

Python profiler

Oleg Shipulin

16 февраля 2026 г.

Содержание

1	Введение	3
1.1	Мотивация	3
1.2	Объект и предмет исследования	3
1.3	Цель исследования	3
2	Анализ и визуализация результатов профилирования	5
2.1	Форматы вывода профилировщиков	5
2.2	Модуль pstats: базовый анализ в консоли	5
2.3	Интерактивная визуализация с SnakeViz	7
2.4	Графы вызовов с gprof2dot и Graphviz	10
2.5	Профессиональные инструменты: KCachegrind / QKachegrind	12
2.6	Сравнение инструментов визуализации	14
3	Профилирование	15
3.1	Виды профилирования	15
3.2	Основные методы профилирования	15
3.3	Инструментирование (tracing)	15
3.4	Статистическое (sampling)	16
3.5	Сравнение методов	16
4	Профилирование на уровне интерпретатора Python	17
4.1	Инструментирующие профилировщики	17
4.1.1	Profile (Legacy)	17
4.1.2	cProfile	18
4.1.3	line_profiler	22
4.2	Статистические профилировщики	24
4.2.1	py-spy	24
4.2.2	pprofile	27
4.3	Специализированные решения	29
4.3.1	yappi	29
5	Профилирование нативных расширений (C extensions)	32
5.1	Особенности профилирование кода на C/C++ внутри Python	32
5.2	Инструменты для анализа C/C++: gprof, Valgrind (Callgrind), Google PerfTools	32
5.3	Получение смешанных стеков вызовов (Python + C) с помощью perf и eBPF	32
6	Системное профилирование Python-приложений	32
6.1	Использование утилиты perf в Linux для анализа на уровне ядра	32
6.2	DTrace и SystemTap: динамическая трассировка	32
6.3	Профилирование операций ввода-вывода и работы с сетью	32
7	Сравнительный анализ и практические рекомендации	32

1 Введение

1.1 Мотивация

Так как Python сам по себе не самый быстрый язык программирования, то по мере роста проекта, усложнения его архитектуры и увеличения объемов обрабатываемых данных, даже незначительные куски неэффективного кода могут накапливать задержки, превращая программу в невероятно медленную. Пример такого кода - применение не самой оптимизированной структуры данных для конкретного сценария.

Найти bottleneck (узкое горлышко) и не самого оптимизированного кода помогает профилирование, о котором и пойдёт речь в последующей статье.

1.2 Объект и предмет исследования

Объектом исследования являются программные средства динамического анализа производительности кода на языке Python, включая как встроенные и сторонние профилировщики, так и системные утилиты, позволяющие исследовать выполнение Python-программ на различных уровнях абстракции: от интерпретируемого кода до нативных расширений и системных вызовов.

Предметом исследования выступают методы и техники профилирования, реализованные в этих средствах: инструментирование (tracing), статистическое профилирование (sampling), построчный анализ, профилирование памяти, а также подходы к анализу производительности на уровне операционной системы (perf, Valgrind) и их применимость к Python-приложениям.

1.3 Цель исследования

Цель работы — провести комплексный сравнительный анализ инструментов профилирования Python, охватывающий все уровни исполнения программы: от исходного кода на Python до нативных расширений и взаимодействия с ядром ОС. На основе этого анализа выявить сильные и слабые стороны каждого класса инструментов, определить области их эффективного применения и предложить рекомендации по выбору средств для решения конкретных задач оптимизации производительности.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Рассмотреть теоретические основы профилирования: определить понятие профилирования, классифицировать существующие методы (инструментирование и статистическое профилирование), описать их достоинства и недостатки.
2. Выполнить обзор профилировщиков, входящих в стандартную библиотеку Python (profile, cProfile), и охарактеризовать особенности их реализации, достоинства и ограничения.
3. Изучить специализированные профилировщики, ориентированные на построчный анализ (line_profiler), анализ памяти (memory_profiler), работу с многопоточными и асинхронными приложениями (yappi).
4. Рассмотреть инструменты статистического профилирования (py-spy, pprofile), позволяющие выполнять анализ без остановки программы и с минимальным вмешательством в её работу.

5. Проанализировать возможности профилирования нативных расширений Python (C extensions) с использованием инструментов для отладки и профилирования на уровне языков C/C++ (gprof, Valgrind, perf) и их интеграцию с Python-процессами.
6. Исследовать системные профилировщики (perf, SystemTap, DTrace) и методы их применения для анализа Python-приложений, включая получение смешанных стеков вызовов (Python + C).
7. Сравнить перечисленные средства по ключевым критериям: скорость работы, точность измерений, область применения, накладываемые ограничения и требования к окружению.
8. На основе проведённого анализа сформулировать практические рекомендации по выбору инструментов профилирования для различных сценариев разработки и эксплуатации Python-программ.

2 Анализ и визуализация результатов профилирования

После сбора данных профилирования перед разработчиком встаёт задача интерпретации полученных метрик. Разные профилировщики используют различные форматы вывода, однако существуют универсальные инструменты, позволяющие анализировать и визуализировать статистику вне зависимости от того, каким именно профилировщиком она была собрана. В этой главе рассматриваются основные форматы хранения профилей, способы их обработки с помощью стандартного модуля `pstats`, а также популярные инструменты визуализации: `snakeviz`, `gprof2dot` и `KCachegrind`.

2.1 Форматы вывода профилировщиков

Профилировщики Python могут сохранять результаты в нескольких форматах:

- **Текстовый** — простой для чтения человеком, но неудобный для автоматической обработки. Используется при вызове `print_stats()` модуля `pstats` или при запуске профилировщика без указания выходного файла.
- **Бинарный формат pstats** — внутренний формат модуля `pstats`, используемый `profile` и `cProfile` по умолчанию. Файлы обычно имеют расширение `.prof` или `.pstats`.
- **Callgrind-совместимый формат** — формат, используемый инструментом `callgrind` из состава Valgrind. Поддерживается профилировщиком `pprofile` и может быть создан из `pstats` с помощью утилиты `pyprof2calltree`.
- **JSON и другие** — некоторые современные инструменты, такие как `py-spy`, могут экспортировать данные в JSON для последующей обработки.

В данной главе основное внимание уделяется работе с форматом `pstats`, как наиболее распространённому среди встроенных средств Python.

2.2 Модуль pstats: базовый анализ в консоли

Модуль `pstats` входит в стандартную библиотеку Python и предоставляет набор инструментов для сортировки, фильтрации и вывода статистики, собранной профилировщиками `profile` и `cProfile`. Работа с ним может осуществляться как в интерактивном режиме, так и скриптово.

Загрузка статистики

Для начала работы необходимо создать объект `Stats`, передав ему имя файла с профилем:

```
1 import pstats
2 stats = pstats.Stats('output.prof')
```

Можно загрузить несколько файлов сразу, перечислив их в конструкторе.

Очистка путей

Часто полные абсолютные пути к файлам загромождают вывод. Метод `strip_dirs()` удаляет пути к каталогам, оставляя только имена файлов:

```
1 stats.strip_dirs()
```

Сортировка

Перед выводом статистику необходимо отсортировать по одному из ключей. Наиболее употребимые:

- `'cumtime'` — суммарное время функции с учётом дочерних вызовов;
- `'tottime'` — собственное время функции (без учёта вызовов);
- `'ncalls'` — количество вызовов;
- `'pcalls'` — количество примитивных вызовов.

Пример сортировки по собственному времени:

```
1 stats.sort_stats('tottime')
```

Вывод

Метод `print_stats(n)` выводит первые `n` записей:

```
1 stats.print_stats(10)
```

Каждая строка вывода содержит:

- `ncalls` — общее число вызовов (в скобках может указываться число рекурсивных вызовов);
- `tottime` — суммарное время, проведённое непосредственно в теле функции (без вызовов);
- `percall` — `tottime`, делённое на `ncalls`;
- `cumtime` — суммарное время, проведённое в функции и во всех вызванных ею функциях;
- `percall` — `cumtime`, делённое на количество примитивных вызовов;
- `filename:lineno(function)` — имя файла, номер строки и имя функции.

Просмотр вызывающих и вызываемых функций

Методы `print_callers()` и `print_callees()` показывают, какие функции вызывают данную и какие вызывает она сама. Это помогает понять структуру взаимодействия:

```
1 stats.print_callers(5)
2 stats.print_callees(5)
```

Фильтрация

Модуль `pstats` позволяет фильтровать вывод по имени файла или функции с помощью метода `print_stats()` с регулярными выражениями. Например, чтобы оставить только функции из файла `mycode.py`:

```
1 stats.print_stats('mycode.py')
```

2.3 Интерактивная визуализация с SnakeViz

SnakeViz — это браузерный инструмент для интерактивного исследования профилей `cProfile`. Он предоставляет два основных вида визуализации: «солнечные лучи» (`sunburst`) и иерархический список (`icicle`).

Установка

```
1 pip install snakeviz
```

Запуск

Для просмотра профиля достаточно выполнить:

```
1 snakeviz output.prof
```

После этого откроется окно браузера с интерактивной диаграммой.

Интерфейс и навигация в SnakeViz

После запуска команды `snakeviz output.prof` в браузере открывается страница с интерактивной визуализацией. По умолчанию отображается диаграмма в стиле «солнечные лучи» (`sunburst`). В левом верхнем углу расположена панель управления, позволяющая:

- **Переключаться между режимами отображения:** `sunburst` (рис. 1) и `icicle` (рис. 2). Режим `icicle` представляет иерархию в виде горизонтальных полос, где каждая полоса соответствует уровню вызовов, а ширина полосы пропорциональна времени.
- **Фильтровать по имени функции или модуля:** в поле поиска можно ввести регулярное выражение, чтобы оставить только те функции, которые соответствуют шаблону. Это особенно полезно для выделения кода конкретного модуля.
- **Настраивать цветовую схему:** по умолчанию цвета назначаются случайным образом, но можно включить окраску по модулям, что упрощает идентификацию различных частей программы.
- **Сортировать дочерние функции** по времени или имени.
- **Экспортировать текущий вид** как изображение (PNG) через стандартные средства браузера (правый клик → «Сохранить картинку как...»).

Интерпретация sunburst-диаграммы

На рис. 1 представлен пример sunburst-диаграммы. Центральный круг — корневая функция (обычно `<module>`). Каждое следующее кольцо соответствует уровню вложенности вызовов. Сегменты внутри кольца представляют функции, вызванные на этом уровне. Угловой размер сегмента пропорционален суммарному времени, проведённому в данной функции и всех её потомках (cumulative time). Таким образом, крупные сегменты сразу указывают на горячие участки.

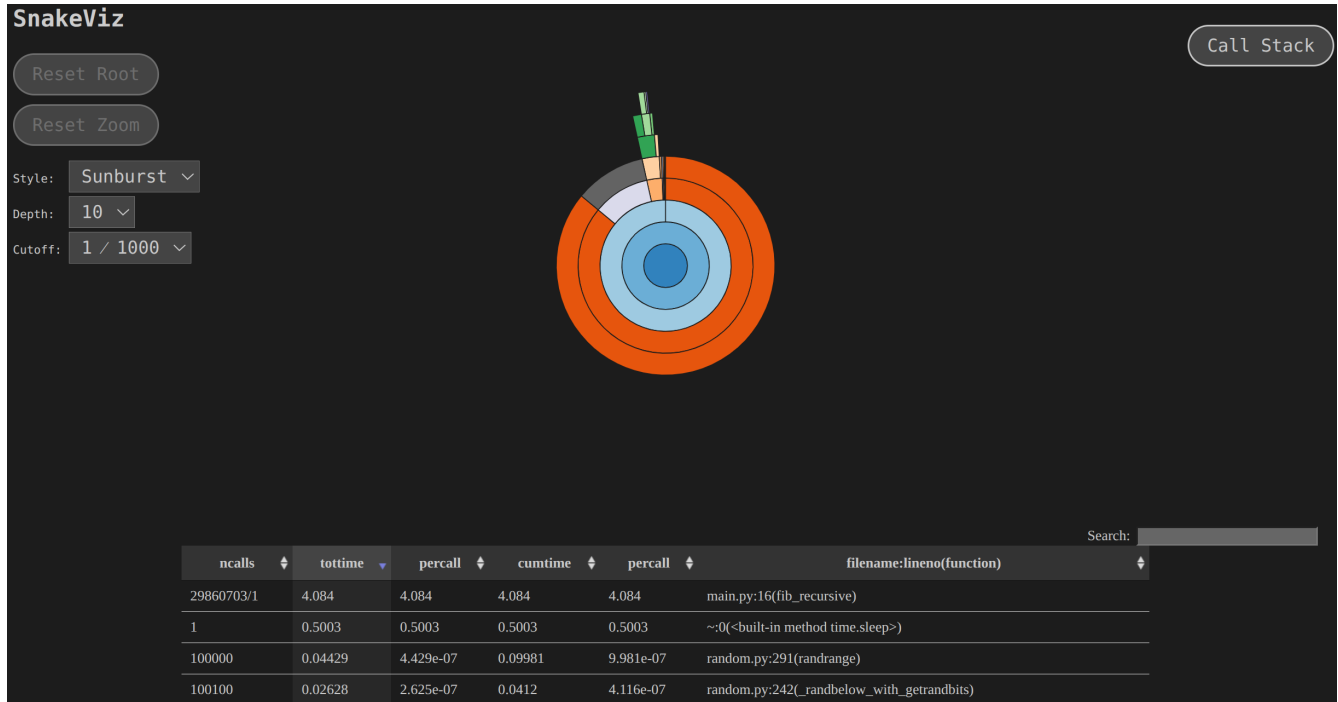


Рис. 1: Sunburst-диаграмма в SnakeViz

При наведении курсора на любой сегмент всплывает подсказка с подробной информацией:

- имя функции и модуль;
- собственное время (tottime) — время, потраченное непосредственно внутри функции (в секундах и процентах);
- суммарное время (cumtime) — включая все вызовы;
- количество вызовов (ncalls).

Цветовая дифференциация помогает различать разные модули или файлы, если включена соответствующая опция.

Интерпретация icicle-диаграммы

Режим icicle (рис. 2) представляет иерархию в виде горизонтальных линий. Верхний уровень — корневая функция, под ним располагаются вызванные функции и т.д. Длина каждой полосы пропорциональна суммарному времени. Этот вид удобен для сравнения долей времени на одном уровне иерархии, а также для отслеживания цепочек вызовов без радиальных искажений.

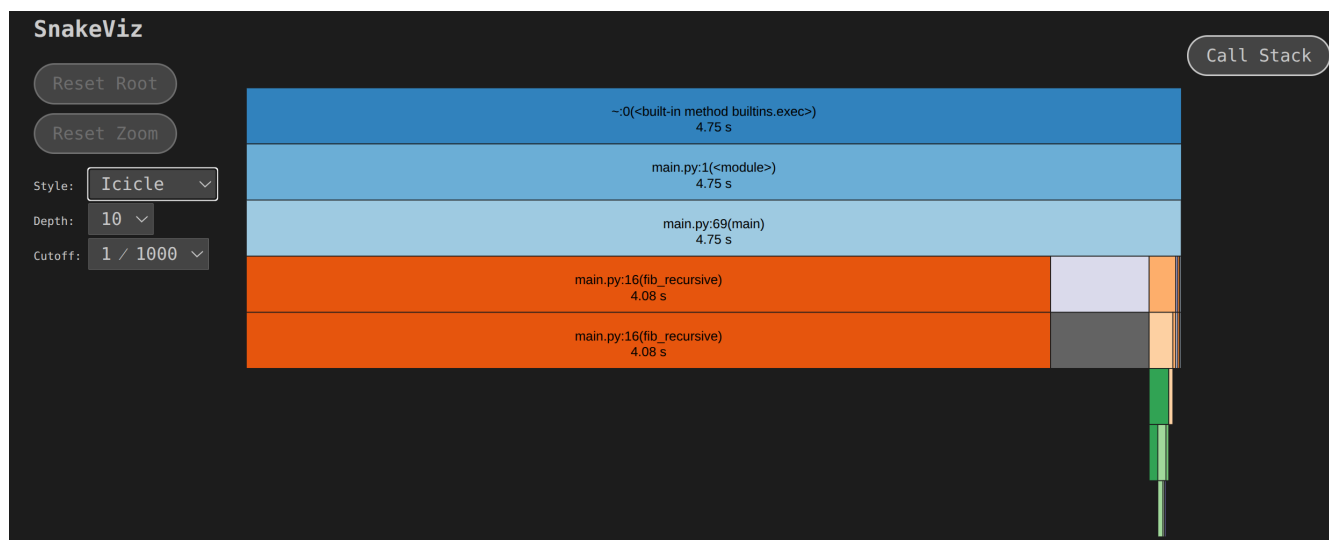


Рис. 2: Icicle-диаграмма в SnakeViz

Что можно понять с помощью SnakeViz

SnakeViz позволяет быстро ответить на ключевые вопросы оптимизации:

- **Какие функции потребляют больше всего времени?** — крупные сегменты в центре или длинные полосы сразу бросаются в глаза.
- **Как распределяется время внутри функции?** — можно увидеть, сколько времени уходит на дочерние вызовы, а сколько — на собственный код.
- **Есть ли рекурсивные или циклические вызовы?** — они проявляются как повторяющиеся сегменты на разных уровнях.
- **Какие модули вносят основной вклад?** — при окраске по модулям легко выделить проблемные библиотеки.
- **Насколько оправданы вызовы вспомогательных функций?** — если мелкая функция вызывается огромное число раз и занимает значительное время, это повод задуматься об оптимизации или кешировании.

SnakeViz не даёт абсолютно точных цифр (для этого лучше использовать `pstats`), но предоставляет интуитивно понятную картину, с которой удобно начинать анализ. Выявив крупные сегменты, разработчик может затем детально исследовать соответствующие функции с помощью `line_profiler` или других средств.

Дополнительные возможности

SnakeViz поддерживает загрузку нескольких профилей одновременно (например, для сравнения до и после оптимизации), а также позволяет сохранять текущее состояние фильтров в URL для передачи коллегам. Документация по проекту доступна на официальном сайте <https://jiffyclub.github.io/snakeviz/>.

2.4 Графы вызовов с gprof2dot и Graphviz

Утилита `gprof2dot` преобразует профили множества форматов (включая `pstats`) в граф в формате DOT, который затем визуализируется программой `dot` из пакета Graphviz.

Установка

```
1 pip install gprof2dot
```

Для работы также необходимо установить Graphviz (доступен через системные менеджеры пакетов: `apt install graphviz`, `brew install graphviz` и т.д.).

Построение графа

Базовая команда для создания PNG-изображения:

```
1 python -m gprof2dot -f pstats output.prof | dot -Tpng -o callgraph.png
```

Ключ `-f pstats` указывает формат входных данных. Дополнительные опции `gprof2dot` позволяют настраивать отображение:

- `-node-thres 0.5` — не показывать узлы, занимающие менее 0.5% времени;
- `-edge-thres 0.1` — не показывать рёбра, по которым передаётся менее 0.1% времени;
- `-color-nodes-by-self` — окрашивать узлы по собственному времени (по умолчанию — по суммарному).

Полученный граф наглядно демонстрирует «горячие» функции и пути вызовов.

Анализ графа вызовов

На рис. 3 представлен типичный граф вызовов, построенный утилитой `gprof2dot` по данным профилирования. Граф состоит из узлов (прямоугольников), соответствующих функциям, и направленных рёбер, показывающих, какая функция какую вызывает. Каждый узел содержит следующую информацию:

- имя функции и модуль (например, `main:16.fib_recursive`);
- общее время, проведённое в функции и её потомках, в процентах от общего времени выполнения программы (верхнее число в узле);
- собственное время функции (без учёта дочерних вызовов) в процентах (число в скобках);
- количество вызовов данной функции (например, `29860703x`).

Рёбра подписаны процентами, показывающими, какая доля времени вызывающей функции была передана вызываемой (или сколько времени заняли вызовы).

В приведённом примере можно сделать следующие наблюдения:

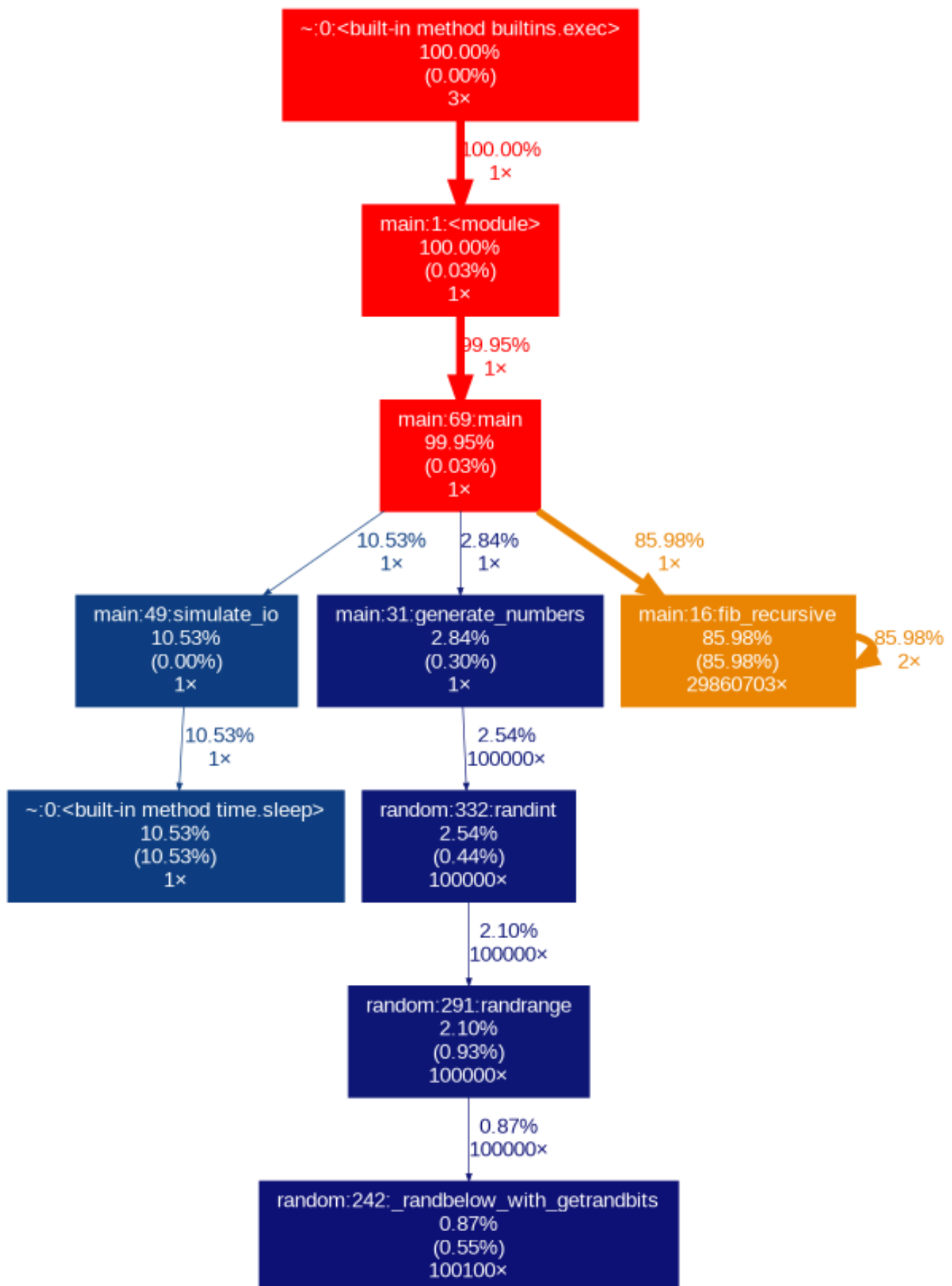


Рис. 3: callgraph

- Основным потребителем времени является рекурсивная функция `fib_recursive`, на которую приходится 85.98% общего времени и столько же собственного (85.98%). Это говорит о том, что почти всё время программа проводит непосредственно в этой функции, а не во вспомогательных вызовах. Количество вызовов (почти 30 миллионов) объясняет такое поведение: функция чрезвычайно неэффективна из-за экспоненциального роста числа рекурсивных вызовов.
- Функция `simulate_io` занимает 10.53% общего времени, но её собственное время близко к нулю (0.00%), поскольку почти всё время уходит на вызов встроенной функции `time.sleep`. Это видно по ребру с надписью 10.53%, ведущему к узлу `<built-in method time.sleep>`, который имеет 10.53% собственного времени. Таким образом, здесь мы имеем дело не с вычислительной нагрузкой, а с ожиданием ввода-вывода.
- Функции генерации случайных чисел (`randint`, `randrange`, `_randbelow_with_getrandbits`) в сумме занимают около 2–3% времени и вызываются 100+000 раз, что соответствует ожидаемому поведению.
- Корневая функция `<module>` и `main` имеют общее время 100% и 99.95% соответственно, что естественно, так как они являются точками входа.

Граф вызовов позволяет быстро идентифицировать горячие пути в программе. Красные/оранжевые узлы обычно соответствуют функциям с наибольшей долей времени (в зависимости от цветовой схемы). Анализируя структуру графа, можно:

- Найти функции, которые вызываются аномально часто (как `fib_recursive`) — это сигнал к оптимизации алгоритма или введению кеширования.
- Отличить вычислительные узкие места от операций ввода-вывода (как `simulate_io`): если собственное время функции мало, а общее велико за счёт вызовов системных или библиотечных функций, проблема может быть в излишних ожиданиях или неэффективных внешних вызовах.
- Проследить цепочки вызовов, ведущие к наиболее затратным функциям, чтобы понять контекст их использования.
- Оценить, насколько равномерно распределено время между различными модулями.

Таким образом, граф вызовов служит мощным инструментом первичной диагностики, позволяя с одного взгляда определить, куда направить дальнейшие усилия по оптимизации.

2.5 Профессиональные инструменты: KCachegrind / QKachegrind

KCachegrind (и его аналог для macOS/Windows **QKachegrind**) — мощный визуализатор профилей, изначально созданный для работы с данными Valgrind. Он поддерживает иерархическое отображение времени, графы вызовов, аннотированный исходный код и множество статистических сводок.

Конвертация профиля в формат callgrind

Для использования pstats-файла в KCachegrind необходимо преобразовать его с помощью утилиты pyprof2calltree:

```
1 pip install pyprof2calltree
2 pyprof2calltree -i output.prof -o callgrind.out
```

Затем полученный файл открывается в KCachegrind:

```
1 kcachegrind callgrind.out
```

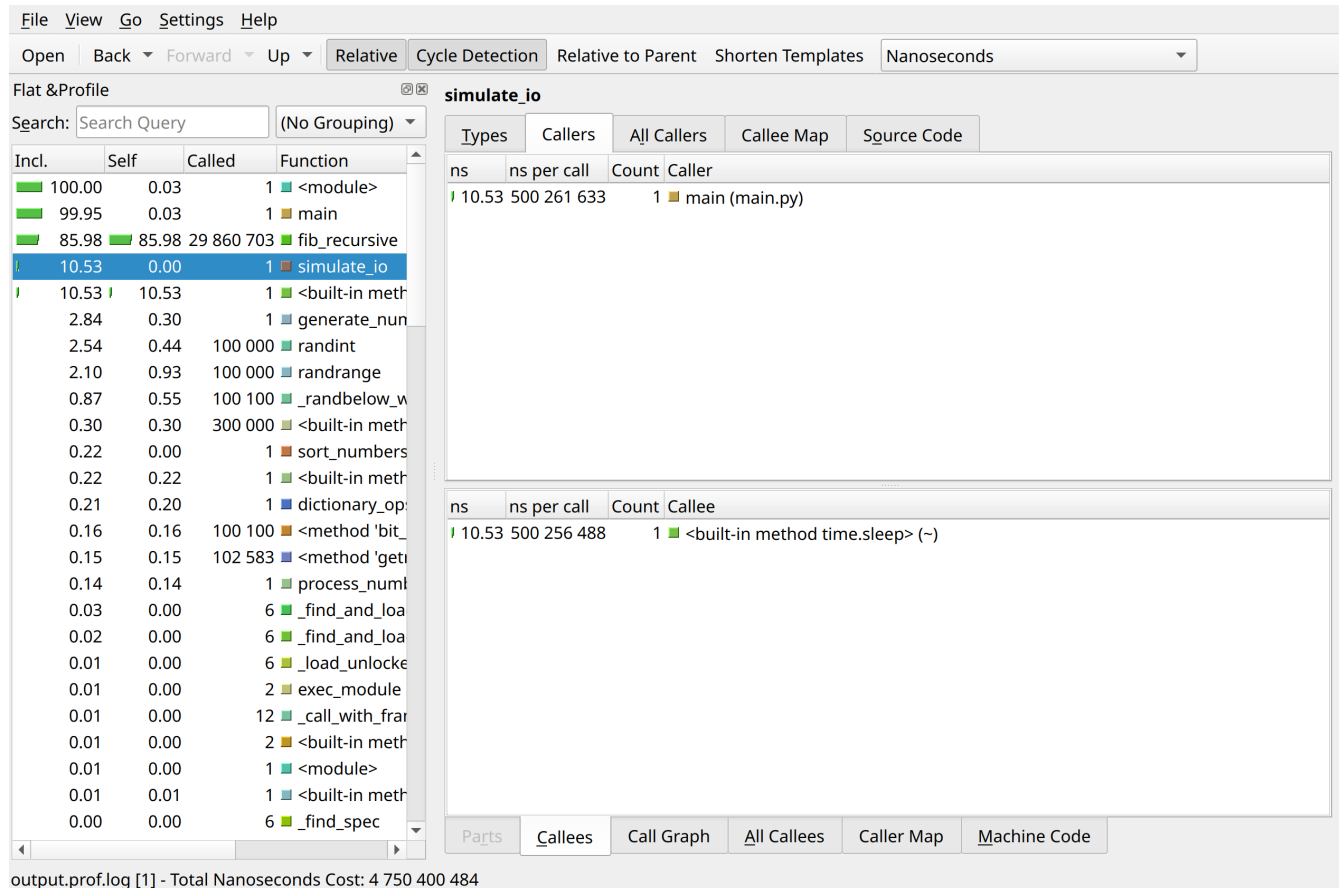


Рис. 4: KCachegrind

Возможности KCachegrind

Программа предоставляет:

- Древовидную карту вызовов (Call Graph) с цветовой индикацией времени;
- Таблицы функций с сортировкой по разным метрикам;
- Просмотр исходного кода с аннотацией времени по строкам;
- Статистику по вызывающим/вызываемым функциям;

- Экспорт в различные форматы.

KCachegrind особенно полезен при глубоком анализе больших проектов, где требуется одновременное изучение множества взаимосвязей.

2.6 Сравнение инструментов визуализации

Выбор инструмента зависит от конкретной задачи:

- **pstats** — незаменим для быстрого консольного анализа, автоматизации в скриптах и интеграции в тесты.
- **SnakeViz** — идеален для интерактивного исследования: позволяет «пройти» по иерархии вызовов, быстро выявить основные потребители времени.
- **gprof2dot** — лучший выбор для создания статичных графов, которые можно встраивать в отчёты или презентации.
- **KCachegrind** — профессиональный инструмент для всестороннего анализа, особенно когда требуется сопоставление с исходным кодом и работа с большими объёмами данных.

Комбинируя эти инструменты, разработчик может эффективно исследовать производительность программы на любом уровне детализации.

3 Профилирование

3.1 Виды профилирования

Профилирование. Определение профилирования везде разное, но в общем смысле *профилирование* - процесс динамического анализа работы кода, направленный на сбор метрик её выполнения. К этим метрикам можно отнести:

- Время выполнения функций, методов, строк;
- Использование оперативной памяти, а именно выделение, освобождение, утечки;
- Частота и количество вызовов функций;
- Загрузка спу;
- Сколько занимает ввод-вывод.

Полученные данные позволяют выявить узкие места (*bottlenecks*) и обоснованно провести оптимизацию.

3.2 Основные методы профилирования

По способу сбора информации профилировщики делятся на два принципиально разных класса:

1. **Инструментирование** (*tracing* или *instrumentation*).
2. **Статистическое профилирование** (*sampling*).

3.3 Инструментирование (tracing)

Инструментирование — метод динамического анализа, при котором в исходный или бинарный код программы автоматически встраиваются специальные счетчики (зонды) для точного измерения производительности: времени выполнения функций, количества вызовов и потребления памяти.

Достоинства:

- Высокая точность — фиксируется каждый вызов и точное время его выполнения
- Собирается исчерпывающая статистика (количество вызовов, длительность, иерархия).

Недостатки:

- Значительное замедление работы программы (от 2–3 до 10–100 раз в зависимости от языка и реализации)
- Сам факт внедрения зондов может исказить реальную производительность, особенно для короткоживущих функций

3.4 Статистическое (sampling)

Статистическое профилирование - метод анализа производительности ПО, при котором профилировщик через заданные интервалы времени делает снимки состояния программы, определяя выполняемые функции, что позволяет с минимальным влиянием на скорость выявить узкие места. На основе этих снимков строится вероятностная картина распределения времени.

Пример - программа прерывается по таймеру, после чего записываются текущий адрес инструкции или стек вызовов.

Вкратце: при периодическом снимке экрана нет точной, как при tracing, но зато почти не замедляется прогон программы. На особо мелких функциях может очень плохо отрабатывать.

Достоинства:

- Минимальное влияние на скорость работы программы
- Возможность профилировать уже запущенные процессы без их остановки и без доступа к исходному коду.

Недостатки:

- Меньшая точность - кратковременные функции могут быть грубо говоря не замечены.
- Результаты носят статистический характер и требуют репрезентативной выборки.

3.5 Сравнение методов

Выбор между инструментированием и статистическим профилированием зависит от конкретной задачи:

- Если требуется абсолютно точная информация о каждом вызове, то предпочтительно инструментирование.
- Если приложение работает в промышленной среде, замедление недопустимо, или нужно диагностировать проблему на лету, то применяют статистический подход.

В следующих разделах будут рассмотрены конкретные реализации этих методов в экосистеме Python, их особенности и примеры использования.

4 Профилирование на уровне интерпретатора Python

4.1 Инструментирующие профилировщики

Экосистема Python предлагает широкий выбор профилировщиков как в стандартной библиотеке, так и в сторонних пакетах. Среди них:

- `profile` (стандартная библиотека, Legacy)
- `cProfile` (стандартная библиотека)
- `line_profiler` (построчное профилирование)
- `memory_profiler` (профилирование памяти)
- `yappi` (многопоточное и асинхронное профилирование)

4.1.1 Profile (Legacy)

Profile - Python-профилировщик, входящий в стандартную библиотеку. Он реализует метод инструментирования (*tracing*) и собирает детальную статистику о вызовах функций, времени их выполнения и количестве обращений.

Реализация

profile написан целиком на Python (pure Python). Это обеспечивает его переносимость, но приводит к значительным накладным расходам. Из-за своей реализации он считается **устаревшим**, так как в стандартную библиотеку добавил `cProfile`, который написан на C extensions (то есть на языке Си, который дергается из кода Python).

Принцип работы

Профилировщик перехватывает события вызова и возврата из каждой функции с помощью механизма `sys.setprofile()`. На каждый вызов засекается время, ведётся подсчёт количества обращений и строится иерархия вызовов.

```
1 import profile
2
3 def heavy_func():
4     total = 0
5     for i in range(1000):
6         total += i ** 2
7     return total
8
9 profile.run('heavy_func()', 'profile_stats.prof')
10 profile.runctx('heavy_func()', globals(), locals(), 'profile_ctx.prof')
```

Анализ:

Для анализа сохранённой статистики используется модуль **pstats**

```
1 import pstats
2 stats = pstats.Stats('profile_stats.prof')
3 stats.sort_stats('cumtime').print_stats(10)
```

Недостатки profile

1. Так как реализация написана на Python, то средство крайне медленное по сравнению с cProfile и может замедлять прогон кода в десятки раз.
2. Из-за высоких накладных расходов искажается реальная производительность.
3. Не поддерживает профилирование потоков.
4. Поддерживает ручную обработку результатов, что неудобно.

4.1.2 cProfile

cProfile - современный, рекомендуемый tracing профилировщик, входящий в стандартную библиотеку Python начиная с версии 2.5. Он выполняет инструментирование аналогично **profile**, но написан на языке C в виде расширения (C extensions), что обеспечивает на порядок более высокую скорость работы.

Принцип работы

cProfile, аналогично **profile**, использует перехват событий вызова/возврата через API профилирования виртуальной машины Python, реализованное на C. Благодаря низкоуровневой интеграции накладные расходы существенно снижены

Использование

Запустить профилирование можно тремя способами:

1. Из командной строки:

```
1 python -m cProfile [-o output_file] script.py
```

2. Внутри скрипта:

```
1 import cProfile
2 cProfile.run('heavy_func()', 'cprofile_stats.prof')
```

3. Контекстный менеджер (Python 3.8+):

```
1 with cProfile.Profile() as profiler:
2     heavy_func()
3     profiler.dump_stats('cprofile_context.prof')
```

Анализ результатов

Как и для `profile`, статистика сохраняется в бинарном формате и обрабатывается модулем `pstats`:

```
1 import pstats
2 stats = pstats.Stats('cprofile_stats.prof')
3 #
4 stats.strip_dirs() # remove absolute paths
5 stats.sort_stats('tottime') # sorting by your own time
6 stats.print_stats(10) # bring out the 10 most "heavy" functions
```

Возможно также получение статистики по вызывающим/вызываемым функциям:

```
1 stats.print_callers(5)
2 stats.print_callees(5)
```

Визуализация

Бинарные файлы, созданные **cProfile**, могут быть преобразованы в графические отчёты с помощью сторонних утилит:

- **snakeviz** — веб-интерфейс для интерактивного исследования (команда `snakeviz cprofile_stats.prof`);
- **gprof2dot** + Graphviz — создание диаграмм в формате DOT (`python -m gprof2dot -f pstats cprofile_stats.prof | dot -Tpng -o output.png`);
- **pyprof2calltree** — конвертация для просмотра в KCachegrind.

Сравнение с profile

- **Скорость:** **cProfile** работает в 5–20 раз быстрее **profile** (накладные расходы приблизительно 10–30% против 200–1000%).
- **Точность:** за счёт меньшего искажения времени выполнения результаты **cProfile** ближе к реальности.
- **Функциональность:** оба инструмента предоставляют одинаковый набор данных и API, но **cProfile** дополнительно поддерживает профилирование многопоточных программ.
- **Статус:** **cProfile** активно поддерживается и рекомендуется официальной документацией; **profile** сохраняется для обратной совместимости.

Недостатки cProfile

1. Всё ещё ощутимое замедление (от 10% до 50% в зависимости от программы), что ограничивает применение в промышленной эксплуатации.
2. Невозможность профилирования асинхронного кода (`async/await`) без дополнительных ухищрений (для этого существуют специализированные инструменты, например `yappi`).
3. Отсутствие встроенных средств для анализа использования памяти.

Пример использования cProfile: от запуска до визуализации

Рассмотрим полный цикл работы с `cProfile` на примере функции, выполняющей вычисление чисел Фибоначчи рекурсивным и итеративным способами. Это позволит наглядно продемонстрировать сбор статистики, её анализ и построение графического отчёта.

```
1 def fib_recursive(n):
2     if n <= 1:
3         return n
4     return fib_recursive(n-1) + fib_recursive(n-2)
5
6 def fib_iterative(n):
7     a, b = 0, 1
8     for _ in range(n):
9         a, b = b, a + b
10    return a
11
12 def main():
13     fib_recursive(35)
14     fib_iterative(100000)
15
16 if __name__ == "__main__":
17     main()
```

1. Запуск профилирования Выполним профилирование из командной строки, сохранив результаты в файл `fib.prof`:

```
1 python -m cProfile -o fib.prof fib.py
```

2. Базовый анализ в консоли с помощью pstats Загрузим статистику и выведем 5 самых «тяжёлых» функций по собственному времени (`tottime`):

```
1 python -m pstats fib.prof
2 strip
3 sort tottime
4 stats 5
```

Результат выполнения этих команд представлен на рис. 5. Как видно из скриншота, рекурсивная функция `fib_recursive` вызвана огромное количество раз и занимает почти всё время выполнения, тогда как итеративная функция `fib_iterative` выполняется практически мгновенно. Уже на этом этапе очевидно узкое место.

```

(.venv) oleg@oleg-HKD-WXX ~/t/python> python -m cProfile -o fib.prof main.py
(.venv) oleg@oleg-HKD-WXX ~/t/python> python -m pstats fib.prof
Welcome to the profile statistics browser.
fib.prof% strip
fib.prof% sort tottime
fib.prof%
fib.prof% stats 5
Sat Feb 14 19:47:58 2026      fib.prof

      29860708 function calls (6 primitive calls) in 4.194 seconds

Ordered by: internal time
List reduced from 6 to 5 due to restriction <5>

   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
29860703/1    4.129    0.000    4.129    4.129  main.py:1(fib_recursive)
      1      0.065    0.065    0.065    0.065  main.py:7(fib_iterative)
      1      0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Profiler' objects}
      1      0.000    0.000    4.194    4.194  main.py:14(main)
      1      0.000    0.000    4.194    4.194  {built-in method builtins.exec}

```

Рис. 5: вывод pstats: 5 самых тяжёлых функций по tottime

3. Интерактивная визуализация с SnakeViz SnakeViz предоставляет браузерный интерфейс для изучения результатов. Установка и запуск:

```

1 pip install snakeviz
2 snakeviz fib.prof

```

Откроется диаграмма в виде «солнечных лучей» (sunburst) или иерархического списка, где размер каждого сектора пропорционален времени выполнения. На ней сразу заметен огромный вклад `fib_recursive` и её рекурсивных вызовов.

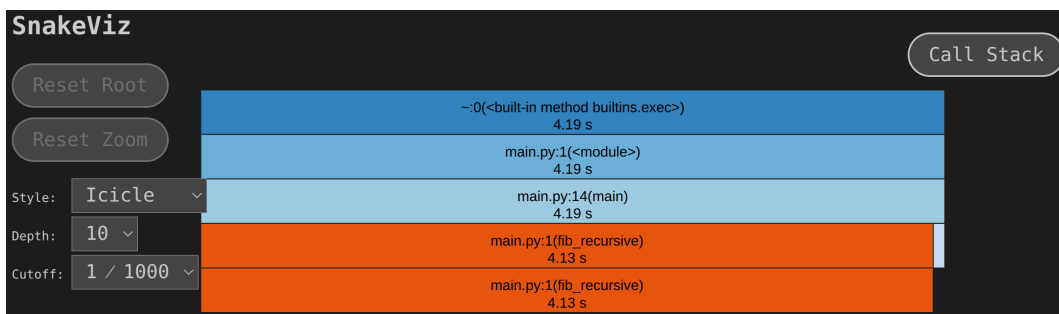


Рис. 6: snakeviz_cprofile

4. Генерация графа вызовов с gprof2dot Для создания диаграммы в формате PNG, отображающей связи между функциями, используем `gprof2dot` и `Graphviz`:

```
1 pip install gprof2dot
2 python -m gprof2dot -f pstats fib.prof | dot -Tpng -o fib_callgraph.png
```

Полученное изображение наглядно показывает, что `fib_recursive` вызывает саму себя многократно, тогда как `fib_iterative` изолирована и не имеет дочерних вызовов.

5. Интеграция с юнит-тестами Профилировщик можно встроить в тесты для контроля регрессий. Например, используя `unittest`:

```
1 import cProfile
2 import unittest
3
4 class TestFibonacci(unittest.TestCase):
5     def test_performance(self):
6         profiler = cProfile.Profile()
7         profiler.enable()
8         fib_iterative(100000)
9         profiler.disable()
10        stats = pstats.Stats(profiler)
11        stats.sort_stats('cumtime')
12        # Checking that the execution time does not exceed the threshold
13        self.assertLess(stats.total_tt, 0.01)
```

4.1.3 line_profiler

line_profiler — сторонний инструмент для построчного профилирования кода Python. В отличие от `cProfile`, который показывает время выполнения функций целиком, `line_profiler` измеряет время, затрачиваемое на каждую отдельную строку внутри функции. Это позволяет с высокой точностью локализовать «узкие места» на уровне отдельных операций.

Установка

Устанавливается через `pip`:

```
1 pip install line_profiler
```

После установки становится доступен исполняемый модуль `kernprof` и сам профилировщик.

Принцип работы

`line_profiler` использует инструментирование исходного кода: он модифицирует байткод функции, вставляя измерительные зонды перед каждой строкой. При выполнении функции замеряется время, прошедшее между зондами. Накладные расходы выше, чем у `cProfile`, но несравнимо более детальная информация оправдывает это при тонкой оптимизации.

Использование

Основной способ применения — декорирование целевой функции `@profile`. Важно: декоратор автоматически добавляется при запуске через `kernprof`, вручную импортировать его не нужно.

```
1 @profile
2 def slow_function():
3     total = 0
4     for i in range(100000):
5         total += i ** 2
6     return total
7
8 if __name__ == "__main__":
9     slow_function()
```

Запуск профилирования:

```
1 kernprof -l -v example.py
```

Ключ `-l` включает построчный режим (`line_profiler`), `-v` выводит результат сразу в консоль. По умолчанию создаётся файл `example.py.lprof`, который можно просмотреть позже утилитой `python -m line_profiler example.py.lprof`.

Пример вывода

Результат представляет собой таблицу со следующими столбцами:

```
(.venv) oleg@oleg-HKD-WXX ~/t/python> kernprof -l -v line_prof.py
Wrote profile results to 'line_prof.py.lprof'
Timer unit: 1e-06 s

Total time: 0.0623324 s
File: line_prof.py
Function: slow_function at line 1

Line #      Hits          Time    Per Hit   % Time  Line Contents
=====
     1              1          1.2        1.2     0.0      @profile
     2             100001      28802.8         0.3    46.2      def slow_function():
     3             100000      33526.3         0.3    53.8          total = 0
     4              1          2.1        2.1     0.0      for i in range(100000):
     5              1          2.1        2.1     0.0          total += i**2
     6              1          2.1        2.1     0.0      return total
```

Рис. 7: `line_profiler`

- **Line** — номер строки;
- **Hits** — количество выполнений данной строки;
- **Time** — общее время, затраченное на строку (в микросекундах);
- **Per Hit** — среднее время одного выполнения;
- **% Time** — доля от общего времени функции;
- **Line Contents** — исходный код строки.

Видно, что основное время тратится на вычисление квадрата и суммирование внутри цикла.

Достоинства

- Высочайшая детализация — информация по каждой строке.
- Простота использования — достаточно добавить декоратор.
- Интеграция с IPython.

Недостатки

- Замедление работы функции в разы (иногда на порядок), что ограничивает применение на реальных данных.
- Необходимость модифицировать исходный код (добавлять декоратор).
- Не работает с асинхронными функциями напрямую.
- Отсутствие встроенной визуализации (но результаты можно экспортировать).

`line_profiler` незаменим, когда нужно понять, какая именно операция внутри большой функции тормозит выполнение, и все другие методы уже не дают достаточной информации.

4.2 Статистические профилировщики

- `py_spy` - семплирование без остановки процесса
- `pprofile` - гибридный подход (детерминированный и статистический режимы)

4.2.1 `py-spy`

py-spy — семплирующий (статистический) профилировщик для Python-программ, написанный на языке Rust. Его ключевая особенность — полная независимость от целевого процесса: `py-spy` запускается вне интерпретатора Python и читает память профилируемой программы напрямую через системные вызовы (`process_vm_readv` в Linux, `vm_read` в macOS, `ReadProcessMemory` в Windows). Это позволяет:

- Профилировать уже запущенные процессы без их остановки;

- Не вносить никаких изменений в исходный код;
- Работать с production сервисами без заметного влияния на производительность
- Собирать данные даже в тех случаях, когда целевой интерпретатор собран без отладочных символов.

Установка

```
1 pip install py-spy
```

Также доступна установка через пакетные менеджеры: `brew install py-spy` (macOS), `yay -S py-spy` (Arch Linux), а для Rust-экосистемы — `cargo install py-spy`.

Принцип работы

`py-spy` читает память процесса и обходит структуры CPython. Достигается это за счет того, что `py-spy` находит глобальный `PyInterpreterState`, из него — все активные потоки, а затем и цепочки объектов `PyFrameObject`, формирующих стек вызовов. Поскольку `py-spy` не выполняется внутри интерпретатора, он не перехватывает события вызова/возврата и не замедляет работу кода. Семплирование происходит с заданной частотой (по умолчанию 100 Гц).

Базовые команды

`py-spy` предоставляет три основных режима работы:

1. `record` — запись профиля в файл с последующей визуализацией:

```
1 # Attach to a running process
2 py-spy record -o profile.svg --pid 12345
3
4 # Start a new process under profiler
5 py-spy record -o profile.svg -- python myprogram.py
```

Поддерживаемые форматы: SVG (интерактивный флеймграф), `speedscope`, сырые данные.

2. `top` — интерактивный просмотр «горячих» функций в реальном времени, аналогичный утилите `top`:

```
1 py-spy top --pid 12345
```

3. `dump` — мгновенный снимок стека вызовов всех потоков (полезно при зависаниях):

```
1 py-spy dump --pid 12345
2 py-spy dump --pid 12345 --locals # also show local variables
```

Ключевые возможности

- **Профилирование нативных расширений** (`-native`) — отображение имён функций из C/C++/Cython (доступно на Linux x86_64 и Windows) .
- **Отслеживание дочерних процессов** (`-subprocesses`) — автоматическое прикрепление к порождённым процессам (например, при использовании `multiprocessing` или `gunicorn`).
- **Фильтрация по GIL** (`-gil`) — учёт только потоков , удерживающих глобальную блокировку интерпретатора.
- **Исключение «спящих» потоков** — `py-spy` опрашивает состояние потоков через `/proc/PID/stat` (Linux), `thread_basic_info` (macOS) или анализ системных вызовов (Windows) и не включает в отчёт стеки, заведомо находящиеся в ожидании. Поведение можно изменить флагом `-idle`.

Особые требования

При профилировании существующего процесса `py-spy` требует прав на чтение чужой памяти. В macOS это всегда требует `sudo`, в Linux — если процесс запущен не самим профилировщиком . В контейнерах Docker и Kubernetes необходимо добавить capability `SYS_PTRACE`, иначе вызовы `process_vm_readv` будут запрещены.

Достоинства

- Нулевое вмешательство в код программы;
- Крайне низкие накладные расходы (применимо в продакшене);
- Работа с уже зависшими процессами (команда `dump`);
- Наглядная визуализация (флеймграфы, `speedscope`);
- Кроссплатформенность (Linux, macOS, Windows, FreeBSD).

Недостатки

- Статистическая природа — кратковременные функции могут быть «пропущены»;
- Требуется повышенных привилегий в ряде сценариев;
- Не даёт точного числа вызовов, только долевое распределение времени.

`py-spy` — оптимальный выбор для диагностики замедлений в промышленной эксплуатации, когда невозможно (или нежелательно) перезапускать приложение и вносить изменения в код.

4.2.2 pprofile

pprofile — легковесный, полностью написанный на Python профилировщик, реализующий оба метода сбора информации: детерминированное (trace) и статистическое (sample) профилирование. Его главная отличительная черта — *строчная гранулярность* (line-granularity) в сочетании с автоматическим отслеживанием всех потоков и возможностью работы без модификации исходного кода.

Установка

```
1 pip install pprofile
```

После установки становится доступен одноимённый исполняемый модуль.

Два режима работы

1. Детерминированное профилирование (по умолчанию) В этом режиме pprofile перехватывает каждое событие выполнения строки (line events), используя механизм `sys.settrace`. Это даёт абсолютно полную картину: сколько раз выполнялась каждая строка, сколько времени заняла, какова иерархия вызовов. Расплата — огромные накладные расходы (замедление в 10–100 раз), что ограничивает применение только короткими тестовыми сценариями.

```
1 # Running the script under a deterministic profiler
2 pprofile myscript.py
3
4 # Ignore system paths (so as not to clutter the output)
5 pprofile --exclude-syspath myscript.py
6
7 # Launching the module
8 pprofile -m mymodule -- --arg-for-module
```

Вывод представляет собой аннотированный исходный код: перед каждой строкой указывается количество попаданий (Hits), общее время, время на один вызов и доля в процентах.

```
1 File: demo/threads.py
2 File duration: 1.00168s (99.60%)
3 Line #| Hits| Time|Time per hit| %|Source code
4 -----+-----+-----+-----+-----+-----
5 4| 2| 3.21865e-05| 1.60933e-05| 0.00%| def func():
6 5| 1| 1.00111| 1.00111| 99.54%| time.sleep(1)
```

2. Статистическое профилирование (-statistic) В этом режиме pprofile периодически опрашивает стеки всех потоков (или только текущего) с заданным интервалом. Накладные расходы снижаются до приемлемых 1–5%, что позволяет применять pprofile к длительно работающим приложениям.

```
1 # Sampling every 0.01 seconds
2 pprofile --statistic 0.01 myscript.py
3
4 # Current stream only (single), 1 ms period
5 pprofile --statistic 0.001 --single myscript.py
```

Недостаток статистического режима — потеря точности: вместо реального времени выполнения строки показывают только количество «попаданий» в сэмплы, что даёт лишь приблизительную картину.

Поддержка многопоточности

В отличие от `cProfile`, `pprofile` «из коробки» корректно отслеживает порождаемые потоки. В детерминированном режиме для этого необходимо установить флаг `-threads 1` (или `threads=True` в API), в статистическом — все потоки процесса видны автоматически.

Визуализация (Callgrind / KCachegrind)

Одна из сильнейших сторон `pprofile` — экспорт в формат `callgrind`, совместимый с `KCachegrind` и `QKachegrind`. Это даёт возможность исследовать профиль в графическом интерфейсе с древовидными диаграммами, поиском «горячих» путей и фильтрацией.

```
1 pprofile —format callgrind —out cachegrind.out.myprofile myscript.py
2 # Or short entry (file name starts with cachegrind.out)
3 pprofile —out cachegrind.out.myprofile myscript.py
4
5 # Additionally — package the source code in a zip to display the paths
  correctly
6 pprofile —out cachegrind.out.myprofile —zipfile src.zip myscript.py
```

Программный API

`pprofile` можно встраивать непосредственно в код, что полезно для профилирования отдельных участков:

```
1 import pprofile
2
3 # Deterministic site profile
4 prof = pprofile.Profile()
5 with prof():
6     # ... the code for profiling ...
7     prof.print_stats()
8
9 # Statistical profile (sampling the current stream)
10 stat_prof = pprofile.StatisticalProfile()
11 with stat_prof(period=0.001):
12     # ... the code for profiling ...
13     stat_prof.print_stats()
```

Сравнение с `line_profiler`

Оба инструмента дают построчную информацию, но принципиально различаются:

- `line_profiler` требует декоратора `@profile` и запуска через `kernprof`, фокусируясь только на отмеченных функциях;
- `pprofile` анализирует *весь* код приложения без каких-либо изменений, что удобно для первичного поиска узких мест, но порождает огромные объёмы вывода.

Достоинства

- Чистый Python — работает там, где нельзя собрать C-расширение;
- Два режима профилирования в одном инструменте;
- Полная поддержка потоков;
- Экспорт в Callgrind для профессионального анализа;
- Не требует модификации исходного кода (в отличие от `line_profiler`).

Недостатки

- Детерминированный режим неприменим к реальным задачам из-за колоссального замедления;
- Статистический режим даёт лишь приблизительные результаты;
- Проект давно не обновлялся (последний релиз — 2016 г.) ;
- Меньше возможностей визуализации по сравнению с `py-spy` (флеймграфы не строятся напрямую).

`rprofile` занимает нишу «тяжёлой артиллерии» для всестороннего исследовательского профилирования в окружениях, где недопустима компиляция или недоступны современные альтернативы. Однако в новых проектах приоритет следует отдавать `py-spy` (продакшен) либо связке `cProfile` + `snakeviz` (разработка).

4.3 Специализированные решения

4.3.1 `yappi`

yappi (Yet Another Python Profiler) — профилировщик, специализирующийся на многопоточных и асинхронных приложениях. Он поддерживает как инструментирование (`tracing`), так и статистический режим (`wall time` и `cpu time`), и отличается низкими накладными расходами при работе с большим количеством потоков.

Установка

```
1 pip install yappi
```

Ключевые особенности

- **Поддержка потоков** — корректно учитывает время выполнения каждого потока отдельно.
- **Поддержка `asyncio`** — профилирование корутин и задач (с версии 1.3.0).
- **Два режима таймера:** `wall-time` (реальное время) и `cpu-time` (процессорное время).

- **Статистическое профилирование** — возможность переключения в режим `sampling` для минимизации накладных расходов.
- **Богатый API** — запуск, остановка, сброс статистики, получение отчётов в различных форматах.

Принцип работы

В режиме `tracing` `yappi` аналогично `cProfile` перехватывает вызовы функций, но реализация оптимизирована для работы в многопоточной среде (использует локальные для потока структуры данных). В режиме `sampling` профилировщик периодически опрашивает стеки всех потоков, что даёт минимальную задержку.

Использование

Рассмотрим пример профилирования многопоточной программы.

```
1 import yappi
2 import threading
3 import time
4
5 def worker(name):
6     for _ in range(5):
7         time.sleep(0.1)
8         print(f"{name} working")
9
10 threads = []
11 for i in range(3):
12     t = threading.Thread(target=worker, args=(f"Thread-{i}",))
13     threads.append(t)
14
15 yappi.set_clock_type("wall") # "wall" or "cpu"
16 yappi.start()
17 for t in threads:
18     t.start()
19 for t in threads:
20     t.join()
21 yappi.stop()
22
23 func_stats = yappi.get_func_stats()
24 func_stats.sort("ttot", type="cum").print_all()
25
26 thread_stats = yappi.get_thread_stats()
27 thread_stats.print_all()
28
29 func_stats.save('yappi_stats.prof', type='pstats')
```

Пример с `asyncio`

Начиная с версии 1.3.0, `yappi` может профилировать асинхронный код.

```

1 import yappi
2 import asyncio
3
4 async def fetch_data():
5     await asyncio.sleep(0.1)
6     return "data"
7
8 async def main():
9     results = await asyncio.gather(fetch_data(), fetch_data(), fetch_data())
10
11     yappi.set_clock_type("wall")
12     yappi.start()
13     asyncio.run(main())
14     yappi.stop()
15
16     stats = yappi.get_func_stats()
17     stats.print_all()

```

Сравнение с cProfile

- **Потоки:** cProfile не разделяет статистику по потокам — все вызовы смешиваются; yappi предоставляет отдельную статистику для каждого потока.
- **Asyncio:** cProfile видит только вызовы функций, но не корутины; yappi корректно отображает время выполнения асинхронных задач.
- **Скорость:** В режиме `tracing` yappi немного медленнее cProfile из-за дополнительных блокировок; в режиме `sampling` — значительно быстрее.
- **Гибкость:** yappi позволяет динамически переключать режимы, очищать статистику и профилировать только интересующие потоки.

Достоинства

- Полноценная поддержка многопоточности и asyncio.
- Два режима профилирования (инструментирование и семплирование).
- Низкие накладные расходы в режиме `sampling` (подходит для продакшена).
- Богатый API и возможность экспорта в формат `pstats`.
- Поддержка профилирования времени ЦП и реального времени.

Недостатки

- Менее распространён, чем cProfile, меньше интеграций с визуализаторами (хотя файлы `pstats` совместимы).
- В режиме `tracing` может быть медленнее cProfile для однопоточных приложений.

- Документация не всегда подробна, хотя проект активно развивается.

`yappi` становится незаменимым инструментом при разработке высоконагруженных сетевых сервисов, веб-приложений на `asynio`, а также при необходимости профилирования в условиях, когда нельзя останавливать сервер (режим семплирования).

5 Профилирование нативных расширений (C extensions)

5.1 Особенности профилирование кода на C/C++ внутри Python

5.2 Инструменты для анализа C/C++: `gprof`, `Valgrind` (`Callgrind`), `Google PerfTools`

5.3 Получение смешанных стеков вызовов (Python + C) с помощью `perf` и `eBPF`

6 Системное профилирование Python-приложений

6.1 Использование утилиты `perf` в Linux для анализа на уровне ядра

6.2 `DTrace` и `SystemTap`: динамическая трассировка

6.3 Профилирование операций ввода-вывода и работы с сетью

7 Сравнительный анализ и практические рекомендации