Московский авиационный институт
(национальный исследовательский университет)

Институт № 8 «Информационные технологии и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: Л. В. Короткевич
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19
Дата:
Оценка:
Подпись:

Москва 2020

Лабораторная работа №3

Задача: Для реализации словаря из предыдущей лабораторной работы, необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить.

1. Введение

Анализ производительности программ, выявление слабых мест в коде, поиск ошибок — неотъемлемая часть процесса разработки ПО. Благо, в наши дни наличествует большой набор инструментов, упрощающих эти задачи. На примере программы из предыдущей лабораторной работы (реализация словаря с использованием PATRICIA Trie) я проведу детальный анализ производительности моей программы, потребления памяти.

2. Анализ времени исполнения

2.1. Callgrind

Пожалуй, одна из самых удобных в плане пользования программ для профилирования кода — *callgrind* (инструмент, находящийся в составе valgrind). Вкупе с утилитой *kcachegrind* он позволяет наблюдать «затратность» тех или иных функций, количество вызовов и т.д. Всего более мне нравится то, что здесь и графический интерфейс, и наглядный граф вызовов функций — все в одном месте. Можно посмотреть, сколько вызывалась функция X в функции Y. А в функции X — сколько раз вызывалась любая другая функция и т.д.

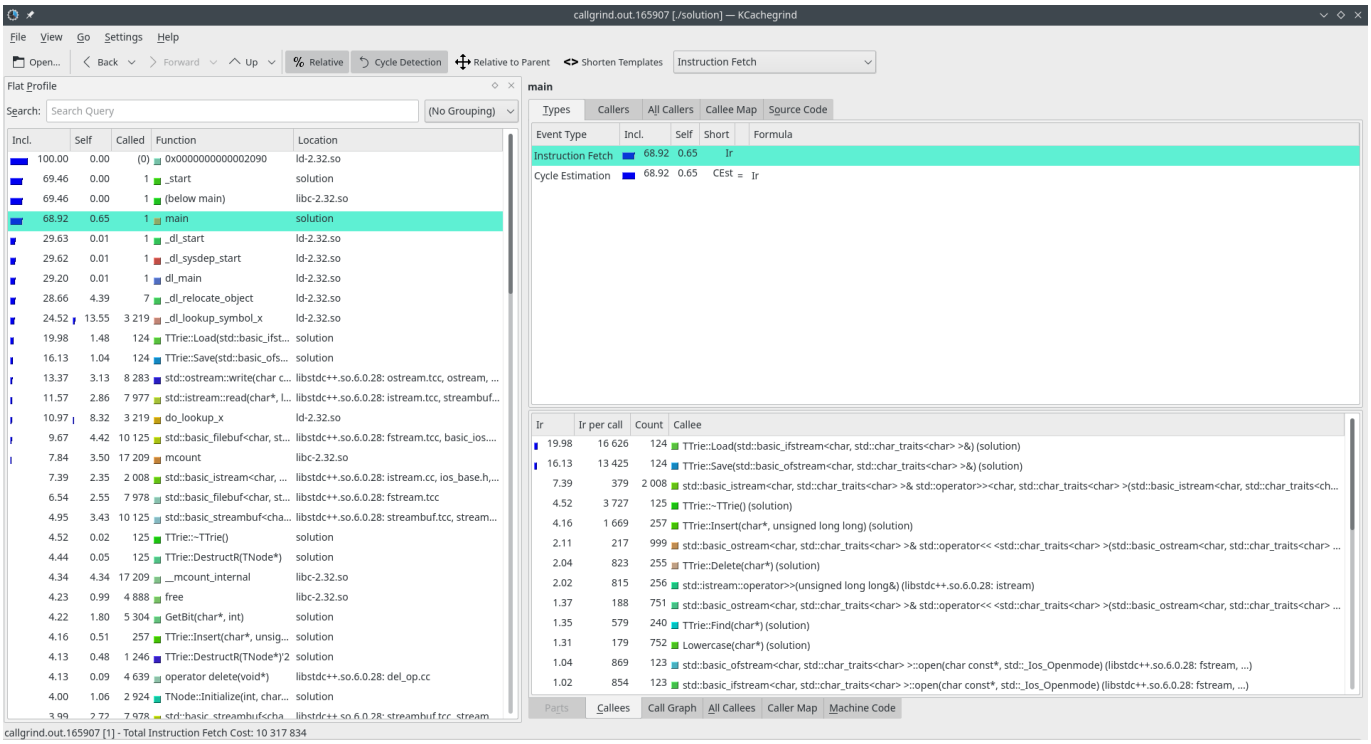
Заранее скомпилировав программу (с обычными ключами; но чтобы наблюдать сурс-код в kcachegrind, ключ -g3 обязателен), запустим профайлер следующим образом:

[leo@pc solution]$ valgrind --tool=callgrind ./solution <test1k >trash

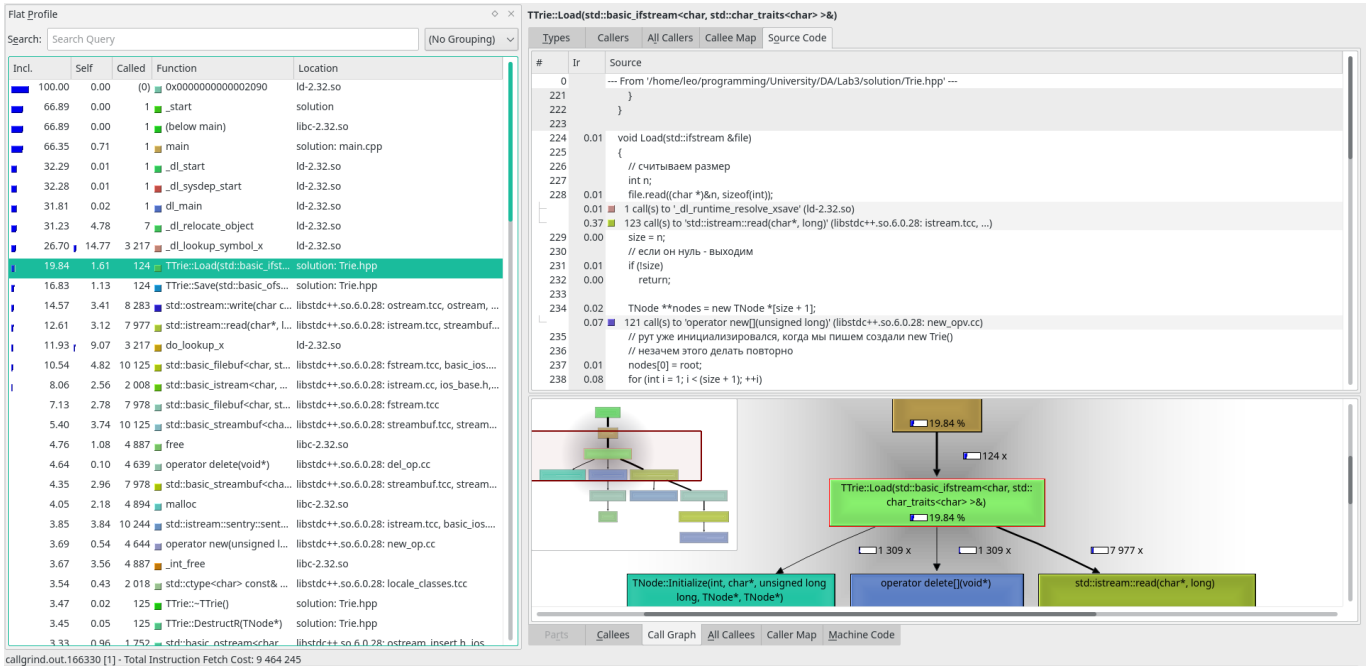
Сгенерировался файлик callgrind.out.XXX – откроем его:

[leo@pc solution]$ kcachegrind callgrind.out.165907

Наблюдаем богатый интерфейс:



Слева видим таблицу функций, отсортированных по степени «прожорливости». Нас интересуют «самописные» функции для работы со словарем — их есть смысл оптимизировать. Выбрав одну из них, например, Load, справа наблюдаем подробную информацию о ней:



Из самых полезных полей здесь, пожалуй, сурс-код и граф вызовов – я могу смотреть, сколько раз вызвалась та или иная функция на определенной строчке; также вижу, кто вызвал Load, сколько раз. Аналогично Load: кого вызывала, сколько раз.

2.2. gcov, lcov

gcov - утилита, которая вкупе с lcov позволяет сгенерировать наглядный отчет о «покрытости» программы: какие стрчоки кода сколько раз исполнялись.

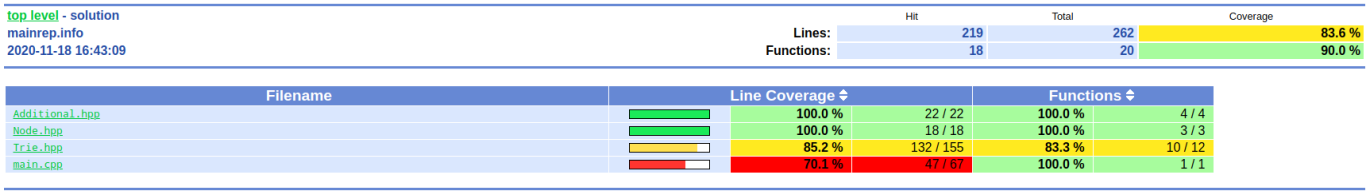
[leo@pc solution]$ g++ --coverage main.cpp -o main

[leo@pc solution]$ ./main <test1k >trash

[leo@pc solution]$ lcov -t "mainrep" -o mainrep.info -c -d .

[leo@pc solution]$ genhtml -o report mainrep.info

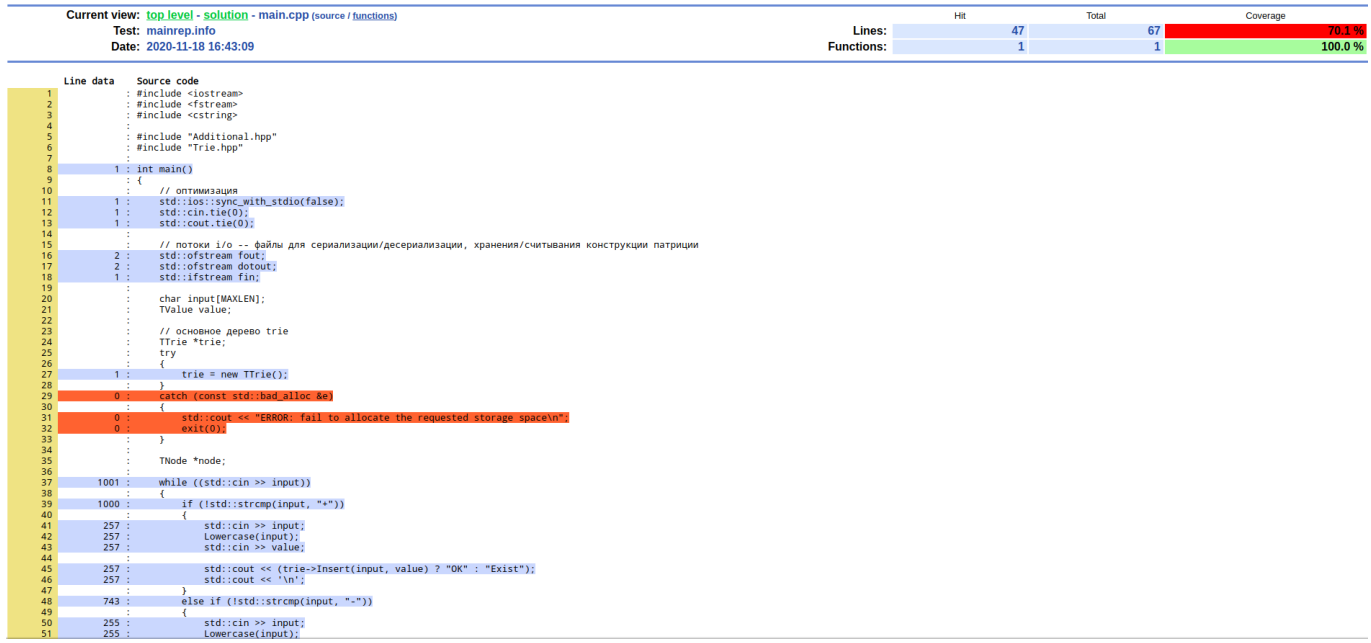
Теперь откроем ./report/index.html:



top level - solution		Hit	Total	Coverage
mainrep.info	Lines:	219	262	83.6 %
2020-11-18 16:43:09	Functions:	18	20	90.0 %

Filename	Line Coverage		Functions	
Additional.hpp	100.0 %	22 / 22	100.0 %	4 / 4
Node.hpp	100.0 %	18 / 18	100.0 %	3 / 3
Trie.hpp	85.2 %	132 / 155	83.3 %	10 / 12
main.cpp	70.1 %	47 / 67	100.0 %	1 / 1

Радует: мой код покрыт на все сто, за исключением участков кода, содержащих catch.



Current view: top level - solution - main.cpp (source / functions)		Hit	Total	Coverage
Test: mainrep.info	Lines:	47	67	70.1 %
Date: 2020-11-18 16:43:09	Functions:	1	1	100.0 %

```
Line data    Source code
   1          : #include <iostream>
   2          : #include <fstream>
   3          : #include <cstring>
   4          :
   5          : #include "Additional.hpp"
   6          : #include "Trie.hpp"
   7          :
   8        1 : int main()
   9          : {
  10          :     // оптимизация
  11        1 :     std::ios::sync_with_stdio(false);
  12        1 :     std::cin.tie(0);
  13        1 :     std::cout.tie(0);
  14          :
  15          :     // потоки i/o -- файлы для сериализации/десериализации, хранения/считывания конструкции патриции
  16        2 :     std::ofstream fout;
  17        2 :     std::ofstream dotout;
  18        1 :     std::ifstream fin;
  19          :
  20          :     char input[MAXLEN];
  21          :     TValue value;
  22          :
  23          :     // основное дерево trie
  24          :     TTrie *trie;
  25          :     try
  26          :     {
  27        1 :         trie = new TTrie();
  28          :     }
  29        0 :     catch (const std::bad_alloc &e)
  30          :     {
  31        0 :         std::cout << "ERROR: fail to allocate the requested storage space\n";
  32        0 :         exit(0);
  33          :     }
  34          :
  35          :     TNode *node;
  36          :
  37     1001 :     while ((std::cin >> input))
  38          :     {
  39     1000 :         if (!std::strcmp(input, "+"))
  40          :         {
  41      257 :             std::cin >> input;
  42      257 :             Lowercase(input);
  43      257 :             std::cin >> value;
  44          :
  45      257 :             std::cout << (trie->Insert(input, value) ? "OK" : "Exist");
  46      257 :             std::cout << '\n';
  47          :         }
  48      743 :         else if (!std::strcmp(input, "-"))
  49          :         {
  50      255 :             std::cin >> input;
  51      255 :             Lowercase(input);
```

3.3. Perf

perf — незаменимый инструмент для анализа производительности отдельных программ и даже всей системы в целом. Он обеспечивает богатую статистику выполнения программы в наглядном виде.

Основные команды perf:

- perf stat: получить количество событий
- perf record: запись событий для последующей отчетности
- perf report: считыание отчета, разбивка событий по процессам, функциям и т.д
- perf top: просмотр всех системных событий в реальном времени

Пример использования perf stat, record & report для тестируемой программы.

[leo@pc solution]$ perf stat ./solution <test1k

...

Performance counter stats for './solution':

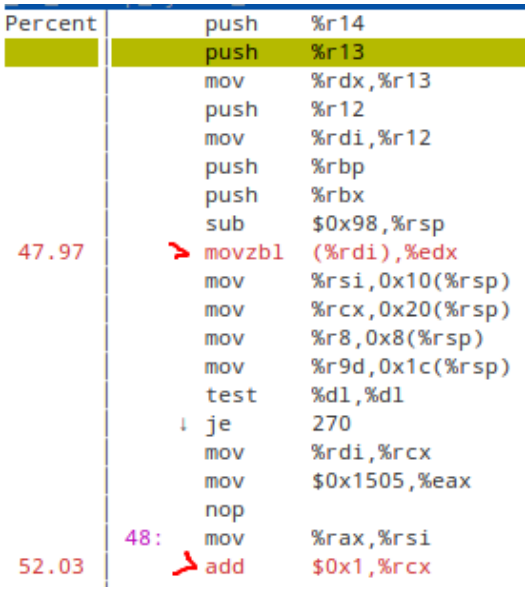
```
          7.55 msec task-clock:u              #    0.632 CPUs utilized
             0      context-switches:u        #    0.000 K/sec
             0      cpu-migrations:u          #    0.000 K/sec
           129      page-faults:u             #    0.017 M/sec
     5,840,562      cycles:u                  #    0.774 GHz
     8,632,037      instructions:u            #    1.48  insn per cycle
     1,600,579      branches:u                #  212.054 M/sec
        39,409      branch-misses:u           #    2.46% of all branches

      0.011944305 seconds time elapsed

      0.000000000 seconds user
      0.007805000 seconds sys
```

[leo@pc solution]$ perf record ./solution <test1k

[leo@pc solution]$ perf report

```
Samples: 36  of event 'cycles:u', Event count (approx.): 7882879
Overhead  Command   Shared Object        Symbol
  13.94%  solution  ld-2.32.so           [.] _dl_lookup_symbol_x
  13.13%  solution  ld-2.32.so           [.] do_lookup_x
  12.58%  solution  libc-2.32.so         [.] _int_malloc
  11.08%  solution  libstdc++.so.6.0.28  [.] std::istream::sentry::sentry
   9.90%  solution  libstdc++.so.6.0.28  [.] std::__ostream_insert<char, std::char_traits<char> >
   9.34%  solution  libc-2.32.so         [.] malloc
   9.22%  solution  solution             [.] TTrie::Load
   8.91%  solution  libc-2.32.so         [.] __GI__IO_un_link.part.0
   7.07%  solution  libstdc++.so.6.0.28  [.] std::basic_ifstream<char, std::char_traits<char> >::open
   2.15%  solution  ld-2.32.so           [.] __GI___tunables_init
   0.75%  solution  libstdc++.so.6.0.28  [.] std::__basic_file<char>::is_open
   0.73%  solution  libstdc++.so.6.0.28  [.] std::basic_filebuf<char, std::char_traits<char> >::open
   0.68%  solution  libc-2.32.so         [.] __strchr_avx2
   0.34%  solution  ld-2.32.so           [.] _dl_start
   0.13%  solution  libc-2.32.so         [.] __GI___libc_open
   0.05%  solution  [unknown]            [k] 0xffffffffa3c00fc7
   0.00%  solution  [unknown]            [k] 0xffffffffa3c00163
   0.00%  solution  [unknown]            [.] 0000000000000000
```

Посмотрим на самую трудозатратную часть программы.



Наблюдательно: movzbl – разыменовывание объекта, лежащего по опред. адресу. A add просто складывает два числа, лежащих по адресу &0x1, &rcx и кладет результат в &0x1.

Смею предположить, что обработка строк: поиск leftmost differ bit`a, получение определенного бита строки и т.д стали наиболее «укзим» местом моей программы.

3. Анализ потребления памяти

3.1. Valgrind

Классика жанра — Valgrind с его бесспорно превосходными инструментами. Чаще всего им пользуются для того, чтобы ловить утечки памяти; благодаря нему можно быть уверенным, что твоя программа надежна: верно обрабатывает запросы на выделение и удаление памяти. Благо перед тем, как заслать программу на чекер я много раз проверял ее волгриндом: утечек не было. Для наглядности я сломаю свою функцию KVCopy:

```
void KVCopy(TNode *src, TNode *dest)
{
	if (strlen(dest->key) < strlen(src->key))
	{
		// delete[] dest->key;
		dest->key = new char[strlen(src->key) + 1];
	}
	strcpy(dest->key, src->key);

	dest->value = src->value;
}
```

Теперь предыдущие данные, хранящиеся в dest→key, не будут уничтожены корректно, но перезапишутся новыми, случайными при выделении памяти с помощью new.

```
[leo@pc solution]$ valgrind --leak-check=full ./solution <test1k >trash
==178703== Memcheck, a memory error detector
==178703== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==178703== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==178703== Command: ./solution
==178703==
==178703== HEAP SUMMARY:
==178703==     in use at exit: 122,955 bytes in 36 blocks
==178703==   total heap usage: 4,894 allocs, 4,858 frees, 2,434,497 bytes allocated
==178703==
==178703== 3 bytes in 1 blocks are definitely lost in loss record 1 of 8
==178703==    at 0x4A37C17: operator new[](unsigned long) (vg_replace_malloc.c:431)
==178703==    by 0x10A82F: Initialize (Node.hpp:23)
==178703==    by 0x10A82F: Insert (Trie.hpp:82)
==178703==    by 0x10A82F: main (main.cpp:45)
==178703==
==178703== 72 bytes in 29 blocks are definitely lost in loss record 2 of 8
==178703==    at 0x4A37C17: operator new[](unsigned long) (vg_replace_malloc.c:431)
==178703==    by 0x10C20A: Initialize (Node.hpp:23)
==178703==    by 0x10C20A: TTrie::Load(std::basic_ifstream<char, std::char_traits<char> >&) (Trie.hpp:264)
==178703==    by 0x10AB42: main (main.cpp:86)
==178703==
==178703== LEAK SUMMARY:
==178703==    definitely lost: 75 bytes in 30 blocks
==178703==    indirectly lost: 0 bytes in 0 blocks
==178703==      possibly lost: 0 bytes in 0 blocks
```

==178703==   still reachable: 122,880 bytes in 6 blocks
==178703==   suppressed: 0 bytes in 0 blocks
==178703== Reachable blocks (those to which a pointer was found) are not shown.
==178703== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==178703==
==178703== For lists of detected and suppressed errors, rerun with: -s
==178703== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

Что и требовалось доказать. Исправим эту пагубную ошибку.

[leo@pc solution]$ valgrind --leak-check=full ./solution <test1k >trash
==178771== Memcheck, a memory error detector
==178771== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==178771== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==178771== Command: ./solution
==178771==
==178771== HEAP SUMMARY:
==178771==   in use at exit: 122,880 bytes in 6 blocks
==178771==   total heap usage: 4,894 allocs, 4,888 frees, 2,434,497 bytes allocated
==178771==
==178771== LEAK SUMMARY:
==178771==   definitely lost: 0 bytes in 0 blocks
==178771==   indirectly lost: 0 bytes in 0 blocks
==178771==   possibly lost: 0 bytes in 0 blocks
==178771==   still reachable: 122,880 bytes in 6 blocks
==178771==   suppressed: 0 bytes in 0 blocks
==178771== Reachable blocks (those to which a pointer was found) are not shown.
==178771== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==178771==
==178771== For lists of detected and suppressed errors, rerun with: -s
==178771== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

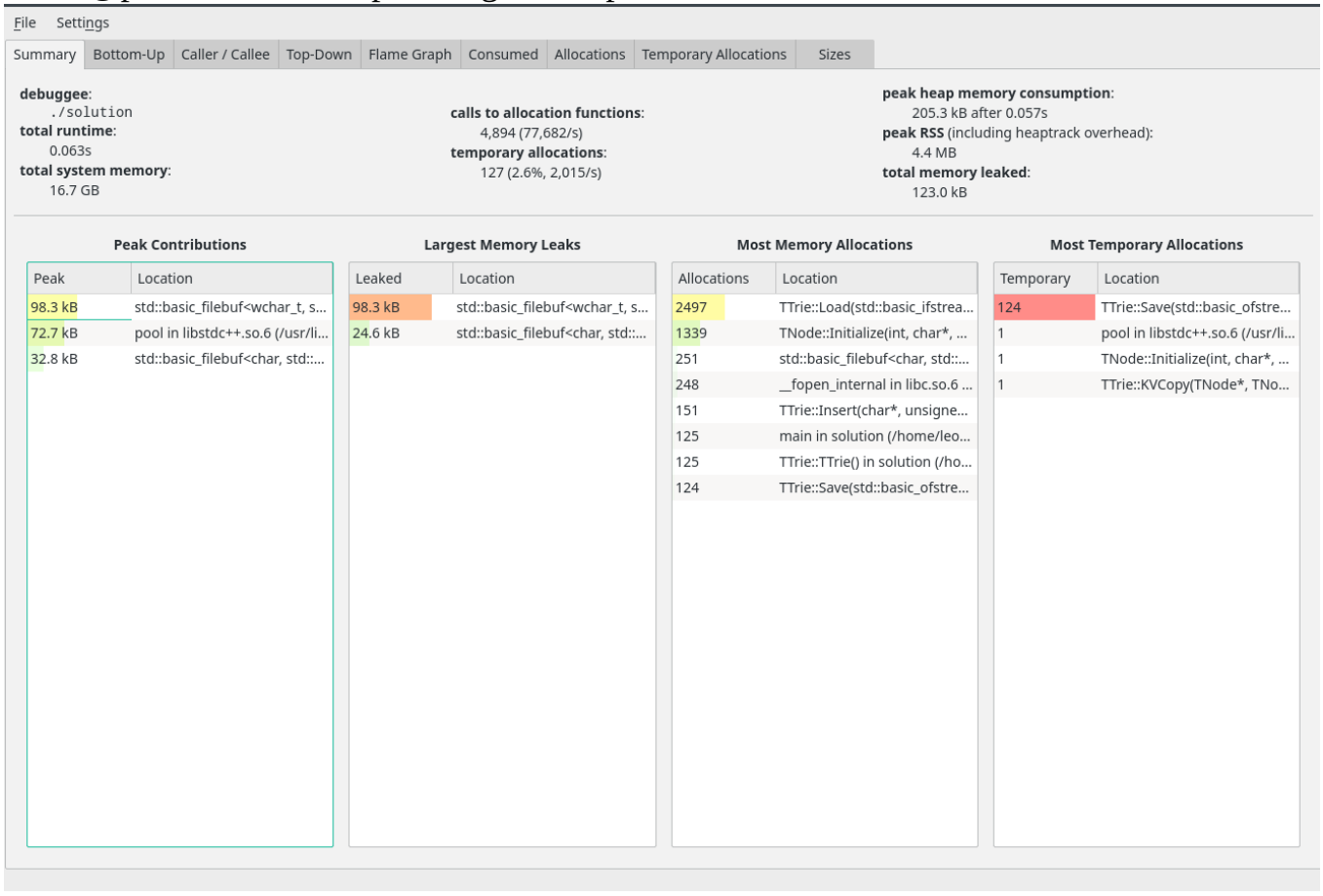
Проблема решена.

3.2. Heaptrack

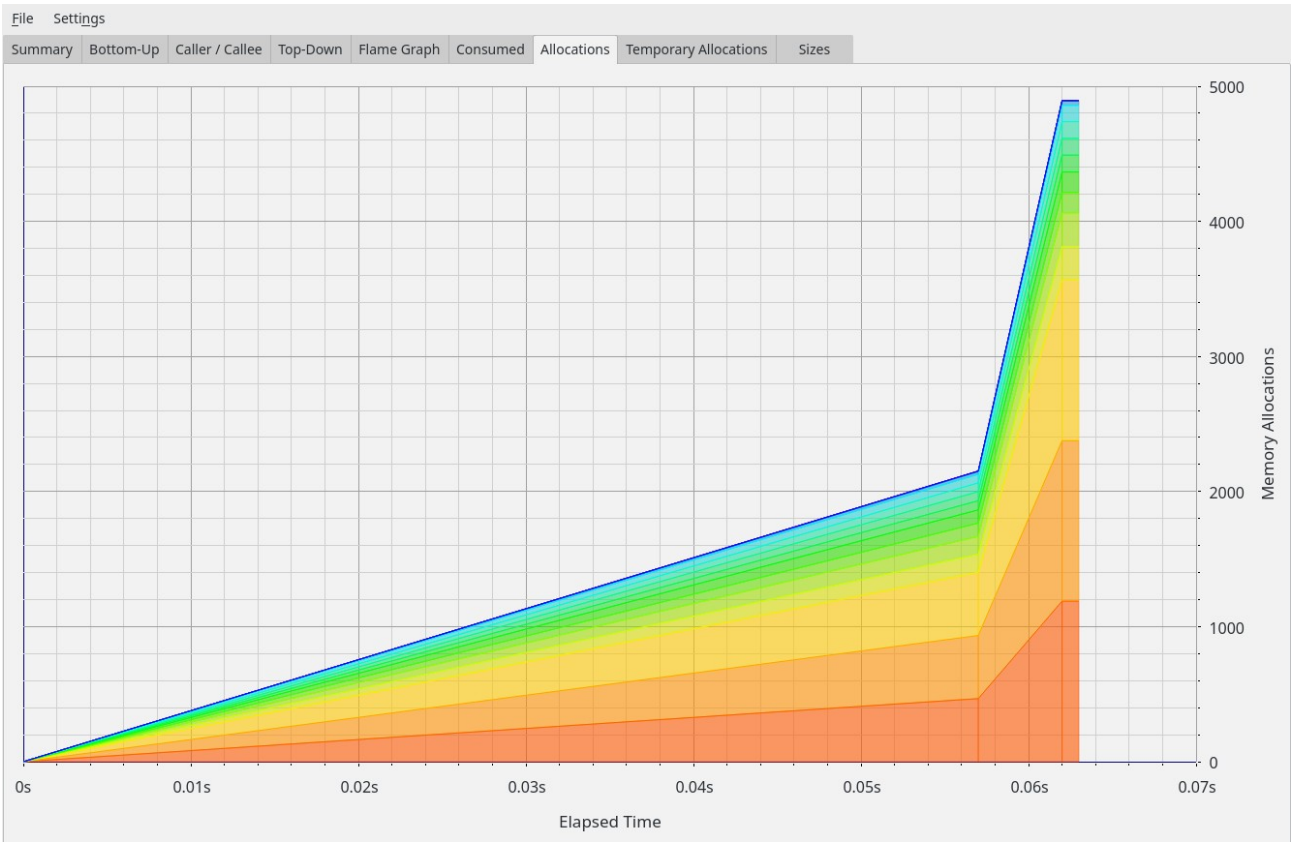
Еще одна потрясающая утилита — Heaptrack; помимо выяявления стандартных ошибок вроде поиска ошибок, она позволяет нам обнаружить части кода, которые больше всего злоупотребляют памятью.

[leo@pc solution]$ heaptrack ./solution <test1k

[leo@pc solution]$ heaptrack_gui ./heaptrack.solution.179379.zst



Интерфейс интуитивно понятен; богатый инструментарий данной программы позволяет просматривать пиковые значения потребляемой памяти, утечки и прочее в наглядном, графическом виде.



4. Вывод

Самым простым инструментом для профайлинга мне показался valgrind с его богатым инструментарием: от профайлинга, процентного соотношения времени исполнения той или иной функции, до непревзойденного обнаружения утечек памяти.

Бесспорно, что повышение производительности кода, борьба со всякого рода утечками, багами — обязательная часть процесса разработки ПО. Никакая высоконагруженная система не должна допускать никаких ошибок, иначе, рано или поздно произойдет какой-либо сбой, в чем приятности мало.