Introspecting hashtable speed up possibilities

Concerning assembly implementations and iteratively-comparative approach

Aleksandr Dremov
Dedinsky Ilya Rudolfovich
31 March 2021

1. Description

Development center Dvoika™ accepted severe challenge of scrutinising and speeding up sophisticatedly implemented SND[1] solid ind its raw form conventional hashmap[2] structure. Our center approached the problem with iterate-and-compare conception. Specifically, on the first steps, compare with built-in data structure. Then, skyrocket performance from version to the version. We considered utilising inline assembly and full-in-parts™ code replacement at least in two structural instances™©[3].

2. Interface

```
- void set (string key, T value)
  - Set key->value
- T* get (string key)
  - Get by key
```

3. Development

3.1. Testing method selection

To conduct testing experiences, we choose Lev Nikolaevich Tolstoy's "War and Peace." Moreover, we copied it **5 times™** as to receive state-of-art performance we need more art.

3.2. First versions

Taking into account desires about functionality details (and ignoring them), the first straightforward version was developed. The ways to make it straighter and forwarder appeared immediately.

---

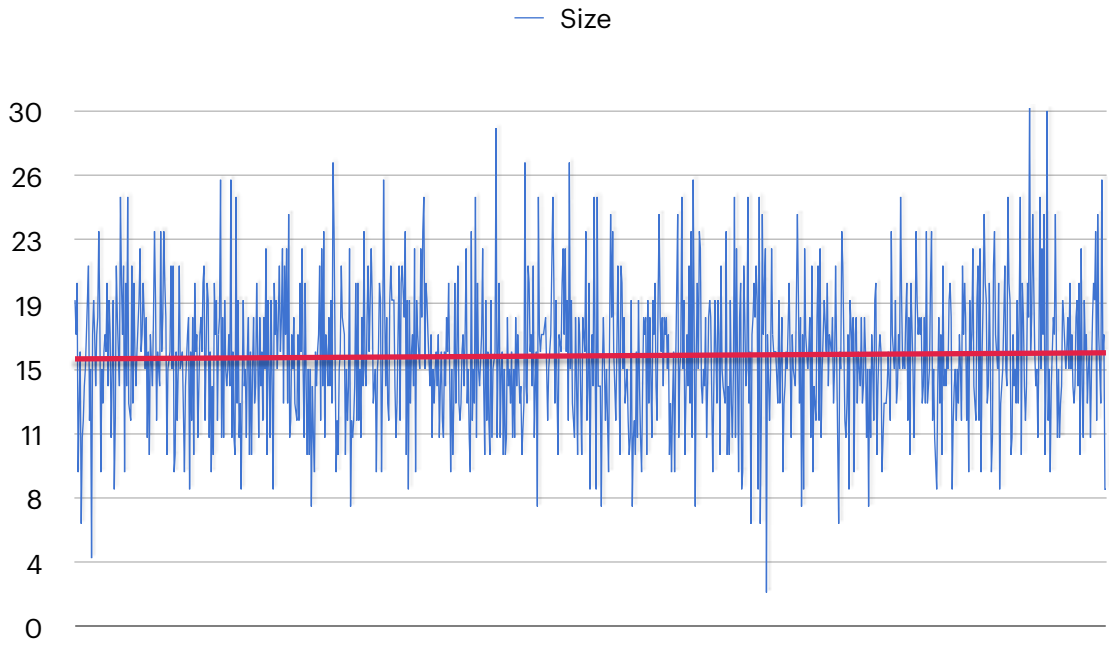
[1] AND but the feeling of personal supremacy did not allow the author to correct the mistake.

[2] data structure that allows to do most (least) of the operations in O(1)

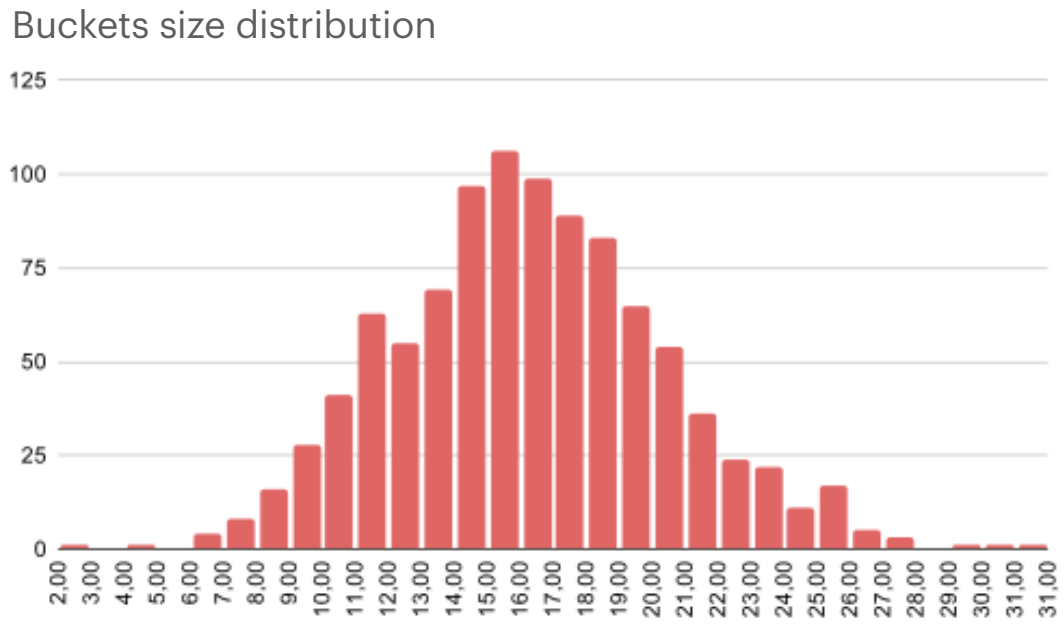
[3] according to ECOLE CoronaVirus con-cilium

3.2.1. Hash function choice

Taking into account assembly optimisation, our team choose CRC32 and implemented it without assembly. Unfortunately, during testing we were not able to see the program finish due to the run time limiting to infinity due to CRC32 bad implementation. Therefore, FNV64 was selected at this point.



FNV64: bucket sizes, rehash turned off



We were concerned of such selection as due to OAI[4], we conducted SKPH with conventional KDA. The results were astonishing and YGII. It was discovered that, SKPH works longer (than FNVCRC (not to be confused with CRCFNV)).

As you can see, the distribution of FNV is **normal[5]** in all interpretations.

3.2.2. Storage structure

It was important to select type of stored in the list values. It could have been either pointer to structure or structure itself. In the first case, there was one extra pointer reference. In the second case, bigger copy amounts during rehash.

It was experimentally proven that pointer reference is much longer than extra copying. The experiment was conducted with the budget of 9 billion rubles which is pretty much humble sum for such task.

```
- 100%
- My hash table: elapsed: 3.096260s
```

Extra pointer reference working time

```
- 100%
- My hash table: elapsed: 2.683230s
```

Bigger copy size

[4] Open American Insults

[5] by Gauss

3.3. First versions results

Even at this point, standard unordered_map was deleted on -O3 and -O0 optimisation levels.

```
- 100%
- My hash table: elapsed: 3.378775
- 100%
- Std unordered map: elapsed: 4.966769
```

Optimisation -O0

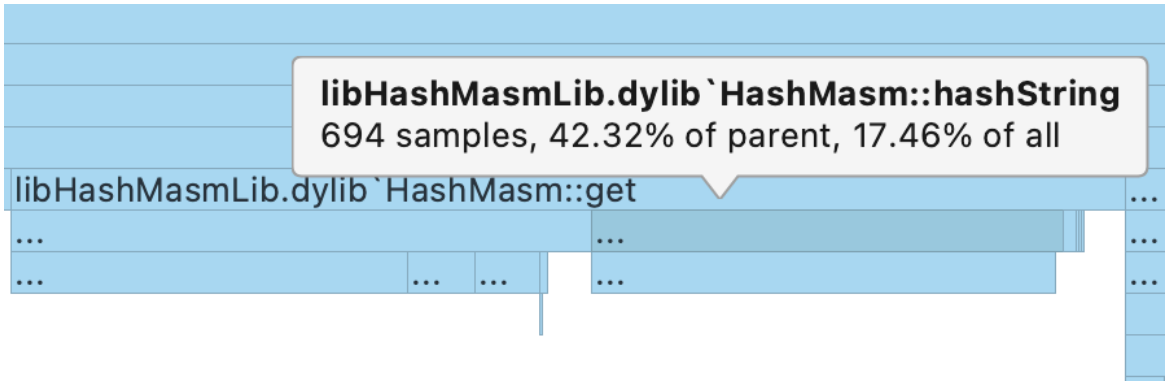
```
- 100%
- My hash table: elapsed: 2.774224
- 100%
- Std unordered map: elapsed: 3.452640
```

Optimisation -O3

4. Optimisation

4.1. First step

Using profiler[6] it was discovered that around 17% of time is taken by hash function. The code generated by -O0 и -O3 is provided below.



Hashing time

It's noticeable that -O0 does no memory access optimisations. In -O3 number of instructions has decreased, but it still can be improved.

```
_ZN3FNV5fnv64EPKcm:   # @_ZN3FNV5fnv64EPKcm
1.push   rbp
2.mov   rbp, rsp
3.....
14.movabs   rax, 1099511628211
15.imul   rax, qword ptr [rbp - 16]
16.mov   qword ptr [rbp - 16], rax
17.mov   rax, qword ptr [rbp - 8]
18.mov   rcx, qword ptr [rbp - 32]
22.....
26..LBB0_4:
27.mov   rax, qword ptr [rbp - 16]
28.pop   rbp
29.ret
```

Hashing with -O0

[6] dtrace

```
_ZN3FNV5fnv64EPKcm:  # @_ZN3FNV5fnv64EPKcm
1.mov  cl, byte ptr [rdi]
2.test  cl, cl
3.je  .LBB0_1
4.add  rdi, 1
5.movabs  rdx, 1099511628211
6..LBB0_4: # =>This Inner Loop Header: Depth=1
7.imul  rsi, rdx
8.movsx  rax, cl
9.xor  rax, rsi
10.movzx  ecx, byte ptr [rdi]
11.add  rdi, 1
12.mov  rsi, rax
13.test  cl, cl
14.jne  .LBB0_4
15.ret
16..LBB0_1:
17.mov  rax, rsi
18.ret
```

Hashing with -O3

```
size_t fnv64Asm(const char *p, size_t hash = ...);
```

My solution:[7]

```
_ZN3FNV5fnv64EPKcm:
1.  mov rax, rsi
2.  mov rcx, FNV64Const
3..loop:
4.  movzx  rdx, byte [rdi]
5.  inc    rdi
6.  imul   rax, rcx
7.  xor    rax, rdx
8.  test   dl, dl
9.  jne .loop
10.ret
```

FNV63 using -OS (-OSasha)

[7] experimental optimisation methods

In System V calling convention the first parameter is placed in rdi. Without any adjystments, I use it for memory access. Return value must be placed in rax, so I calculate the answer in rax immediately.

Perfomance evaluation:

-O0	**-OSasha**	
44,08	15,96	
44,38	17,76	
41,50	17,79	
43,88	13,26	
44,28	17,08	
44,16	16,69	
43,7	**16,4**	**27,3**

-O3	**-OSasha**	
6,91	6,97	
6,61	6,68	
6,97	6,88	
6,68	6,61	
6,72	7,08	
6,62	6,7	
6,8	**6,8**	**-0,1**

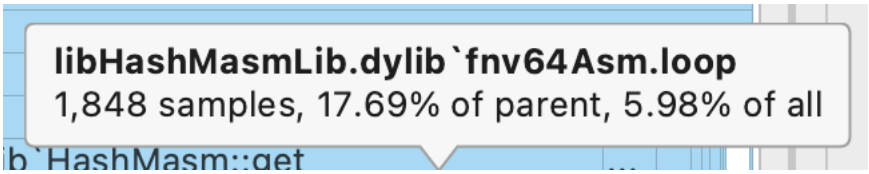
The speed up is estimated to be 27% compared to -O0. Comparison with - O3 is impossible due to the fact that difference is neglected due to the error. But it can be said that performance is similar.

Dedinsky number on -O0 relative to hash function speed up:

$$\frac{43.7}{5 \cdot 16} \cdot 1000 = \textbf{455 (speedup/lines)}$$

That's why I am this high-paid.[8]

[8] high ~ average



Asm hashing

4.2. In details

Second most time-taking procedure was the search of value by key. To alleviate this issue, inline asm was utilised.

```
size_t begin() const
```

```
volatile size_t retVal = 0;
asm volatile(
    "mov (%1), %0\n"
    "mov (%0), %0\n"
    : "=r"(retVal)
    :"r"(this));
return retVal;
```

Begin speed up

```
asm goto ("test %0, %0\n"
          "je %l1"::"r"(i):"cc": findCellLoopEnd);
{
    storage[hashed].get(i, &tmpNode);
    if (strcmp(tmpNode->key, key) == 0) {
        node = tmpNode;
        asm goto("jmp %l0"::: findCellLoopEnd);
    }
    storage[hashed].nextIterator((size_t*)&i);
}
asm goto("jmp %l0"::: findCellLoop);
findCellLoopEnd:
```

FindCell loop speed up

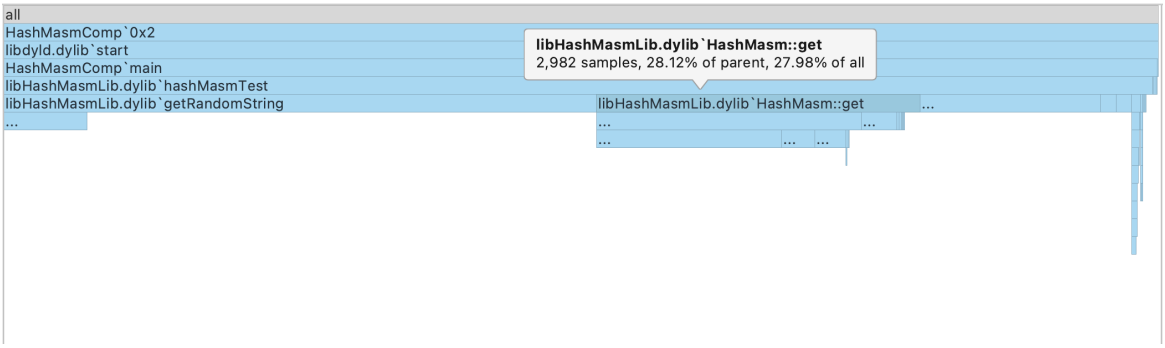
All in all, about improvement 1.3% was reached.

4.3. SIMD approach

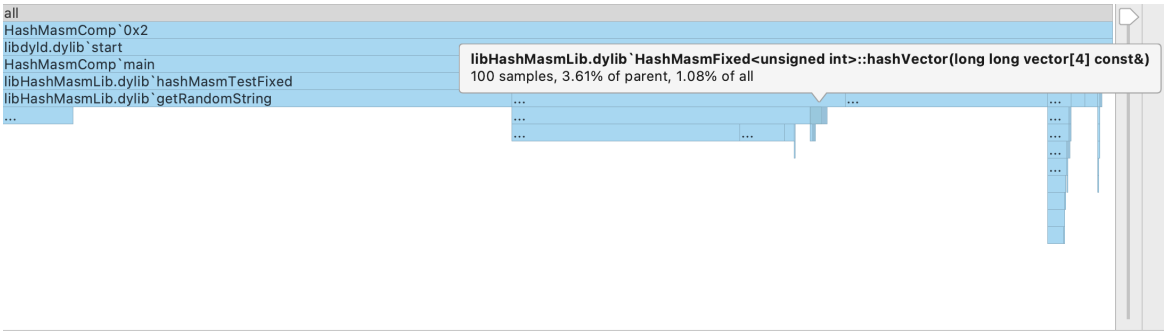
The new idea of using SIMD in connection with fixed length key was proposed. The limitation is 32 bytes maximum key. But the speedup is vivid.

```
1._hash32AsmFLen:
2.mov rax, rdx
3.crc32 rax, qword [rdi]
4.crc32 rax, qword [rdi + 8]
5.crc32 rax, qword [rdi + 8 * 2]
6.crc32 rax, qword [rdi + 8 * 3]
7.ret
```

CRC32 for -OS (-OSasha)



Hash function time with no fixed key and no asm optimisation



Hash function time with fixed key and asm optimisation

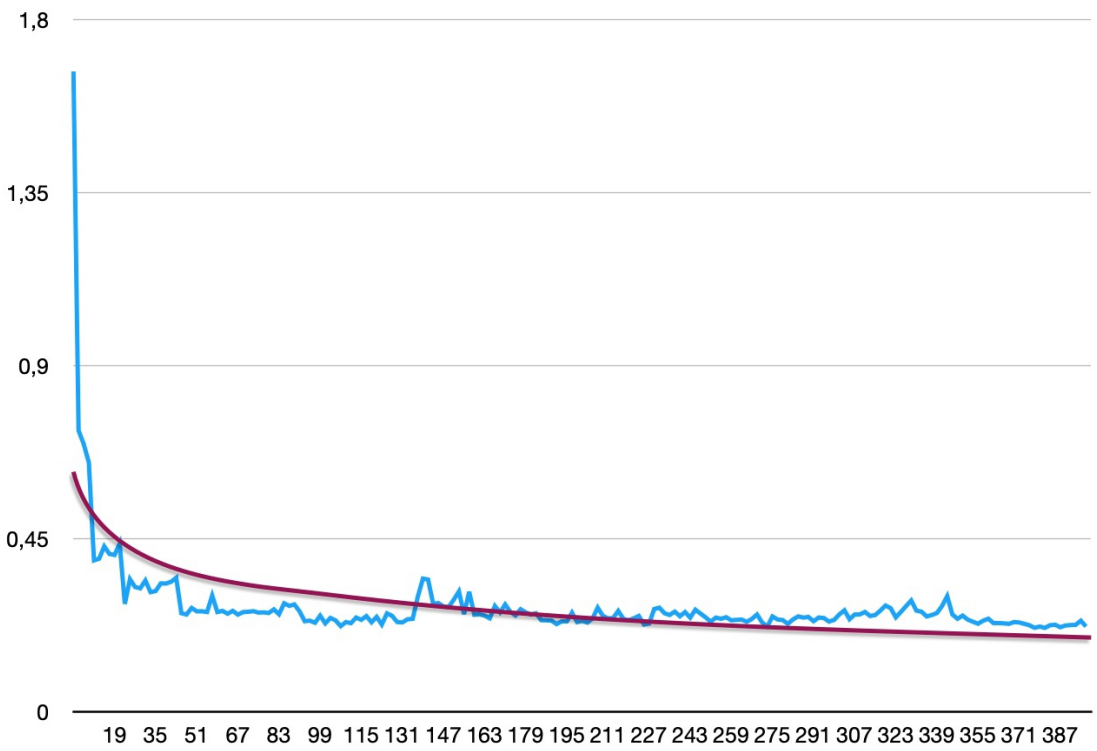
5. Results

Finally, the speedup of **28,6%** was obtained without fixed key and **32.7%** with fixed key solution. It resulted in **3.1 times** faster than *unordered_map* in -O0 and **2.35 times** faster with -O3

Final Dedinsky number: $\frac{3.1}{8} \cdot 1000 = 387$ **(speedup/lines)**

6. Advices

As hyperparameter tuning is crucial, we conducted research on selecting load rate.



It's vivid that 75% seems just about right. In case of disagreement look at the figure above.