

## P5: Build-a-Cache

100 Points Possible

12/4/2023

 Add Comment

11/13/2023

## ▼ Details

**FOR 2024 PEOPLE MAKING PREPARING THIS PROJECT:** Many people interpreted "explain your observation in this graph" as "described the graph" instead of "give a reason for why this would have occurred." Possibly add some phrase like "reference hardware" or say something like "explain why..." instead.

**Late Policy:** No slip days can be used for this project.

**Reminder:** This is a group project. You can see your group assignment on CMS. Do not show your work to any other groups or look at the work of anyone in the class. Other than the generative AI resource you have been given permission to use (Copilot) you must not consult any other AI or online code.

**You are not forced to use Copilot, but it is a resource available to you for this project. Copilot may not necessarily make this project easier, but rather be a new tool/skill you learn to use. Remember, Copilot is never guaranteed to generate correct code, and it is your job to vet its generated output. However, if you feel comfortable coding without it, do that!**

**ALERT:** Changes to the writeup have been made in green

Before you begin coding and running experiments, make sure you also read the **P5 Supplement** (<https://canvas.cornell.edu/courses/54972/pages/p5-supplement>) which is very helpful.

(<https://canvas.cornell.edu/courses/4522/quizzes/15746>)

## Overview






In this assignment, you will write a performance simulator: a C program that simulates the behavior of a cache. Software performance simulators are used to explore cache designs because they are faster to implement and modify than circuit-level simulators (like Logisim, for example). The simulator will be configurable so that it can model the behavior of a variety of cache configurations as well as multiple coherence protocols.

Your cache simulator will take as input a **memory trace**: a list of load and store instructions that were executed during a single run of a single program. We refer to each load and store instruction as a **memory reference**. You will use a single-core memory trace to evaluate the impact of cache size, associativity, block size, etc. on cache performance. Once you extend your simulator to model multiple caches, you will use a 4-core memory trace to evaluate the impact of different cache coherence protocols.

You will be asked to run a series of experiments with your cache simulator. You will write up a report of your experiments (including graphed results and questions answered).

**Heads up!** Last semester, students who filled out the survey reported the following (however they did not have Copilot access):

How long did you (personally) spend working on this project?

5-10 hours	74 respondents	28 %	
> 20 hours	48 respondents	18 %	
11-20 hours	134 respondents	51 %	
0-4 hours	4 respondents	2 %	
No Answer	1 respondent	0 %	

## The Trace

The memory trace you will be using was generated by running a program and recording all of the loads and stores issued by the processor during that run. What it's actually doing doesn't really matter for the purposes of this assignment. Normally you would not just look at one trace file, but for the purposes of CS 3410, one trace is sufficient. The trace file looks like this (trace.4t.long.txt):

```
0 r 1ce026e0
0 r 7d300000
2 w 026d7680
0 r 1ce024d0
3 r f6b3fca0
3 r f6b3fc98
2 r 2db23e10
3 r f6b3fca8
3 r f6b33178
1 w f7706ab8
...
```

The first field indicates the core number. In the first part of the project, you only need to model a single-threaded workload running on a single core, so this number will always be a 0. Later, cores 1, 2, and 3 will be present in the workload.

The second field is always **r** or **w**. An **r** indicates a load (read) and a **w** indicates a store (write).

The third field is the address of the memory operation (in hex). Taking a look at action variable in simulator.c will help you further understand the format of the trace file.

**Assume that addresses are 32 bits for this entire project.**

**For our purposes in CS 3410, a kilobyte (KB) is equivalent to a KiB, which is equal to 1024 ( $2^{10}$ ) bytes. A megabyte (MB) is equivalent to an MiB, which is 1,048,576 ( $2^{20}$ ) bytes.** For example, 32KB is equal to  $32 \cdot 2^{10} \text{B} = 2^5 \cdot 2^{10} \text{B} = 2^{15} \text{B}$  and 64MB is equal to  $64 \cdot 2^{20} \text{B} = 2^6 \cdot 2^{20} \text{B} = 2^{26} \text{B}$ .

## Part 1: Single Core Simulation

### Task 1: Cache Tag/Index/Offset [on paper]

A cache has three primary configuration parameters:

- **A:** Associativity (number of ways per set)
- **B:** Block size (size of a single block, a block is also referred to as a cache line)
- **C:** Capacity or cache size (total amount of data in the cache)

These three parameters determine the number of sets and ways in the cache. These parameters also specify how a memory reference address is split into its three components: tag, index, and offset.

## Run through an example

**Refresh your memory:** calculate the following values for a two-way set associative 32KB cache with 64B blocks (that is:  $A=2$ ,  $B=64B$ ,  $C=32KB$ ):

- Number of offset bits in the address
- Number of sets in the cache
- Total number of cache lines in the cache
- Number of index bits in the address
- Number of tag bits in the address

There is no submission associated with this task, but for the sake of your understanding this assignment, please do this!

## Generalize the Calculations

Write formulas in terms of  $A$ ,  $B$ , and  $C$  for calculating the following quantities. You may find using  $\log(A)$ ,  $\log(B)$ , and  $\log(C)$  in the formulas easier than  $A$ ,  $B$ , and  $C$  directly. For this assignment, you may assume  $A$ ,  $B$ , and  $C$  are all powers of 2. Write the formulas for:

- Number of offset bits in the address
- Number of sets in the cache
- Total number of cache lines in the cache
- Number of index bits in the address
- Number of tag bits in the address

Hint: you may find it helpful to first work out the numbers for a direct-mapped cache (that is, consider just  $B$  and  $C$ ), and then extend it to set-associative caches (in which multiple cache lines reside in a single set).

There is no submission associated with this task, but for the love of caches, please do this!

## Task 2: Initialize your Cache

(Here's when you start writing code and start using Copilot)

Now use your formulas from Task 1 to complete 5 lines of the `make_cache` function in `cache.c`. You need to set the variables for `n_offset_bit`, `n_set`, `n_cache_line`, `n_index_bit`, and `n_tag_bit`.

When Copilot autocompletes this code for you, make sure you compare it with your answers from Task 1. It's possible Copilot has made incorrect assumptions regarding address size or cache size or variable names, and it's your responsibility to edit this as you proceed in this assignment to make sure the errors don't cascade.

Notice there is a `log2` function in the included `<math.h>` library.

Because we're not manipulating the actual data values (as a real processor would), but only calculating hits and misses for each memory reference, there is no need to store data values in the cache. (In fact, the trace doesn't even give you the data values).

The cache can be represented by a 2-dimensional array of lines containing only the metadata: tags, dirty bits, and state. (The second dimension only becomes relevant when the associativity of the cache is higher than 1.) Finish the `make_cache` function by creating the array of lines that will be used to determine whether a cache hit or miss has occurred.

Initialize all tag values to zero, dirty bit flags to false, and state to `INVALID`. The state won't play a big role until you start implementing coherence protocols, but it's a good idea to start incorporating them into your code.

You need to call `make_cache_stats()` from inside `make_cache()` in `cache.c`. This should already be done for you, but think about why this call is important.

## Task 3: Extract Tag/Index/Offset from Addresses

Continue coding up your `cache.c` by completing these three functions used to extract the tag bits, index bits, and block address from any given address:

- `get_cache_tag()`
- `get_cache_index()`
- `get_cache_block_addr()`

Please see the function headers for function specifications and example behavior. Use these headers to your advantage in generating code with Copilot. Again, make sure you vet the code generated, as it may not make the right assumptions.

**IMPORTANT:** Do *not* convert the integer values to character strings (which is slow and unnecessarily verbose). Integer bit manipulation instructions such as shifts, bitwise AND, OR, and NOT, can be used to easily and efficiently accomplish such bit extraction operations.

Hint #1: to generate an integer value with all bits set to one, you can use `~0` (the bit-wise NOT of zero).

Hint #2: shifting your address in both directions may prove useful.

Hint #3: If you're looking to get the right AI generated code, it's helpful to give examples of input and output for a function to get the right functionality. Make sure to give examples that cover specific cases and help Copilot generalize.

Hint #4: If you're confused about what code the AI has written and whether its correct, it'll be helpful to make separate test functions for these individual functions.

The autograder will call these three functions twice, beginning with the address `0b111101010001` and print out the values for these variables. Note: they will not match the examples in the function headers because the tag, offset, and index bits will not each be 4 bits long (**the addresses in the simulator are all 32 bits**). Your solution should be able to handle any valid setting of tag, offset, and index bits.

## Completing the Access Function

The most challenging part of the assignment will be to write the access function `access_cache`, which performs the cache lookup for each load or store. It returns true if the access is a hit, and false if the access results in a miss. Read the rest of the assignment and the code you have been given to correctly implement the access function. Do not try to implement the entire function at once. Instead, build on the function as each new concept is introduced below. Every time you change your code, re-run a test that used to work and make sure that as you create new functionality, you don't break something that used to work.

## Task 4: Implement a Direct-Mapped Cache to Explore Hit Rate vs. Capacity

Since these tasks are incremental, make sure you clearly mention in comments the feature that you're focusing to implement (in this task, direct mapped single core cache) so the AI doesn't immediately try to jump the gun in more

complex implementations. This is both for your iterative understanding of the code you and AI write; AI will **almost certainly** not generate complex/long code correctly. **Use Copilot to generate small snippets of code.**

Complete the access function for a direct-mapped cache by following the steps described in the function's comment (you can ignore `lru_way` and `dirty_f` for now). Your simulator can now simulate a direct-mapped cache parameterized by B and C. Let's test it out!

For now, handle loads and stores in the same fashion. In the next task, you will set `dirty_f` depending on whether or not it is a store. Additionally, if there is a tag match, then it is a hit, regardless of whether it was a load or a store.

### Experiment 1A: impact of cache size on miss rate for a direct-mapped cache

Use your cache simulator to produce cache miss rates for varying cache sizes. Generate the data for caches capacity from 1024 bytes ( $2^{10}$ ) to 1MB ( $2^{20}$ ). Configure the block size to 64 bytes. Below are the first two commands you should run. The first sets the cache capacity to  $2^{10} = 1024$  bytes and the block size to  $2^6 = 64$  bytes. The second increases the cache capacity to  $2^{11} = 2048$  bytes.

```
./p5 -t trace.1t.long.txt -cache 10 6 1
./p5 -t trace.1t.long.txt -cache 11 6 1
```

- Associativity = 1
- Block size = 64B
- Capacity = 2KB, 4KB, ..., 1MB

### Graph #1: Miss Rate vs Cache Size

Generate a line plot of this data. On the y-axis, plot the "cache miss rate (percent of all memory references)". The smaller the miss rate, the better. On the x-axis, plot the log of the cache size, in essence giving us a log scale on the x-axis. (Heads up: later we will ask you to add data from Experiment 1B to this graph.)

### Questions

1. What is the smallest capacity that brings the miss rate to less than 10%?
2. What is the total memory requirement (i.e. maximum memory needed) for this address space? For this RAM size and your answer to question 1, what is the ratio of cache to memory size?
3. Today's processors generally have 32KB to 128KB first-level (L1) data caches. By what ratio does increasing the cache size from 16KB to 32KB reduce the miss rate? (2.0 would correspond to halving the miss rate; 1.0 would correspond to no change in miss rate; less than 1.0 would correspond to an increase in misses).
4. By what ratio does increasing the cache size from 32KB to 64KB reduce the miss rate?
5. When deciding on the ideal cache size, engineers typically look for the **"knee" of the curve**. [↗ \(https://en.wikipedia.org/wiki/Knee\\_of\\_a\\_curve\)](https://en.wikipedia.org/wiki/Knee_of_a_curve) When considering various cache sizes, we want the point at which increasing *to* that size yields a great benefit, but increasing *beyond* that size yields far less benefit. What would you say is the ideal cache size for a direct-mapped cache?

## Task 5: Implement Set Associativity to Explore Hit Rate vs. Associativity

**Before moving on:** create a copy of your working cache up to this point. Run the following command:

```
> cp cache.c cache_direct_mapped.c
```

Ultimately, your final `cache.c` will be a super-set of `cache_direct_mapped.c`, but you may find it helpful to have this working snapshot to refer to as you evolve your cache simulator.

Now modify your simulator to support set associative caches. Each set in the cache will now need an **array** of cache lines (one for each way), and a single "LRU" (least recently used) field used for cache replacement. Note: you will not be implementing *true* LRU (which would require a counter per way and is not practical). Instead, you will implement an approximation of LRU (you might want to think of it as NMRU "not most recently used").

- direct mapped: LRU field is always 0
- 2 way set associative: LRU field always identifies to the way you *didn't* just touch
- > 2 way set associative: LRU field always identifies the way 1 higher (with wrap-around) than the way you just touched. Examples:
  - if there are 4 ways, and you touch way 0, then the new LRU way should be way 1
  - if there are 8 ways, and you touch way 7, the new LRU way should be 0

When replacing a block, update the LRU field (and don't forget to update the LRU field to point to the other entry). You should also update the LRU entry after a hit because a cache hit counts as a "use" of that cache line.

### Experiment 1B: impact of cache size on miss rate with increased associativity

Generate miss rate data for the same block size and caches sizes as in the previous question, but simulate 2-way and 4-way set-associative caches. Plot this result with the original data collected for a direct-mapped cache (three lines on the graph: one for direct-mapped cache and the other two for the 2- and 4-way set-associative cache).

- **Associativity = 2,4**
- Block size = 64B
- Capacity = 2KB, 4KB, ..., 1MB

### Graph #1 (continued)

Add the data from Experiment 1B to your Experiment 1 Graph. You will submit a single graph with 3 lines from Experiments 1A & 1B.

### Questions

6. What is the smallest capacity that brings the miss rate of the 2-way set associative cache to less than 10%?
7. What is the smallest capacity that brings the miss rate of the 4-way set associative cache to less than 10%?
8. How large must the direct-mapped cache be before it equals or exceeds the performance of the 8 KB 4-way associative?

*Because the set-associative cache generally performs better than the direct-mapped cache, **all subsequent experiments use the 4-way set associative cache.***

## Task 6: Explore the Memory Traffic of Write-Back Caches

Have your simulator produce the 3 statistics associated with memory traffic in `cache_stats.c`:

**B\_bus\_to\_cache**, **B\_cache\_to\_bus\_wb**, and **B\_total\_traffic\_wb**. We will also model write-through caches in Task 7!

Traffic *into* the cache (**B\_bus\_to\_cache**) is easy to calculate: it is just the number of misses multiplied by the block size because for each miss you must bring into the cache that entire block of memory. Traffic *out of* the cache is write traffic. Your simulator can calculate the traffic for write-back caches in a simulation as follows:

- For the write-back cache system, the traffic out of the cache (**B\_cache\_to\_bus\_wb**) is the number of writebacks multiplied by the block size. Because at most one writeback happens per miss, this write-back traffic can at most

double the overall traffic (in the case in which all evicted blocks were dirty). To calculate the number of writebacks, modify your simulator to update the "dirty" bit for each block in the cache. Set the dirty bit whenever a store operation writes to the block. This dirty bit is not per word or per byte; it is one dirty bit for the entire cache line.

Lastly, compute and store the total traffic for your write-back cache in the `cache_stats.c` variable **B\_total\_traffic\_wb**.

Use a write-allocate policy, meaning if a write misses in the cache, that cache line should be brought into the cache.

For a 1-, 2-, and 4-way set-associative cache with 64-byte blocks, generate a graph plotting the same cache sizes as the previous question (on the x-axis) versus number of bytes written from cache to the bus.

### Experiment 2: impact of cache size with increased associativity on cache to bus traffic

- **Associativity = 1, 2, 4**
- Block size = 64B
- Capacity = 2KB, 4KB, ..., 1MB

### Graph #2: Bus Write Traffic vs Cache Size for Write-Back Caches

The x-axis of this graph should be the same as Graph #1 (**in log scale**). The y-axis shows traffic in bytes, again, **in log scale**. Plot 3 lines: bytes written to the bus for direct-mapped and 2- and 4-way set associative caches.

### Questions

(original q9 is now q11 in Task 7 section)

9. What happens to traffic as the capacity increases? Is there a pattern? If so, explain what you see.
10. What role does associativity play in traffic generation? Is there a pattern? If so, explain what you see.

## Task 7: Write-Thru Statistics

Have your simulator produce 2 more statistics - **B\_cache\_to\_bus\_wt** and **B\_total\_traffic\_wt**. These are hypothetical statistics if the cache was a **write-thru** cache instead of a write-back cache.

In a write-thru cache, dirty bits are not used - any write operations pull that word (and therefore block) to the cache, in addition, that word is immediately written back to memory.

Since each word is 4 bytes, **number of bytes written cache to bus in write-thru is 4 times the number of store operations encountered in the trace**. Note that this therefore has no dependency on cache parameters - block size, associativity.

### Experiment 3

Use the simulator to calculate the **bytes written from cache to bus for a write-back** and **write-thru** cache in bytes (**B\_cache\_to\_bus\_wb** and **B\_cache\_to\_bus\_wt**). **details:** for a 64KB, four-way set associative with several different block sizes: 2B, 4B, 8B, 16B, 32B, 64B, 128B, 256B, 512B, 1 KB, 2 KB, 4 KB, 8 KB blocks.

- Associativity = 4
- Block size = 2B, 4B, ... , 8KB
- Capacity = 64KB



### Graph #3: Traffic in Bytes vs Block Size

Plot the data with traffic (in bytes) on the y-axis (log scale) and the cache block size on the x-axis (log scale). The graph should have two lines plotted; one for write-back and one for write-through traffic.

#### Questions

11. At what block size do the two write policies generate approximately the same number of writes to the bus?
12. Explain the difference in traffic observed for smaller block sizes.
13. Explain the difference in traffic observed for larger block sizes.

### Impact of different cache block sizes

Your simulator should already have the functionality to explore block sizes in the context of a four-way set associative cache.

#### Experiment 4A

Use the simulator to calculate the miss rate for a 64KB, four-way set associative, write-back cache with several different block sizes: 4B, 8B, 16B, 32B, 64B, 128B, 256B, 512B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB blocks.

- Associativity = 4
- Block size = 4B, 8B, ... , 16 KB
- Capacity = 64KB

### Graph #4: Miss Rate vs Block Size

Plot the data on a graph with miss rate on the y-axis, log of the cache block size on the x-axis, and the miss rate plotted as a line.

#### Questions

14. Explain the observed miss rate associated with a small block size.
15. Explain the observed miss rate associated with a large block size.
16. What is the block size with the lowest miss rate?

## Part 2: Multi-Core Simulation

Now that you have a single core cache simulator up and running, let's expand your simulation abilities to multiple cores! You have been given traces memory operations from 1, 2, and 4 threads.

### Task 8: multiple cores, no coherence protocol

Notice that the function `access_cache` takes an action as one of its arguments. So far, these actions have been simply LOAD or STORE. Once multiple cores are in play, you'll start to see LD\_MISS and ST\_MISS actions being sent to your cache (see how it is called in the `process_trace()` function in `simulator.c`.)

To begin with, your simulator should be able to run a 2t thread with 2 cores with no coherence protocol. Core 0 should produce the exact same statistics as it did before, except now it is snooping on the bus (but without a



coherence protocol it's not doing anything in response to these snoops). Update your simulator to keep track of **n\_bus\_snoops** (how many remote bus events does each core observe?) and **n\_snoop\_hits** (of these remote bus events observed, how many concern a valid cache line in the local cache?) in `cache_stats.c`.

## Task 9: VI protocol

Implement the VI protocol as specified in the lecture notes.

- A "use" in the context of LRU (least recently *used*) should always be with respect to actions coming from the CPU, not from the bus.
- If a cache line is in the V state and then moves to the I state, the cache needs to perform a writeback iff the line is dirty.

### Experiment 4B

Use the simulator to study the change in miss rates **for core 0** associated with a 64KB, four-way set associative, write-back cache with several different block sizes: 4B, 8B, 16B, 32B, 64B, 128B, 256B, 512B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB blocks across 2 and 4 cores, using the **VI** protocol.

- Associativity = 4
- Block size = 4B, 8B, ..., 16 KB
- Capacity = 64KB
- Cores = 2, 4

**Graph #4 [continued]:** add in Miss Rate vs Block Size for 2 and 4 cores, VI Protocol

Go back to Graph #4 and add in 2 more lines: 1 for 2 cores and 1 for 4 cores, using the VI protocol. [Submit two separate graphs for experiment 4A and 4B.](#)

### Questions

17. Why does the miss rate get worse with more cores?
18. If the miss rate is so bad, why would one use the VI protocol over no protocol?

## Task 10: MSI protocol

Implement the MSI protocol as specified in the lecture notes.

- A "use" in the context of LRU (least recently *used*) should always be with respect to actions coming from the CPU, not from the bus.
- If a cache line is in the M state and then has to move to the S or I state, the cache needs to perform a writeback.
- An upgrade miss (which you should keep stats for) does not generate memory traffic; the core needs to upgrade its cache line but it already has the actual data.

Note: The MSI protocol is rather involved compared with the VI protocol. It may be easy enough to write a single **access\_cache** function that supports both **No** protocol and the **VI** protocol, but you will want a separate function that implements the MSI protocol. It is sufficient to have **access\_cache** call a different access function if the MSI protocol is enabled. Otherwise you will end up in a very complicated set of if/else statements which will be very hard to write correctly and/or debug.

*Tip:* look at the state transition diagram and make sure your code covers all possible transitions.

### Experiment 5

Use the simulator to study the change in miss rates **for core 0** associated with a 64KB, four-way set associative, write-back cache with several different block sizes: 32B, 64B, 128B, 256B, ..., 2KB blocks across 2 and 4 cores, using the **MSI** protocol.

- Associativity = 4
- Block size = 32B, 64B, ..., 2KB
- Capacity = 64KB
- Cores = 1, 2, 4

### Graph #5:

replicate Graph #4 (Miss Rate vs Block Size for 1, 2, & 4 cores) but this time with MSI Protocol

### Questions

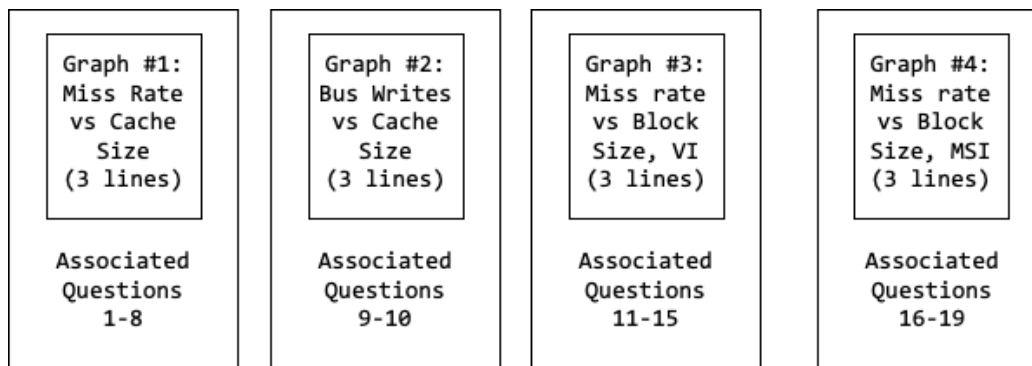
- For the 1 core trace, which has a lower miss rate: the VI protocol or the MSI protocol? Explain why.
- For the 4 core trace, which has a lower miss rate: the VI protocol or the MSI protocol? Explain why.
- For a 2 core trace, with a block size of 64B, what fraction of bus snoops by Core 0 are "hits" (i.e., the LD\_MISS or ST\_MISS on the bus is for a cache line that is currently valid in Core 0's cache).
- For a 4 core trace, with a block size of 64B, what fraction of bus snoops by Core 0 are "hits"?

## Deliverables

**Source Code.** Submit your completed code via **CMS** under the P5 assignment. This assignment *will* be graded for style, so make sure to read the **C Programming Style Guide** (<https://canvas.cornell.edu/courses/54972/pages/c-programming-style-guide>) and look over your P4 style comments!

- `cache.c`
- `cache_stats.c`

**Writeup.** You will submit your graphs and answers to all of the above questions on **Gradescope**. *Have one member upload the writeup and then add the other two group members to the group.* Your writeup should look approximately like this:



Most questions can be answered in just a few sentences. We are **never** looking for paragraphs. Please do not ask "Is this right?" or "I'm stuck on #18. Can you help?" These questions are meant to be worked out amongst your team members. Between the 3 of you, you should be able to come up with the right answers.

**P5 Completion Survey** (<https://canvas.cornell.edu/courses/54972/quizzes/119247>) on **Canvas**. This is the usual project survey that you've completed for previous projects,

**P5 Peer Evaluation** (<https://canvas.cornell.edu/courses/54972/assignments/522721>) on Canvas. You will be asked to fill out a team evaluation, assessing how well your team worked and detailing the contributions of each member (yourself included).

**View P5 Supplement** (<https://canvas.cornell.edu/courses/54972/pages/p5-supplement>).

<b>P4</b> ( <a href="https://canvas.cornell.edu/courses/54972/assignments/522718">https://canvas.cornell.edu/courses/54972/assignments/522718</a> )	
--	--

<

(<https://canvas.cornell.edu/courses/54972/modules/items/2052988>)

>

(<https://canvas.cornell.edu/courses/54972/module>)