README

Project 5 Least Recently Used Analysis

Student Information:

Guillermo Rojas

Student ID: 008008657

https://github.com/CarnivoreXD/Least-Recently-Used-Study

Running my Project

When running from CLion IDE set the working directory for lru(this is the lru trace) and harness-lru as follows:

Target	Working Directory
`lru`	`$PROJECT_DIR$/src/trace-generators/lru`
`harness-lru`	`$PROJECT_DIR$/src/harness-lru`

Collaboration & Sources:

This work is primarily my own I used the following as troubleshooting resources and ideas:

-ChatGPT and Claude for debugging and understanding the project architecture. I also used this to understand JSON launch files and how to properly use them to run the project on VSCode since CLion uses .idea and CMake as the source of the configurations.

-GeeksForGeeks and StackOverflow for C++ syntax issues and questions

Implementation Details:

The trace generators creates files we can use to test the harness by simulating an LRU cache. it uses a list to maintain O(1) order and a hashmap for O(1) lookups. The program has an access stream consisting of 12N total accesses that is from 4N unique words. To make sure it is a good and valid test the words are shuffled using a fixed seed of 23 before being written to a file as a sequence of inserts and extracts. The hash table implementation uses open adressing with double probing and single probing to compare them. The table handles the deletions using tombstones and includes a compaction mechanism that makes the space more efficient.

Testing & Status:

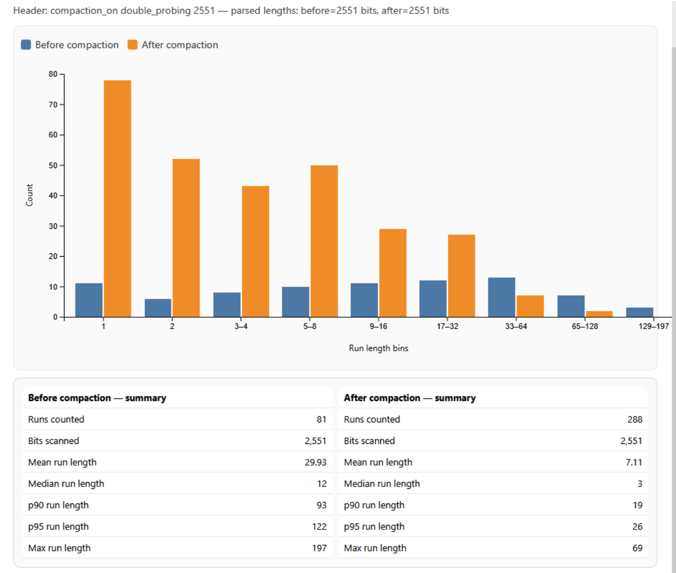
I coded everything on VScode and the way I tested my program I uploaded it to Blue ran it there with set JSON files and then ran it locally on my Windows 10 computer with a different set of JSON files, then on WSL on my system, then I finally sent it over to a mac to test my program and ran into issues where the directories werent being loaded automatically so I created an .idea folder to save the settings but was having issues but eventually settled with manually setting the working directory. For testing of my code I generated all the trace files and executed the harness to output the CSV and .txt files on all 4 systems with almost no issues (it would have been much easier with Java's JVM). I confirmed the CSV contained exactly 22 rows and that my data made sense. In the end, I imported the data into the D3 plotting tool to generate histograms which visually helped me analyze the clustering behavior.

Final report and anaylsis

LRU analysis
Going into this experiment my expectations were skeptical because the result from part 1 of the project made me realize that maybe the “most efficient” isn’t always the best in practice. I hypothesized that for small table sizes single probing would be hands-down faster because it moves linearly through memory which takes advantage of the CPU’s local caching and it being the fastest way to access memory. But as N got bigger then obviously its performance would take a hit in the form of clustering and taking longer to find a spot for the data. As for the double probing I expected it to be slower with a smaller table size due to the cost of having a second hash function but as N got larger it would be more efficient because double probing would not have the same issue as single probing when it came to clustering. The data showed that in practice it does not work out how I expected it to and it depended on how full the table was not just the size of the table.
When I looked at the elapsed time versus N the data showed three different phases which were at small scales of N single probing was indeed faster completing the trace in 14ms compared to 15.35ms for double probing. In the middle N range like at 65536, double probing took a small lead when it came down to speed which finished in 1050ms vs 1106 for single probing. What I did not expect that at the larger N sizes the trend reversed where when N was 524288 single probing was finishing in 2591ms and double probing was at 3757ms. This result occurred because the load factor at that size dropped to 21%. With the table mostly empty there were no clusters that it needed to avoid and double probing was wasting time by calculating a second hash function that was not needed.
So what I figured out from the data was that the trade-off between how much “work” an algorithm does and the actual time it takes is not always straightforward. For example, when N = 8192 the hash table was pretty full around 90% load. Single probing ended up needing way more probes per operation (18.63 on average) compared to double probing (7.77). From jus looking at the numbers it would seem that double probing was indeed faster, but it ended up

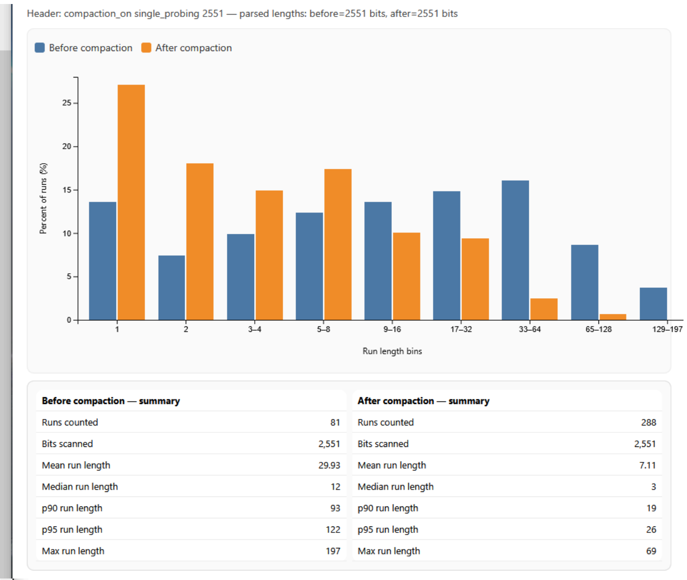
being slower 121.48ms vs 117.86ms for single probing which shows that double probing must save a lot of probes to make up for the overhead.

Finally, visualizing the tables helped me see why it worked out the way it did. In medium sized tables single probing created long clusters of filled slots so one would have to step through 20 slots just to find one that wasn’t taken. Double probing on the other hand spread things out more evenly so runs were generally short. But in a big empty table there was no need to jump around double probing where single probing works better because of the cheaper cost. That is why the probe counts and histograms looked identical except at large N’s because when there is plenty of empty space single probing has less overhead with better performance


Header: compaction_on double_probing 2551 — parsed lengths: before=2551 bits, after=2551 bits

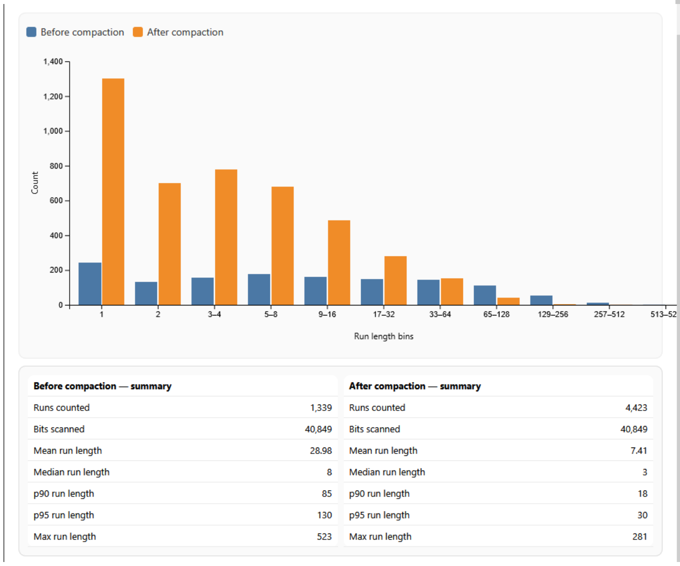
Before compaction — summary	
Runs counted	81
Bits scanned	2,551
Mean run length	29.93
Median run length	12
p90 run length	93
p95 run length	122
Max run length	197

After compaction — summary	
Runs counted	288
Bits scanned	2,551
Mean run length	7.11
Median run length	3
p90 run length	19
p95 run length	26
Max run length	69


Header: compaction_on single_probing 2551 — parsed lengths: before=2551 bits, after=2551 bits

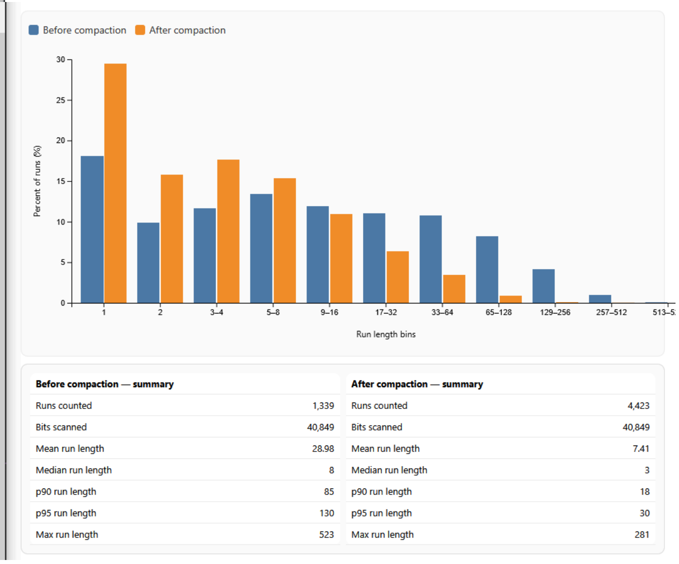
Before compaction — summary	
Runs counted	81
Bits scanned	2,551
Mean run length	29.93
Median run length	12
p90 run length	93
p95 run length	122
Max run length	197

After compaction — summary	
Runs counted	288
Bits scanned	2,551
Mean run length	7.11
Median run length	3
p90 run length	19
p95 run length	26
Max run length	69



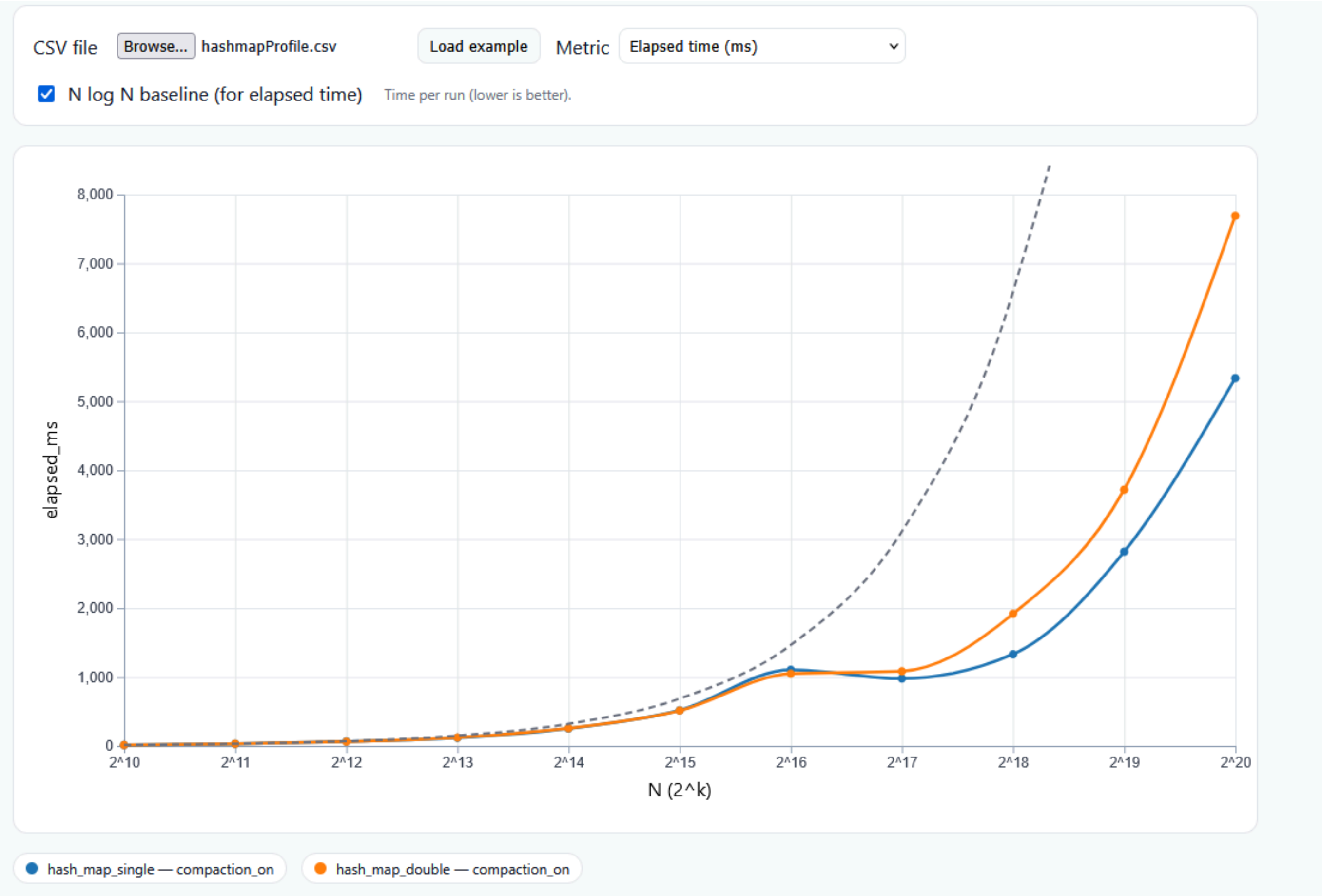
Before compaction — summary	
Runs counted	1,339
Bits scanned	40,849
Mean run length	28.98
Median run length	8
p90 run length	85
p95 run length	130
Max run length	523

After compaction — summary	
Runs counted	4,423
Bits scanned	40,849
Mean run length	7.41
Median run length	3
p90 run length	18
p95 run length	30
Max run length	281



Before compaction — summary	
Runs counted	1,339
Bits scanned	40,849
Mean run length	28.98
Median run length	8
p90 run length	85
p95 run length	130
Max run length	523

After compaction — summary	
Runs counted	4,423
Bits scanned	40,849
Mean run length	7.41
Median run length	3
p90 run length	18
p95 run length	30
Max run length	281



CSV file Browse... hashmapProfile.csv

Load example

Metric Elapsed time (ms)

☑ N log N baseline (for elapsed time)

Time per run (lower is better).