Hashtable Optimization Log

by Belyaev Andrey Alexeevich, 2021

This document covers the process of development and subsequent optimization of the hashtable implementation present in the same repository as this report. This research was not sponsored by any fund or organization, but if you'd like to become a sponsor of this or any further research, please contact me via email at admin@localhost.

Initial development

I decided to implement the open addressing with linear probing and a step of 1 in order to maintain the cache friendliness from the very beginning. For that same reason I decided to store the keys in-place and with a fixed length of 64, without any extra pointers. Among other key design decisions was: storing the last result in a mutable field, as well as returning it from every function where applicable (for convenience). I also decided to support deletions via marking cells with a special value of the value pointer. Other than that, I implemented a debug dump (initially in graphviz dot format, however it turned out not to be a feasible option, as it didn't support windows-1251 encoding, and even with the transliterated dictionary, it failed due to the sheer size of the input file; then in a human-friendly format for testing purposes, then in a machine-friendly format for hash efficiency measurement).

```
def estimate(name):
    with open(name) as f:
            data = f.read()
    distr = {}
    for i in list(map(int, data.split())):
            distr.setdefault(i, 0)
            distr[i] += 1
    return sum(distr.values()) / len(distr)
```

Figure 1: The hash estimation

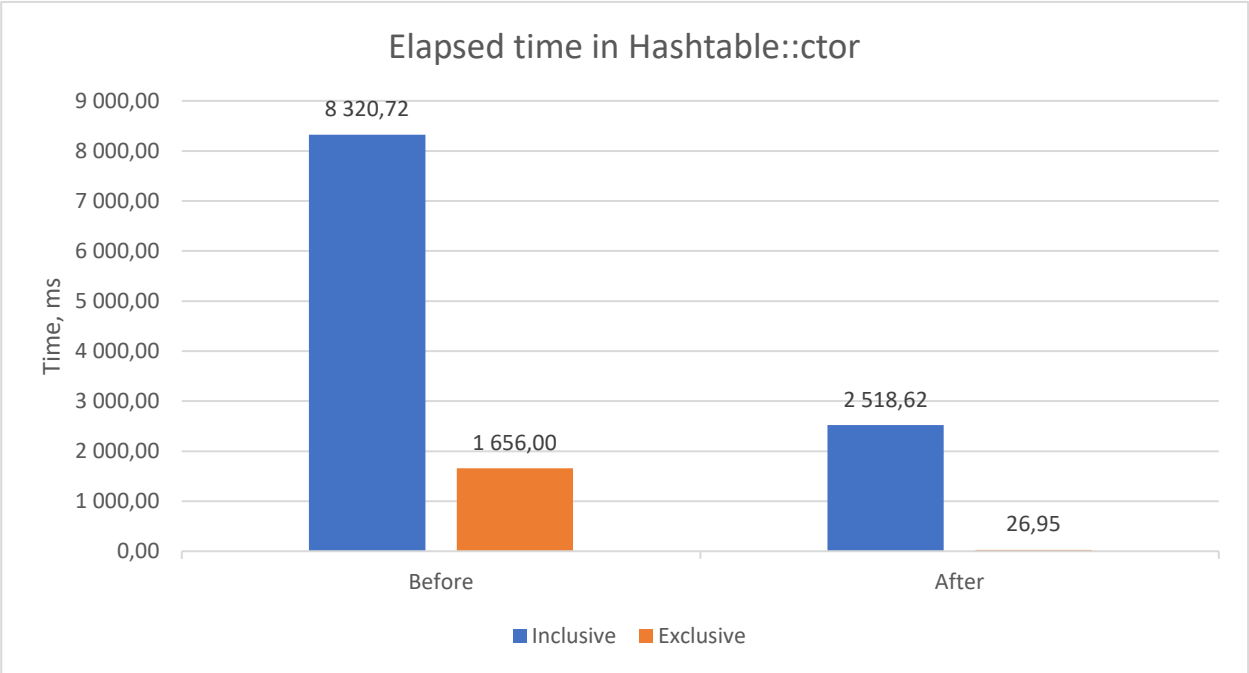
The two hashing algorithms I tried employing were crc32 and fnv-1a. The first is very widely spread and has a form of a machine instruction, but the second is specifically designed for hashtables and is considered very fast. To estimate their performance, I defined the efficiency of a hash to be the average number of collisions per hash value among the present hashes. Then crc32 (polynomial 0x1EDC6F41, a.k.a. crc32c) has a score of 2.26, whereas fnv-1a – 1.68. However, switching the capacity from 32+exponential incrementation to a fixed 2*entry count from the very beginning didn't impact fnv-1a petty much whatsoever – its score in fact improved down to 1.58 – but severely struck the crc32, increasing it to a horrendous value of 1129.44, showing that it might turn out to be very bad as a hash when divided by a wrong modulo.

This seems to be all the key points covered, so we shall capture a performance report and move on. For the performance test, we read the dictionary from a file (approx. 1e5 entries) and then perform two loops of get's: one from an array of keys present in the hashtable, for 1e5 iterations; another – for keys absent from it, for 5e3 iterations. The performance report from this state is saved as *Report_PreOptimization* (.vsps and _*.csv) (bear in mind that all times are increased due to profiled code instrumentation).

main	1	100,00%	0,00%	36 892,11	0,02	HashTable.exe
testMixed	1	72,12%	0,00%	26 607,06	0,67	HashTable.exe
abel::Hashtable::get	5 000	72,12%	72,11%	5,32	5,32	HashTable.exe
abel::Hashtable::hash	5 000	0,01%	0,00%	0,00	0,00	HashTable.exe
printf	2	0,00%	0,00%	0,20	0,00	HashTable.exe
fnv1a_64	5 000	0,00%	0,00%	0,00	0,00	HashTable.exe
std::chrono::steady_clock::...	2	0,00%	0,00%	0,00	0,00	HashTable.exe
std::chrono::operator-<__i...	1	0,00%	0,00%	0,00	0,00	HashTable.exe
__security_check_cookie	1	0,00%	0,00%	0,00	0,00	HashTable.exe
abel::Hashtable::ctor	1	22,55%	4,49%	8 320,72	1 656,00	HashTable.exe
abel::Hashtable::set	157 372	7,88%	7,25%	0,02	0,02	HashTable.exe
abel::Hashtable::hash	157 372	0,63%	0,54%	0,00	0,00	HashTable.exe
abel::FileBufIterator::next	4 300 173	4,57%	3,65%	0,00	0,00	HashTable.exe
abel::FileBufIterator::isEof	4 614 918	4,14%	3,15%	0,00	0,00	HashTable.exe
abel::FileBufIterator::cur	4 300 173	1,15%	1,15%	0,00	0,00	HashTable.exe
abel::FileBufIterator::skipS...	157 372	0,31%	0,27%	0,00	0,00	HashTable.exe
abel::FileBufIterator::getPtr	157 372	0,02%	0,02%	0,00	0,00	HashTable.exe
abel::Hashtable::ctor	1	0,00%	0,00%	0,02	0,00	HashTable.exe
__security_check_cookie	1	0,00%	0,00%	0,00	0,00	HashTable.exe
abel::FileBufIterator::ctor	1	0,00%	0,00%	0,00	0,00	HashTable.exe
abel::FileBufIterator::dtor	1	0,00%	0,00%	0,00	0,00	HashTable.exe
testGood	1	5,20%	0,02%	1 918,92	8,13	HashTable.exe
abel::Hashtable::get	100 000	5,17%	5,07%	0,02	0,02	HashTable.exe
fnv1a_64	100 000	0,01%	0,01%	0,00	0,00	HashTable.exe
printf	2	0,00%	0,00%	0,17	0,00	HashTable.exe
std::chrono::steady_clock::...	2	0,00%	0,00%	0,00	0,00	HashTable.exe

Optimizations

As it can clearly be seen from the performance report, the primary bottleneck is Hashtable::get on mixed keys. It is a predictable outcome, as this is the function that has to iterate over the longest chains during lookup of absent keys. Due to this, we shall postpone optimizing this exact function, and for now move to the next candidate. Hashtable::set is the second longest function which suggests that there may in fact be what to optimize there (as well as in get, in that case, due to their major similarity), but for now I'd like to postpone this as well. One group of functions that takes a particularly long time, but probably shouldn't, is FileBufIterator methods next, cur and isEof. The problem with these functions seems to be that they weren't automatically inlined, as they were intended to. We could easily solve this without even resorting to assembly. And voila – doing so has already brought dramatic improvements:



The profiling report from this stage is saved as *Report_OptimizeFBI*.

The next thing we could easily optimize is the hash function. (I'm still postponing the set and get, because they scare me). During the optimization I found a small bug in my fnv implementation, but it didn't change the collision score, so it is unlikely to affect performance. On the other hand, the original fnv code was full of extraneous memory references, which we were able to get rid of in out assembler variant. Profiling this exact function became somewhat troublesome, however, as code instrumentation worked very poorly on our raw assembler function (possibly due to it lacking a stack frame). Thus, I captured performance reports without instrumentation before and after the change, and the time difference was roughly 5 percent. The profiling report from this stage is saved as *Report_OptimizeHash*. Before moving on to the next point, I decided to test the efficiency of crc32, since it could be implemented with no loops whatsoever in mere 10 assembler lines. Its performance was flamboyant: a flat 34% decrease in time. It was quite unexpected, since, according to the profiler, the hash didn't play that major of a role in execution time, but it may be down to inlining confusing the profiler, or the collision count somehow decreasing. In any case, this is obviously a better solution. The profiling report from this stage is saved as *Report_OptimizeHash2*.

But now something I have been avoiding for a long time: optimizing set and get for hashtable. As figure 2 shows, more than half the time in it is to be blamed on lines 123 and 132. What we shall do to optimize this is rewrite part of the code in inline assembly. First of all, memcmp: we could replace it with a few avx comparisons. However, while doing this I have encountered a significant problem: MSVC cannot handle inline assembler on x64. Very well, let us then switch to gcc. After quickly throwing together a

```
194 (13,70%)  122     for (;; curInd = (curInd + 1) % capacity) {
533 (37,64%)  123         if (buf[curInd].value < NODE_SPECIAL) {
              124             if (buf[curInd].value == NODE_FREE) {
              125                 lastResult = R_NOTFOUND;
              126                 return nullptr;
              127             }
              128
              129             continue;
              130         }
              131
289 (20,41%)  132         if (memcmp(key, buf[curInd].key, KEY_LEN) == 0) {
              133             lastResult = R_OK;
              134             return const_cast<mvalue_t>(buf[curInd].value);
              135         }
```

Figure 2: Hashtable::get bottlenecks

makefile, we are ready once again to optimize, this time also being able to employ O1 and enjoy some inlining and such. Now that everything is set up to work with gcc, I guess I may as well profile it using callgrind. (The gcc port is located in .gcc, the linux adaptation – in .linux). Callgrind supports our opinion that line 132 is the primary bottleneck. It calls __memcmp_avx2_movbe, but the problem is that the call cannot be inlined. This can be helped by providing our own implementation (and straight away the same is applicable to set and del). And indeed – doing so boosts the total execution speed by approximately 25% (results are shown in figure 3). At first glance it may seem that get's exclusive time increased almost twice, but that is because what used to a separate __memcmp_axv2_movbe function is now inlined. Overall, I believe this to be a very significant improvement.

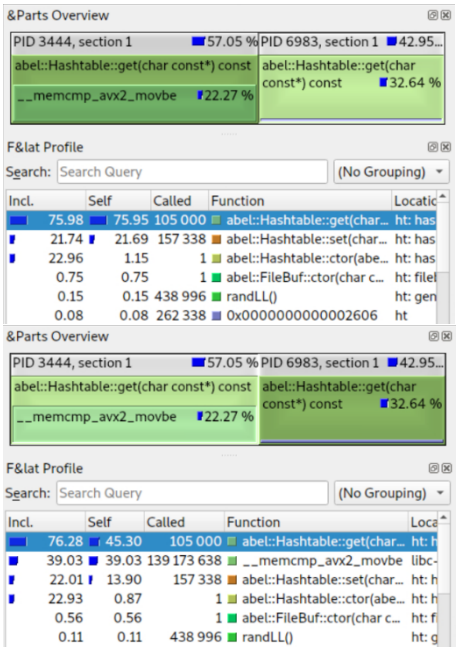


Figure 3: Effects of memcmp optimizations (bottom is before, top - after)

Summary

While the immediate execution times are incomparable between the first and the last development iterations, we may now restore the code to its original form and measure the speeds with and without optimizations. The program's own time statistic shows 49.37ms vs 31.28ms on present keys and 316.2ms vs 145.6ms on absent, which is a drastic improvement in my opinion. Callgrind shows a 60% improvement (I'm comparing against the program after the fbi optimization, since the former was achieved only through raw C means, and could have been optimized from the very beginning. Among the side effects of this research was me coming up with a way to out-source c++ functions (including methods, template instances and overloads) into assembler while maintaining compatibility with several compilers (although I haven't had a chance t employ it). Moreover, I have gained invaluable experience in writing platform-independent programs, as well as in utilizing profilers and debugging tools under different operating systems. Last but not least, I have obtained a potentially incredibly useful library (which could, for example, significantly speed up my assembler from the previous semester. Overall, I believe this project to have been beneficial even if no one ultimately pays me.

References

1. **Bryant, Randal E. and O'Hallaron, David R.** *Computer Systems: A Programmer's Perspective, 3rd Edition.* 2010.

2. **Belyaev, Andrew Alexeevich.** Github repository with the corresponding code. [Online] 2021. https://github.com/abel1502/mipt_2s/tree/master/ded32/HashTable.