Tal Derei

CSE 375: Assignment 3
Open-Hash

Design Framework:

The ***sequential* implementation** of Cuckoo Hash incorporates a 2D vector of pointers to bucket structs containing keys (which also represents the value for simplicity). The single-threaded application initializes the 2D vector with 2 hash tables of size key_max, and populates them with key_max / 2 random values. The program calls a driver function which executes contains/inserts/removes API functions respectively. If the number of iterations drastically exceeds the number of keys, then the likelihood of encountering a resize/rehash operation increases accordingly (currently the field for detecting a loop is set to 10).

The ***concurrent* implementation** of Cuckoo Hash incorporates a 2D vector of smart pointers (shared pointers) to a bucket struct of atomic keys. The multi-threaded application moves shared pointers around rather than copying atomics, and also implements a sort of lazy method for removing elements by resetting the bucket's key value. Using atomics, threads are able to load and store keys atomically using std::load and std::store attributes, and enforce a global memory order (sequential consistency) with std::memory_order_seq_cst.

The ***transactional version*** used 'synchronized {}' blocks, which is a synchronization mechanism that combines groups of statements in transactions that are atomic (either all statements occur, or nothing occurs). But it's important to note that synchronized blocks are NOT transactions. Transactional portions of the code (the parts previously wrapped in locks) wrapped in __transaction_atomic {} blocks are transactions in the C++ STM reference docs. I initially implemented these 'atomic transaction statements' with the __transaction_atomic keyword, but had too many 'unsafe function' errors that weren't being solved with the 'unsafe_function' keyword, so I decided to switch to synchronized blocks.

In regards to rehash/resize operations, both the sequential and concurrent implementations implement the rehash() function as a single-threaded execution with single lock, locking the entire function and allowing only one thread to enter. Once inside the function, the thread locks all the buckets in the global hashtable_t struct (containing the individual hashtables) one at a time, and then performs the [1] **resize** and [2] **rehash** operations in order. The sequential and concurrent implementations differ in how they implement this function. Both create two separate temporary hash tables for storing values, resize the old hashtables, rehash the values in the temporary hashtable into the global hashtable, and then copy the values back into the appropriate index. But in the sequential implementation, .clear() is used to clear the hash tables, while in the concurrent implementation, a smarter technique is used: simply just adding new buckets to the appropriate newly sized locations.

Issues Encountered:

There were a number of issues encountered during testing. [1] Firstly, there were issues with .clear(), .resize(), and .shring_to_fit() operations in a multithreaded 2D vector. I suspect that some pointers were references to memory locations that were corrupted when clearing the vectors inside the rehash() function. This was solved by manually clearing and rehashing the hash tables by changing the values atomically to 0, and then pushing new buckets (rather than invoking the clear() and resize() functions). This doesn't seem to be a problem when defining 2 **individual** vectors, but seems to be an issue with a 2D vector definition. [2] There were deadlock and reference counting problems associated with my rehash() function in the concurrent implementation, again associated with the .clear() and .resize() calls. This was solved by removing those function calls, and redesigning my rehash() function by stripping down all the program components until I found the issue.

Performance Results:

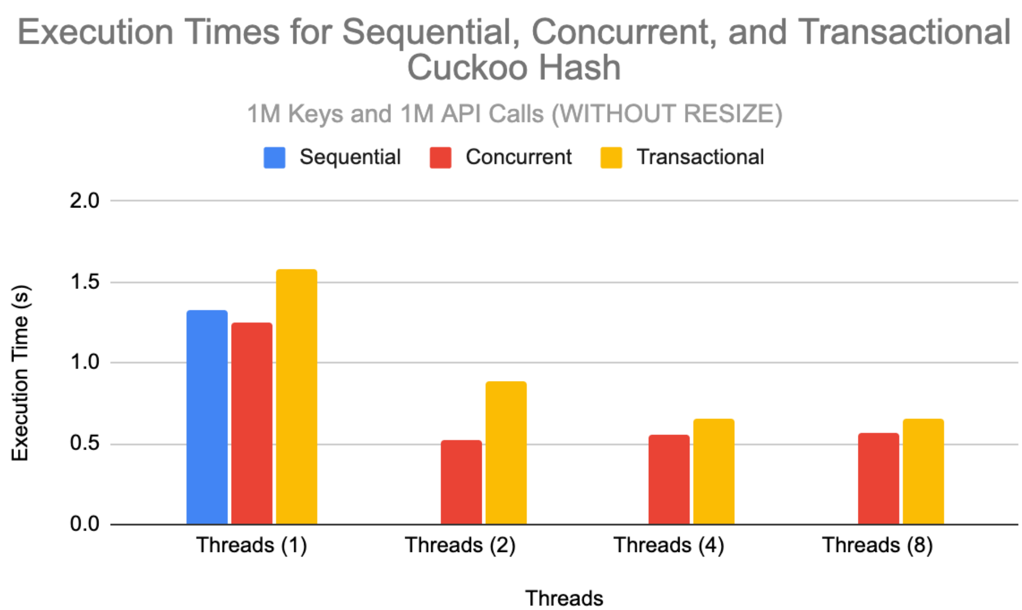

Figure 1: Sequential, Concurrent, and Transactional Implementation WTHOUT resize/rehash operations

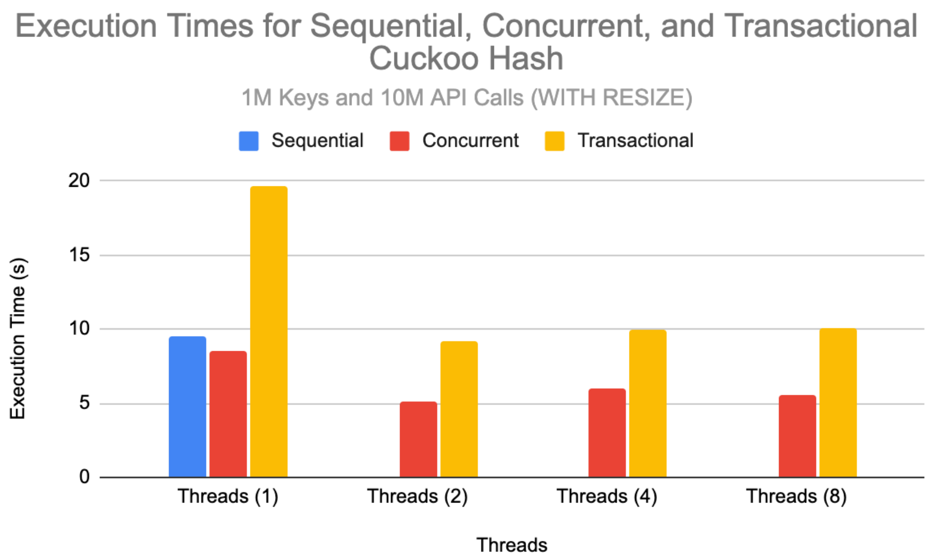

Figure 2: Sequential, Concurrent, and Transactional Implementation WITH resize/rehash operations

The results indicate that resizing and rehashing all the hashtable elements is an EXPENSIVE operation since it's single-threaded, resulting in a **10x** slowdown across the board from the sequential, concurrent, and transactional applications. The transactional implementation suffered the largest hit, since it's also using transaction-based synchronization primitives, which results in a larger slowdown in the overall program execution compared to both the sequential and concurrent implementations (as shown above).

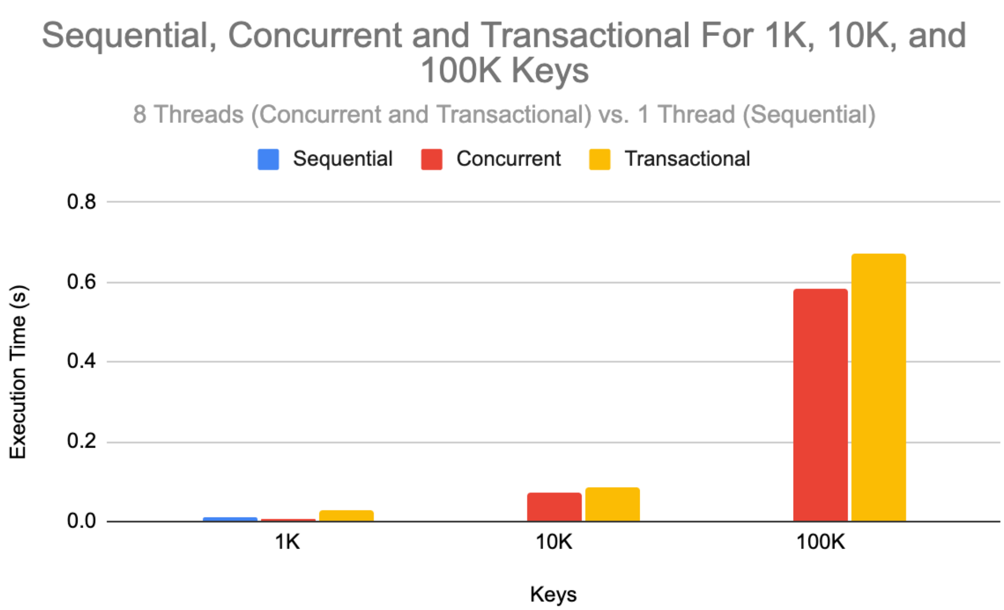

Figure 3: Sequential, Concurrent, and Transactional Implementation For Small Keys

Threads	Sequential	Concurrent	Transactional	Key Size	Data Structure
Threads (1)	1.32536	1.2523	1.58246	1M	W/O RESIZE
Threads (2)		0.525823	0.880877		
Threads (4)		0.555609	0.650723		
Threads (8)		0.565063	0.656737		
Threads	**Sequential**	**Concurrent**	**Transactional**	**Key Size**	**Data Structure**
Threads (1)	9.52855	8.5301	19.6046	1M	W/ RESIZE
Threads (2)		5.16483	9.13887		
Threads (4)		6.03343	9.96289		
Threads (8)		5.59087	10.052		
Keys	**Sequential**	**Concurrent**	**Transactional**		
1K	0.00861758	0.00601383	0.0287665		
10K		0.0725114	0.0841181		
100K		0.583123	0.673023		

Figure 4: Data Collection

In regards to the console output, executing the program prints the following to the console, for example. The distribution of key is as expected.

```
root@84ea0235a59c:~/open-hash# ./obj64/main.exe -a sequential -b 500 -c 5000 -d 1
Loop detected. Triggering RESIZE AND REHASH!
------------------------------------------------------------------------------------------------------------------------
Number of 'lookup' operations: 4094
Number of lookups succeeded: 221
Number of lookups failed: 3873
Number of 'insert' operations: 416
Number of inserts succeeded: 396
Number of inserts failed: 20
Number of 'remove' operations: 51
Number of remove succeeded: 31
Number of remove failed: 459

First and Second Hashtables:
0 362 0 0 191 94 0 0 0 0 357 227 0 0 0 0 0 0 261 478 0 0 243 0 260 0 0 0 492 208 0 184 243 0 0 0 307 0 0 422 304 58 238 406 96 0 141 271 142
159 0 38 0 0 317 309 439 69 0 441 0 0 223 0 0 318 0 392 237 342 0 222 0 0 0 249 435 0 33 0 341 236 44 98 307 0 0 0 355 0 220 321 0 306 0 0 0
0 0 0 247 456 0 193 60 0 466 455 150 0 0 416 0 0 279 0 0 0 0 5 0 0 0 0 0 437 0 371 490 158 352 0 206 301 244 0 0 0 0 0 496 0 0 0 191 174 0 70
 0 226 0 0 11 83 378 0 275 0 0 244 0 0 26 0 68 474 0 280 134 0 70 290 334 221 0 386 210 0 0 176 0 0 0 0 423 0 0 24 0 82 325 453 0 0 241 0 426
 0 60 97 419 0 0 438 0 0 417 124 361 1 498 425 0 397 219 97 120 457 0 0 428 418 0 0 0 0 0 0 0 0 0 0 0 0 365 55 0 296 496 0 100 203 417 0 217
265 0 141 187 0 139 0 443 0 0 0 284 0 256 0 225 127 450 334 0 57 221 0 312 0 258 0 0 0 333 442 0 160 0 0 387 0 0 0 493 66 0 328 462 0 100 172
 466 15 0 0 365 212 401 0 232 214 0 0 0 0 91 0 0 442 0 374 232 0 291 109 0 18 288 0 0 0 0 0 0 0 0 0 0 315 0 499 0 118 88 0 30 0 0 0 496 349 0
 0 0 433 408 314 0 458 0 14 0 0 149 337 0 335 0 234 0 0 0 364 0 0 0 105 0 0 429 362 0 0 444 274 0 11 0 0 407 0 183 297 309 0 0 283 0 0 73 344
 53 0 0 0 0 416 0 0 0 389 0 0 0 389 115 316 0 0 313 0 486 25 167 0 376 0 198 0 225 0 0 0 0 136 431 143 0 0 0 0 142 489 292 0 226 40 0 141 60
295 217 0 158 67 469 0 0 0 0 110 247 0 249 0 0 351 0 0 0 396 29 431 0 386 499 86 379 355 0 371 0 0 0 0 279 0 23 75 0 0 0 96 0 348 220 0 377 1
17 340 0 0 0 429 112 97 0 471 0 0 269 400 0 0 423 202 0 129 0 0 0 253 0 470 0 0 0 0 12 0 258 0 0 236 0 0 0 0 0 33 43 90 398 270 0 177 0 0 0 1
58 0 0 155 213 0 48 345 347 0 0 372 0 0 94 0 0 297 0 0 0 0 0 234 358 0 0 0 0 0 0 268 0 0 0 449 9 0 152 492 433 303 402 155 0 456 0 318 12 0 0
 0 375 81 0 39 0 240 0 470 448 0 329 0 269 211 0 0 0 0 465 309 0 0 287 361 0 352 0 345 0 371 0 462 0 0 34 0 28 50 0 0 0 0 0 390 495 56 0 135
47 0 172 31 0 0 205 0 0 164 189 257 396 0 0 0 93 0 0 0 0 453 0 0 391 68 179 0 0 0 0 0 457 478 0 0 216 0 0 0 0 0 0 394 0 0 0 0 439 0 0 0 409 0
 0 0 0 0 111 0 0 4 0 408 0 358 0 183 0 0 0 0 0 0 0 0 0 0 0 0 0 430 236 0 441 0

390 0 0 0 0 0 209 0 127 0 0 0 0 0 0 0 0 0 0 0 444 0 0 0 177 0 206 289 0 0 0 0 437 0 0 0 0 0 0 0 416 0 275 0 0 421 451 0 30 0 396 0 0 0 0 0 0
0 335 0 0 0 16 0 0 0 187 0 0 347 0 491 64 100 0 0 0 0 0 162 0 370 0 462 302 173 487 382 0 0 228 0 0 0 0 0 0 230 0 217 0 308 229 0 160 0 163 2
42 0 387 239 0 3 0 0 0 0 0 385 0 220 0 0 0 0 161 249 25 343 451 0 425 0 0 0 0 0 0 0 0 168 0 0 83 3 0 0 0 60 443 0 237 229 0 0 0 168 0 147 143
 0 356 152 0 0 0 0 0 0 0 87 255 0 0 0 0 0 0 0 7 0 0 0 0 0 114 0 0 477 145 0 0 465 404 406 0 38 0 0 0 0 0 357 0 0 0 400 384 85 0 0 0 0 79 0 0
0 0 0 0 74 0 0 0 0 0 0 0 0 340 0 0 225 0 155 0 323 0 0 0 0 0 0 0 317 0 0 0 0 0 0 0 0 0 0 0 0 464 0 0 0 0 428 0 165 0 183 18 287 0 0 113 20 0
121 62 0 0 0 0 266 0 0 149 0 268 0 327 0 0 314 0 397 0 0 0 0 0 301 0 0 0 0 0 308 0 0 0 0 47 189 0 0 54 347 0 0 398 0 0 0 0 247 0 0 224 0 452
0 0 201 4 0 0 0 0 23 0 0 0 98 0 53 0 336 73 477 439 0 162 0 0 0 0 154 0 0 312 128 0 341 0 0 0 101 0 494 258 0 0 87 0 0 43 36 0 0 0 0 0 0 309
73 0 0 0 0 0 0 0 0 140 337 412 0 279 0 289 0 0 0 488 0 0 0 163 0 0 0 0 0 338 0 238 0 0 0 0 0 472 359 0 0 0 0 167 0 0 0 484 54 19 194 118 0 0
0 0 0 0 0 74 0 0 0 219 130 223 0 0 0 0 0 0 0 116 0 0 0 0 0 0 0 0 0 372 21 293 0 0 0 0 401 0 96 0 0 0 0 0 188 0 0 0 0 0 487 185 101 326 0 0 0
0 0 0 0 0 0 0 0 208 0 0 169 0 256 0 0 0 0 485 0 0 281 0 0 31 350 269 0 104 0 193 0 365 0 0 125 0 0 0 423 0 0 0 0 439 0 0 0 0 402 113 0 0 0 41
2 348 0 0 467 230 0 41 0 0 0 0 90 0 0 435 0 0 99 0 425 0 495 0 332 280 157 0 0 0 0 251 0 0 0 0 0 0 0 0 0 357 0 0 320 358 0 51 129 117 493 461
 490 0 0 0 0 0 330 391 0 0 0 0 215 0 88 0 0 0 0 112 0 0 415 0 0 262 479 0 261 144 214 0 0 176 0 0 0 476 288 136 58 0 0 466 0 0 0 0 303 125 0
278 0 0 0 0 0 231 0 0 318 234 118 7 331 290 285 0 0 0 133 0 0 0 259 6 0 0 491 0 0 0 328 0 453 0 0 0 0 0 11 0 295 0 486 52 429 181 0 0 148 310
 355 119 0 0 0 0 0 315 0 0 67 199 349 63 411 111 0 0 0 0 302 399 0 0 430 0 273 0 277 0 0 65 0 150 0


Size of first hashtable: 725
Size of second hashtable: 736
Number of rehash operations executed is: 1
------------------------------------------------------------------------------------------------------------------------
Execution time elapsed is: 0.00694033
root@84ea0235a59c:~/open-hash#
```

Figure 3: Sample Console Output

***
Note: The transactional version is compiled separately and includes its own main() function.
Compile with ‘g++ --std=c++17 -pthread -fgnu-tm transactional.cc -o transactional’
***