# Профайлинг в C++

## Обзор ПО для профилирования

# О себе

- C++ разработчик в SolarWinds
- Участвую в деятельности Российской РГ21
- Разработчик Boost.Algorithm и gImageReader
- zamazan4ik везде - Telegram, Twitter, GitHub, Reddit, etc.
- https://github.com/ZaMaZaN4iK

# Постановка проблемы

- Есть приложение
- Оно тормозит
- Мы хотим (нас заставляют) это исправить
- А мы понятия не имеем, с какой стороны подступиться
- Искать руками долго/лень/сроки поджимают/что-то ещё (нужное подчеркнуть)

# Профилирование, профилировщики и т.д.

- **Профилирование** служит для получения информации о работе программы в целом: какие функции сколько времени выполняются, нахождение «горячих» мест и других особенностей работы программы
- **Профилировщик** - инструмент, который занимается профилировкой

# Классификация ПО

- Эмуляторы\симуляторы
- Сэмплеры
- Ручные (да-да, так тоже можно :)
- Event-based

# Сэмплеры

- Регулярно «срезают» ресурсы программы
- Почти не влияют на скорость выполнения (зависит от частоты)
- Некоторые функции пропадают :-)
- Погрешности

# Эмуляторы

- Эмуляция профилируемого устройства
- Значительное замедление программы
- Функции не пропадают :-)
- Сложность реализации
- Соответствие реальному аппаратному обеспечению (нет)

# Проблемы

- Разные производители процессоров (Intel, AMD, etc.)
- Разные архитектуры
- Разные ОС (GNU/Linux-based, Windows, macOS, Android, iOS, etc.)




# gprof

- GNU/Linux
- Консольный интерфейс
- Требует перекомпиляции проекта
- Профилирует только user-space код
- Не может профилировать многопоточные приложения
- https://sourceware.org/binutils/docs/gprof/

# gprof: как использовать

1. Пишем программу
2. Компилируем с флагом “pg”
3. Запускаем программу
4. Запускаем gprof с получившимся файлом
5. Анализируем результат

# gcov

- Консольный интерфейс
- Считает покрытие тестами кода
- Но оказывается тоже может профилировать!
- Занимается line-by-line профилированием

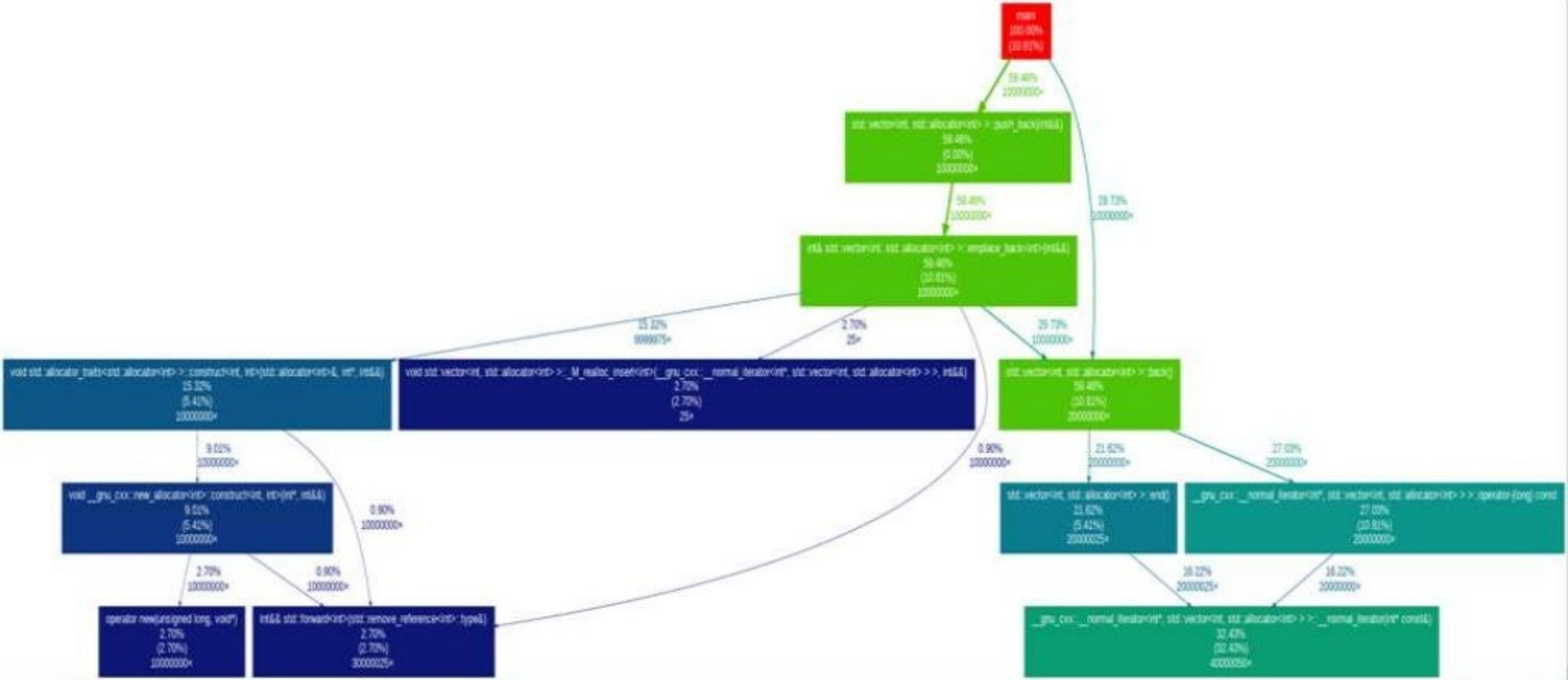
# gcov: как использовать для профилирования

1. Пишем программу
2. Компилируем её с флагами “**-fprofile-arcs -ftest-coverage**”
3. Запускаем программу (появятся файлы для gcov)
4. Запускаем gcov с файлом исходного кода
5. Анализируем результат

# gprof2dot

- Написан на Python 3
- Понадобится Graphviz - https://www.graphviz.org/
- Конвертер в dot graph
- Поддерживает: perf, callgrind, oprofile, sysprof, Vtune, prof, gprof, xperf, Very Sleepy
- https://github.com/jrfonseca/gprof2dot

# gprof2dot: пример результата



# gprof2dot: примеры использования

- gprof path/to/your/executable | gprof2dot.py | dot -Tpng -o output.png
- perf record -g -- /path/to/your/executable
  perf script | c++filt | gprof2dot.py -f perf | dot -Tpng -o output.png
- amplxe-cl -report gprof-cc -result-dir output -format text -report-output output.txt
  gprof2dot.py -f axe output.txt | dot -Tpng -o output.png

# gperftools (Google Performance Tools)

- GNU\Linux, ...
- Умеет в многопоточные приложения
- Ручная разметка интересующих вас мест для профилирования
- Крайне низкие накладные расходы
- https://github.com/gperftools/gperftools

# gperftools: как использовать

1. Написать программу, подключить **<gperftools/profiler.h>**
2. Пометить интересующие вас места ProfilerStart(“file.log”)/ProfilerStop()
3. Скомпилировать с отладочными символами
4. Слинковать с profiler.so
5. Запустить
6. Полученный file.log конвертировать в тот же callgrind:
    a. pprof --callgrind ./test file.log > profile.callgrind

# Valgrind

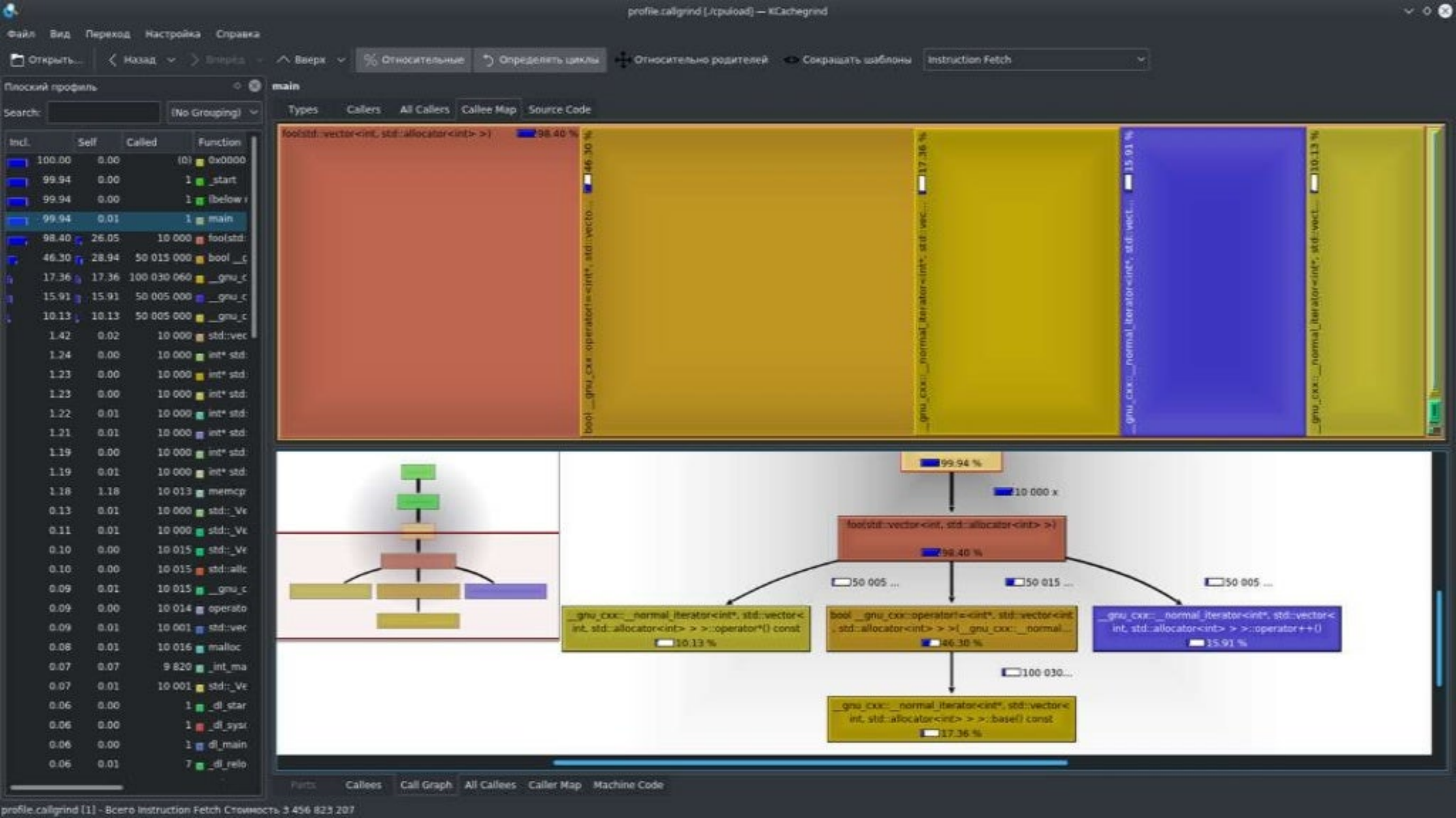
- Эмулятор
- GNU/Linux, macOS
- Open-Source
- Утилиты: Cachegrind, Callgrind, DRD, Helgrind, Massif, Memcheck
- Расширяемость
- http://valgrind.org/

# Valgrind: особенности

- Очень сильно замедляет работу программы
  - 10-50x
  - Зависит от утилиты и настроек
- Старается эмулировать процессор
  - Branch-prediction на уровне процессоров 2004 года
  - Попытки смоделировать ваш кеш (гадает по CPUID)
  - Если не смог подобрать под Ваш процессор, то... всё грустно
  - Можете ему помочь и сами написать свою реализацию кеша :)

# kcachegrind/qcachegrind

- Платформы
  - kcachegrind: GNU/Linux
  - qcachegrind: GNU/Linux, Windows, macOS
- Программа для просмотра “выхлопа” профилировщика
- https://github.com/KDE/kcachegrind


profile.callgrind [./cpuload] — 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
100.00 0.00 (0) 0x0000
99.94 0.00 1 _start
99.94 0.00 1 libelow
99.94 0.01 1 main
98.40 26.05 10 000 foo(std:
46.30 28.94 50 015 000 bool __g
17.36 17.36 100 030 060 __gnu_c
15.91 15.91 50 005 000 __gnu_c
10.13 10.13 50 005 000 __gnu_c
1.42 0.02 10 000 std::vec
1.24 0.00 10 000 int* std:
1.23 0.00 10 000 int* std:
1.23 0.00 10 000 int* std:
1.22 0.01 10 000 int* std:
1.21 0.01 10 000 int* std:
1.19 0.00 10 000 int* std:
1.19 0.01 10 000 int* std:
1.18 1.18 10 013 memcp
0.13 0.01 10 000 std::_Ve
0.11 0.01 10 000 std::_Ve
0.10 0.00 10 015 std::_Ve
0.10 0.00 10 015 std::allc
0.09 0.01 10 015 __gnu_c
0.09 0.00 10 014 operato
0.09 0.01 10 001 std::vec
0.08 0.01 10 016 malloc
0.07 0.07 9 820 _int_ma
0.07 0.01 10 001 std::_Ve
0.06 0.00 1 _dl_star
0.06 0.00 1 _dl_sysc
0.06 0.00 1 dl_main
0.06 0.01 7 _dl_relo
main
Types Callers All Callers Callee Map Source Code
foo(std::vector<int, std::allocator<int> >) 98.40 %
bool __gnu_cxx::operator!=<int*, std::vecto... 46.30 %
__gnu_cxx::__normal_iterator<int*, std::vec... 17.36 %
__gnu_cxx::__normal_iterator<int*, std::vect... 15.91 %
__gnu_cxx::__normal_iterator<int*, std::vect... 10.13 %
99.94 %
10 000 x
foo(std::vector<int, std::allocator<int> >)
98.40 %
50 005 ...
50 015 ...
50 005 ...
__gnu_cxx::__normal_iterator<int*, std::vector<
int, std::allocator<int> > >::operator*() const
10.13 %
bool __gnu_cxx::operator!=<int*, std::vector<int
, std::allocator<int> > >(__gnu_cxx::__normal...
46.30 %
__gnu_cxx::__normal_iterator<int*, std::vector<
int, std::allocator<int> > >::operator++()
15.91 %
100 030...
__gnu_cxx::__normal_iterator<int*, std::vector<
int, std::allocator<int> > >::base() const
17.36 %
Parts Callees Call Graph All Callees Caller Map Machine Code
profile.callgrind [1] - Всего Instruction Fetch Стоимость 3 456 823 207

# Heaptrack

- https://github.com/KDE/heaptrack
- Linux-only
- GUI + консольный интерфейс
- Профилирует и визуализирует:
  - Динамику потребления памяти во времени
  - Пиковые потребления
  - Количество “утёкшей” памяти
  - Распределение потребления памяти по функциям
  - Flame Graph потребления памяти

File

Summary | Вызовы | Caller / Callee | Top-Down | Flame Graph | Consumed | Выделения | Temporary Allocations | Выделено | Sizes

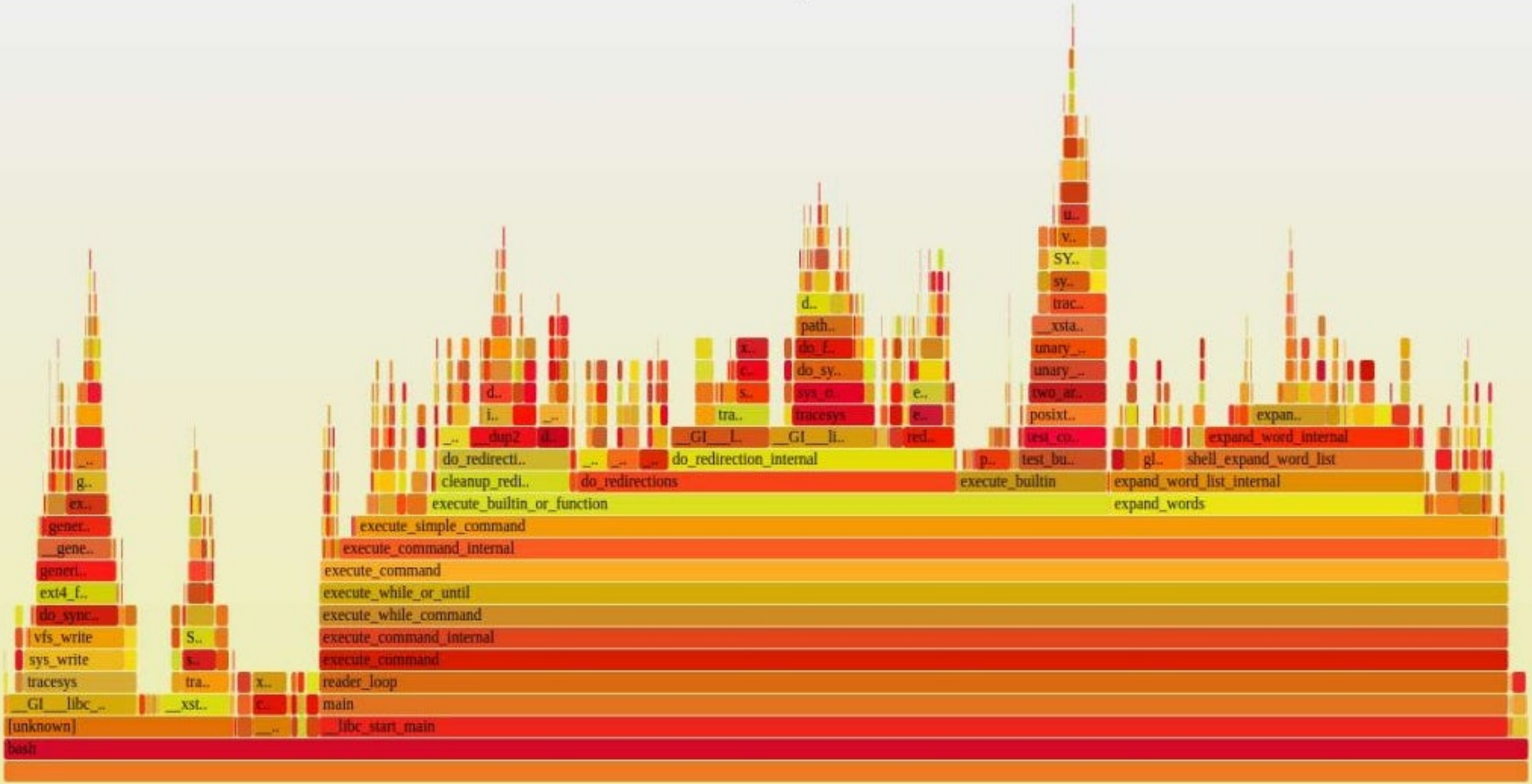
Выделения | Bottom-Down View | Collapse Recursion | Cost Threshold: 0.10%


1.12628e+6 allocations in total

1.12628e+6 allocations in total

# FlameGraph

- Визуализация стека чего-нибудь
- Не ограничена только CPU (Memory, IO)
- Интерактивно
- Кастомизируемо
- Хорошая поддержка профилировщиков
- https://github.com/brendangregg/FlameGraph (аккуратно - Perl)


Flame Graph
Search
u..
v..
SY..
sy..
trac..
__xsta..
unary_..
unary_..
two_ar..
posixt..
test_co..
p..
test_bu..
gl..
shell_expand_word_list
expand_word_internal
expan..
d..
path..
do_f..
do_sy..
sys_o..
tracesys
e..
e..
red..
__GI___l..
__GI___li..
x..
c..
s..
tra..
d..
i..
_..
_..
__dup2
d..
do_redirecti..
_..
_..
_..
do_redirection_internal
cleanup_redi..
do_redirections
execute_builtin
expand_word_list_internal
execute_builtin_or_function
expand_words
execute_simple_command
execute_command_internal
execute_command
execute_while_or_until
execute_while_command
execute_command_internal
execute_command
reader_loop
main
__libc_start_main
bash
_..
g..
ex..
gener..
__gene..
generi..
ext4_f..
do_sync..
vfs_write
sys_write
tracesys
__GI___libc_..
[unknown]
S..
s..
tra..
__xst..
__..
x..
c..

# Intel VTune Amplifier

- От Intel
- Проприетарный
- GNU/Linux, Windows, macOS (только как хост)
- Уже бесплатный для коммерческого использования
- GUI (Eclipse) + командная строка
- Только на CPU от Intel

# VTune: что умеет

- Hotspot analysis
- Concurrency profiling
- Cache profiling
- Branch-prediction profiling
- GPU profiling
- Disk IO profiling

Project Navigator

/home/zamazan4ik/intel/amplxe/proj...

- test
  - r000hs
  - r001mc
  - r002cc
  - r003hs
  - r004mc
  - r005hpc
  - r006ge
  - r007macc
  - r008cgc

Welcome | r004mc | r005hpc | r006ge

# General Exploration

General Exploration viewpoint (change)

INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Event Count | Platform

## Elapsed Time: 49.350s

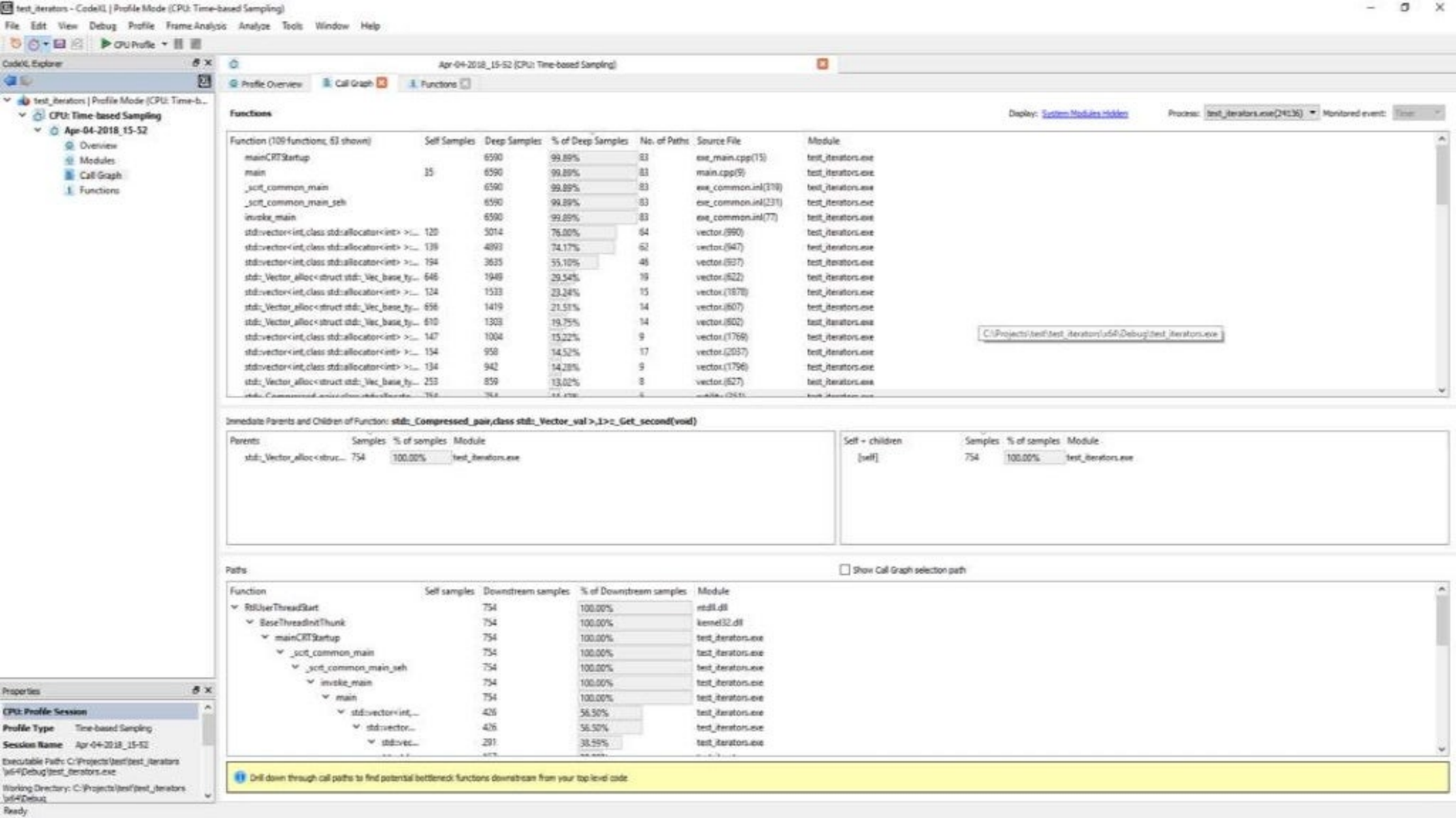
| | | |
|---|---|---|
| Clockticks: | 156,180,000,000 | |
| Instructions Retired: | 175,588,800,000 | |
| CPI Rate | 0.889 | |
| MUX Reliability | 0.995 | |
| Front-End Bound | 10.7% | of Pipeline Slots |
| Bad Speculation | 5.3% | of Pipeline Slots |
| Back-End Bound | 53.2% | of Pipeline Slots |
| Memory Bound | 36.7% | of Pipeline Slots |
| L1 Bound | 16.7% | of Clockticks |
| DTLB Overhead | 0.4% | of Clockticks |
| Loads Blocked by Store Forwarding | 0.3% | of Clockticks |
| Lock Latency | 0.0% | of Clockticks |
| Split Loads | 0.0% | of Clockticks |
| 4K Aliasing | 3.5% | of Clockticks |
| FB Full | 0.2% | of Clockticks |
| L2 Bound | 0.0% | of Clockticks |
| L3 Bound | 1.1% | of Clockticks |
| Contested Accesses | 0.0% | of Clockticks |
| Data Sharing | 0.0% | of Clockticks |
| L3 Latency | 0.2% | of Clockticks |
| SQ Full | 0.0% | of Clockticks |
| DRAM Bound | 0.0% | of Clockticks |
| Memory Bandwidth | 25.4% | of Clockticks |
| Memory Latency | 26.4% | of Clockticks |
| Store Bound | 42.7% | of Clockticks |
| Store Latency | 53.6% | of Clockticks |
| False Sharing | 0.0% | of Clockticks |
| Split Stores | 0.0% | of Clockticks |
| DTLB Store Overhead | 0.0% | of Clockticks |
| Core Bound | 16.5% | of Pipeline Slots |
| Retiring | 30.9% | of Pipeline Slots |
| Total Thread Count: | 2 | |
| Paused Time | 0s | |

## CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

# CodeXL

- От AMD
- Работает с CPU от AMD
- https://github.com/GPUOpen-Tools/CodeXL
- GNU/Linux, Windows
- Бывают баги

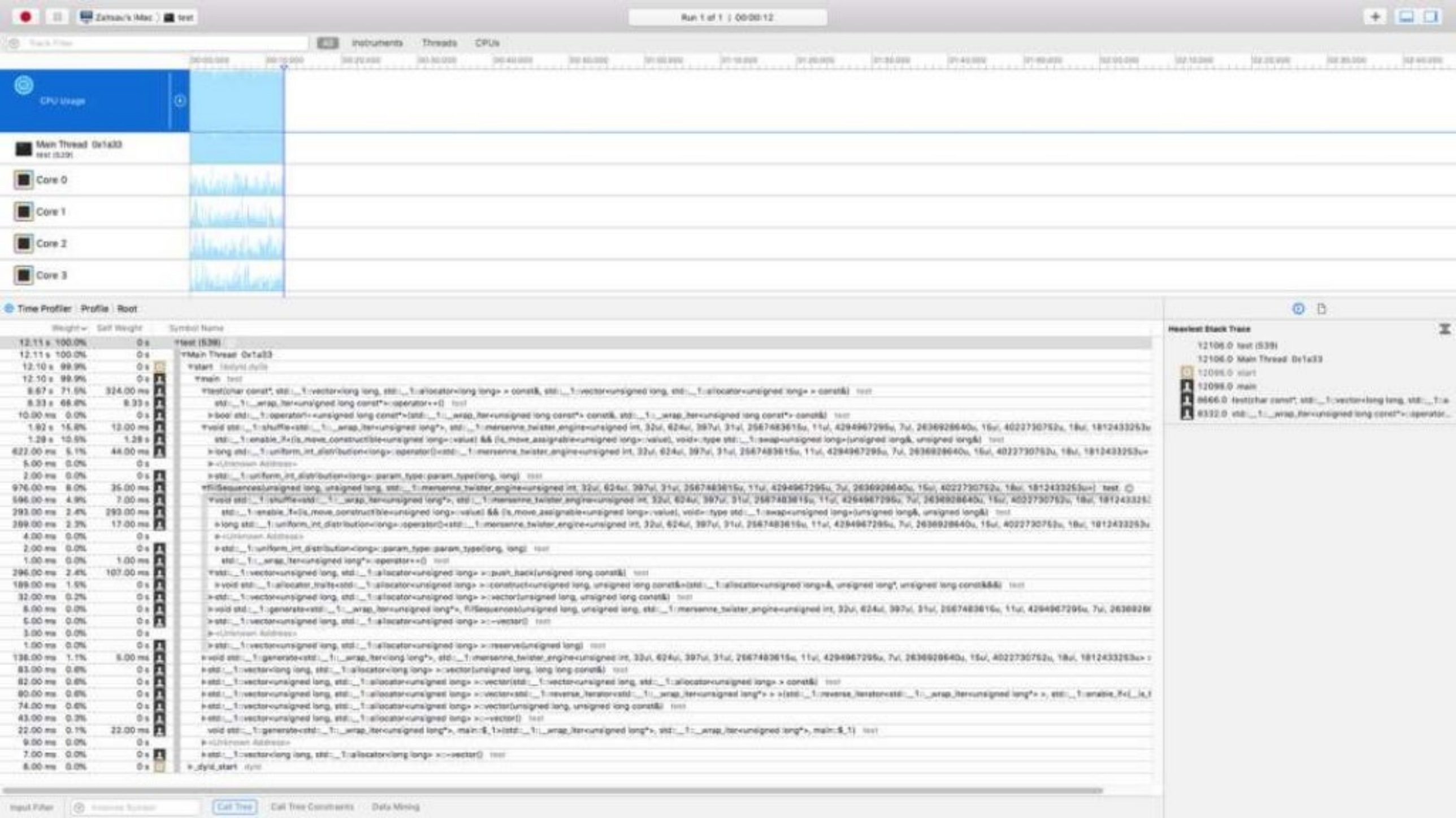

test_iterators - CodeXL | Profile Mode (CPU: Time-based Sampling)
File Edit View Debug Profile Frame Analysis Analyze Tools Window Help
CPU Profile
CodeXL Explorer
test_iterators | Profile Mode (CPU: Time-b...
CPU: Time-based Sampling
Apr-04-2018_15-52
Overview
Modules
Call Graph
Functions
Apr-04-2018_15-52 [CPU: Time-based Sampling]
Profile Overview
Call Graph
Functions
Functions
Display: System Modules Hidden
Process: test_iterators.exe(24136)
Monitored event:
Function (109 functions; 63 shown) Self Samples Deep Samples % of Deep Samples No. of Paths Source File Module
mainCRTStartup 6590 99.89% 83 exe_main.cpp(15) test_iterators.exe
main 35 6590 99.89% 83 main.cpp(9) test_iterators.exe
_scrt_common_main 6590 99.89% 83 exe_common.inl(319) test_iterators.exe
_scrt_common_main_seh 6590 99.89% 83 exe_common.inl(231) test_iterators.exe
invoke_main 6590 99.89% 83 exe_common.inl(77) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 120 5014 76.00% 64 vector.(990) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 139 4893 74.17% 62 vector.(947) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 194 3635 55.10% 46 vector.(937) test_iterators.exe
std::_Vector_alloc<struct std::_Vec_base_ty... 646 1949 29.54% 19 vector.(622) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 124 1533 23.24% 15 vector.(1870) test_iterators.exe
std::_Vector_alloc<struct std::_Vec_base_ty... 656 1419 21.51% 14 vector.(607) test_iterators.exe
std::_Vector_alloc<struct std::_Vec_base_ty... 610 1303 19.75% 14 vector.(602) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 147 1004 15.22% 9 vector.(1766) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 154 958 14.52% 17 vector.(2037) test_iterators.exe
std::vector<int,class std::allocator<int> >:... 134 942 14.28% 9 vector.(1796) test_iterators.exe
std::_Vector_alloc<struct std::_Vec_base_ty... 253 859 13.02% 8 vector.(627) test_iterators.exe
C:\Projects\test\test_iterators\x64\Debug\test_iterators.exe
Immediate Parents and Children of Function: std::_Compressed_pair,class std::_Vector_val >,1>::_Get_second(void)
Parents Samples % of samples Module
std::_Vector_alloc<struc... 754 100.00% test_iterators.exe
Self + children Samples % of samples Module
[self] 754 100.00% test_iterators.exe
Paths
Show Call Graph selection path
Function Self samples Downstream samples % of Downstream samples Module
RtlUserThreadStart 754 100.00% ntdll.dll
BaseThreadInitThunk 754 100.00% kernel32.dll
mainCRTStartup 754 100.00% test_iterators.exe
_scrt_common_main 754 100.00% test_iterators.exe
_scrt_common_main_seh 754 100.00% test_iterators.exe
invoke_main 754 100.00% test_iterators.exe
main 754 100.00% test_iterators.exe
std::vector<int,... 426 56.50% test_iterators.exe
std::vector... 426 56.50% test_iterators.exe
std::vec... 291 38.59% test_iterators.exe
Drill down through call paths to find potential bottleneck functions downstream from your top level code
Properties
CPU: Profile Session
Profile Type Time-based Sampling
Session Name Apr-04-2018_15-52
Executable Path: C:\Projects\test\test_iterators\x64\Debug\test_iterators.exe
Working Directory: C:\Projects\test\test_iterators\x64\Debug
Ready

# CodeXL: функционал

- Cache line utilization
- Instruction-based sampling
- Branch profiling
- Data access profiling
- Time-based sampling
- Instruction access
- Call graph
- Power profiling
- GPU profiling

# Xcode + Instruments

- Идёт в комплекте с Xcode
- В ряде случаев использует “под капотом” DTrace
- Умеет профилировать приложения на macOS, iOS
- Time profiler, memory profiler
- Wi-Fi, GPS, Energy, etc. (iOS)


Run 1 of 1 | 00:00:12
All
Instruments
Threads
CPUs
CPU Usage
Main Thread 0x1a33
Core 0
Core 1
Core 2
Core 3
Time Profiler
Profile
Root
Weight
Self Weight
Symbol Name
12.11 s 100.0% 0 s test (539)
12.11 s 100.0% 0 s Main Thread 0x1a33
12.10 s 99.9% 0 s start
12.10 s 99.9% 0 s main
8.67 s 71.5% 324.00 ms test(char const*, std::__1::vector<long long, std::__1::allocator<long long> > const&, std::__1::vector<unsigned long, std::__1::allocator<unsigned long> > const&)
8.33 s 68.8% 8.33 s std::__1::__wrap_iter<unsigned long const*>::operator++()
10.00 ms 0.0% 0 s bool std::__1::operator!=<unsigned long const*>(std::__1::__wrap_iter<unsigned long const*> const&, std::__1::__wrap_iter<unsigned long const*> const&)
1.92 s 15.8% 12.00 ms void std::__1::shuffle<std::__1::__wrap_iter<unsigned long*>, std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928640u, 15ul, 4022730752u, 18ul, 1812433253u
1.28 s 10.5% 1.28 s std::__1::enable_if<(is_move_constructible<unsigned long>::value) && (is_move_assignable<unsigned long>::value), void>::type std::__1::swap<unsigned long>(unsigned long&, unsigned long&)
622.00 ms 5.1% 44.00 ms long std::__1::uniform_int_distribution<long>::operator()<std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928640u, 15ul, 4022730752u, 18ul, 1812433253u>
5.00 ms 0.0% 0 s <Unknown Address>
2.00 ms 0.0% 0 s std::__1::uniform_int_distribution<long>::param_type::param_type(long, long)
976.00 ms 8.0% 35.00 ms fillSequence(unsigned long, unsigned long, std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928640u, 15ul, 4022730752u, 18ul, 1812433253u>) test
596.00 ms 4.9% 7.00 ms void std::__1::shuffle<std::__1::__wrap_iter<unsigned long*>, std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928640u, 15ul, 4022730752u, 18ul, 1812433253
293.00 ms 2.4% 293.00 ms std::__1::enable_if<(is_move_constructible<unsigned long>::value) && (is_move_assignable<unsigned long>::value), void>::type std::__1::swap<unsigned long>(unsigned long&, unsigned long&)
289.00 ms 2.3% 17.00 ms long std::__1::uniform_int_distribution<long>::operator()<std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928640u, 15ul, 4022730752u, 18ul, 1812433253u
4.00 ms 0.0% 0 s <Unknown Address>
2.00 ms 0.0% 0 s std::__1::uniform_int_distribution<long>::param_type::param_type(long, long)
1.00 ms 0.0% 1.00 ms std::__1::__wrap_iter<unsigned long*>::operator++()
296.00 ms 2.4% 107.00 ms std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::push_back(unsigned long const&)
189.00 ms 1.5% 0 s void std::__1::allocator_traits<std::__1::allocator<unsigned long> >::construct<unsigned long, unsigned long const&>(std::__1::allocator<unsigned long>&, unsigned long*, unsigned long const&&)
32.00 ms 0.2% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::vector(unsigned long, unsigned long const&)
8.00 ms 0.0% 0 s void std::__1::generate<std::__1::__wrap_iter<unsigned long*>, fillSequence(unsigned long, unsigned long, std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928
5.00 ms 0.0% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::~vector()
3.00 ms 0.0% 0 s <Unknown Address>
1.00 ms 0.0% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::reserve(unsigned long)
138.00 ms 1.1% 5.00 ms void std::__1::generate<std::__1::__wrap_iter<long long*>, std::__1::mersenne_twister_engine<unsigned int, 32ul, 624ul, 397ul, 31ul, 2567483615u, 11ul, 4294967295u, 7ul, 2636928640u, 15ul, 4022730752u, 18ul, 1812433253u>
83.00 ms 0.6% 0 s std::__1::vector<long long, std::__1::allocator<long long> >::vector(unsigned long, long long const&)
82.00 ms 0.6% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::vector(std::__1::vector<unsigned long, std::__1::allocator<unsigned long> > const&)
80.00 ms 0.6% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::vector<std::__1::reverse_iterator<std::__1::__wrap_iter<unsigned long*> > >(std::__1::reverse_iterator<std::__1::__wrap_iter<unsigned long*> >, std::__1::enable_if<(__is_f
74.00 ms 0.6% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::vector(unsigned long, unsigned long const&)
43.00 ms 0.3% 0 s std::__1::vector<unsigned long, std::__1::allocator<unsigned long> >::~vector()
22.00 ms 0.1% 22.00 ms void std::__1::generate<std::__1::__wrap_iter<unsigned long*>, main::$_1>(std::__1::__wrap_iter<unsigned long*>, std::__1::__wrap_iter<unsigned long*>, main::$_1)
9.00 ms 0.0% 0 s <Unknown Address>
7.00 ms 0.0% 0 s std::__1::vector<long long, std::__1::allocator<long long> >::~vector()
8.00 ms 0.0% 0 s dyld_start
Heaviest Stack Trace
12106.0 test (539)
12106.0 Main Thread 0x1a33
12098.0 start
12098.0 main
8666.0 test(char const*, std::__1::vector<long long, std::__1::a
6332.0 std::__1::__wrap_iter<unsigned long const*>::operator
Input Filter
Call Tree
Call Tree Constraints
Data Mining

# SimplePerf (+ Android Studio 3.1)

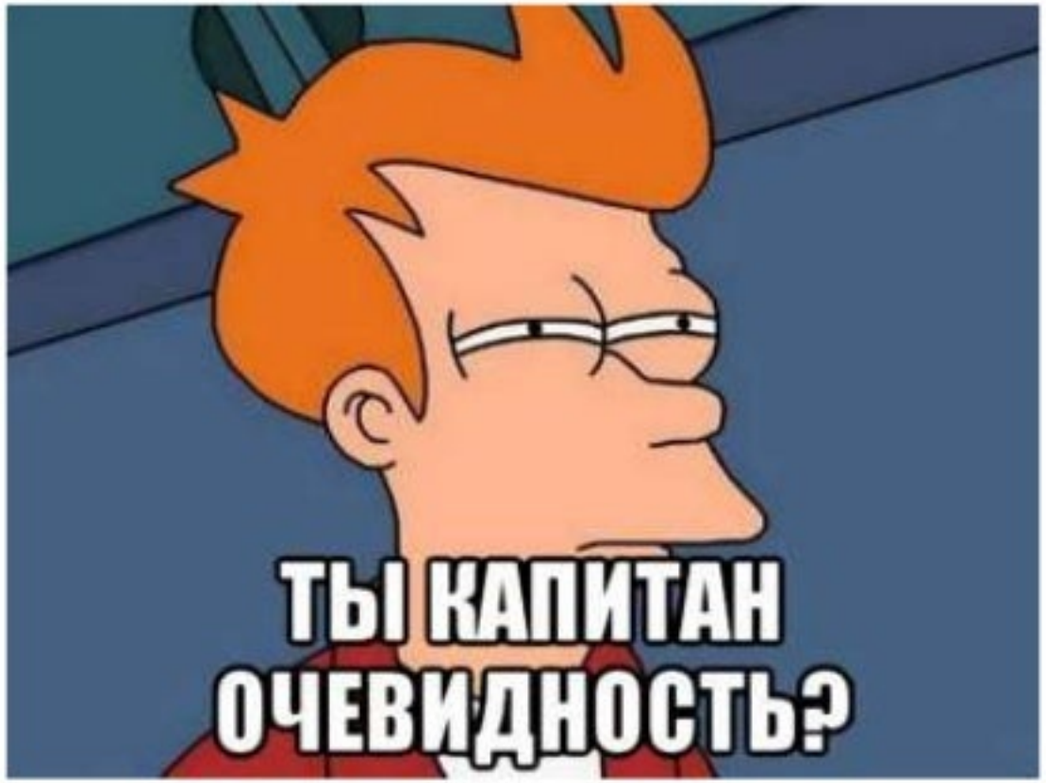
- Профайлер для Android
- Консольный интерфейс
- https://android.googlesource.com/platform/system/extras/+/master/simpleperf
- Начиная с Android 3.1 может профилировать C++ код прямо из IDE
  - Требует Android 8 (API Level 26) и выше

# Остались за рамками доклада (извините)

- OProfile - http://oprofile.sourceforge.net
- TAU - https://www.cs.uoregon.edu/research/tau/home.php
- LTTng - https://lttng.org/
- xperf - https://docs.microsoft.com/en-us/windows-hardware/test/wpt/
- GlowCode - https://www.glowcode.com/
- DS-5 (built-in profiler) - https://developer.arm.com/
- Visual Studio (built-in profiler)
- Oracle Developer Studio (built-in profiler)
- Intel IACA
- DTrace - <link>

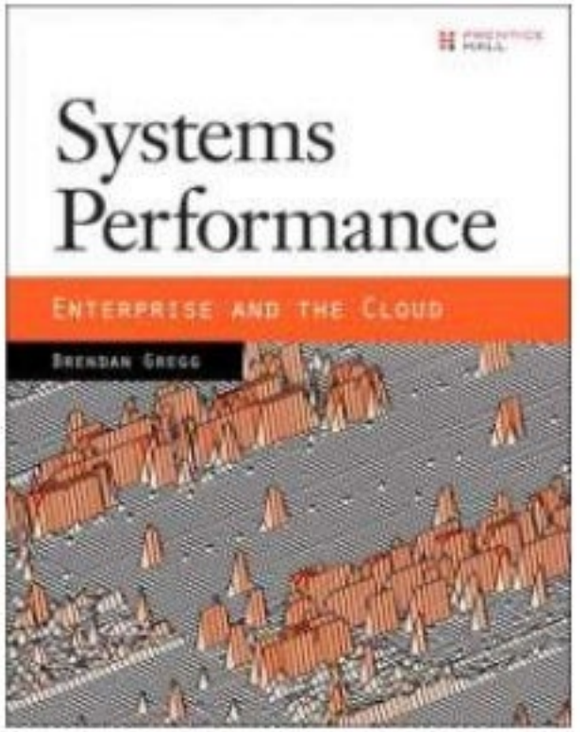
# Советы

- Не оптимизируйте без профилирования
- Выбирайте профайлер внимательно
- Изучите профайлер перед использованием



# Полезные ссылки

- https://eax.me/c-cpp-profiling
- https://github.com/fenbf/AwesomePerfCpp
- https://software.intel.com/en-us/vtune-amplifier-help
- http://www.brendangregg.com



Спасибо за внимание

Вопросы?