneously detect \(E_{\text{detect}}\) errors and correct \(E_{\text{correct}}\) errors? Why or why not?

If your answer is “yes”, do the errors detected overlap with those corrected? If your answer is “no”, what is the maximum simultaneous values of both of these?

No. When you are correcting, you attempt to discover which code-word is closest to the received word. The problem is that you can’t really tell the difference between 1 error and \(d_{\text{min}}-1\) errors. So, if you try to correct up to \(E_{\text{correct}}\) errors and more than \(\left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor\) errors occur, you will actually correct them incorrectly (i.e. you haven’t really corrected them). So, if working simultaneously, max combined values:

\[
E_{\text{correct}} = \left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor, \ E_{\text{detect}} = \left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor
\]
2f) One common type of linear code that is used in memory systems is the even parity code. This code adds one additional bit to each byte (for a total of 9 bits) so that the number of 1s in the result is even. What is the minimum distance of the (9,8) bit parity code? What are \( E_{detect} \) and \( E_{correct} \) for this code? [Amusing side note: the odd parity version is not linear!]

\[
d_{min} = 2. \quad \text{Whenever you change 1 bit in the original data, you change bits in the code (or, since there are always an even number of bits in a code word, the minimum number of differing bits between two code words must be even and clearly you can find two words which differ by 2 bits).}\]

\[
E_{detect} = 1, \quad E_{correct} = 0
\]

2g) What is the \( H \) matrix for checking the (9,8) parity code?

Simple – it is an “\( n \)” by “\( n-k \)” or 9 x 1 matrix. \( H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \)

2h) Suppose that we have 9-bit drams (in fact, some of the RAMBUS DRAMS are like this). If we put 8 of them together we would have enough bits for 64 data bits and 8 parity bits. Suppose we code each group of 8 bits with a single parity bit. The result could be called a (72, 64) code. What is its minimum distance of this code and why?

Min distance still = 2. Hold 7 of the code words constant and vary bits in one of them. You can still find two such code words which differ by 2 bits. This has the same minimum distance properties as a single parity code.

2i) You could produce a “better” (72,64) code which has a higher minimum distance. What would be the “maximum” minimum distance and how can you prove this (hint: give a qualitative argument about the \( H \) matrix without actually constructing it)? Can you give an “intuitive” explanation for why this code is better?

Maximum \( d_{min} = 4. \) “Proof:”

a) this code exists – with 8 parity bits, have 256 words to chose from in the \( H \) matrix. Take only the odd weight ones. Still 128. Enough for 72 columns of \( H \) matrix. This has distance 4 for reasons discussed in class (you also read a paper on such codes).

b) Can we make a dist 5 code? Need any combination of 4 or less columns of \( H \) matrix to sum to something non-zero. So, need to through out any additional words from remaining 128 odd codes that might combine in sum of 4 to make 0. Clearly, all weight-3 words must go, since they can be combined with 3 of the columns from the identity matrix. There are 56 of these, leaving only 72 left. However, consider two 5-weight words which have 4 of the same ones. These can be combined with two weight-1 words to make zero. So, can’t have dist 5 or more with only 8 parity bits.

This code is “better” than the code in 2h, since is spreads information across all 8 parity bits. This is why we can actually identify **and** correct 1.

The other way to know this is to ask what happens if we try to correct 2 errors in 64 bits (i.e. \( d_{min} = 5 \)). Then we would need \( 2^{n-k} \geq \sum_{t=0}^{2} \binom{64}{t} = 2081 \), which we don’t have...
2j) RAID level-5 is formed by xor-ing a number of data blocks together to produce a single parity block. Thinking of this code as grouping all of the first bits of every disk block together, then all of the second bits, etc, we see that this code is an extreme version of (2h) with 4K parity-encoded “bytes” concatenated together. What is its minimum distance of this code and why?

Dist still = 2. Why? For the same reason as in 2h. Keep all bits in all blocks constant except for the first bit of the first word. Then if we change one of these first bits, we must also change another one because of the XORing. So, the minimum distance between two code words is 2.

2k) Note that RAID level-5 is able to correct 1 bad disk block (i.e. 1 error). Is that consistent with your answer in (2j)? If not, how do you explain the inconsistency?

It is not consistent if you view this code as correcting random errors. The inconsistency comes from the fact that the disks themselves know when blocks are bad. Thus, we can treat the RAID combination as an erasure channel. Thus, rather than correcting

\[ E_{\text{correct}} = \left\lceil \frac{d_{\text{min}} - 1}{2} \right\rceil = 0 \text{ errors}, \]

we can correct \( D_{\text{min}} - 1 = 1 \) error.

2l) Suppose that our original RAID level-5 was formed by xoring 8 data blocks together to produce one parity block. Could we use the code in (2i) to get a larger minimum distance for our RAID code? What would this minimum distance be?

Only with extreme care. The obvious thing doesn’t actually work: i.e. if we combined the first byte of each block together to produce the first byte of the parity block, the second bytes, etc.... This would seem to be a (72, 64) code on each byte. However, when we lose a block, we loose 8 bits at a time. So, the minimum distance would not be better.

To use the code in (2i), we would have to combine 64 blocks to produce 72 (one bit at a time). If we used 72 disks, we could have a minimum distance = 4 code.

2m) Finally, taking into account your answer to (2k), how many disk block failures could this new code correct (with the same overhead)? Explain. (hint: it is larger). Why isn’t this code used in practice?

This could correct 3 disk errors (\( d_{\text{min}}-1 \)). This code is not used for several reasons. The simplest reason is that it is complicated. It requires breaking apart blocks one bit at a time, running the bits through an XOR function, then splitting the results back into the blocks. This causes way too much of a performance hit.
Problem #3: Grab bag
3a) What is the alias problem in a virtually indexed, virtually tagged cache?

Multiple virtual addresses can map to the same physical address and thus map to different cache lines. Writes and reads can go to different cache lines and thus coherency on that data will be lost.

3b) Name 2 techniques for eliminating aliases in a virtually indexed cache.

- Page colouring if physically tagged.
- Separate physical index table with pointers to cache entries.
- Have OS guarantee that no 1 process will map >1 VA to 1 PA and flush $ on context switches.

3c) Name and define the four “C’s” for caches?

- Cold start misses – first time the cache tag has been seen.
- Conflict misses – misses that would not occur in a fully associative cache.
- Capacity misses – misses that are not conflict misses and that which would not occur in a bigger cache.
- Coherence misses – misses on lines that were present in the cache but were either knocked out or downgraded due to other processors sharing that data.

3d) Name and define three hardware techniques for reducing cache misses.

- Increase size of the cache – reduce capacity misses
- Increase associativity of cache – reduce conflict misses
- Vary block sizes – reduce conflict and capacity misses
- Improve coherence protocol – reduce coherence misses
- Stream buffers / hardware prefetching – predict memory usage, can reduce all 4 misses
3e) What does it mean for a multiprocessor to be sequentially consistent? Why is this a desirable requirement for a multiprocessor?

*In any execution of a multiprocessor environment, if the execution result will be the same as when all the memory accesses were interleaved in some sequential order without violating the inherent program order, then the multiprocessor is sequentially consistent.*

*Programmers expect the same consistency as in uniprocessor execution, thus it is easier to code for multiprocessors that are SC.*

3f) List the three conditions sufficient for sequential consistency that were presented in class.

1. All processors issue memory requests in order.
2. If a processor has issued a store, it will wait for the store to complete before it issues another request.
3. If a store is in flight, a processor will wait for the store to complete (be visible to all processors) before allowing a load to that same address to see the new value.

3g) Explain how a multiprocessor with a snoopy-cache protocol on an atomic bus can be made to be sequentially consistent (*hint: explain how each of the three conditions of (3f) are satisfied)*:

1. Each processor has a re-order buffer that retires memory operations in order (or each processor is an in-order processor).
2. If a cache is servicing a write miss, the cache will send out no other local processor requests (it especially can’t on an atomic bus).
3. Atomic bus – only one transaction can be in flight – when one processor sees a store value, all others will see it too.
Problem #4: To Queue or Not to Queue

Computer architects are beginning to consider the use of generic networks as a replacement for buses in modern computer systems. Suppose that we have a multiprocessor which consists of N different processors connected through a general network to a single DRAM “server”. Suppose that the computation on each of these processors have the following average behavior, and that each of the individual processors are completely uncorrelated (so that the combined behavior appears memoryless):

Instruction Mix: 20% loads, 10% stores, 20% floating-point, 20% integer, 30% branches
Base CPI without stalls: 1 (i.e. this is a single-issue processor). Average additional integer stalls due to hazards: 0.2 cycles/instruction. Average floating-point stalls due to hazards: 3.0 cycles/instruction. Branches mispredict 10% of the time with a cost of 4 cycles.

Now, assume that each processor has a single-level of cache, with separate I and D caches. Assume a write-back cache, 4-word cache lines, a 99.5% hit rate on instructions, a 90% hit rate on loads and a 80% hit rate on stores. Cache hits happen at 1 per cycle with no stalls. Also, assume a write-allocate policy, and that 12.5% of the cache lines in the data cache are dirty and must be written back to memory on a miss. Assume no self-modifying code.

Assume that the network is one word (32-bits) wide and that it operates at half the processor speed. A one-way network trip to or from the DRAM takes 8 processor cycles. As far as the DRAM is concerned, assume that it can handle only one request at a time and that it has a service time of 10 processor cycles for the first word + 2 processor cycles for each additional word.

4a) Assume no queueing in the network or DRAM (only one request outstanding at a time). The network and cache are pipelined for fills, however. Also, assume a single processor with a blocking cache. Assume that the time for sending data into the network or taking data from the network must be accounted for (i.e. when filling or replacing data in the cache). What is the average memory access time for instructions? For Data? How about the overall average memory access time?

There are many ways to interpret this problem; we allowed many of them. Further, in grading the problem, we gave you points for being consistent in later stages of the problem, even when you had problems with an early part of the problem. Here is one solution:

*Time for read miss = 8 + 10 + 8 = 26 (for first word) + 6 (for remaining words) = 32*

*Time for dirty flush = 8 cycles (merely the time to get data into the network)*

\[
AMAT_I = 1 + (0.005 \times 32) = 1.16.
\]

\[
AMAT_D = 1 + \left[0.2 \times 0.1 + 0.1 \times 0.2\right]/0.3 \times (32 + 0.125 \times 8) = 5.4
\]

*Note on AMAT_D: must scale instruction frequencies by total load/store frequency (0.3). Also, the total cost of a miss is (32 + 0.125 \times 8). This is 32 for reading data and an extra 8 in the 12.5 percent of the time that we need to dump data from the cache. This average miss penalty is try for both reads and writes because we have write-allocate*

\[
AMAT_{TOTAL} = (AMAT_I + 0.3 \times AMAT_D)/1.3 = 2.138
\]
4b) What is the total CPI for the situation in (4a)?

\[ CPI = 1 + STALLS_{non-memory} + STALLS_{memory} \]
\[ = 1 + (0.2 \times 3 + 0.2 \times 0.2 + 0.3 \times 0.1 \times 4) + 1.3 \times (AMAT_{TOTAL} - 1) = 1 + 0.76 + 1.48 = 3.24 \]

Alternative way to compute \( STALLS_{memory} = STALLS_{data} + STALLS_{inst} \)
\[ = (0.1 \times 0.2 + 0.2 \times 0.1) \times (32 + 0.125 \times 8) + 0.005 \times 32 \]

4c) Again, assuming no queueing in the network or DRAM, what is the \( RATE \) (in accesses per processor cycle) at which requests are made to the DRAM?

\[ \text{Number of accesses / inst} = (0.1 \times 0.2 + 0.2 \times 0.1) \times 1.25 + 0.005 = 0.05 \text{ accesses/inst} \]

Note that the factor of 1.125 is from the fact that every cache miss causes 1 access for reading information and 0.125 accesses (on average) to flush a dirty line.

So, \( RATE = \text{accesses/cycle} = \text{accesses/inst} / CPI = 0.05 / 3.24 = 0.0154 \text{ accesses/cycle} \)

4d) What is the maximum rate at which the DRAM can service requests? (in accesses per processor cycle)?

To get one complete cache line, the DRAM takes \( 10 + 3 \times 2 = 16 \text{ cycles/access} \)
So, rate = \( 1/16 = 0.0625 \text{ accesses/cycle} \)

Note that some people tried to consider an access to be 1 word. Thus, they computed something like \( 16/4 = 4 \text{ cycles/access} \). The reason this is not correct is because all cache misses result in a full cache line being read or written to the DRAM.

4e) What is the DRAM utilization? Can this ever be greater than one without queueing? Why or why not?

\[ \text{Utilization} = \frac{\lambda}{\mu} = \frac{0.0154}{0.0625} = 0.2464 \]

This is not going to be greater than one because the processor is forced to wait for each access to complete before issuing another request. Thus, there is no way to issue instructions faster than the DRAM can service them, since the DRAM automatically pushes back on the processor.
4f) Next, assume that we have queueing of requests at the DRAM. Assume that we have a non-blocking cache that can issue many requests (unbounded) without stalling. Assume that the processor can make forward progress with an outstanding instruction miss (weird assumption, but makes math easier). Finally, assume a fully pipelined network, but not fully pipelined DRAM. Cache fills and replacements still must happen at network bandwidth, but now we can overlap the network time.

Question: What is the rate of requests into the network now? What is the DRAM utilization?

Mirroring 4b and c, we must compute the new CPI with a non-blocking architecture. There are several assumptions that we could make about what the impact of this is, but one possibility is that the only impact of memory operations is now the network bandwidth required to pull cache lines into and out of the network. This takes 2 cycles/word × 4 = 8 cycles for both fills and replacements.

New \( STALLS_{memory} = (0.2 \times 0.1 + 0.1 \times 0.2) \times 1.125 \times 8 + 0.005 \times 8 = 0.4 \)

So, \( CPI = 1 + 0.76 + 0.4 = 2.6 \)

Note that the number of accesses/instruction has not changed from 4c.

Rate = access/inst / CPI = 0.05/2.6 = 0.0231 accesses/cycle

Utilization = \( \lambda/\mu = 0.0231/0.0625 = 0.370 \)

4g) Now, assume that the processor requests are exponentially distributed. The DRAM service time is deterministic (i.e. not exponentially distributed). What is the average time that a request spends at the DRAM, including any queuing time?

\[
T_{system} = T_{queue} + T_{serve} = \left( \frac{\xi}{1 - \xi} \right) \times \left( \frac{1 + C}{2} \right) \times T_{serve} + T_{serve}
\]

Here, \( \xi \) is the utilization. Because the DRAM is a deterministic server, \( C = 0 \) here.
Note that to get the total time in the system, we need to add queuing time (first term) to average service time after exiting the queue (which is \( T_{serve} \)). Note also that \( T_{serve} = 16 \) for the DRAM.

Hence: \( T_{sys} = \left[ \left( \frac{0.370}{1 - 0.370} \right) \left( \frac{1}{2} \right) + 1 \right] \times 16 = 20.70 \)

4h) What is the average number of requests in the DRAM system? (hint: Little’s law may help)

Little’s law: \( L_{sys} = \lambda T_{sys} = 0.0231 \times 20.70 = 0.478 \)
4i) What is the average number of read requests that the processor will have outstanding at any one time (hint: consider the average time to service a request as seen by the processor)?

This is easy if you use Little’s law from the viewpoint of the processor.
The time that a cache miss takes is a network roundtrip + $T_{sys}$ from 4h
\[ T_{MISS} = 8 + 8 + 20.70 = 36.70 \]

We only want to look at the reads coming from the instruction stream (since, for instance, these will need to be queued until the results come back). So:

\[ \text{Read Misses/inst} = 0.2 \times 0.1 = 0.02; \ CPI = 2.16 \]
\[ \Rightarrow \text{Read Misses/cycle} = \text{Rate of readmisses} = 0.02/2.16 = 0.0093. \]

Average # outstanding (“in system”) = \( \text{Rate} \times \text{Time} = 0.0093 \times 36.7 = 0.241 \)

4j) Now, assume that we have multiple processors with non-blocking caches and queuing within the network and DRAM. Assume that the requests combine together to produce an aggregate request rate into the DRAM. What is the maximum number of processors that we can support without the DRAM request queue becoming infinitely long?

Need to avoid DRAM utilization > 1 to prevent unbounded queue requirements.

Since utilization = 0.370 and DRAM request rates add, we see that we can’t have more than two processors for a total utilization of $0.370 \times 2 = 0.740$

\( \text{(This is also easily done as: Maxprocessors} = \left| \frac{1}{\text{utilization}} \right| = \left| \frac{1}{0.370} \right| = 2 \) \)

4k) What is the average number of outstanding requests that each processor would have now?

Repeating 4h and 4j with new utilization:

\[ T_{sys} = \left[ \left( \frac{0.740}{1 - 0.740} \right) \times \frac{1}{2} + 1 \right] \times 16 = 38.77 \]
\[ T_{MISS} = 8 + 8 + 38.77 = 54.77 \]

Average # outstanding (“in system”) = \( \text{Rate} \times \text{Time} = 0.0093 \times 54.77 = 0.509 \text{ outstanding} \)
[This is another blank page for you to use]