Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: С. А. Петрин
Преподаватель: Н. А. Зацепин
Группа: М8О-307Б-18
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №3

Задача:

Для реализации словаря из предыдущей лабораторной работы 2 необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить.

Результатом лабораторной работы является отчёт, состоящий из:

Дневника выполнения работы, в котором отражено что и когда делалось, какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.

Выводов о найденных недочётов.

Сравнение работы исправленной программы с предыдущей версии.

Общих выводов о выполнении лабораторной работы, полученном опыте.

Минимальный набор используемых средств должен содержать утилиту gprof и библиотеку dmalloc, однако их можно заменять на любые другие аналогичные или более известные утилиты (например, Valgrind или Shark) или добавлять к ним новые (например, gcov).

Используемая для тестирования структура данных: красно - чёрное дерево.

1 Описание

Так как предыдущая лабораторная работа уже была успешно выполнена, ошибок в финальной версии программы нет. В процессе написания кода у меня возникали различные ошибки, которые я смог исправить самостоятельно. Однако для устранения таких проблем намного проще и эффективнее использовать специальные инструменты, такие как профилировщики, менеджеры памяти - программы, собирающие и анализирующие различные характеристики работы исполняемого процесса (время выполнения, количество вызовов отдельных функций, количество обращений к строкам кода, наличие утечек памяти, наличие ошибок при работе с памятью и т.д.).

Для демонстрации работы таких инструментов я «верну» обратно несколько ошибок, возникших в процессе выполнения прошлой лабораторной работы и с помощью специальных инструментов (Valgrind, gprof, gcov, KCachegrind) проведу анализ, показав эффективность их использования.

2 Менеджер памяти Valgrind

В процессе выполнения лабораторной работы 2 у меня часто по невнимательности возникали глупые ошибки. Вот одна из таких:

```
1 if (unc->Color == R) { //1
2     unc->Color = B;
3     ins->P->Color = B;
4     ins->P->P->Color = R;
5     InsertFix(ins->P); //InsertFix(ins->P->P); is right
```

Такие опечатки часто приводят к ошибкам использования памяти. Используем Valgrind для их выявления. Вывод до исправления ошибки:

```
root@voozer-VirtualBox:~/Загрузки# g++ -g finalMain.cpp
root@voozer-VirtualBox:~/Загрузки# valgrind ./a.out < test
==3136== Memcheck, a memory error detector
==3136== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3136== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3136== Command: ./a.out
==3136==
OK
OK
OK
==3136== Invalid read of size 8
==3136==    at 0x10B3D8: TRBTree<TString, unsigned long long>::LeftRotate
(TRBTree<TString, unsigned long long>::TNode*) (finalMain.cpp:394)
==3136==    by 0x10AF91: TRBTree<TString, unsigned long long>::InsertFix
(TRBTree<TString, unsigned long long>::TNode*) (finalMain.cpp:260)
==3136==    by 0x10AE6A: TRBTree<TString, unsigned long long>::InsertFix
(TRBTree<TString, unsigned long long>::TNode*) (finalMain.cpp:246)
==3136==    by 0x10A761: TRBTree<TString, unsigned long long>::Insert
(TString const&, unsigned long long const&) (finalMain.cpp:230)
==3136==    by 0x109E4A: TRBTree<TString, unsigned long long>::VocAdd()
(finalMain.cpp:449)
==3136==    by 0x109722: main (finalMain.cpp:587)
==3136==  Address 0x28 is not stack'd, malloc'd or (recently) free'd
==3136==
==3136==
==3136== Process terminating with default action of signal 11 (SIGSEGV)
==3136==  Access not within mapped region at address 0x28
==3136==    at 0x10B3D8: TRBTree<TString, unsigned long long>::LeftRotate
(TRBTree<TString, unsigned long long>::TNode*) (finalMain.cpp:394)
```

```
==3136==    by 0x10AF91: TRBTree<TString, unsigned long long>::InsertFix
(TRBTree<TString, unsigned long long>::TNode*) (finalMain.cpp:260)
==3136==    by 0x10AE6A: TRBTree<TString, unsigned long long>::InsertFix
(TRBTree<TString, unsigned long long>::TNode*) (finalMain.cpp:246)
==3136==    by 0x10A761: TRBTree<TString, unsigned long long>::Insert
(TString const&, unsigned long long const&) (finalMain.cpp:230)
==3136==    by 0x109E4A: TRBTree<TString, unsigned long long>::VocAdd()
(finalMain.cpp:449)
==3136==    by 0x109722: main (finalMain.cpp:587)
==3136==  If you believe this happened as a result of a stack
==3136==  overflow in your program's main thread (unlikely but
==3136==  possible), you can try to increase the size of the
==3136==  main thread stack using the --main-stacksize= flag.
==3136==  The main thread stack size used in this run was 8388608.
==3136==
==3136== HEAP SUMMARY:
==3136==     in use at exit: 292 bytes in 11 blocks
==3136==   total heap usage: 20 allocs, 9 frees, 78,128 bytes allocated
==3136==
==3136== LEAK SUMMARY:
==3136==    definitely lost: 0 bytes in 0 blocks
==3136==    indirectly lost: 0 bytes in 0 blocks
==3136==      possibly lost: 0 bytes in 0 blocks
==3136==    still reachable: 292 bytes in 11 blocks
==3136==         suppressed: 0 bytes in 0 blocks
==3136== Rerun with --leak-check=full to see details of leaked memory
==3136==
==3136== For counts of detected and suppressed errors, rerun with: -v
==3136== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Ошибка сегментирования (стек памяти сброшен на диск)
```

Как можно заметить, Valgrind сообщает нам о том, что в программе была обнаружена одна ошибка, которая заставила программу завершиться по сигналу Segmentation Fault. Выше приводится описание вызванной ошибки: Invalid read of size 8 - чтение 8 байт за пределами области выделенной для процесса памяти; Address 0x28 is not stack'd, malloc'd or (recently) free'd - адрес области памяти, из которой совершено чтение, начинается с 0x28. Обычно такое описание (начальный адрес довольно мал, размер 8 байт - размер указателя на 64-битной машине) означает, что в процессе выполнения произошло разыменование nullptr'a. По приведённому стеку вызовов функций с указанием строк (если программа компилирована с ключом -g) можно проследить и узнать, что вызвало ошибку. Просмотрев более внимательно эти функ-

ции, я смог найти опечатку на 246 строке.

Также часто возникали утечки памяти. Искать их легко при помощи Valgrind и ключа –leak-check=full. Вывод до исправления:

```
root@voozer-VirtualBox:~/Загрузки# valgrind --leak-check=full ./a.out < test
==3399== Memcheck, a memory error detector
==3399== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3399== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3399== Command: ./a.out
==3399==
OK
OK
OK
OK
OK
...
==3399==
==3399== HEAP SUMMARY:
==3399==     in use at exit: 414 bytes in 13 blocks
==3399==   total heap usage: 75 allocs, 62 frees, 78,795 bytes allocated
==3399==

==3399== 414 (56 direct, 358 indirect) bytes in 1 blocks are
definitely lost in loss record 3 of 3
==3399==    at 0x4C3017F: operator new(unsigned long)
(in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3399==    by 0x109CB4: TRBTree<TString, unsigned long long>::TRBTree()
(finalMain.cpp:170)
==3399==    by 0x1096A6: main (finalMain.cpp:580)
==3399==
==3399== LEAK SUMMARY:
==3399==    definitely lost: 56 bytes in 1 blocks
==3399==    indirectly lost: 358 bytes in 12 blocks
==3399==      possibly lost: 0 bytes in 0 blocks
==3399==    still reachable: 0 bytes in 0 blocks
==3399==         suppressed: 0 bytes in 0 blocks
==3399==
==3399== For counts of detected and suppressed errors, rerun with: -v
==3399== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Также видим, что Valgrind нашёл ошибку: 414 (56 direct, 358 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 3 - выделенная память для указателя не

была освобождена, причём указатель вышел за область видимости. По стеку вызовов видим, где была выделена данная область памяти: by 0x109CB4: TRBTree<TString, unsigned long long>::TRBTree() (finalMain.cpp:170). Эта строка: Nil = new TNode;. Почему не была освобождена память? Потому что я забыл написать деконструктор в классе красно-чёрного дерева, вызывающий delete Nil.

Исправив эти ошибки, я убедился, что всё в порядке:

```
root@voozer-VirtualBox:~/Загрузки# valgrind ./a.out < test
==3330== Memcheck, a memory error detector
==3330== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3330== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3330== Command: ./a.out
==3330==
==3330==ASan runtime does not come first in initial library list; you should
either link runtime to your application or manually preload it with LD_PRELOAD.
==3330==
==3330== HEAP SUMMARY:
==3330==     in use at exit: 0 bytes in 0 blocks
==3330==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3330==
==3330== All heap blocks were freed -- no leaks are possible
==3330==
==3330== For counts of detected and suppressed errors, rerun with: -v
==3330== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

3 Профилировщики gprof, gcov, KCachegrind

Менеджеры памяти хотя и помогают исправить многие ошибки, связанные с памятью, но не позволяют оценить быстродействие программы и выявить самые «тяжёлые» участки кода. В этом нам помогут профилировщики.

Моя написанная в прошлой лабораторной работе программа работала без ошибок, но очень медленно. Эта проблема была выявлена путём тестирования программы на большом количестве тестов. Но было неизвестно, что именно замедляет работу программы. В решении этой проблемы мне помогли данные инструменты.

Я использовал профилятор gprof, позволяющий узнать общее количество вызовов используемых функций, а также общее время выполнения для каждой из них. Однако время gprof показать не смог, наверное, потому что я работаю на виртуальной машине. Тем не менее из выведенного отчета я смог понять, какие функции выполняются больше всего раз (столбец calls).

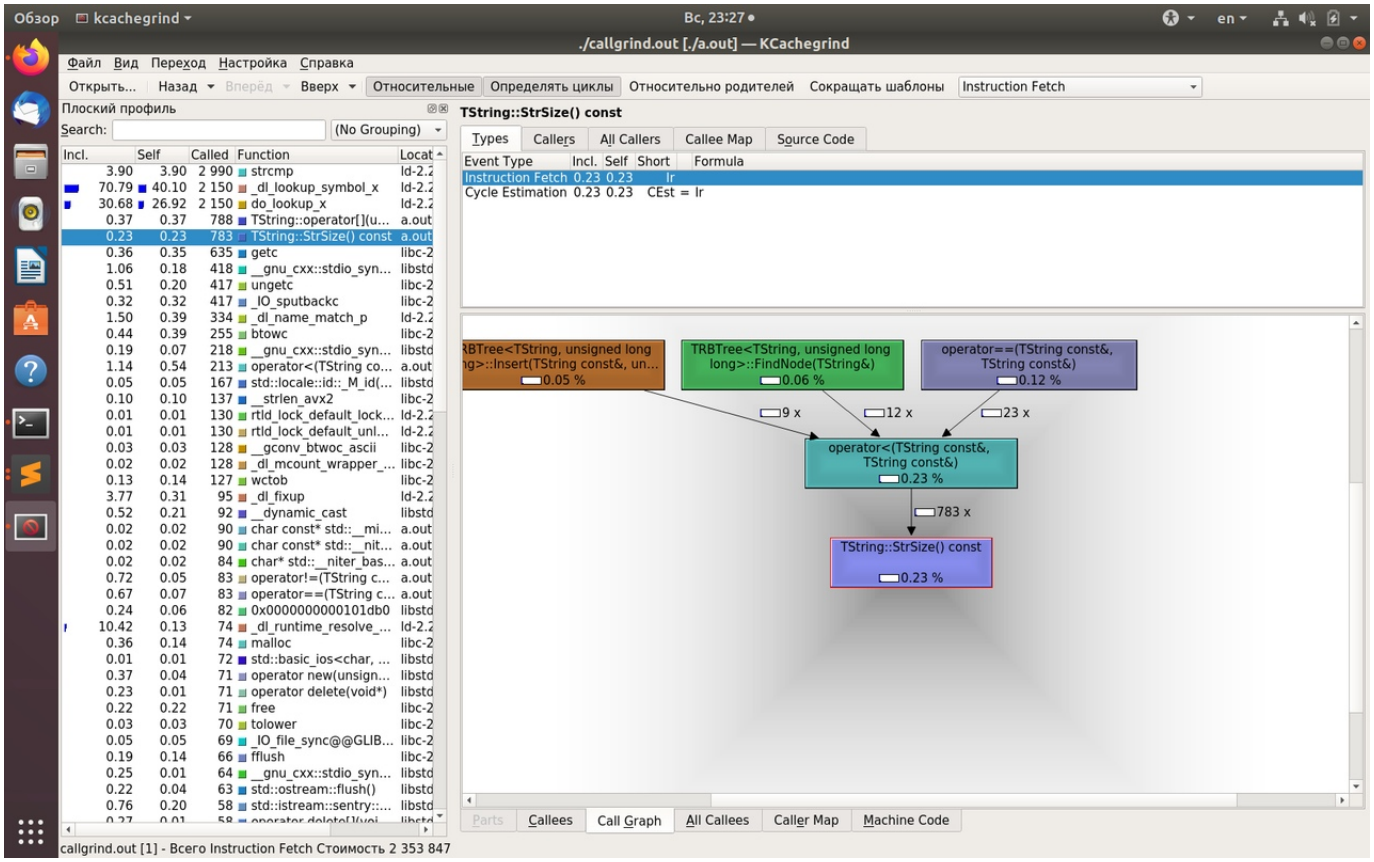
```
root@voozer-VirtualBox:~/Загрузки# g++ -pg finalMain.cpp
root@voozer-VirtualBox:~/Загрузки# ./a.out < test
root@voozer-VirtualBox:~/Загрузки# gprof
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      788     0.00     0.00  TString::operator[](unsigned long) const
  0.00      0.00     0.00      783     0.00     0.00  TString::StrSize() const
  0.00      0.00     0.00      213     0.00     0.00  operator<(TString const&, TString const&)
  0.00      0.00     0.00       90     0.00     0.00  char const* std::__miter_base<char const*>(char const*)
  0.00      0.00     0.00       90     0.00     0.00  char const* std::__niter_base<char const*>(char const*)
  0.00      0.00     0.00       84     0.00     0.00  char* std::__niter_base<char*>(char*)
  0.00      0.00     0.00       83     0.00     0.00  operator==(TString const&, TString const&)
  0.00      0.00     0.00       83     0.00     0.00  operator!=(TString const&, TString const&)
  0.00      0.00     0.00       58     0.00     0.00  char* std::__copy_move<false, true, std::random_access_iterator_tag>::__copy_m<char>(char const*, char const*,
  0.00      0.00     0.00       50     0.00     0.00  TString::begin()
  0.00      0.00     0.00       46     0.00     0.00  operator>>(std::istream&, TString&)
  0.00      0.00     0.00       45     0.00     0.00  TString::operator=(char const*)
  0.00      0.00     0.00       45     0.00     0.00  char* std::__copy_move_a<false, char const*, char*>(char const*, char const*, char*)
  0.00      0.00     0.00       45     0.00     0.00  char* std::__copy_move_a2<false, char const*, char*>(char const*, char const*, char*)
  0.00      0.00     0.00       45     0.00     0.00  char* std::copy<char const*, char*>(char const*, char const*, char*)
  0.00      0.00     0.00       36     0.00     0.00  TString::~TString()
  0.00      0.00     0.00       26     0.00     0.00  char* std::__miter_base<char*>(char*)
  0.00      0.00     0.00       25     0.00     0.00  TRBTree<TString, unsigned long long>::FindNode(TString&)
  0.00      0.00     0.00       25     0.00     0.00  TString::end()
  0.00      0.00     0.00       25     0.00     0.00  TString::operator[](unsigned long)
  0.00      0.00     0.00       25     0.00     0.00  char* std::transform<char*, char*, int (*)(int)>(char*, char*, char*, int (*)(int))
  0.00      0.00     0.00       23     0.00     0.00  TString::TString()
    ...
  0.00      0.00     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZltRK7TStringS1_
  0.00      0.00     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.00     0.00        1     0.00     0.00  TRBTree<TString, unsigned long long>::RightRotate(TRBTree<TString, unsigned long long>::TNode*)
  0.00      0.00     0.00        1     0.00     0.00  TRBTree<TString, unsigned long long>::Clean(TRBTree<TString, unsigned long long>::TNode*)
  0.00      0.00     0.00        1     0.00     0.00  TRBTree<TString, unsigned long long>::TNode::TNode()
  0.00      0.00     0.00        1     0.00     0.00  TRBTree<TString, unsigned long long>::TRBTree()
  0.00      0.00     0.00        1     0.00     0.00  TRBTree<TString, unsigned long long>::~TRBTree()
```

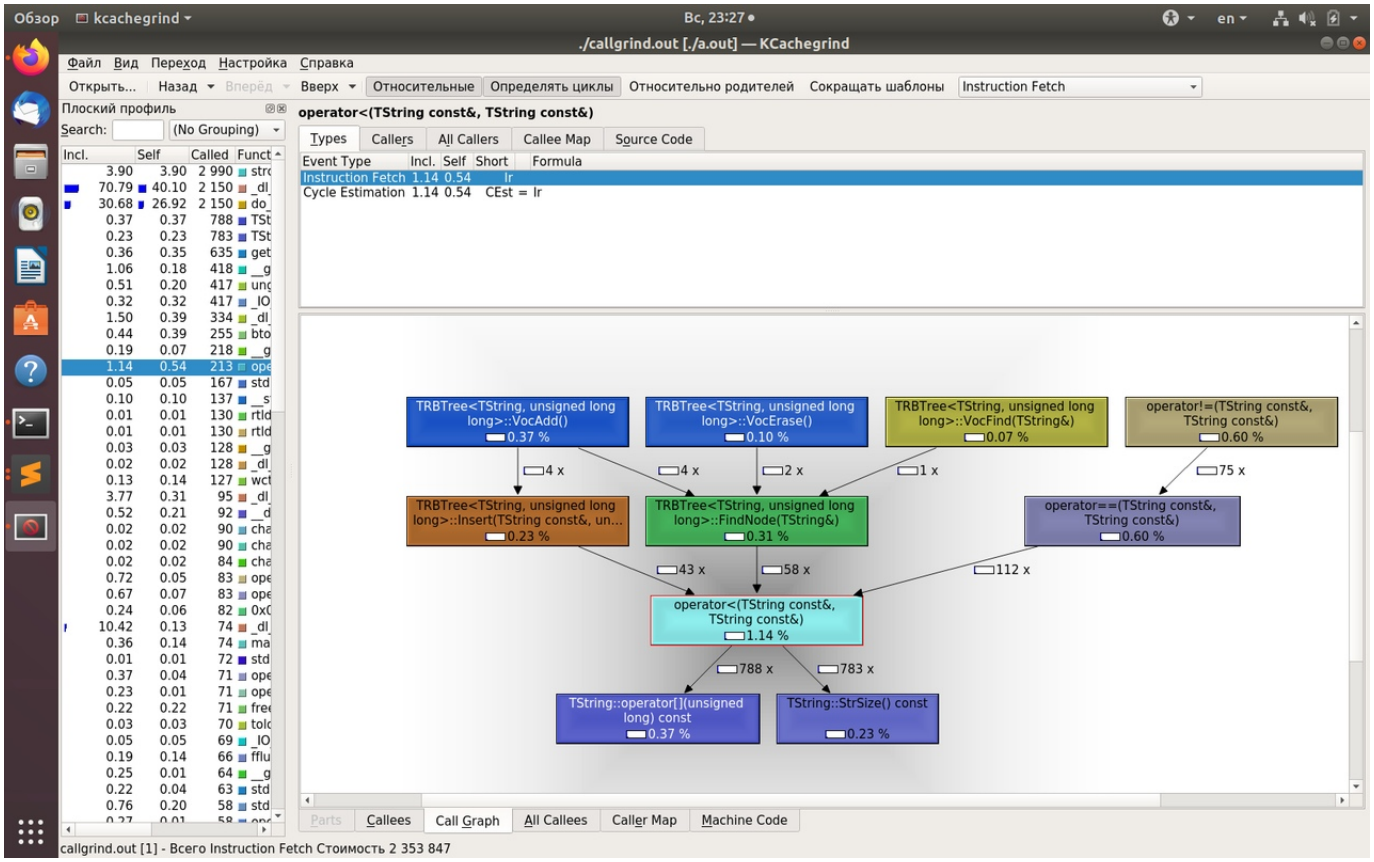
Также gprof генерирует call graph - граф вызовов функций, однако предоставляемый в текстовом виде граф, по моему мнению, сложно и неудобно читать, поэтому я

решил не включать его в отчёт и вместо этого использовать другой инструмент - KCachegrind.

KCachegrind предоставляет удобный графический интерфейс на основе отчёта, полученного из программы Callgrind, которая идёт вместе с Valgrind. Вывод программы:

```
root@voozer-VirtualBox:~/Загрузки# valgrind --tool=callgrind ./a.out < test
==7557== Callgrind, a call-graph generating cache profiler
==7557== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7557== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7557== Command: ./a.out
==7557==
==7557== For interactive control, run 'callgrind_control -h'.
OK
OK
OK
OK
OK
...
==7557==
==7557== Events    : Ir
==7557== Collected : 2353847
==7557==
==7557== I   refs:      2,353,847
root@voozer-VirtualBox:~/Загрузки# kcachegrind
```


Обзор
kcachegrind
Вс, 23:27
en
./callgrind.out [./a.out] — KCachegrind
Файл Вид Переход Настройка Справка
Открыть... Назад Вперёд Вверх Относительные Определять циклы Относительно родителей Сокращать шаблоны Instruction Fetch
Плоский профиль
Search: (No Grouping)
Incl. Self Called Function Locat
3.90 3.90 2 990 strcmp ld-2.2
70.79 40.10 2 150 _dl_lookup_symbol_x ld-2.2
30.68 26.92 2 150 do_lookup_x ld-2.2
0.37 0.37 788 TString::operator[](u... a.out
0.23 0.23 783 TString::StrSize() const a.out
0.36 0.35 635 getc libc-2
1.06 0.18 418 __gnu_cxx::stdio_syn... libstd
0.51 0.20 417 ungetc libc-2
0.32 0.32 417 _IO_sputbackc libc-2
1.50 0.39 334 _dl_name_match_p ld-2.2
0.44 0.39 255 btowc libc-2
0.19 0.07 218 __gnu_cxx::stdio_syn... libstd
1.14 0.54 213 operator<(TString co... a.out
0.05 0.05 167 std::locale::id::_M_id(... libstd
0.10 0.10 137 __strlen_avx2 libc-2
0.01 0.01 130 rtld_lock_default_lock... ld-2.2
0.01 0.01 130 rtld_lock_default_unl... ld-2.2
0.03 0.03 128 __gconv_btwoc_ascii libc-2
0.02 0.02 128 _dl_mcount_wrapper_... libc-2
0.13 0.14 127 wctob libc-2
3.77 0.31 95 _dl_fixup ld-2.2
0.52 0.21 92 __dynamic_cast libstd
0.02 0.02 90 char const* std::__mi... a.out
0.02 0.02 90 char const* std::__nit... a.out
0.02 0.02 84 char* std::__niter_bas... a.out
0.72 0.05 83 operator!=(TString c... a.out
0.67 0.07 83 operator==(TString c... a.out
0.24 0.06 82 0x0000000000101db0 libstd
10.42 0.13 74 _dl_runtime_resolve_... ld-2.2
0.36 0.14 74 malloc libc-2
0.01 0.01 72 std::basic_ios<char, ... libstd
0.37 0.04 71 operator new(unsign... libstd
0.23 0.01 71 operator delete(void*) libstd
0.22 0.22 71 free libc-2
0.03 0.03 70 tolower libc-2
0.05 0.05 69 _IO_file_sync@@GLIB... libc-2
0.19 0.14 66 fflush libc-2
0.25 0.01 64 __gnu_cxx::stdio_syn... libstd
0.22 0.04 63 std::ostream::flush() libstd
0.76 0.20 58 std::istream::sentry::... libstd
callgrind.out [1] - Всего Instruction Fetch Стоимость 2 353 847
TString::StrSize() const
Types Callers All Callers Callee Map Source Code
Event Type Incl. Self Short Formula
Instruction Fetch 0.23 0.23 Ir
Cycle Estimation 0.23 0.23 CEst = Ir
RBTree<TString, unsigned long ng>::Insert(TString const&, un... 0.05 %
TRBTree<TString, unsigned long long>::FindNode(TString&) 0.06 %
operator==(TString const&, TString const&) 0.12 %
9 x
12 x
23 x
operator<(TString const&, TString const&) 0.23 %
783 x
TString::StrSize() const 0.23 %
Parts Callees Call Graph All Callees Caller Map Machine Code



В окне слева отображены все функции, выполненные в исполняемом файле, количество их вызовов, а также степень использования других функций (столбец Incl.), степень использования отдельно самой функции (столбец Self), также указывается расположение.

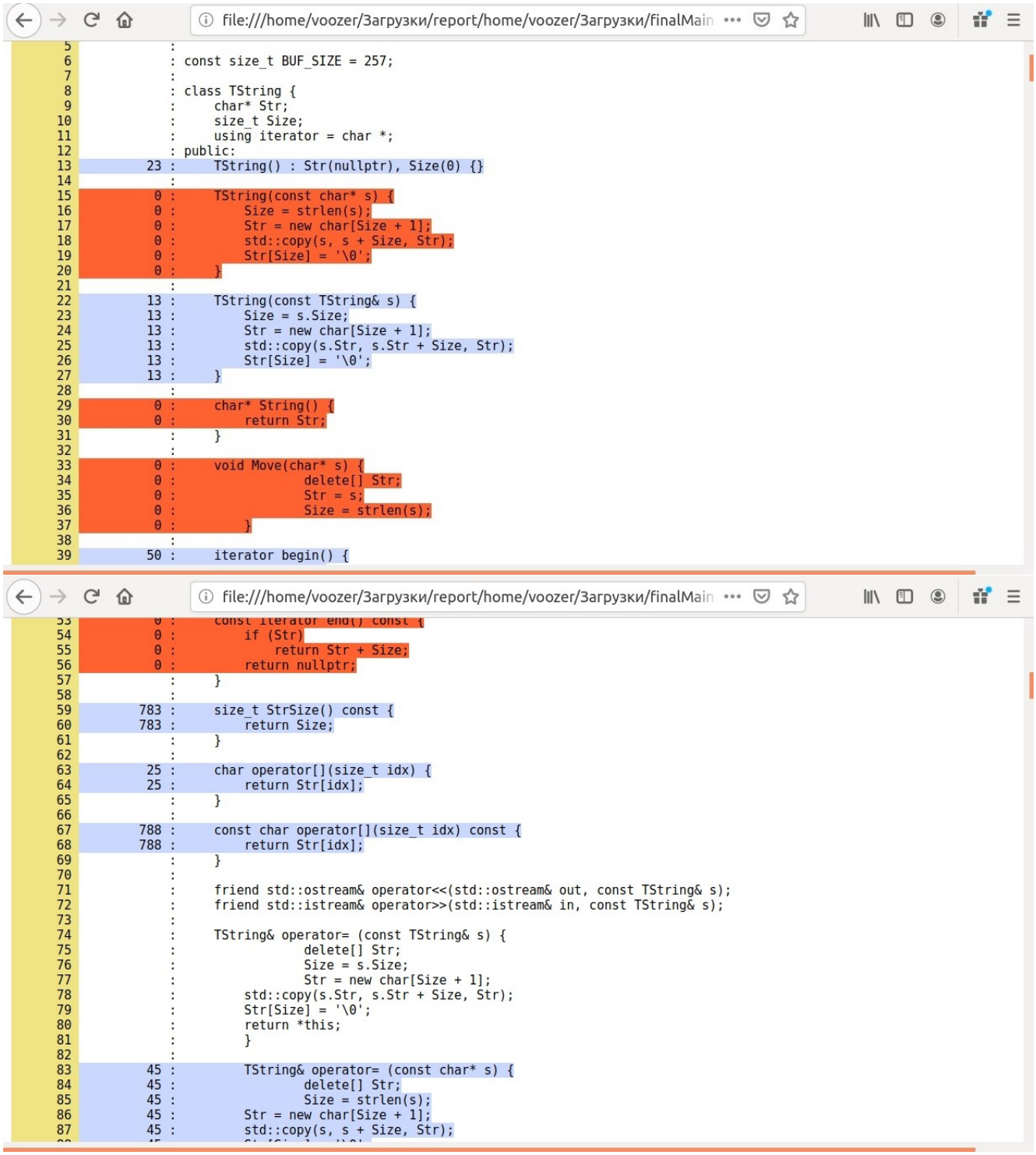
В правом верхнем окне отображена более подробная информация для выбранной функции, включающая список функций, вызывающих её, а также source code.

И, наконец, в правом нижнем углу располагается граф вызовов для выбранной функции. Граф интерактивный, можно перемещаться через вложенные вершины и таким образом определять весь путь до нужной функции.

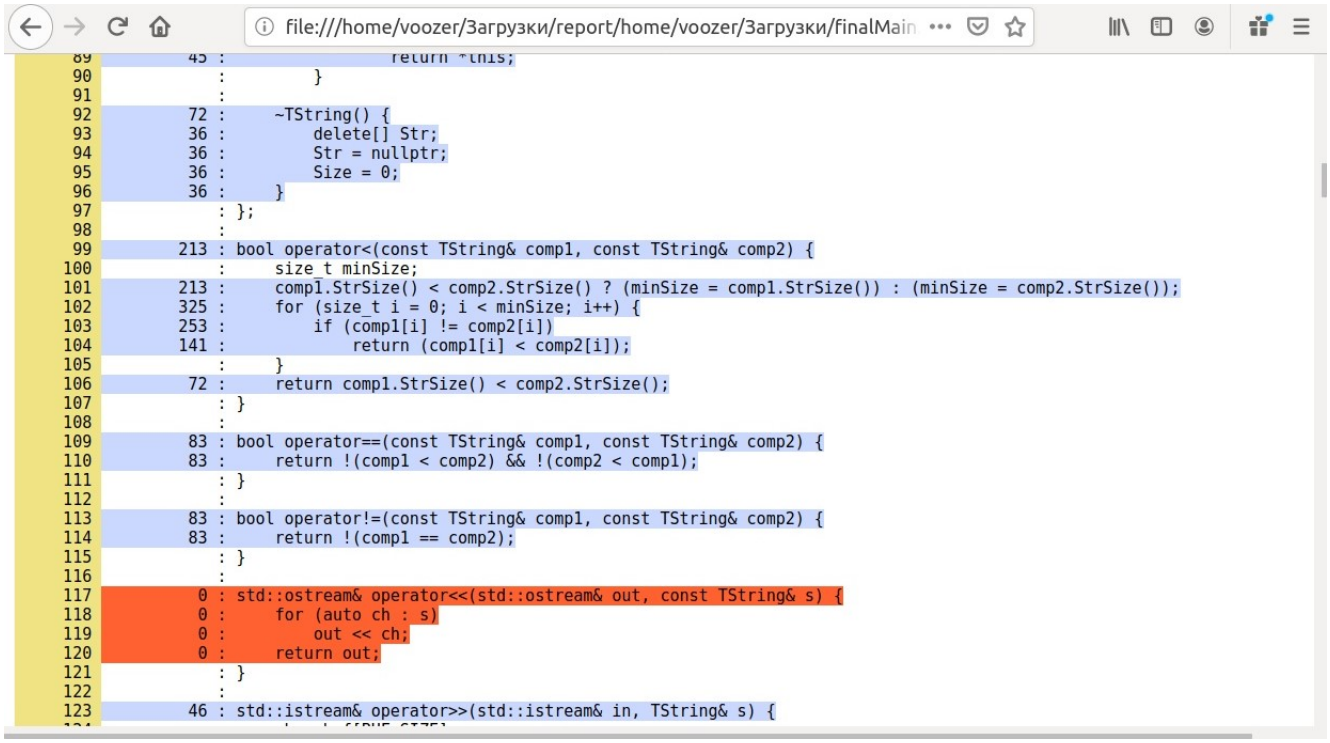
Еще одним интересным и удобным в использовании профилировщиком является gcov. Он позволяет просмотреть загруженность участков программы построчно, прямо в самом коде исполняемого файла. Для более удобного интерфейса используется графическое дополнение lcov, позволяющее перевести информацию в отдельный файл и загрузить её при помощи генерации html-страницы. Gcov также позвоялет определить code coverage rate - степень использования строк кода. При помощи этого можно выявить неиспользуемые участки и сократить программу, сделать её проще, быстрее и менее объёмной.

Данный вывод для моей программы:

```
root@voozer-VirtualBox:~/Загрузки# g++ -g -coverage finalMain.cpp
root@voozer-VirtualBox:~/Загрузки# ./a.out < test
root@voozer-VirtualBox:~/Загрузки# lcov -t "report" -o "report.info" -c -d .
Capturing coverage data from .
Found gcov version: 7.5.0
Scanning . for .gcda files ...
Found 1 data files in .
Processing finalMain.gcda
Finished .info-file creation

root@voozer-VirtualBox:~/Загрузки# genhtml -o report report.info
Reading data file report.info
Found 6 entries.
Found common filename prefix "/usr/include/c++"
Writing .css and .png files.
Generating output.
Processing file /home/voozer/Загрузки/finalMain.cpp
Processing file 7/iostream
Processing file 7/bits/stl_algo.h
Processing file 7/bits/stl_algobase.h
Processing file 7/bits/cpp_type_traits.h
Processing file 7/bits/ios_base.h
Writing directory view page.
Overall coverage rate:
  lines......: 66.3% (264 of 398 lines)
  functions..: 80.7% (46 of 57 functions)
```




```
        :
        : const size_t BUF_SIZE = 257;
        :
        : class TString {
        :     char* Str;
        :     size_t Size;
        :     using iterator = char *;
        : public:
     23 :     TString() : Str(nullptr), Size(0) {}
        :
      0 :     TString(const char* s) {
      0 :         Size = strlen(s);
      0 :         Str = new char[Size + 1];
      0 :         std::copy(s, s + Size, Str);
      0 :         Str[Size] = '\0';
      0 :     }
        :
     13 :     TString(const TString& s) {
     13 :         Size = s.Size;
     13 :         Str = new char[Size + 1];
     13 :         std::copy(s.Str, s.Str + Size, Str);
     13 :         Str[Size] = '\0';
     13 :     }
        :
      0 :     char* String() {
      0 :         return Str;
        :     }
        :
      0 :     void Move(char* s) {
      0 :                 delete[] Str;
      0 :                 Str = s;
      0 :                 Size = strlen(s);
      0 :         }
        :
     50 :     iterator begin() {
      0 :     const iterator end() const {
      0 :         if (Str)
      0 :             return Str + Size;
      0 :         return nullptr;
        :     }
        :
    783 :     size_t StrSize() const {
    783 :         return Size;
        :     }
        :
     25 :     char operator[](size_t idx) {
     25 :         return Str[idx];
        :     }
        :
    788 :     const char operator[](size_t idx) const {
    788 :         return Str[idx];
        :     }
        :
        :     friend std::ostream& operator<<(std::ostream& out, const TString& s);
        :     friend std::istream& operator>>(std::istream& in, const TString& s);
        :
        :     TString& operator= (const TString& s) {
        :                 delete[] Str;
        :                 Size = s.Size;
        :                 Str = new char[Size + 1];
        :         std::copy(s.Str, s.Str + Size, Str);
        :         Str[Size] = '\0';
        :         return *this;
        :         }
        :
     45 :         TString& operator= (const char* s) {
     45 :                 delete[] Str;
     45 :                 Size = strlen(s);
     45 :         Str = new char[Size + 1];
     45 :         std::copy(s, s + Size, Str);
```



Вернёмся к проблеме. При помощи этих инструментов я смог выявить самые используемые функции - это были функции обращения по оператору [], получения размера строки, а также обращения к перегруженному оператору $<$. Изучив граф вызовов и сами эти функции, я понял, что первые две функции относятся к реализации перегрузки оператора $<$, который используется при вставке в дерево, а в самой реализации нашёл лишние строки, выполняющие бесполезное и долгое перемещение элементов, которое я использовал для понижения регистра. Затем я убрал эти строки, заменив их алгоритмом std::tolower, и моя программа заработала в разы быстрее.

4 Выводы

Выполнив третью лабораторную работу, я смог изучить новые для себя инструменты анализа работы программ, которые, очень вероятно, пригодятся мне при выполнении дальнейших проектов. Раньше я обходился только использованием Valgrind и поэтому не мог выявить все ошибки, но теперь у меня появились более эффективные способы отлдаки программ, с помощью которых можно обнаружить большую часть ошибок и нерационального использования ресурсов. Без этих инструментов программист не будет эффективен: профилировщики и менеджеры памяти позволяют, во-первых, намного быстрее найти ошибки в программе, во-вторых, выявить недостатки программы, которые не возможно выявить «глазами», в-третьих, оптимизировать использование наиболее важных ресурсов.

Список литературы

[1] *Understanding Valgrind errors*
URL: `https://derickrethans.nl/valgrind-null.html` (дата обращения: 26.04.2020).

[2] *How to profile C++ application with Callgrind / KCacheGrind*
URL: `https://baptiste-wicht.com/posts/2011/09/profile-c-application-with-callgrind` (дата обращения: 26.04.2020).

[3] *Профилятор gprof*
URL: `https://www.opennet.ru/docs/RUS/gprof/` (дата обращения: 26.04.2020).

[4] *Code Coverage средствами GCC*
URL: `https://ps-group.github.io/cxx/coverage_gcc` (дата обращения: 26.04.2020).