Hash table optimization

Kolesnikova Xenia

April 2021

1 Introduction

In field of working with data using hash tables is widely spread. A hash table is a data structure that provides fast, instant access to elements, their search and insertion in constant O(1) time. A conflict occurs when two keys map to the same index. This situation is called a collision. There are several ways of handling it. To combat collisions, it was decided to use a hash table with chains. It will be an array of linked lists. All keys mapping to the same index will be stored as linked list nodes at that index.

But this data structure is not always fast. This is due to the fact that the compiler does not always know how to optimize the cunning algorithms written by the programmer. The programmer, on the other hand, has access to this knowledge. This is where the idea to speed up the hash table comes in.

The author will speed up this data structure by rewriting the two functions in the assembler in two different ways.

2 Materials

The input data is supposed to be an English-Russian dictionary with more than 6500 key-value pairs. Also, in this work, a fast and secure doubly linked list is used, which was previously written by the author. To measure the duration of the callgrind profiler was used the kcachegrind application. The work was done on Ubuntu 20.04.1 with an Intel Core i5 processor.

Below are the formulas for calculating the acceleration factors:

$$coeff_{boost} = \frac{time_1}{time_2} \tag{1}$$

where $time_1$ and $time_2$ - results before and and after accelerations.

To calculate the total acceleration by the formula, we will use the following formula:

$$coeff_{total\ accelerate} = \frac{coeff_{boost}}{lines_{assembler}} * 1000 \quad (2)$$

where $lines_{assembler}$ - the number of lines rewritten to assembler.

3 Work progress

3.1 Analysis of the program runtime

Let's analyze how the program functions work using callgrind.

Incl.	Self	Called	Function	Location
26.18	26.18	3 241 128	random_r	libc-2.31.so: random_r.c
19.71	19.71	106 507	get_hash_word(char co...	hashtable: hash_table.cpp, smmintrin.h
45.07	18.89	3 241 128	random	libc-2.31.so: random.c, lowlevellock.h
85.58	16.19	1	testing_hash_table(Has...	hashtable: hash_table.cpp
9.58	9.58	8 192	__memset_avx2_unalig...	libc-2.31.so: memset-vec-unaligned-erms.S
49.18	4.11	3 241 128	rand	libc-2.31.so: rand.c
3.29	1.37	1	parsing_buffer(File*, Ha...	hashtable: hash_table.cpp
20.21	1.26	100 000	hash_table_is_contain_...	hashtable: hash_table.cpp
0.65	0.58	6 507	list_insert_before(List*,...	hashtable: list.cpp, string_fortified.h
0.45	0.45	78 150	__strcmp_avx2	libc-2.31.so: strcmp-avx2.S
0.40	0.40	8 192	list_initializate(List*, un...	hashtable: list.cpp, string_fortified.h
0.30	0.29	8 199	_int_malloc	libc-2.31.so: malloc.c
0.42	0.26	8 225	_int_free	libc-2.31.so: malloc.c
10.04	0.16	8 194	calloc	libc-2.31.so: malloc.c
10.54	0.09	8 192	list_construct(List*, uns...	hashtable: list.cpp
1.92	0.07	6 513	hash_table_insert_elem...	hashtable: hash_table.cpp
0.12	0.07	7 774	systrim.isra.0.constprop.0	libc-2.31.so: malloc.c
0.05	0.05	13 015	__memcpy_avx_unalign...	libc-2.31.so: memmove-vec-unaligned-erms.S
0.46	0.04	8 200	free	libc-2.31.so: malloc.c
0.04	0.04	8 191	unlink_chunk.isra.0	libc-2.31.so: malloc.c
0.04	0.04	7 864	sbrk	libc-2.31.so: sbrk.c
0.03	0.03	21 206	list_verifier(List*, call_o...	hashtable: list.cpp
0.49	0.02	8 192	list_destruct(List*)	hashtable: list.cpp
10.56	0.02	1	hash_table_construct(H...	hashtable: hash_table.cpp
0.02	0.02	1	_dl_addr	libc-2.31.so: dl-addr.c
0.50	0.02	1	hash_table_destruct(Ha...	hashtable: hash_table.cpp
0.05	0.02	7 864	__default_morecore	libc-2.31.so: morecore.c
0.01	0.01	1	__GI___tunables_init	ld-2.31.so: dl-tunables.c, dl-tunables.h

Figure 1: Functions running time and the number of their calls.

Profiling by the category "self" showed that the longest functions (that be written not by developers C++) is:

- get_hash_word: the function that calculate hash of the string. It just needs to be accelerated, because it is called very often

- testing_hash_table: the function that creates 100 000 random strings and find its in hash table. Obviously, it is useless to optimize.

- parsing_buffer: the function is called at the stage of text processing. Optimization is not required as it is only done once.

- hash_table_is_contain_element: the function checks if the hash table contains an item or not. It is also highly desirable to speed it up due to frequent calls.

Great, we found two features that we want to speed up. Let's get down to deal!

3.2 Optimization №1

First, let's optimize the function that calculates the hash of words. Crc32 was chosen as the hashing algorithm, the polynomial 0x82f63b78 was used. The optimization took 11 lines in the assembler, in which the hash is calculated.

```
section .text

global asm_get_hash

asm_get_hash:

    crc32 edi, esi
    not edi

    mov rcx, rdx

    mov eax, edi
    sar edi, cl
    sal edi, cl

    sub eax, edi

    ret
```

Figure 2: First assembler optimization.

3.3 First comparison

You can see the optimization results (performance comparison) in Figure 3 and Table 1.

With the optimization flag -O1, you can notice the following: according to formula (1), the program has accelerated 1.26 times, and Ded's number (formula 2) is approximately equal to 114. O2: 1.25 and 113. O3: 1.31 and 119. Author believes that these are good indicators.

Let's try to speed up the following function.

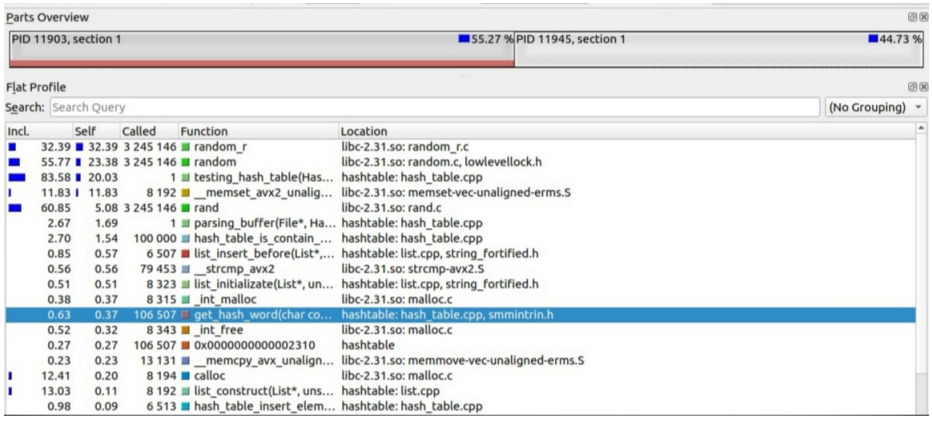

Figure 3: Comparison of times before and after the first acceleration (-O1).

	-O1	-O2	-O3
$coeff_{boost}$	1.25	1.26	1.31
$coeff_{total\ accelerate}$	114	114	119
$Runtime\ without\ optimize\ (ms)$	41	40	37
$Runtime\ with\ first\ optimizes\ (ms)$	39	30	30

Table 1: Comparison of times before and after the first acceleration.

3.4 Optimization №2

As you can see from the analysis above, the next longest execution time function is hash_table_is_contain_element. This function contains a call to a hash function and a loop that loops through the list and looks for a search word. Note that the loop contains a call to the strcmp function, which checks the strings for equivalence. Let's be honest: this function is already sped up (it takes over 800 lines in assembler!). But if the word length is fixed and equal to a power of two, we can use the SSE registers. So it was decided to rewrite the entire search loop (including the strcmp that is used in the search loop) by assembler insertion using Intel syntax.

In Figure 4, you can see an assembler insertion that replaces the search loop and strcmp. The number of lines written in assembler is 24.

```
asm ("mov rcx, 0                                    \n " // now_pos_into_word
     "mov rsi, %[size_list]                         \n " // size_list
     "mov rdi, %[ptr_to_list_value]                 \n " // ptr_to_list_value

  "finding_word:                                    \n "
     "cmp rcx, rsi                                  \n "
     "je end_of_compare                             \n "

     "vmovdqu ymm0, [%[string_to_cmp]]              \n " // the value we are looking for
     "vmovdqu ymm1, [rdi + rcx * %[node_size]]      \n "
     "inc rcx                                       \n "

     "vpcmpeqq ymm2, ymm0, ymm1                     \n " // 0, if equal
     "vpmovmskb %[is_equals_first_half], ymm2       \n "

     "vmovdqu ymm0, [%[string_to_cmp]         + 32] \n "
     "vmovdqu ymm1, [rdi + rcx * %[node_size] + 32] \n "
     "vpcmpeqq ymm2, ymm0, ymm1                     \n " // 0, if equal
     "vpmovmskb %[is_equals_second_half], ymm2      \n "

     "not %[is_equals_first_half]                   \n "
     "not %[is_equals_second_half]                  \n "

     "cmp %[is_equals_first_half],  0               \n"
     "jne finding_word                              \n "

     "cmp %[is_equals_second_half], 0               \n "
     "jne finding_word                              \n "

     "mov %[is_find], 1                             \n "
     "jmp end_of_compare                            \n "


  "end_of_compare:                                  \n "
```

Figure 4: Second assembler optimization.

3.5 Second comparison

Now let's compare programs with one and two optimizations.

	-O1	-O2	-O3
$coeff_{boost}$	1.00	1.00	1.00
$coeff_{total\ accelerate}$	42	41	42
$Runtime\ with\ first\ optimize\ (ms)$	35	30	29
$Runtime\ with\ both\ optimizes\ (ms)$	29	27	26

Table 2: Comparison of times before and after the first and both accelerations.

3.6 Final comparison

Now let's compare the initial and final (with two accelerations) versions.

	-O1	-O2	-O3
$coeff_{boost}$	1.24	1.25	1.31
$coeff_{total\ accelerate}$	35	36	37
Runtime without optimize (ms)	41	40	37
Runtime with both optimizes (ms)	29	27	26

Table 3: Comparison of times before and after accelerations.

4 Results

We got the following result: with the O3 flag the program accelerated by 23.6%, with the O2 flag - by 20.3%, with the O1 flag - by 19.9%.

Results of this investigation can be divided into three parts:

1. Boosting with assembly lines.
2. Boosting calculating the hash using intrinsics.
3. Boosting with using different flags.

That shows different ways of decreasing the time program works.

5 Literature and links

- Author's github page: https://github.com/owl1234/MIPT_projects_2_sem/tree/master/Hash
- Intel website with intrinsics: https://software.intel.com/sites/landingpage/IntrinsicsGuide
- Documentation for using extended assembly: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html