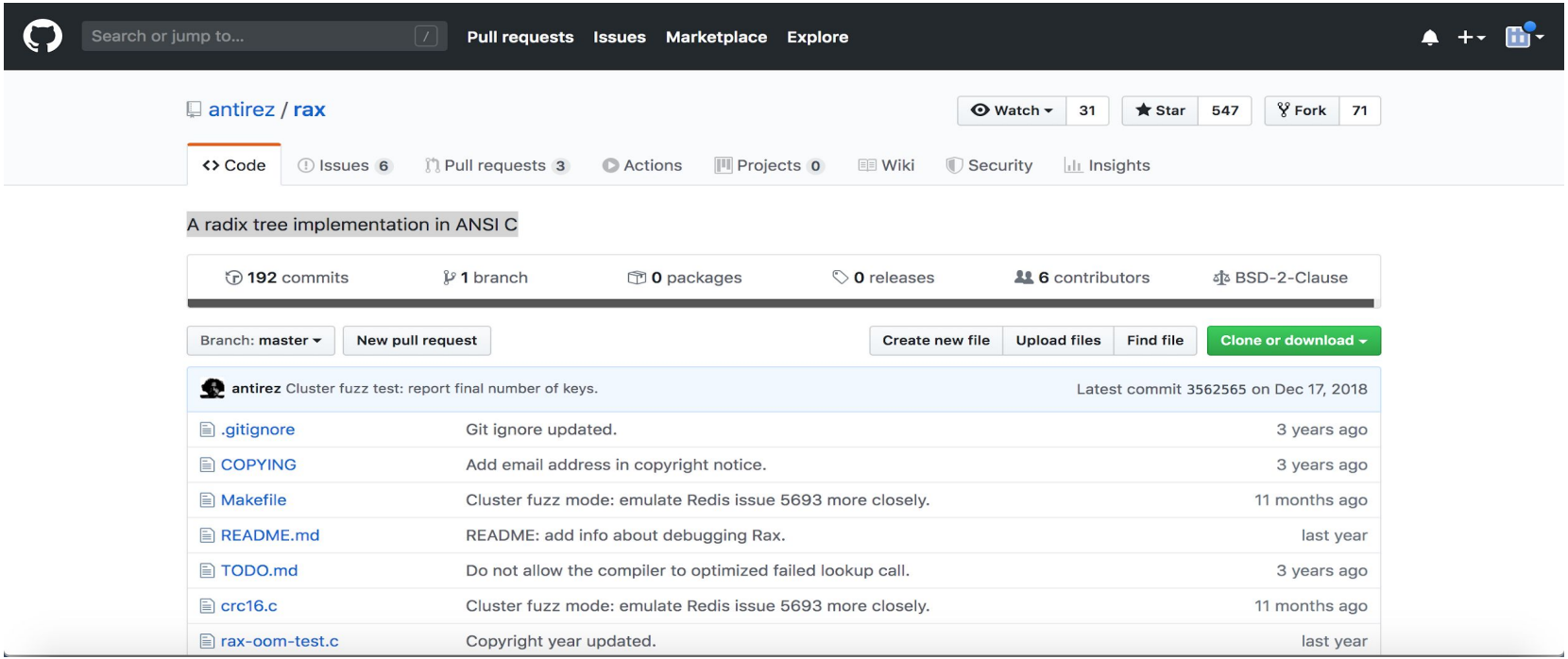
CSE220 Project Part 3 Rax - a radix tree implementation in ANSI C

Adil Rahman

Link to Github Repo

- https://github.com/antirez/rax



Brief description of what the workload is

- **Rax** is a radix tree implementation in ANSI C that was originally created to fix a performance issue with Redis.
  - Redis: an open source in-memory data structure store that can be used as a database, cache, or message broker [1].
  - Radix tree: a space-optimized trie in which each node that is the only child is merged with its parent [2].
    - A trie is a specific type of search tree or ordered tree data structure that is used to store dynamic sets or associative sets where the keys are strings [3].
- The intention of Rax as an application was to create a balance between performance and memory usage through the use of radix trees [1].
  - The project uses fuzz testing techniques to test the implementation on a large amount of potential states [1].

Brief description of what the workload does

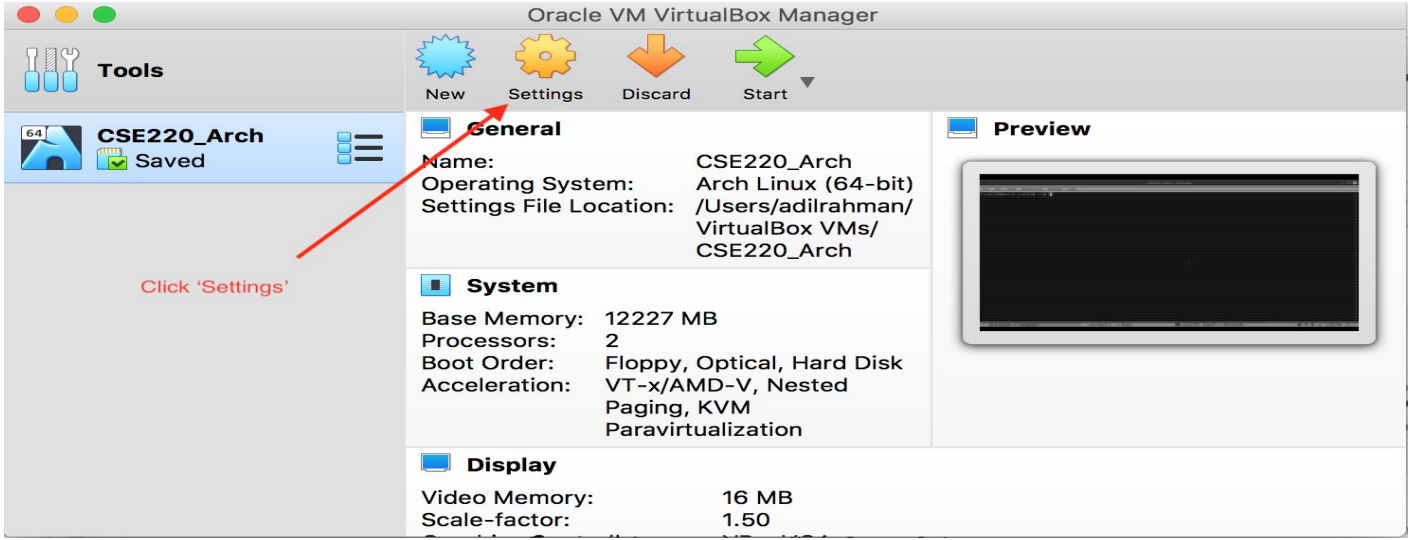
- The file rax-test.c is present to test the actual implementation.
- Functions:
  - hash table look-up, addition and removal of an element, locating an element within the table, and freeing the entire hash table.
  - uint32 to uint32 conversion, uint32 to key, int to key, and returning UNIX time in microseconds.
- Benchmark Tests:
  - do_benchmark: creates radix tree with 5,000,000 keys and elements, perform linear and random lookups by key and then deletes all 5,000,000 elements from tree.
  - do_units: perform walks through radix tree while testing proper functionality of iterator, and insertion of elements.
  - do_fuzz_cluster: fuzz test for generating random keys and inserting and removing elements
  - do_fuzz: fuzz test for generating random keys, inserting and removing elements, iterator functionality, and reporting of all elements.
  - do_regression: series of regression tests testing functionality of functions for radix tree
  - do_hugekey: testing creation of key bigger than (2^29)-1 characters

Steps to compile and run the workload

- Setting up Internet for VirtualBox
- Running CSE220_Arch on VirtualBox
- Git clone repo into VirtualBox
- Adding Static Flags to Makefile
- Editing Input File
- Compiling the Files
- Making the Script to Run the Workload

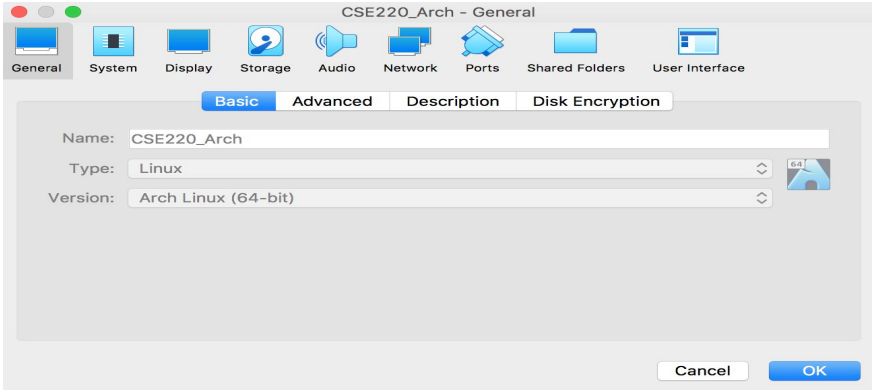
Setting up Internet for VirtualBox

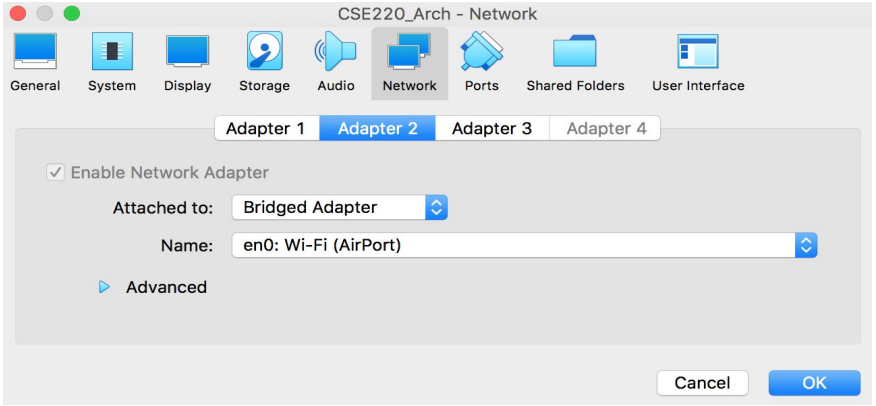
- **<u>Ensure that your VirtualBox is connected to the internet.</u>** This will be necessary when we use git clone to copy the repo contents into the VirtualBox. The following steps were performed on the Mac High Sierra OS.
- Open VirtualBox and click the settings button:



Setting up Internet for VirtualBox (2)

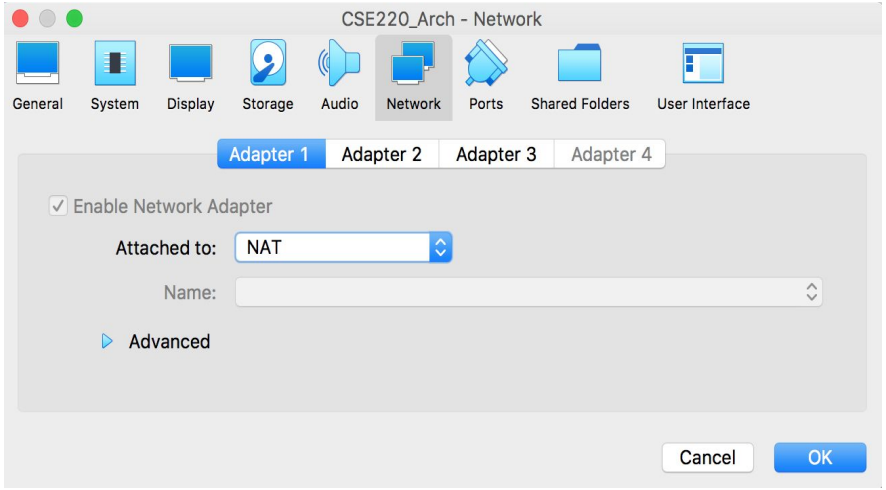
- Click on the ‘Network’ button
- Ensure that Adapter 1, Adapter 2, and Adapter 3 are set to the following settings:
  - Adapter 1:
    - Attached to: NAT
  - Adapter 2:
    - Attached to: Bridged Adapter
    - Name: en0: Wi-Fi (AirPort)
  - Adapter 3:
    - Attached to: Internal Network
    - Name: intnet
- Click the ‘OK’ button.

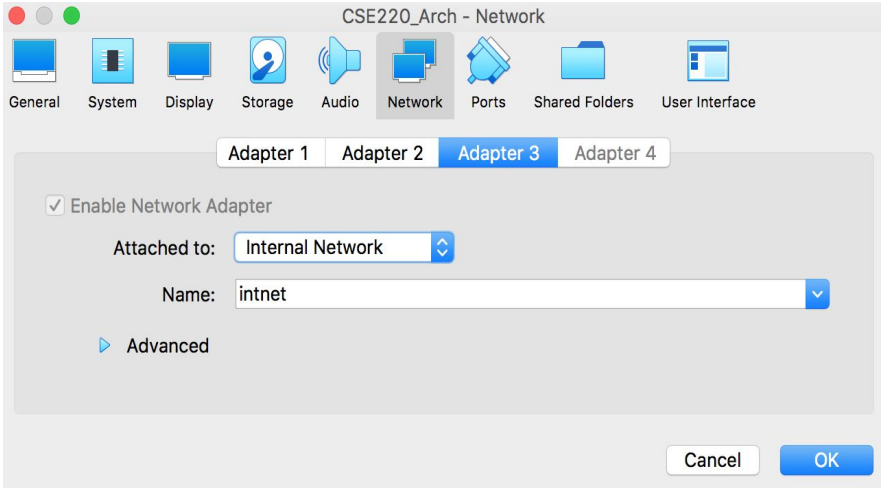




Setting up Internet for VirtualBox (3)

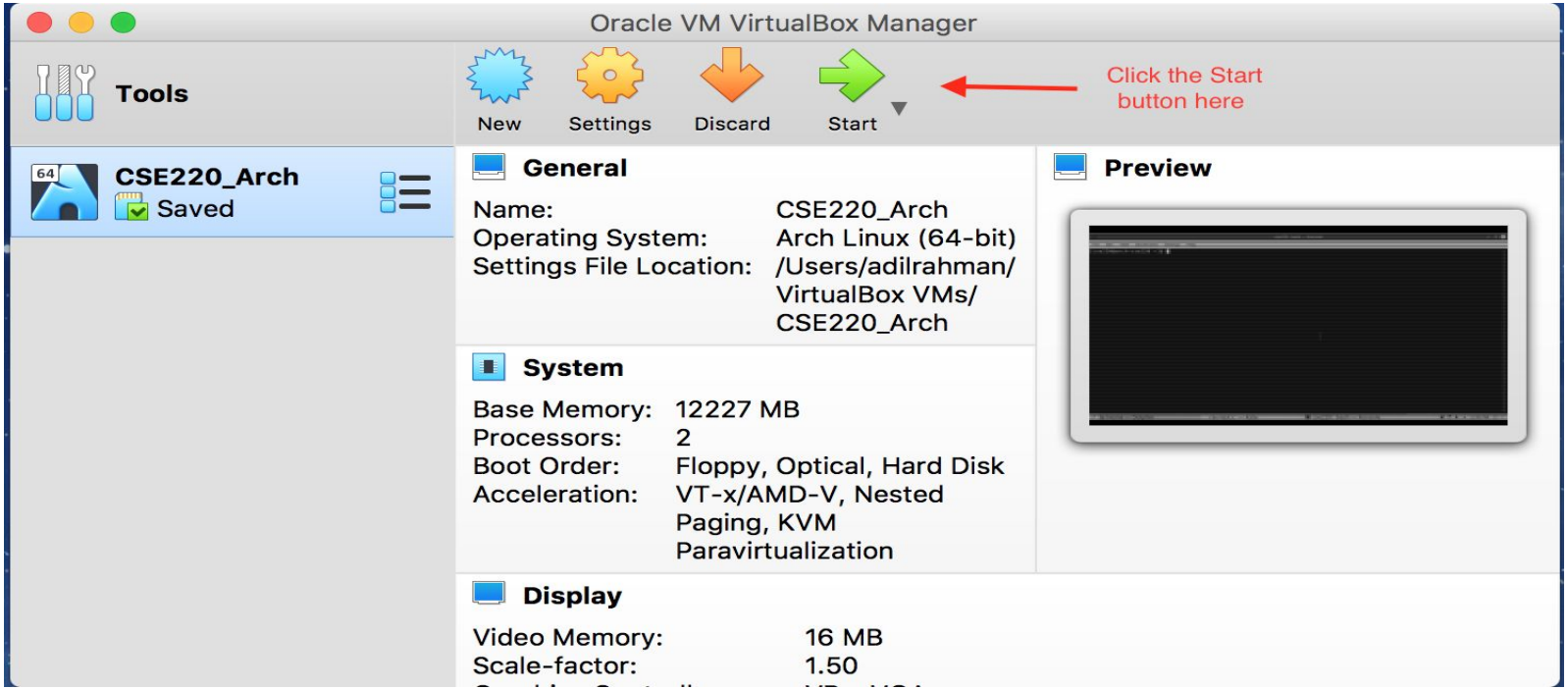
- Screenshots of Adapter 1 and Adapter 3 settings





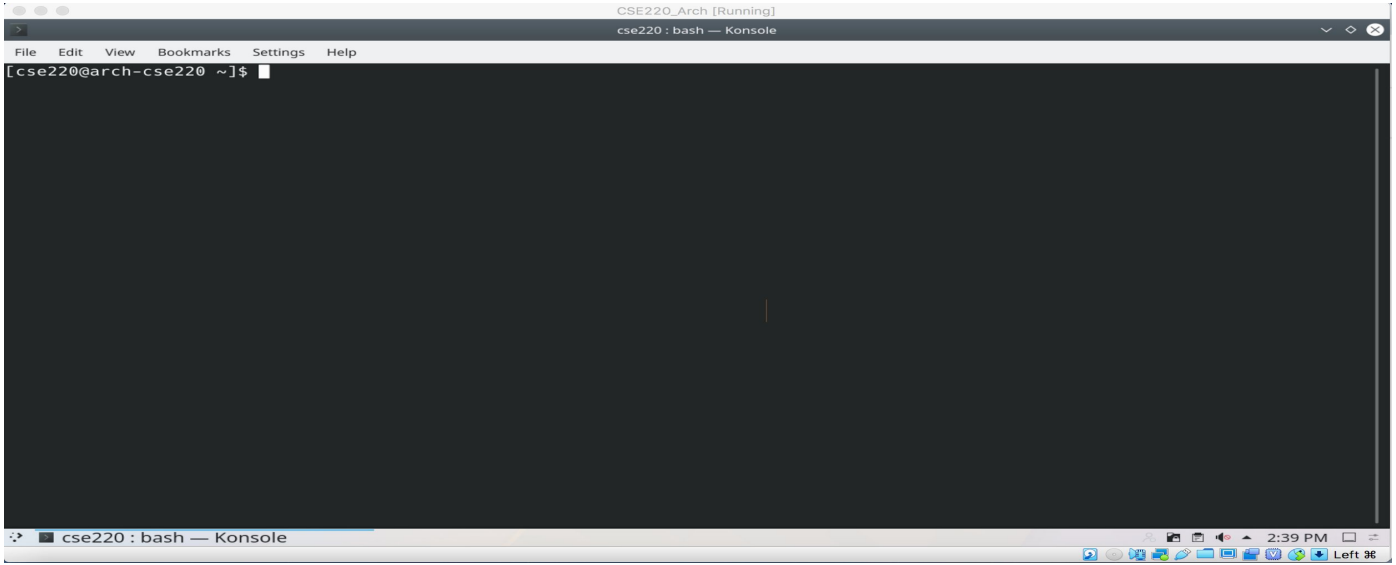
Running CSE220_Arch on VirtualBox

- Open the VirtualBox application and run ‘CSE220_Arch’ as seen below:



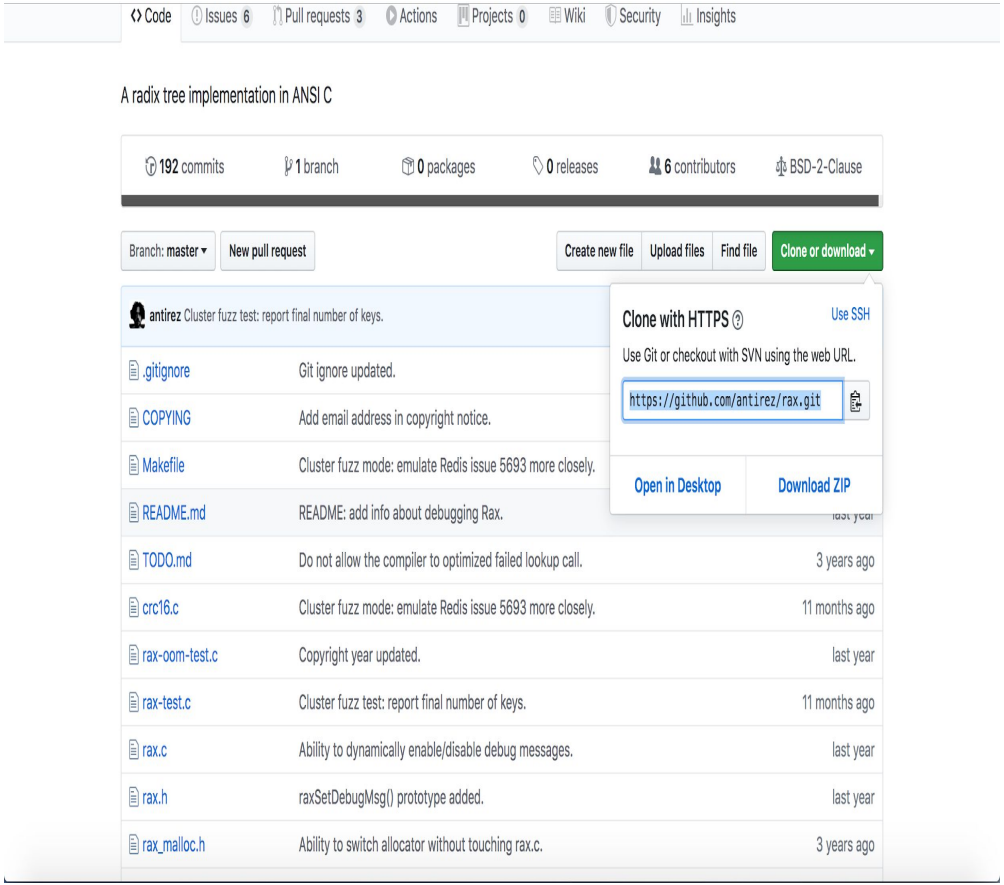
Running CSE220_Arch on VirtualBox (2)

- Following the steps of Project Part 1, you should successfully see the Linux OS.
- Open up a new Konsole at the bottom left of the screen as seen below:



Git clone repo into VirtualBox

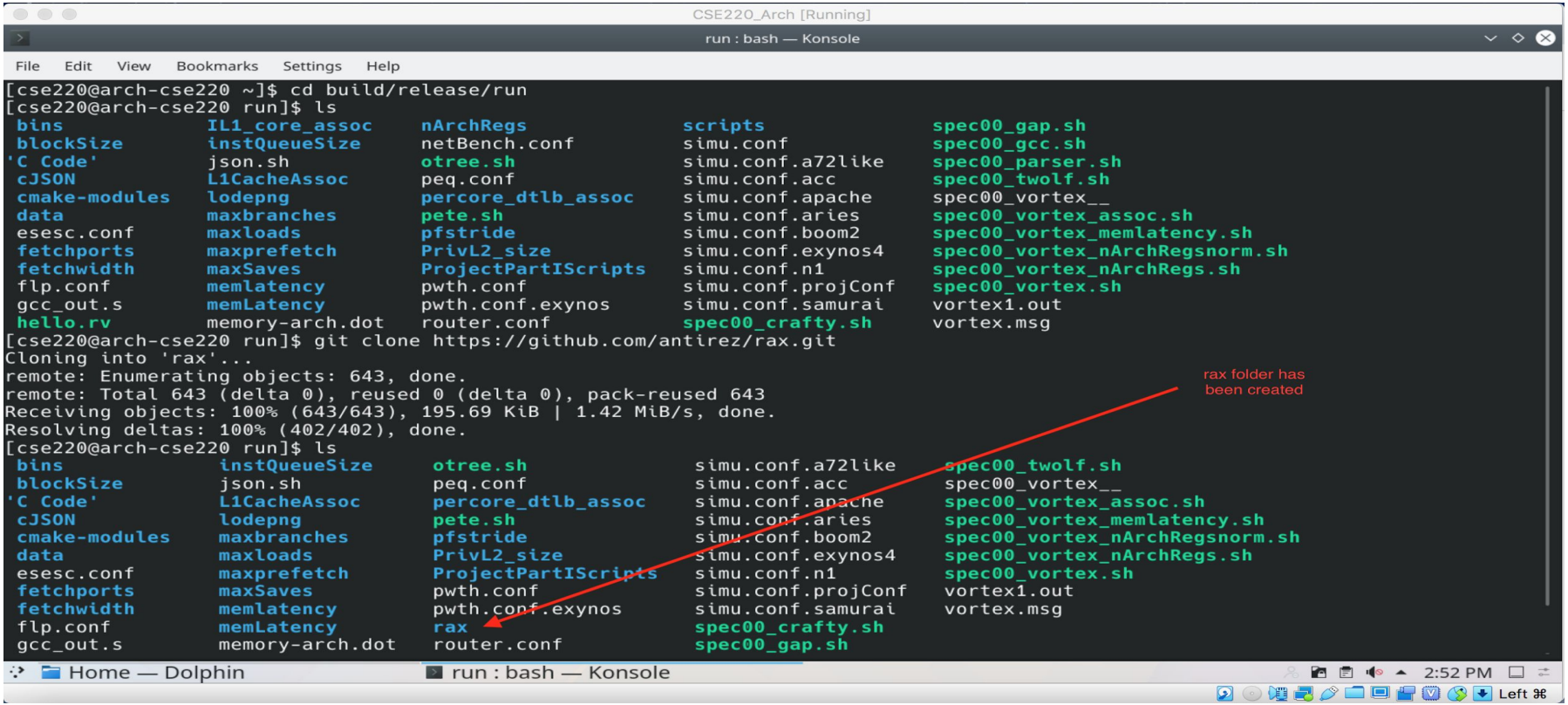
- Click on the following link to find the repo for Rax:
  - https://github.com/antirez/rax
- Locate the ‘Clone or download’ button on the repo. It should look like this:
  - https://github.com/antirez/rax.git
- Copy this previous link for a future step. If your VirtualBox does not have a shared clipboard capability with your actual operating system, you will have to manually type this command in the git clone step.



Git clone repo into VirtualBox (2)

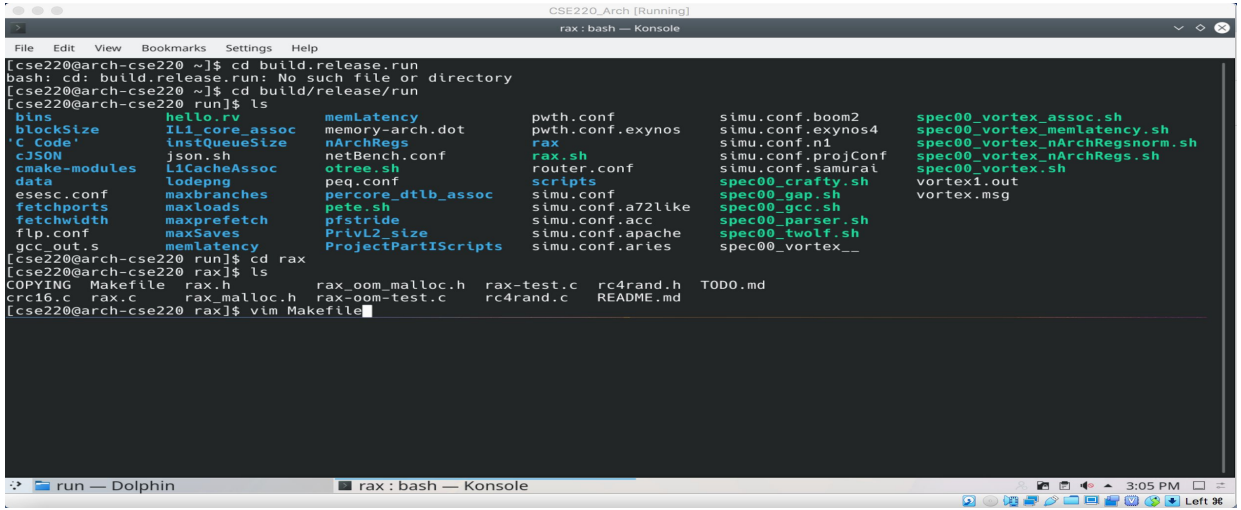
- First change to the run folder by inputting the following command from the home directory:
  - cd build/release/run
- Ensure simu.conf.apache is inside of your run folder.
- From the previous slide, input the following command to obtain the files from the repo into the run folder.
  - git clone https://github.com/antirez/rax.git

Git clone repo into VirtualBox (3)



Adding Static Flags to Makefile

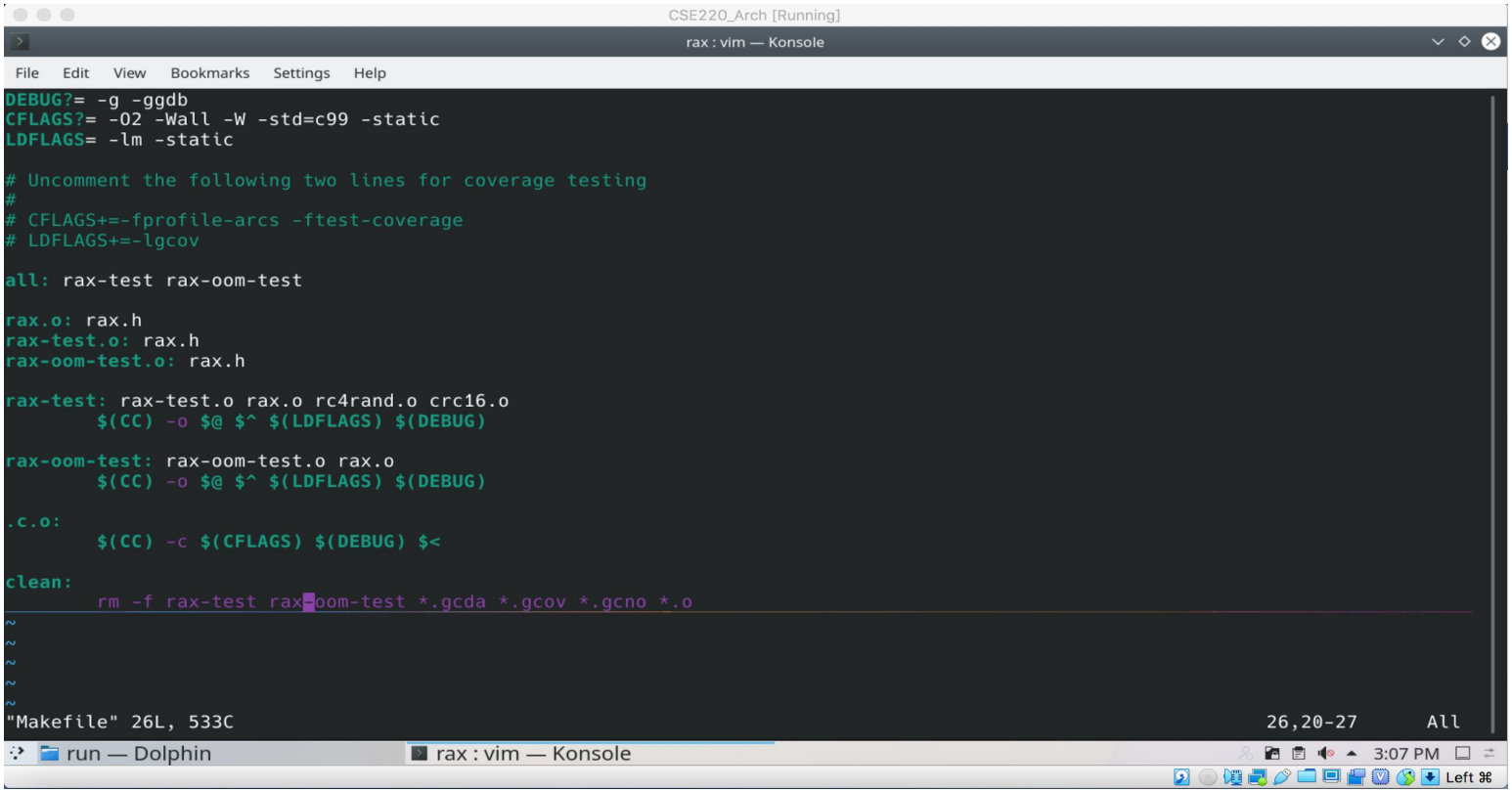
- Now go into the rax folder by inputting the following command:
  - cd rax
- Static flags will need to be added to the Makefile. Open up the file by inputting the following command:
  - vim Makefile



Adding Static Flags to Makefile (2)

- Once you are in the editor, ensure that ‘-static’ has been added to CFLAGS? and LDFLAGS.
- While in vim, press ‘i’ to be in the INSERT mode. Add ‘-static’ to the sections mentioned in the previous bullet. Then follow these steps:
  - Press the ‘esc’ button to exit the INSERT mode
  - Press the ‘:’ button, followed by ‘wq’ to make the writes to the file
- It should look like the following after the steps are done:
  - CFLAGS?= -O2 -Wall -W -std=c99 -static
  - LDFLAGS= -lm -static

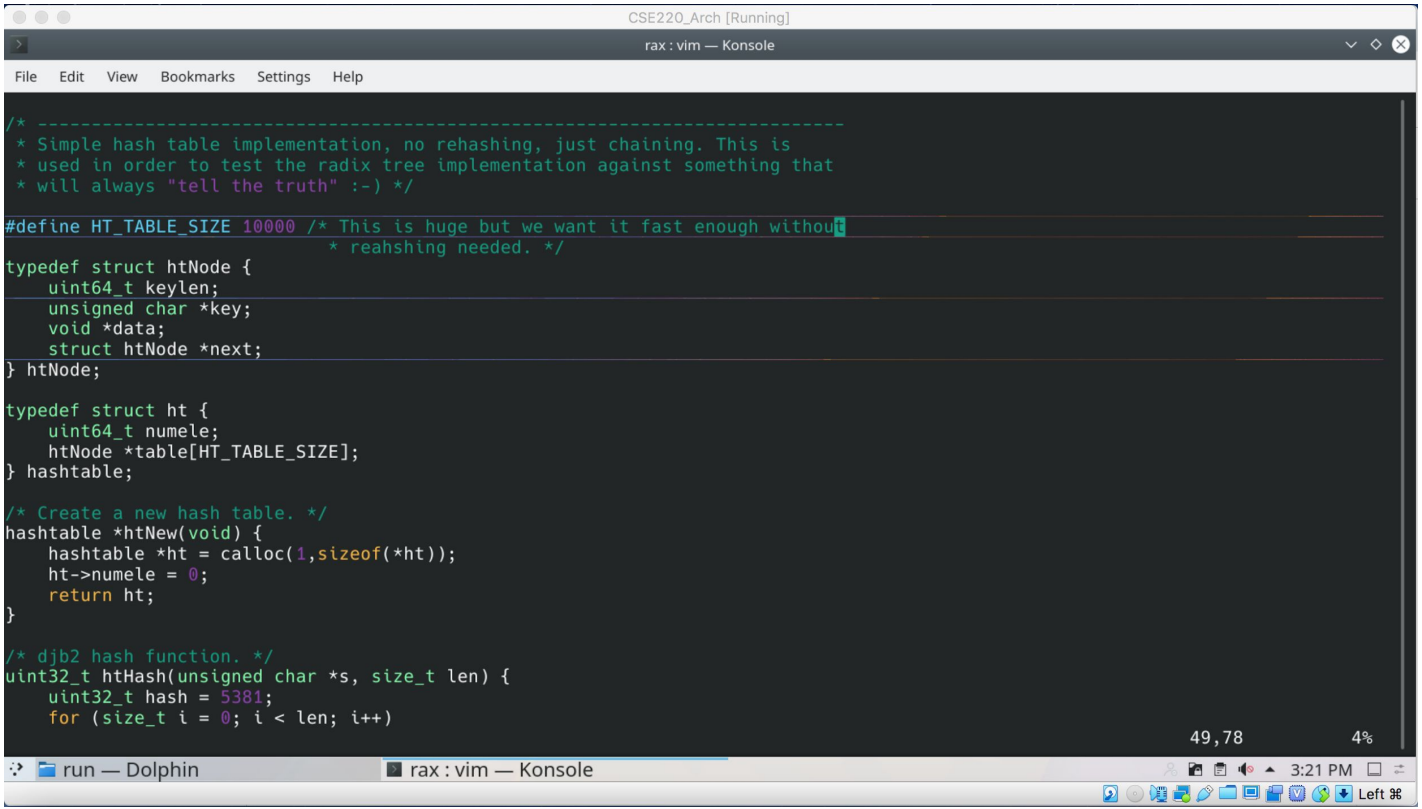
Adding Static Flags to Makefile (3)



Editing Input File

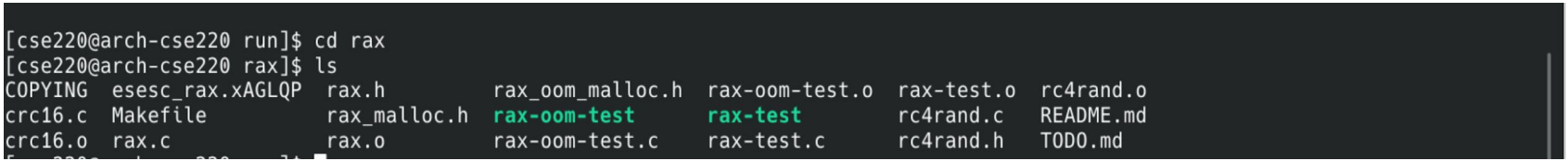
- You will now need to make an edit to the following input file:
  - rax-test.c
- Input the following command to edit the file:
  - vim rax-test.c
- Line 49 of the code should have the following:
  - #define HT_TABLE_SIZE 100000
  - Edit this so that the table size is now 10000 instead:
    - Press ‘i’ for INSERT mode, make the change, then press ‘esc’, then ‘:’, then ‘wq’.
    - The line should now look like this: #define HT_TABLE_SIZE 10000

Editing Input File (2)

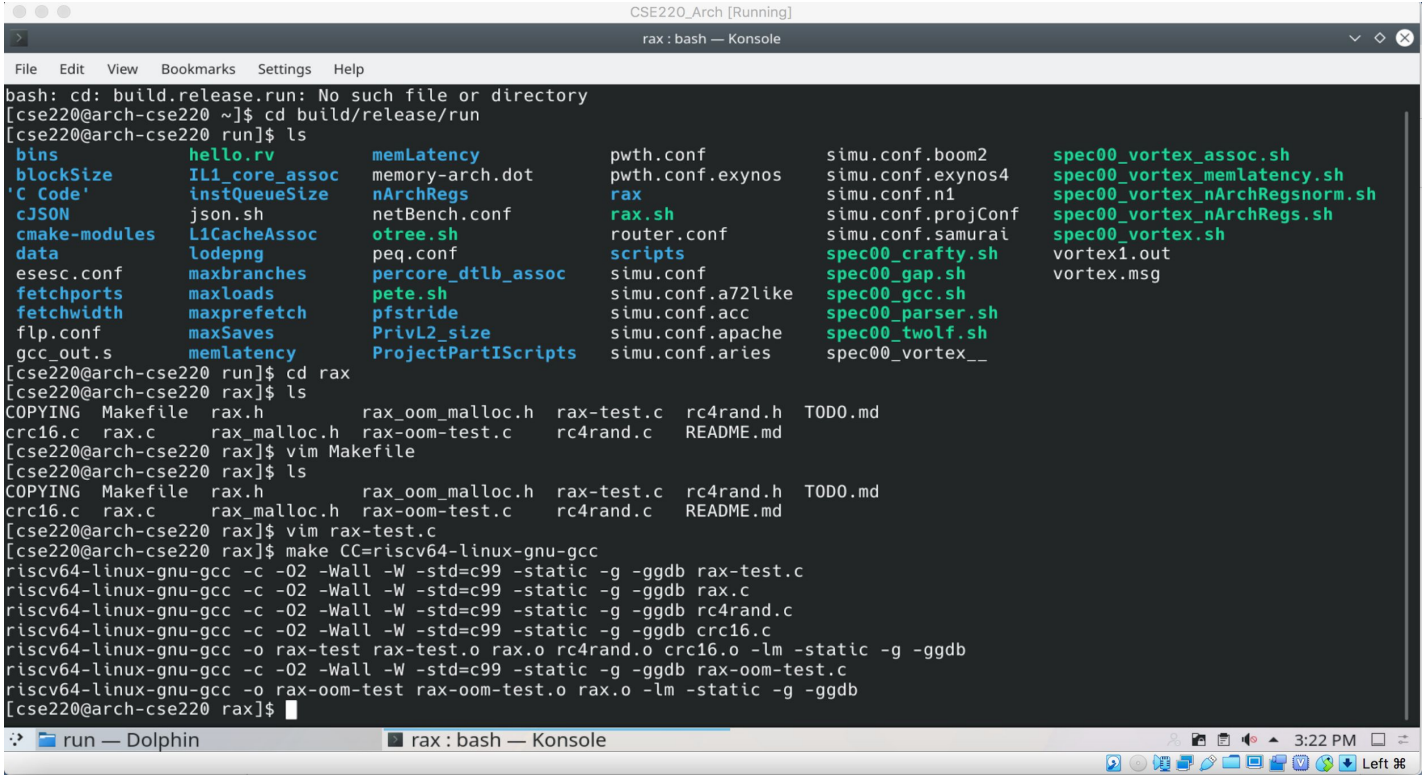


Compiling the Files

- Ensure that you are still in the rax folder.
- Input the following command to commence the compiling:
  - make CC=riscv64-linux-gnu-gcc
- You should now see the riscv file in the rax folder for rax-test as follows:



Compiling the Files (2)

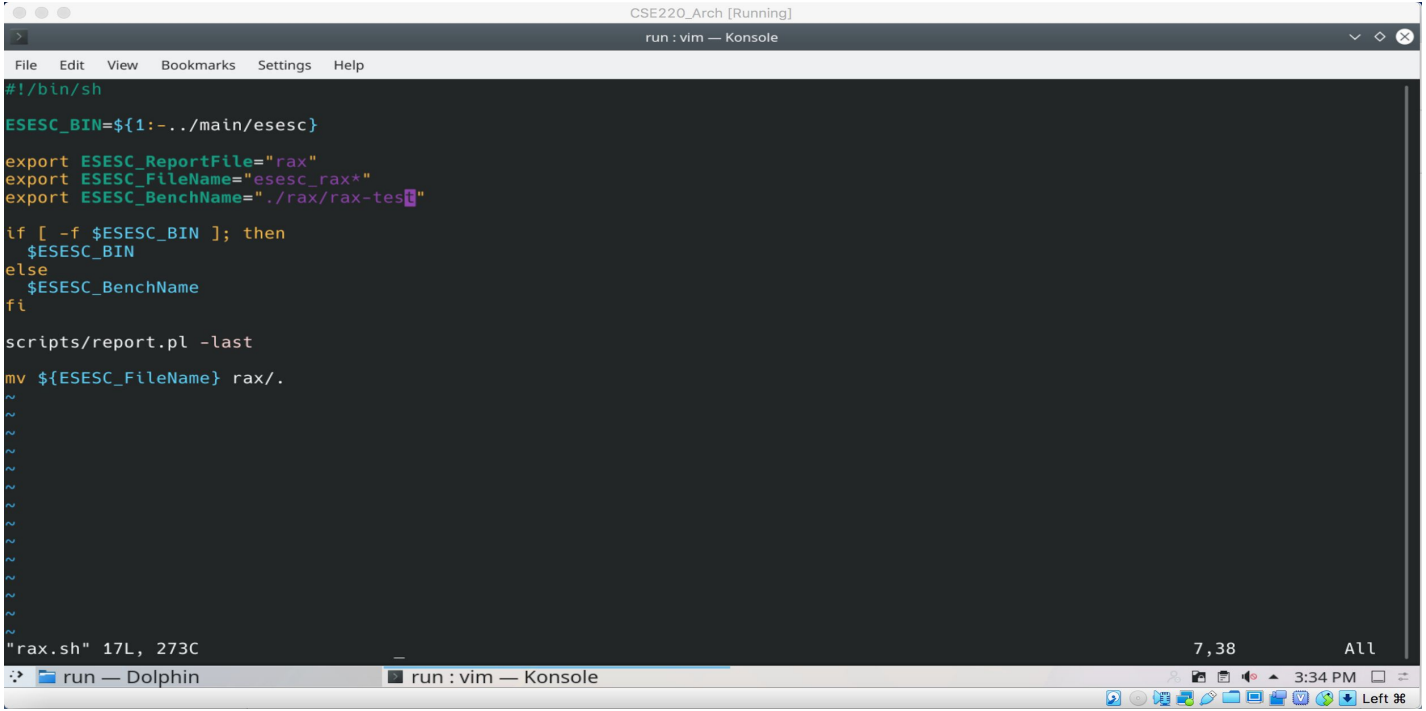


Making the Script to Run the Workload

- Ensure that you are now in the run folder.
- Create a script called ‘rax.sh’ to perform the run in ESESC
- This can be done through the GUI. In the run folder, perform the following steps:
  - Right click and select ‘Create New’ and then ‘Text File …’
  - Name the file ‘rax.sh’
  - In the Konsole, ensure you are still in the run folder and input ‘vim rax.sh’
  - Fill in the file with the text seen in the following slide
- Once this is done, you will need to give execute permission for the file. While in the run folder where rax.sh is present, input the following command:
  - chmod +x rax.sh

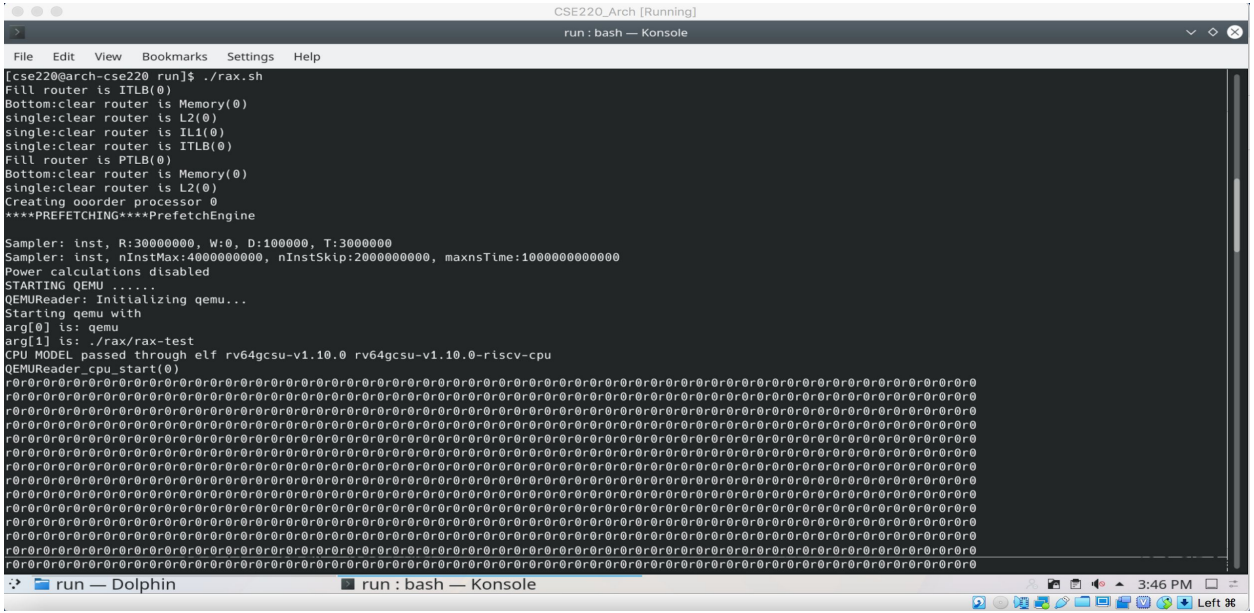
Making the Script to Run the Workload (2)

- Type the following into the rax.sh file and save it



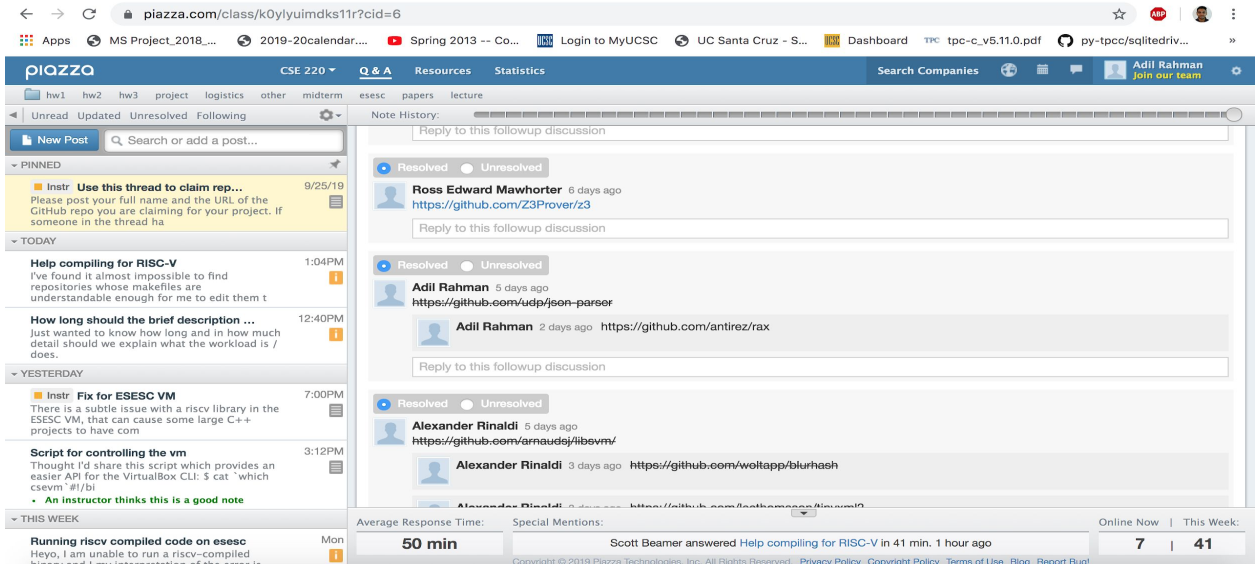
Making the Script to Run the Workload (3)

- Now run the script file rax.sh
- In the run folder, input the following command to run the script file:
  - ./rax.sh
- The code will now run

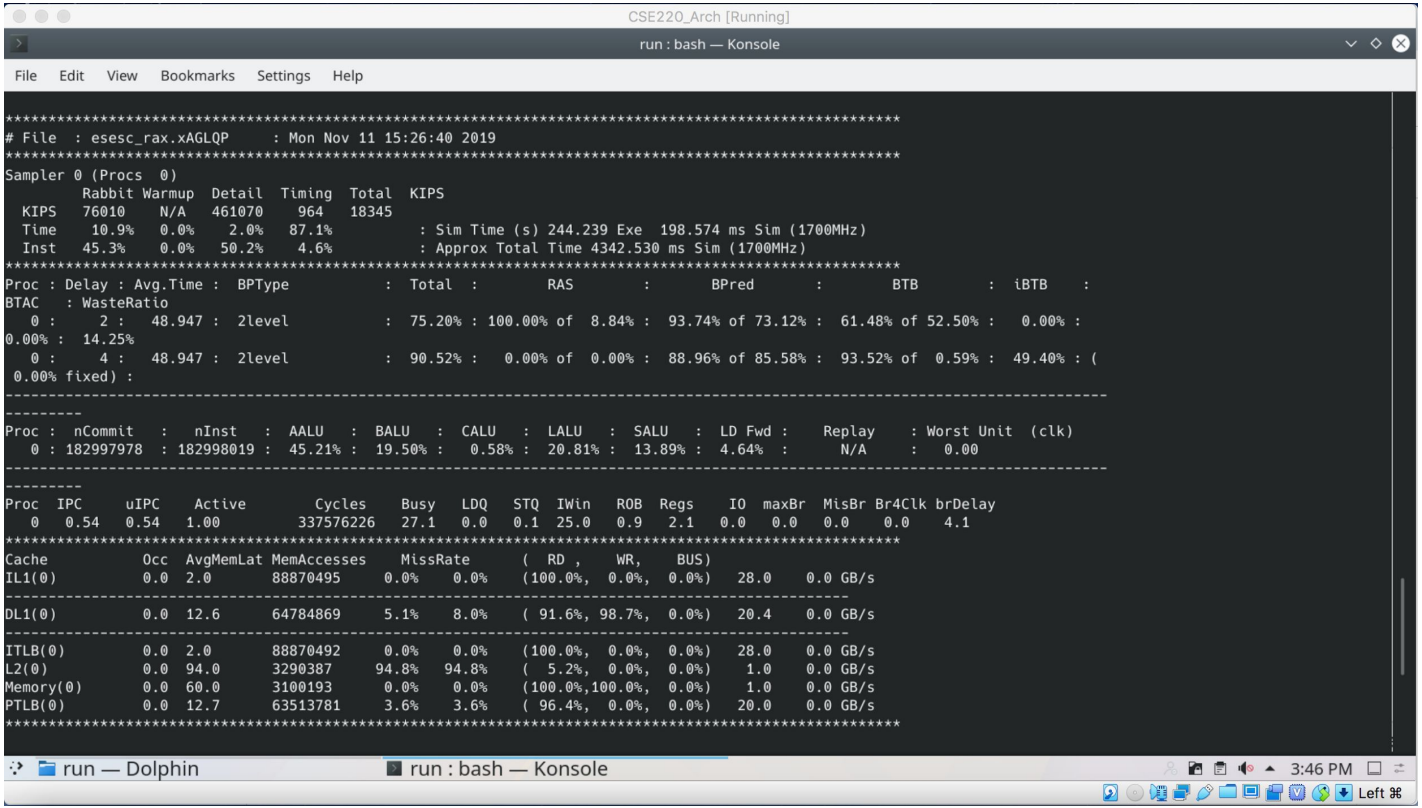


Initial results showing low IPC and reasonable instruction count

GitHub Stars	Instruction Count	IPC
547	182998019	0.54



Raw Report from running it on ESESC



Explanation of bottlenecks and how you found it

- The following bottlenecks are occurring as seen in the initial raw report:
  - **Very low branch prediction success**, with the first 2 level branch predictor only having a 93.74% accuracy and the second 2 level branch predictor only having a 88.96% accuracy.
    - Was found by looking at the report percentage accuracies.
  - **Very low branch target buffer (BTB) success**, with the first 2 level branch predictor only having a 61.48% accuracy and the second 2 level only having a 93.52% accuracy.
    - Was found by looking at the report percentage accuracies.
  - **Very high primary and secondary miss rates at 94.8% in the L2 cache level of the memory hierarchy.** There are 3,290,387 memory accesses actually going into the L2 cache that miss in the L1 cache.
    - Was found by looking at the report percentage accuracies.
  - **Low instruction level parallelism (ILP)**. Since the insertions and removals in the radix tree are not dependent on each other, multiple instructions should be able to be performed simultaneously, however they are limited currently by the architecture.
    - Was found by increasing issueWidth and retireWidth parameters.

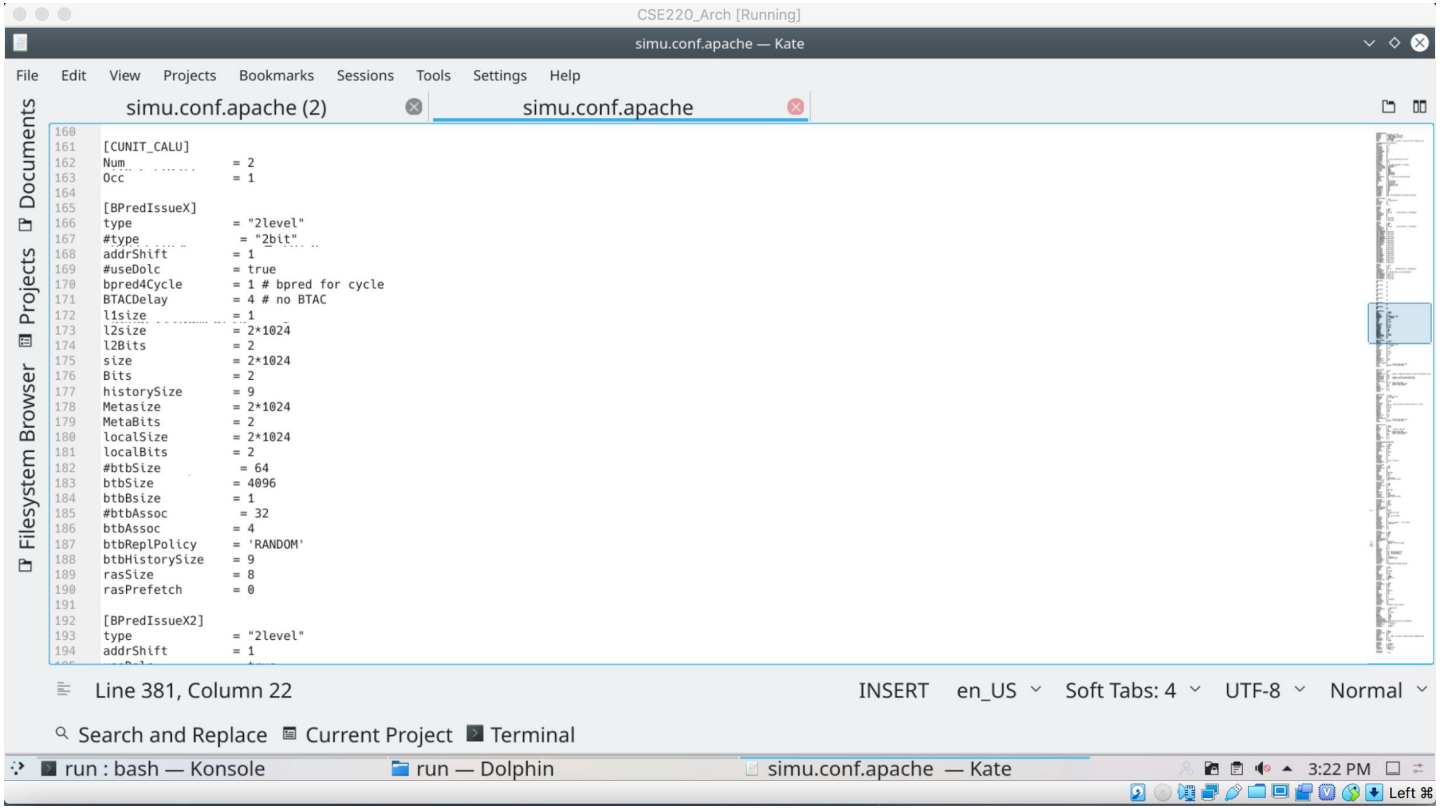
Adjusting the branch predictor parameters for bpred

- We tackle the first branch predictor that was using a 2 level as the bpred parameter.
- The code has multiple forward loops and if statements causing branches and jump instructions in the underlying assembly.
- The benchmark loops many times to do several insertions and removals in the radix tree. Inside of these forward loops are if statements to track whether errors have occurred or not.
- Since there is incredibly low BTB success, it was hypothesized that the BTB simply was not big enough to hold enough target PCs for the number of times the code jumped in the benchmark.
- In addition, there may have been some associativity issues that was causing misses in the BTB.

What architectural parameters you changed to improve performance for bpred

- We have decided to increase the size and also decrease the associativity.
  - btbSize went from 64 to 4096.
  - assoc went from 32 to 4.

Parameters for hybrid predictor for bpred



Explanation of how your changes fix the bottleneck

- Increasing the size allows more target PCs to be held, and changing the associativity to make it lower makes it so there are more blocks within an index and increases the chance of hitting in the BTB.
- Making this change results in a BTB success of 98.51%.

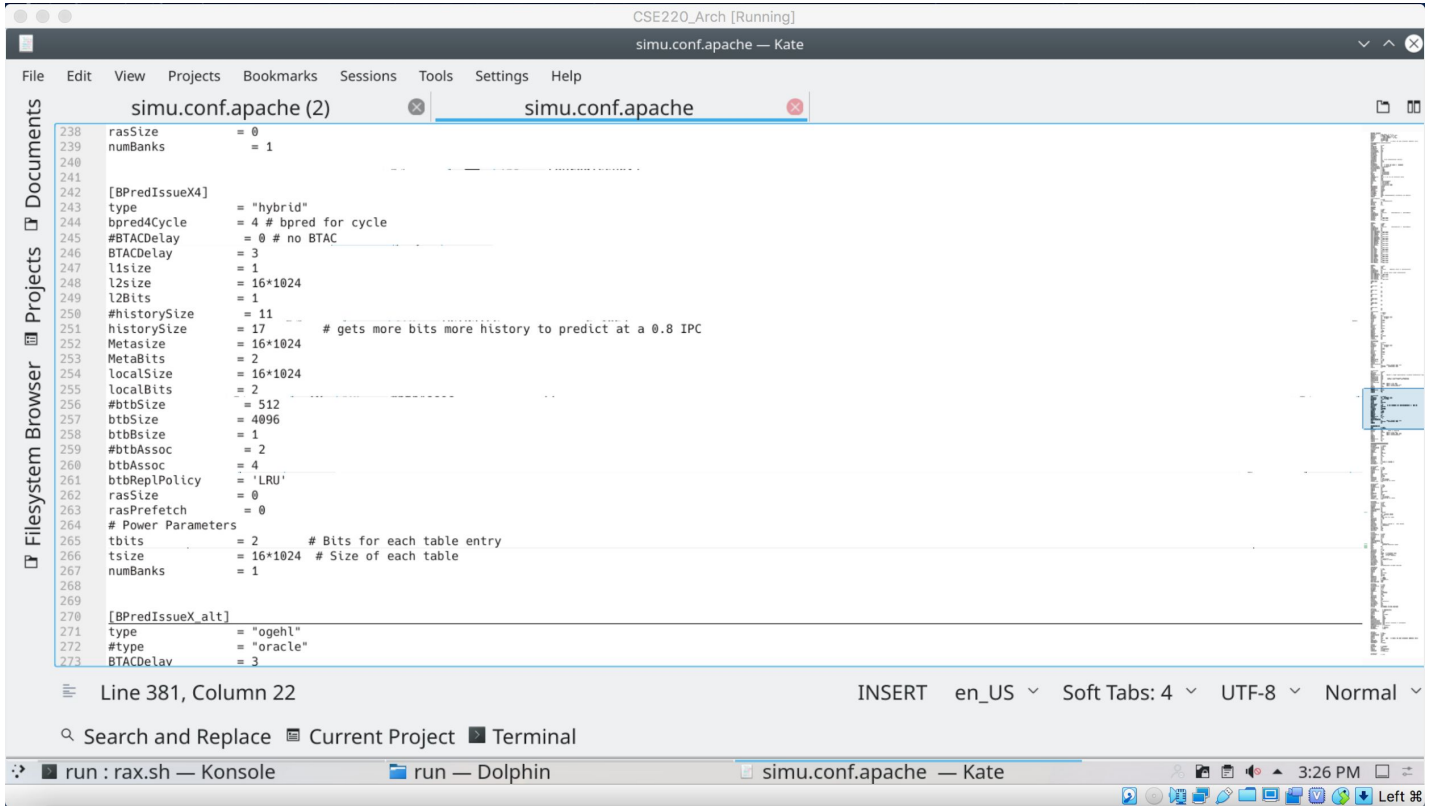
Adjusting the branch predictor parameters for bpred2

- We tackle the second branch predictor that was originally using a 2 level as the bpred2 parameter.
- This branch predictor had an incredibly low success rate at 88.96% in branch prediction.
- It was decided that a more sophisticated branch predictor could be used to improve the performance.
- The hybrid predictor was chosen because it could retain more knowledge of how the forward loops and if statements occur in the code. Having bits to keep track of the history would help this along with an increased BTB size and adjusted associativity.

What architectural parameters you changed to improve performance for bpred2

- Since the original simu.config.apache file does not have a hybrid predictor, we established one.
  - The initial hybrid branch predictor was taken from the simu.config file which was located in the home folder with the directory change build/debug/run
- We have decided to increase the history size, increase the BTB size and also decrease the BTB associativity.
  - historySize went from 11 to 17.
  - btbSize went from 64 to 4096.
  - assoc went from 32 to 4.

Parameters for hybrid predictor for bpred2



Explanation of how your changes fix the bottleneck

- Increasing the number of bits to keep track of the history allowed for more success of the branch predictor to track how the code progressed through if statements and forward loops. More information could be retained since there were more bits available to store these patterns.
- Increasing the size allows more target PCs to be held, and adjusting the associativity makes it so there are more blocks within an index and increases the chance of hitting in the BTB.
- Making this change results in a branch prediction accuracy of 97.18% and a success of 99.31% in the BTB.

Why the architectural parameter changes are reasonable for bpred and bpred2

- Increasing the size of the BTB to 4096 and the associativity to 4 is a reasonable change because these parameter settings are seen within the **Intel Ivy Bridge CPU** [4].
- Since branch predictors take up a large portion of the space on CPUs, having this size increase is not unreasonable, especially considering the performance gain from it.
- In addition, having 6 additional bits for the historySize is a very small addition and does not take up much additional space.
- 6 bits should also be very cheap to add in terms of cost.

Adjusting the L1 cache parameters

- We now tackle the L1 cache.
- There are many accesses that are missing the L1 cache and going into the L2 cache causing a high miss rate in the L2 cache. We attempt to minimize the amount of memory accesses that reach the L2 cache by creating more hits in the L1 cache.
- The benchmark itself first tests various insertions of all int keys, then string keys, and then alphanumeric keys.
- There will be many challenges dealing with storing the alphanumeric keys due to the varying byte length between an int (4 bytes) and a char (1 byte).

Adjusting the L1 cache parameters (2)

- However many of these generated keys were first placed into an array. We attempted to take advantage of this since the keys would be still next to each other since these arrays are stored contiguously in memory.
- It is hypothesized that there will be many hits when the keys are consisting of only ints or only chars. The majority of the misses will then potentially occur when alphanumeric keys are generated.
- Varying tests were performed where the associativity was adjusted to values of 1, 2, 4, 8, and 16.
    - bsize had the original value of 32
    - size had the original value of 64*1024

Changes in associativity with capacity as 64 KB and block size as 32 B

bsize	assoc	size	IPC after
32	1	64*1024	0.52
32	2	64*1024	0.6
32	4	64*1024	0.62
32	8	64*1024	0.62
32	16	64*1024	0.62

Changes in associativity with set capacity and block size

- It was seen that increasing associativity did help the IPC. This gave evidence that the type of cache misses that were occurring could be due to conflict misses.
- In order to reduce the possibility of these misses, we decided to increase the associativity so that more of the cache could be properly utilized.
- In addition to this, we decided that increasing the block size to 64 would also help with this. By increasing the block size, more data could fit inside a single block.
- This also would make it more likely that with a given index you would get a hit based off the index and tag bits.

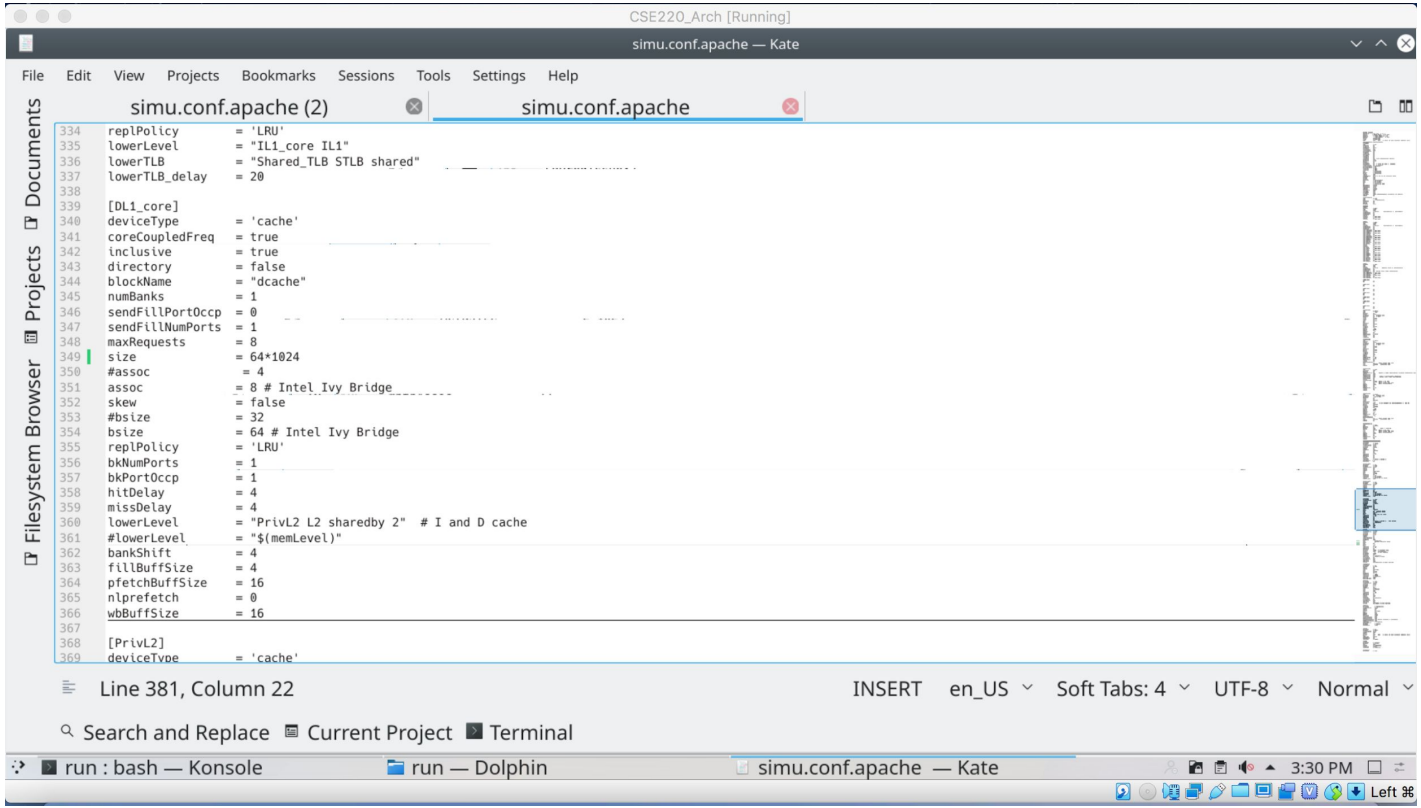
Changes in associativity with capacity as 64 KB and block size as 64 B

bsize	assoc	size	IPC after
64	1	64*1024	0.52
64	2	64*1024	0.61
64	4	64*1024	0.63
64	8	64*1024	0.64
64	16	64*1024	0.64

What architectural parameters you changed to improve performance for DL1_core

- We have decided to increase the block size and also increase the associativity.
  - bsize went from 32 to 64
  - assoc went from 4 to 8.

Parameters for L1 cache for DL1_core



Explanation of how your changes fix the bottleneck

- Increasing the block size helped to take advantage of the spatial locality between the varying keys that were stored contiguously in memory. It also allowed the cache to be arranged in such a way that more hits could occur for a given index.
- Increasing the associativity helped arrange the cache in such a way that more blocks of the cache were being utilized than before, which reduces conflict misses.
- This resulted in the memory accesses going to L2 reducing from 3,290,387 memory accesses to 2,357,703 memory accesses, reducing a big amount of latency cost.

Adjusting the L2 cache parameters

- We now tackle the L2 cache.
- A large majority of the latency is due to misses in the L2 cache, as the miss rate was originally 94.8%.
- It is hypothesized that the alphanumeric keys that are being generated are the majority of the causes for misses due to the varying byte length of the keys depending on how many ints versus chars there were in the key.
- After all of the changes above, only a little more would be required to obtain the 1.5x speedup necessary.
- Varying tests were performed where the associativity was adjusted to values of 2, 4, and 8.
    - bsize had the original value of 32 and was changed to 8
    - size had the original value of 2*1024*1024 and was changed to 256*1024

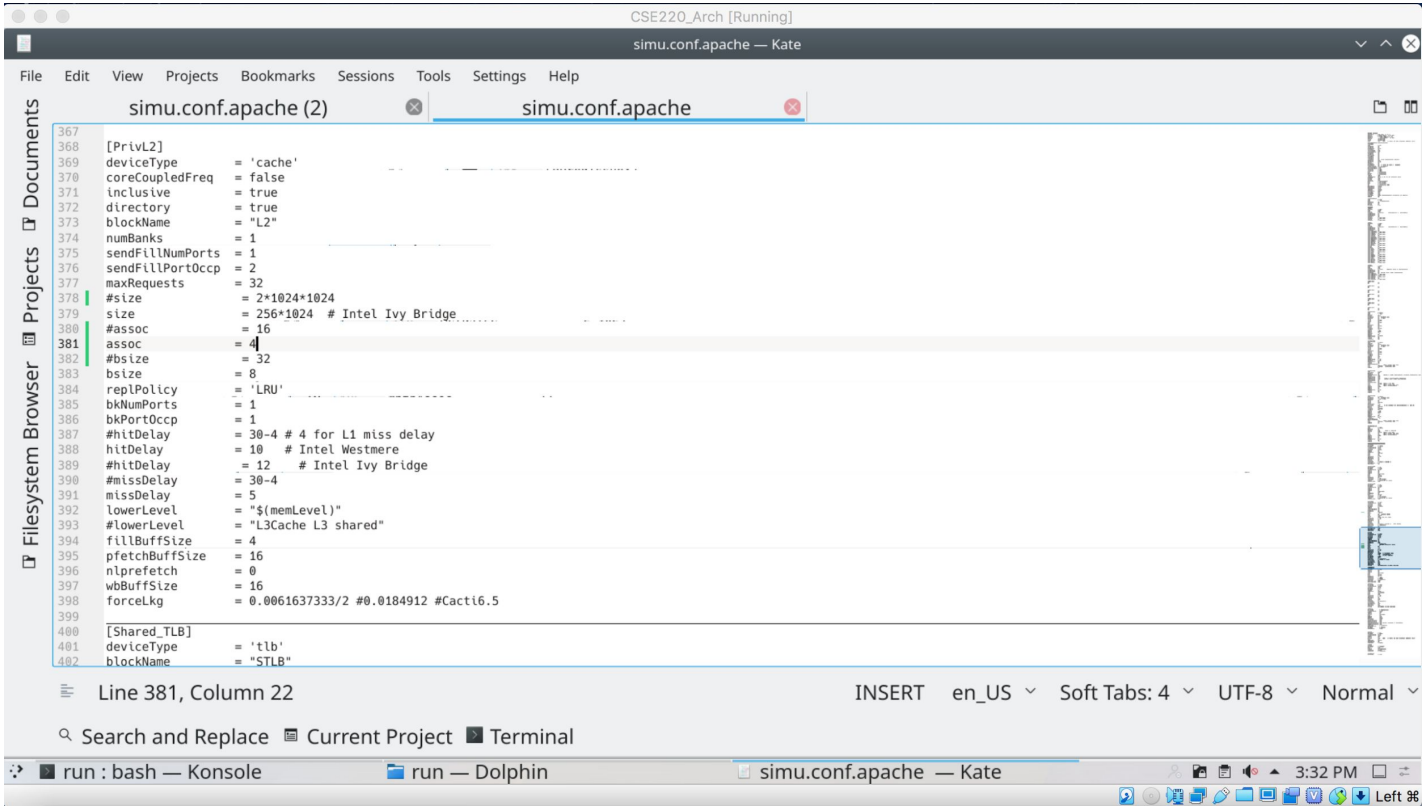
Changes in associativity with capacity as 256 KB and block size as 8 B

bsize	assoc	size	IPC after
8	2	256*1024	0.63
8	4	256*1024	0.75
8	8	256*1024	0.75

What architectural parameters you changed to improve performance for PrivL2

- We have decided to decrease the capacity and the block size and also decrease the associativity.
  - size went from 2*1024*1024 to 256*1024
  - bsize went from 32 to 8
  - assoc went from 16 to 4
- In addition to this, since the capacity was decreased, the hit delay and miss delays were reduced.
  - hitDelay went from 30-4 to 10
  - missDelay went from 30-4 to 5

Parameters for L2 cache for PrivL2



Explanation of how your changes fix the bottleneck

- For the L2 cache, the size was actually decreased quite a bit from the original value of 2 MB to 256 KB. By decreasing the cache size, we were able to deal with much smaller hit and miss delays in the second level cache.
- Thus although the L2 cache miss rate was relatively unchanged, having a smaller cache to search through to allow us to change the delays to be much lower allowed the IPC to increase. Exploring the 2 MB space previously was useless and created a lot of wasted time in just searching the cache that was not going to succeed in obtaining hits.
- This bottleneck was dealt with by just placing a much smaller L2 cache so that the exploration space was smaller. It will still get some advantage of having that 5% hit rate, but take way less time to do so due to the decreased capacity.

Why the architectural parameter changes are reasonable for L1 cache and L2 cache

- For the L1 cache, the size was not touched upon at all. The associativity and the bsize were changed. These values in associativity and bsize are also seen in the **Intel Ivy Bridge CPU** [4]. The latencies for 4 as the hit and miss delay are accurate, as many CPUs with varying sizes of 32 KB or 64 KB take around this much time.
- For the L2 cache, the size decrease allowed these hit and miss delays, which are seen in the **Intel Ivy Bridge as well as the Intel Westmere CPU** [5, 6].
- The associativity and block size are reasonable, as they do not require much hardware complexity to implement.

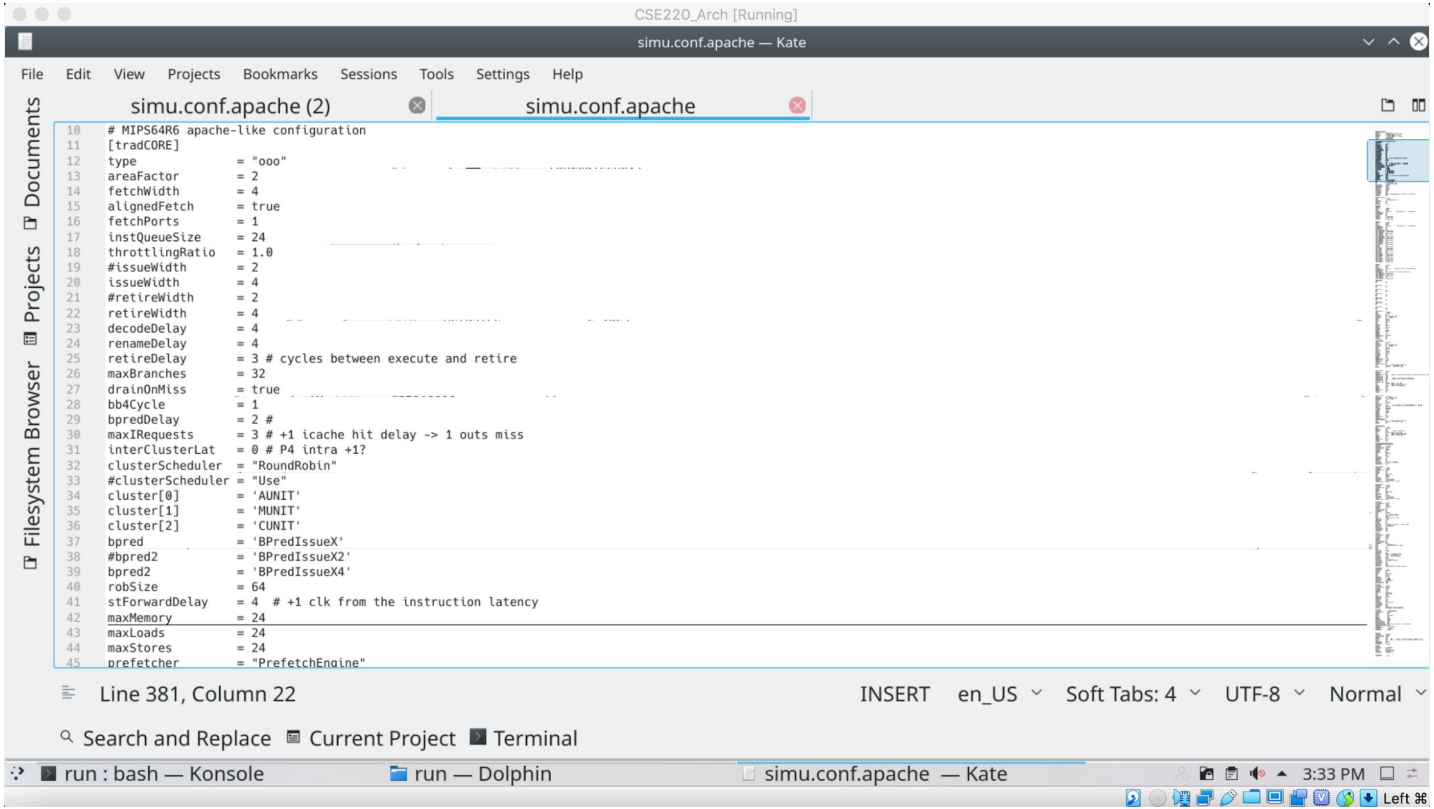
Improving upon Instruction Level Parallelism

- Lastly, it was recognized that the code itself has very few read after write (RAW) hazards to deal with with every insertion.
- In the insertion process, there is a check to see how to optimize the placing of keys as child nodes in the structure. Even if multiple insertions occur at the same time, future insertions will rearrange the tree such that the tree is correct. The last insertion will also ensure that the entirety of the tree is built and organized correctly.
- Although the instruction fetch is originally set to 4, the issue width and retirement width are only set to 2. This gives suspicion that a bottleneck appears because they are not all equivalent.

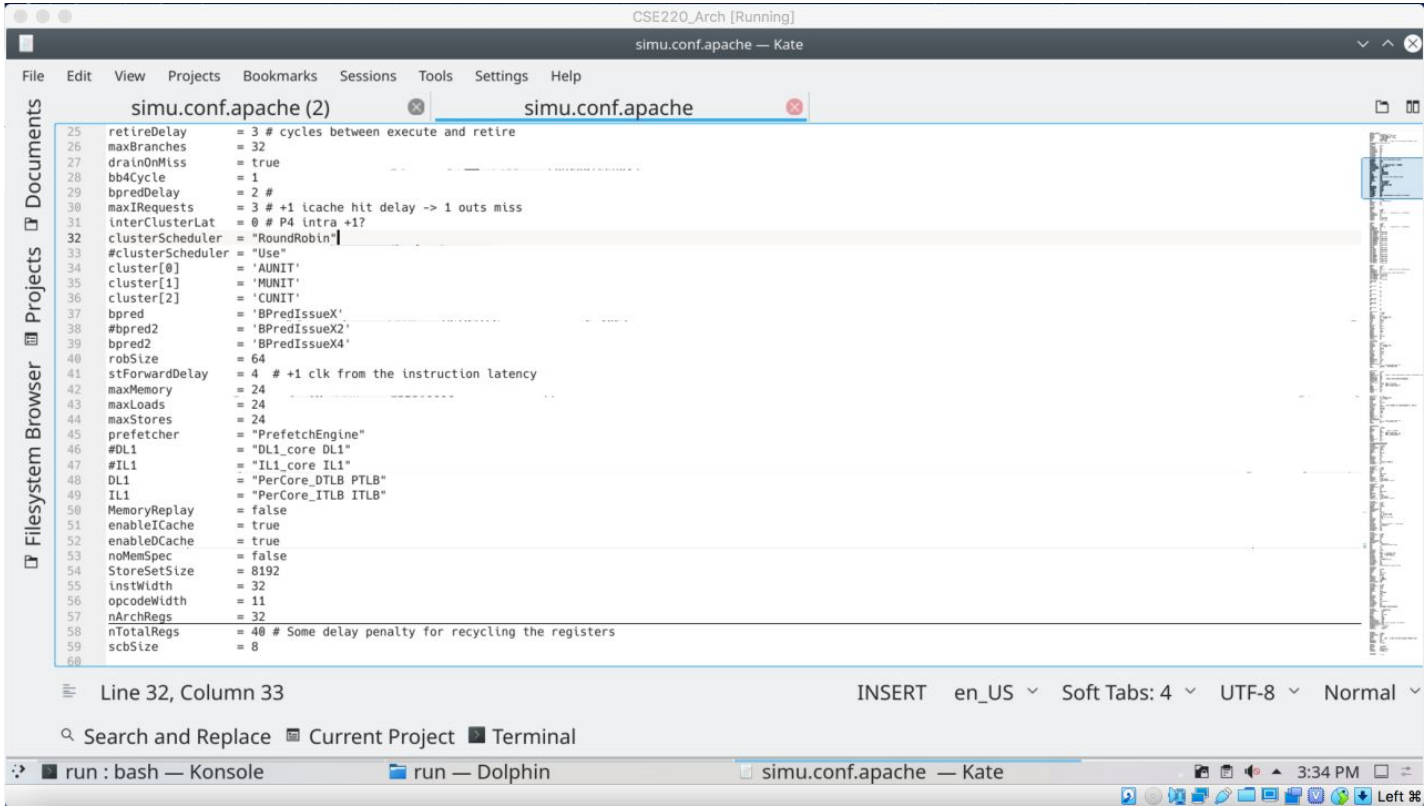
What architectural parameters you changed to improve performance ILP

- We have increased the issue width and retirement width to match the instruction fetch.
  - issueWidth went from 2 to 4
  - retireWidth went from 2 to 4

Parameters for tradCORE



Parameters for tradCORE (2)



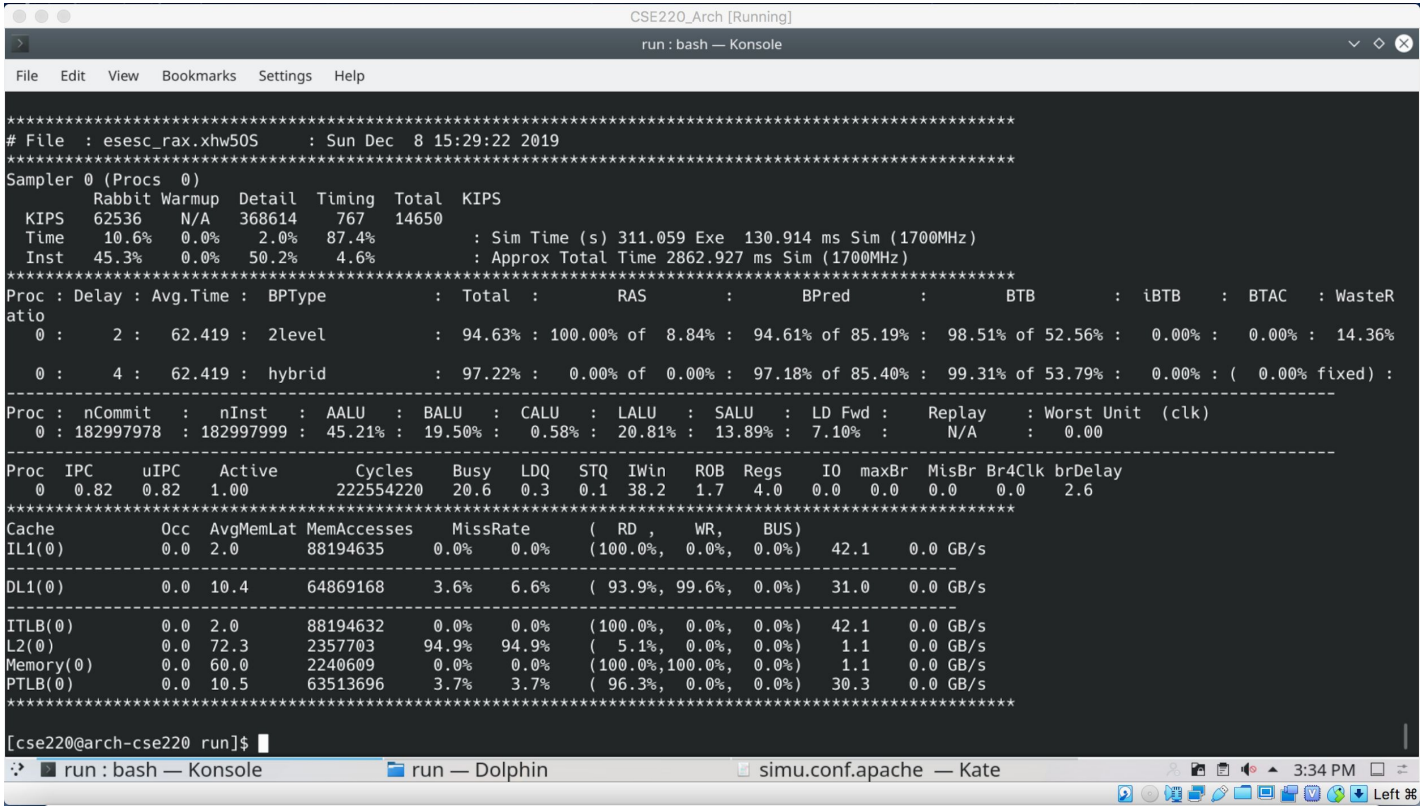
Why the architectural parameter changes are reasonable for issueWidth and retire Width

- It is seen in the **AMD Zen and Zen 2** that have issueWidths of 4 and 8 respectively [7]. Since these exist, it is reasonable to implement these changes.
- retireWidths have also been seen to have been increased from 2 to values of 4 to 8 [7].

What architectural parameters you changed to improve performance

- The following github link contains the simu.conf.apache file with all of the above proposed changes:
  - https://github.com/rahmanan4/CompArch/blob/master/simu.conf.apache
- The screenshots in the previous slides also show the proposed changes made.
- Final IPC: 0.82
- 0.82 / 0.54 = 1.52x speedup

ESESC report showing final results of improved IPC



References

- [1] “Introduction to Redis”, Redis, 14 November 2019, https://redis.io/topics/introduction
- [2] “Radix tree”, Wikipedia, 18 September 2019, https://en.wikipedia.org/wiki/Radix_tree
- [3] “Trie”, Wikipedia, 5 November 2019, https://en.wikipedia.org/wiki/Trie
- [4] “Inside the Ivy Bridge and Haswell BTB”, Matt Godbolt’s blog, 23 February 2016, https://xania.org/201602/haswell-and-ivy-btb
- [5] “Intel Ivy Bridge”, 7-cpu, 8 December 2019, https://www.7-cpu.com/cpu/IvyBridge.html
- [6] “Intel Westmere”, 7-cpu, 8 December 2019, https://www.7-cpu.com/cpu/Westmere.html
- [7] “Zen - Microarchitectures - AMD”, Wikichip, 8 December 2019, https://en.wikichip.org/wiki/amd/microarchitectures/zen