Yandex Crowd JPoint

Анализ производительности IO-систем на примере асинхронных HTTP-клиентов

Данил Деминцев

О чем пойдет речь?

Что мы
будем
анализировать?

01

Как подобрать
железо
для измерений?

02

Как настроить
выбранный
стенд?

03

Как написать
бенчмарк
и не наступить
на грабли?

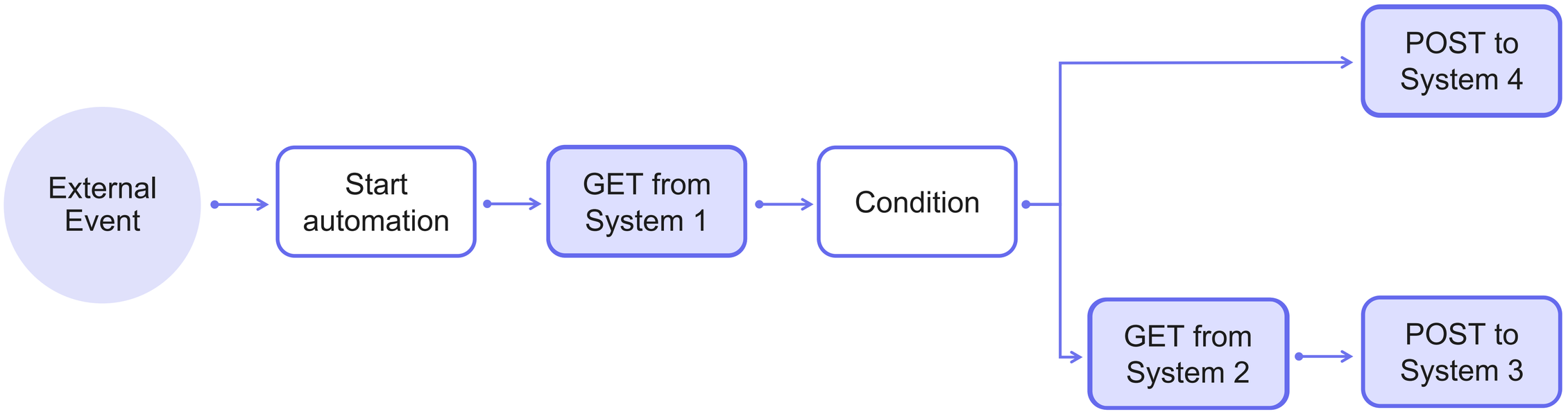
04

Как анализировать
полученные
результаты?

05

С чего все началось?

Одновременное исполнение множества user-defined автоматизаций:



Нужно выбрать HTTP клиент с максимальной пропускной способностью!

Железо
для бенчмаркинга

01

Требования к железу

Isolated

бенчмарк не должен ломать внешний мир

Stable

железо должно вести себя одинаково

Production-like

результаты должны быть релевантны

Localhost vs network

Localhost

- \+ stable
- \+ simple
- − irrelevant

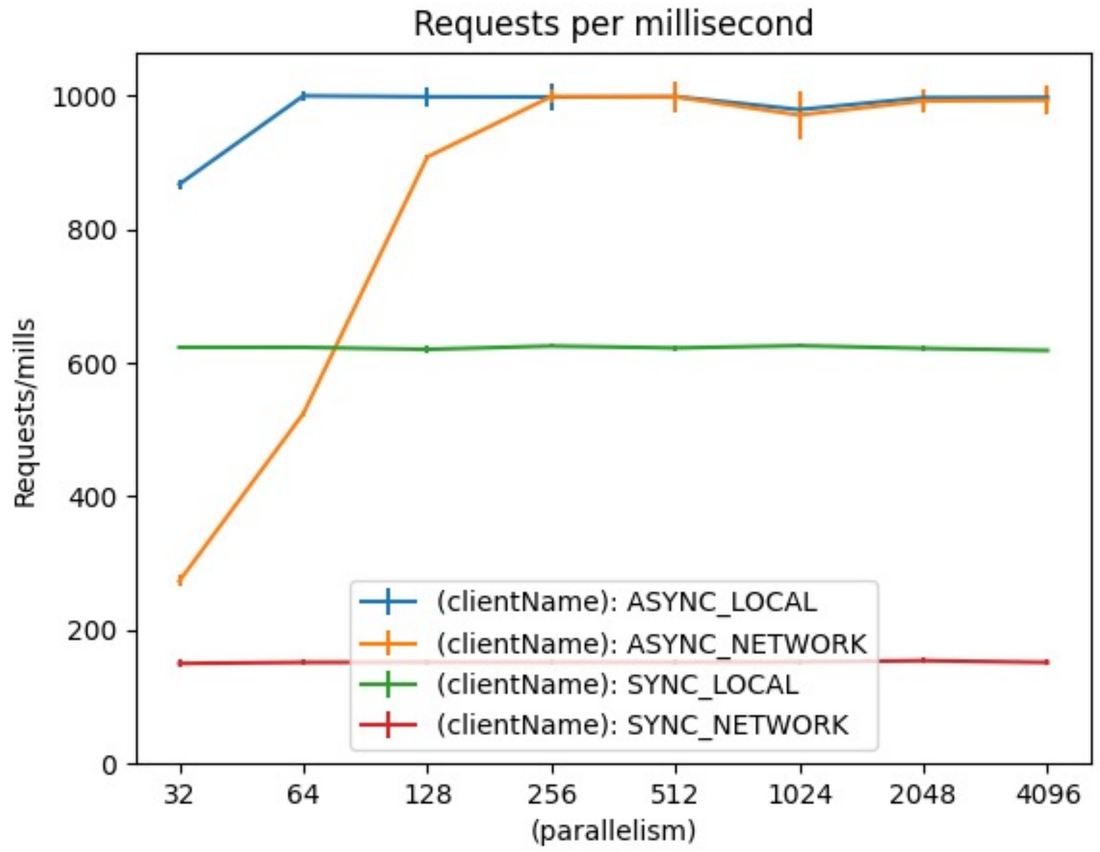
Synthetic network

- \+ stable
- ± simple
- ± relevant

Real network

- \+ relevant
- − unstable
- − too complex

Localhost vs network



Async (сверху) —
localhost ~ network

Sync (снизу) —
localhost != network

Параметры стенда

Debian 12 x86_64

100+Gb RAM

16 physical cores

32 logical cores

```
root@ddd127-benchmarks-experiments:~# screenfetch
          _,met$$$$$gg.          root@ddd127-benchmarks-experiments
       ,g$$$$$$$$$$$$$$$P.       OS: Debian 12 bookworm
     ,g$$P"         """Y$$.".    Kernel: x86_64 Linux 6.1.0-17-amd64
    ,$$P'               `$$$.    Uptime: 34m
   ',$$P       ,ggs.     `$$b:   Packages: 559
   `d$$'     ,$P"'   .    $$$    Shell: bash 5.2.15
    $$P      d$'     ,    $$P    Disk: 11G / 625G (2%)
    $$:      $$.   -    ,d$$'    CPU: AMD EPYC-Milan @ 32x 2.396GHz
    $$\;      Y$b._   _,d$P'     GPU: Red Hat, Inc. Virtio 1.0 GPU (rev 01)
    Y$$.    `.`"Y$$$$P"'        RAM: 1340MiB / 125781MiB
    `$$b      "-.__
     `Y$$
      `Y$$.
        `$$b.
          `Y$$b.
             `"Y$b._
                 `"""
```

```
root@ddd127-benchmarks-experiments:~# lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          40 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   32
  On-line CPU(s) list:    0-31
Vendor ID:                AuthenticAMD
  BIOS Vendor ID:         QEMU
  Model name:             AMD EPYC-Milan Processor
    BIOS Model name:      NotSpecified  CPU @ 2.0GHz
    BIOS CPU family:      1
    CPU family:           25
    Model:                1
    Thread(s) per core:   2
    Core(s) per socket:   16
    Socket(s):            1
    Stepping:             1
    BogoMIPS:             4792.79
```

Промежуточная мораль №1

При выбор железа мы обратили внимание на:

Изоляцию, стабильность и релевантность самой машинки

Разные варианты сетевого соединения

Возможно, для вашего случая что-то можно упростить

Настройка окружения для бенчмаркинга

02

Выбираем web server

Чтобы бенчмаркать клиент, нужен сервер
Open Source решений, в целом, достаточно

01

02

03

Какой-нибудь
внутренний велосипед

Nginx

/etc/nginx/nginx.conf:

```
root@ddd127-test:~# \
> apt-get install nginx -y
 ...
root@ddd127-test:~# \
> curl -v http://localhost:80
*   Trying 127.0.0.1:80 ...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.88.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx/1.22.1
```

Какую нагрузку держит nginx на localhost?

Nginx: из коробки

Запустим wrk2 (http benchmarking tool):

```
root@ddd127:~# ./wrk2-master/wrk -c 600 -d 30 -t 16 -R 2000000 \
http://localhost:80/

Running 30s test @ http://localhost:80/

  <...>

Requests/sec: 367555.35
```

wrk2 — https://github.com/giltene/wrk2

Nginx: подкручиваем

/etc/nginx/nginx.conf:

```
##
# Logging Settings
##

# access_log /var/log/nginx/access.log;
access_log off;

server {
        listen [::]:8080 default_server;

        location /do_request {
                return 200 'You are welcome!';
        }
}
```

Отключаем логирование

Хардкодим респонс

Nginx: подкрученный

Запустим wrk2 ещё раз:

```
root@ddd127:~# ./wrk2-master/wrk -c 600 -d 30 -t 16 -R 2000000 ...

  <...>


Requests/sec: 1851509.47
```

1.8m RPS

Итого

Если клиенты приблизятся к этому значению, надо тюнить ещё

Изоляция ресурсов

Для корректности измерений нужно, чтобы Nginx
и клиенты не конкурировали за ресурсы машинки

Disk bandwidth

ни клиенту,
ни серверу
не критичен

Network bandwidth

не ограничитель,
т.к. localhost

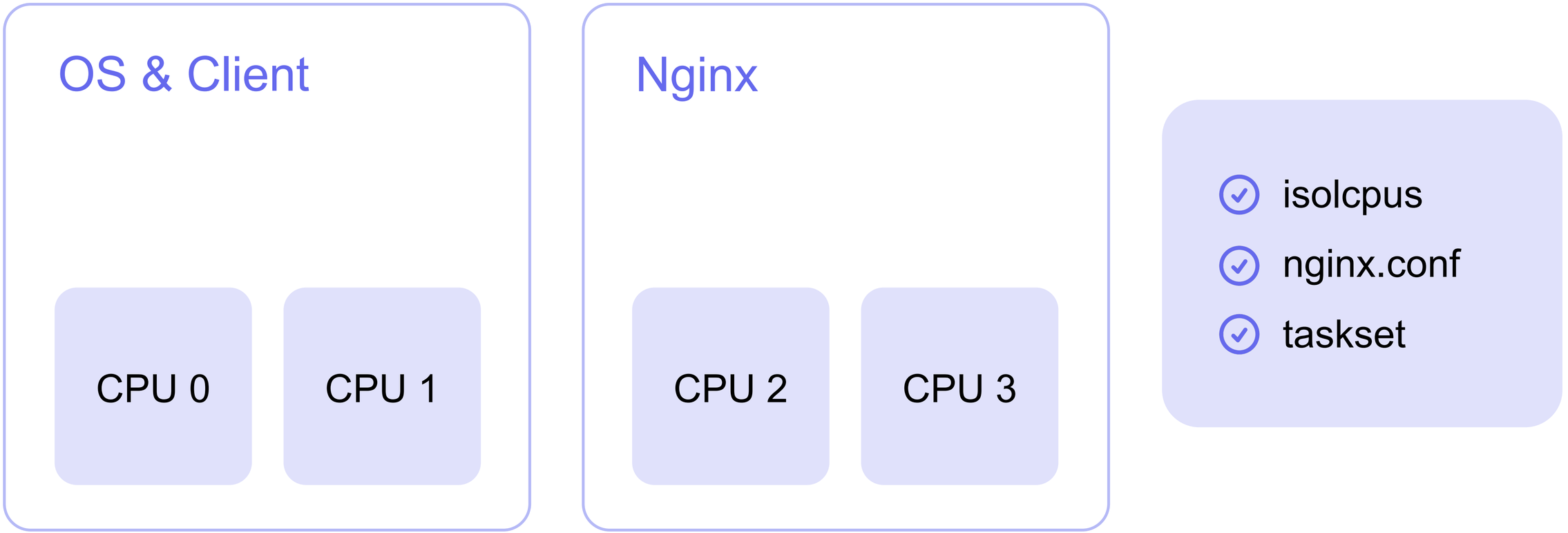
100+Gb RAM

«должно хватить каждому» ©

CPU

Изоляция CPU

Можно изолировать отдельные CPU ядра для веб-сервера



(non-)isolated CPU

Non-isolated CPU

Isolated & Network

Промежуточная мораль №2

При настройке окружения мы:

Провели тюнинг задействованных в измерении компонент

Получили baseline измеряемых величин сторонними тулзами

Уделили внимание ресурсам и их изоляции

Бенчмаркинг асинхронных API

03

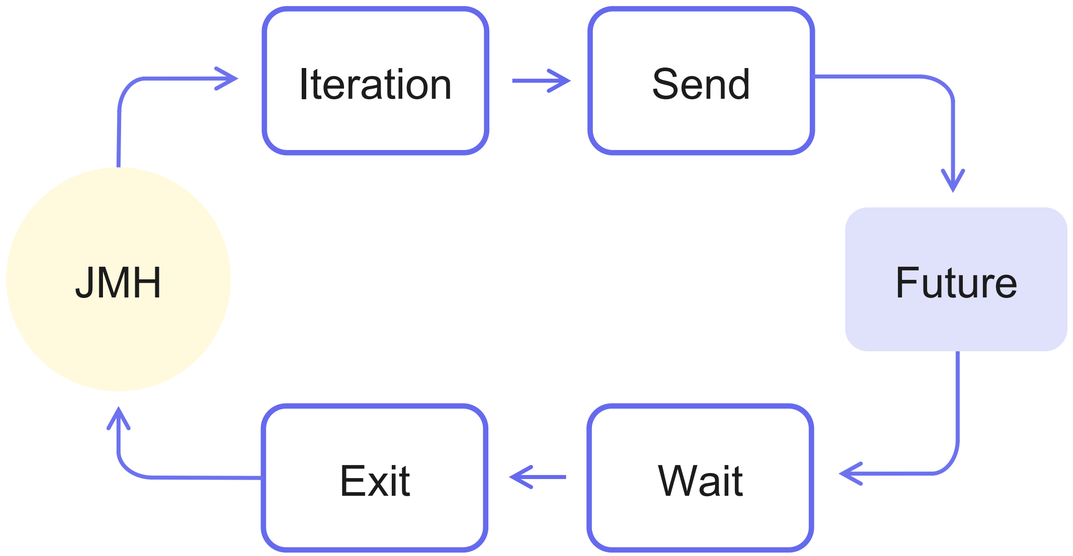
JMH

Java Microbenchmark Harness
https://github.com/openjdk/jmh

> “JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.”
>
> © JMH Readme

Single thread

Throughput, rq/mills	Error (99.9%)
9.156	± 0.700

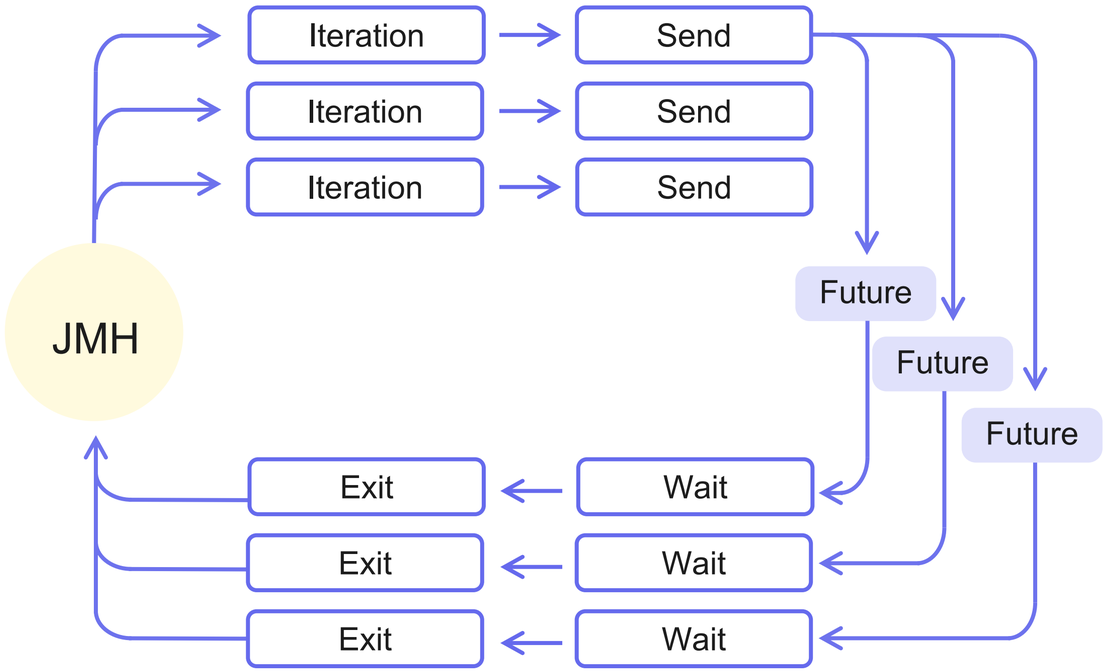



```java
/* other benchmark annotations */
@BenchmarkMode(Mode.Throughput)
public class Benchmark {

    @State(Scope.Benchmark)
    public static class BenchmarkState {

        @Param(value = {"8"})
        private int threads;
        private HttpClient client;

        @Setup(Level.Trial)
        public void setup() {
            client = /* init client with given threads count */;
        }
    }

    @Benchmark
    public HttpResponse<byte[]> benchmark(final BenchmarkState benchmarkState) {
        final CompletableFuture<HttpResponse<byte[]>> future =
                benchmarkState.getClient().sendAsync(/* build request */);
        return future.get();
    }
}
```

future.get()

Multiple thread

Threads	Throughput, rq/mills	Error (99.9%)
2	19.687	± 0.400
4	36.630	± 0.602
8	68.795	± 0.764
16	107.856	± 1.456

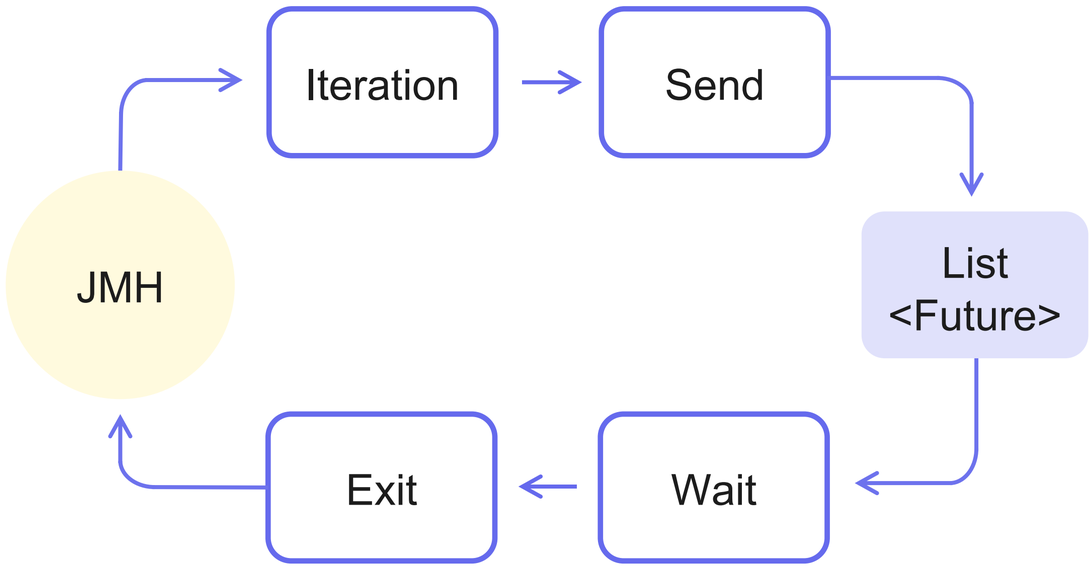



```java
/* ... */
public class Benchmark {

    /* ... */

    @Benchmark
    @Thread(2)
    public HttpResponse<byte[]> benchmark_threads_2(/* ... */) {
        /* ... */
    }

    /* ... */

    @Benchmark
    @Thread(16)
    public HttpResponse<byte[]> benchmark_threads_16(/* ... */) {
        /* ... */
    }
}
```

parallelism

Batch benchmark

Batch size	Throughput, rq/mills	Error (99.9%)
4	32.779	± 0.499
16	73.503	± 1.683
64	106.449	± 1.295
256	105.982	± 1.357

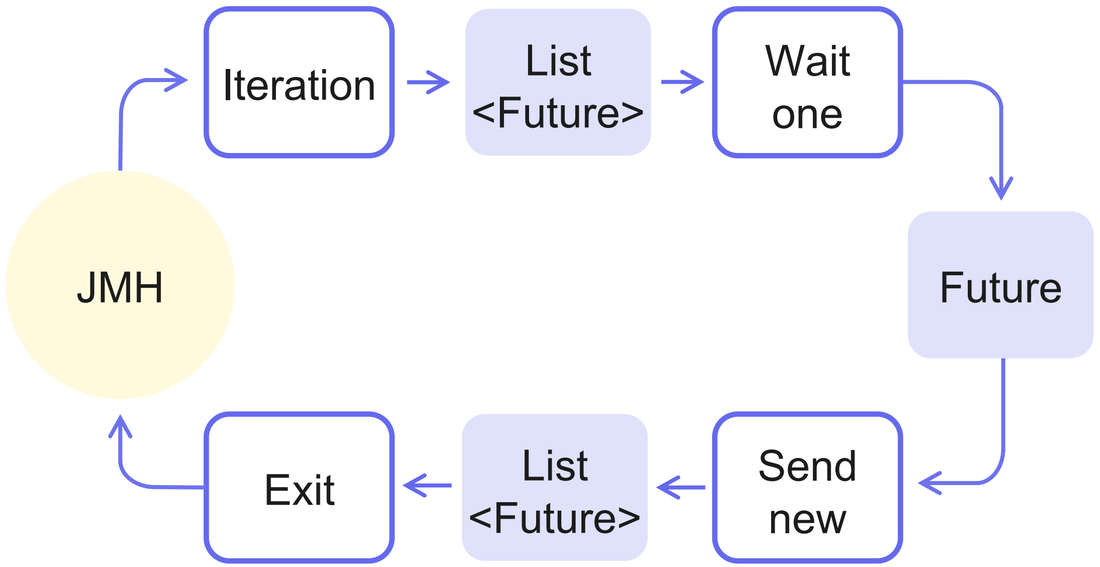



```java
/* ... */
public class Benchmark {

    /* ... */

    private List</* ... */> doRequests(BenchmarkState state, int requests) {
        /* send and await for given requests number */
    }

    @Benchmark
    @OperationsPerInvocation(4)
    public List</* ... */> benchmark_threads_2(/* ... */) {
        return doRequests(state, requests: 4);
    }

    /* ... */

    @Benchmark
    @OperationsPerInvocation(256)
    public List</* ... */> benchmark_threads_16(/* ... */) {
        return doRequests(state, requests: 256);
    }
}
```

multiple requests

Constant load

Parallel	Throughput, rq/mills	Error (99.9%)
4	40.662	± 0.895
16	117.540	± 1.521
64	126.508	± 0.873
256	123.301	± 0.630




```java
public class Example_05_ParallelizeSingleThread {

    /* ... */
    public static class BenchmarkState {

        @Param(value = {"4", "16", "64", "128"})
        private int parallelism;
        private List<CompletableFuture<HttpResponse<byte[]>>> futures;

        /* ... */

        @Setup(Level.Trial)
        public void setup() {
            futures = /* initialize list of nulls using parallelism size */;
        }
    }

    @Benchmark
    public HttpResponse<byte[]> benchmark(BenchmarkState benchmarkState) {
        while (true) {
            for (int i = 0; i < futures.size(); ++i) {
                final var future = futures.get(i);
                if (future != null && !future.isDone()) continue;
                futures.set(i, client.sendAsync(/* build new request */));
                if (future != null) return future.get();
            }
        }
    }
}
```

running requests

wait one

Body example

Body	Throughput, rq/mills	Error (99.9%)
0	128.508	± 1.465
8192	68.531	± 0.824
65k	11.893	± 0.930
524k	1.928	± 0.367

```java
public static class BenchmarkState {

    /* ... */
    @Param(value = {"0", "8192", "65536", "524288"})
    private int bodySize;

    public byte[] getBodyOrNull() {
        if (bodySize == 0) return null;
        final byte[] body = new byte[bodySize];
        ThreadLocalRandom.current().nextBytes(body);
        return body;
    }
}

@Benchmark
public HttpResponse<byte[]> benchmark(final BenchmarkState benchmarkState) {
    while (true) {
        for (int i = 0; i < futures.size(); ++i) {
            /* ... */
            final var body = benchmarkState.getBodyOrNull(benchmarkState.getBodySize());
            builder = (body != null)
                    ? builder.POST(HttpRequest.BodyPublishers.ofByteArray(body))
                    : builder.GET();
            /* ... */
        }
    }
}
```

generation

Body fix

Body	Throughput, rq/mills	Error (99.9%)
0	126.340	± 1.793
8192	113.528	± 1.951
65k	68.513	± 0.907
524k	17.748	± 0.281

```java
public static class BenchmarkState {

    /* ... */
    @Param(value = {"0", "8192", "65536", "524288"})
    private int bodySize;
    private byte[] body;

    /* ... */
    public void setup() {
        body = getBodyOrNull();
        /* ... */
    }
}

@Benchmark
public HttpResponse<byte[]> benchmark(final BenchmarkState benchmarkState) {
    while (true) {
        for (int i = 0; i < futures.size(); ++i) {
            /* ... */
            final var body = benchmarkState.getBody();
            builder = (body != null)
                    ? builder.POST(HttpRequest.BodyPublishers.ofByteArray(body))
                    : builder.GET();
            /* ... */
        }
    }
}
```

⟵ generate once

⟵ use

Промежуточная мораль №3

Прежде, чем что-то бенчмаркать, мы проверили, что:

Бенчмарк измеряет то, что мы хотели

Бенчмарк отражает реальный workload

Значения параметров имеют достаточное покрытие

Бенчмаркинг клиентов

04

Рассматриваемые клиенты

01 SYNC_BASELINE — Sync HttpClient5 + ThreadPool

version: 5.3 https://github.com/apache/httpcomponents-client

02 ASYNC_JAVA — Java HttpClient

version: openjdk (build 17.0.9) https://github.com/openjdk/jdk17u

03 ASYNC_OVER_NETTY — AsyncHttpClient over Netty Framework

version: 2.12.3 https://github.com/AsyncHttpClient/async-http-client

04 ASYNC_APACHE — Async Apache HttpClient5

version: 5.3 https://github.com/apache/httpcomponents-client

Изменяемые параметры

IO threads

число тредов
внутри клиента
[2, 4, 6, 8, 10]

Producer threads

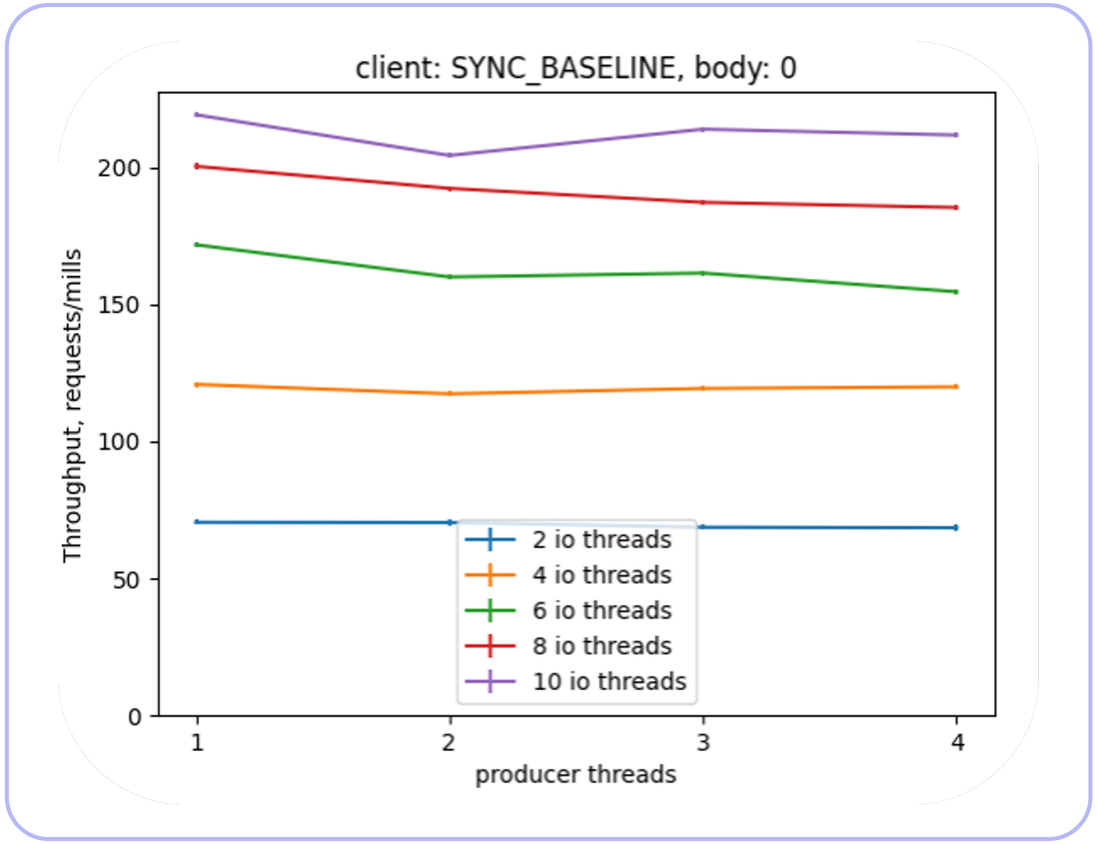
число тредов
бенчмарка,
отправляющих
запросы [1-4]

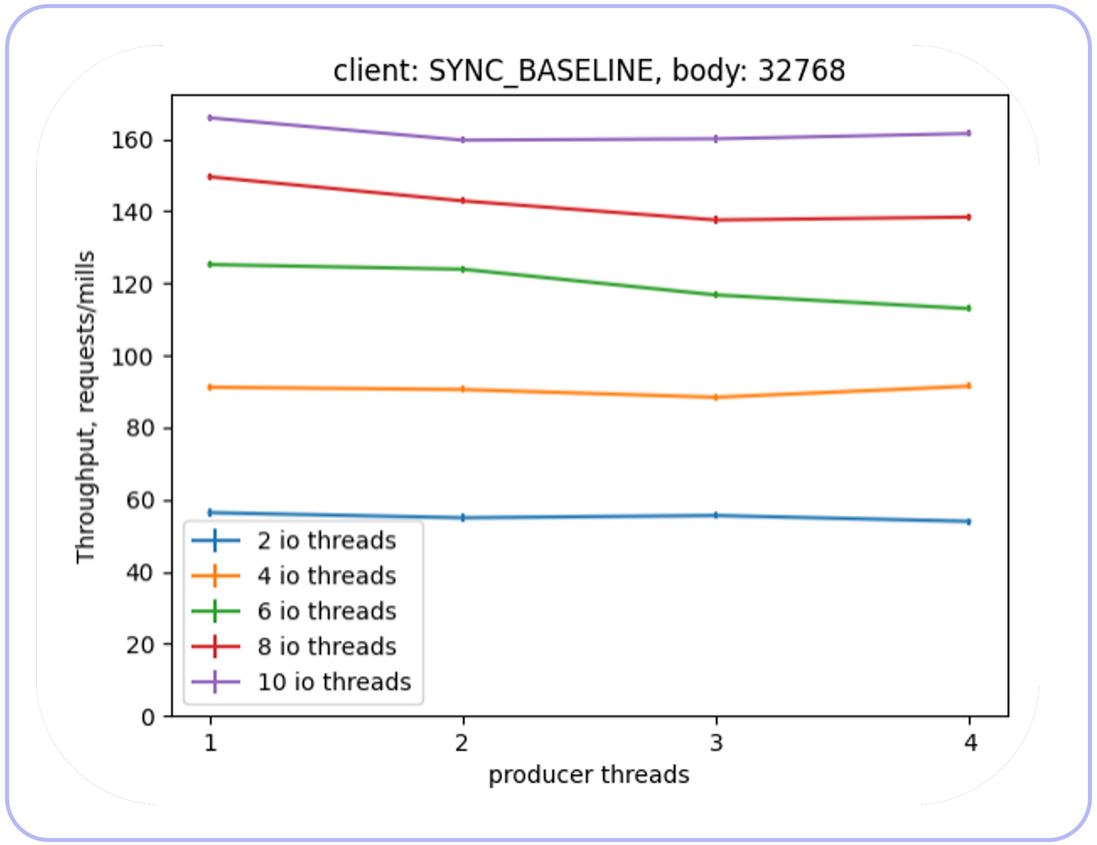
Body

размер тела
запроса
[0, 8kB, 32kB,
128kB, 512kB]

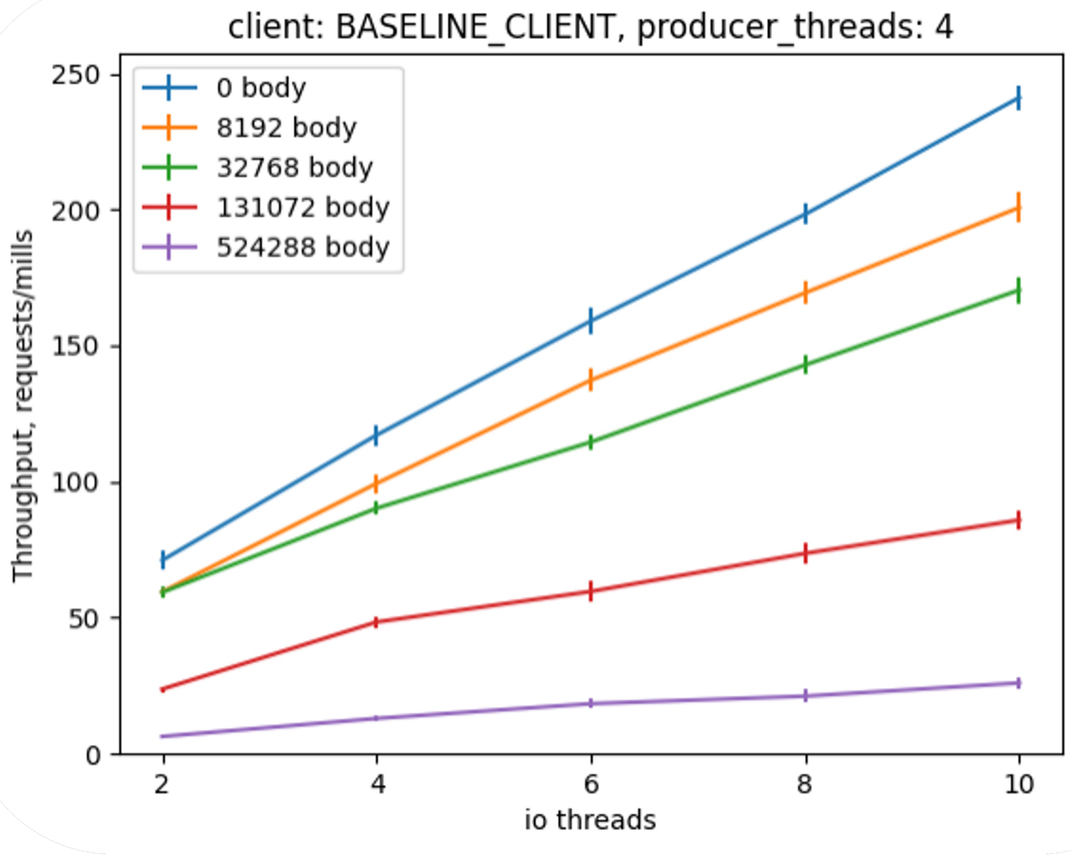
Sync + ThreadPool: producer threads

SYNC_BASELINE, зависимость RPS от числа producer тредов





Sync + ThreadPool: IO threads



SYNC_BASELINE:
зависимость RPS
от IO тредов

Sync + ThreadPool: summary

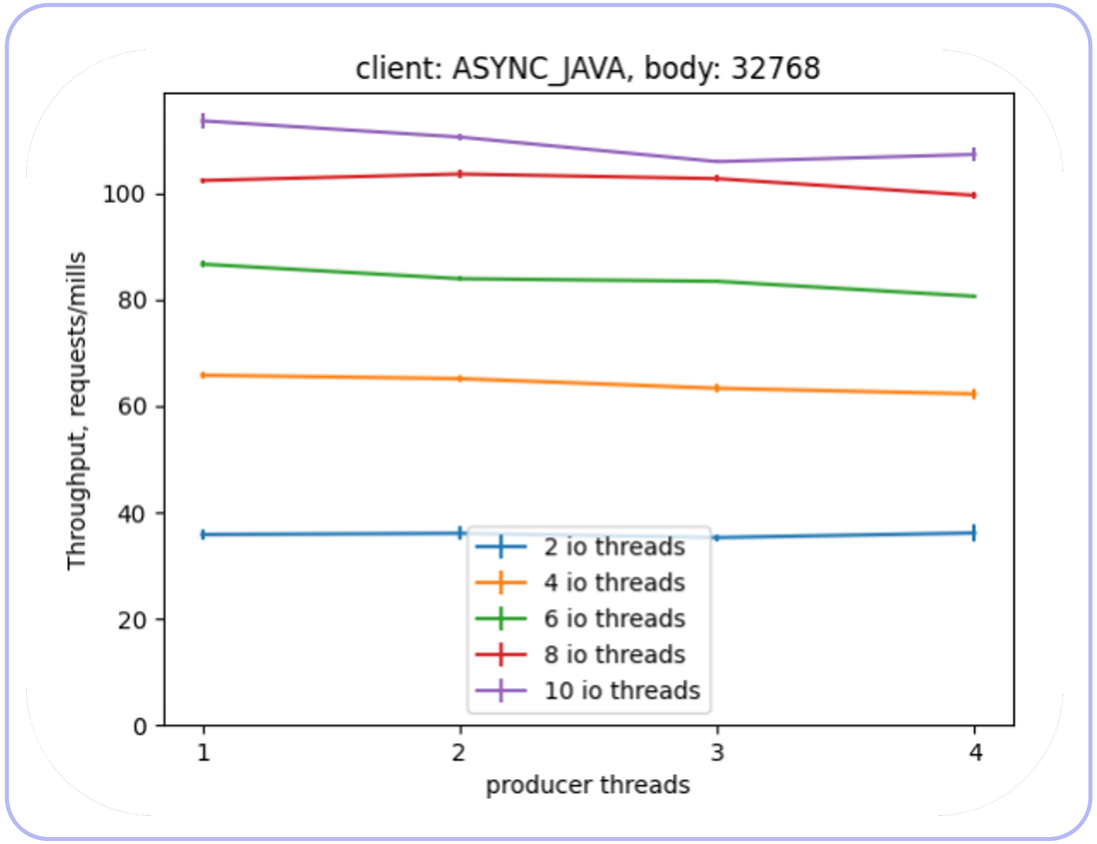
Sync Apache HttpClient5 + ThreadPool:

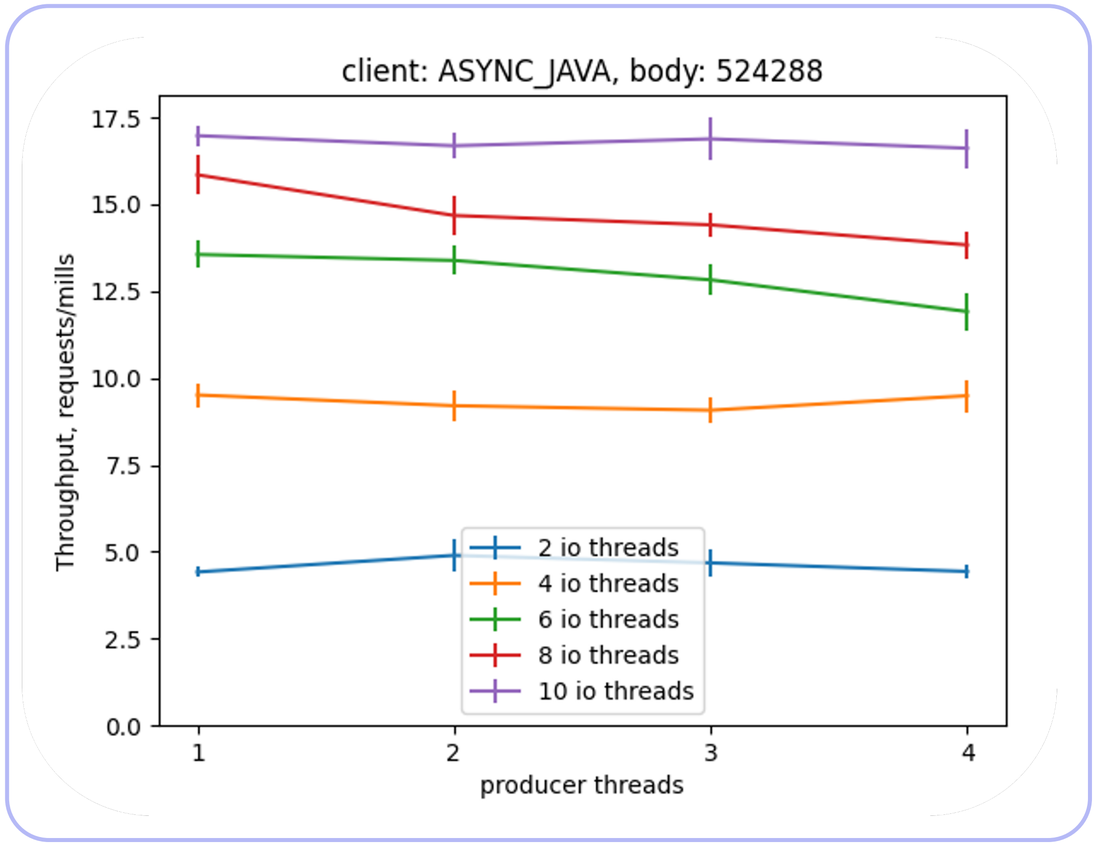
Не зависит от числа
producer-тредов

RPS линейно растет
при росте числа IO тредов

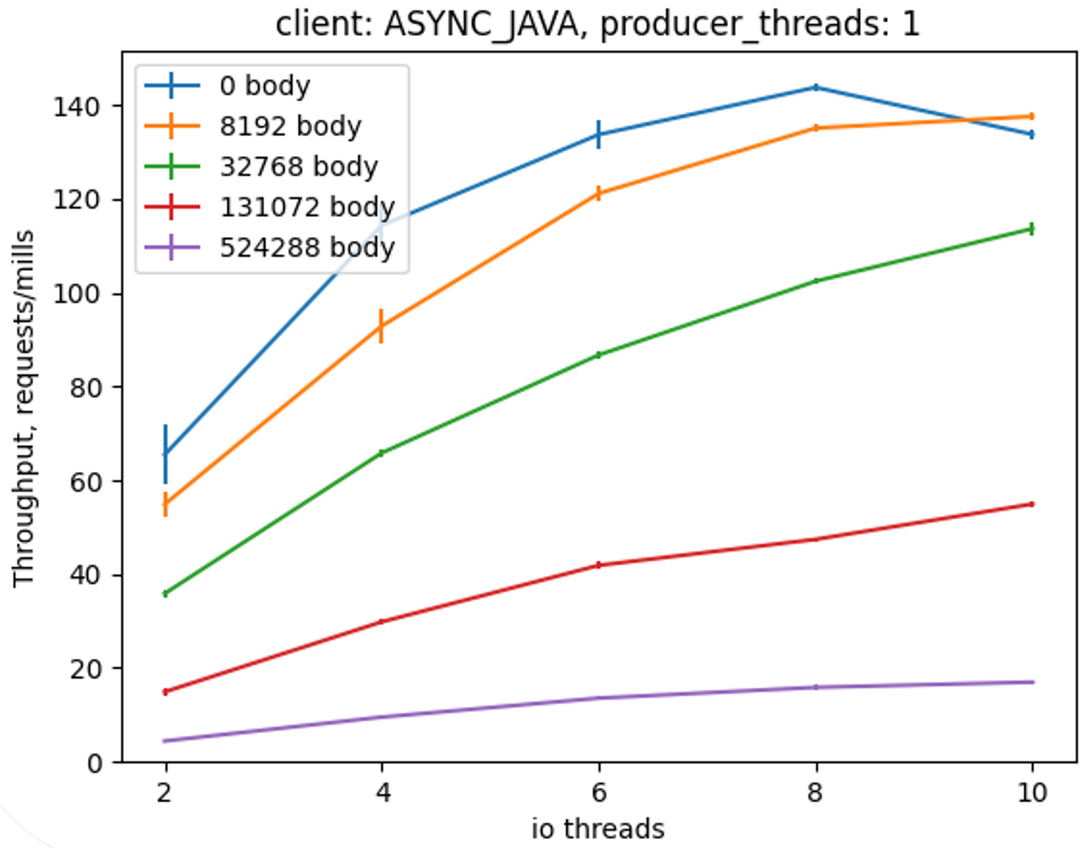
Async Java: producer threads

ASYNC_JAVA: зависимость PRS от producer тредов



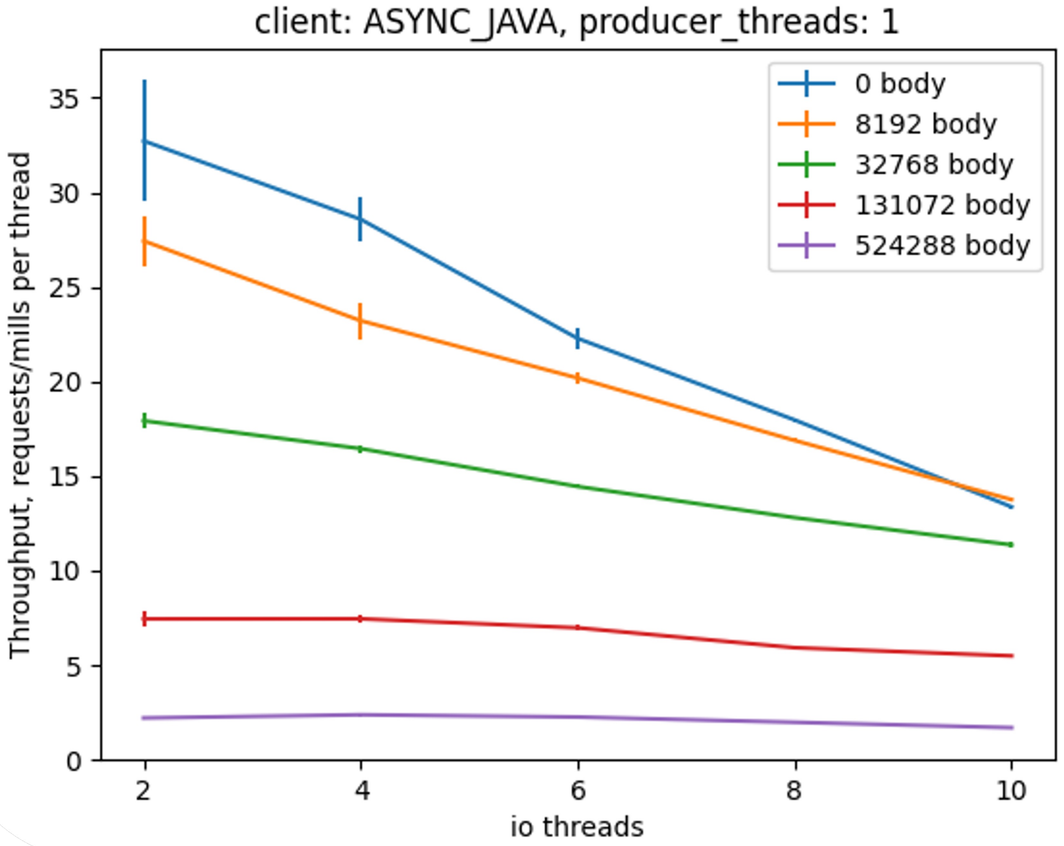


Async Java: IO threads



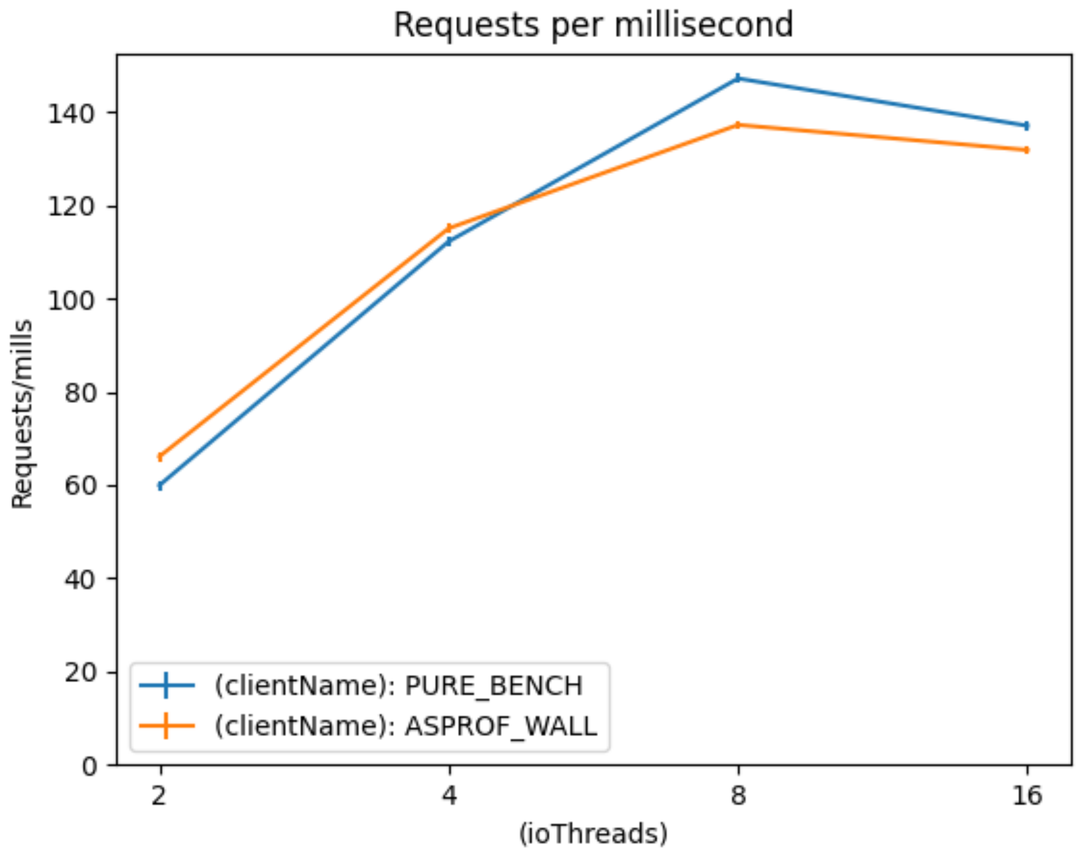
ASYNC_JAVA:
зависимость от IO тредов

Async Java: RPS per IO thread



ASYNC_JAVA:
зависимость от IO тредов,
RPS на IO тред

Async Java: Wall clock profiling



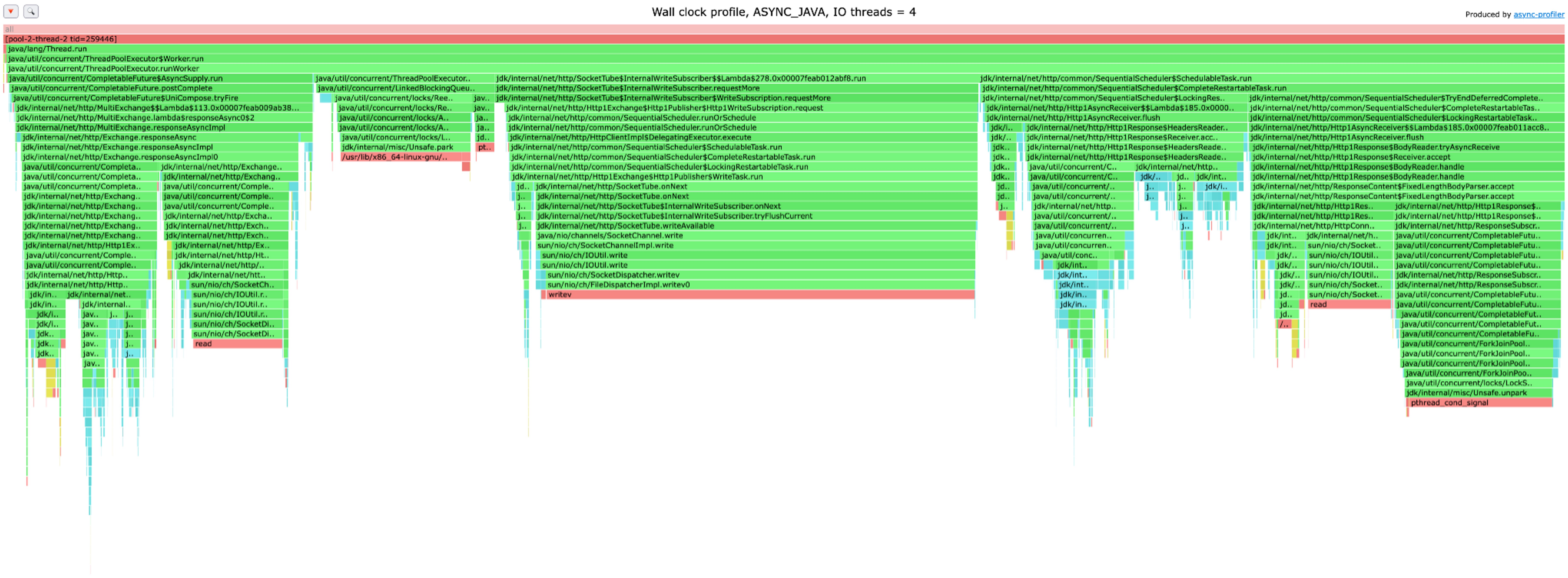
ASYNC_JAVA:
зависимость RPS
от IO тредов

```
asprof -e wall -t \
-o flamegraph \
-d <duration> \
<pid>
```

https://github.com/async-profiler/async-profiler

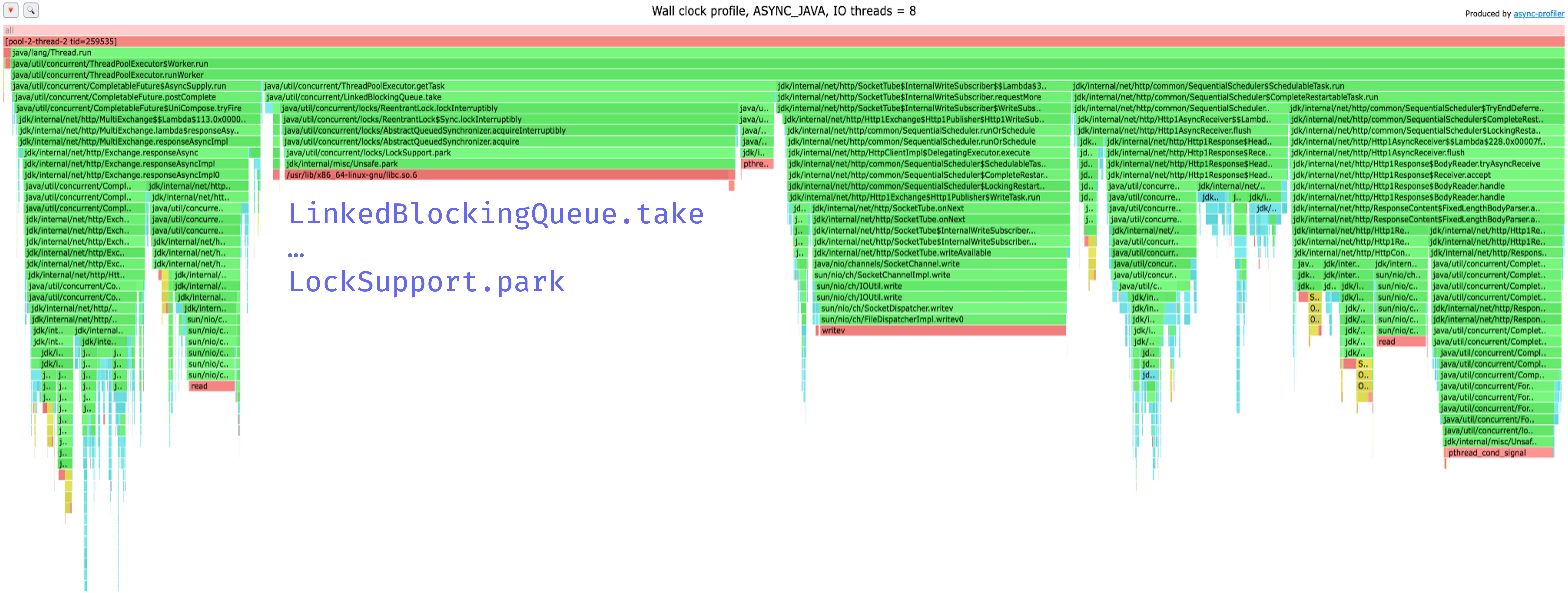
```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

IO threads = 4, pool thread profile



```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

IO threads = 8, pool thread profile

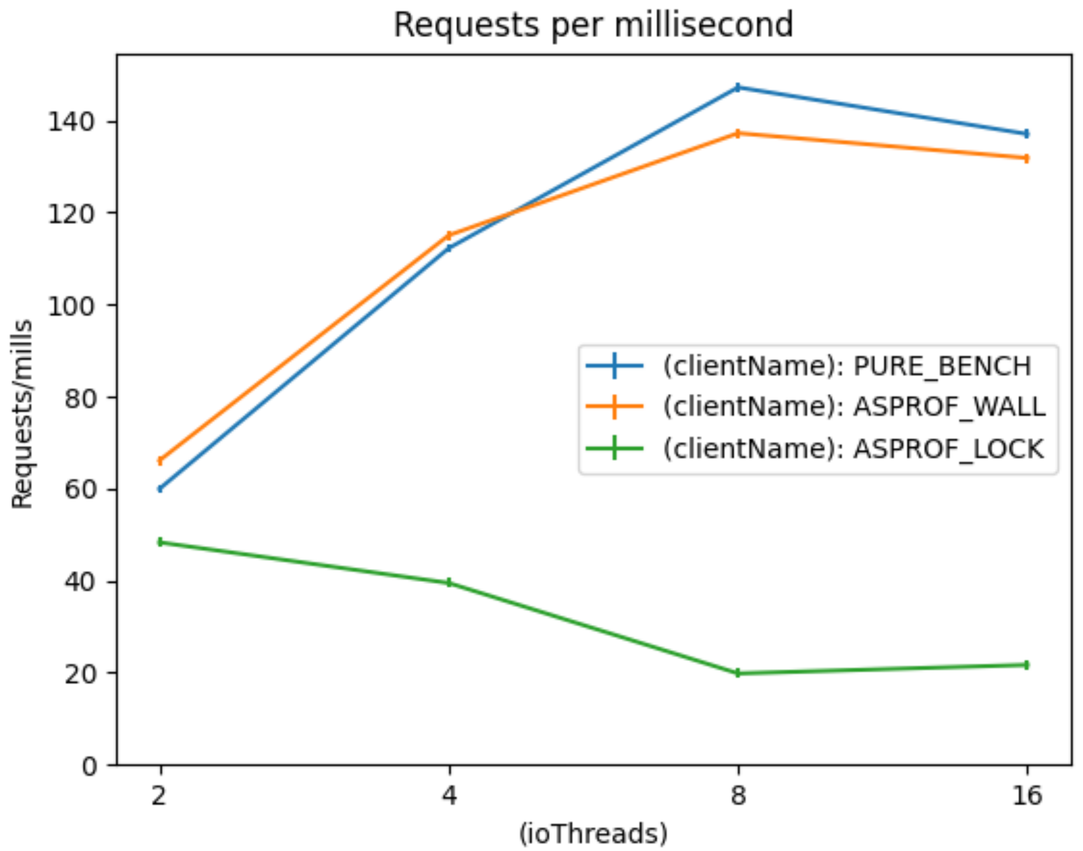


LinkedBlockingQueue.take

…

LockSupport.park

Async Java: Lock profiling



ASYNC_JAVA:
зависимость RPS
от IO тредов

```
asprof -e lock -t \
-o flamegraph \
-d <duration> \
<pid>
```

https://github.com/async-profiler/async-profiler

Async Java: BCC tools

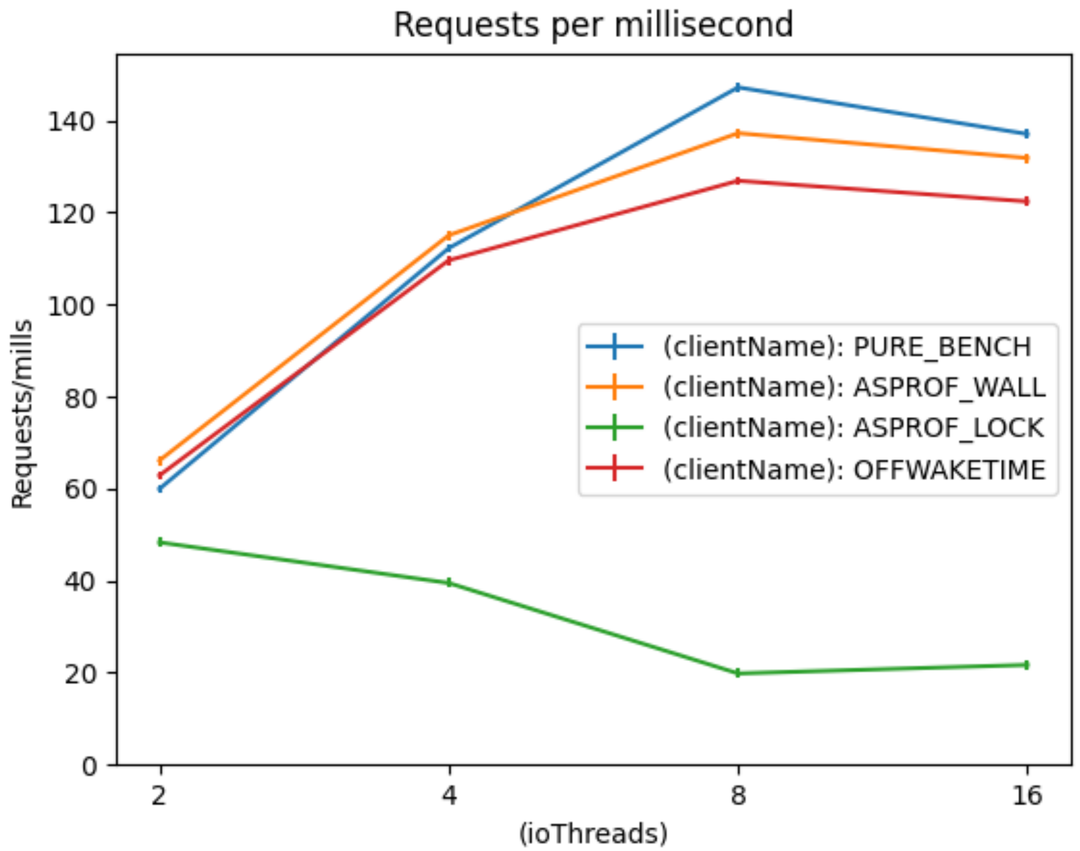
https://github.com/iovisor/bcc

1. IO analysis
2. Networking
3. Monitoring
4. And more…

Нас интересует **offwaketime** —
стектрейсы «разбуженных» и «будящих» потоков:

```
offwaketime-bpfcc -df -p <pid> 20 > out.stacks &&
flamegraph.pl --color=chain --countname=us < out.stacks > out.svg
```

Async Java: offwaketime



ASYNC_JAVA:
зависимость RPS
от IO тредов

Flame graph example

Сверху:

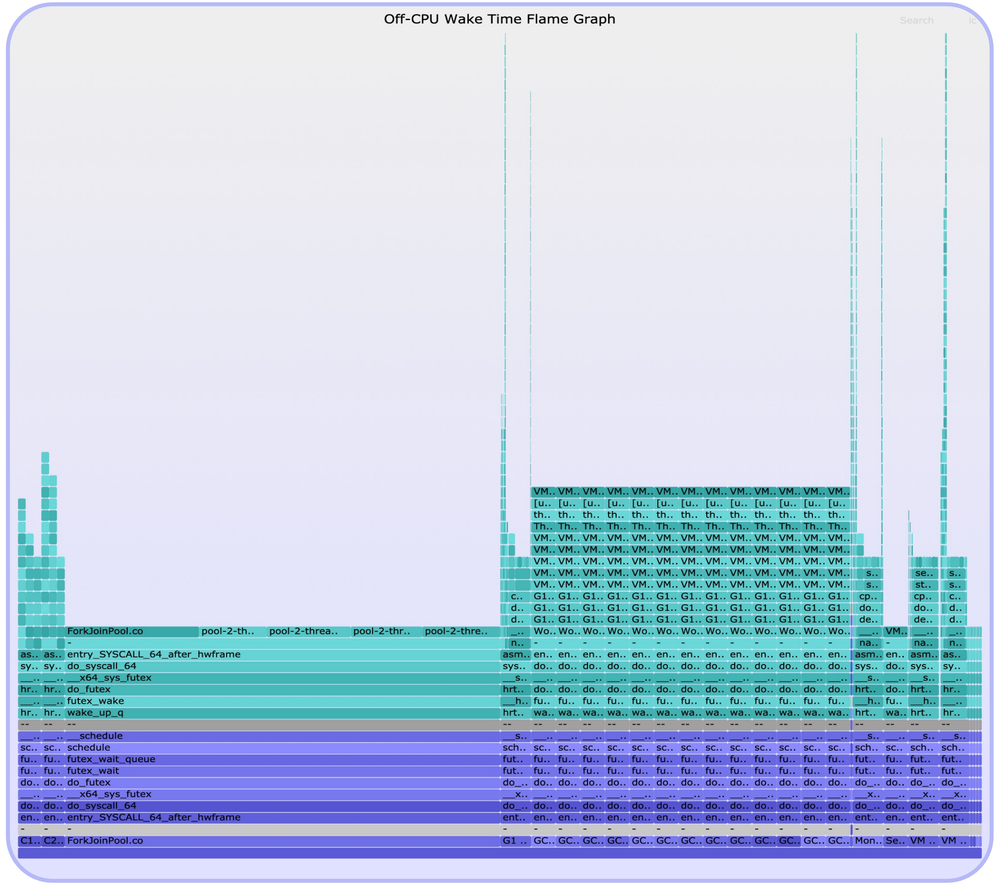
Стектрейсы тех, кто «разбудил» поток

Снизу:

Стектрейс потока в idle, которого разбудили

Горизонтальный размер:

Время в микросекундах, когда idle поток спал



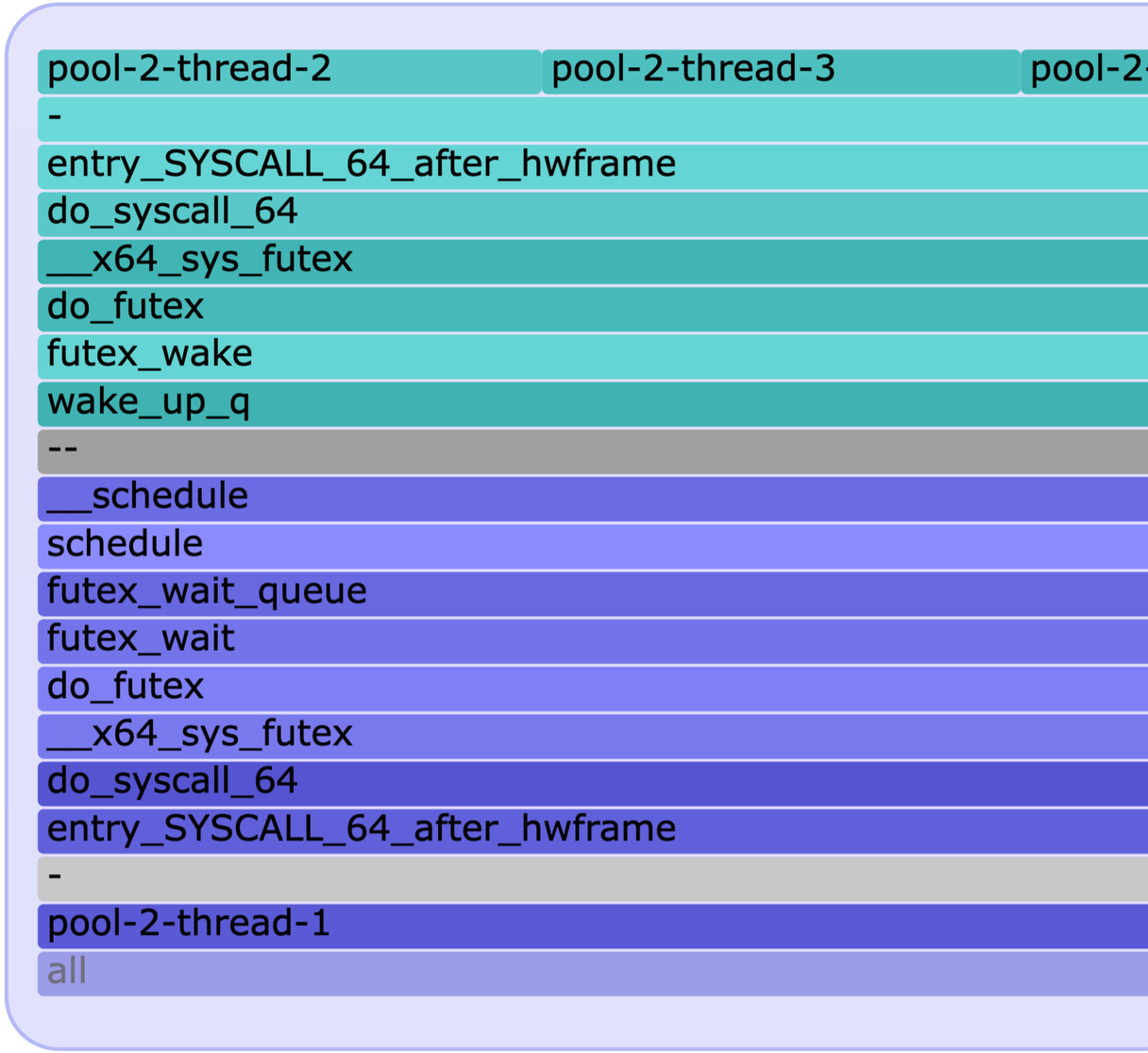
Async Java: offwaketime

Сверху:

Стектрейсы тредов пула,
которые освободили лок

Снизу:

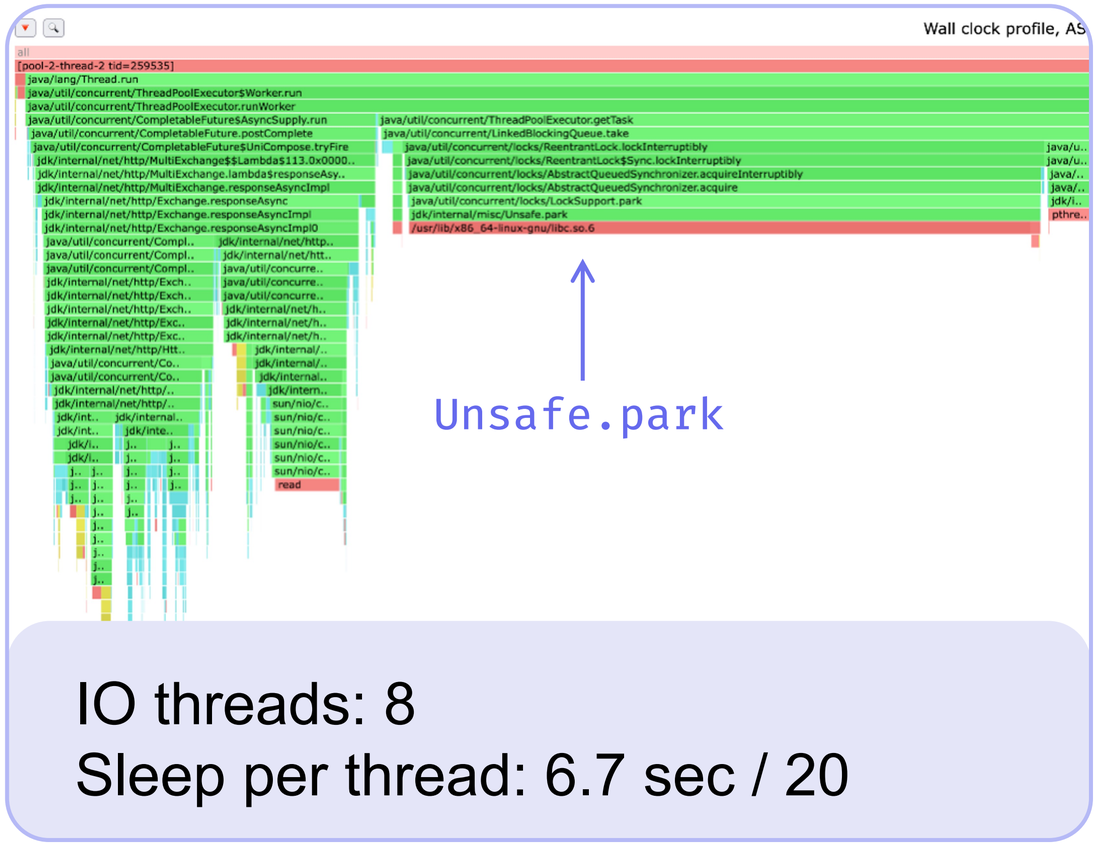
Стектрейс треда пула,
который ждет лок



Async Java: sleeping threads

Thread lock waiting time: 4 IO threads vs 8 IO threads

IO threads: 4
Sleep per thread: 1.8 sec / 20



IO threads: 8
Sleep per thread: 6.7 sec / 20

Async Java: summary

Async Java Client:

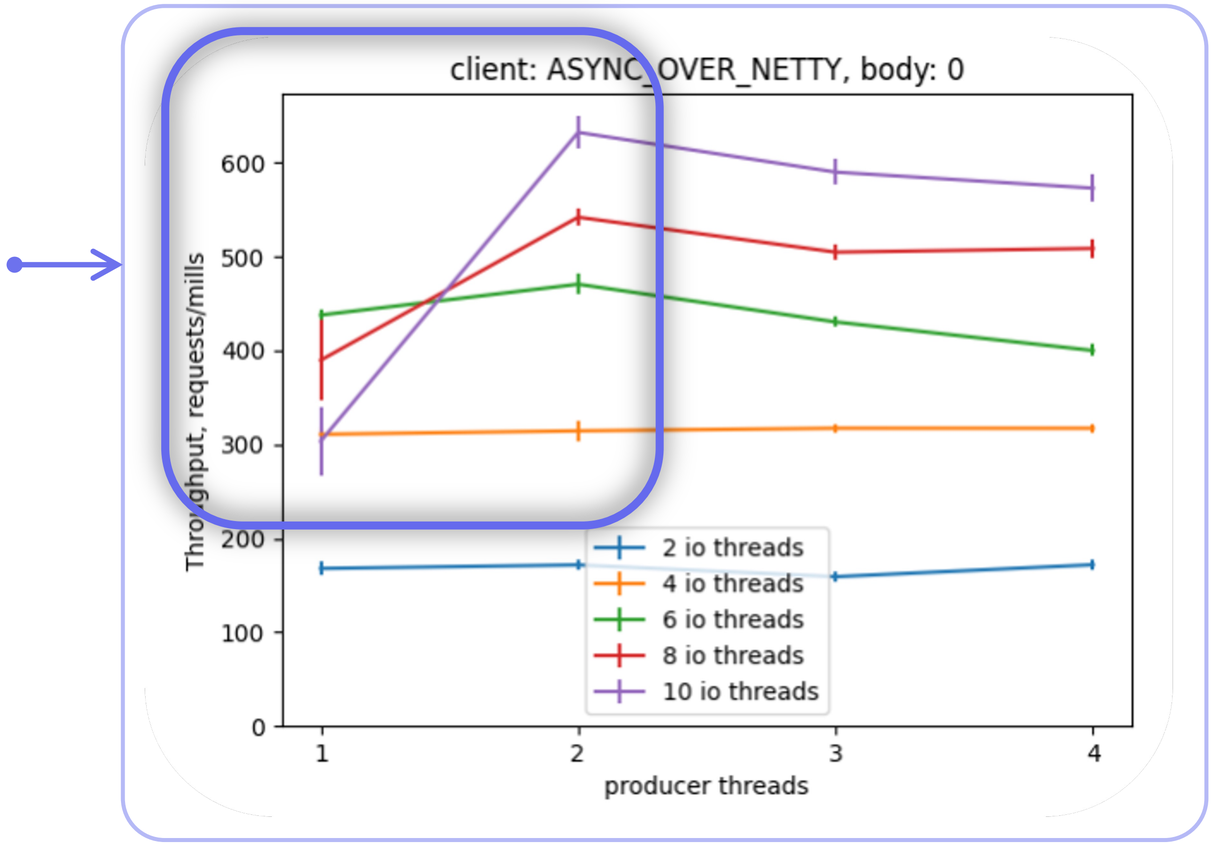
- Не зависит от числа producer-тредов
- IO треды контендятся на take lock-е

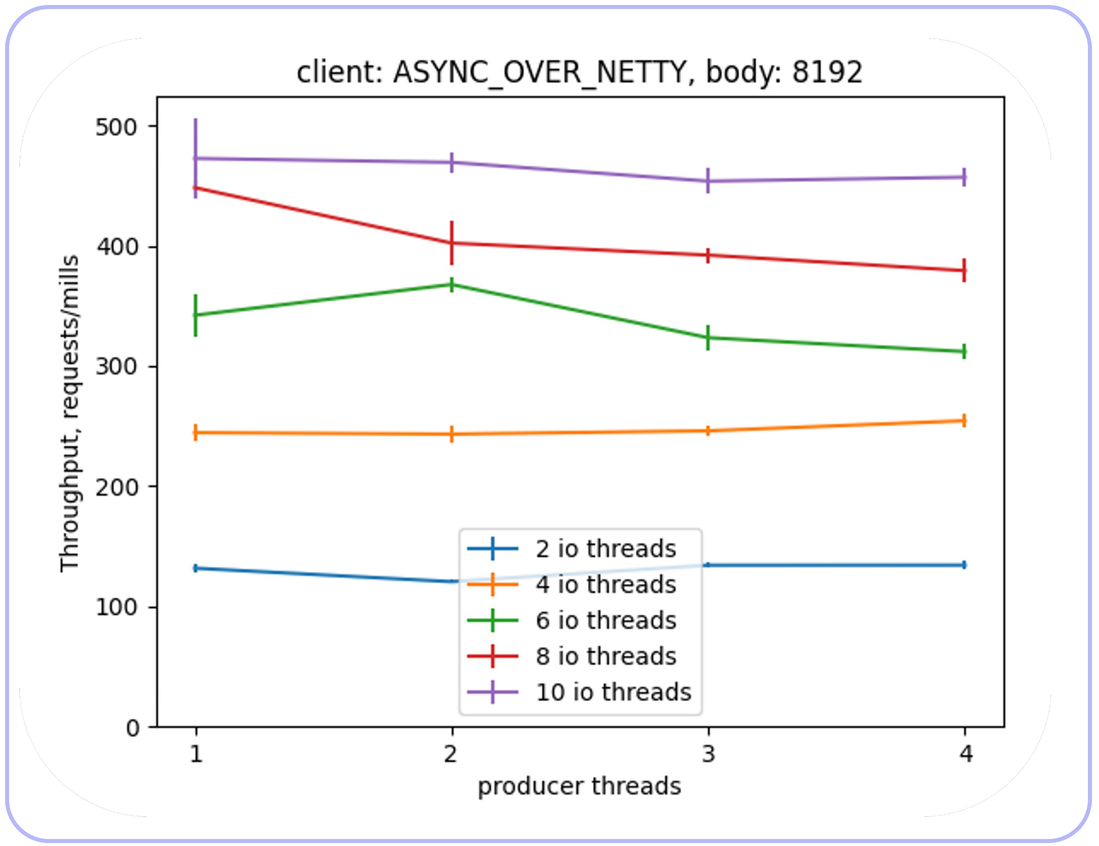
Анализ:

- Профилировать локи может быть очень дорого
- BCC tools могут с этим помочь

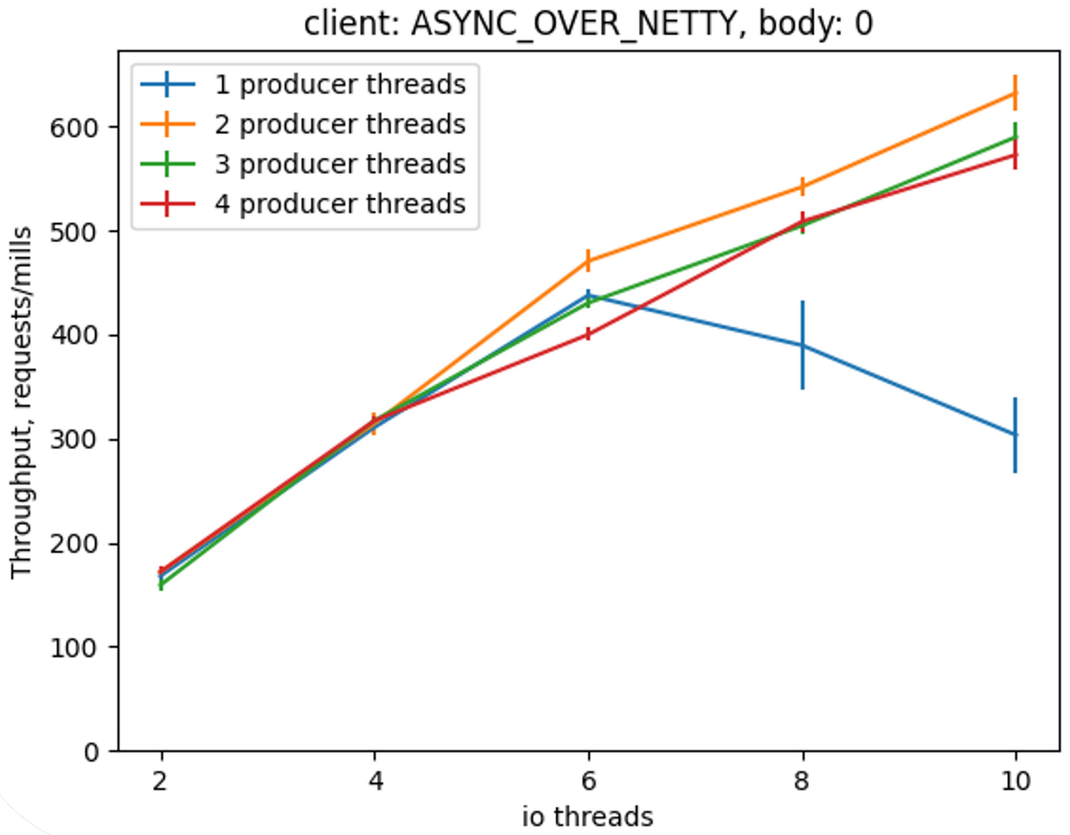
Async over Netty: producer threads

ASYNC_OVER_NETTY, зависимость PRS от producer тредов



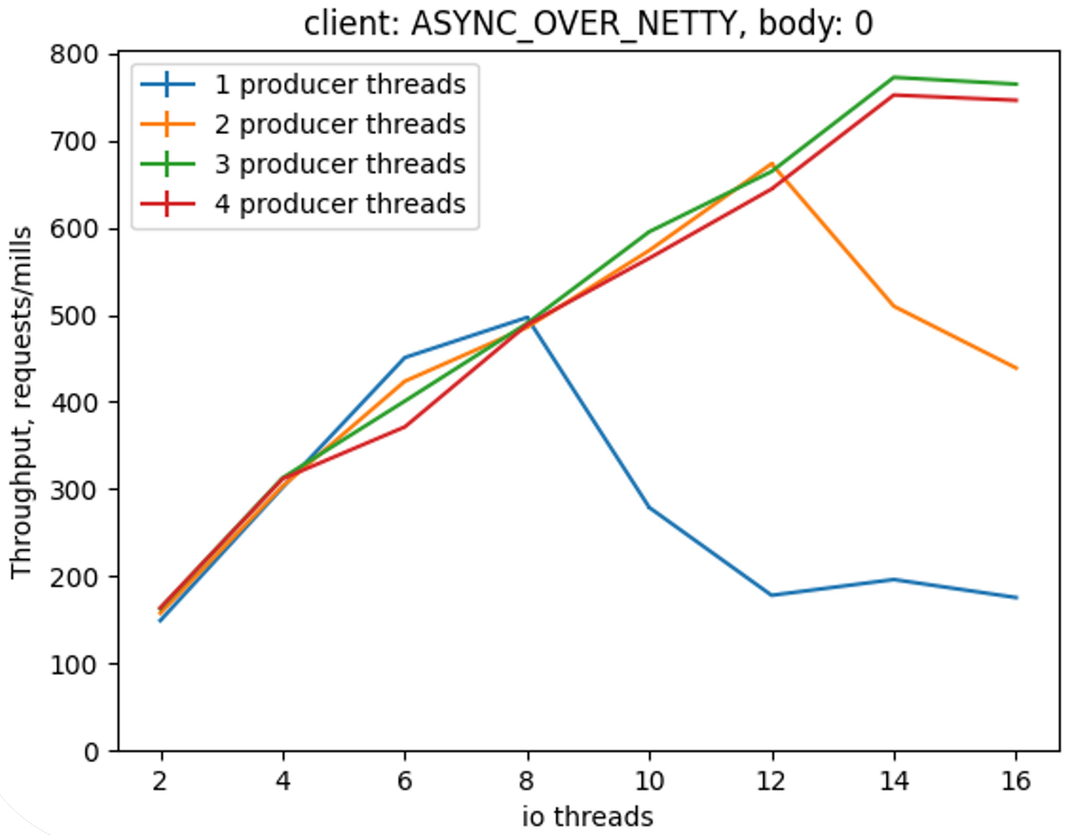


Async over Netty: IO threads



ASYNC_OVER_NETTY:
зависимость от IO тредов

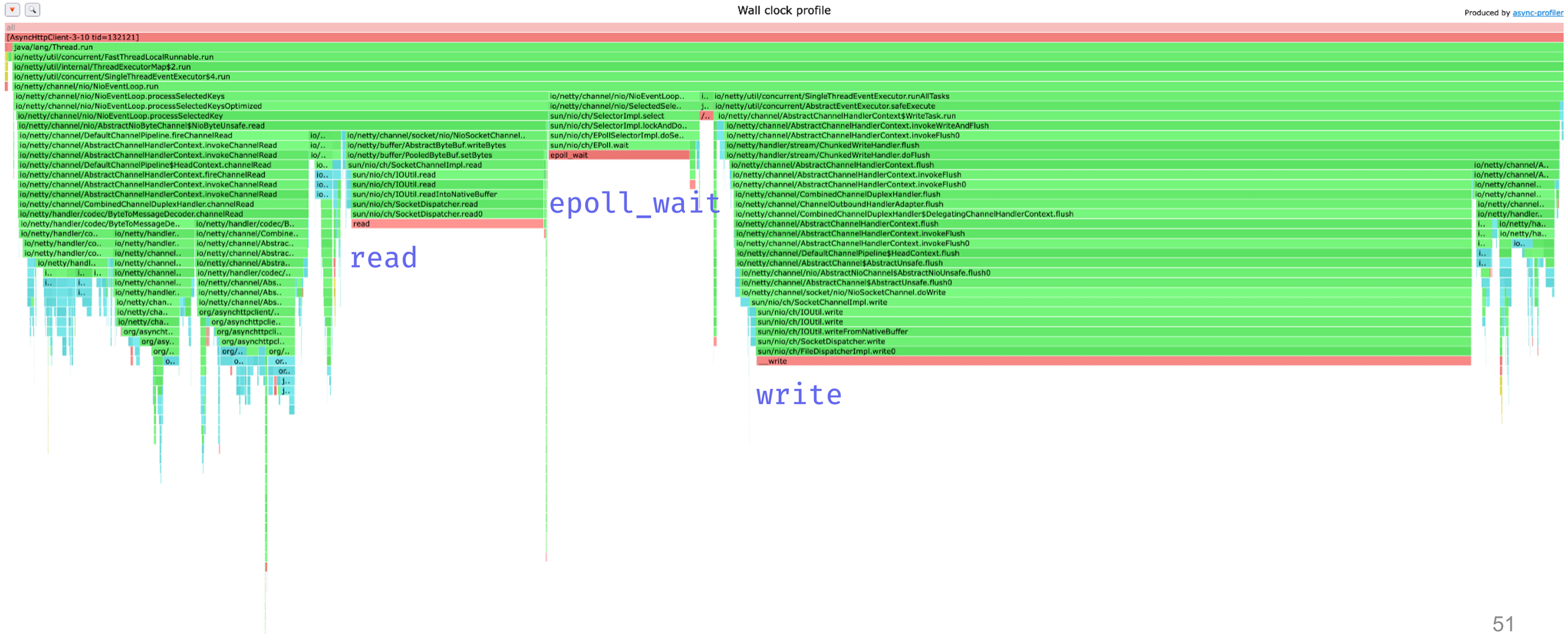
Async over Netty: Analysis



ASYNC_OVER_NETTY:
зависимость от IO тредов

```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

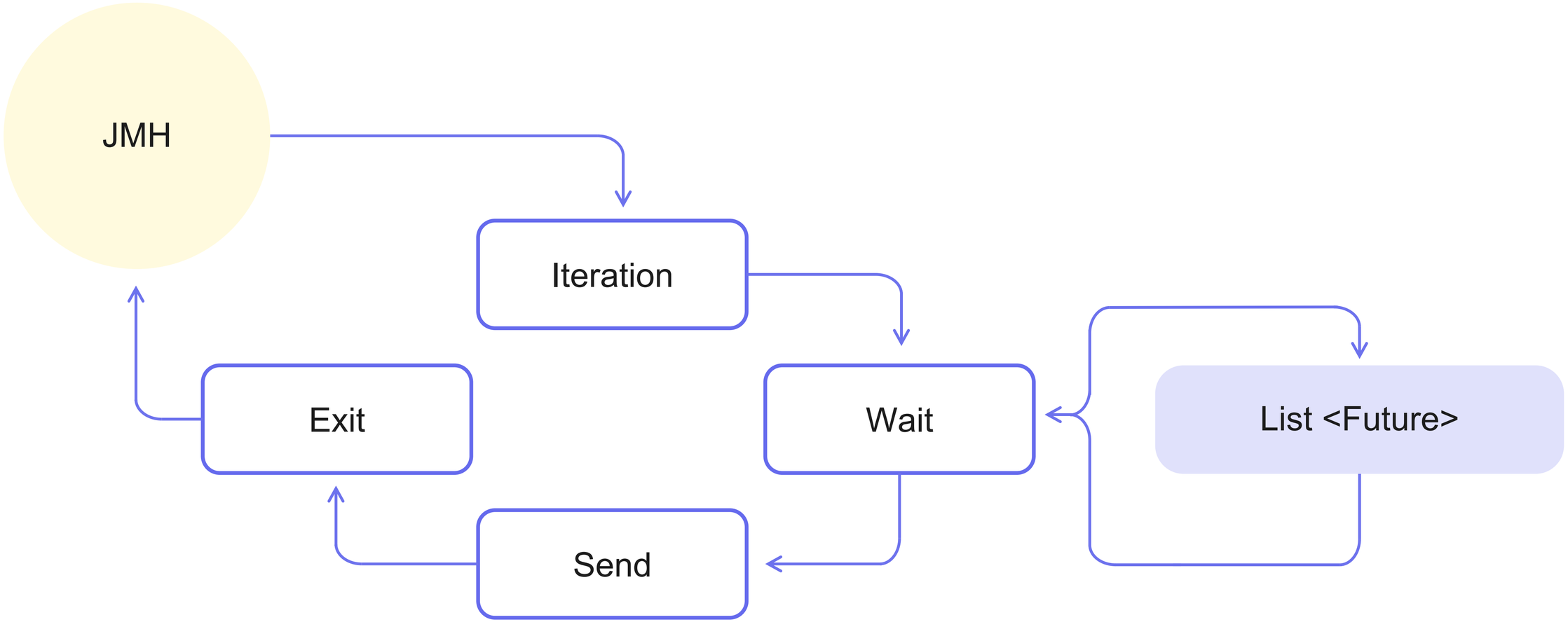
IO threads = 10, producers = 2, IO thread profile



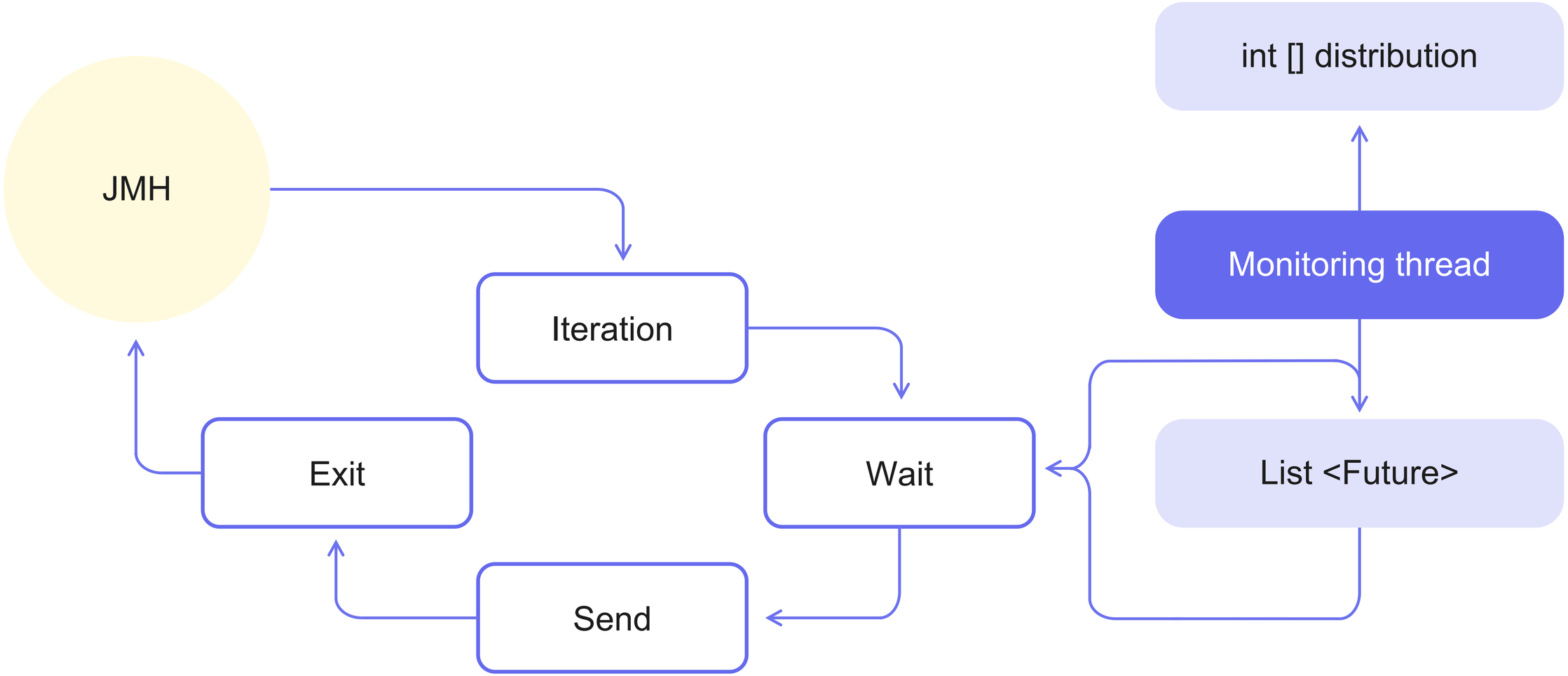
```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

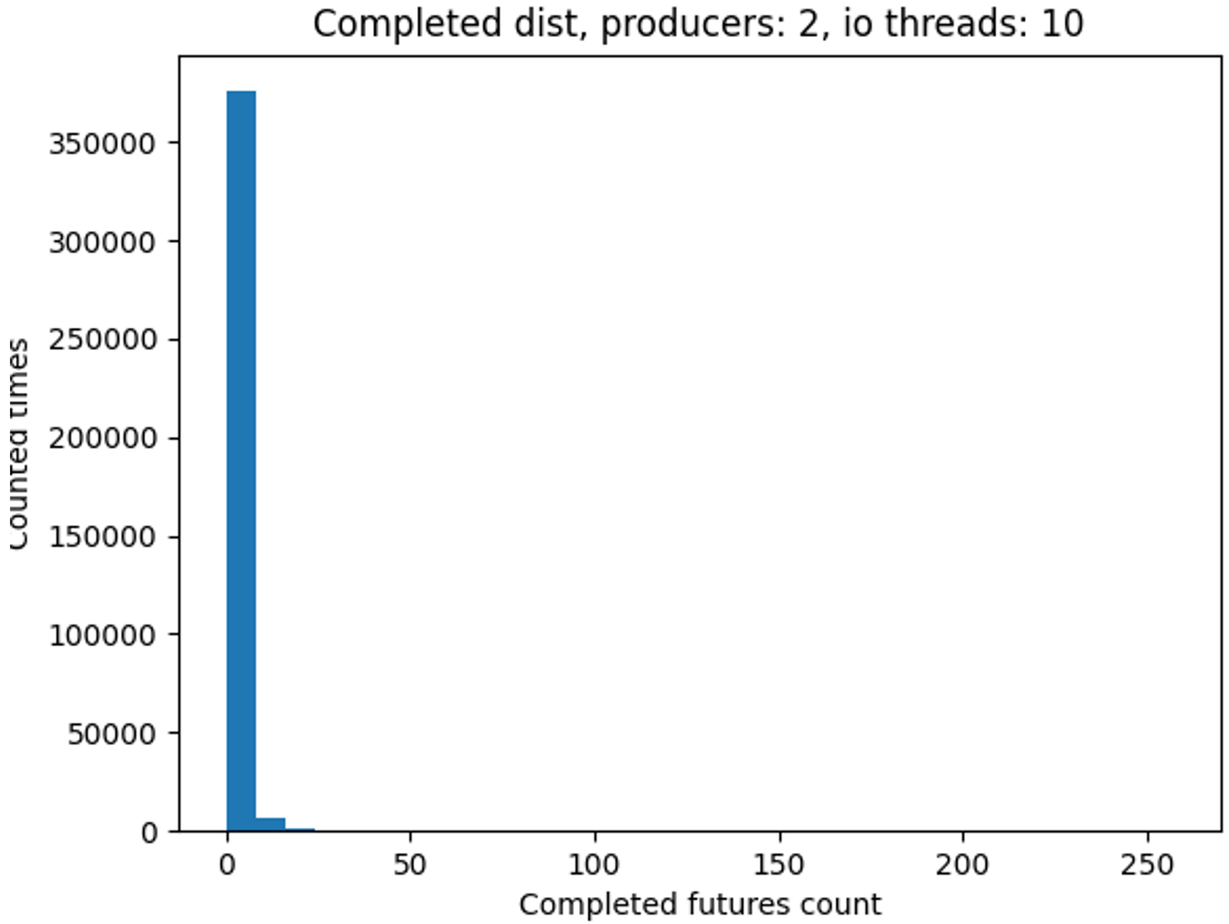
IO threads = 10, producers = 1, IO thread profile

Benchmark: Reminder



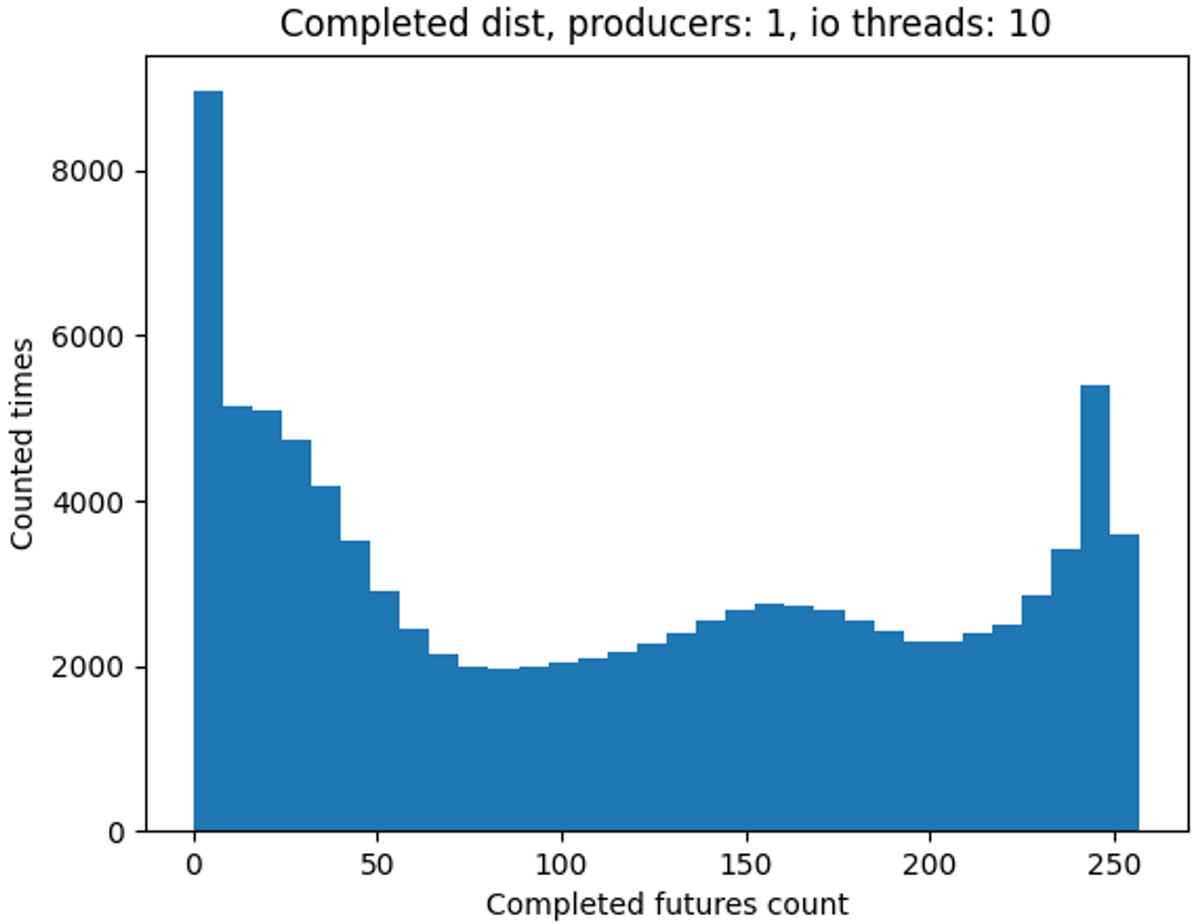
Benchmark: Monitoring thread





Распределение
завершенных,
но не замененных
запросов

```
producers = 2,
io threads = 10
```

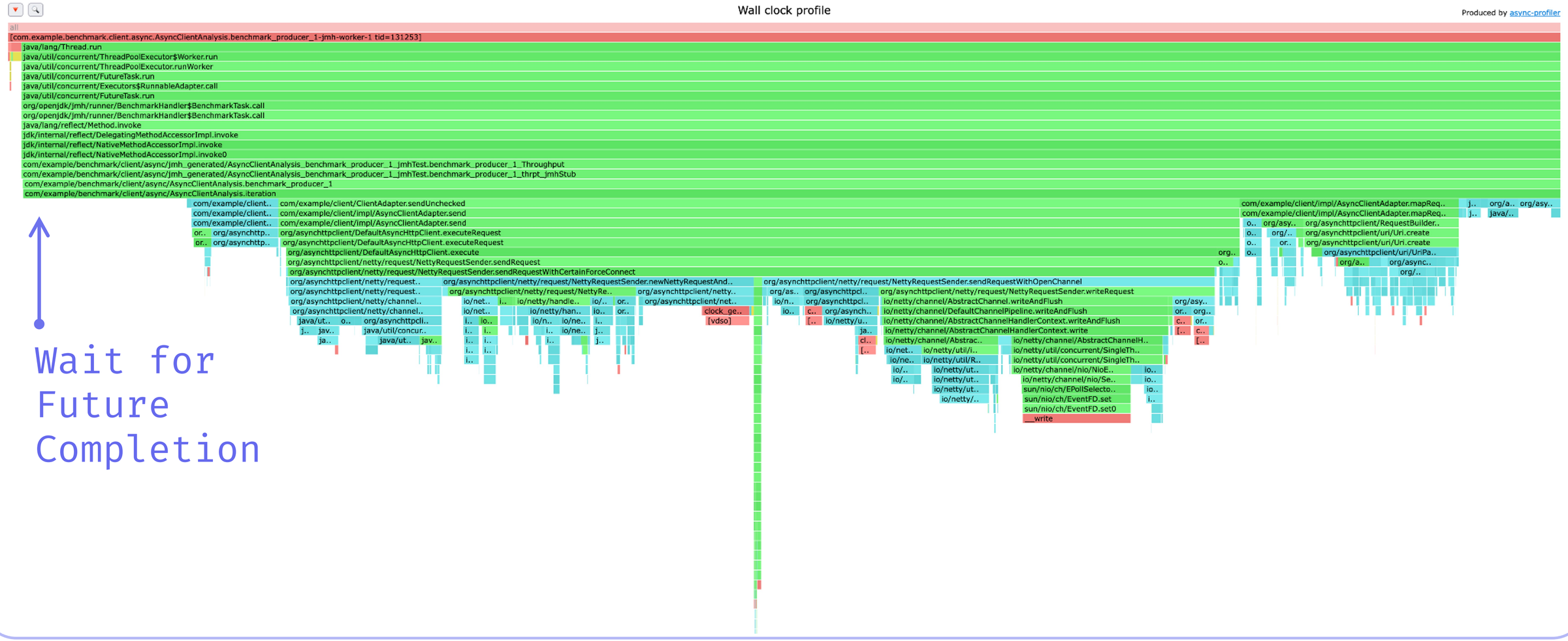


Распределение
завершенных,
но не замененных
запросов

```
producers = 1,
io threads = 10
```

```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

IO threads = 10, producers = 2, IO thread profile



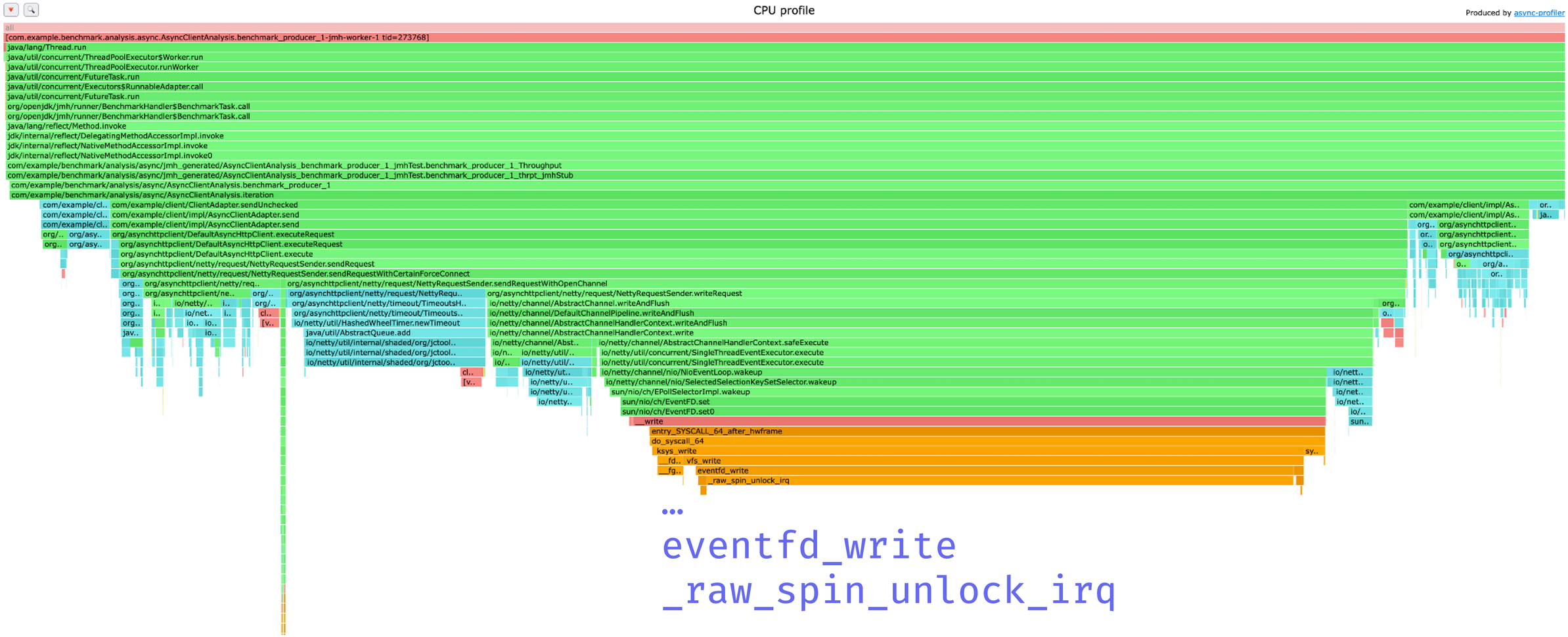
```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

IO threads = 10, producers = 1, IO thread profile

…
EventFD.set0
__write

```
./asprof -e cpu -t -o flamegraph -d <duration> <pid>
```

IO threads = 10, producers = 1, IO thread profile



…

eventfd_write

_raw_spin_unlock_irq

Producer thread

когда делает запрос, тратит ресурсы
на то, чтобы разбудить IO тред,
если тот в ожидании

`eventfd_write,`
`producer thread`

`epoll_wait,`
`io thread`

Async over Netty: summary

Async Netty-based Client:

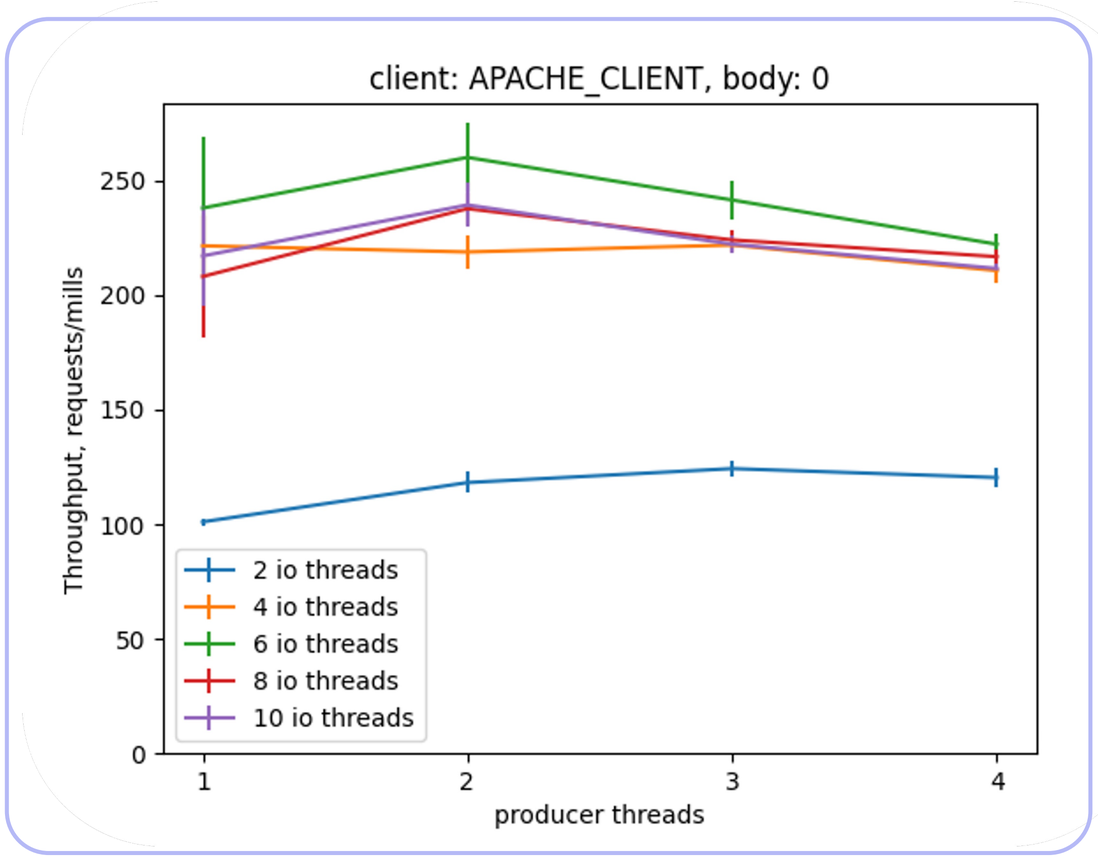
- Может страдать
  от избытка ресурсов
- Использует producer-треды,
  чтобы делать часть работы

Анализ:

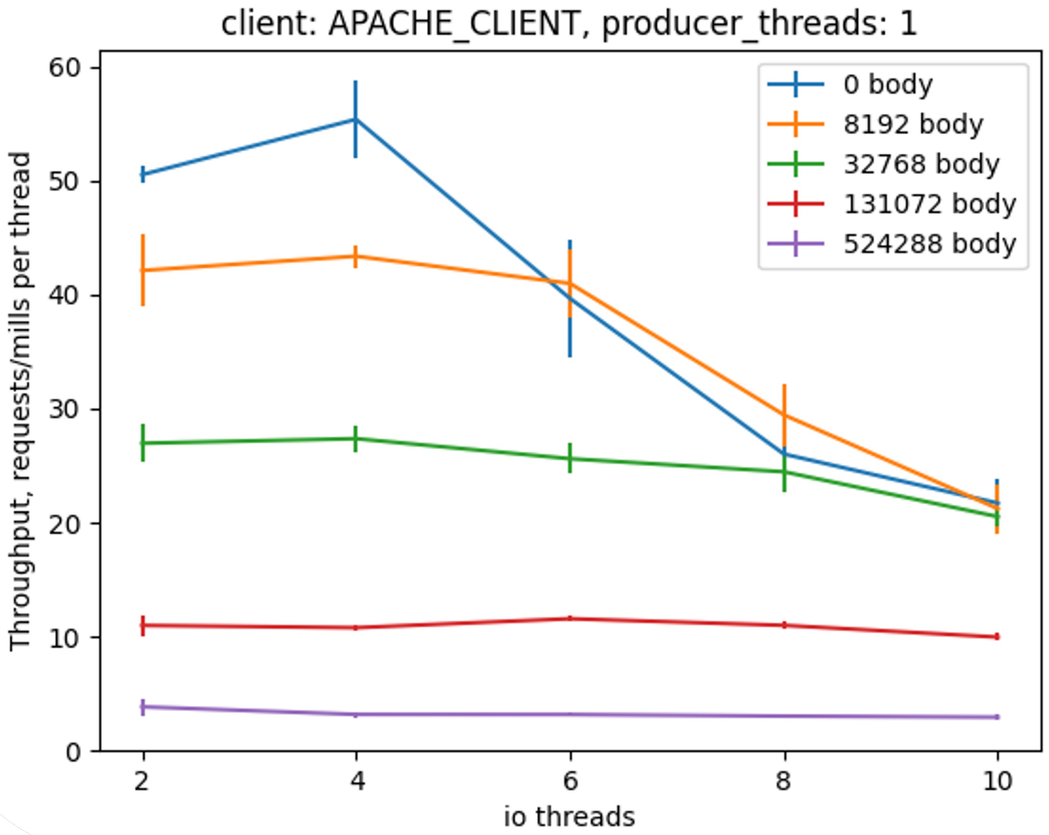
- Собирать данные можно
  и ручками, без инструментов
- CPU профиль тоже
  может быть полезен
- BCC tools
  помогают посмотреть
  на взаимодействие потоков

Async Apache: producer threads

ASYNC_APACHE, зависимость PRS от producer тредов



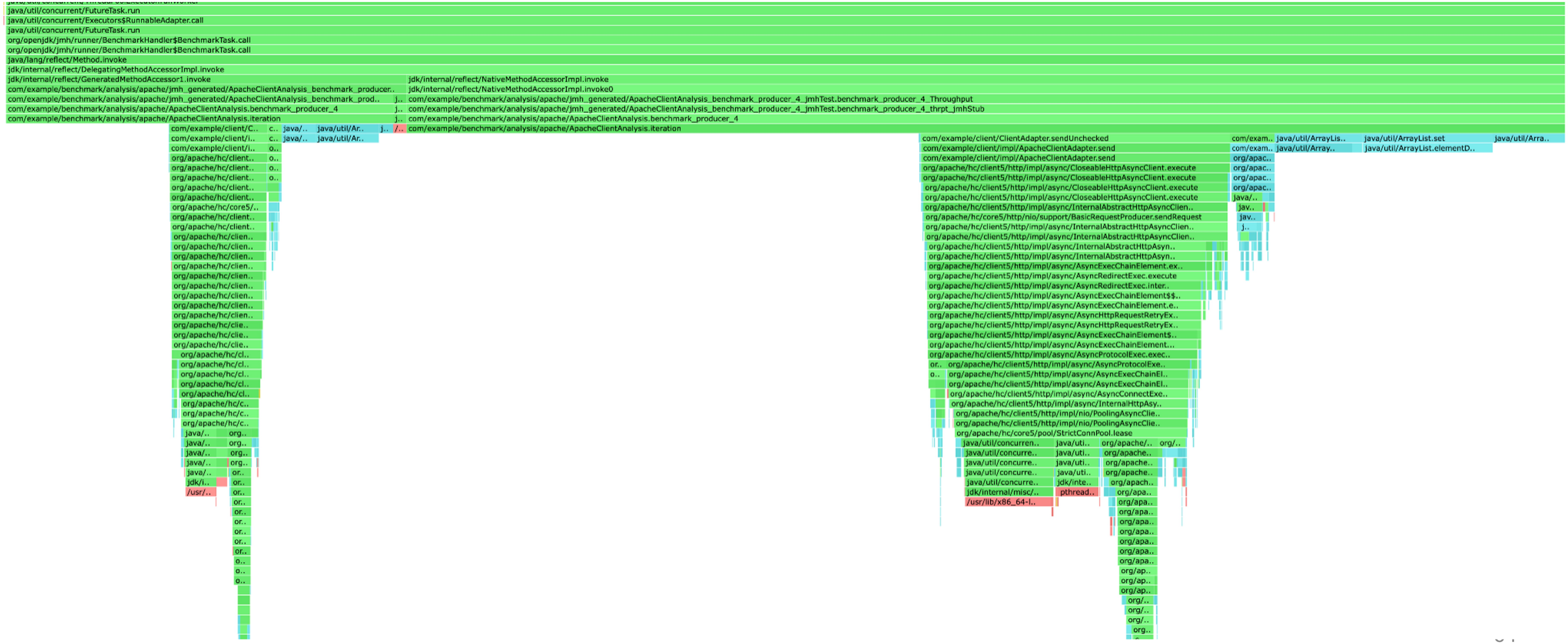
Async Apache: IO threads



ASYNC_APACHE:
зависимость от IO тредов,
RPS на IO тред

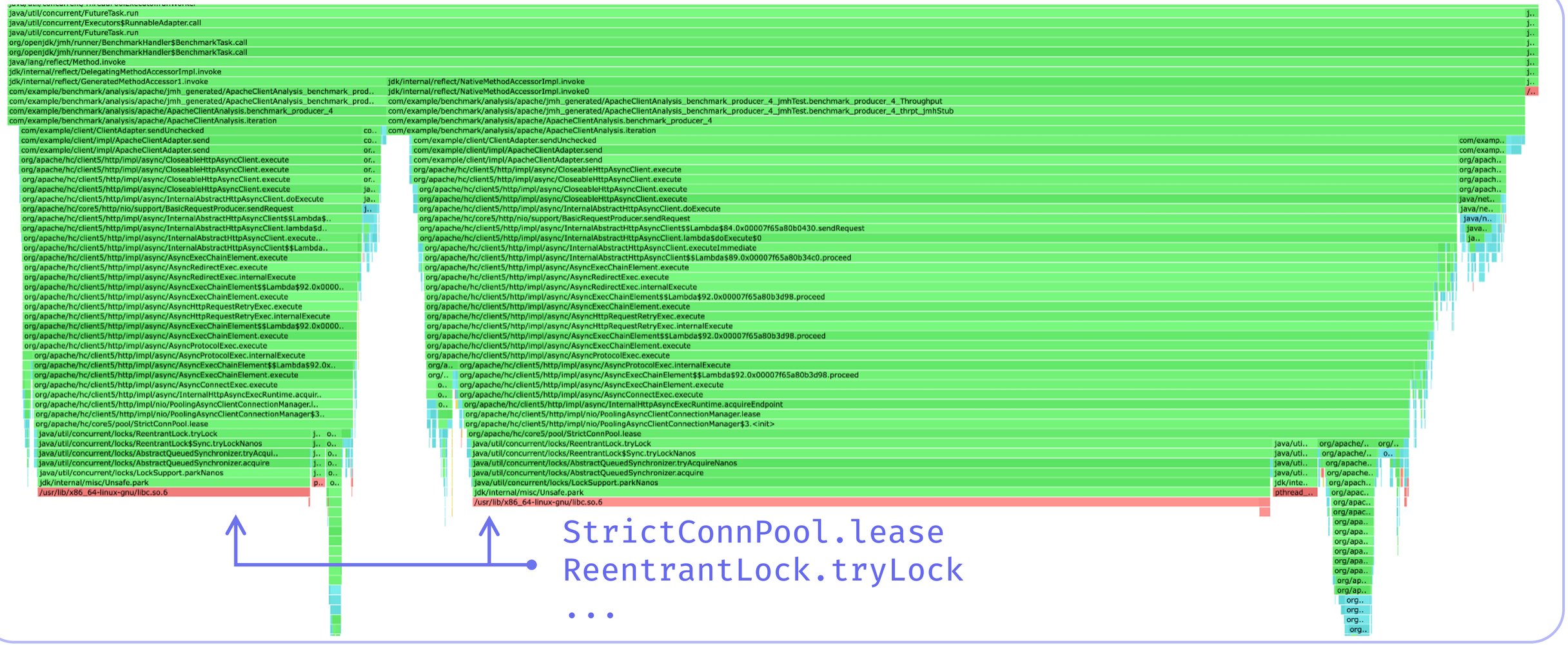
```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

IO threads = 4, producers = 4, Producer thread profile

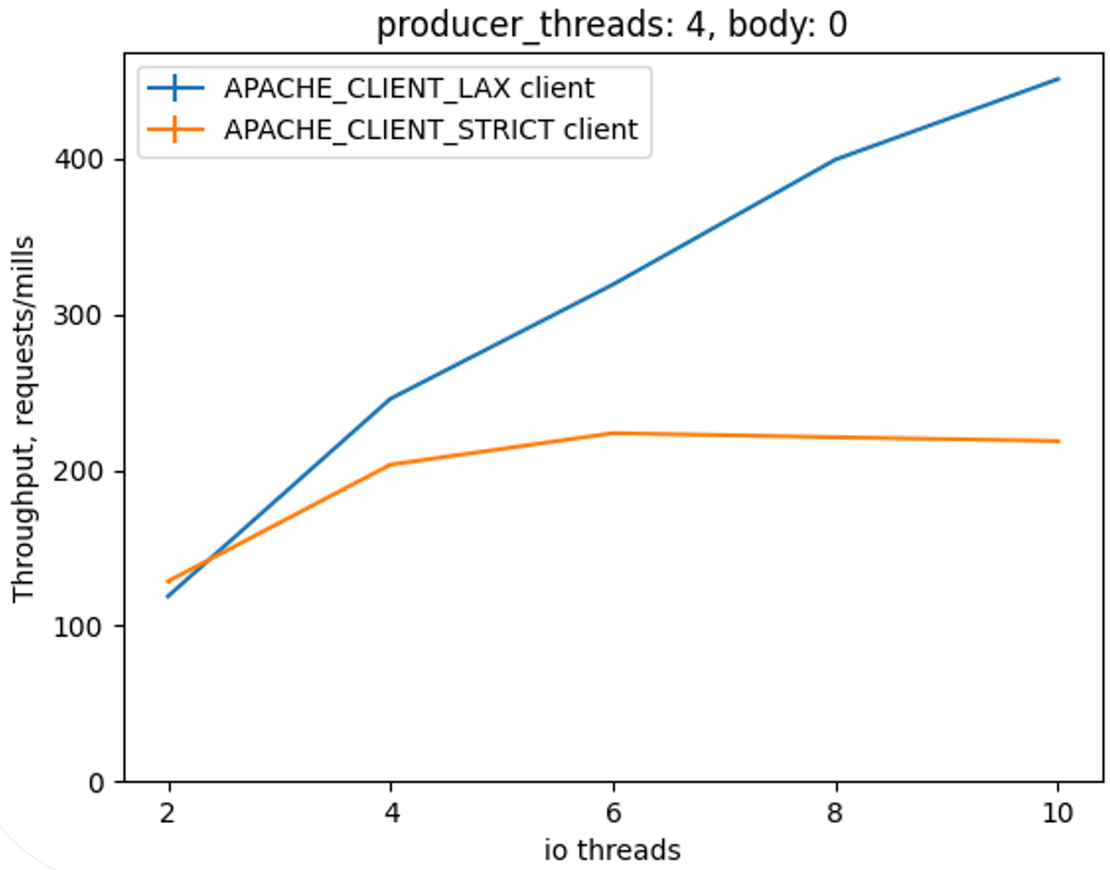


```
./asprof -e wall -t -o flamegraph -d <duration> <pid>
```

IO threads = 8, producers = 4, Producer thread profile



Async Apache: PoolPolicy LAX



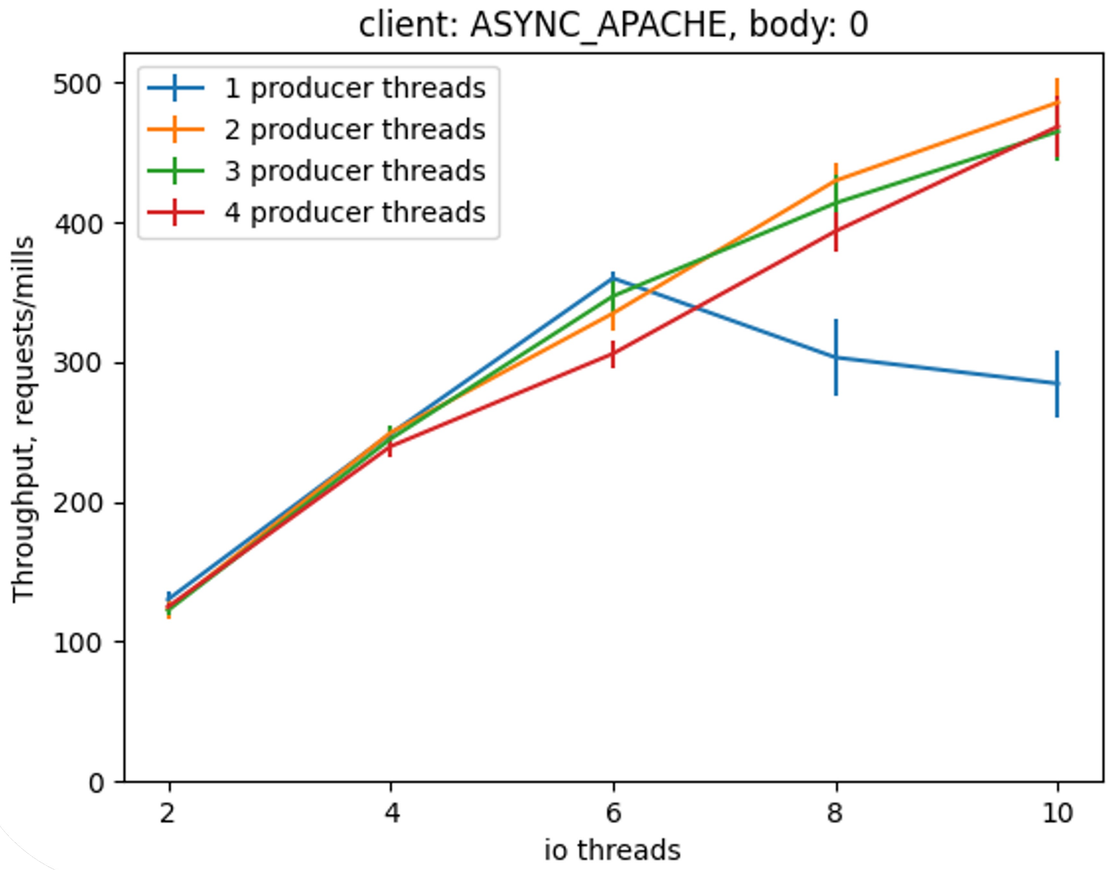
ASYNC_APACHE:
зависимость от IO тредов

PoolConcurrencyPolicy:

- STRICT
- LAX

“Higher concurrency but with lax connection max limit guarantees.”

Async Apache: IO threads updated



ASYNC_APACHE:
зависимость от IO тредов

PoolConcurrencyPolicy: LAX

Async Apache: summary

Async Apache HttpClient5:

В стандартной конфигурации упирается в PoolConnectionPolicy

Для PoolConnectionPolicy.LAX динамика аналогична Netty-based Client

Промежуточная мораль №4

После того, как мы получили какие-то числа:

Изучили, почему числа такие, какие есть

Поставили эксперименты, собрали данные, попрофилировали

Воспользовались системными тулзами, они много умеют

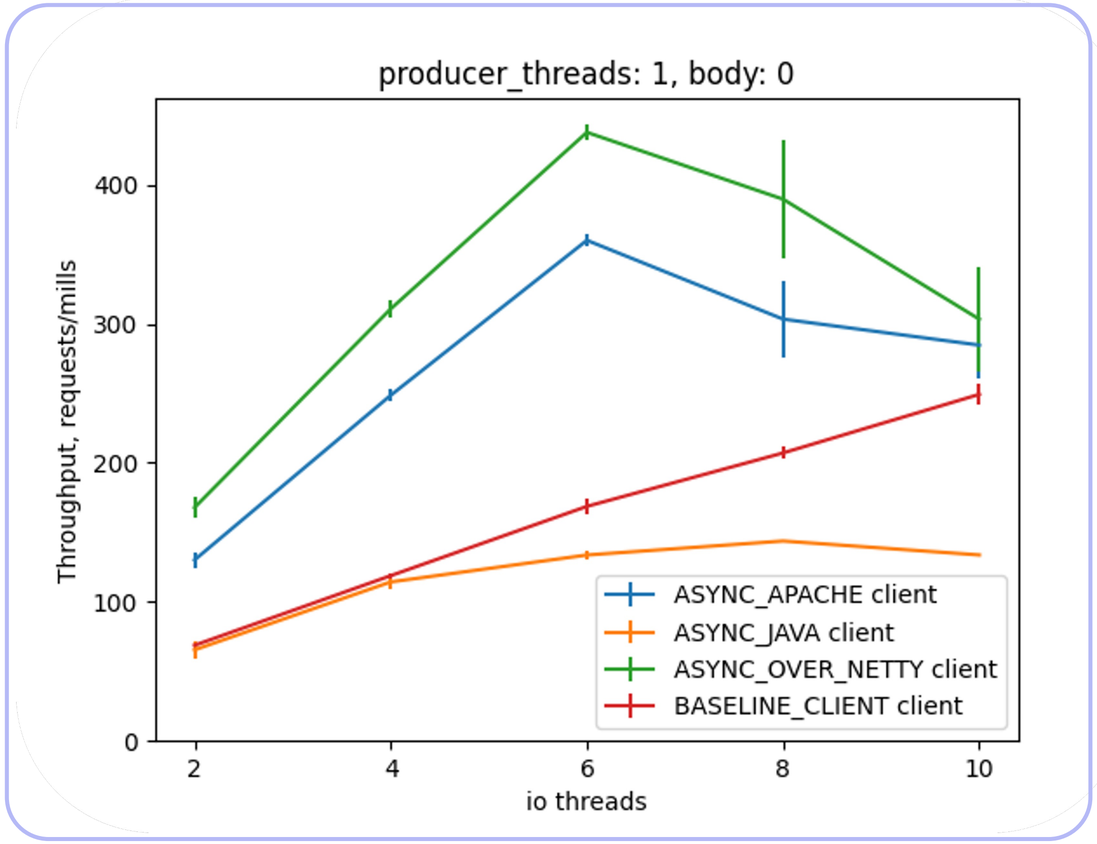
“REMEMBER: The numbers below are just data.” — JMH output

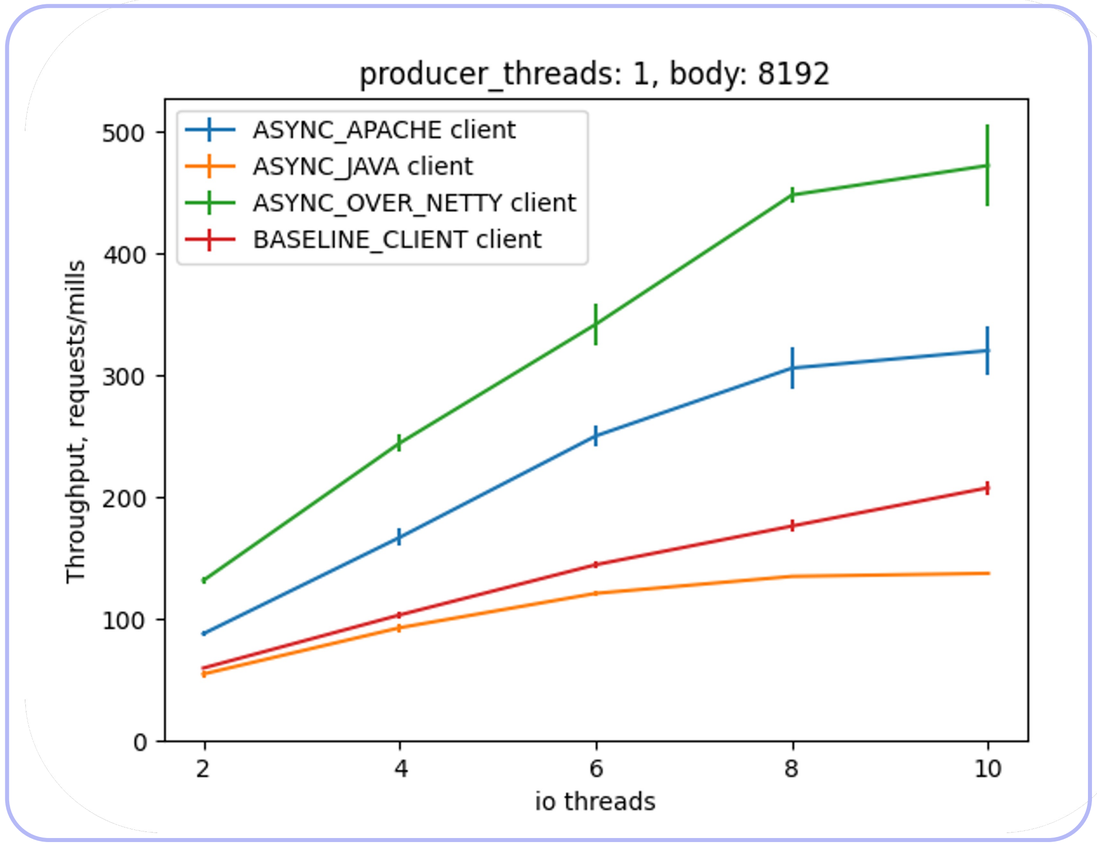
Сравнение клиентов

05

Single producer

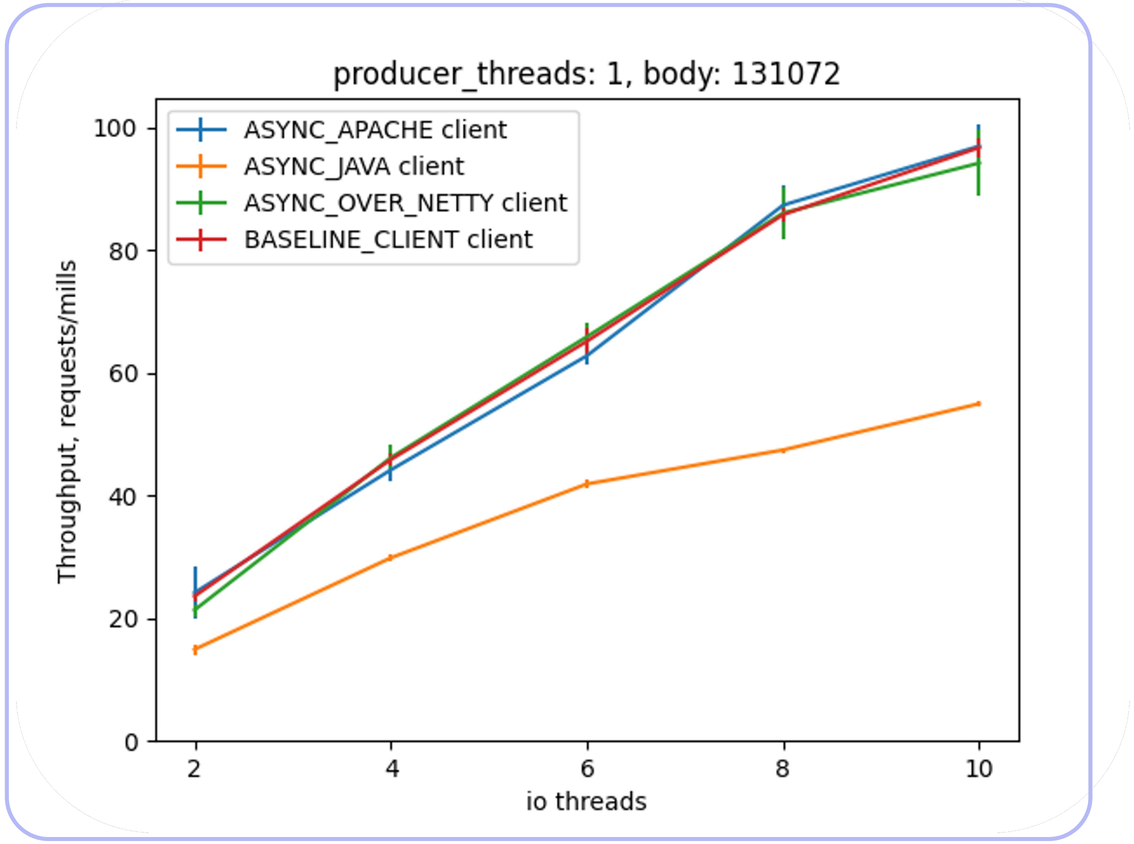
Разные клиенты в зависимости от IO threads, небольшие запросы

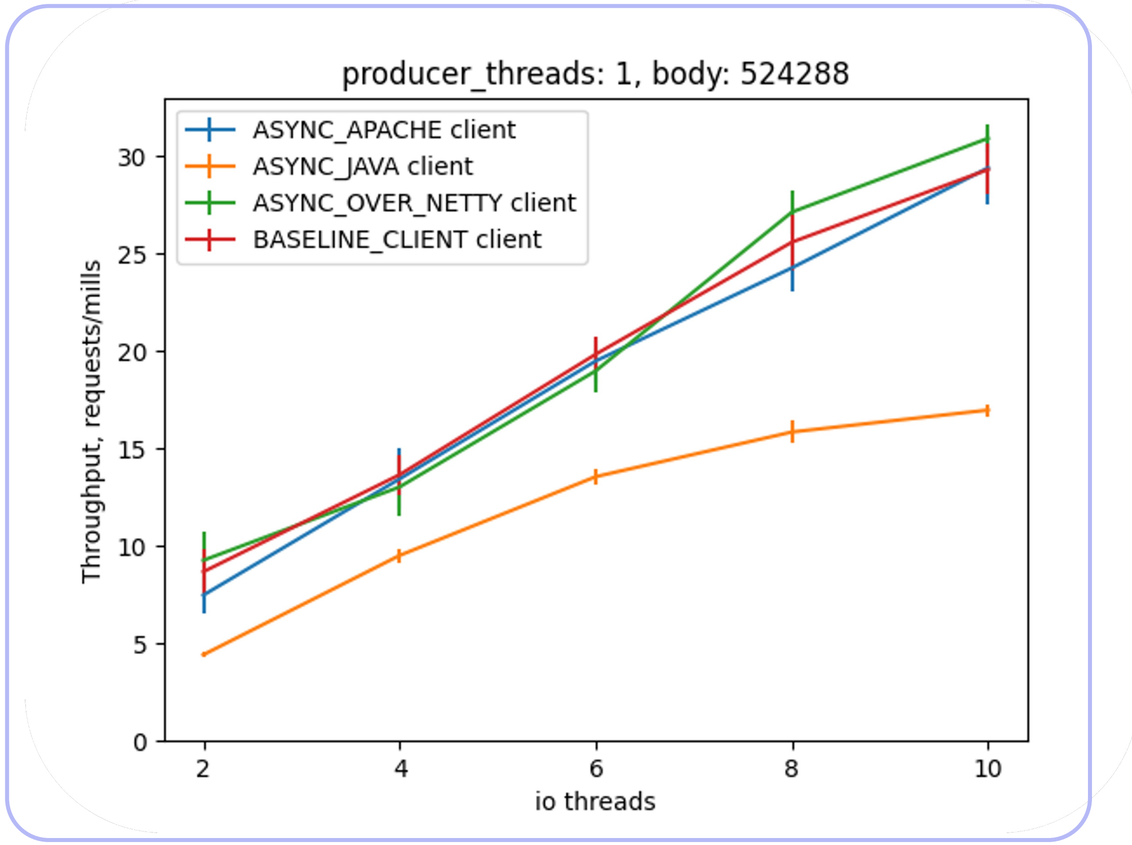




Single producer

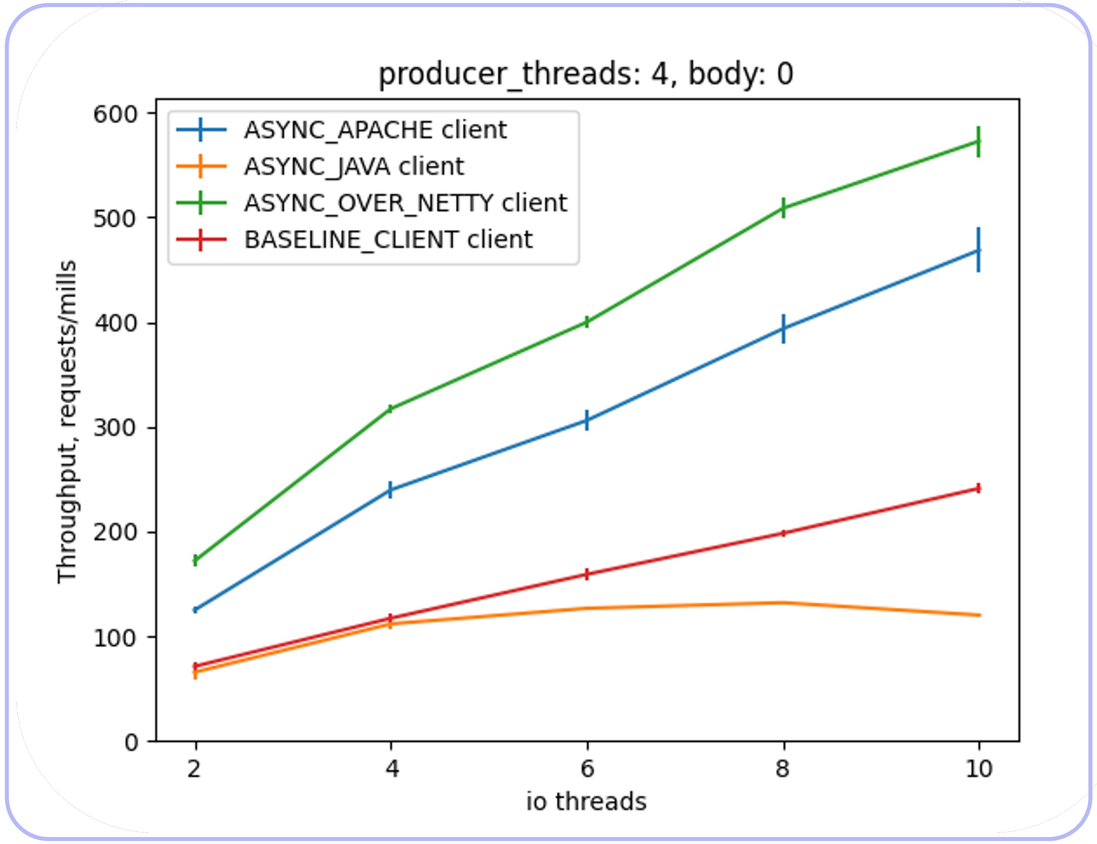
Разные клиенты в зависимости от IO threads, большие запросы

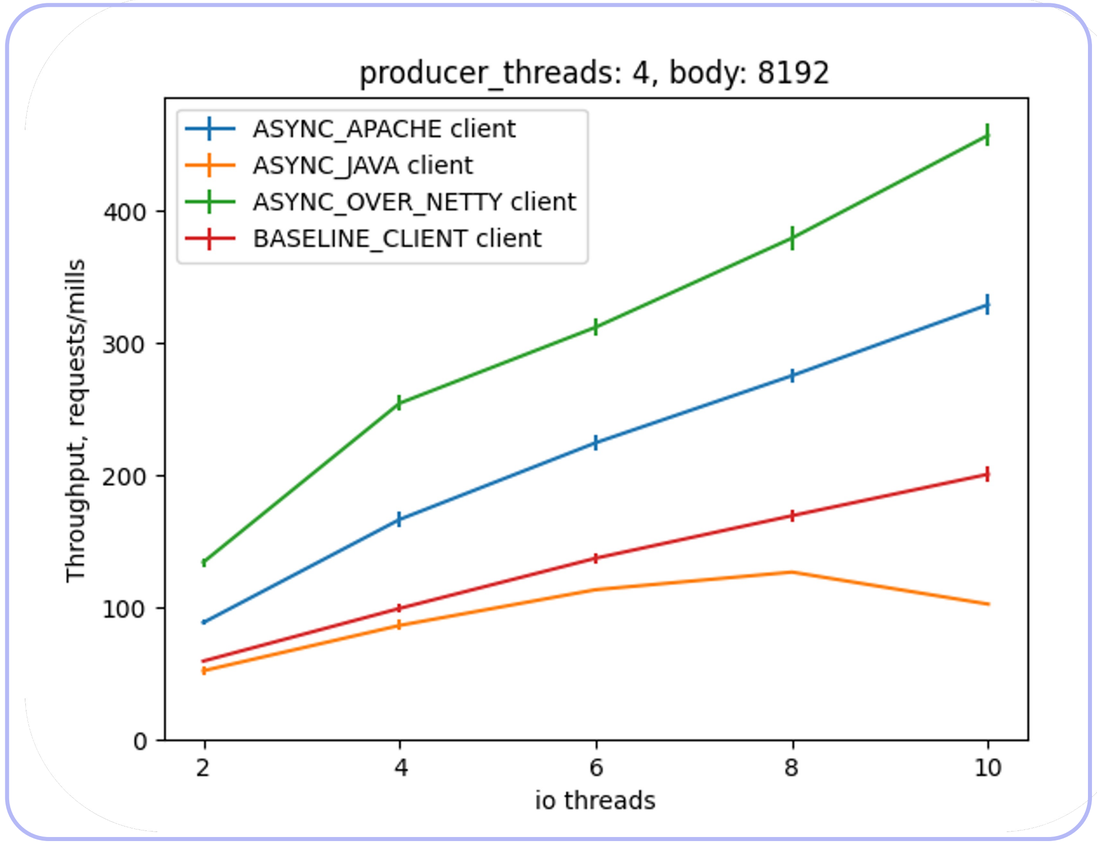




Multiple producers

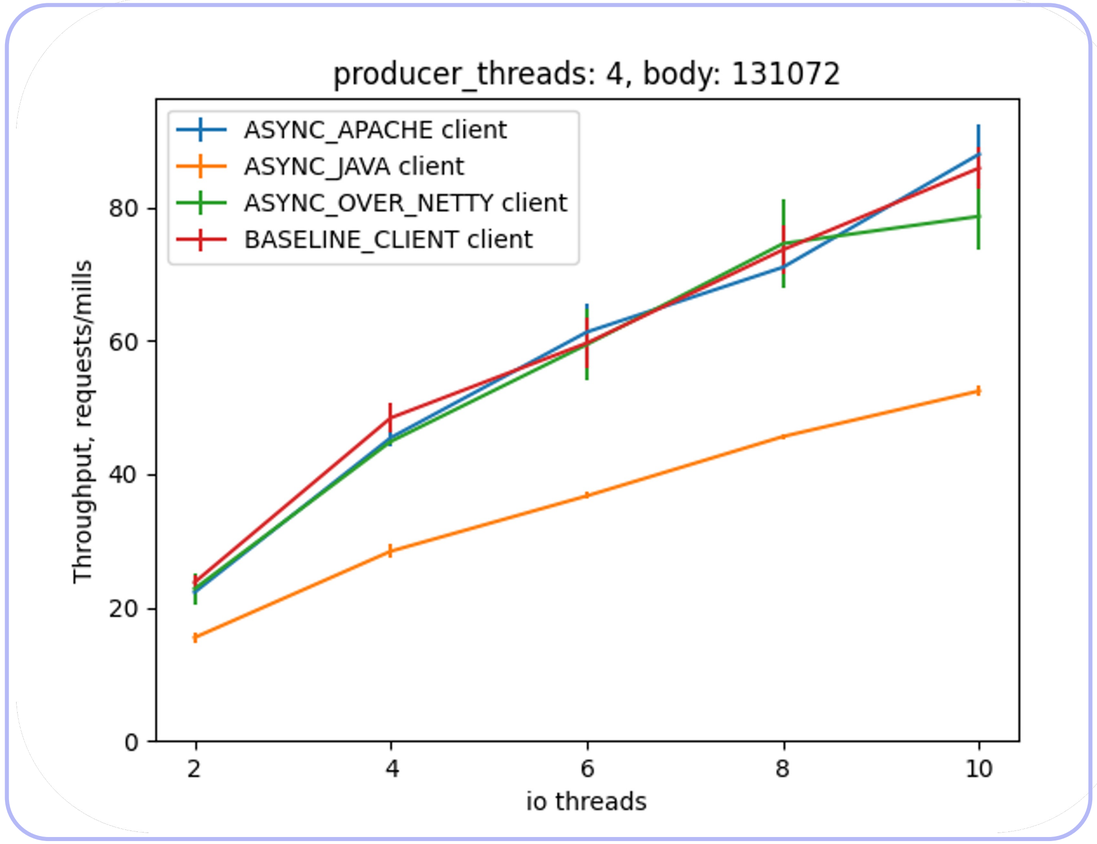
Разные клиенты в зависимости от IO threads, небольшие запросы

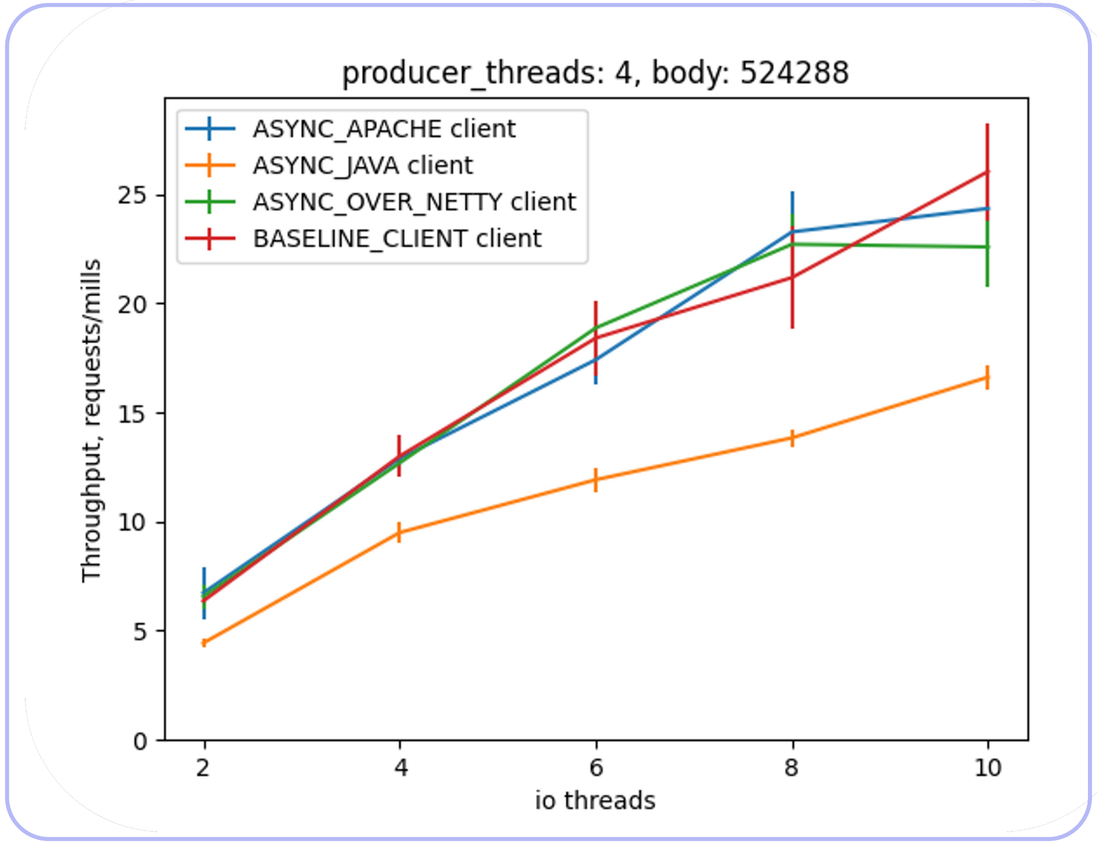




Multiple producers

Разные клиенты в зависимости от IO threads, большие запросы





Заключение

Выбор железа

Изолированное, стабильное, релевантное

01

Настройка стенда

Измерить baseline, разделить ресурсы, настроить инструменты

02

Методология

Реальный workload, достаточное покрытие параметров

03

Анализ

Поставить эксперименты, объяснить числа

04

Существует бесчисленное множество способов измерить что-нибудь не то.
Не забывайте спрашивать себя «Правда ли я делаю то, что нужно?»

Yandex Crowd JPoint

Question & Answers

https://github.com/ddd127/http-clients-benchmarks

demintsievd@gmail.com

@ddd127

Данил Деминцев
Yandex Crowd