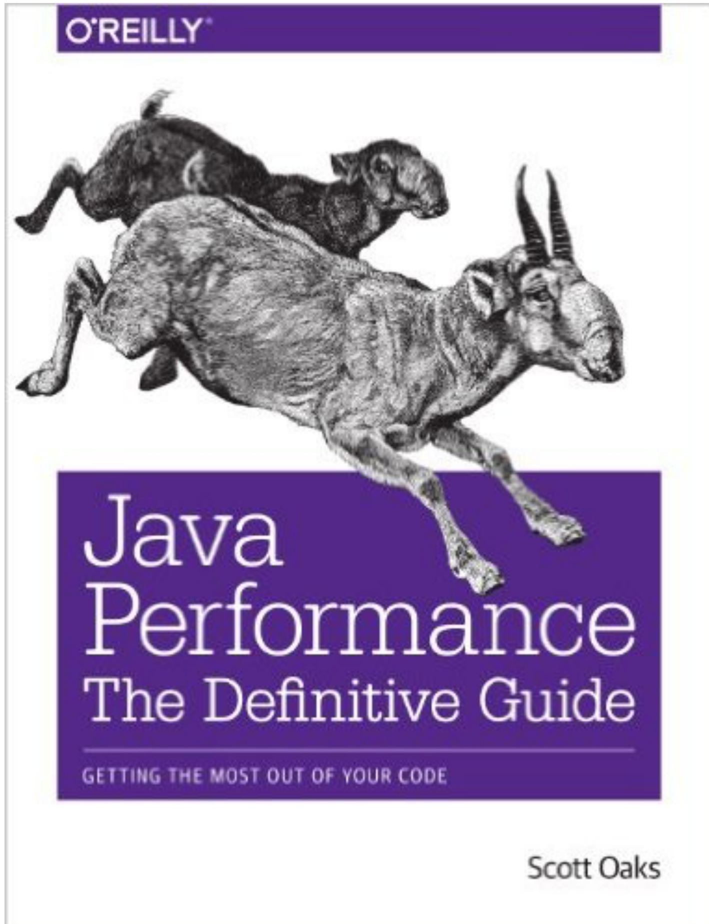
Java Performance

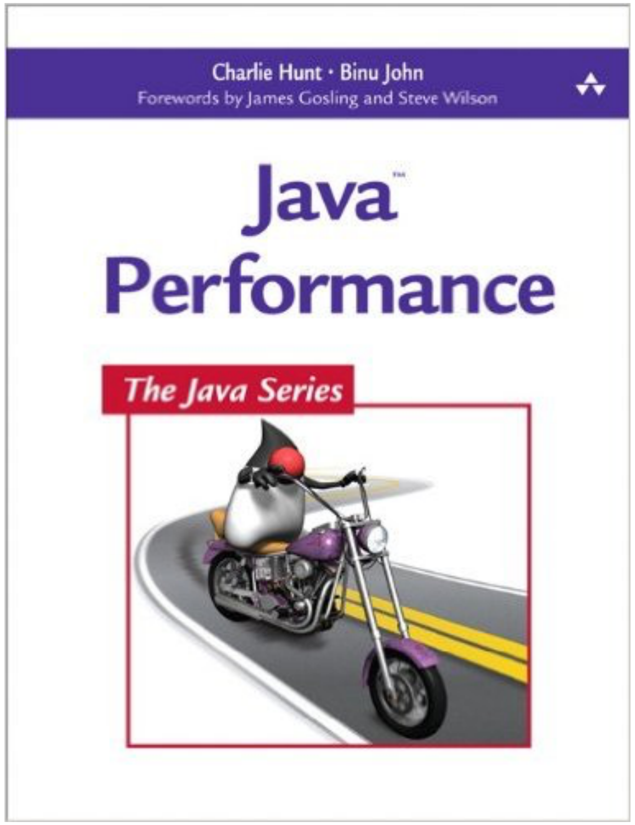
Alexandre Bergel
http://bergel.eu
01-12-2021

Source



Java Performance: The Definitive Guide
O'Reilly Media, 978-1449358457
May 2014

Source



Java Performance
Charlie Hunt, Binu John, 2011
978-0137142521

Objective

Lay out a basic methodology to measure the performance of an application

Describe the parameters that impact software execution performance

Outline

1. The complete story
2. An approach to performance testing
3. Specific tools
4. Example: Genetic Algorithm

The complete story

Write better algorithm

- Performance is based on how well the application is written
- There is no magical -XX:+RunReallyFast option to be provided to the virtual machine
- A good algorithm (or a proper use of an API) is the most important thing

The complete story

Write less code

A small well-written program will have tendency to run faster than a large well-written program

More code is poured into an application, harder it is to keep it fast

The complete story

Premature optimization

Donald Knuth said: “We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil”

Optimizing inevitably makes an application complex. It is important to make sure that each optimization is _really_ necessary

The complete story

The database is always the bottleneck

If you are developing standalone java applications that use no external resources, the performance of that application is all that matter

Once a database is added, then the performance of both programs is important

In a distributed environment, (e.g., Java EE application server), a load balancer, a database, and a backend enterprise information system, then the app performance does not matter much

Extracting multiple times the same information from the database is a common cause of poor performance

Or making too many requests instead of a single (but more complex) one

The complete story

Optimize the common case

Not all performance aspects are equally important

Focus should be given to the common use case scenario

Write *simple algorithms for most common operations*

An approach to performance testing

Measuring performance is fundamental in order to take any action

Here is a simple recipe

1. Define a set of *representative program executions*. These executions are called *benchmarks*
2. Identify the *relevant metrics to measure* these executions (time (e.g., seconds, milliseconds), memory (e.g., used bytes, number of garbage collections, number of objects instantiated)
3. *Measure* your benchmarks
4. *Modify* your application
5. Measure your benchmarks *again and compare*. Jump to 4

Metrics for measurements

Measuring the time is a natural and intuitive approach

However, time is a dimension that is difficult to measure (*)

Not stable: two executions produces two different execution times

Not comparable across different machine: if you buy a new laptop, then you can throw away all your measurements

Very sensitive to the environment: having no warmup may significantly impact your measurements

Multiple time: wall-clock time is a natural measurement. However, execution time is consumed within the application, within the virtual machine (primitives or garbage collector), within the operating system. So which notion of time do you need?

(*) Counting Messages as a Proxy for Average Execution Time in Pharo. Alexandre Bergel. *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*

Metrics for measurements

In addition to measuring time, it is important to pick a metric that is likely to drive your optimization effort

Some metrics may greatly simplify the benchmarks measurement

For example

- number of instances of a particular class
- number of processed requests
- number of accesses to a data base

An approach to performance testing

There are several kinds of benchmarks. We will revise the most common kind of benchmarks, micro and macro

Micro benchmark

Is a test designed to measure a very small unit of performance.

E.g., the time to call a synchronized method versus a non synchronized method

E.g., the time to execute a recursive algorithm versus an iterative implementation

Micro-benchmark

```
public class MicroBenchmark {
    private int nLoops;

    public MicroBenchmark(int l) { nLoops = l; }

    public void doTest() {
        long then = System.currentTimeMillis();
        for (int i = 0; i < nLoops; i++) {
            fibImpl1(35);
        }
        long now = System.currentTimeMillis();
        System.out.println("Elapsed time: " + (now - then));
    }
    …
```

Micro-benchmark

```
public class MicroBenchmark {
    private int nLoops;

    public MicroBenchmark(int l) { nLoops = l; }

    public void doTest() {
        long then = System.currentTimeMillis();
        for (int i = 0; i < nLoops; i++) {
            fibImpl1(35);
        }
        long now = System.currentTimeMillis();
        System.out.println("Elapsed time: " + (now - then));
    }
    …
```

This is what we are interested in measuring

Micro-benchmark

```
  …
  private double fibImpl1(int n) {
      if (n < 0) throw new IllegalArgumentException ("Must be > 0");
      if (n == 0) return 0d;
      if (n == 1) return 1d;
      double d = fibImpl1(n - 2) + fibImpl1(n - 1);
      if (Double.isInfinite(d)) throw new ArithmeticException("Overflow");
      return d;
  }

  public static void main(String[] argv) {
      new MicroBenchmark(2).doTest();
  }
}
```

Micro-benchmark: Common pitfall

A smart compiler can guess that the expression:

```
fibImpl1(35);
```

produces a result that is not used (i.e., the result is not stored somewhere)

And since this computation does not do any side effect, the compiler may simply remove it, to actually have:

```
public void doTest() {
        long then = System.currentTimeMillis();
        long now = System.currentTimeMillis();
        System.out.println("Elapsed time: " + (now - then));
    }
    …
```

Micro-benchmark

```
public class MicroBenchmark {
    private int nLoops;

    public MicroBenchmark(int l) { nLoops = l; }

    public void doTest() {
        double l;
        long then = System.currentTimeMillis();
        for (int i = 0; i < nLoops; i++) {
            l = fibImpl1(35);
        }
        long now = System.currentTimeMillis();
        System.out.println("Elapsed time: " + (now - then));
    }
    …
```

Micro-benchmark: Common pitfall

A smart compiler can guess that the expression:

```
l = fibImpl1(35);
```

is within a loop

And maybe avoid the multiple executions of that expression. In such a case, the argument has to vary, picked from an array for example

As you can see, micro-benchmarking is not a trivial thing to do

Without prior execution

```
public void doTest() {
    double l;
    //fibImpl1(42);
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(42);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
}
```

```
Elapsed time: 4548
```

With a prior execution

```
public void doTest() {
    double l;
    fibImpl1(42);
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(42);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
}
```

```
Elapsed time: 3405
```

Warm-up period

```
public void doTest() {
    double l;
    fibImpl1(42);
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(42);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
}
```

```
Elapsed time: 3405
```

Why is there a difference in execution time?

Warm-up period

```
public void doTest() {
    double l;
    fibImpl1(42);
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(42);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
}
```

```
Elapsed time: 3405
```

This additional execution is called "warm-up"

Warm-up

Code performs better the more it is executed

Micro-benchmarks *must include* a warm-up period

To give the *Just-In-Time compiler* a chance to produce optimal code

A warm-up period is required; otherwise, the micro benchmark is measuring the performance of compilation rather than the code it is attempting to measure

Warm-up

A warm-up phase is typically designed to make the compiler optimize code

A warm-up should not involve access to resources

E.g., when an application reads a file, the OS caches the file into memory. Reading the file a second time will therefore be faster

An approach to performance testing

Macro-Benchmark

- Complex systems are more than the sum of their part
- They will behave quite differently when those parts are assembled

Performance Anti-Patterns

Anti-patterns are certain patterns in software development that are considered bad programming practices.

A number of anti-patterns have been identified

- Fixing Performance at the End of the Project
- Measuring and Comparing the Wrong Things
- Algorithmic Antipathy
- Reusing Software
- Iterating Because That's What Computers Do Well
- …

Performance Anti-Patterns

…

Premature Optimization

Focusing on What You Can See Rather Than on the Problem

Software Layering

Excessive Numbers of Threads

Asymmetric Hardware Utilization

Not Optimizing for the Common Case

https://queue.acm.org/detail.cfm?id=1117403

Specific tools: monitoring system resource consumption

Outside Java, several tools may be used, accessible from the command line:

vmstat, top, iostat, nicstat

These tools are useful to check for the disk, memory, and CPUs consumption

Naturally, graphical tools exists, however, it may be useful to programmatically get these info

The case of web applications

There are many open source and commercial load-generating tools

Faban (http://faban.org) is used to measure the performance of a simple URL

Run a simple GET request of logo.gif for 1000 clients:

```
fhb -J -server -J -Xmx3500m -J -Xms3500m -c 1000 http://localhost:8000/logo.gif
```

Specific tools: monitoring Java activities

- The Java-Development-Kit offers many command lines tools
  - jconsole: provide a graphical view of JVM activities, including thread usage, class usage
  - jvisualvm: a GUI tool to monitoring a JVM, profile a running application, and analyze JVM heap
  - jcmd, jconsole, jhat, map, info, jstack, …

Example: Genetic Algorithm

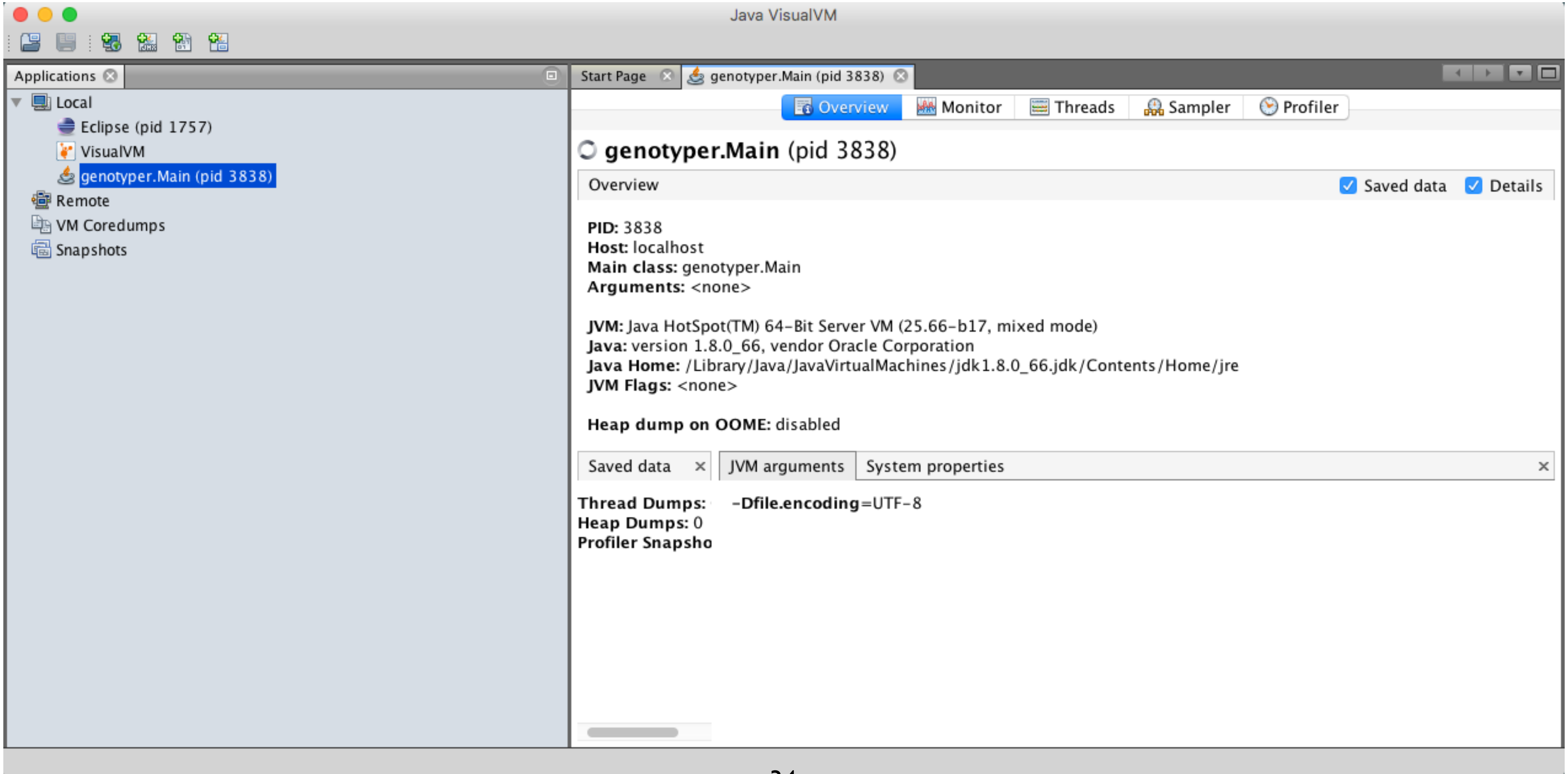
Very useful algorithm from the field of Artificial Intelligence

Mimic the process of natural selection of living species:

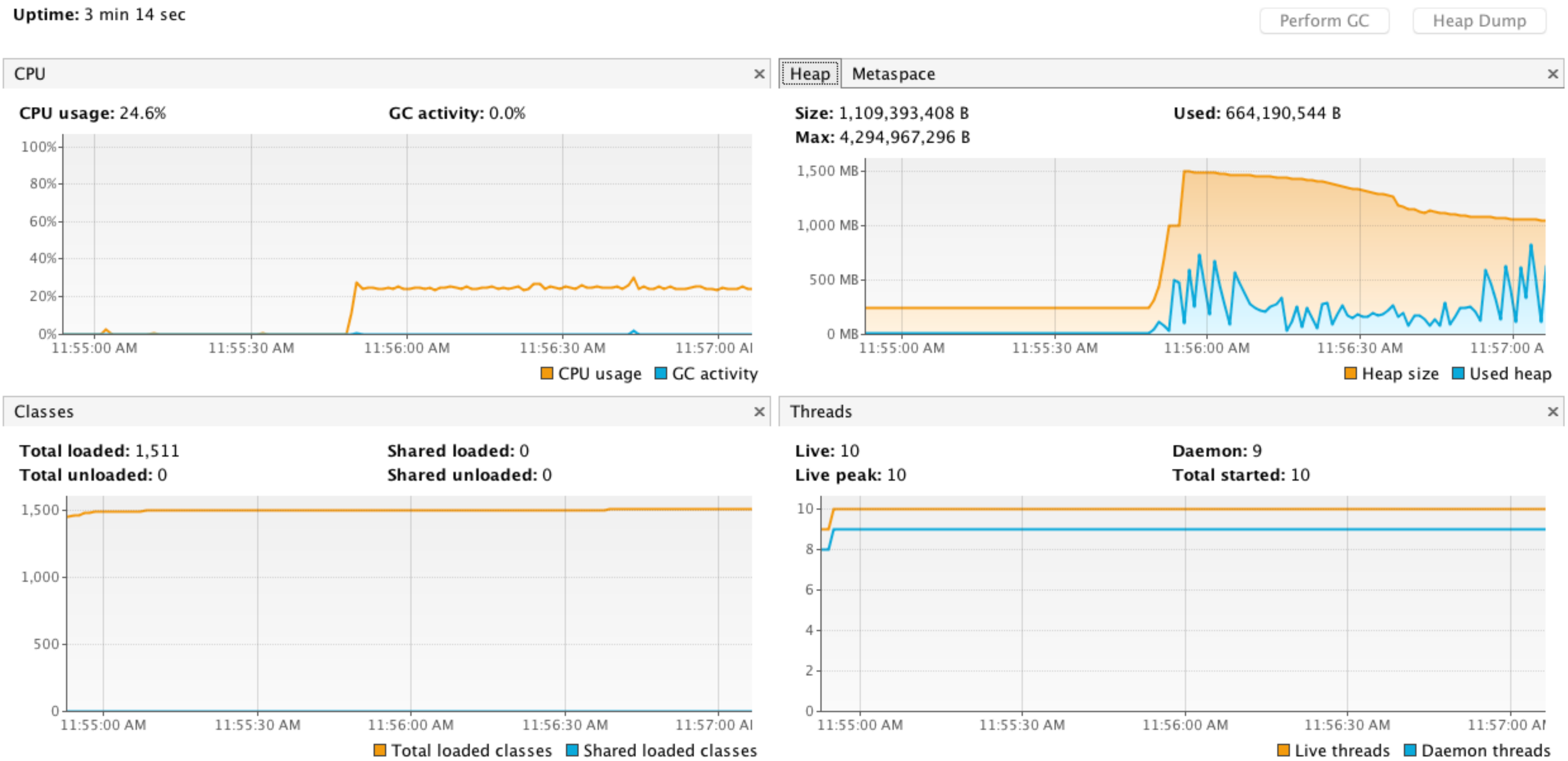
- A population is made of individual
- An individual has a genotype, also called ADN
- At each generation, only the individual that are the closest of the problem solution survive
- At each generation, individual’s genes may be subject to a cross-over or a mutation

Example: Profiling GenoTyper

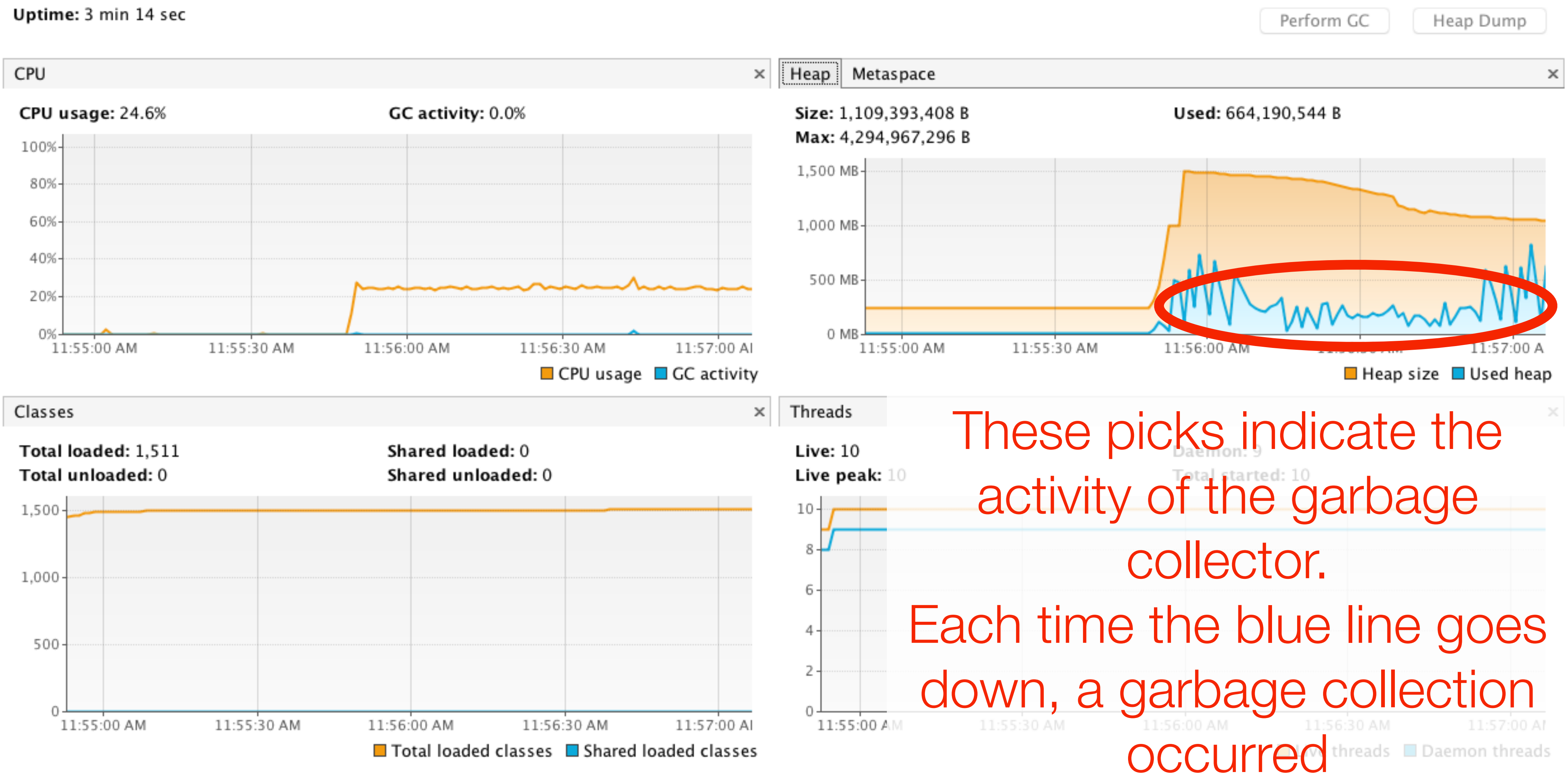
Run Main.java and execute jvisualvm in a terminal



Example: Profiling GenoTyper



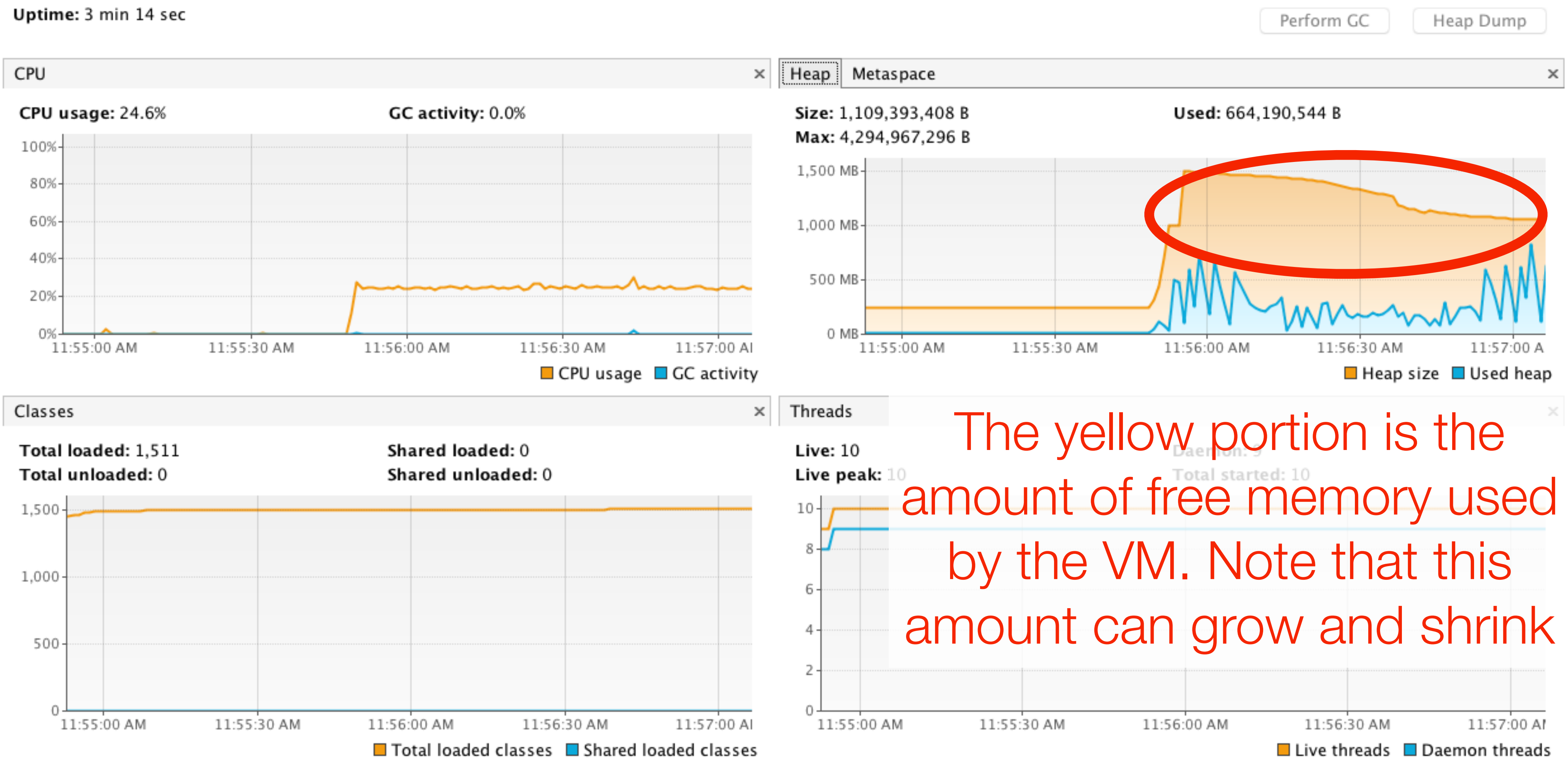
Example: Profiling GenoTyper



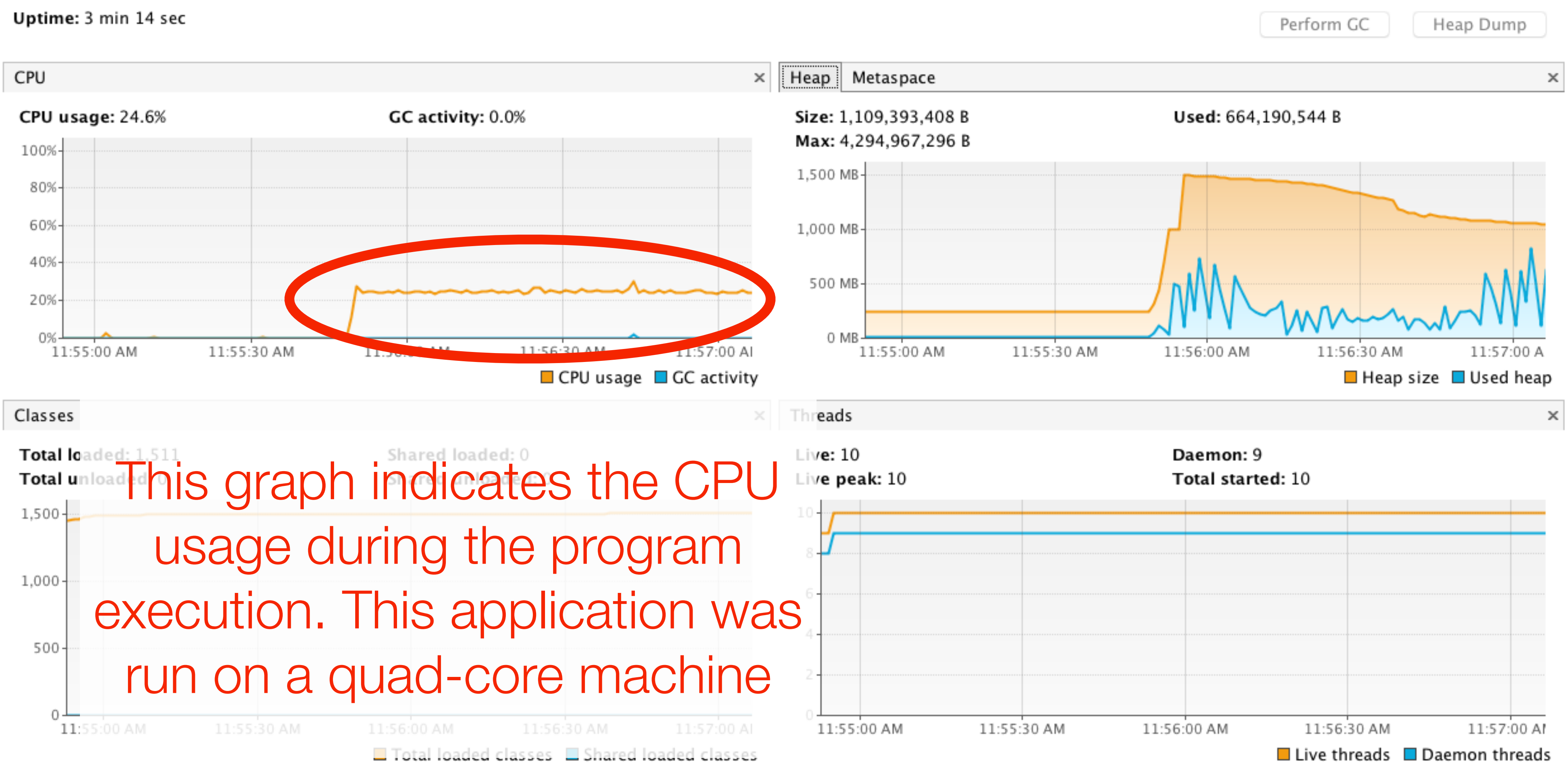
These picks indicate the activity of the garbage collector.

Each time the blue line goes down, a garbage collection occurred

Example: Profiling GenoTyper

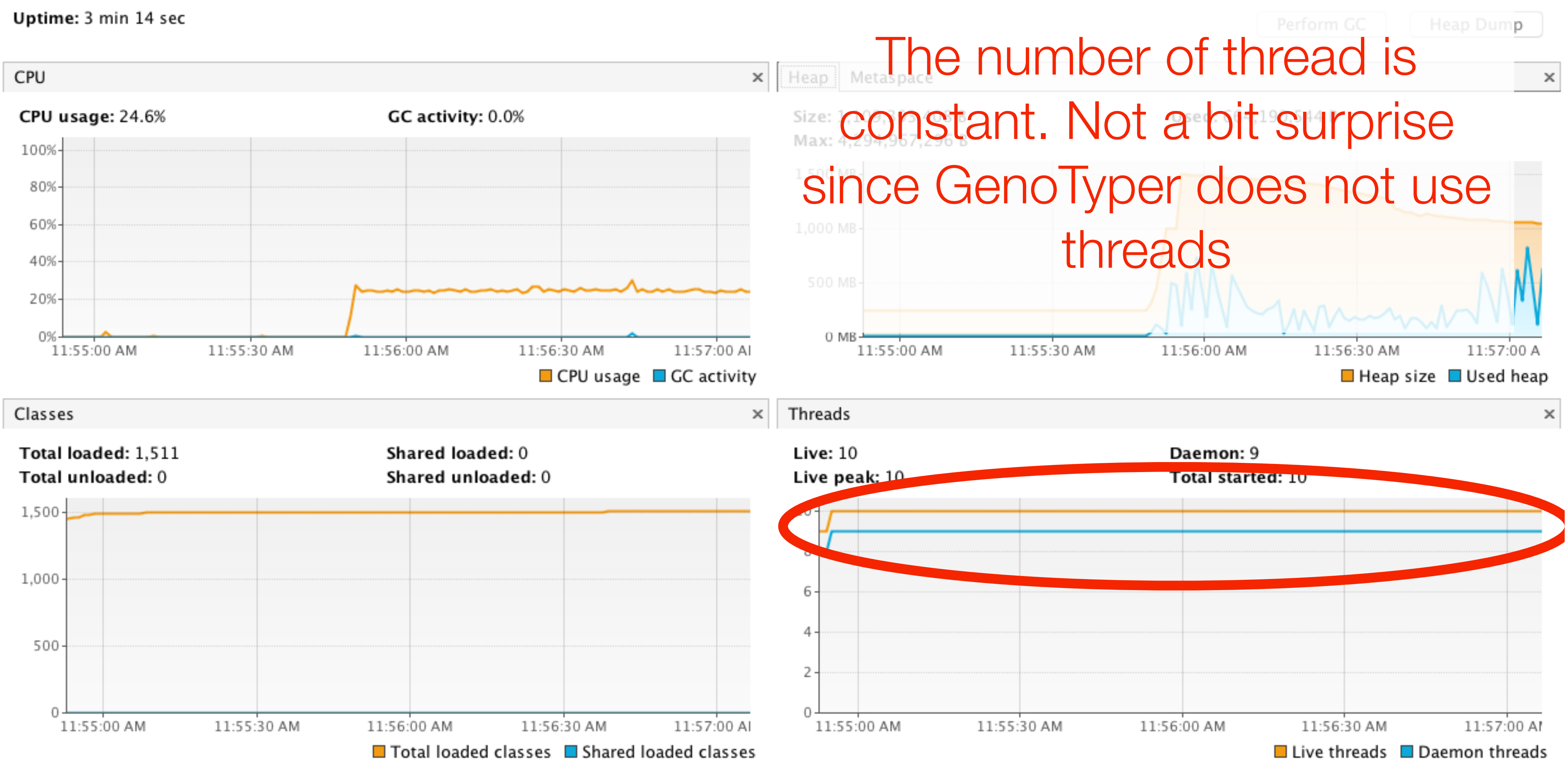


Example: Profiling GenoTyper

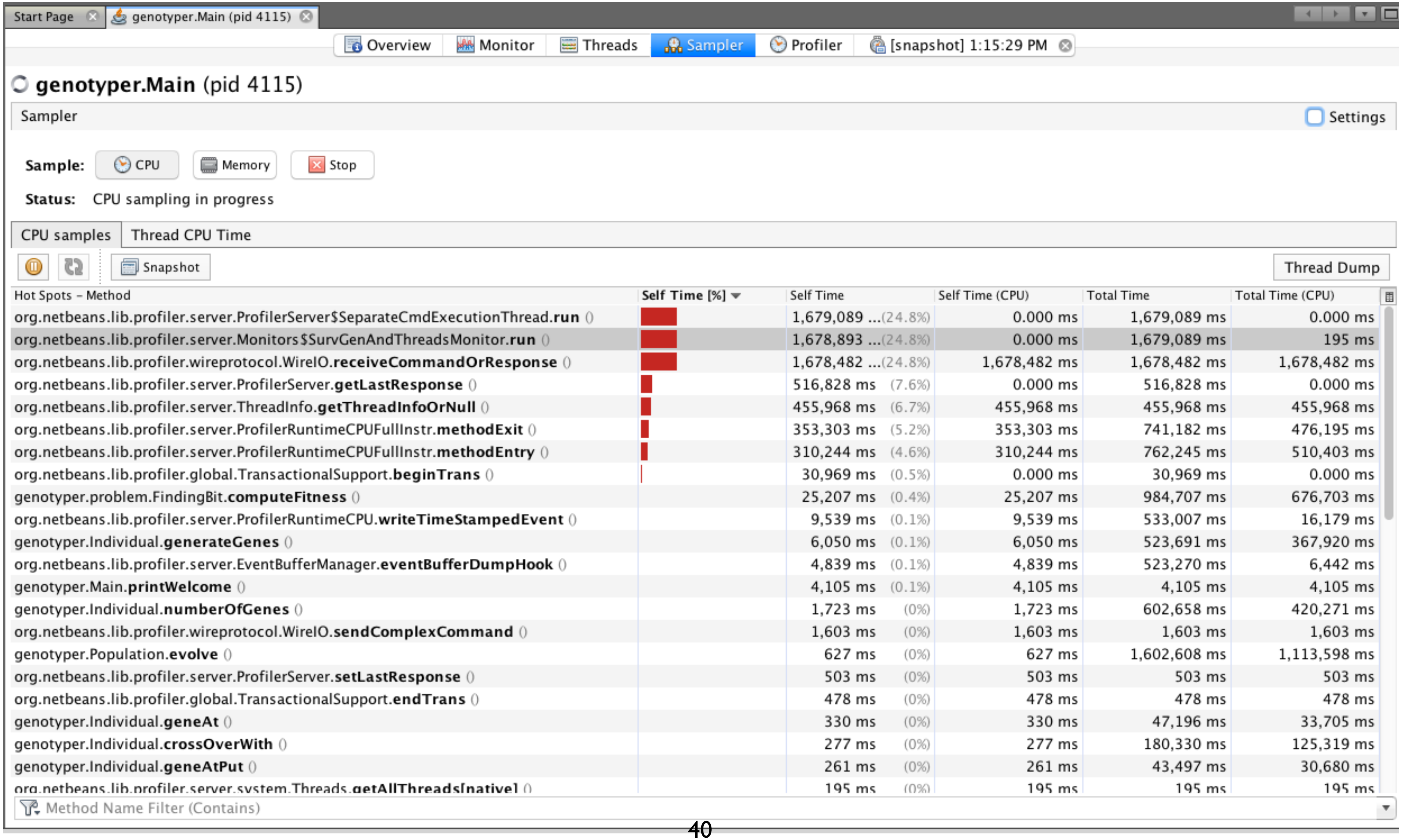


This graph indicates the CPU usage during the program execution. This application was run on a quad-core machine

Example: Profiling GenoTyper



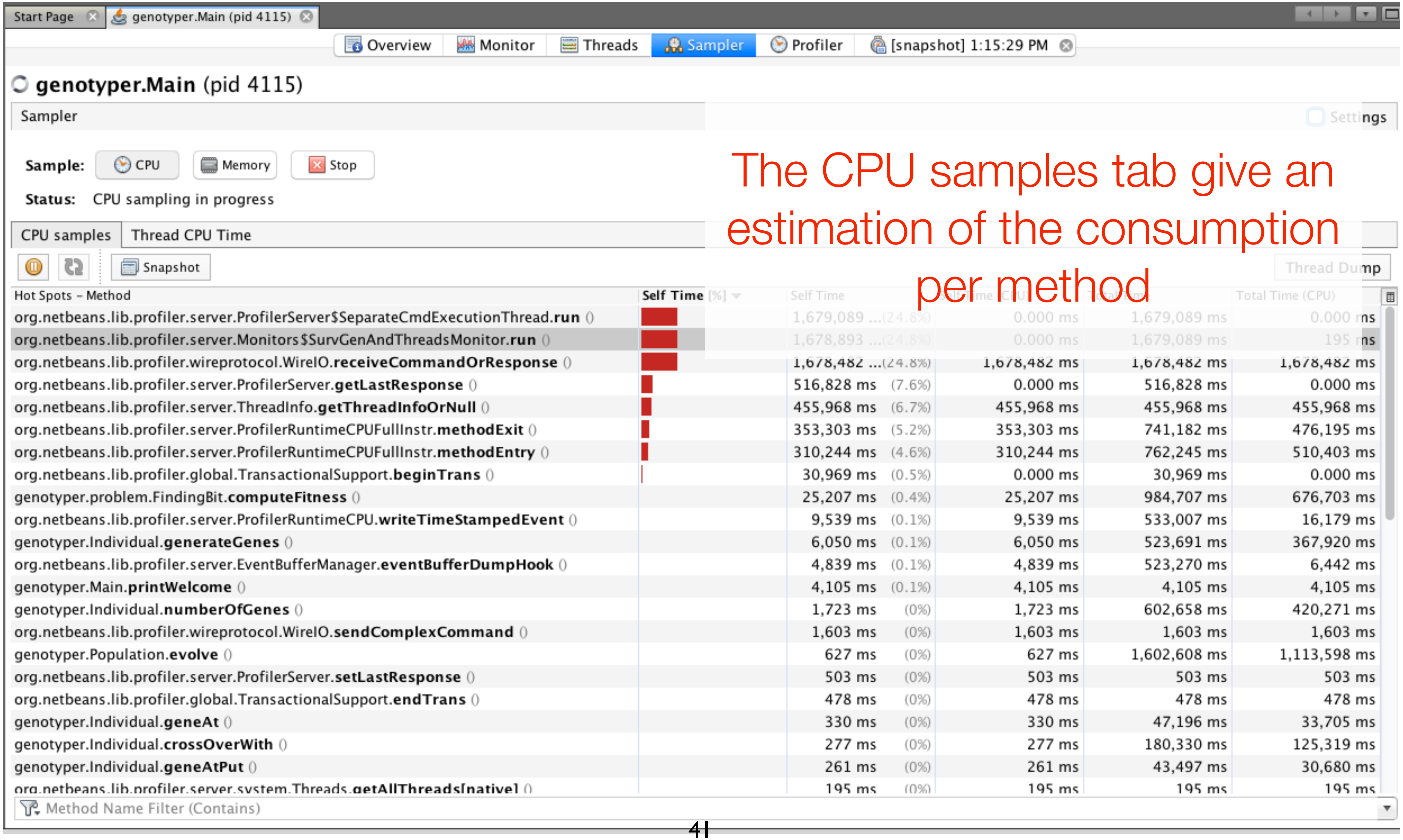
Example: Profiling GenoTyper



Hot Spots – Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
org.netbeans.lib.profiler.server.ProfilerServer$SeparateCmdExecutionThread.**run** ()		1,679,089 ...(24.8%)	0.000 ms	1,679,089 ms	0.000 ms
org.netbeans.lib.profiler.server.Monitors$SurvGenAndThreadsMonitor.**run** ()		1,678,893 ...(24.8%)	0.000 ms	1,679,089 ms	195 ms
org.netbeans.lib.profiler.wireprotocol.WireIO.**receiveCommandOrResponse** ()		1,678,482 ...(24.8%)	1,678,482 ms	1,678,482 ms	1,678,482 ms
org.netbeans.lib.profiler.server.ProfilerServer.**getLastResponse** ()		516,828 ms (7.6%)	0.000 ms	516,828 ms	0.000 ms
org.netbeans.lib.profiler.server.ThreadInfo.**getThreadInfoOrNull** ()		455,968 ms (6.7%)	455,968 ms	455,968 ms	455,968 ms
org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr.**methodExit** ()		353,303 ms (5.2%)	353,303 ms	741,182 ms	476,195 ms
org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr.**methodEntry** ()		310,244 ms (4.6%)	310,244 ms	762,245 ms	510,403 ms
org.netbeans.lib.profiler.global.TransactionalSupport.**beginTrans** ()		30,969 ms (0.5%)	0.000 ms	30,969 ms	0.000 ms
genotyper.problem.FindingBit.**computeFitness** ()		25,207 ms (0.4%)	25,207 ms	984,707 ms	676,703 ms
org.netbeans.lib.profiler.server.ProfilerRuntimeCPU.**writeTimeStampedEvent** ()		9,539 ms (0.1%)	9,539 ms	533,007 ms	16,179 ms
genotyper.Individual.**generateGenes** ()		6,050 ms (0.1%)	6,050 ms	523,691 ms	367,920 ms
org.netbeans.lib.profiler.server.EventBufferManager.**eventBufferDumpHook** ()		4,839 ms (0.1%)	4,839 ms	523,270 ms	6,442 ms
genotyper.Main.**printWelcome** ()		4,105 ms (0.1%)	4,105 ms	4,105 ms	4,105 ms
genotyper.Individual.**numberOfGenes** ()		1,723 ms (0%)	1,723 ms	602,658 ms	420,271 ms
org.netbeans.lib.profiler.wireprotocol.WireIO.**sendComplexCommand** ()		1,603 ms (0%)	1,603 ms	1,603 ms	1,603 ms
genotyper.Population.**evolve** ()		627 ms (0%)	627 ms	1,602,608 ms	1,113,598 ms
org.netbeans.lib.profiler.server.ProfilerServer.**setLastResponse** ()		503 ms (0%)	503 ms	503 ms	503 ms
org.netbeans.lib.profiler.global.TransactionalSupport.**endTrans** ()		478 ms (0%)	478 ms	478 ms	478 ms
genotyper.Individual.**geneAt** ()		330 ms (0%)	330 ms	47,196 ms	33,705 ms
genotyper.Individual.**crossOverWith** ()		277 ms (0%)	277 ms	180,330 ms	125,319 ms
genotyper.Individual.**geneAtPut** ()		261 ms (0%)	261 ms	43,497 ms	30,680 ms
org.netbeans.lib.profiler.server.system.Threads.**getAllThreads[native]** ()		195 ms (0%)	195 ms	195 ms	195 ms

Example: Profiling GenoTyper

The CPU samples tab give an estimation of the consumption per method



Hot Spots – Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
org.netbeans.lib.profiler.server.ProfilerServer$SeparateCmdExecutionThread.**run** ()		1,679,089 ...(24.8%)	0.000 ms	1,679,089 ms	0.000 ms
org.netbeans.lib.profiler.server.Monitors$SurvGenAndThreadsMonitor.**run** ()		1,678,893 ...(24.8%)	0.000 ms	1,679,089 ms	195 ms
org.netbeans.lib.profiler.wireprotocol.WireIO.**receiveCommandOrResponse** ()		1,678,482 ...(24.8%)	1,678,482 ms	1,678,482 ms	1,678,482 ms
org.netbeans.lib.profiler.server.ProfilerServer.**getLastResponse** ()		516,828 ms (7.6%)	0.000 ms	516,828 ms	0.000 ms
org.netbeans.lib.profiler.server.ThreadInfo.**getThreadInfoOrNull** ()		455,968 ms (6.7%)	455,968 ms	455,968 ms	455,968 ms
org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr.**methodExit** ()		353,303 ms (5.2%)	353,303 ms	741,182 ms	476,195 ms
org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr.**methodEntry** ()		310,244 ms (4.6%)	310,244 ms	762,245 ms	510,403 ms
org.netbeans.lib.profiler.global.TransactionalSupport.**beginTrans** ()		30,969 ms (0.5%)	0.000 ms	30,969 ms	0.000 ms
genotyper.problem.FindingBit.**computeFitness** ()		25,207 ms (0.4%)	25,207 ms	984,707 ms	676,703 ms
org.netbeans.lib.profiler.server.ProfilerRuntimeCPU.**writeTimeStampedEvent** ()		9,539 ms (0.1%)	9,539 ms	533,007 ms	16,179 ms
genotyper.Individual.**generateGenes** ()		6,050 ms (0.1%)	6,050 ms	523,691 ms	367,920 ms
org.netbeans.lib.profiler.server.EventBufferManager.**eventBufferDumpHook** ()		4,839 ms (0.1%)	4,839 ms	523,270 ms	6,442 ms
genotyper.Main.**printWelcome** ()		4,105 ms (0.1%)	4,105 ms	4,105 ms	4,105 ms
genotyper.Individual.**numberOfGenes** ()		1,723 ms (0%)	1,723 ms	602,658 ms	420,271 ms
org.netbeans.lib.profiler.wireprotocol.WireIO.**sendComplexCommand** ()		1,603 ms (0%)	1,603 ms	1,603 ms	1,603 ms
genotyper.Population.**evolve** ()		627 ms (0%)	627 ms	1,602,608 ms	1,113,598 ms
org.netbeans.lib.profiler.server.ProfilerServer.**setLastResponse** ()		503 ms (0%)	503 ms	503 ms	503 ms
org.netbeans.lib.profiler.global.TransactionalSupport.**endTrans** ()		478 ms (0%)	478 ms	478 ms	478 ms
genotyper.Individual.**geneAt** ()		330 ms (0%)	330 ms	47,196 ms	33,705 ms
genotyper.Individual.**crossOverWith** ()		277 ms (0%)	277 ms	180,330 ms	125,319 ms
genotyper.Individual.**geneAtPut** ()		261 ms (0%)	261 ms	43,497 ms	30,680 ms
org.netbeans.lib.profiler.server.system.Threads.**getAllThreads[native]** ()		195 ms (0%)	195 ms	195 ms	195 ms

Estimating the CPU consumption

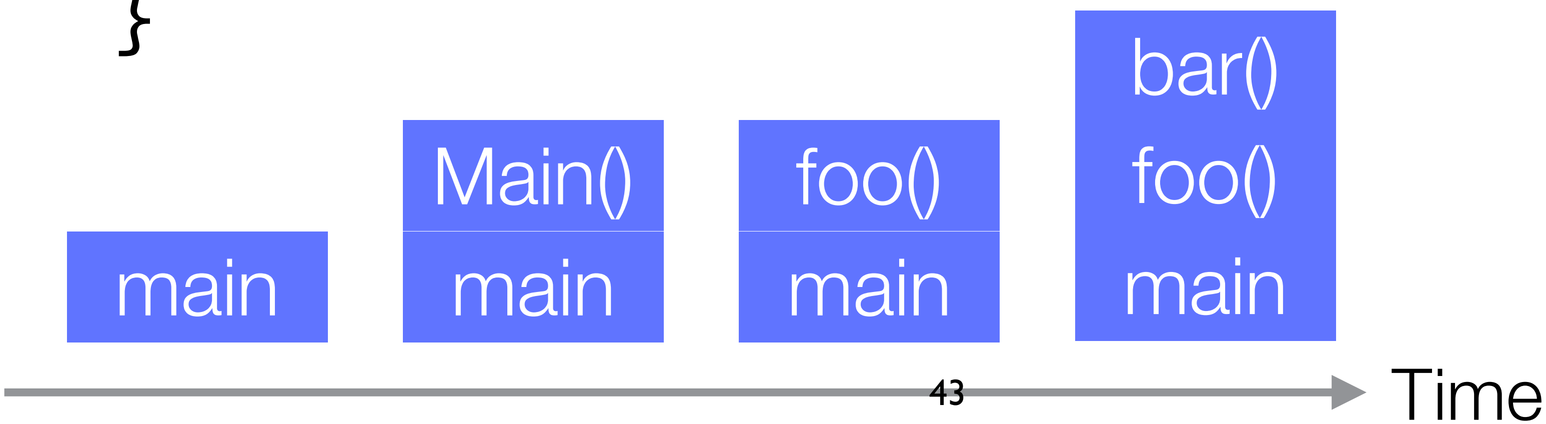
```
public class Main {
    public void foo() {
      this.bar();
    }
    public void bar() { }

    public static void main(String[] args) {
      new Main().foo();
    }
}
```

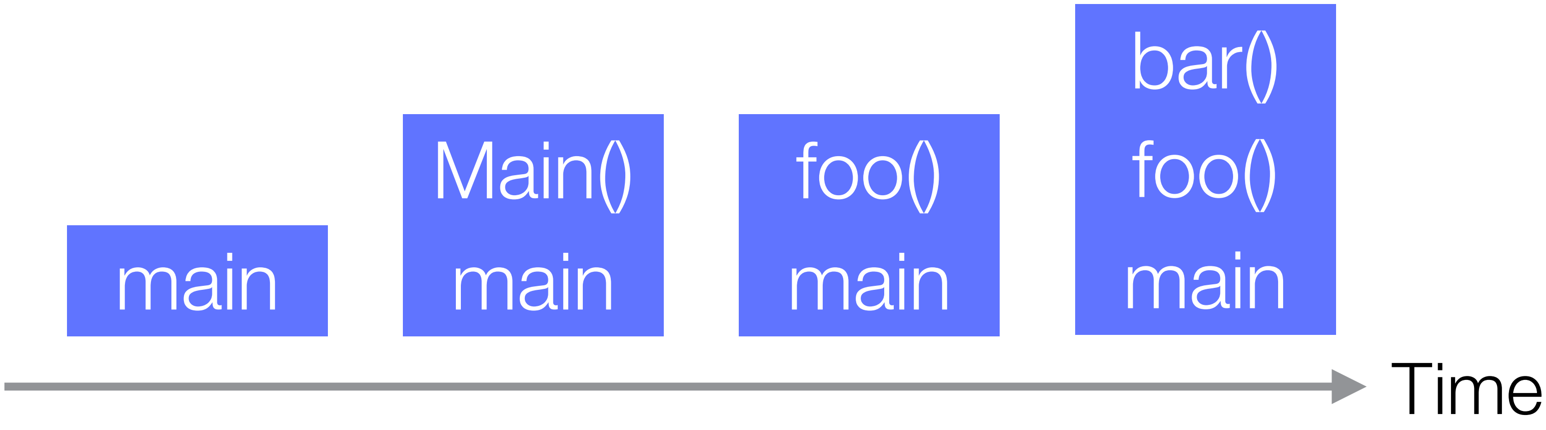
Estimating the CPU consumption

```
public class Main {
    public void foo() {
      this.bar();
    }
    public void bar() { }

    public static void main(String[] args) {
      new Main().foo();
    }
}
```



Estimating the CPU consumption



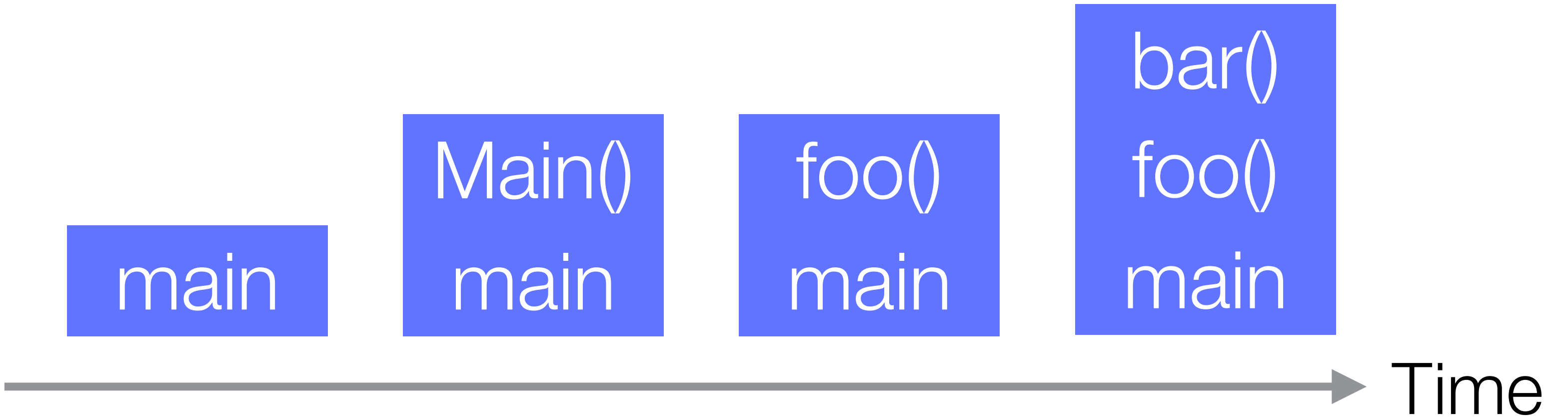
main consumes 100% of the execution time

Main() consumes 25%

foo() 50%

bar() 25%

Estimating the CPU consumption



main consumes 100% of the execution time

Main() consumes 25%

foo() 50%

bar() 25%

This example is a rough approximation

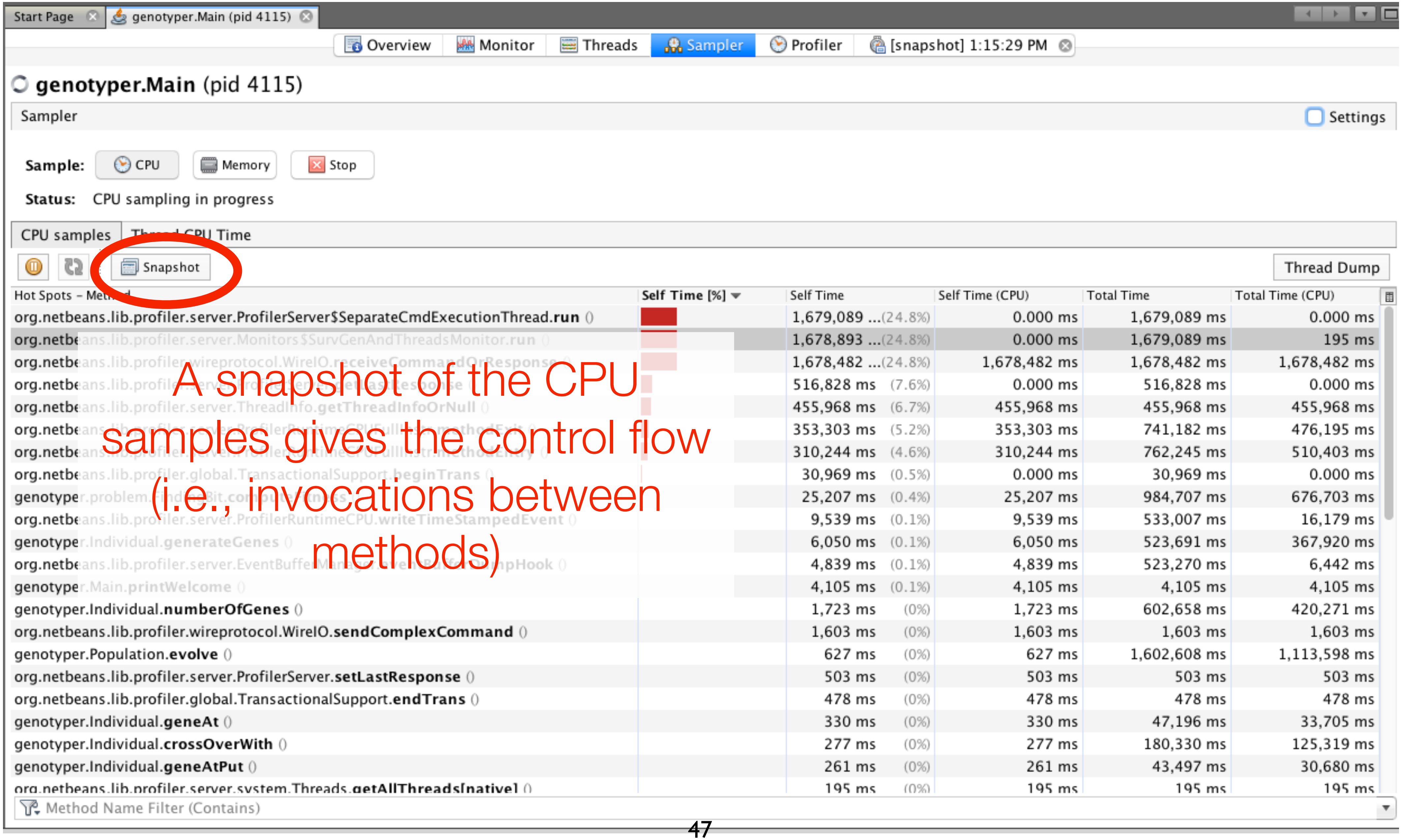
Estimating the CPU consumption

Sampling the execution is typically done every 10 milliseconds (approx)

At each sample, info are extracted from the method call stack

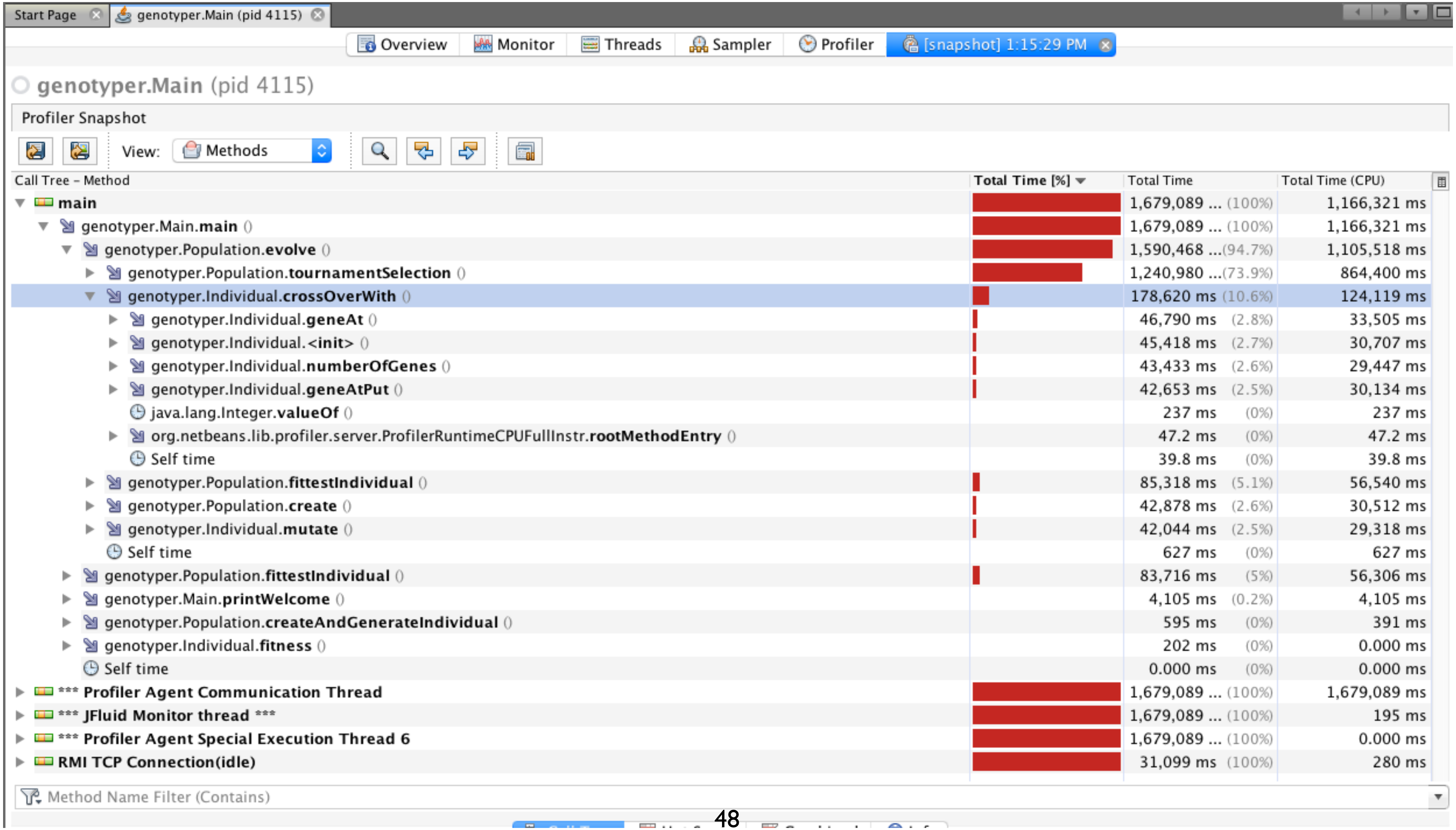
The control flow can also be obtained and used for analysis

Example: Profiling GenoTyper



A snapshot of the CPU samples gives the control flow (i.e., invocations between methods)

Example: Profiling GenoTyper



What you should know!

How to measure an application performance?

What is a warm up phase?

Why is it important to consider warm up?

Can you answer these questions?

How the garbage collectors and the thread contribute to the instability of the execution time?

How to measure the number of instances of a particular class

What are the limitation of CPU sampling profiling?

License