The Grand Optimisation Research

Климов Артем, Б05-033

In an attempt to speed up the hash table

Chapter 1: Inventory. What are the introductions?

The initial goal is to use an existing old implementation of the linked list from the first semester to write a hash table on chains and further optimize it by partially rewriting the program in assembly language. We need to beat the compiler with disabled optimisations (-O0). The input data is supposed to be a Russian-English dictionary with 150 thousand key-value pairs.

The first table benchmark showed an impressive result. It took 2 seconds to read the file. Slow enough, but afterwards it turned out that due to a bug in the software, the table was not resizing as it filled up.

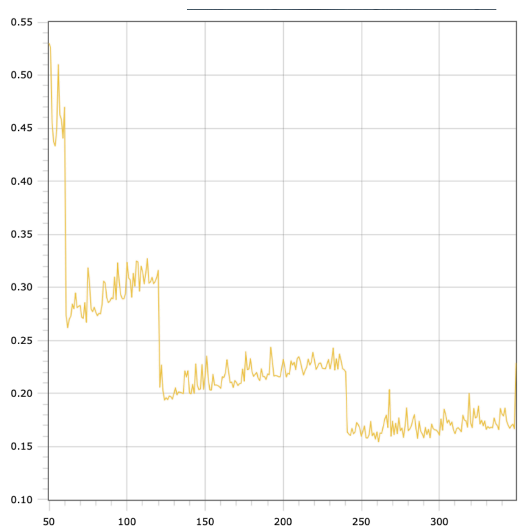
5 minutes of debugging later, the rehash mechanism was fixed and the table performed four times better: 0.55 seconds with a set load factor of 50%.

Before optimizing the table with assembler inserts, I thought it was a good idea to optimize the table parameters and recreate the "tried everything, it's still slow" situation. However, in this case it is not clear why they didn't try -O3, but we assume that this option is forbidden by the penal code.

Chapter 2: Academic Reading

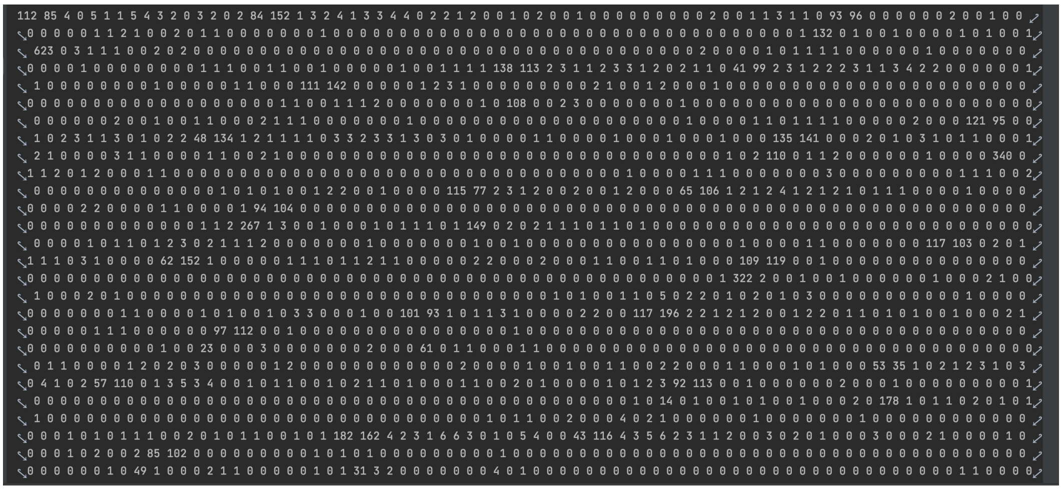
Why did I choose 50% for the first test? Well, I took the number that first came to mind. That's tentative. According to information on the Internet, values between 70 and 95 percent are usually used. The optimal value can be very different depending on the input data. So there is a very small chance that my 50% is optimal.

The easiest and most obvious way to see the dependency of algorithm's efficiency on some constant is to plot it. After Iterating through all possible integer load factors from 50% to 350%, I got some interesting results:



In my case, the optimum value was neither 70% nor 95%, but all of 250%.

Next I checked how my hash function worked. And I was disappointed.
In the picture below, each number is the length of the corresponding chain.



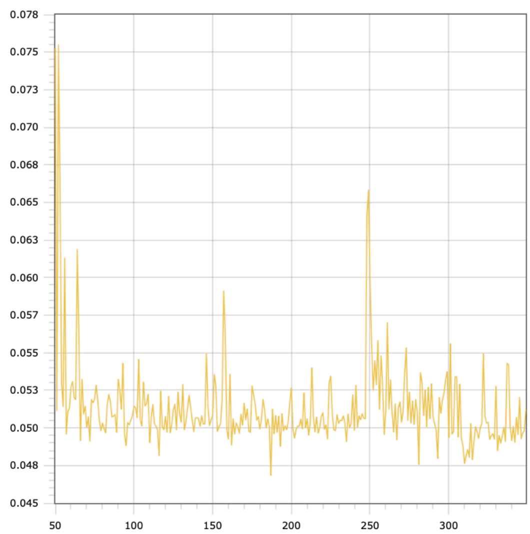
Looks terrible. The greatest efficiency is achieved when the numbers are more or less the same.

After a moment the old hash function was trashed, and its place was now taken by FNV1A with a more even distribution (the average non-zero chain length was 1.29). The network says that fnv1a is the best choice for a hash table. The acceleration was between 4-20% on different loading factors), but the appearance of the graph did not change.

The next improvement is not to recalculate the hash every time the table size changes, but to store it in the internal chain structure. This made it possible to speed up writing to the table by a another 21%.

I also found out that this kind of staircase in the graph is normal for chained hash tables, if only writing operations are benchmarked. When the table is resized, it should be fully copied. As soon as the load-factor becomes greater than some value, the table is no longer expanded by a factor of two. This is an expensive operation, which takes 94% of the time to write to a table with a load-factor of 70 (according to CLion's built-in profiler), so it takes 1 / ((0.94 / 2) + 0.06) = 1.88 ~ twice as fast.

If you create a hash table of the right size at once, the graph becomes like this:

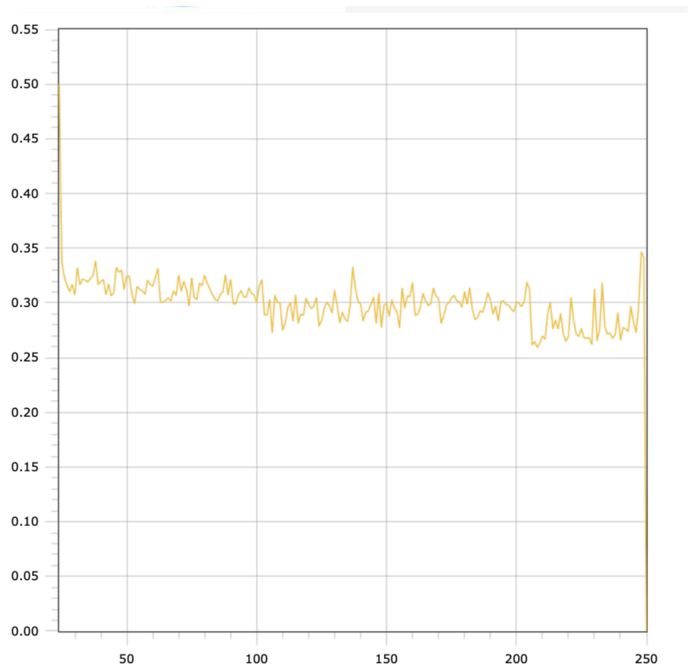


The times are almost identical. This was expected, as the rehash function is now not called and cannot affect the time.

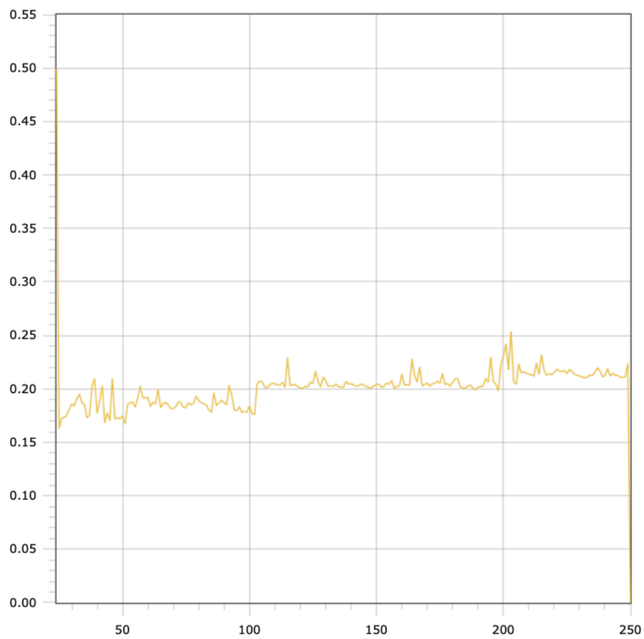
Chapter 3: Academic Writing.

At this point we have considered only the write speed of the table. To have a better idea of what values you will have to work with when optimising, you should also check the reading speed, which usually happens more often than the writing speed.

As an example, we measured the read times of a million existing random keys from a table with different load-factor values.



The results are about the same, but it is noticeable how the speed of reading values from existing keys increases as the load factor increases. However, if you read non-existing keys from the table, the situation changes dramatically.



If you give a table a non-existent key, the probability that there is no hash for such a key is quite high. Therefore, the table will be able to quickly understand that the chain is empty, and this element does not exist in the table. However, when the load factor is greater than 100%, each chain would contain at least one element, so the table would have to compare keys by their values, which already slows down the program considerably. The graph clearly shows a "step" around the 100% mark.

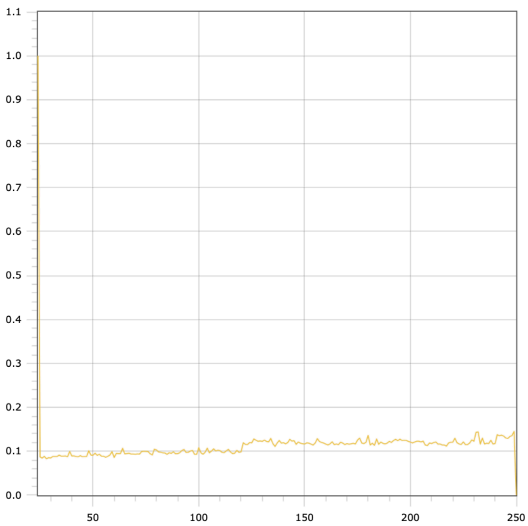
Chapter 4: Diving deep into optimisations

Below is the result of the profiler for a million table read operations. The obvious bottleneck in the program is the hash function.

```
˅ 100.0%   fast_hash_table_clion`main
  ˅ 99.5%   fast_hash_table_clion`doReadBenchmark
    ˅ 99.5%   fast_hash_table_clion`hash_table::get
      ˅ 98.2%   fast_hash_table_clion`hash_table::get_pair
          42.8%   fast_hash_table_clion`hash_table::hash
          11.6%   libsystem_platform.dylib`_platform_strcmp
          < 1%   fast_hash_table_clion`hash_table::hash
          < 1%   libsystem_platform.dylib`_platform_strcmp
        < 1%   fast_hash_table_clion`hash_table::get
```

The current version of the hash function counts one character per cycle. This is inefficient because the processor can compute hashes over 64-bit numbers in a single clock cycle using the crc32 instruction, i.e. it can compute 8 characters at a time. However, this creates its own inconvenience: the key must be of length divisible by 8. Let the length of the key always be 64 characters. This way, we won't need to use a loop at a critical point of the program. Keys from the file don't exceed 64 characters in length. Let's consider that we optimize the program sacrificing abstractness and still being able to solve an initially given concrete problem.

Rewriting the hash function body into inline assembly made it more than 40 times faster. In addition, memcmp is now used for key comparisons, which is also much faster than strcmp. The hash function itself now takes less than 1 percent of the get_pair function's running time. Key comparisons are also rare. This means that the number of collisions is minimal. (The average non-empty chain length is 1.331)




```
Method
˅ 100.0%   fast_hash_table_clion`hash_table::get
  › 97.8%   fast_hash_table_clion`hash_table::get_pair
    < 1%   libsystem_platform.dylib`_platform_memcmp
    < 1%   fast_hash_table_clion`hash_table::hash
```

Interestingly, despite official claims, the results are twice as bad with FNV1A. For the same number of collisions, the hash running time is 76% of the get time. The total running time doubles - 0.23s vs. 0.11s. However, this figure cannot give us a real estimation of hash table running time now, since reading time from it becomes comparable to the running time of random key generator.

The only thing left to optimise is the get_pair function. It takes an unreasonably long time, even though it seems to be just accessing the chain by index.

```cpp
160 mlist_element_type hash_table::get_pair(const char* key) const {
161
162     unsigned long long key_hash = string_hash(key);
163     unsigned long long chain_index = key_hash % capacity;
164
165     mlist* list = this->lists + chain_index;
166     mlist_element_type* storage = list->element_array;
167
168     mlist_index head = list->head;
169     mlist_index tail = list->tail;
170     mlist_index* next = list->next_indices;
171
172     for(int walker = head; walker != tail; walker = next[walker]) {
173         mlist_element_type* entry = storage + walker;
174         if(memcmp(entry->key, key, n: 64) == 0) {
175             return *entry;
176         }
177     }
178
179     return {
180         .key = nullptr,
181         .value = nullptr,
182         .hash = 0
183     };
184 }
185
```

For the sake of experiment I decided to remove lines 168-177 from the program. The speed of get_pair remained almost the same.

Having opened the disassembler, I understood the reason for such a long running time - dozens and dozens of unnecessary memory accesses.

```
(lldb) dis
fast_hash_table_clion`hash_table::get_pair:
    push   rbp
    mov    rbp, rsp
    sub    rsp, 0x70
    mov    rax, rdi
    mov    qword ptr [rbp - 0x8], rsi
    mov    qword ptr [rbp - 0x10], rdx
    mov    rcx, qword ptr [rbp - 0x8]
    mov    rdx, qword ptr [rbp - 0x10]
    mov    qword ptr [rbp - 0x58], rdi
    mov    rdi, rdx
    mov    qword ptr [rbp - 0x60], rax
    mov    qword ptr [rbp - 0x68], rcx
    call   0x10f90aaf0               ; string_hash
    mov    qword ptr [rbp - 0x18], rax
    mov    rax, qword ptr [rbp - 0x18]
    mov    rcx, qword ptr [rbp - 0x68]
    movsxd rdx, dword ptr [rcx]
    xor    r8d, r8d
    mov    qword ptr [rbp - 0x70], rdx
    mov    edx, r8d
    mov    rsi, qword ptr [rbp - 0x70]
    div    rsi
    mov    qword ptr [rbp - 0x20], rdx
    mov    rdx, qword ptr [rcx + 0x10]
    imul   rdi, qword ptr [rbp - 0x20], 0x30
    add    rdx, rdi
    mov    qword ptr [rbp - 0x28], rdx
    mov    rdx, qword ptr [rbp - 0x28]
    mov    rdx, qword ptr [rdx]
    mov    qword ptr [rbp - 0x30], rdx
    mov    rdx, qword ptr [rbp - 0x28]
    mov    r8d, dword ptr [rdx + 0x18]
    mov    dword ptr [rbp - 0x34], r8d
    mov    rdx, qword ptr [rbp - 0x28]
    mov    r8d, dword ptr [rdx + 0x1c]
    mov    dword ptr [rbp - 0x38], r8d
    mov    rdx, qword ptr [rbp - 0x28]
    mov    rdx, qword ptr [rdx + 0x10]
    mov    qword ptr [rbp - 0x40], rdx
    mov    r8d, dword ptr [rbp - 0x34]
    mov    dword ptr [rbp - 0x44], r8d
    mov    eax, dword ptr [rbp - 0x44]
    cmp    eax, dword ptr [rbp - 0x38]
    je     0x10f90c708               ; <+280> at hash-table.cpp
    mov    rax, qword ptr [rbp - 0x30]
    movsxd rcx, dword ptr [rbp - 0x44]
    imul   rcx, rcx, 0x18
    add    rax, rcx
    mov    qword ptr [rbp - 0x50], rax
    mov    rax, qword ptr [rbp - 0x50]
    mov    rdi, qword ptr [rax]
    mov    rsi, qword ptr [rbp - 0x10]
    mov    edx, 0x40
    call   0x10f90cdb8               ; symbol stub for: memcmp
    cmp    eax, 0x0
    jne    0x10f90c6f0               ; <+256> at hash-table.cpp:177:5
    mov    rax, qword ptr [rbp - 0x50]
    mov    rcx, qword ptr [rax]
    mov    rdx, qword ptr [rbp - 0x58]
    mov    qword ptr [rdx], rcx
    mov    rcx, qword ptr [rax + 0x8]
    mov    qword ptr [rdx + 0x8], rcx
    mov    rax, qword ptr [rax + 0x10]
    mov    qword ptr [rdx + 0x10], rax
    jmp    0x10f90c723               ; <+307> at hash-table.cpp
    jmp    0x10f90c6f5               ; <+261> at hash-table.cpp:172:53
    mov    rax, qword ptr [rbp - 0x40]
    movsxd rcx, dword ptr [rbp - 0x44]
    mov    edx, dword ptr [rax + 4*rcx]
    mov    dword ptr [rbp - 0x44], edx
    jmp    0x10f90c68d               ; <+157> at hash-table.cpp:172:28
    mov    rax, qword ptr [rbp - 0x58]
    mov    qword ptr [rax], 0x0
    mov    qword ptr [rax + 0x8], 0x0
    mov    qword ptr [rax + 0x10], 0x0
    mov    rax, qword ptr [rbp - 0x60]
```

The whole function, unfortunately, does not fit into the screenshot. Below are a few lines responsible for exiting the function: stack shift, etc.
Almost every line in this function is a memory reference. What could be worse for a hash table?

A decision was made to rewrite this function entirely in assembly language.

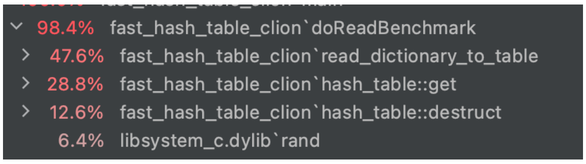
```
        ; Saving registers
        push    rbp
        push    r15
        push    r14
        push    r13
        push    r12
        ; Preparing string_hash call
        ; Optimised excessive memory accesses here
        ; Local variables are now stored in registers
        mov     r12, rdx
        mov     r15, rsi
        mov     qword [rsp], rdi
        mov     rdi, rdx
        ; Calculating string hash
        call    _string_hash
        ; Dividing string hash by capacity
        movsxd  rcx, dword [r15]      ; rcx = this->capacity
        xor     edx, edx
        div     rcx                       ; edx = chain_index
        mov     rcx, qword [r15 + 16] ; rcx = this->lists

        ; chain_index *= sizeof(s_list_entry)
        lea     rdx, [rdx + 2*rdx]        ; chain_index *= 3
        shl     rdx, 4                    ; chain_index *= 8
        mov     ebp, dword [rcx + rdx + 28] ; ebp = lists[chain_index]->head
        mov     eax, dword [rcx + rdx + 24] ; eax = lists[chain_index]->tail
        ; if(eax == ebp), don't jump in the cycle
        cmp     eax, ebp
        je      .return_zero

        mov     r13, qword [rcx + rdx]
        mov     r15, qword [rcx + rdx + 16]
.cycle:
        ; comparing keys
        ; rbx = walker
        movsxd  rbx, eax
        ; offset = walker * 8 * 3 = r14 * 8
        lea     r14, [rbx + 2*rbx]
        ; rdi = lists->entries[walker]
        mov     rdi, qword [r13 + 8*r14]
        mov     edx, 64
        mov     rsi, r12
        call    _memcmp
        test    eax, eax
        je      .return_found_entry
        mov     eax, dword [r15 + 4*rbx]
        cmp     eax, ebp
        jne     .cycle
.return_zero:
        ; xmm0 = 0
        xorps   xmm0, xmm0
        ; rax = returned structure address
        mov     rax, qword [rsp]
        ; writing zeroes in result
        movups  [rax], xmm0
        mov     qword [rax + 16], 0
        jmp     .return
.return_found_entry:
        ; rdx = offset of pair
        lea     rdx, [8*r14]
        add     rdx, r13
        ; reading .hash field
        mov     rcx, qword [rdx + 16]
        ; rax = returned stucture address
        mov     rax, qword [rsp]
        ; writing .hash field
        mov     qword [rax + 16], rcx
        movups  xmm0, [rdx]
        ; writing .key and .value field
        movups  [rax], xmm0
.return:
        ; restoring stack
        pop     r12
        pop     r13
        pop     r14
        pop     r15
        pop     rbp
        ret
```

89.3% fast_hash_table_clion`doReadBenchmark
56.8% fast_hash_table_clion`read_dictionary_to_table
16.3% fast_hash_table_clion`hash_table::destruct
8.7% libsystem_c.dylib`rand
2.5% fast_hash_table_clion`hash_table::get_pair

99.9% fast_hash_table_clion`main
98.3% fast_hash_table_clion`doReadBenchmark
57.6% fast_hash_table_clion`read_dictionary_to_table
15.3% fast_hash_table_clion`hash_table::destruct
12.0% fast_hash_table_clion`hash_table::get
9.8% libsystem_c.dylib`rand

If we compare the running time of rand with the running time of the old and new insertion functions (they are called proportionally many times), we get the ratio (8.7 / 2.5) / (9.8 / 12) = 4.2.

Let's carry out a control experiment by rolling back all optimizations and compare the running times.
(8.7 / 2.5) / (28.8 / 6.4) = 15.6 times.



Chapter 5: Conclusion

The total number of lines written in assembly language is 58 (50 lines are taken by the get_pair instruction rewritten in intel syntax and another 8 lines of assembly language insertion for the hash function in AT&T syntax). The program was accelerated by a factor of 15.6. Thus, the Dedinsky Number for this solution is 15.6/58*1000 = 268.