How accurately do Java profilers predict runtime performance bottlenecks?

Master Software Engineering

University of Amsterdam

# How accurately do Java profilers predict runtime performance bottlenecks?

**Peter Klijn**:

| | |
|---|---|
| Student number | 10458565 |
| Email address | peter.klijn@student.uva.nl |
| Tutor | Jurgen Vinju |
| Host company | Finalist |
| Company tutor | Ton Swieb |

January 23, 2014

# Contents

# Abstract

When optimizing a program's performance, one can improve most at the program's biggest bottleneck. To find the biggest bottleneck in a program, people often use a profiler. However, if a program does not speed up as expected, the profiler is rarely blamed. In this thesis we will show why Java profilers are more often wrong than right when profiling programs.

In Java there are two types of CPU profiling techniques: instrumenting and sampling. We will show that the impact of an instrumenting profiler affects a program's performance and how it affects the JVM's optimization decisions, on which Java heavily depends.

Next we will investigate sampling profilers, which disagree more often than they agree with one another. We will show how sampling profilers suffer from inaccuracy due to the Java Virtual Machine's implementation which in some cases causes them to miss even the most obvious bottlenecks.

We will end the thesis by explaining another approach to sampling, introduced by Mytkowicz in 2010, and give advice on future work to further investigate this technique, as we believe it might be the best way to find performance bottlenecks in Java software.

# Preface

Let me start by thanking Finalist. I could always count on their help and assistance during my thesis, a good working environment and hospitality in the various locations where I had the pleasure to work and of course the fun and competitive table soccer matches. In special I would like to thank Ton Swieb for his supervision, valuable feedback and always keeping things in perspective.

Also, I would like to thank Jurgen Vinju from CWI/UvA. Despite his busy schedule he always found the time to help me. I could rely on Jurgen to constantly provide me with a lot of pointers to help me improve my thesis and always finding a way to boost my mood when I got stuck.

Next I would like to thank Liam Blythe from his valuable feedback and spell checking.

Last but certainly not least, I would like to thank Vladimir Kondratyev from YourKit for providing me with a license for YourKit profiler and Ingo Kegel from ej-technologies for providing me with a license for JProfiler. Without those this thesis would not be possible.

# 1 Introduction

To optimize a program's performance, it makes sense to start with the code that takes most time to execute, and when optimized, would make the program benefit the most. In other words, it would make sense to look for the biggest bottleneck in a program.

Unfortunately it is hard to predict which part is the bottleneck in a program, even if one would know and understand the entire code base. This is where a profiler comes in handy. A profiler is a piece of software that looks at what the code is doing while it is running, and is able to calculate (or estimate) which parts of the code most time is spent in.

The only problem is, how do you know if a profiler is right? A program might behave different when it is being profiled compared to a normal run, the so called 'observer effect' [1]. Also, a profiler might have a bias because of the way it is implemented [2].

In this thesis we will investigate whether a profiler can be used to find performance bottlenecks in Java software.

## 1.1 Motivation

It is a common belief that in order to find performance bottlenecks in Java code, one should use a profiler. Which makes sense since a profiler is able to give fairly precise data about the code at runtime. There is however some previous research which shows concerns about a profiler's accuracy.

Because CPU profiling (1.4.1) in Java can be done via two techniques, this thesis will be split up and handle them both separately.

### 1.1.1 CPU profiling via instrumenting

If a program is profiled with an instrumenting profiler, it will know exactly which method is called by who, how often and in what order. This technique however has the downside that in order to collect this information the profiler has to add code to that program, causing a significant overhead [1, 3–6].

Although this additional code slows down the program, it is not our primary concern. The fact that the code is changed should cause more concern. When code is changed, it will behave differently, especially in the Java language [7].

The Java programming language has been specifically designed to have as few implementation dependencies as possible. It is intended to let application developers write their code once and run it anywhere. To do this, Java does not compile its code directly to machine dependent native code, but instead compiles to a machine independent intermediate code called bytecode. The bytecode will then be interpreted by the different JVMs (Java Virtual Machines) specifically made for each platform [8].

Because of the intermediate layer and the overhead of the JVM, Java in its early days was considered a slow language [9]. To cope with this problem Java added optimizations which improve the runtime significantly [8, 10, 11].

When code is injected in a program, it can cause these optimizations to behave differently or even not be executed at all. This makes the profiled program different from the unprofiled program. Machkasova talks about this so called *observer effect* in [1] and shows how an instrumenting profiler causes a very powerful optimization called inlining to not be executed.

### 1.1.2 CPU profiling via sampling

Another technique used to profile Java software is sampling, which has a far lower overhead because it doesn't change the code but instead pauses it at a certain interval to take a sample. Each sample tells what the program is doing at the time of that sample, and even though a lot of information is lost from parts that aren't sampled, it is believed that with enough samples, a realistic estimate can be made [2].

The problem however is that Java can't be interrupted at any point in time because it could conflict with the stack space of the running code. This is further complicated when multiple threads are running. Therefore, taking samples is only allowed at safe points in the code, so called 'yield points' [7, 12].

Still, interrupting a program causes an overhead, and compilers may omit yield points from a loop if they know code does not allocate memory and does not run for an unbounded amount of time. As such samples may not only be postponed, but may even get canceled in a potentially hot part of the code [2, 13].

## 1.2 Research questions

The main research question that will be answered in this thesis is the following:

*How accurately do Java profilers predict runtime performance bottlenecks?*

To answer this question, we will first find out the influence of profiling on the runtime of a program, for which we have the following question:

*What is the impact of profiling on the execution time of a program?*

Knowing the overhead of the different profiler techniques can be useful, but it will not answer the main question. The next step is to find out how this overhead influences different parts of the program and biases the profiler results. Which leads us to the next subquestion:

*What is the impact of profiling on the relative time per function?*

Java uses many optimizations at runtime to increase its performance. The JVM (Java Virtual Machine) determines which parts of the code would benefit from optimizations by monitoring / profiling the code. When profiling a program, it might influence the JVM's decisions which leads us to the subquestion:

*How does the execution of a Java program differ when profiling?*

Knowing the effect of a profiler on a program, we now know how we should interpret the information a profiler reports. This brings us to our final subquestion:

*How should we use a profiler to find bottlenecks?*

By answering these subquestions, we expect to have a better understanding of how the presence of a profiler influences a program's behavior and how that impacts the accurately with which it can predict performance bottlenecks.

Because there are two types of CPU profilers in Java (1.4.1) which largely differ from one another, not all subquestions are fitting for each profiling method.

## 1.3 Research methods

In this thesis, the following research methods are used.

### 1.3.1 Exploratory research

Find previous related research. What has been discovered, concluded and what has been reported for future work? How do results differ from each other and what are possible explanations for the

different results (if there are any).

### 1.3.2 Controlled Experiments

We will use controlled experiments to measure JVM optimizations and compare the different profilers and their impact on programs. We will use various controlled experiments like running Micro tests (small programs made for doing only one or a few operations) and the DaCapo Benchmark Suite.

The results from those experiments will be analyzed via statistical analysis and / or visual analysis.

#### Action research

With action research we hope to be able to prove what we have found in previous research and controlled experiments, such as injecting code in existing methods to test if a profiler recognizes the slower method.

#### Statistical analysis

Statistical analysis sums large datasets and bases conclusions on averages / smallest / biggest outcomes.

#### Visual analysis

Visual analysis presents large datasets in a visually compelling way. Facilitating the user to discover interesting relationships or patterns which otherwise would be hard to find.

## 1.4 Background

This section contains information which helps explain the thesis.

### 1.4.1 Definitions

#### Profiler

A profiler is a piece of software that observes and interacts with a program to gather information about its execution, such as data on how often methods get called [14].

In Java, there are three profiling forms commonly used. CPU profiling determines how much time the program spends executing various parts of its code (called CPU profiling because it uses the CPU's free-running timer (1.4.2)); memory profiling the number, types and lifetime of objects that the program allocates; hot lock profiling determines threading issues like congested monitors [3].

In this thesis we will focus on CPU profiling, of which there are two techniques in Java: instrumenting and sampling.

#### Instrumenting

Instrumenting is a profiling technique where a profiler injects code in the program's (byte)code, usually to record the entering and exiting of a method [15]. By doing this it gathers all data needed to generate a detailed overview of the number of times a method is called, where each call is coming from and the time spent in each method.

#### Sampling

Sampling is a profiling technique where the profiler interrupts the JVM at a certain interval (usually between 5 and 20 ms) in which it finds out what the stack is like to get an estimate of how often methods are being called and where they are most commonly found on the stack [14]. Sampling profilers are built on the assumption that the number of samples for a method is proportional to the time spent in the method [2, 12].

#### Observer effect

The observer effect is the change that occurs in a program's behavior when it is being executed together with a monitoring tool [14].

#### Bottleneck

A bottleneck, in this thesis, is the part of a program that consumes the most time and / or resources. It is the part in a program, which has the greatest influence on the program's speed. In other words, the part that, when optimized to 0 seconds, would speed up the program the most.

#### Hot and cold methods

A hot method is a method that has a significant impact on the execution time of a program. The hottest method is a program's biggest bottleneck. A cold method does not have a significant impact on the execution of a program and optimizing it will have little effect on the performance of a program.

#### Yield points

A yield point is a location in the program's code where it is safe to interrupt. It is used by the garbage collector and to collect samples [2]. Each compiler generates yield points which makes the running thread check a bit in the machine control register to determine if it should yield the virtual processor [12].

Yield points can be skipped for performance reasons which could lead to inaccurate results of sampling profilers [2], which can be found in chapter 3.6.

#### Inlining

The overhead of method invocation is one of the major reasons for the poor performance of Java [11]. For this reason, modern JVMs use the inlining optimization, which gets triggered when a method gets called often. It will replace the method call with the body of the called method.

Inlining is one of the few optimizations which can be prevented by using the `--XX:-Inline` parameter, which we use throughout this thesis because it helps illustrate the impact of instrumenting profilers (2.3) and is used by Mytkowicz's experimental profiler tprof [2] to get accurate results which caused us to use it for our tests with the traditional sampling (chapter 3) and the experimental profiler of Manson (chapter 4).

#### Dead code elimination

Dead code elimination is an optimization which, at run time, does a live variable analysis [16]. It will look for code that does not affect the output and will remove that code to reduce calls, memory usage and calculations. In some circumstances, the compiler can determine that some code will never affect the output, and eliminate that code.

### 1.4.2 How to measure?

In this thesis, several conclusions are based on small Java sample programs (so called micro tests). When running these small programs, it is important to understand the impact of optimizations and external factors that can clutter the results.

This chapter will describe the biggest result changing factors that we have found, but for a complete overview the *Robust Java Benchmarking* web pages from Boyer provide more information [1] [2].

[1] `https://www.ibm.com/developerworks/java/library/j-benchmark1`
[2] `https://www.ibm.com/developerworks/java/library/j-benchmark2`

### Time measuring in Java

There are three different ways to measure elapsed time in Java:

- *Absolute-time clock* relates to some external time reference (such as UTC). This clock is often called the time-of-day clock or wall-clock and has a low resolution (varies between systems but generally 10 milliseconds or worse) [17].
  The absolute-time clock is represented by `System.currentTimeMillis()`.
- *Relative-time clock* uses a high-resolution counter from which a "free-running" time can be calculated (generally the resolution is at microsecond level or better, but if the hardware doesn't support that it will have a resolution which is at least as good as the absolute clock) [17].
  The relative-time clock is represented by `System.nanoTime()`.
- *CPU time* is the time actively spent by the CPU per thread. It also uses relative time (based on a high-resolution counter) but ignores system time (waiting, blocking, IO, etc). CPU time is disabled by default because it can be expensive in some JVM implementations [3].
  The CPU time is represented by `java.lang.management.ThreadMXBean.getCurrentCpuTime()`.

For measuring / calculating the elapsed time, the relative-time clock is recommended because it uses the highest resolution clock available on the system.

When measuring the time of small Java sample programs it is important to know that requesting time causes some overhead and therefore the tests can not be to small. We ran several tests to measure the overhead of requesting time (Appendix A) and noticed an average overhead of 1 ms (varying between 0,05 ms and 1,04 ms). It is important to make sure a test runs long enough to make this overhead negligible.

We believe the relative-time is best suited for finding bottlenecks because is uses the highest resolution available, and also shows 'non-CPU time' in its results (such as logging, garbage collection, thread-blocks) which we believe can cause a program to be slow just as much as an inefficient algorithm.

### Java Virtual Machine warmup

The JVM profiles its code while running to find out which parts would benefit from optimizations. This, together with optimizing it causes overhead. When the JVM believes a piece of code will be executed so often that it will benefit from an optimization enough to overcome the optimization penalty, it will perform the optimization [12, 18].

When measuring, this so called 'warmup' should be taken into account to prevent 'unexpected' results from cluttering the conclusions [19]. The graph listed below illustrates the overhead when running the same test 50 times (Appendix B) without a profiler, but using a profiler had comparable results.

[3] `http://docs.oracle.com/javase/7/docs/api/java/lang/management/ThreadMXBean.html`

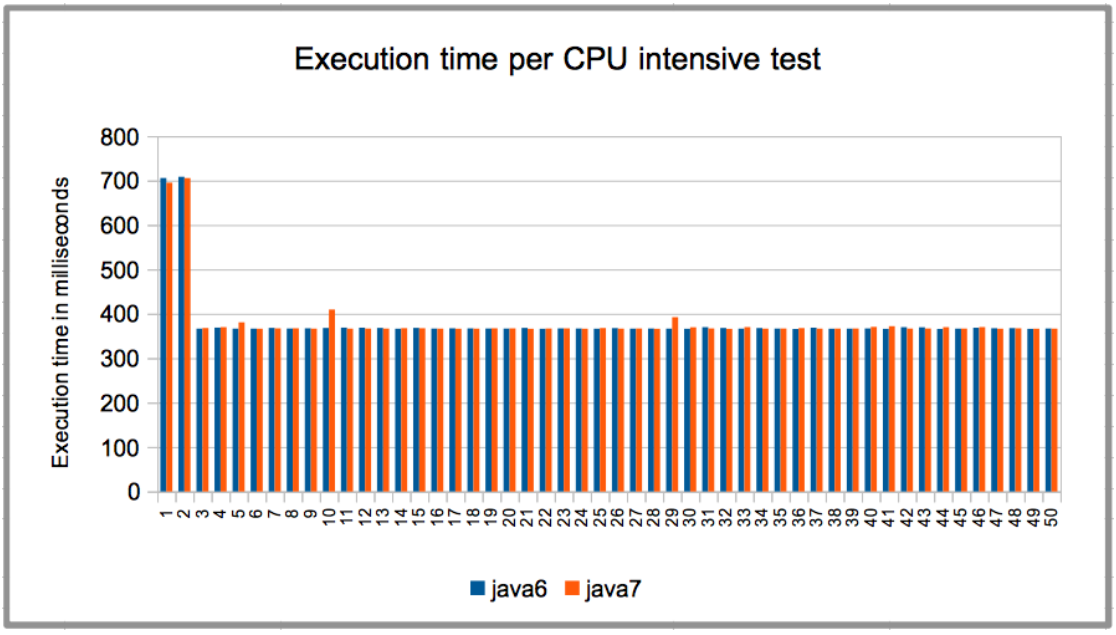



*Figure 1.1: Time spent for each CPU intensive run. This visualization shows that the execution time of the first two runs is almost double due to the warmup of the JVM.*

**Hardware energy saving issues**

When running small tests one should always keep in mind that hardware might need some time to "wake up". Modern computers have many energy saving settings which will impact the results. Especially with IO the results will suffer from idle hardware like conventional hard drives and networks that needs time to get up to speed.

The graph below shows the results of running an IO intensive test (reading 100.000 text files divided over 50 runs) after not using the hard drive for some time.

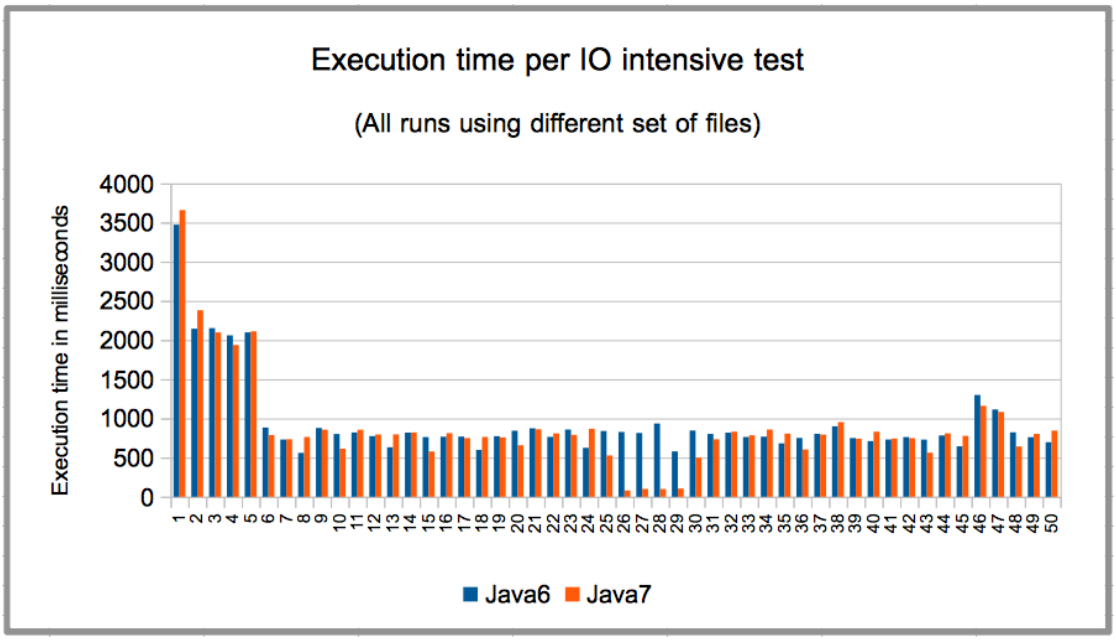



*Figure 1.2: Time spent for each IO intensive run. This visualization shows how different IO intensive tests (reading 2000 text files) speeds up after a few runs, this could be caused by the hard drive which needs to get up to speed after being idle for some time.*

** The low run times of the Java 7 results between run 25 and 30 are unexpected and we suspect could be because of an interrupted test which ran before the Java 7 one. After rerunning the tests we were not able to reproduce it but decided to show the results of the first test because it at the very least shows the importance of using averages of many runs and having a stable environment for tests.*

**Impact of external optimizations**

It is important to keep in mind that the OS and hardware also optimize themselves, for instance when reading IO. Reading small text files more then once will give an unrealistic result when measuring the impact of reading files.

In the graph (figure 1.3, displayed below) we show the second test of reading a batch of 2000 files 50 times in a row. The first runs are not displayed as they contained the wake up effects.

Even though the first few runs were slower than the remaining runs because of the JVM warmup, the reading times are significantly lower than the results of figure 1.2 which reads different files in each of the 50 runs.

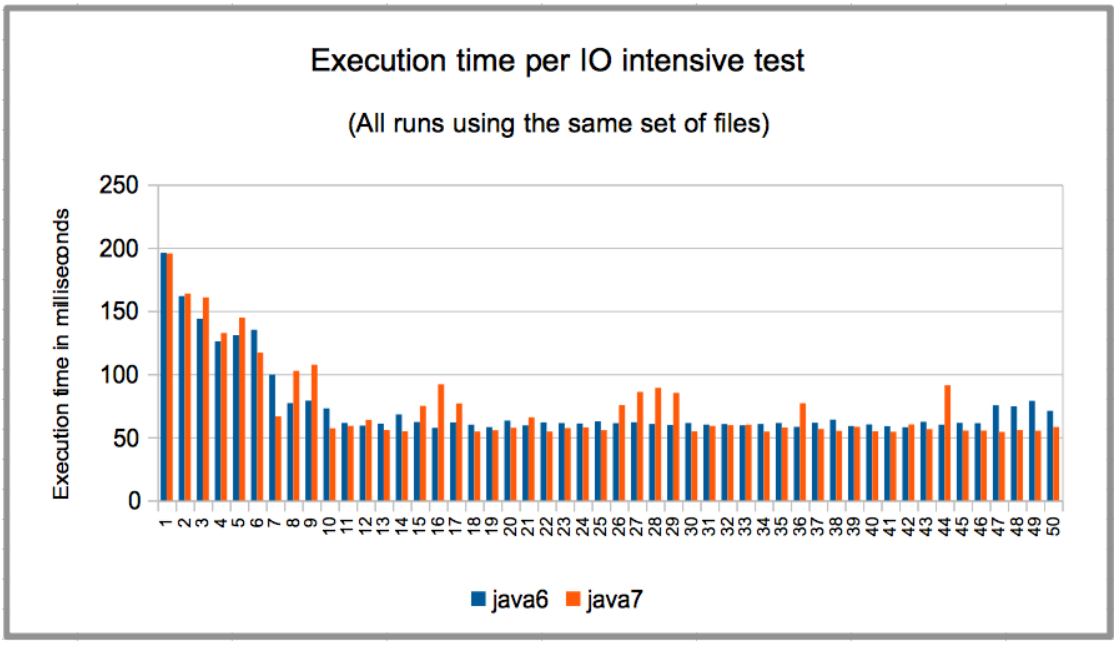



*Figure 1.3: Time spent for each IO intensive run. This visualization shows running the same IO intensive tests (reading 2000 text files) speeds up after a few runs, this is likely because of caching of the read files.*

** The spikes in the Java 7 results can have multiple reasons. It is important to keep in mind that many background processes are running which can intervene. Therefore it is important to reduce the number of background processes and applications, run the test multiple times and use averages of the results.*

## 1.5 Hardware and software

In this thesis we used a few different machines for measurement, below we will describe the different machines and their specifications.

### Machine A: MacBook Pro running Mac OSX 10.8.4 (64 bit)

This MacBook Pro is the stock 2010 model configured with a 2,4 GHz Intel Core i5 with 4GB of 1067MHz DDR3 memory.

It has two Java versions installed:

- Java 1.6.0_45-b06 (called java6 in this thesis)
- Java 1.7.0_17-b02 (called java7 in this thesis)

We used both version of Java in the first experiments because Java 1.6 is still used in real world applications and most previous research is based on this version. We also used Java 1.7 because we wanted to see if there were a lot of differences between the two versions which could make our research irrelevant.

Due to limited time and the minimal difference between these two versions we decided to continue using Java 1.6 only.

### Machine B: MacBook Pro running Ubuntu 12.04.1 LTS desktop (64 bit) virtual on Mac OSX 10.8.4

This MacBook Pro is the stock 2010 model configured with a 2,4 GHz Intel Core i5 with 4GB of 1067MHz DDR3 memory. Using Oracle Virtual Box (version 4.2.16) we gave Ubuntu access to two of the four cores and 906MB of memory.

We used this environment to run the experiments with LightWeight profiler. LightWeight profiler requires a Linux environment and has been made on an Ubuntu environment. We used Ubuntu 12.04.3 because it is the latest LTS version of Ubuntu. We used this machine to run the LightWeight profiler tests because LightWeight profiler often crashed when profiling large projects like the DaCapo Benchmark Suite.

Because the LightWeight profiler is built for OpenJDK we used OpenJDK 6 build 27 (64 bit).

### Machine C: TransIP VPS X1 running Ubuntu 12.04.1 LTS server (2x)

These virtual servers have 1 Intel Xeon core (speed around 2GHz) and 1 GB of memory available.

We used this environment because some tests were very time consuming, often running for many days non stop and we were able to rent more of these machines spread the tests over multiple machines to save time.

We used Java 1.6.0_45 (64 bit) for all tests on this environment.

### Software

We used YourKit version 12.0.5 and JProfiler version 7.2.3 on all tests. All tests with the DaCapo Benchmark Suite are based on version 9.12-bach. The Xprof and Hprof profilers are included in the JVMs, therefore their versions are not mentioned here.

## 1.6 Document structure

The remainder of this document is structured as followed:

- Chapter 2 contains the methods, results and analyses of the instrumenting profiling technique.
- Chapter 3 describes the methods, results and analyses used for the sampling profiling technique.
- Chapter 4 discusses an experimental profiling technique which was introduced in previous research. It also uses an experimental profiler which uses some principles of that previous research.
- Chapter 5 contains the conclusion.
- Chapter 6 contains our advice for future work.

# 2 Instrumenting

## 2.1 What is the impact of profiling on the execution time of a program?

Instrumenting is known for its big overhead on the execution time of a program [3, 14]. How much overhead instrumenting has on a program will differ per program.

### 2.1.1 Method

To measure the overhead, we made two small test programs of which one contains a high amount of method calls which do a simple dividing calculation and the other reads a collection of text files from a local traditional hard drive. We hope to see how much overhead instrumenting has on CPU intensive applications (method calls and calculations) and IO heavy applications.

### 2.1.2 Results

**Impact of instrumenting on CPU heavy operations**

The source of the program which we've used to measure the overhead can be found in Appendix B.

| Mode | Java 6 | Java 7 |
|---|---|---|
| Normal | 367 ms | 369 ms |
| No-inline | 3.508 ms | 3448 ms |
| Hprof | 2.334.390 ms | 2.182.181 ms |
| YourKit | 124.940 ms | 149.542 ms |
| JProfiler | 253.267 ms | 259.127 ms |

**Impact of instrumenting on IO heavy operations**

The source of the program which we've used to measure the overhead can be found in Appendix D.

| Mode | Java 6 | Java 7 |
|---|---|---|
| Normal | 661 ms | 652 ms |
| No-inline | 664 ms | 653 ms |
| Hprof | 3.631 ms | 3.299 ms |
| YourKit | 659 ms | 714 ms |
| JProfiler | 693 ms | 683 ms |

**How does the instrumenting overhead influence CPU and IO results**

The results above show that instrumenting profilers have a bigger overhead on the CPU heavy operations than on IO related operations. This is explainable because IO operations are system calls which are out of scope for the profiler while each method call of the CPU heavy operations requires additional logging.

To show how this impacts the results, we've made a combination of the two programs which we've used to measure these results. We measure the time required to do a high number of CPU intensive operations and measure the time to do a high number of IO intensive operations, both with Java's relative time function `nanoTime` and compare the time spent in the CPU and IO part of the program.

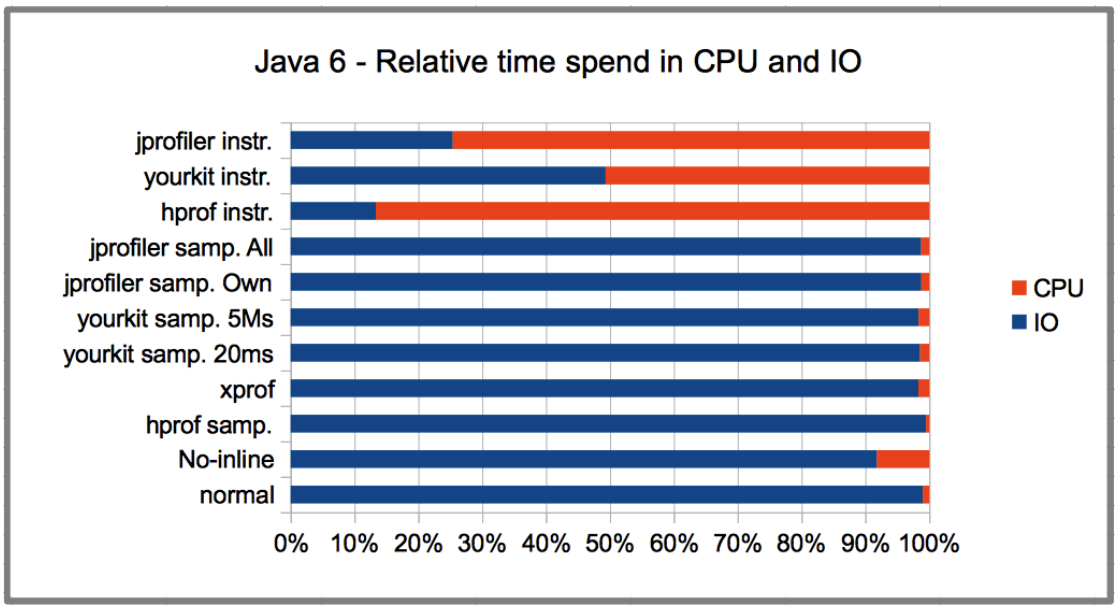



*Figure 2.1: Visualization of the time spent in CPU heavy vs IO heavy parts of a test program and how profilers impact this ratio (Java 1.6)*

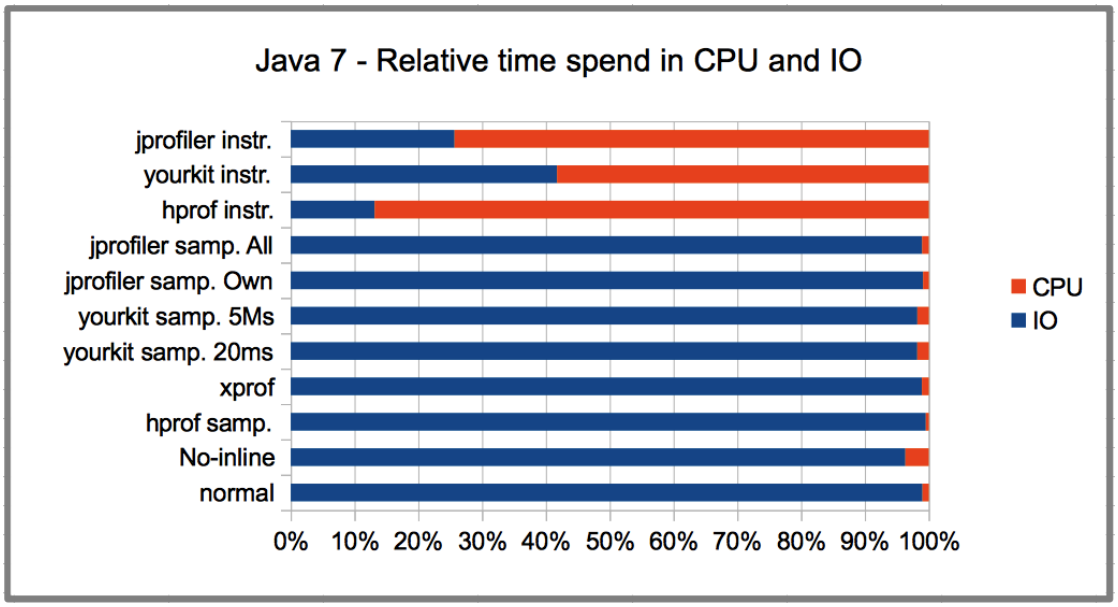



*Figure 2.2: Visualization of the time spent in CPU heavy vs IO heavy parts of a test program and how profilers impact this ratio (Java 1.7)*

Looking at the graphs above, it becomes clear that between 92% and 99% of the execution time is spent in the IO part, but when an instrumenting profiler is run it gives the impression that more then half the time is spent in the CPU part.

The source of this program can be found in Appendix E.

### 2.1.3 Analysis

We have shown that instrumenting causes a high overhead on small CPU intensive methods while IO related activity has a far lower overhead. This is explainable because when Java tries to do an IO operation, it calls the operating system, which is out of the scope of the profiler. This difference in overhead causes the instrumenting profiler's results to have a bias towards CPU intensive small methods.

Because the difference between Java 6 and Java 7 is minimal, we will run the remaining tests in Java 6.

### 2.1.4 Threats to validity

To make sure that external factors didn't impact our results, we ran the CPU tests 50 times on the same JVM instance and removed the first few results of each run because of the JVM warmup which clutters the results. Each measured time in this section is an average of remaining runs on a single JVM instance (47 runs with one exception for JProfiler which required 5 runs before giving stable times) measured with Java's relative time function `nanoTime`.

We ran the IO tests 50 times (all with different text files) and removed the first 10 runs because of the JVM warmup. We did this three times and removed the first run to prevent hard drive wake ups from cluttering the results. The results are averages of the 80 remaining runs measured with Java's relative time function `nanoTime`.

Because the overhead of instrumenting was so high, the tests took too long to execute. The results mentioned above are multiplications of smaller tests. To be sure that this wouldn't invalidate the results we first tested if the time scale was linear, which it was. This can be found in Appendix C.

Java optimizes code if it detects repeating calculations, which is what we are doing in the micro tests in this chapter. To prevent various Java optimizations from optimizing our tests and therefore cluttering our results, we have to trick the JVM in thinking each run is unique to prevent it from optimizing.

Each CPU test consists of a random start point (varying between 0 and 8). Because the loop continues to 100.000 we believe the difference is small enough not to impact our results, while still fooling the JVM enough to prevent optimizations from replacing the loop with the end value.

Each IO test grabs two random characters from each IO file and adds them to a variable which gets printed at the end of the test which prevents the JVM from thinking of this test as dead code and removing it.

## 2.2 What is the impact of profiling on the relative time per function?

Knowing the overhead of instrumenting profilers on the execution time of a program can be useful but doesn't tell if it can be used for finding performance bottlenecks. What if instrumenting makes a program significantly slower, but it does find the bottleneck? We can assume that the bottleneck of a slow program also is the bottleneck of a fast program.

It's more important to find out what instrumenting does to the relative time spent in each method because it gives us a better insight of the effect that the profiler has on the program and therefore its result.

### 2.2.1 Method

To test the overhead of instrumenting on the relative time per function, we wrote a test program that runs three methods, which differ in size and required execution time, and try to find out if and how instrumenting profilers impact their execution time.

### 2.2.2 Results

The program starts by calling `runTests` which in turn calls the methods `simple` (which is the simplest of the three and takes the least time), `middle` and `hard` (which takes most time) in separate loops and measures the time it takes to complete each loop using Java's `nanoTime` function.

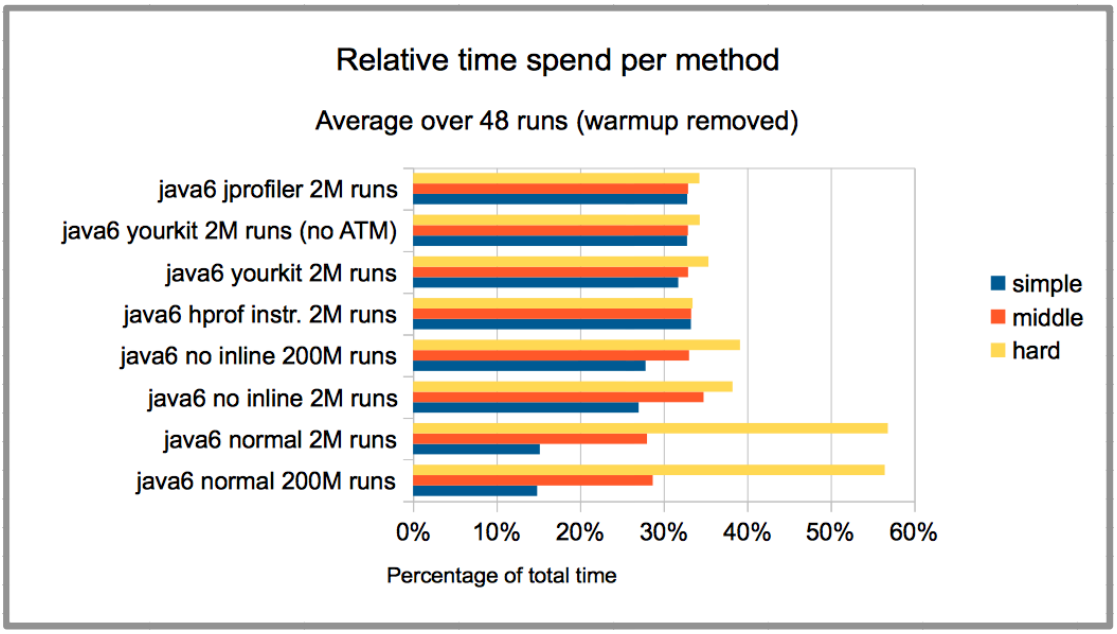



*Figure 2.3: Visualization of the percentage of measured time spent in each loop of methods simple, middle and hard*

The graph above shows how the overhead added by instrumenting impacts the difference in time between methods. The time difference is clearly visible in the normal runs as well as the runs without the inlining optimization. But the overhead that instrumenting causes makes this differences much smaller.

The table below shows the measured times (according to `nanoTime`) which helps to explain the graph above. Take for example the small method times of the normal run and JProfiler run:
Normal run: 8 ms / ( 8 ms + 15 ms + 31 ms) * 100 = 14,8%
JProfiler run: 532 ms / ( 532 ms + 534 ms + 555 ms) * 100 = 32,8%

| | simple method | middle method | hard method |
|---|---|---|---|
| java6 normal 2M runs | 8 ms | 15 ms | 31 ms |
| java6 (inlining disabled) 2M runs | 29 ms | 37 ms | 41 ms |
| java6 Hprof instr. 2M runs | 4731 ms | 4735 ms | 4759 ms |
| java6 YourKit 2M runs | 256 ms | 265 ms | 285 ms |
| java6 YourKit 2M runs (no ATM) * | 979 ms | 983 ms | 1025 ms |
| java6 JProfiler 2M runs | 532 ms | 534 ms | 555 ms |

** The ATM mentioned at YourKit's result is a setting they call "Adaptive Tracing Mode" which, when enabled (which it is by default), does not instrument small methods with a high invocation count to reduce instrumenting overhead. It is mentioned here because with this function enabled, the profiler is unable to find the simple, middle and hard methods in its result.*

The source code of this program can be found in Appendix F.

### 2.2.3 Analysis

The time between the simple and middle method, and the time between the middle and hard method increases by roughly the same amount, but because of the big overhead of instrumenting on these

methods their differences seem negligible. The overhead is so big that differences between methods (almost) disappears, which may prevent one from finding the bottleneck.

### 2.2.4 Threats to validity

To prevent outliers from cluttering the results, we ran the 3 functions 50 times, removed the first runs because of the JVM warmup. The results in this chapter are averages of the remaining 48 runs.

The results of the profilers are left out of this chapter because they do not add value to this section as they can't be compared with unprofiled programs and the difference between the three methods is comparable to the results above (with exception to YourKit which finds the methods to be 0% of the execution time). The results of the profilers can be found in Appendix G.

## 2.3 How does the execution of a Java program differ when profiling?

We know that an instrumenting profiler injects code to be able to measure programs activity in such detail. But changing the size of the code changes the optimizing decisions which the JVM's Just In Time (JIT) compiler makes [7, 11]. Java heavily depends on its optimizations to be able to compete with other languages in terms of performance [9].

Until now, we showed the overhead of instrumenting, and how this overhead makes the differences between methods feel negligible. But previous research suggests and some even prove that instrumenting causes code to behave differently from non-instrumented code.

### 2.3.1 Method

To answer this subquestion, we will sum existing literature and their results to explain how an instrumenting profiler impacts the execution decisions of the JVM and how this makes the execution different from a normal run.

### 2.3.2 Results

Dmitriev says that instrumenting causes overhead because it takes time to execute its own methods, but may in its presence also prevent optimizations from taking place. It also may cause processor cache misses [3].

Machkasova demonstrated the effect of an instrumenting profiler on the dead code elimination optimization using the Hprof profiler and a simple program [15]. Below the two test programs they used for their tests are listed. The Dead code example does not print the result which causes the JVM to prevent the code from being executed.

Live code (prevents dead code elimination):

```
1  public static void main(String[] args) {
2    long t1 = System.nanoTime();
3
4    int result = 0;
5    for (int i = 0; i < 1000 * 1000; i++) {
6      result += sum();
7    }
8
9    long t2 = System.nanoTime();
10   System.out.println("Execution time: " +
           ((t2 - t1) * 1e-9) +
11     " seconds to compute result = " +
             result);
12 }
13
14 private static int sum() {
15   int sum = 0;
16   for (int j = 0; j < 10 * 1000; j++) {
17     sum += j;
18   }
19   return sum;
20 }
```

Dead code (triggers dead code elimination):

```
1  public static void main(String[] args) {
2    long t1 = System.nanoTime();
3
4    int result = 0;
5    for (int i = 0; i < 1000 * 1000; i++) {
6      result += sum();
7    }
8
9    long t2 = System.nanoTime();
10   System.out.println("Execution time: " +
           ((t2 - t1) * 1e-9) +
11     " seconds to compute result = ");
12 }
13
14 private static int sum() {
15   int sum = 0;
16   for (int j = 0; j < 10 * 1000; j++) {
17     sum += j;
18   }
19   return sum;
20 }
```

When running the live code example without a profiler, it takes 5,86 seconds to execute. Running the dead code example takes 0,14 seconds to execute. Which shows that the dead code elimination did take place. Running the live code with a profiler takes 10,41 seconds to execute while running the dead code takes 10,65 seconds, which shows there is hardly any difference between the two runs. See the complete data below.

| | Dead code | Live code | Dead code (10x) | Live code (10x) |
|---|---|---|---|---|
| Not profiled | 0,14 seconds | 5,86 seconds | 0,145 seconds | 68,82 seconds |
| Profiled | 10,41 seconds | 10,65 seconds | 103,49 seconds | 102,81 seconds |

In another study Machkasova shows how the Hprof profiler impacts the inlining optimization [1]. To demonstrate this, she used the following code samples:

Easy inline code:

```
1  for(int i=0;i<2147483647;i++)
2          counter += return1();
3
4  public int return1(){
5    return 1;
6  }
```

Complex inline code:

```
1  for(int i=0;i<2147483647;i++)
2    counter = addCount(int i);
3
4  public int addCount(int add) {
5    add=add+1;
6    return add;
7  }
```

Hand inlined code:

```
1  for(int i=0;i<2147483647;i++)
2    counter++;
```

The easy inline and complex inline code should be inlined and comparable to the hand inlined code, the time it takes to run each test will show if the code is inlined.

| run | Complex (S) | Easy (S) | Hand (S) | Complex (C) | Easy (C) | Hand (C) |
|---|---|---|---|---|---|---|
| Unix | 0,282 | 0,284 | 0,282 | 6,446 | 6,45 | 6,456 |
| Unix, no inlined | 11,92 | 14,06 | 0,28 | 29,878 | 18,174 | 6,372 |
| Unix, Hprof * | 4,634 | 4,876 | 0,176 | 4,92 | 4,64 | 0,204 |
| Windows | 0,43 | 0,474 | 0,45 | 7,982 | 8,05 | 7,972 |
| Windows, no inline | 21,578 | 20,706 | 0,422 | 27,048 | 26,992 | 7,864 |
| Windows, Hprof * | 5,882 | 5,268 | 0,244 | 5,482 | 5,242 | 0,252 |

*Runtimes in seconds. (S = Server mode, C = Client mode, * The Hprof runs used a smaller loop because of the instrumenting overhead)*

The results above show that under normal run conditions the inlining optimization on the complex and easy test both get executed which causes their runtimes to be comparable with the hand inlined version.

When the tests are run with Java's `--no-inline` flag, the difference between the complex, easy and the hand inlined code becomes visible. This demonstrates the improvement that the inlining optimization has on the execution time of a program.

The instrumenting profiler's results are comparable to the run without inlining (it requires far more time to execute the complex and easy tests compared to the hand inlined version). This shows the impact of instrumenting profilers on the inlining optimization.

### 2.3.3 Analysis

The results from Machkasova's papers confirm Dmitriev's assumption that instrumenting profilers cause the JVM to behave different. The results show examples of how the dead code elimination is prevented when an instrumenting profiler is used. It also shows that inlining likely gets disabled when an instrumenting profiler is used, which has a high impact on the code's execution time.

### 2.3.4 Threats to validity

The tests that Mytkowicz ran with the Hprof profiler, were also done with JProfiler. The results of these tests we're not included in the papers because they showed a similar pattern as the Hprof tests.

Because this behavior is reproducible with another profiler, they explain it is not an artifact of the Hprof profiler.

## 2.4 How should we use a instrumenting profiler to find bottlenecks?

In this chapter we investigated the overhead of instrumenting a program. We show that instrumenting causes a high overhead on small CPU intensive methods while IO related activity has a far lower overhead. The difference in overhead causes the instrumenting profiler's results to have a bias towards CPU intensive small methods.

We have also shown that the overhead caused by an instrumenting profiler is so big that differences between methods of different length seem to (almost) disappear, which may prevent a bottleneck from being found.

Finally we summed some research which shows that instrumenting causes a program to behave differently because of their influence on the JVMs optimizing decisions.

These arguments together lead us to believe that instrumenting is not suited as a technique for finding runtime CPU performance bottlenecks. It can help to understand a program as it can show precise information like which method calls which and how often, but we believe it will likely fail in actually finding the true bottleneck in a program.

This however does not mean that there is no use for instrumenting profilers. The main advantage of instrumentation-based profilers over techniques like sampling-based profilers is its flexibility. Virtually any kind of data can be captured using this technique, from EJB security check occurrences to GUI event firing.

For CPU performance measurements, the advantage of instrumentation is that it records the exact number of events such as method invocations; is capable of measuring precise timings (instead of a statistical approximation); and can be used selectively (for example on just a few methods) [3].

## 2.5 Discussion

Mccanne and Torek explain that instrumenting profilers are not well suited as 'offline' profilers because of their enormous overhead, but would be far more useful if a user could dynamically, at runtime, instrument only a part of the code and, when done, remove the instrumenting code bringing the program back its original state [20]. We agree that having this kind of instrumenting profiler would make it more practical in use, as it decreases the overhead significantly, but still doesn't solve the key problems of instrumenting. As soon as an instrumenting profiler starts, the code will behave differently causing an observer effect. It might help in investigating once a bottleneck is found, but we doubt it will help solve the problems an instrumenting profiler has for finding the actual bottlenecks.

# 3 Sampling

## 3.1 What is the impact of profiling on the execution time of a program?

Unlike instrumenting, sampling profilers are believed to have little overhead because they don't modify the source code but instead interrupt the JVM on a regular interval. To test the impact of sampling on Java programs, we will use the same two programs as used in the instrumenting profiler tests which stress either the CPU or IO and see how sampling impacts execution time of those programs.

### 3.1.1 Method

To measure the overhead, we made two small test programs off which one contains a high amount of method calls which do a simple dividing calculation and the other reads a collection of text files from a local traditional hard drive. We hope to see how much overhead sampling has on CPU heavy applications (method calls and calculations) and IO heavy applications.

### 3.1.2 Results

**Impact of sampling on CPU heavy operations**

The source of the program which we've used to measure the overhead can be found in Appendix B.

| Mode | Java 6 | Java 7 |
|---:|---:|---:|
| Normal | 367 ms | 369 ms |
| No-inline | 3.508 ms | 3.448 ms |
| Xprof | 382 ms | 427 ms |
| Hprof | 374 ms | 410 ms |
| YourKit (20 ms sampling interval) | 377 ms | 484 ms |
| YourKit (5 ms sampling interval) | 399 ms | 485 ms |
| JProfiler (own method filter) | 376 ms | 431 ms |
| JProfiler (no filter) | 376 ms | 436 ms |

**Impact of sampling on IO heavy operations**

The source of the program which we've used to measure the overhead can be found in Appendix D.

| Mode | Java 6 | Java 7 |
|---:|---:|---:|
| Normal | 661 ms | 652 ms |
| No-inline | 664 ms | 653 ms |
| Xprof | 665 ms | 704 ms |
| Hprof | 798 ms | 772 ms |
| YourKit (20 ms sampling interval) | 706 ms | 701 ms |
| YourKit (5 ms sampling interval) | 871 ms | 900 ms |
| JProfiler (own method filter) | 677 ms | 677 ms |
| JProfiler (no filter) | 673 ms | 749 ms |

### 3.1.3 Analysis

We have shown that sampling has a very low overhead on performance. Also the overhead on the CPU intensive tests is roughly the same as the overhead of the IO intentsive operations. Compared to the instrumenting results of chapter 2.1, this profile technique shows a more realistic spread of overhead.

### 3.1.4 Threats to validity

To prevent external factors from cluttering the results, we ran the CPU tests 50 times and removed the first 3 runs because of the JVM warmup. The results in this chapter are averages of 47 runs measured with Java's relative time function `nanoTime`.

We ran the IO tests 50 times (all with different text files) and removed the first 10 runs because of the JVM warmup. We did this three times and removed the first run to prevent hard drive wake ups from cluttering the results. The results are averages of the 80 remaining runs measured with Java's relative time function `nanoTime`.

Java optimizes code if it detects repeating calculations, which is what we are doing in the micro tests in this chapter. To prevent various Java optimizations from optimizing our tests and therefore cluttering our results, we have to trick the JVM in thinking each run is unique to prevent it from optimizing.

Each CPU test consists of a random start point (varying between 0 and 8). Because the loop continues to 100.000 we believe the difference is small enough not to impact our results, while still fooling the JVM enough to prevent optimizations from replacing the loop with the end value.

Each IO test grabs two random characters from each IO file and adds them to a variable which gets printed at the end of the test which prevents the JVM from thinking of this test as dead code and removing it.

## 3.2 Do profilers agree on their hottest method?

It makes sense that sampling has a far lower overhead than instrumenting, instrumenting alters existing code by adding its own code which adds overhead because is has more code to execute and the growing size of methods makes it harder to optimize them. Sampling does not alter code but instead interrupts the JVM every few milliseconds.

Sampling profilers assume that the number of samples for a method is proportional to the time spent in the method [2]. For this to be true, the number of samples taken should be high enough to make it statistically valid. This number of samples depends per program, the more methods a program has, the more samples it would need to get a realistic result. When enough samples are taken, the various sampling profilers should all have a similar result. If two profilers have different results, only one (if any) can be right.

### 3.2.1 Method

To test if profilers agree with each other, we will profile 11 programs from the DaCapo Benchmark Suite [21] with four different popular profilers: Xprof, Hprof, YourKit and JProfiler in both the normal mode and without the inlining optimization (which gave us some unexpected results in the Hot and Cold test in chapter 3.6 and the experimental profiler's Hot and Cold test in chapter 4.1).

### 3.2.2 Results

From the 11 programs which we've profiled of the DaCapo Benchmark Suite, there were only 3 programs on which the profilers agreed on the hottest method (Batik I.7, Lucene Index I.11 and Xalan I.14).

### 3.2.3 Analysis

The results above show that the four used sampling profilers more often disagree then agree. Keeping in mind that only one profile can possibly be the correct one, this draws into question the trustworthiness of profilers. If a profiler is not able to predict the hottest method of a program correctly, how can we trust a profiler to be correct about the less hot and therefore less easy to find methods?

### 3.2.4 Threats to validity

We've chosen the DaCapo Benchmark Suite because it consists of a set of open source, real world applications. It also has the ability to run benchmarks several times on the same JVM instance, which helps minimizing the JVM warmup from cluttering the results.

All results are based on averages of 100 runs per program, spread over 5 JVM instances (So 5 times 20 runs per JVM instance). All programs where profiled 100 times with each of the 10 profiler / setting variations, which required between 2 and 3 hours per program. Each profiler ran for approximately 12 to 18 minutes per program, which we believe is high enough to get a statistically valid number of samples.

When inspecting the results, we sometimes ignored the results from JProfiler labeled 'jp-own'. JProfiler by default has a setting that filters some of Java's classes, while other profilers do not filter the results.

We decided to run the tests in both settings (labeled 'jp-own' for the default setting and 'jp-all' for the unfiltered results) because we didn't want to ignore the default setting as we assume it is default for a reason.

The results from this setting however appeared to sometimes be close to the unfiltered results of JProfiler, and the remaining times the results appeared unlikely.

## 3.3 Is the hottest method disagreement innocent?

There can be different explanations for the disagreement between profilers. We have to keep in mind that sampling profilers estimate hotness based on their recorded samples [22]. What if, for example, profiler $\boldsymbol{A}$ finds method $\boldsymbol{x}$ to be 19% and method $\boldsymbol{y}$ to be 18% and profiler $\boldsymbol{B}$ finds method $\boldsymbol{y}$ to be 19% and method $\boldsymbol{x}$ to be 18%. Then technically the two profilers would disagree, but since both method $\boldsymbol{x}$ and $\boldsymbol{y}$ are considered hot and close to each other, one could say this disagreement is innocent.

If we take the top $\boldsymbol{n}$ methods of all four profilers and union their results, we could see that for instance the four profilers would disagree on each others hottest, but maybe agree on each others top $\boldsymbol{n}$ methods if we ignore the order in which they occur.

### 3.3.1 Method

For this test, we will make a union from the top 1 to top 10 methods found across all profilers to see if profilers start agreeing with each other if the number of methods grow. We will do this for all 11 programs which we profiled from the DaCapo Benchmark Suite. We will calculate the average over these 11 programs.

### 3.3.2 Results

The graph below visualizes the average number of different methods found in the results of the 11 profiled programs of the DaCapo Benchmark Suite. It shows the growth from the top 1 to top 10 method list together with the ideal and worst case scenario's.

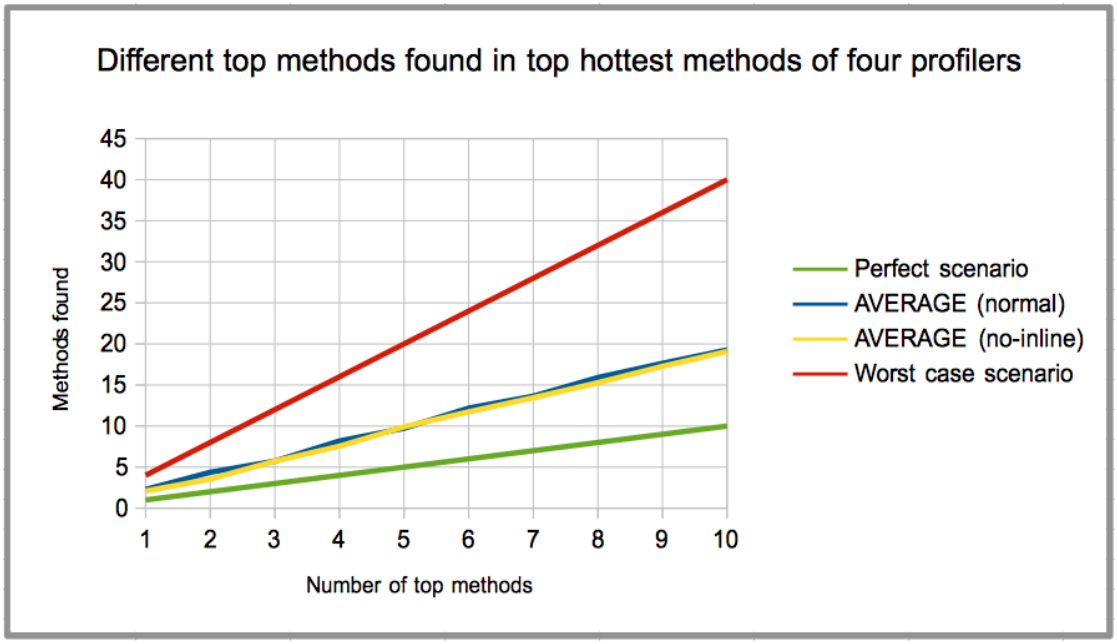



*Figure 3.1: The number of different methods found in the top methods lists of Hprof, Xprof, YourKit and JProfiler based on an average of 11 programs from the DaCapo Benchmark Suite.*

The data used for this graph can be found in Appendix I.15.

### 3.3.3 Analysis

The graph above shows that the disagreement between profilers is not only in the first method but persists if in the union of more methods.

This indicates that profilers do not only disagree on the hottest method, but seem to continue to disagree with each other in a close to linear pattern. Which makes their disagreement more serious and their outcome questionable.

### 3.3.4 Threats to validity

Because an average of 11 programs can be invalidated if we have a big outlier, we did a manual inspection to make sure this was not the case with our data.

Other possible threats to validity equal to the previous chapter can be found in section 3.2.4.

## 3.4 Is the difference in top methods explainable?

Another explanation of profiler disagreement could be that the different hottest methods have a caller/callee relationship. Two profilers may disagree, but the disagreement may not be problematic; it is often difficult to understand a method's performance without also considering its callers and callees.

If a profile is incorrect, it does not have to mean it cannot help find a bottleneck. For example the caller/callee scenario might lead to the true bottleneck without much effort, and even though a profiler was incorrect, one would benefit from its report.

### 3.4.1 Method

To find out if profiler disagreement can be explainable, we will do a manual code inspection in 3 of the programs we profiled which appear to be explainable (some profilers agree on each other and the found method names give the impression that they are related) to find out if some of the disagreements are explainable.

### 3.4.2 Results

After a more extensive inspection, some of the disagreements between profilers, seem to have a reasonable explanation. We will list three cases where the hottest method disagreement has a caller/callee relationship.

The graphs in this chapter are visualizations of a union of the top $n$ hottest methods of each profiler and how the percentages relate to the results of other profilers. Therefore, if we mention the top $n$ methods, the number of methods in the legend can be higher.

#### Profiler disagreement in DaCapo's Avrora

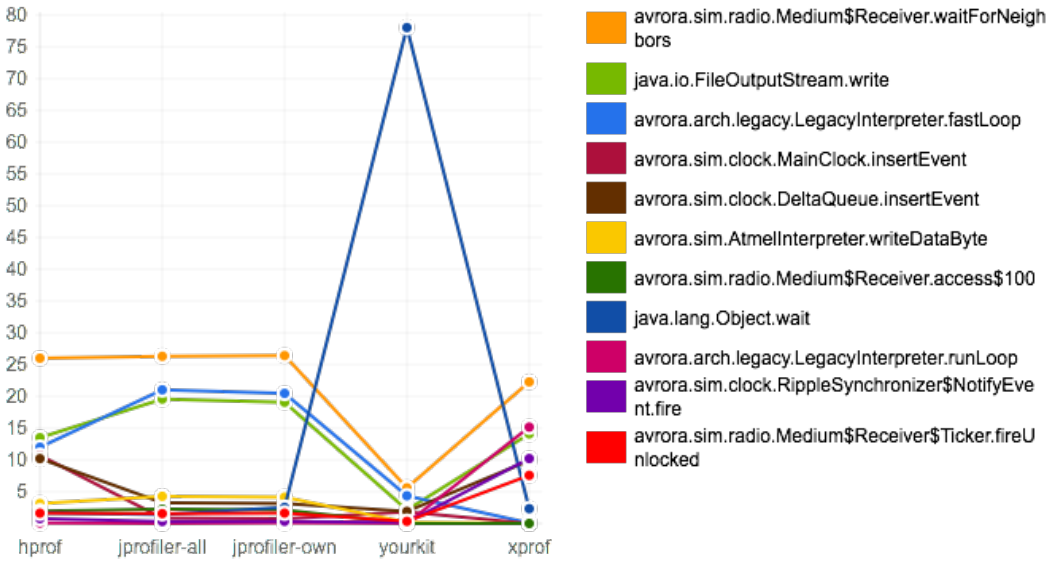



*Figure 3.2: Visualisation of the top 6 hottest methods of each profiler on the Avrora program.*

`Medium$Receiver.waitForNeigbors` is considered the hottest method according to Hprof, JProfiler and Xprof (all between 22% and 27%) but YourKit finds `java.lang.Object.wait` to be the hottest method with 78% (Appendix I.6).

Code inspection showed that `Medium$Receiver.waitForNeigbors` calls an abstract `Synchroniser` class for its waitForNeigbors method. The RippleSynchronizer and BarrierSynchronizer `waitForNeigbors` methods for example contain the `synchronized` statement. The `synchronized` statement uses the `Object.wait` method if another thread is active in the marked code [1] [2].

The code of `Medium$Receiver.waitForNeigbors` (listed below) seems unlikely to be as hot as the profilers think. YourKit is more likely correct although we believe a better prediction would be the method containing the `synchronized` statement as `Object.wait` will be very hard to trace back to the caller.

```
1  // public static abstract class Receiver extends Medium.TXRX
2  private void waitForNeighbors(long gtime) {
3    if (this.medium.synch != null) this.medium.synch.waitForNeighbors(gtime);
4  }
```

#### Profiler disagreement in DaCapo's H2

The profilers Hprof, Xprof and JProfiler (without filter) agree that method `BaseIndex.compareRows` is the hottest or close to hottest method of the H2 program when we profiled it with inlining disabled. YourKit however gave only a few percentages to this method and instead blames `Row.getValue` to be the hottest method.

[1] `http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object`
[2] `http://en.wikibooks.org/wiki/Java_Programming/Keywords/synchronized`

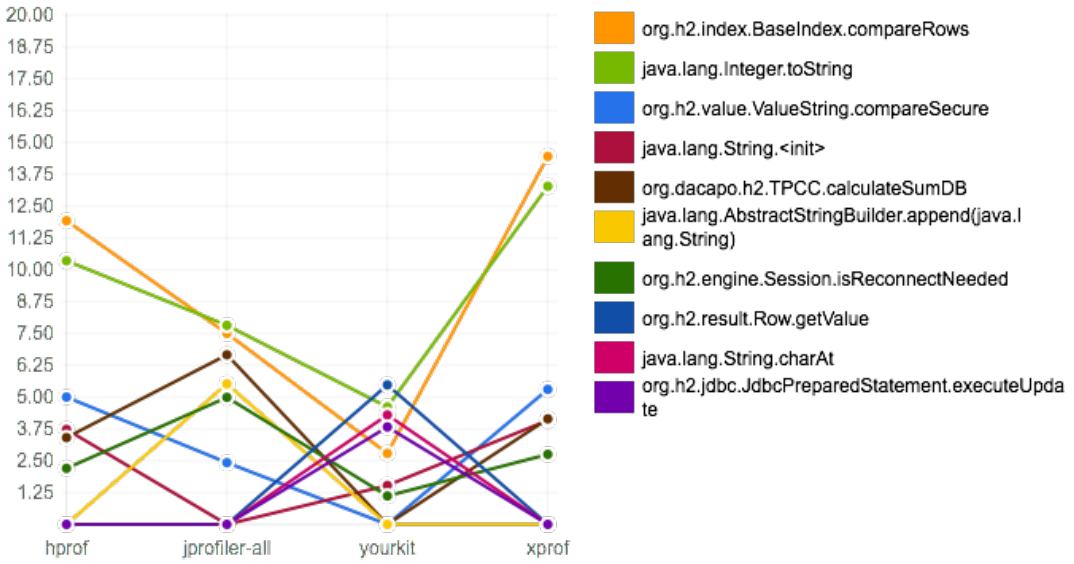



*Figure 3.3: Visualisation of the top 5 hottest methods of the H2 application profiled without the inlining optimization (excluding JProfiler with own method filter).*

After looking into the code (listed below) we found that the `compareRows` method uses the `getValue` method of the abstract interface `SearchRow` which gets implemented by `Row` class.

BaseIndex.compareRows

```
1  public int compareRows(SearchRow paramSearchRow1, SearchRow paramSearchRow2) throws
       SQLException {
2    for (int i = 0; i < this.indexColumns.length; i++) {
3      int j = this.columnIds[i];
4      Value localValue = paramSearchRow2.getValue(j);
5      if (localValue == null)
6      {
7        return 0;
8      }
9      int k = compareValues(paramSearchRow1.getValue(j), localValue,
           this.indexColumns[i].sortType);
10     if (k != 0) {
11       return k;
12     }
13   }
14   return 0;
15 }
```

Row.getValue

```
1  private final Value[] data;
2  // ...
3  public Value getValue(int paramInt) {
4    return this.data[paramInt];
5  }
```

### Profiler disagreement in DaCapo's Eclipse

The profilers Hprof, Xprof and JProfiler agree that `SimpleWordSet.add` is the hottest method of Eclipse but YourKit finds `CharOperation.equals` to be the hottest and sees the `SimpleWordSet.add` as a far less hot method.

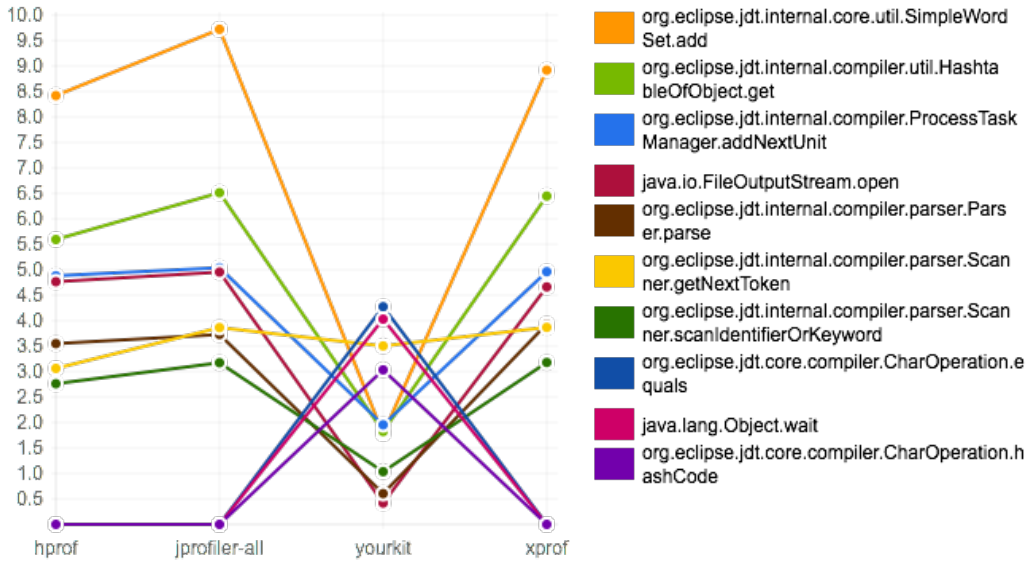



*Figure 3.4: Visualisation of the top 7 hottest methods of the Eclipse application profiled without the inlining optimization (excluding JProfiler with own method filter).*

The code of the two methods is displayed below: Looking into the code (listed below) we found that the `equals` method gets called by the `add` method, which might explain the difference between the profilers.

SimpleWordSet.add

```
1  public char[] add(char[] word) {
2    int length = this.words.length;
3    int index = CharOperation.hashCode(word) % length;
4    char[] current;
5    while ((current = this.words[index]) != null)
6    {
7      char[] current;
8      if (CharOperation.equals(current, word)) return current;
9      index++; if (index != length) continue; index = 0;
10   }
11   this.words[index] = word;
12
13   if (++this.elementSize > this.threshold) rehash();
14   return word;
15 }
```

CharOperation.equals

```
1  public static final boolean equals(char[] first, char[] second)
2  {
3    if (first == second)
4      return true;
5    if ((first == null) || (second == null))
6      return false;
7    if (first.length != second.length) {
8      return false;
9    }
10   int i = first.length;
11   do { if (first[i] != second[i])
12       return false;
13     i--; } while (i >= 0);
14
15   return true;
16 }
```

*Side note: The code samples in this section are reverse engineered from .class from the jar files included in the DaCapo Benchmark Suite by JD-GUI and therefore missing comments and might look different from the original .java files.*

### 3.4.3 Analysis

The investigated programs, all with a minor disagreement, reported methods which had a close relation to one another. The different methods call each other, either directly or indirectly.

The results from the Avrora program are somewhat strange and we suspect to be an example of yield point inaccuracy which we will describe in chapter 3.6 as the majority of the profilers mark a one line abstract method as hottest.

The results of H2 and Eclipse are clearly examples of caller/callee relationships. Strangely the methods that YourKit found were not seen as hot by the other profilers, which could be a depth setting in the profilers although its very unlikely that three different profilers would have this setting and also in the same depth.

This shows that not all disagreements are bad, although a profiler would technically be wrong, it might be close enough to the hot method that the actual hot method will be found.

### 3.4.4 Threats to validity

Possible threats to validity are equal to the previous chapters and can be found in section 3.2.4.

## 3.5 Do profilers see their hottest methods increase after increasing their weight?

The ideal situation for testing a profiler's correctness would be to optimize the hottest method and validate it by re-profiling it. Unfortunately optimizing a method can be very hard and time consuming, it would be easier to add code to a hot method and validate if the profiler recognizes the increased hotness of that method.

### 3.5.1 Method

We will run a series of tests for hot methods found in the programs PMD and Sunflow from the DaCapo Benchmark Suite. For each hottest and second hottest PMD and Sunflow owned method, we will run 5 different tests, injecting an algorithm to calculate the first 100, 200, 300, 400 or 500 Fibonacci numbers and profile the altered versions of these programs.

If we compare the results of the different tests, the injected method should increase in hotness, while other methods' hotness should decrease.

After injecting the method, we believe 4 things can happen to the data:

- Result 1: The injected method increases in percentages while others decrease or stay the same. This confirms the hypothesis which could mean that the profiler was correct about the method.
- Result 2: The injected method increases in percentages but some other methods increase as well. This breaks the hypothesis as other methods should not change.
- Result 3: The injected method does not increase in percentages but some others do. This clearly breaks the hypothesis, and can be seen as a conformation that the profiler was wrong about a method.
- Result 4: Nothing changes or barely changes. This is the least conclusive, which means the injected algorithm didn't have enough impact to show differences in the results which likely means the method wasn't as hot as the profiler reported.

### 3.5.2 Results

**Results of injecting PMD**

| Profiler | Method<br>*(All net.sourceforge.pmd)* | Result 1 | Result 2 | Result 3 | Result 4 |
|---|---|---|---|---|---|
| Xprof | SimpleNode.findChildrenOfType | X | | | |
| | ClassScope.findVariableHere | | X | | |
| Xprof | AbstractJavaRule.visit | | | | X |
| *(no inlining)* | ClassScope.findVariableHere | | X | | |
| Hprof | ast.JavaParser.jj_3R_91 | X | | | |
| | ast.JavaParser.jj_3R_120 | | X | | |
| Hprof | ast.JavaParser.jj_3R_91 | | X | | |
| *(no inlining)* | ast.JavaParser.jj_3_41 | | | | X |
| JProfiler | SimpleNode.findChildrenOfType | X | | | |
| | ast.JavaParser.jj_3R_91 | X | | | |
| JProfiler | ast.JavaParser.jj_3R_179 | | X | | |
| *(no inlining)* | ClassScope.findVariableHere | | X | | |
| YourKit | JavaParser.jj_scan_token | | | X | |
| | JavaParser.jj_2_41 | | | | X |
| YourKit | JavaParser.jj_scan_token | X | | | |
| *(no inlining)* | JavaParser.jj_2_41 | | | | X |

**Results of injecting Sunflow**

| Profiler | Method<br>*(All org.sunflow.core)* | Result 1 | Result 2 | Result 3 | Result 4 |
|---|---|---|---|---|---|
| Xprof | accel.KDTree.intersect | | X | | |
| | accel.BoundingIntervalHierarchy.intersect | X | | | |
| Xprof | primitive.TriangleMesh$WaldTriangle.intersect * | | | | |
| *(no inlining)* | accel.KDTree.intersect | X | | | |
| Hprof | primitive.TriangleMesh.intersectPrimitive | X | | | |
| | accel.KDTree.intersect | | X | | |
| Hprof | primitive.TriangleMesh$WaldTriangle.intersect * | | | | |
| *(no inlining)* | accel.KDTree.intersect | X | | | |
| JProfiler | primitive.TriangleMesh.intersectPrimitive | X | | | |
| | accel.KDTree.intersect | | X | | |
| JProfiler | primitive.TriangleMesh$WaldTriangle.intersect * | | | | |
| *(no inlining)* | accel.KDTree.intersect | X | | | |
| YourKit | primitive.TriangleMesh.intersectPrimitive | X | | | |
| | accel.KDTree.intersect | | X | | |

** The results for the `TriangleMesh$WaldTriangle.intersect` are not mentioned in this table because those tests weren't run. See the threats of validity for more details.*

The data which led to these conclusions are left out of this chapter to keep it short, but can be found in Appendix J and K.

### 3.5.3 Analysis

The injecting and re-profiling of the PMD and Sunflow programs had the total opposite results from one another.

The PMD results were often unclear. In some cases the injected method increased in hotness, but another increased just as much, or even more. In some cases the injected method increased and later decreased. In some cases nothing seemed to change.

The Sunflow injections however had a far clearer effect. Most of the injections had the expected result of increasing the injected method while others decrease (or at the very least stay the same),

and the remaining results the injected method increased and some others increased as well, but not in the phase the injected method increased.

The impact in execution time after injecting the method also differed. While the PMD results showed little overhead after being injected, the Sunflow injections had such a big effect that we had to stop the tests due to time limitations. The most extreme example was the injection of `TriangleMesh.intersectPrimitive` where a normal average run with xprof was around 35 seconds, but after injecting the 300 Fibonacci numbers algorithm, it increased to an average of 1182 seconds per run.

### 3.5.4 Threats to validity

A very important factor that can invalidate the results mentioned above is that the PMD injection tests have been done on a Quad core MacBook Pro running OSX (Machine A) while the Sunflow injection tests have been done on a Single core Ubuntu VPS (Machine C).

Because this is a totally different hardware platform, we ran the PMD program without injections on that same Ubuntu VPS and compared the top 5 hottest methods found on both platforms (Appendix I.4).

We found roughly the same methods on these two totally different environments, not always in the same order and not with the same percentages, but because of the variation in hardware this was expected. The similarity however, was enough for us to believe the injection results would be comparable on another platform.

A few test runs of the Sunflow injections on the MacBook Pro also showed an extreme increase in time just like on the Ubuntu VPS.

Due to a misconfiguration in the YourKit profiler with inlining disabled in the Sunflow program which we found later on, the results are not included. Because the normal run had nearly the same top 10 run, and was able to detect the injections, we assume the run with inlining disabled will also recognize it.

Because the injection had an extreme impact on the runtime of the tests (going from 2 hours for the not injected class, up to 2 weeks for an injected run), we had to stop the tests because of time limitations, for this reason, the `primitive.TriangleMesh$WaldTriangle.intersect` injections were not executed.

## 3.6 What could cause profilers to be wrong?

Sampling profilers base their results on the assumption that number of samples for a method is proportional to the time spent in the method, which can only be true if every part of the program has an equal probability to be profiled. Unfortunately a profiler cannot 'just' interrupt a program at any time it desires. Instead, the built-in stack trace sampling methods that sampling profilers use will only make stack trace sampling happen at *safe points*. Safe points (yield points) are places in the code of which the VM knows it can do a whole host of things safely (like initiate garbage collection). The locations of these safe points are determined by the JIT (Just In Time compiler), which often puts them in places not ideal for CPU profiling [7].

Yield points are not free and compilers often optimize their placement or decide to omit yield points from a loop if it can establish that the loop will not do any allocation and will not run indefinitely [2].

### 3.6.1 Method

To demonstrate the impact of suboptimal placement and removal of yield points, we will reproduce the Hot and Cold program results from Mytkowicz's paper [2]. The hot method clearly accounts for most of the execution of this program while the cold accounts for almost none.

```
1  static int[] array = new int[1024];
2
3  public static void hot (int i) {
4    int ii = (i + 10 * 100) % array.length;
5    int jj = (ii + i / 33) % array.length;
```

```
6     if (ii < 0) ii = -ii;
7     if (jj < 0) jj = -jj;
8     array[ii] = array[jj] + 1;
9   }
10
11  public static void cold() {
12    for (int i = 0; i < Integer.MAX_VALUE; i++) {
13      hot(i);
14    }
15  }
```

The complete code can be found in Appendix H.

### 3.6.2 Results

Profiling the Hot and Cold program with the different profilers confirmed the results from Mytkowicz (With exception of the YourKit results when inlining was disabled which we will discuss in the threats to validity).

| | Hot | Cold | Main | Other |
|---|---|---|---|---|
| Hprof | 0,37% | 1,13% | 0% | 98,49% |
| Xprof | 0% | 1,09% | 0% | 98,91% |
| JProfiler | 0% | 0,4% | 0% | 99,6% |
| YourKit | 0% | 0,0003% | 0% | 99,9997% |
| Hprof (inlining disabled) | 0% | 100% | 0% | 0% |
| Xprof (inlining disabled) | 0% | 99,995% | 0% | 0,005% |
| JProfiler (inlining disabled) | 0% | 99,9996% | 0% | 0,0004% |
| YourKit (inlining disabled) | 98,1357% | 1,8635% | 0% | 0,0007% * |

*The other contains a collection of Java native functions like: `AbstractStringBuilder.<init>`, `StringBuilder.<init>`, `AbstractStringBuilder.append`, `PrintStream.println`, etc.*

The Xprof and Hprof profiler also display the number of samples taken, which was surprisingly low in their normal runs. Xprof took only 281 samples during the nearly 83 minute lasting test and Hprof took 265 samples during the 82,5 minute lasting test. This is a clear indicator of yield point omission.

### 3.6.3 Analysis

The results above show how the four sampling profilers are not able to detect the `hot` method as hottest (See threats to validity for the YourKit exception). When a profiler is not able to detect a method as obvious as the `hot` method due to the placement and optimization of yield points, how can one guarantee a realistic profile of a real world application where the hottest method will be far less obvious?

### 3.6.4 Threats to validity

The YourKit results when we disabled the inlining optimization gave some unexpected results. We think this can be explained because the YourKit profiler is believed to do its own bytecode re-writing, which also got noticed by Mytkowicz [2].

We were not able to profile the DaCapo Benchmark Suite with YourKit without using the `--no-validation` flag, which ignores the validation errors of DaCapo Benchmark Suite. These validations are based on line counts and byte counts. This made YourKit suspicious of doing bytecode rewriting as the other profilers were had no problems running the tests without this flag.

Another reason is that the YourKit results contain their own methods, which could only happen if the profiler injects some code of its own. For example the Lucene Index program's second hottest method, according to YourKit, is the `Table.createRow` from the `com.yourkit.probes` package (Appendix I.11).

To make sure enough samples were taken we ran the Hot and Cold loop 250 times, which took between 1 and 5 hours depending on the overhead of the profiler and the 'no inline' flag.

## 3.7 How does the execution of a Java program differ when profiling?

While yield points explain why profilers might produce wrong profiles, it does not explain why profilers disagree with each other. If the profilers all use the yield points for sampling, they should all be biased in the same way and thus produce the same data.

### 3.7.1 Method

To answer this question, we will sum existing research from Mytkowicz [13] to explain why profilers disagree with each other.

### 3.7.2 Results

Mytkowicz explains that, due to the presence of a sampling profiler, a program behaves different in two ways: (i) directly, because the presence of different profilers causes differently optimized code, and (ii) indirectly, because the presence of different profilers causes differently placed yield points. While (i) directly affects the programs performance, (ii) does not affect the program's performance, but it affects the location of the "probes" that measure performance.

They show how (ii) contributes (more) to disagreement than (i) by profiling 8 programs with Xprof and with Xprof together with either Hprof, JProfiler or YourKit. They visualize the absolute difference of the hottest method (the percentage rapported) of Xprof without another profiler (variable $x$ in the graph) and Xprof when ran together with another profiler (variable $y$ in the graph).

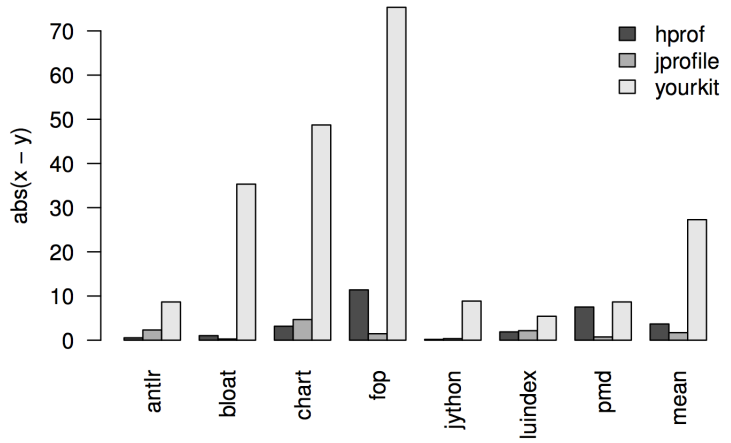



*Figure 3.5: Visualisation of the observer effect of a sampling profiler on a program (From [2])*

Another experiment which they did was to use a debug build of Hotspot to count the number of yield points the JIT places in a method. They found that, the number of yield points placed in a method, depends on which profiler is used. As an example they explain how profiling the Antlr program with Xprof gave them an average of 9 yield points per method for the hottest 10 methods while profiling with Hprof gave them an average of 7 yield points per method.

### 3.7.3 Analysis

That a profiler doesn't alter a program's code, doesn't mean it can't change the behavior of that program. When a profiler runs it launches some background threads which also can have effect on the memory layout. This can change the behavior of a program. Because profilers have different memory requirements and may perform different background activities, the effect of a program's

behavior differs between profilers. Because program behavior affects the JVM's dynamic optimization decisions, using a different profiler can lead to differences in compiled code [7, 13].

### 3.7.4 Threats to validity

To ensure that the profilers have enough samples and avoid start-up effects, all tests were executed 20 times on the same JVM invocation.

## 3.8 How should we use a sampling profiler to find bottlenecks?

Throughout this chapter, we have explained several concerns about sampling profilers. Compared to instrumenting profilers, sampling profilers have a very low overhead.

The problem with sampling profilers however is not their overhead, but the way they are implemented. Sampling profilers often do not agree with each other, and even though the disagreement is sometimes because of a caller/callee relation of methods, it does not change that they still disagree.

If profilers would only disagree on caller/callee method relations, they would still be useful as a performance analyst will probably look at the callers and callees to figure out what a method is doing. But after we increased the weight of the hottest methods of the PMD program, the number of times a profiler was able to detect an increase in a method's hotness was surprisingly low which suggests a profiler might have been wrong about that method.

Previous research showed that a sampling profiler's dependency on yield points prevents the code from having an equal probability to get sampled; together with the influence that a profiler's presence has on the JVM's optimization decisions makes their result unreliable. A sampling profiler is based on the assumption that the number of samples for a method is proportional to the time spent in the method, which can only be true if every part of the code has an equal probability to get sampled. But because of a sampling profiler relies on yield points, of which the placement will be optimized and sometimes omitted by the JVM and influenced by the presents of the profiler, this cannot be guaranteed.

Another point that we haven't emphasized in this thesis is that a sampling profiler's outcome can only be reliable if a high number of samples is taken. This limits the programs to those who spent a significant amount of time doing something CPU intensive.

This however does not mean that sampling profilers are unable to find performance bottlenecks. Looking at the Sunflow application the profilers agreed on most of each others results and the injection tests clearly showed that the hot methods were indeed hot (section 3.5.2). The problem we found is not that sampling profilers don't work, but that one can't be sure if their outcome is wrong or right.

If a sampling profiler is being used to find a performance bottleneck, keep in mind that the profiler could be wrong just as easy as it could be right. It is important to validate if the profiler is right or wrong by, for example, testing the profiler's output by increasing the weight of the method and testing if the profiler recognizes the increase in hotness like we did in section 3.5.

# 4 Another approach to sampling

Current sampling profilers are believed to be inaccurate because of their dependency and influence on yield points which has effect on the samples that those profilers try to take. Mytkowicz introduced another way to build a sampling profiler, which does not suffer from the inaccuracies caused by yield point dependency. He made a proof of concept profiler, called tprof, which does not use the JVMTI (Java Virtual Machine Tool Interface) to take its samples. Instead it uses standard UNIX signals to pause the Java application thread and take a sample of the current executing method. This technique is found to be effective, as shown in [2].

Unfortunately tprof is no longer available and therefore can't be tested. Manson luckily found this approach promising [7] and recently made a proof of concept profiler with some similarities to tprof called LightWeight profiler[1].

## 4.1 Is LightWeight profiler able to detect the hot method in the HotAndCold test?

Section 3.6 showed how the traditional sampling profiler suffers from being limited to only profile at yield points. If this approach of sampling does not suffer from yield point inaccuracies, it should be able to detect the `hot` method from the previously done Hot and Cold test.

### 4.1.1 Method

To test if LightWeight profiler does not suffer from suboptimal placement and removal of yield points, we will run the same Hot and Cold program as done in chapter 3.6 where the hot method clearly accounts for most of the execution of this program while the cold accounts for almost none.

If this profiling technique does not depend on yield points for measurement, it should be able to find the `hot` method and report it as hot.

### 4.1.2 Results

| | Hot | Cold | Main | Other |
|---|---|---|---|---|
| LightWeight | 23,43% | 74,83% | 0% | 1,67% * |
| LightWeight (inlining disabled) | 86,05% | 9,77% | 0% | 4,15% |

The results above show that both the normal and the run without the inlining optimization do find the `hot` method, but the normal run does not mark the `hot` as the hottest.

After inspecting the results we found that there is an explanation for the difference between the normal run and the run without the inlining optimization. The visualization below shows that the profiler does not find the `hot` method during the entire run.

[1] `https://code.google.com/p/lightweight-java-profiler/`

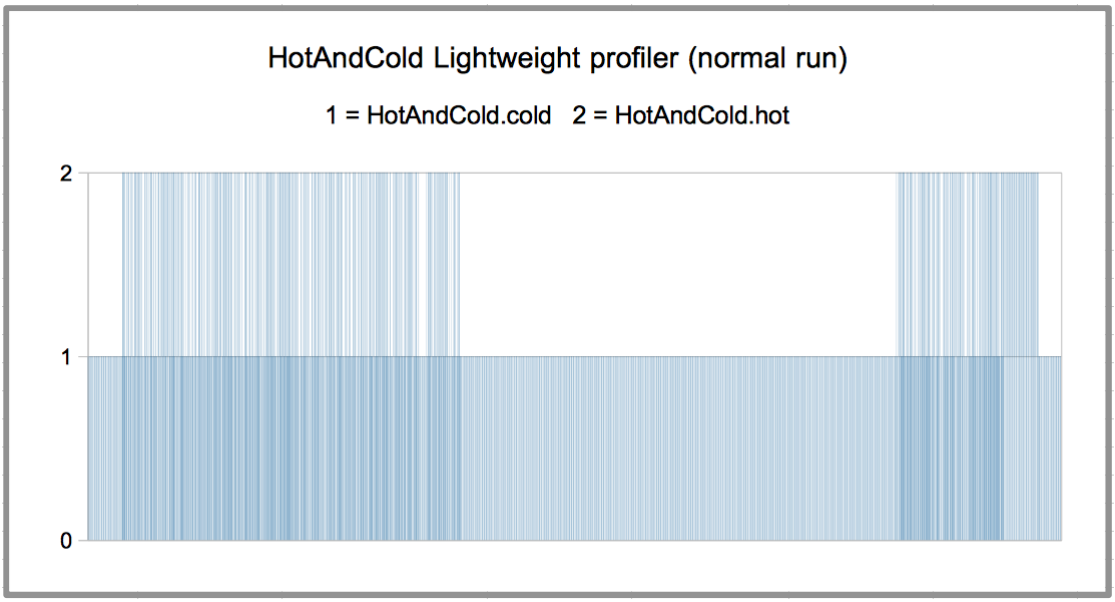



*Figure 4.1: Visualization of the spreading of found hot method samples during the Hot and Cold test running normally*

The visualization below shows that the test without the inlining optimization has a far better spreading of the hot method.

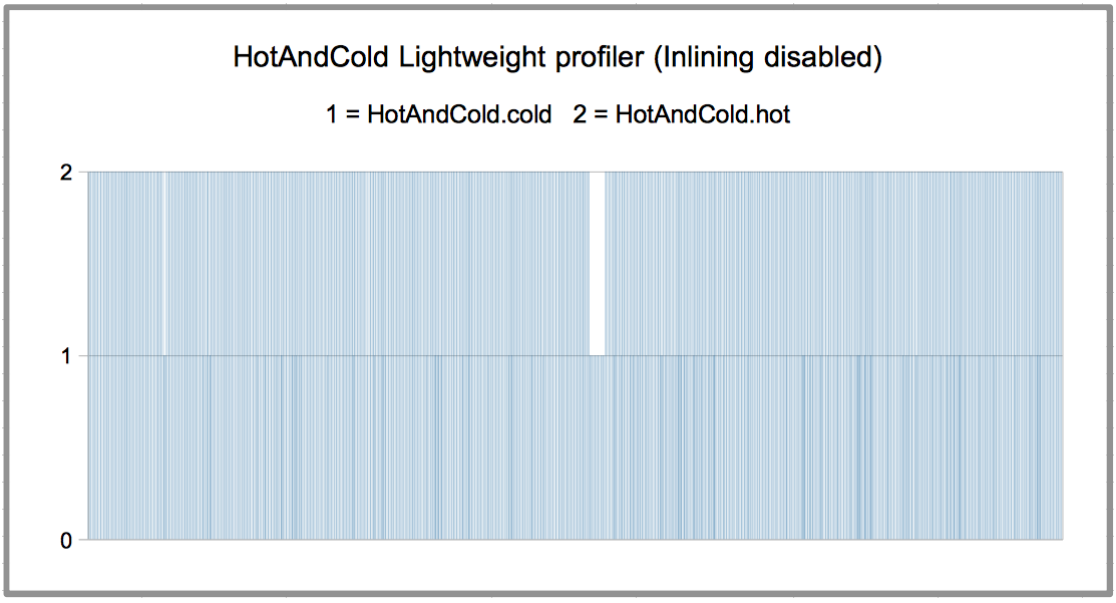



*Figure 4.2: Visualization of the spreading of found hot method samples during the Hot and Cold test running without the inlining optimization*

The two visualizations above show a clear difference in the spreading of found samples of the hot method. While the run without inlining seems to find the hot method during the entire run, the normal run has a large part where it isn't able to find the hot method.

The table below shows the number of samples found in the each of the runs:

| Method | Found samples in normal run | Found samples in run without inlining |
|---|---|---|
| HotAndCold.cold | 2245 | 308 |
| HotAndCold.hot | 703 | 2714 |
| . | 50 | 131 |
| Non Java thread (GC/JIT/pure native) | 2733 | 52 |
| In java code, unknown frame | 1232 | 719 |
| sun.misc.floatingDecimal.<clinit> | 1 | 0 |
| java.security.SecureClassLoader.getPermissions | 1 | 0 |
| sun.misc.URLClassPath$FileLoader.getResource | 0 | 1 |
| Total | 6965 | 3925 |

What stands out is that the normal run seems to find a lot more GC/JIT/pure native samples, which could explain the gap in the results of the normal run is due to some optimizations.

### 4.1.3 Analysis

The results of the Hot and Cold test with the inlining optimization were surprising at first. The profiler was able to detect the `hot` method far more often than the traditional profilers, but it was not the result we expected. Mytkowicz however explained in this paper that his profiler tprof had to be run with inlining disabled because of the limitation of reading the stack and optimizations. We were able to confirm that this is also true for LightWeight profiler after an inspection in the raw output which we visualized above.

### 4.1.4 Threats to validity

To make sure enough samples were taken we ran the Hot and Cold loop 250 times, which for both tests took at least an hour.

The results in this chapter are measured on the OpenJDK JVM running on a virtual Ubuntu (Machine B).

## 4.2 Do profilers see their hottest methods increase after increasing their weight?

Because this sampling approach had far better results in the Hot and Cold test, we decided to profile the PMD program and see how the results which this profiler found would be influenced by increasing its weight.

### 4.2.1 Method

We will run the PMD program from the DaCapo Benchmark Suite with the LightWeight profiler to find the two hottest methods of PMD with and without the inlining optimization. For each of the hottest methods, the Fibonacci algorithm calculating the first 100, 200, 300, 400 and 500 Fibonacci numbers will be injected and the program will be re-profiled.

If we compare the results of the different tests, the injected method should increase in hotness, while other method's hotness should decrease.

After injecting the method, we believe the same 4 things can happen to the data:

- Result 1: The injected method increases in percentages while others decrease or stay the same. This confirms the hypothesis which could mean that the profiler was correct about the method.
- Result 2: The injected method increases in percentages but some other methods increase as well. This breaks the hypothesis as other methods should not change.
- Result 3: The injected method does not increase in percentages but some others do. This clearly breaks the hypothesis, and can be seen as a conformation that the profiler was wrong about a method.

- Result 4: Nothing changes or barely changes. This is the least conclusive, which means the injected algorithm didn't have enough impact to show differences in the results which likely means the method wasn't as hot as the profiler reported.

### 4.2.2 Results

| Profiler | Method<br>*(All net.sourceforge.pmd)* | Result 1 | Result 2 | Result 3 | Result 4 |
|---|---|---|---|---|---|
| LightWeight | ast.SimpleJavaNode.childrenAccept | X | | | |
| | util.Benchmark.mark | | | X | |
| LightWeight | AbstractRuleChainVisitor.visitAll | | | X | |
| *(no inlining)* | ast.SimpleJavaNode.childrenAccept | | X | | |

The data that led to these conclusions are not included in this chapter, but can be found in Appendix J.8 and J.9.

### 4.2.3 Analysis

The results above show that the `ast.SimpleJavaNode.childrenAccept` was indeed a hot method. The other two methods however did not have the expected results.

Unfortunately the LightWeight profiler was very unstable which made running tests with it a very time consuming job and the amount of data we managed to obtain is not enough to base any conclusions on.

### 4.2.4 Threats to validity

To be sure our results are statistically valid we used the same preventative measures as described in section 3.2.4.

The tests of this chapter were ran on a virtual Ubuntu (Machine B) using a different JVM than the tests the traditional sampling profiler. There are to many different variables therefore we cannot make a trustworthy comparison between these results and the results from chapter 3.5.

## 4.3 Future work

We believe that even though the PMD injection tests were a bit disappointing, this profiling technique is promising. Unfortunately we were not able to test this profiler and profiling technique in more detail due to time limitations, but based on the results described by Mytkowicz on his profiler tprof [2], we believe this technique is worth investigating.

Mytkowicz explains that traditional sampling profilers have two major problems, the first being their dependency on yield points for taking samples and their second being the lack of a random factor in taking samples. Sampling profilers, including LightWeight profiler, only take samples on a fixed interval. If a profiler lacks a random factor, the interval could be scheduled together with other activities like thread switches which would bias the results [2, 23].

The profiler tprof, which took samples on interval $t + r$ (where $t$ is a fixed interval of 10 milliseconds and $r$ is a random value between -3 and 3 milliseconds), was able to find bottlenecks in two programs by not having yield point dependency and having a random factor. They however don't show how this profiler performs without the random factor.

We would like to see how the LightWeight profiler performs with a random factor in its intervals so a comparison can be made with and without it.

# 5 Conclusion

The research question was stated as follows: *How accurately do Java profilers predict runtime performance bottlenecks?*

There are two types of CPU profiling techniques currently used in Java: Instrumenting and Sampling. We have shown that instrumenting, due to its enormous overhead on performance, is not a well suited technique to find performance bottlenecks. The overhead will likely be too big to run this profiler in production environment and differences between methods might appear smaller and neglectable. More importantly, instrumenting profilers alter the program's (byte)code which it impacts the JVM's optimization decisions which will cause the program to behave different when being profiled.

Instrumenting may have very detailed reporting, which can be useful for understanding which methods call which, how often, etc. But we believe it is not suited for finding CPU performance bottlenecks.

Sampling has a far lower overhead on performance, which makes it possible to run this type of profiler in a production environment. Sampling profilers estimate their results based on samples which they take at a predefined interval, and rely on the hypothesis that the number of samples for a method is proportional to the time spent in a method. For this hypothesis to be true, each part of the code should have an equal probability to be sampled.

Unfortunately taking samples is not free, and cannot be done at every moment the profiler desires it. Sampling profilers depend on the JVMTI for collecting stack traces, which uses yield points (safe points in the code) for stack calls. Yield points are heavily optimized by the JVM and JIT to lower the performance penalty which impacts the profilers accuracy a lot. Previous research has shown that some JVMs do not only optimize their placement, but may also decide to skip yield points which will impact the profiler's accuracy even more.

This however does not mean that sampling profilers are wrong, we had some programs for which they were unable to predict bottlenecks, and others which they predicted bottlenecks very well. A simple way to test the trustworthiness of the profilers we used was to compare their hottest method with other profilers. We used 4 well known profilers on 11 programs, on which they agreed on the hottest methods of 3 programs. If two profilers show different results for the same program, only one (if any) can be right.

We cannot conclude with any reliability whether sampling profilers are suited for finding performance bottlenecks, due to the conflicting nature of our results. As such, when using a sampling profiler, one should keep in mind that the results could easily be wrong.

There is however previous research which explains an alternative to current sampling implementations, which might lend itself far better to finding performance bottlenecks in Java software. First introduced by Mytkowicz and used as a form of inspiration by Manson, which uses standard UNIX calls to interrupt the program from outside the JVM and therefore skips the yield point inaccuracy problem. It does however remove the platform independent advantage which existing Java profilers have and needs more research before we can conclude if this technique suitable for finding performance bottlenecks, but it gives some promising results which makes it an interesting candidate.

# 6 Future work

In the Sunflow program the sampling profilers were able to find bottlenecks which after injection had an enormous impact on the execution time of the tests. The exact data is not included in this thesis because we had to stop the tests from running. The most extreme method was the `TriangleMesh.intersectPrimitive` which before injection took around 2 hours to be profiled 1000 times in various profiling settings but we had to terminated while running with the Fibonacci algorithm (300 numbers) after being finished with the 100, and 200 because it was running for more than 2 weeks non stop.

This impact in time was far higher than the PMD program of which the highest increase in time (with 500 Fibonacci numbers) was around 5% (from 2550 ms to 2675 ms for `JavaParser.jj_3R_120`).

Were the bottlenecks in Sunflow so obvious that profilers just couldn't get it wrong or was the PMD program more likely to suffer from yield point inaccuracy and omitting like we saw in the Hot and Cold test?

We would recommend future work to find out what program implementations, or perhaps design patterns, prevent or help profilers in predicting bottlenecks. If we would know what implementations hinder a sampling profiler's accuracy, we would have more certainty about a profiler's results.

Another aspect of finding a bottleneck which we haven't discussed is the reporting of a profiler's findings. If we assume that there is a profiler which is able to profile an application in such a way that it is able to find the actual bottlenecks of that application. Then it still does not mean the bottleneck is found.

Profilers can be used for far more than only finding CPU performance bottlenecks, and all this data has to be presented in some way. Because a profiler's reporting tool is not specifically designed to find a CPU performance bottleneck, but rather to report whatever it is the profiler finds, it might report too much information, making it hard for a user to find the actual bottleneck.

Sampling profilers have a big downside of not knowing much of the code they are profiling, but since the user is likely to profile a program of which he has the source code, why not use that program's code as a source of information to help the profiler?

Profilers report bottlenecks on a 'per method' basis, but methods are not of equal length. This can be an unfair comparison which can lead the user away from a possible bottleneck. If method $\boldsymbol{a}$ consists of 50 lines of code and has the same hotness as method $\boldsymbol{b}$ which only has 4 lines of code, than each line of code (on average) of method $\boldsymbol{b}$ would be far hotter and could possibly be a bigger bottleneck than method $\boldsymbol{a}$ which might be more of a object oriented code smell.

Zaparanuks and Hauswirth introduced a profiler called AlgoProf [24], which searches for loops and recursions in programs to detect algorithms and estimated a cost function for each algorithm. They explain that a traditional profiler does not help in understanding how the cost was affected by the algorithm, the program input and the underlying implementation. Their approach addresses this limitation by providing a cost function that relates program input to algorithmic steps.

# Bibliography


[1] E. Machkasova, K. Arhelger, and F. Trinciante, "The observer effect of profiling on dynamic java optimizations," 2009.

[2] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of java profilers," 2010.

[3] M. Dmitriev, "Profiling java applications using code hotswapping and dynamic call graph revelation," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 139–150, jan 2004. .

[4] S. Liang and D. Viswanathan, "Comprehensive profiling support in the java virtual machine," 1999.

[5] "Jprofiler manual." http://resources.ej-technologies.com/jprofiler/help/doc/help.pdf, 2013.

[6] "Yourkit java profiler 12 help." http://www.yourkit.com/docs/java/help/index.jsp, 2013.

[7] J. Manson, "Why many profilers have serious problems (more on profiling with signals)." http://jeremymanson.blogspot.nl/2010/07/why-many-profilers-have-serious.html, August 2010. .

[8] "Java (programming language)." http://en.wikipedia.org/wiki/Java_(programming_language), May 2013. .

[9] J. Lewis and U. Neumann, "Performance of java versus c++." http://scribblethink.org/Computer/javaCbenchmark.html, 2004. .

[10] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan, "Java runtime systems: Characterization and architectural implications," 2001.

[11] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the ibm java just-in-time compiler," 2000.

[12] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the jalapeño jvm," *SIGPLAN Not.*, vol. 35, pp. 47–65, Oct. 2000.

[13] T. Mytkowicz, "Supporting experiments in computer systems research," Master's thesis, University of Colorado, 2010.

[14] I. Sjoblom, T. S. Snyder, and E. Machkasova, "Can you trust your jvm diagnostic tools," 2011.

[15] K. Arhelger, F. Trinciante, and E. Machkasova, "Use of profilers for studying java dynamic optimizations," 2009.

[16] J. Blech, L. Gesellensetter, and S. Glesner, "Formal verification of dead code elimination in isabelle/hol," 2005.

[17] D. Holmes, "Inside the hotspot vm: Clocks, timers and scheduling events - part 1 - windows." https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks, 2006 Oktober.

[18] M. Arnold, M. Hind, and B. G. Ryder, "An empirical study of selective optimization," in *In 13th International Workshop on Languages and Compilers for Parallel Computing*, 2000.

[19] B. Goetz, "Java theory and practice: Anatomy of a flawed microbenchmark," *IBM developerWorks*, 2005.

[20] M. Dmitriev, "Application of the hotswap technology to advanced profiling," 2002.

[21] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java benchmarking development and analysis (extended version)," Tech. Rep. TR-CS-06-01, 2006. http://www.dacapobench.org.

[22] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof - a call graph execution profiler," 1982.

[23] S. Mccanne and C. Torek, "A randomized sampling clock for cpu utilization estimation and code profiling," in *In Proc. Winter 1993 USENIX Conference*, pp. 387–394, USENIX Association, 1993. .

[24] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 67–76, ACM, 2012.

# A Efficiency of Time measurement in Java

In this thesis, some results are based on measuring the time it takes to execute micro tests. When measuring, it is important to know the inaccuracy of measuring time.

We have done some micro tests to find out how accurate Java measures time, so we know how large our tests should be for this inaccuracy to be negligible.

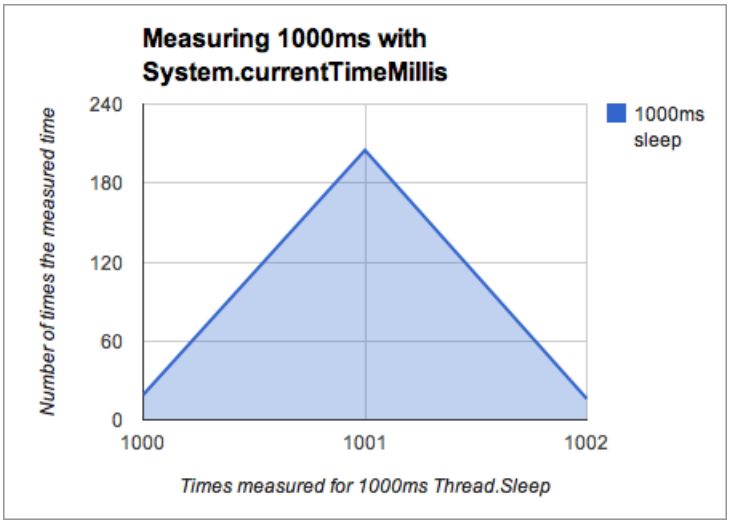
## Used code

Below you see the code used for the measuring.

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3  import java.util.HashMap;
4
5  public class TimeMeasurement {
6
7    private static final int loops = 240;
8    private static final int millisDelay = 1000;
9
10   public static void main(String[] args) {
11     HashMap<Long,Integer> times = new HashMap<Long,Integer>();
12     long total = 0;
13     try {
14       for (int i = 0; i < loops; i++) {
15 //       long t = runCurrentMillisTimeTest();
16         long t = runNanoTimeTest();
17         total += t;
18         if (times.containsKey(t)) {
19           times.put(t, times.get(t) + 1);
20         }
21         else {
22           times.put(t, 1);
23         }
24       }
25     }
26     catch(InterruptedException e) {
27       System.out.println(e.getMessage());
28     }
29     System.out.println(loops + " loops of " + millisDelay + "ms delay:");
30     ArrayList<Long> keys = new ArrayList<Long>(times.keySet());
31     Collections.sort(keys);
32     for (long key : keys) {
33       System.out.println("Time " + key + ": " + times.get(key));
34     }
35     System.out.println("Avg: " + ((double)total / loops));
36   }
37
38   public static Long runNanoTimeTest() throws InterruptedException {
39     Long startTime = System.nanoTime();
40     Thread.sleep(millisDelay);
41     Long endTime = System.nanoTime();
42     return endTime - startTime;
43   }
44
45   public static Long runCurrentMillisTimeTest() throws InterruptedException {
46     Long startTime = System.currentTimeMillis();
47     Thread.sleep(millisDelay);
48     Long endTime = System.currentTimeMillis();
49     return endTime - startTime;
50   }
51
52 }
```

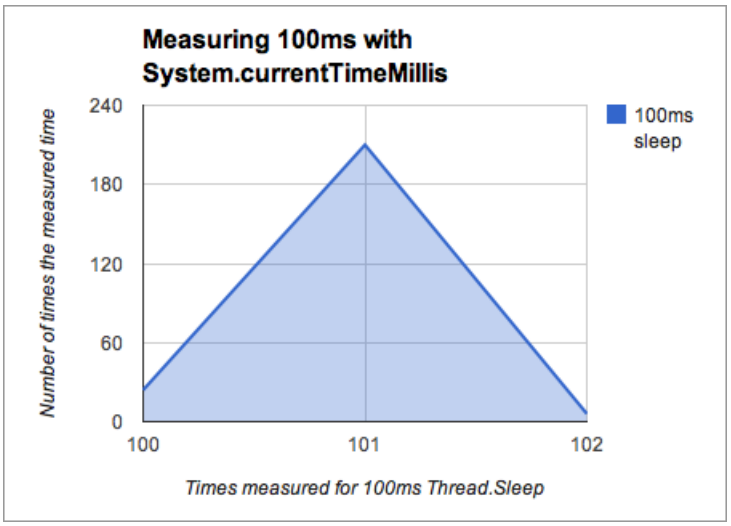
# Results

## 240 sleeps of 1000ms measured with System.currentTimeMillis



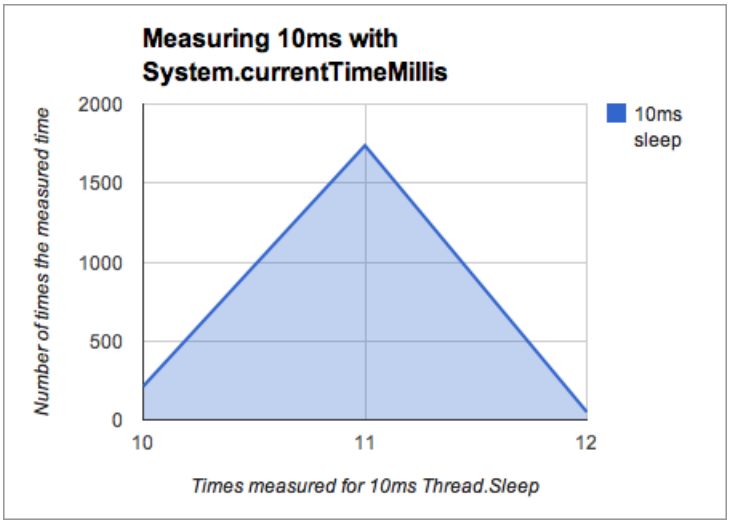

As seen in the graph above the majority of all runs are rounded to 1001ms which gives an 1ms additional runtime.
Average time is 1000,9875ms.

## 240 sleeps of 100ms measured with System.currentTimeMillis



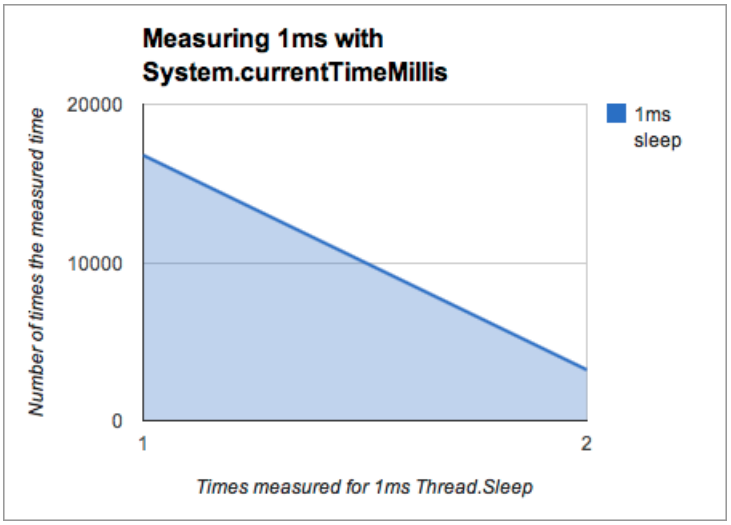

Same as with the 1000ms sleeps, most of the 100ms sleep runs take 101ms to run which also gives an 1ms additional runtime.
Average time is 100,925ms.

### 2000 sleeps of 10ms measured with System.currentTimeMillis



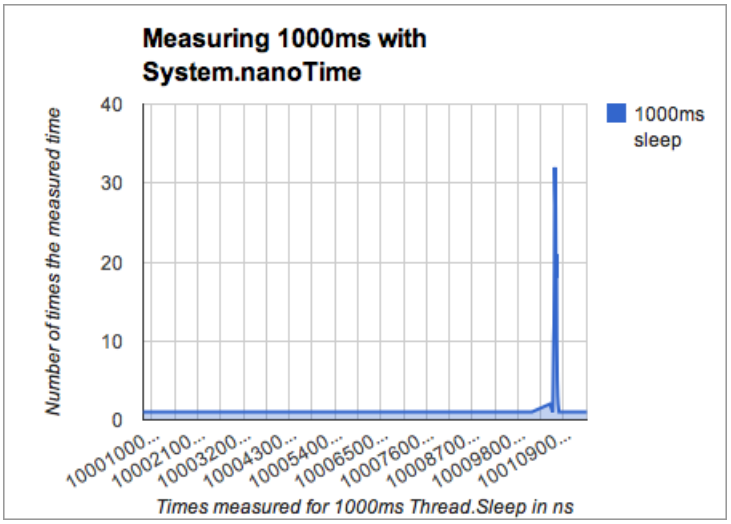

The 10ms sleep runs take about 11ms to run, which gives a comparable result as the 100 and 1000ms runs of 1ms additional runtime.
Average time is 10,919ms.

### 20000 sleeps of 1ms measured with System.currentTimeMillis



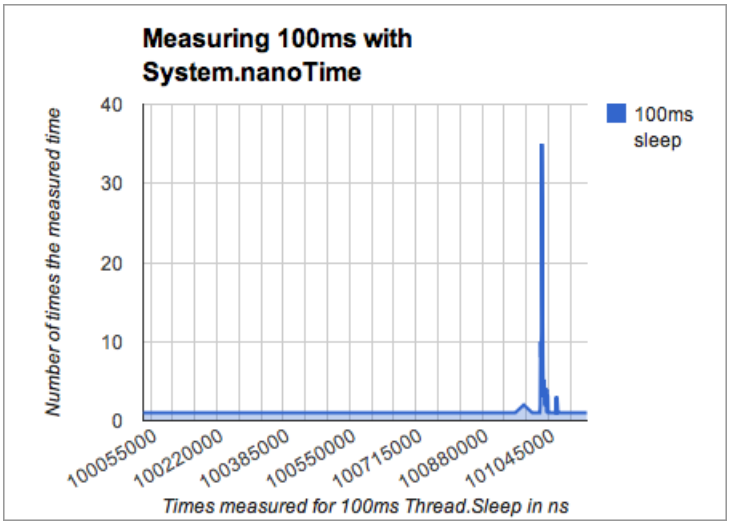

Different from other tests, the majority of 1ms sleep runs are rounded to 1ms runtime giving no overhead. Because of the rounding to milliseconds these results don't say anything, although I was hoping on a majority of 2ms because of an 1ms overhead found in all previous tests.
Average time is 1,1608ms.

### 240 sleeps of 1000ms measured with System.nanoTime



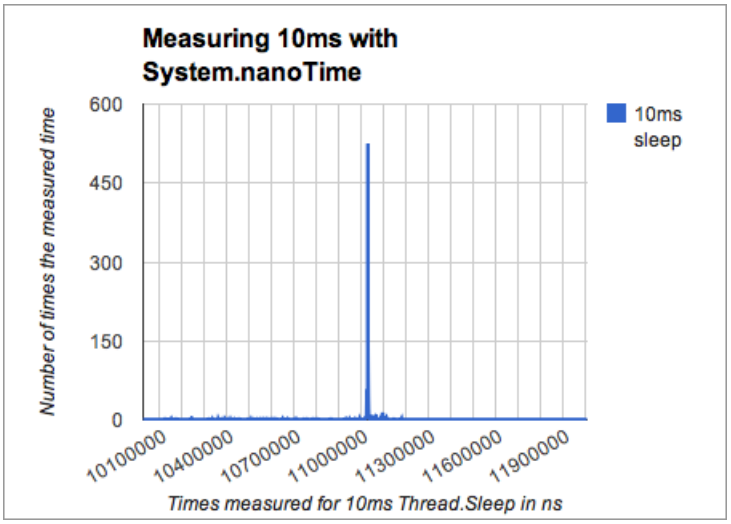

As seen in the graph above the vast majority of the results has around 1ms overhead. Because of the large nano time many different results are found and the graph is not as clear as with the millisecond times. To clarify the peak is starts at 1001065000 and ends with 1001075000.
Average time is 1000994096.

### 240 sleeps of 100ms measured with System.nanoTime



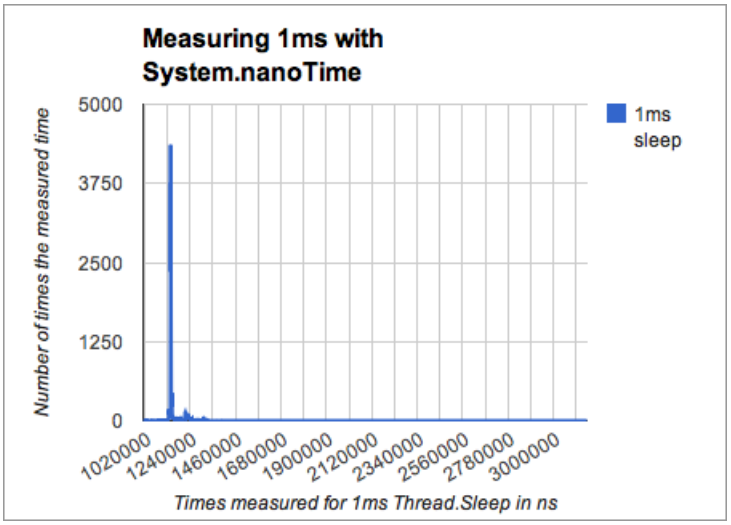

Comparable with the 1000ms sleeps, most of the 100ms runs are also a little over 1ms. The peak starts at 101026000 and ends at 101032000.
Average time is 100908296.

### 2000 sleeps of 10ms measured with System.nanoTime




The 10ms sleep runs also take a little over 1ms more then the time spent 'sleeping', to be exact the peak begins at 11022000 and ends at 11036000.
Average time is 10939243.

### 20000 sleeps of 1ms measured with System.nanoTime




Same as with the millisecond tests, most results are close to the 1ms. The peak starts at 1136000, reaches its highest by far at 1146000 and slowly decreases to until 1155000 with a small peak again at 1217000.
Average time is 1158916,75.

## Discussing results

The results above show that measuring time has some overhead. The different scopes all show that the overhead is close to 1 millisecond but can vary a little. Therefore we believe it is important to make the tests big enough for these differences to be negligible.

# B Source code CPU intensive test

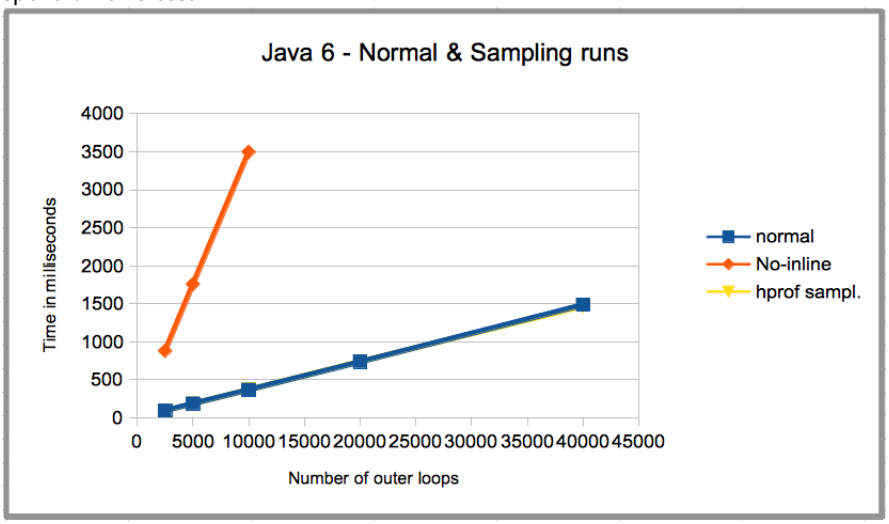
The code below is used in chapters 1.4.2, 2.1 and 3.1. All tests were run on Machine A.

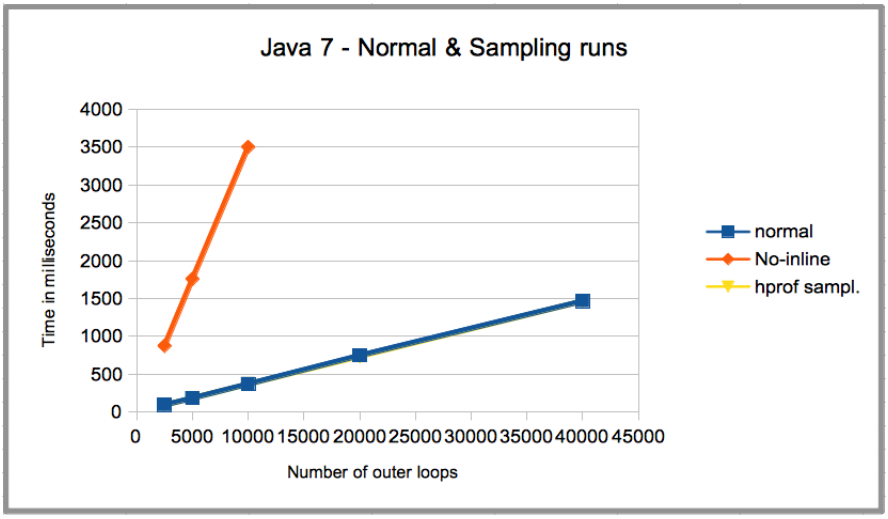
```
1  import java.util.Random;
2
3  import tools.Helpers;
4
5  public class DivideInlineShort {
6
7    public static void main(String[] args) {
8      int n = args.length > 0 ? Integer.parseInt(args[0]) : 10;
9      Random r = new Random();
10     for (int i = 0; i < 50; i++)
11       runTest(n, r.nextInt(8) + 1);
12   }
13
14   public static void runTest(int n, int ran) {
15     int counter = 0;
16     long timeStart = System.nanoTime();
17
18     for (int j = 0; j < n; j++)
19       for (int i = ran; i < 100000 + ran; i++)
20         counter += return1(i);
21
22     long timeEnd = System.nanoTime();
23     System.out.println(
24         Helpers.roundedTimeInMillis(timeStart, timeEnd) + "\t\t" +
25         counter);
26   }
27
28   public static int return1(int i) {
29     return i / i;
30   }
31 }
```

# C Results of linear growth test running CPU intensive runs

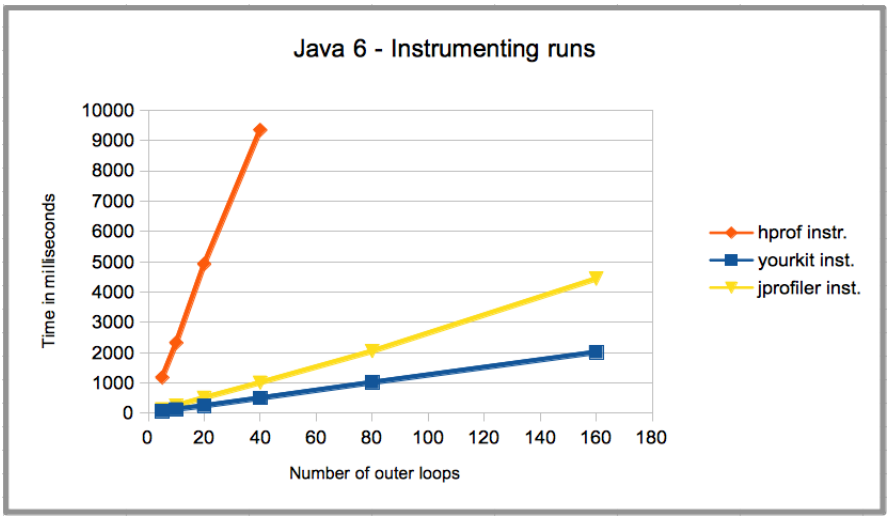
The results below are used in chapters 2.1 and 3.1. All tests were run on Machine A.

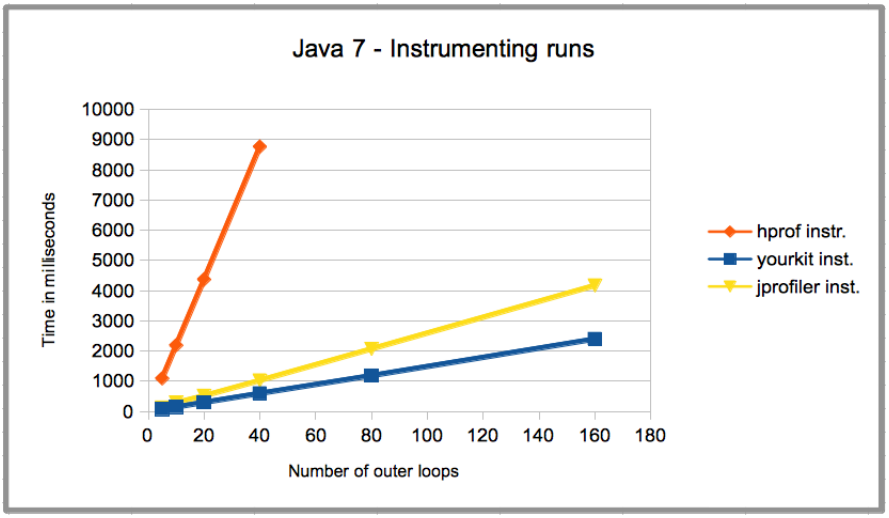
The results below show that normal runs and sampling runs increase linearly when the number of outer loops are increased. It shows that doubling the number of loops also doubles the total time spent on this test.




The results below show that instrumented runs also increase linearly when the number of outer loops are increased.


Java 6 - Instrumenting runs
Time in milliseconds
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
20
40
60
80
100
120
140
160
180
Number of outer loops
hprof instr.
yourkit inst.
jprofiler inst.


Java 7 - Instrumenting runs
Time in milliseconds
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
20
40
60
80
100
120
140
160
180
Number of outer loops
hprof instr.
yourkit inst.
jprofiler inst.

# D Source code IO intensive test

The code below is used in chapters 2.1 and 3.1. All tests were run on Machine A.

```
1  import java.io.BufferedReader;
2  import java.io.FileNotFoundException;
3  import java.io.FileReader;
4  import java.io.IOException;
5  import java.util.Random;
6
7  import tools.Helpers;
8
9  public class IOLoop {
10   private static final String relativeFileLocation = "/textfiles/";
11   private static final String fileName = "demo-file-";
12   private static final String fileExtention = ".txt";
13   private static final int numberOfLoops = 50;
14   private static final int numberOfFilesPerLoop = 2000;
15
16   public static void main(String[] args) {
17     IOLoop l = new IOLoop();
18     // If the string 'createfiles' is passed first generate all files
19     if (args.length > 0 && args[0].equals("createfiles")) {
20       Helpers h = new Helpers();
21       h.createTextFiles(numberOfFilesPerLoop * numberOfLoops,
22           Helpers.getCurrentDirectory(l),
23           relativeFileLocation,
24           fileName,
25           fileExtention);
26     }
27     l.runTest();
28   }
29
30   public void runTest() {
31     Random r = new Random();
32     String path = Helpers.getCurrentDirectory(this) + relativeFileLocation + fileName;
33
34     for (int j = 0; j < numberOfLoops; j++) {
35       String s = "";
36       long startTime = System.nanoTime();
37
38       for (int i = (j * numberOfFilesPerLoop); i < ((j + 1) * numberOfFilesPerLoop); i++) {
39         String fileContent = getFirstLineOfFile(path + i + fileExtention);
40         int randomPos = r.nextInt(fileContent.length() - 2);
41         s += fileContent.substring(randomPos, randomPos + 1);
42       }
43
44       long endTime = System.nanoTime();
45       System.out.println(
46           Helpers.roundedTimeInMillis(startTime, endTime) + "\t\t" +
47           (s.length() > 10 ? s.substring(0, 10) : s));
48     }
49   }
50
51   private String getFirstLineOfFile(String url) {
52     try {
53       BufferedReader br = new BufferedReader(new FileReader(url));
54       String t, text = "";
55       if ((t = br.readLine()) != null) {
56         text = t;
57       }
58       br.close();
59       return text;
60     }
61     catch (FileNotFoundException e) { e.printStackTrace(); }
62     catch (IOException e) { e.printStackTrace(); }
63     return "";
64   }
65 }
```

# E Source code CPU and IO intensive test

The code below is used in chapter 2.1. All tests were run on Machine A.

```
1  import java.io.BufferedReader;
2  import java.io.FileNotFoundException;
3  import java.io.FileReader;
4  import java.io.IOException;
5  import java.util.Random;
6  import tools.Helpers;
7
8  public class DivideAndIOTest {
9
10   private static final String relativeFileLocation = "/textfiles/";
11   private static final String fileName = "demo-file-";
12   private static final String fileExtention = ".txt";
13   private static final int numberOfLoops = 50;
14   private static int numberOfDividesPerLoop = 1000;
15   private static int numberOfFilesPerIOLoop = 2000;
16
17   public static void main(String[] args) {
18     DivideAndIOTest di = new DivideAndIOTest();
19     // If the string 'createfiles' is passed first generate all files
20     if (args.length > 0) {
21       if (args[0].equals("createfiles")) {
22         Helpers h = new Helpers();
23         h.createTextFiles(numberOfFilesPerIOLoop * numberOfLoops,
24             Helpers.getCurrentDirectory(di),
25             relativeFileLocation,
26             fileName,
27             fileExtention);
28       }
29       else if (args.length > 1) {
30         numberOfDividesPerLoop = Integer.parseInt(args[0]);
31         numberOfFilesPerIOLoop = Integer.parseInt(args[1]);
32       }
33     }
34     di.runTest();
35   }
36
37   public void runTest() {
38     Random r = new Random();
39     String path = Helpers.getCurrentDirectory(this) + relativeFileLocation + fileName;
40
41     for (int i = 0; i < numberOfLoops; i++) {
42       // Loop preps:
43       String tempIO = "";
44       int tempDivide = 0;
45       int randomDivide = r.nextInt(8) + 1;
46
47       // Loop:
48       long startTime = System.nanoTime();
49
50       for (int j = (i * numberOfFilesPerIOLoop); j < ((i + 1) * numberOfFilesPerIOLoop); j++) {
51         String fileContent = getFirstLineOfFile(path + j + fileExtention);
52         int randomPos = r.nextInt(fileContent.length() - 2);
53         tempIO += fileContent.substring(randomPos, randomPos + 1);
54       }
55
56       long betweenTime = System.nanoTime();
57
58       for (int j = 0; j < numberOfDividesPerLoop; j++) {
59         for (int k = randomDivide; k < (100000 + randomDivide); k++) {
60           tempDivide += return1(k);
61         }
62       }
63
64       long endTime = System.nanoTime();
65       // End loop, rest is anti-deadcode elimination and results
```

```
66        System.out.println(
67            " Total: " + Helpers.roundedTimeInMillis(startTime, endTime) + "\t\t" +
68            " IO: " + Helpers.roundedTimeInMillis(startTime, betweenTime) + "\t\t" +
69            " CPU: " + Helpers.roundedTimeInMillis(betweenTime, endTime) + "\t\t" +
70            (tempIO.length() > 10 ? tempIO.substring(0, 10) : tempIO) + " " +
71            tempDivide);
72      }
73    }
74
75    private String getFirstLineOfFile(String url) {
76      try {
77        BufferedReader br = new BufferedReader(new FileReader(url));
78        String t, text = "";
79        if ((t = br.readLine()) != null) {
80          text = t;
81        }
82        br.close();
83        return text;
84      }
85      catch (FileNotFoundException e) { e.printStackTrace(); }
86      catch (IOException e) { e.printStackTrace(); }
87      return "";
88    }
89
90    private int return1(int i) {
91      return i / i;
92    }
93
94  }
```

# F Source code instrumenting overhead demo code

The code below is used in chapter 2.2. All tests were run on Machine A.

```
1  import java.util.Random;
2
3  import tools.Helpers;
4
5  public class InstrumentingOverhead {
6
7    static final int calculationLoopSize = 2000000;
8    static final int numberOfRuns = 50;
9    static int[] array = new int[1024];
10   static double[][] data = new double[numberOfRuns][8];
11
12   public static void main(String[] args) {
13     Random r = new Random();
14     for (int i = 0; i < numberOfRuns; i++)
15       runTests(r.nextInt(10) + 1,i);
16     printData();
17   }
18
19   public static void runTests(int randomNumber, int cycle) {
20     long a,b,c,d = 0L;
21     a = System.nanoTime();
22     for (int i = randomNumber; i < calculationLoopSize; i++) simple(i);
23     b = System.nanoTime();
24     for (int i = randomNumber; i < calculationLoopSize; i++) middle(i);
25     c = System.nanoTime();
26     for (int i = randomNumber; i < calculationLoopSize; i++) hard(i);
27     d = System.nanoTime();
28     data[cycle][0] = Helpers.roundedTimeInMillis(a, b);
29     data[cycle][1] = Helpers.roundedTimeInMillis(b, c);
30     data[cycle][2] = Helpers.roundedTimeInMillis(c, d);
31     data[cycle][3] = Helpers.roundedTimeInMillis(a, d);
32   }
33
34   public static void simple(int i) {
35     i = i / 2 % array.length;
36     if (i < 0) i = -i;
37     array[i] = array[i] + 1;
38   }
39
40   public static void middle(int i) {
41     int ii = (i + 10) * 3 / 4 % array.length;
42     i = i % array.length;
43     if (ii < 0) ii = -ii;
44     if (i < 0) i = -i;
45     array[ii] = array[i] + 1;
46   }
47
48   public static void hard(int i) {
49     int ii = (i + 10 * 100) % array.length;
50     int jj = (ii + i / 33);
51     jj = jj / 2 * 4 / 3 * 5 / 13;
52     jj %= array.length;
53     i = i % array.length;
54     if (ii < 0) ii = -ii;
55     if (jj < 0) jj = -jj;
56     if (i < 0) i = -i;
57     array[ii] = array[jj] + 1;
58   }
59
60   public static void printData() {
61     System.out.print("easy: \t"); printRow(0);
```

```
62      System.out.print("middle: \t"); printRow(1);
63      System.out.print("hard: \t"); printRow(2);
64      System.out.print("total: \t"); printRow(3);
65    }
66
67    public static void printRow(int row) {
68      for (int i = 0; i < numberOfRuns; i++)
69        System.out.print(data[i][row] + "\t");
70      System.out.println("");
71    }
72  }
```

# G Instrumenting profiler results for the relative time per method of section 2.2

## Hprof instrumenting results

| # | Method | Percentage of time spent | Invocation count |
|---|---|---|---|
| 1 | InstrumentingOverhead.runTests() | 57,33% | 50 |
| 2 | InstrumentingOverhead.hard() | 14,33% | 99.999.725 |
| 3 | InstrumentingOverhead.middle() | 14,12% | 99.999.725 |
| 4 | InstrumentingOverhead.simple() | 14,10% | 99.999.725 |

## YourKit instrumenting results

| # | Method | Percentage of time spent | Time spent | Invocation count |
|---|---|---|---|---|
| 1 | InstrumentingOverhead.runTests() | 99% | 40.376 ms | 50 |

## YourKit instrumenting results (without ATM optimization)

| # | Method | Percentage of time spent | Time spent | Invocation count |
|---|---|---|---|---|
| 1 | InstrumentingOverhead.runTests() | 99% | 94.614 ms | 50 |
| 2 | InstrumentingOverhead.hard() | 0% | 101 ms | 99.999.735 |
| 3 | InstrumentingOverhead.middle() | 0% | 96 ms | 99.999.735 |
| 4 | InstrumentingOverhead.simple() | 0% | 96 ms | 99.999.735 |

## JProfiler instrumenting results

| # | Method | Percentage of time spent | Time spent | Invocation count |
|---|---|---|---|---|
| 1 | InstrumentingOverhead.runTests() | 51% | 39.323 ms | 50 |
| 2 | InstrumentingOverhead.hard() | 17% | 13.416 ms | 99.859.044 |
| 3 | InstrumentingOverhead.middle() | 15% | 11.897 ms | 97.999.731 |
| 4 | InstrumentingOverhead.simple() | 15% | 11.756 ms | 97.999.731 |

# H Source code Hot and Cold test

The code below is used in chapters 3.6 and 4.1. The tests for the traditional sampling profilers was run on Machine A, the tests for LightWeight profiler were run on Machine B.

```
1  import java.util.Random;
2
3  import tools.Helpers;
4
5  public class HotAndCold {
6
7    static int[] array = new int[1024];
8
9    public static void main(String[] args) {
10     int n = args.length > 0 ? Integer.parseInt(args[0]) : 5;
11     Random r = new Random();
12     for (int i = 0; i < n; i++)
13       cold(r.nextInt(n));
14   }
15
16   public static void hot (int i) {
17     int ii = (i + 10 * 100) % array.length;
18     int jj = (ii + i / 33) % array.length;
19     if (ii < 0) ii = -ii;
20     if (jj < 0) jj = -jj;
21     array[ii] = array[jj] + 1;
22   }
23
24   public static void cold(int start) {
25     System.out.println("begin");
26     long begin = System.nanoTime();
27     for (int i = start; i < Integer.MAX_VALUE; i++) {
28       hot(i);
29     }
30     long end = System.nanoTime();
31     System.out.println("end, total time: " + Helpers.roundedTimeInMillis(begin, end));
32   }
33 }
```

# I DaCapo profile results

## I.1 Design desicions

The problem with comparing results from one profiler with another is that each has its own output form, and some can be very limiting. For instance, Xprof and Hprof made the do not show the parameters nor line numbers of methods making it impossible to separate which method is called when a method name occurs multiple times in a program.

To be able to compare the big amount of data generated by the various tests automatically I've decided to remove line number and parameter references from methods. The downside of this implementation is that double methods are now summed together and making them look heavier. I've made this decision because often methods with a same name call each other and are often short. This however will affect designs like the visitor pattern. I believe that, when known, these few impacted results can be ignored by the programmer and will not affect the overall results.

## I.2 PMD (ran on quad-core OSX)

### I.2.1 Top methods compared

To visualize profiler disagreement we created graphs showing the top 3 methods of each profiler and where they end up at different profilers.

**Normal run:**

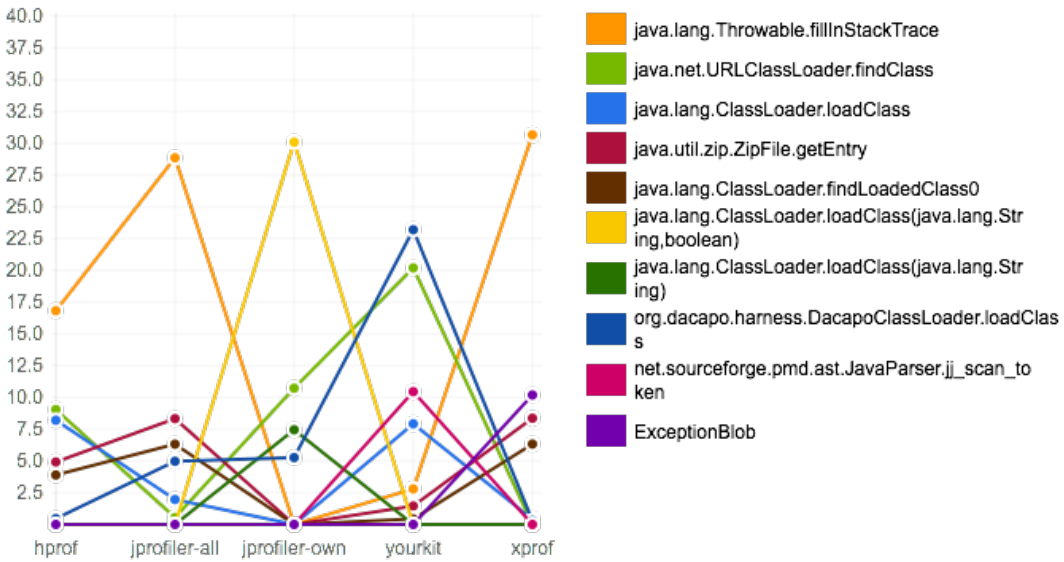



*Figure I.1: Top 3 hottest methods of PMD*

**Run with inlining disabled:**

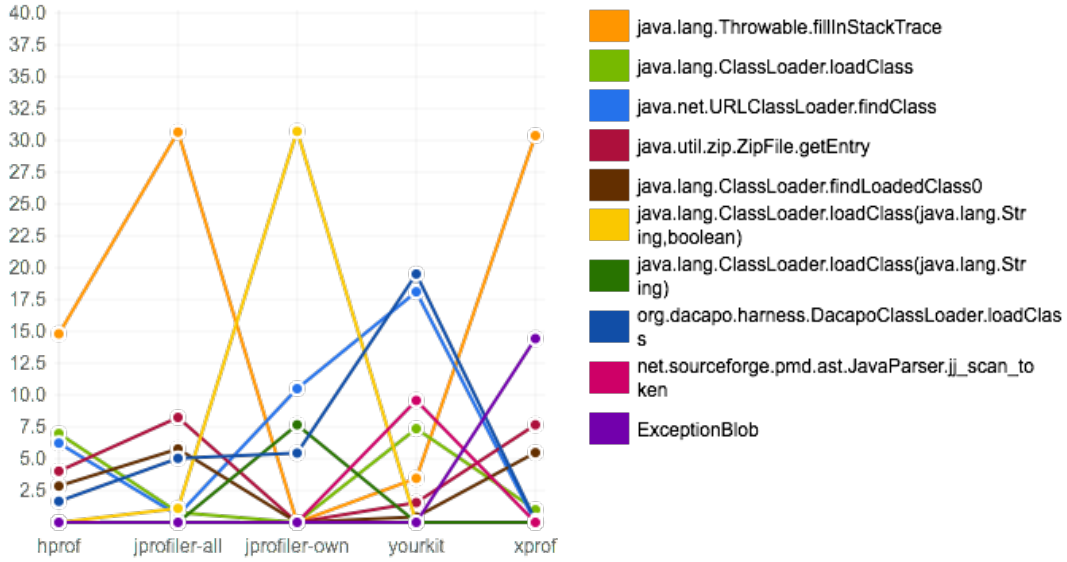



*Figure I.2: Top 3 hottest methods of PMD without inlining*

## I.3 PMD (ran on single-core Ubuntu)

### I.3.1 Top methods compared

To visualize profiler disagreement we created graphs showing the top 4 methods of each profiler and where they end up at different profilers.

**Normal run:**

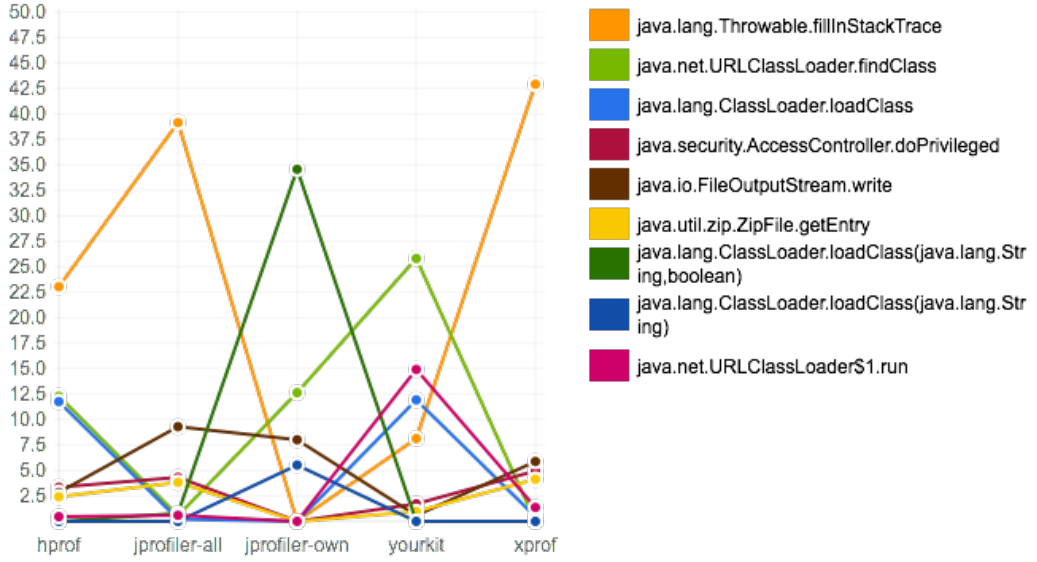



*Figure I.3: Top 4 hottest methods of PMD*

**Run with inlining disabled:**

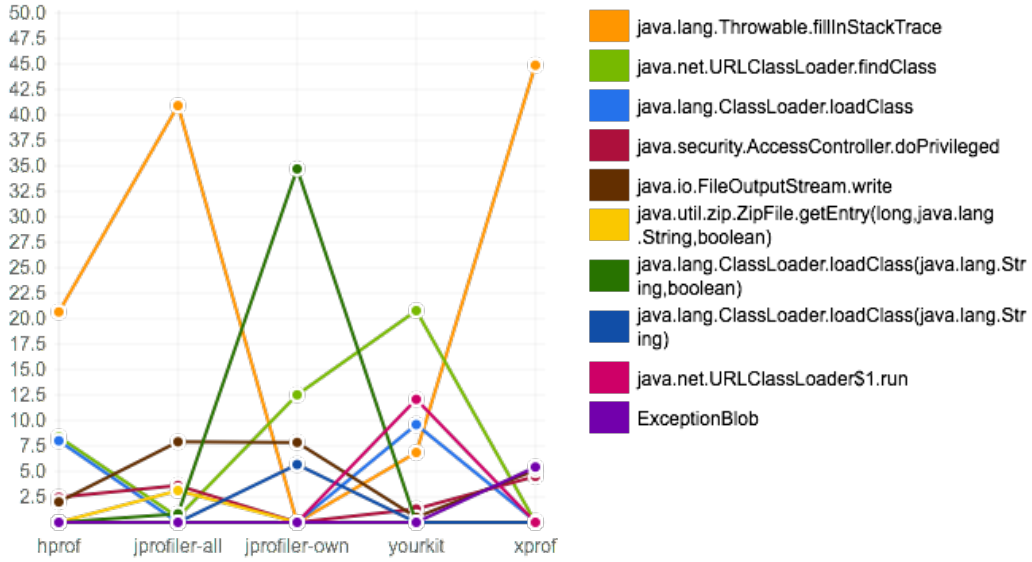



*Figure I.4: Top 4 hottest methods of PMD without inlining*

## I.4 PMD top 5 Mac osx quad-core vs Ubuntu single-core per profiler

**Xprof**

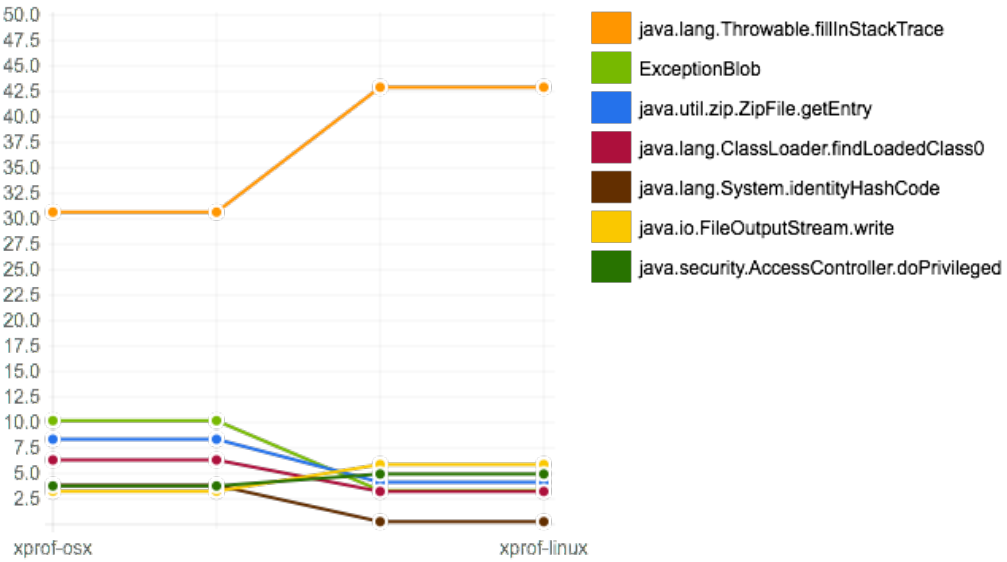



*Figure I.5: Top 5 hottest methods compared of PMD OSX vs Ubuntu*

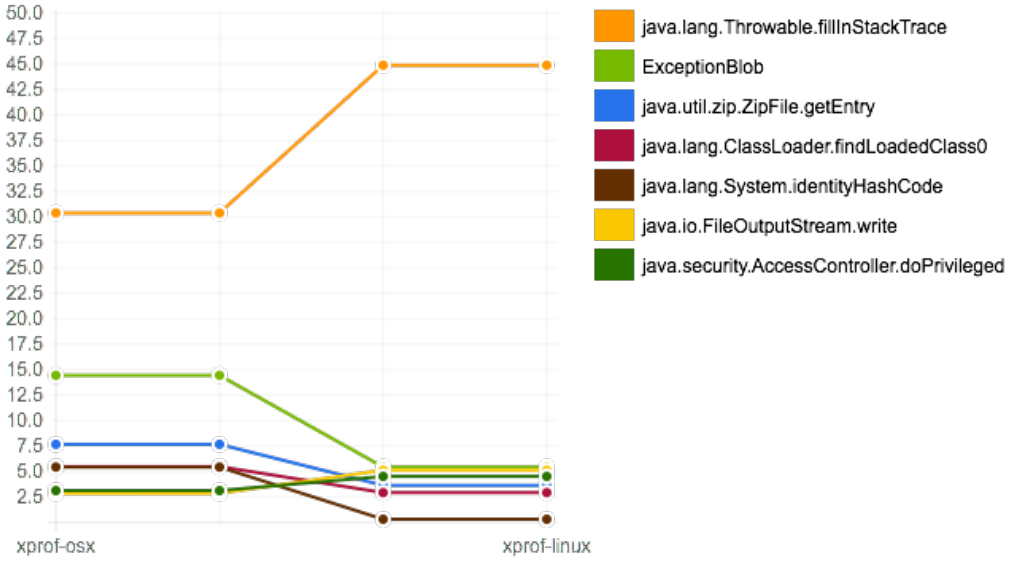



*Figure I.6: Top 5 hottest methods compared of PMD OSX vs Ubuntu without inlining*

**Hprof**

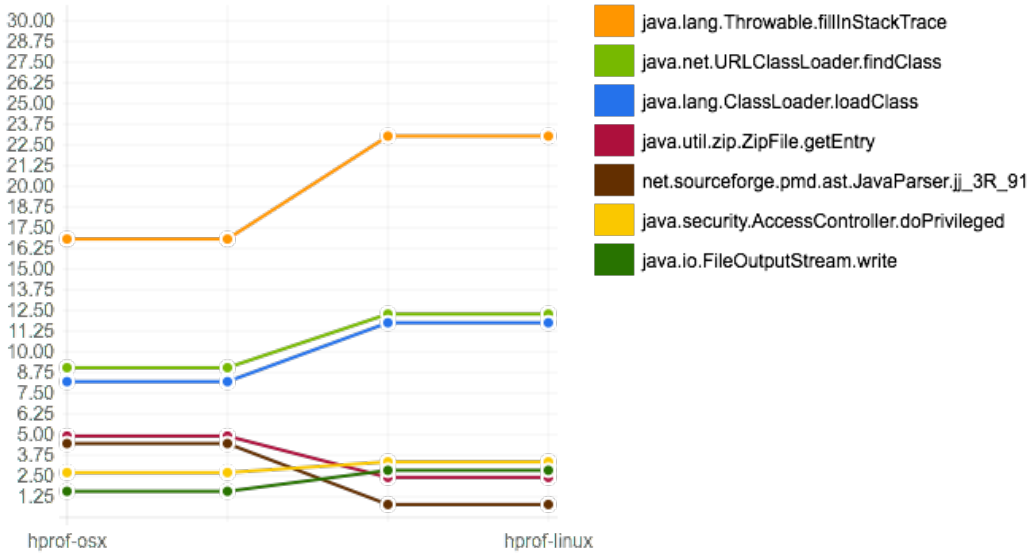



*Figure I.7: Top 5 hottest methods compared of PMD OSX vs Ubuntu*

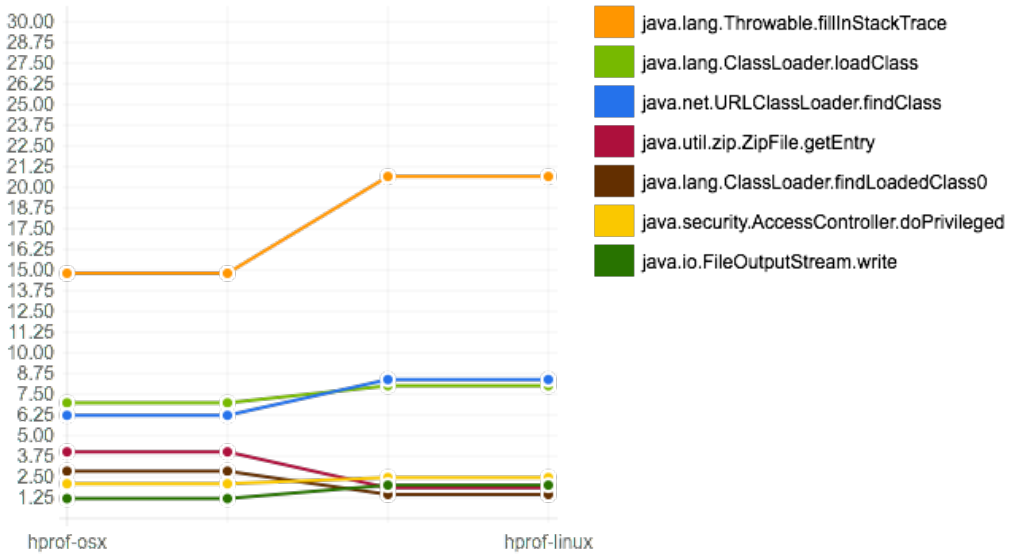



*Figure I.8: Top 5 hottest methods compared of PMD OSX vs Ubuntu without inlining*

**JProfiler (own methods filter)**

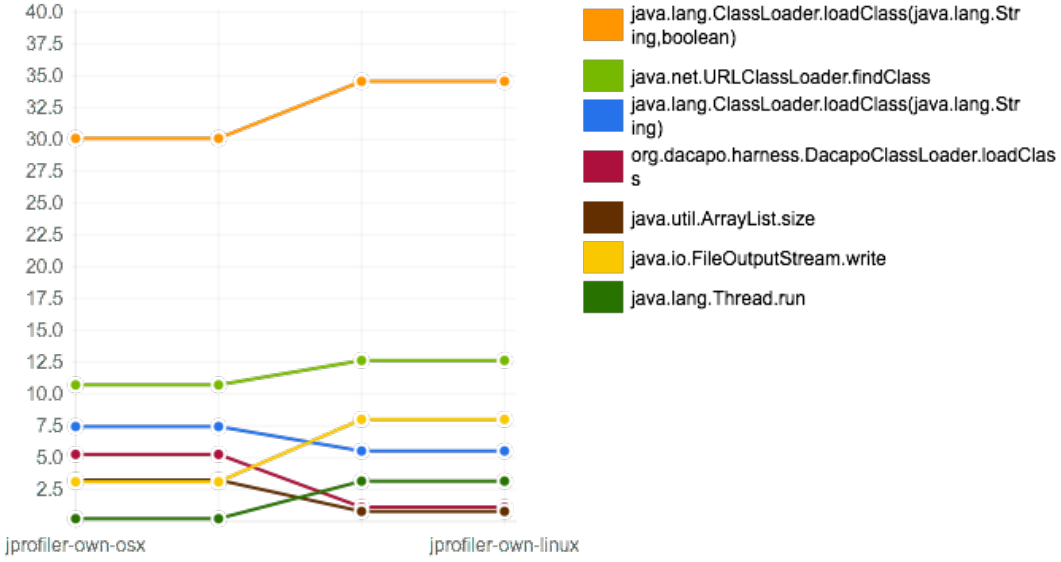



*Figure I.9: Top 5 hottest methods compared of PMD OSX vs Ubuntu*

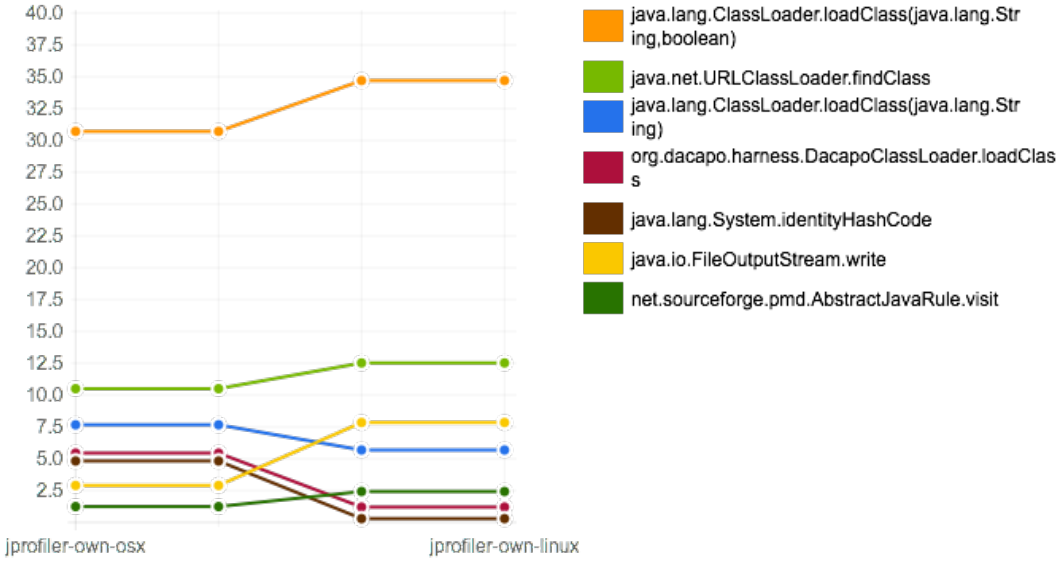



*Figure I.10: Top 5 hottest methods compared of PMD OSX vs Ubuntu without inlining*

**JProfiler (all methods)**

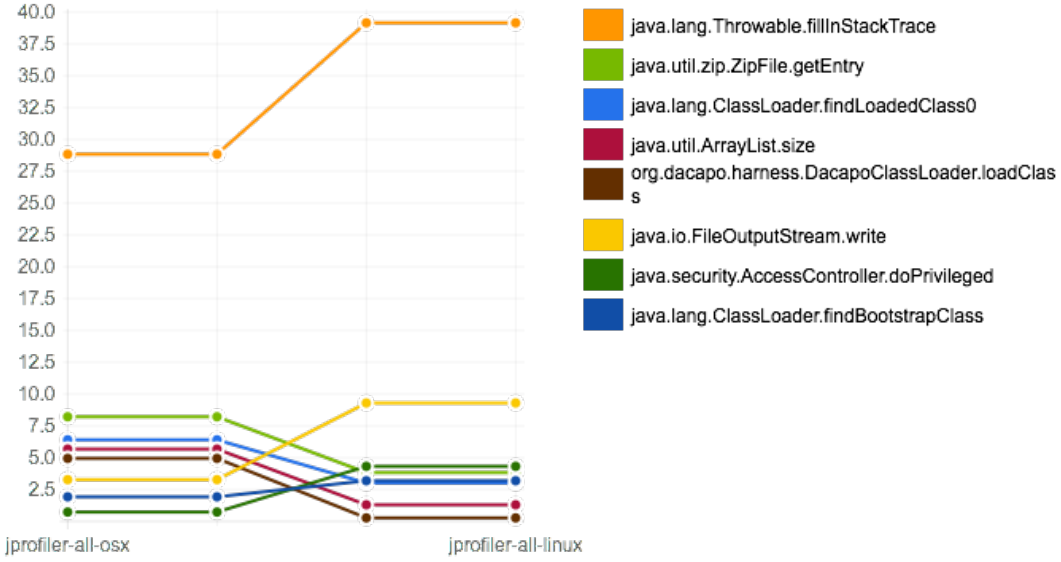



*Figure I.11: Top 5 hottest methods compared of PMD OSX vs Ubuntu*

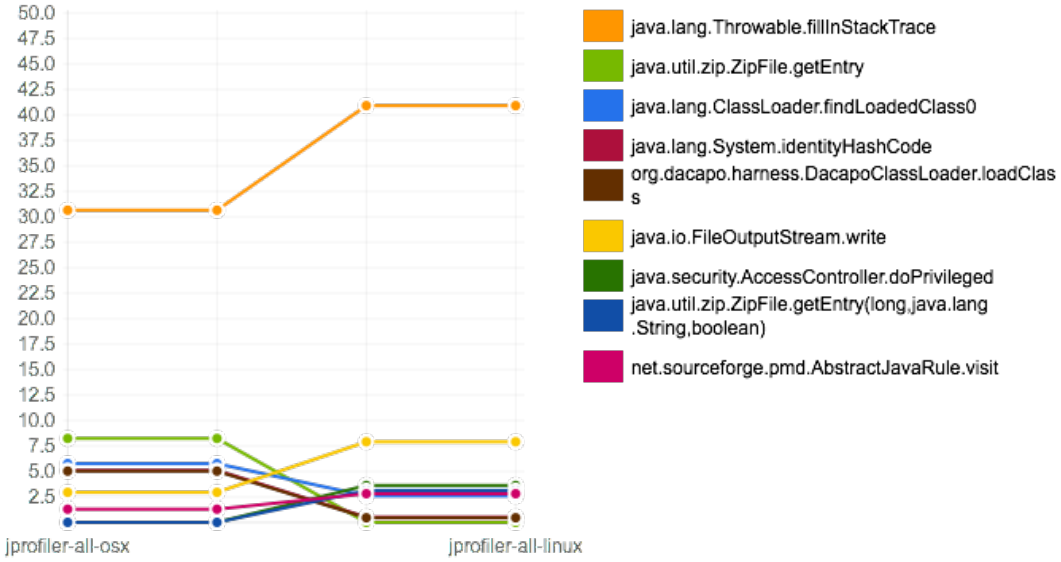



*Figure I.12: Top 5 hottest methods compared of PMD OSX vs Ubuntu without inlining*

**YourKit**

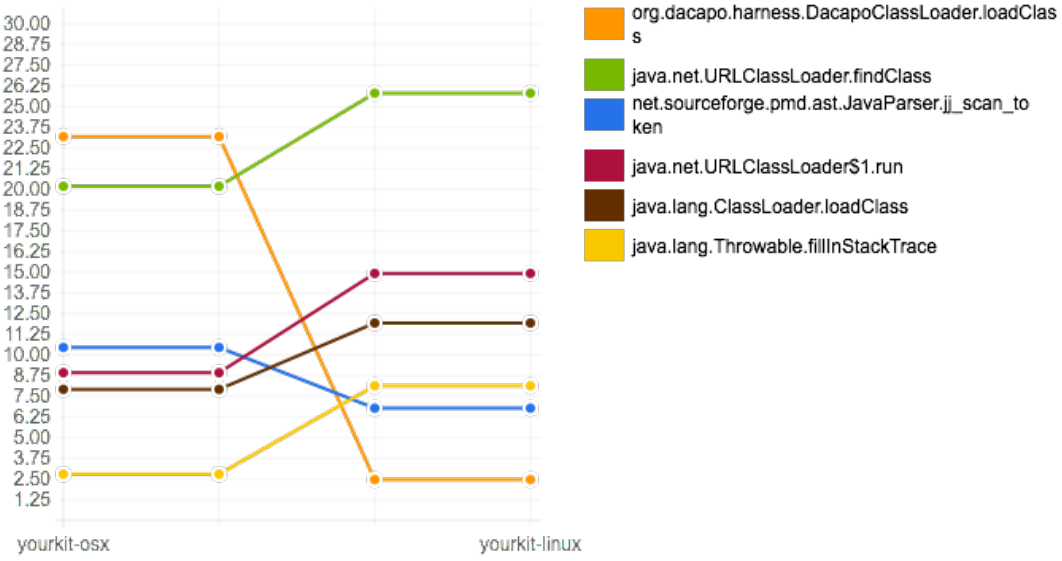



*Figure I.13: Top 5 hottest methods compared of PMD OSX vs Ubuntu*

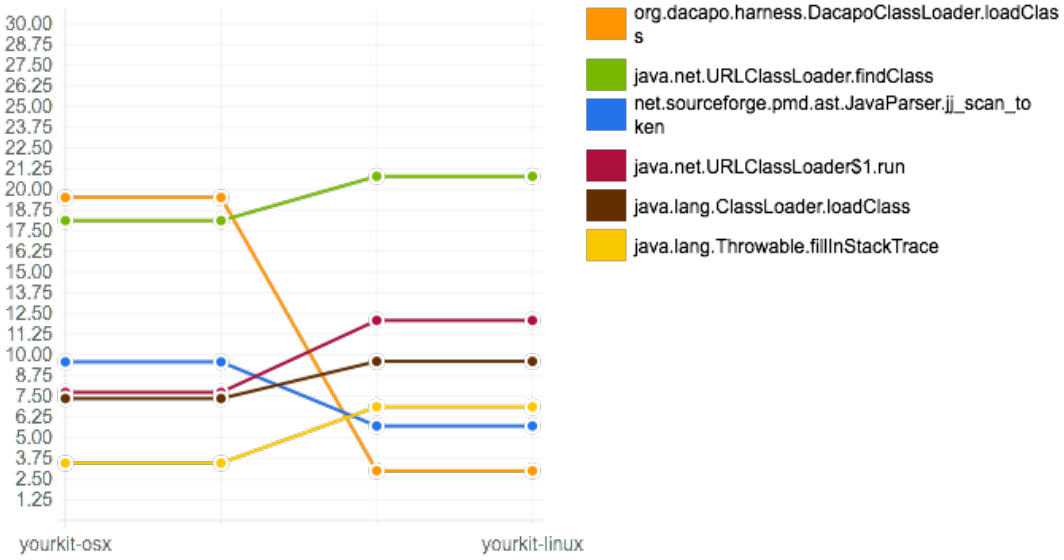



*Figure I.14: Top 5 hottest methods compared of PMD OSX vs Ubuntu without inlining*

## I.5 Sunflow (ran on single-core Ubuntu)

### I.5.1 Top methods compared

To visualize profiler disagreement we created graphs showing the top 4 and top 6 methods of each profiler and where they end up at different profilers.

**Normal run:**

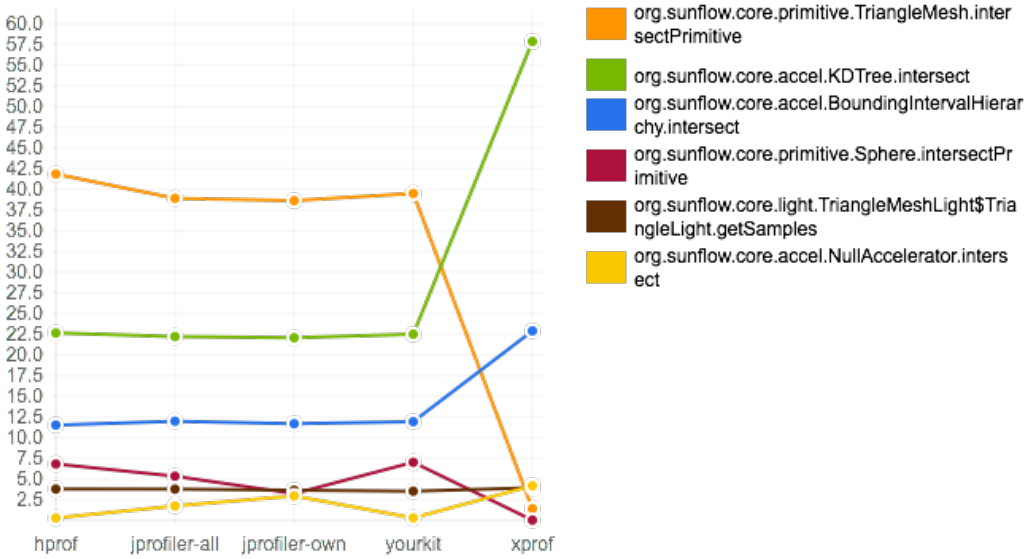



*Figure I.15: Top 4 hottest methods of Sunflow*

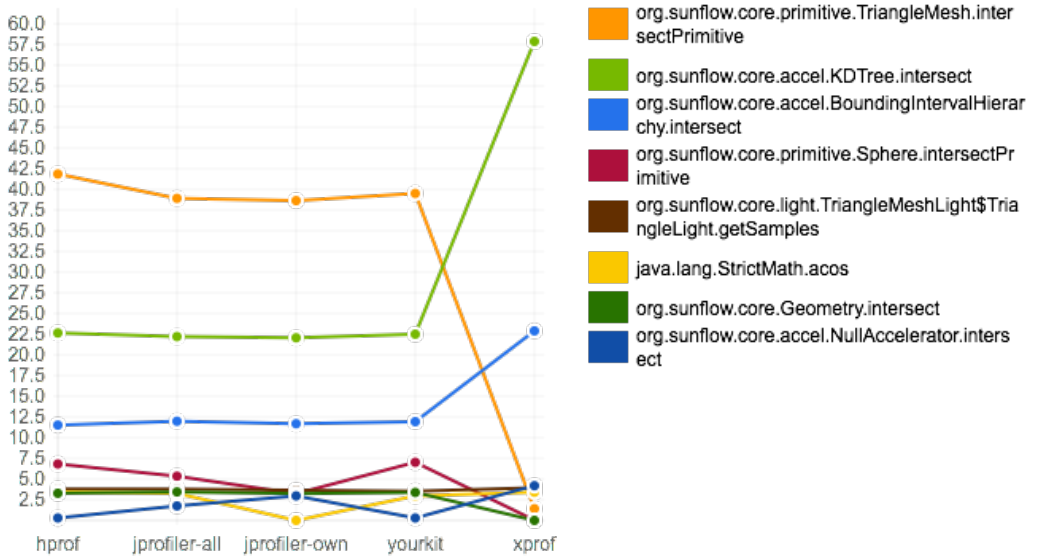



*Figure I.16: Top 6 hottest methods of Sunflow*

**Run with inlining disabled:**

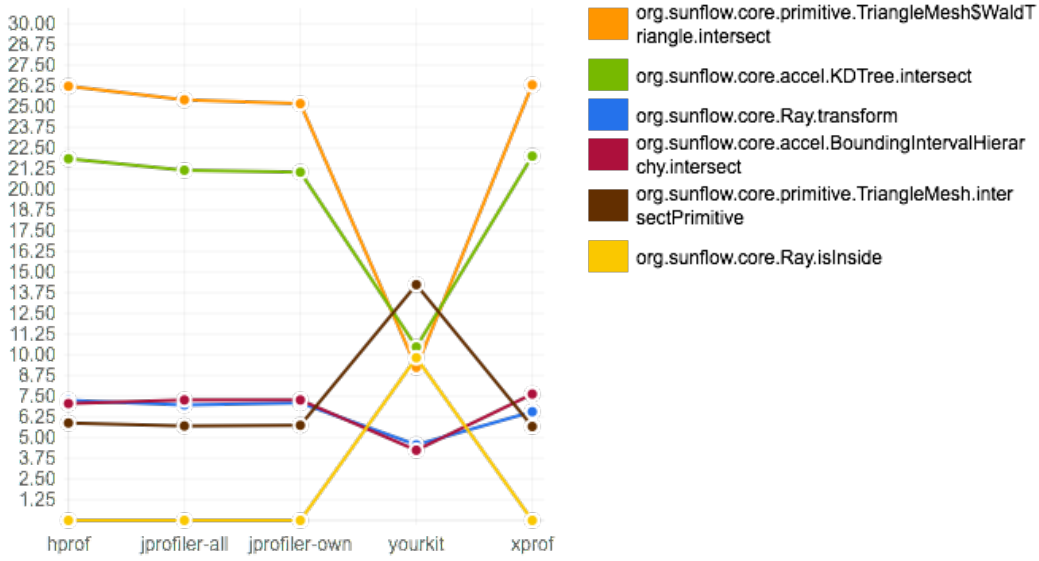



*Figure I.17: Top 4 hottest methods of Sunflow without inlining*

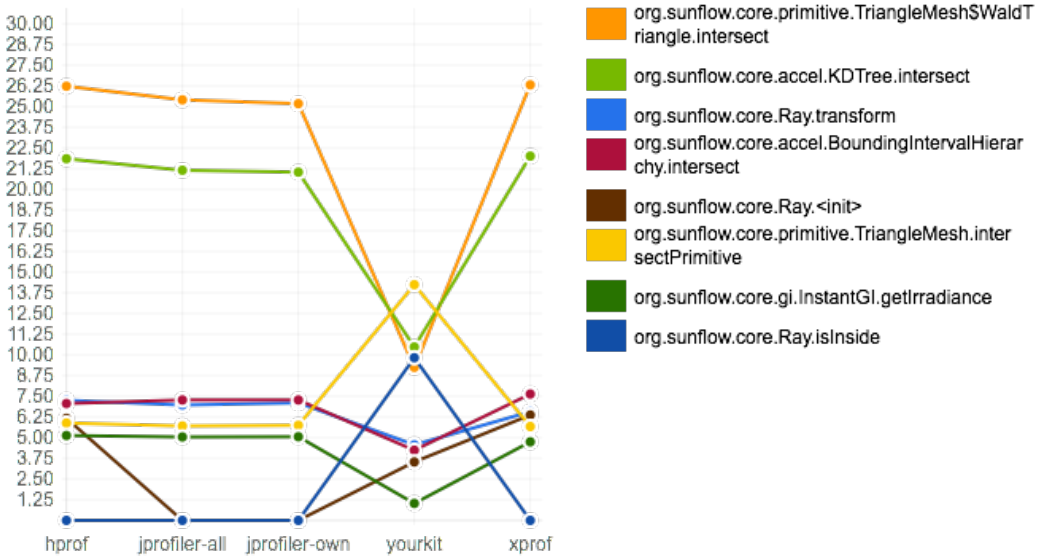



*Figure I.18: Top 6 hottest methods of Sunflow without inlining*

## I.6 Avrora (ran on single-core Ubuntu)

**Normal run top 6 method comparison:**

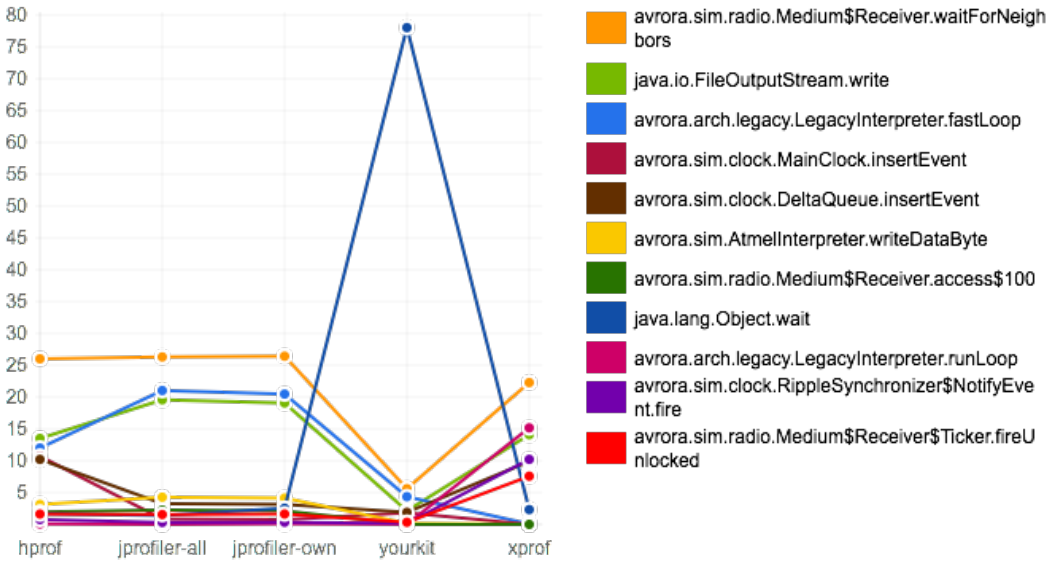



*Figure I.19: Top 6 hottest methods of Avrora*

**Normal run top 6 method comparison (without YourKit):**

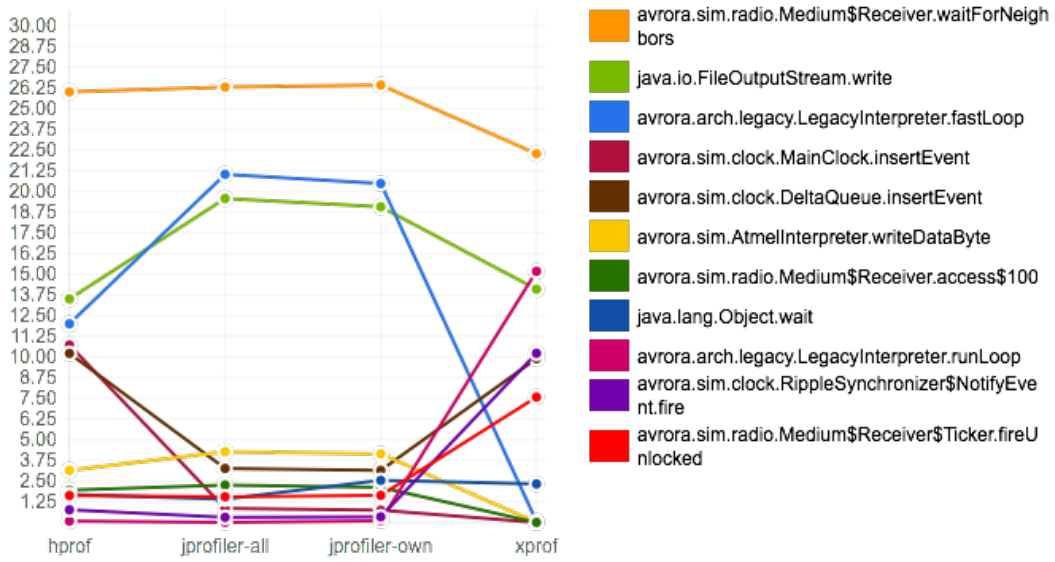



*Figure I.20: Top 6 hottest methods of Avrora (filtered out YourKit)*

**Run with inlining disabled top 4 comparison:**

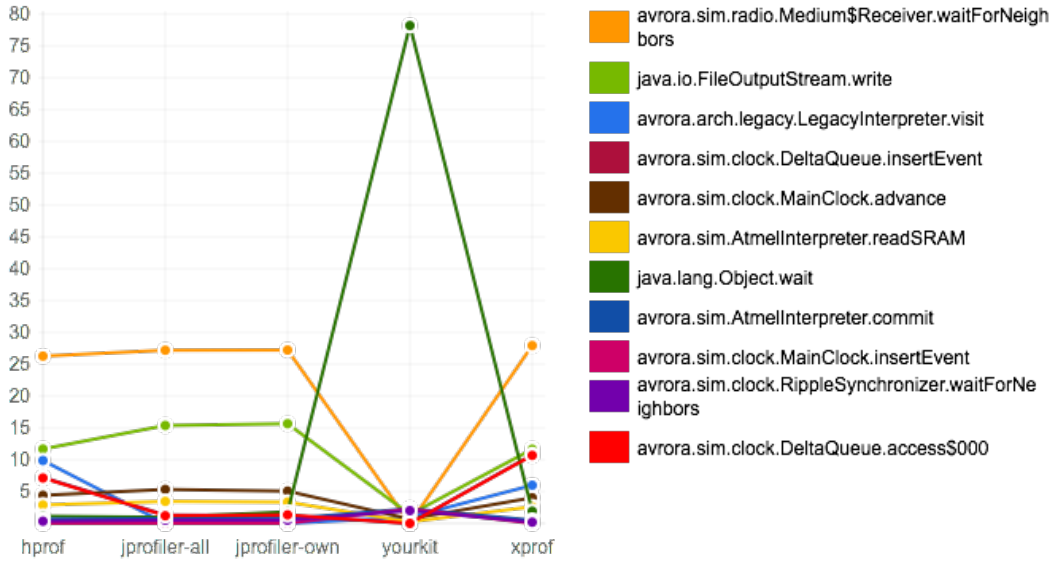



*Figure I.21: Top 4 hottest methods of Avrora without inlining*

**Run with inlining disabled top 7 method comparison (without YourKit):**

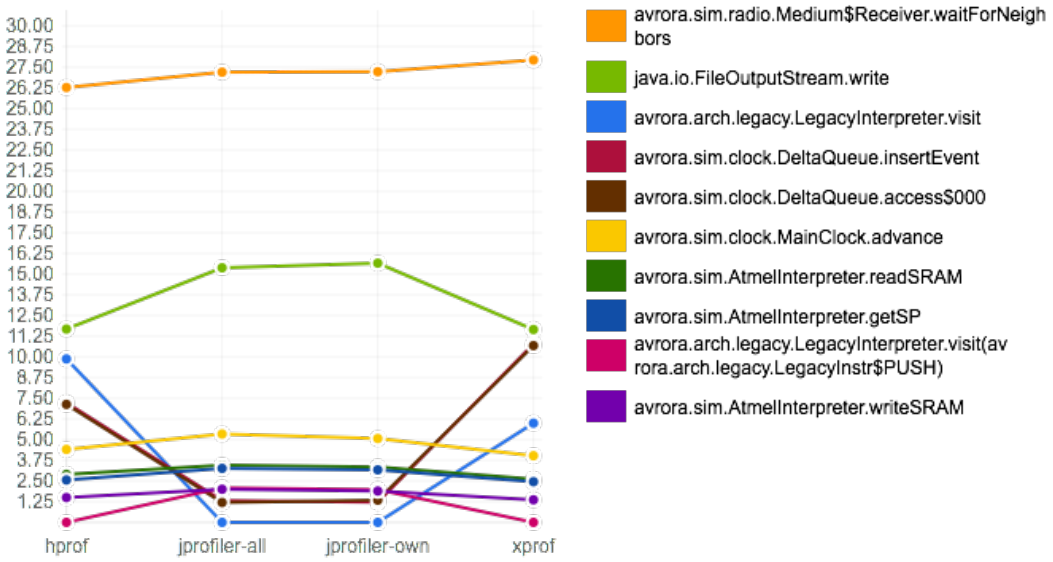



*Figure I.22: Top 7 hottest methods of Avrora without inlining (filtered out YourKit)*

## I.7 Batik (ran on single-core Ubuntu)

**Normal run top 5 method comparison:**

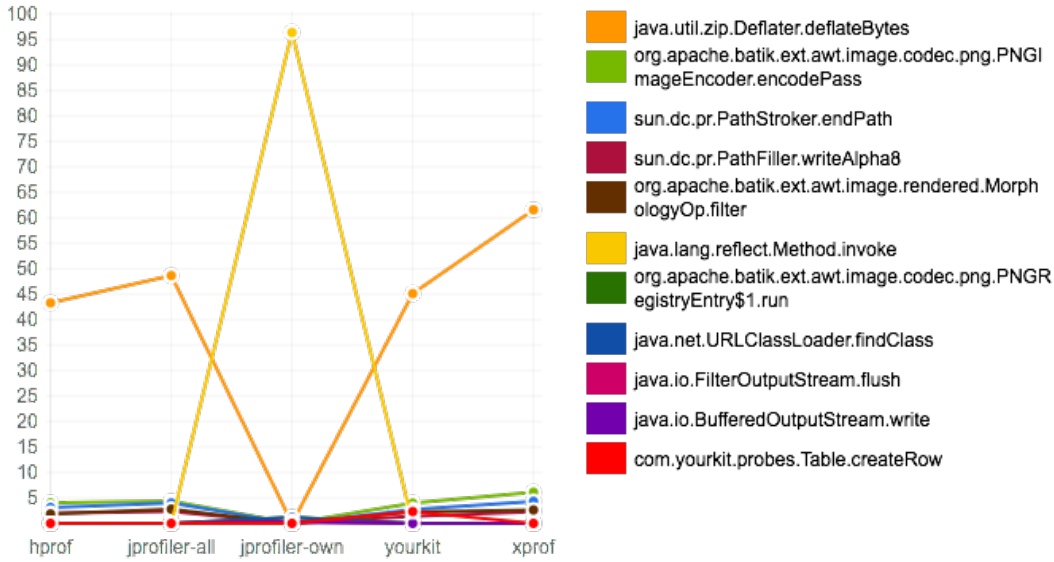



*Figure I.23: Top 5 hottest methods of Batik*

**Normal run top 6 method comparison (without JProfiler with own method filter):**

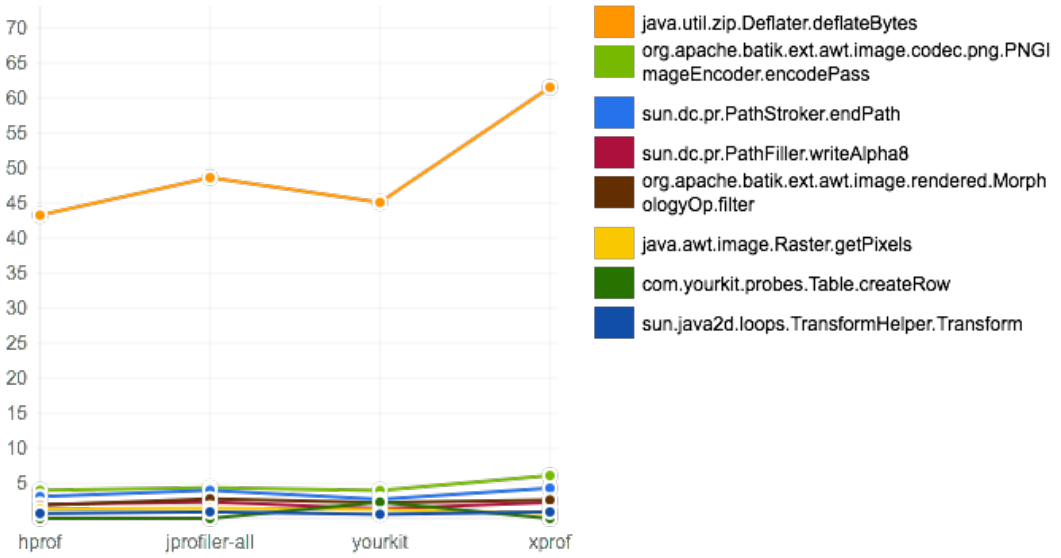



*Figure I.24: Top 6 hottest methods of Batik (filtered out JProfiler own method filter)*

**Run with inlining disabled top 4 method comparison:**

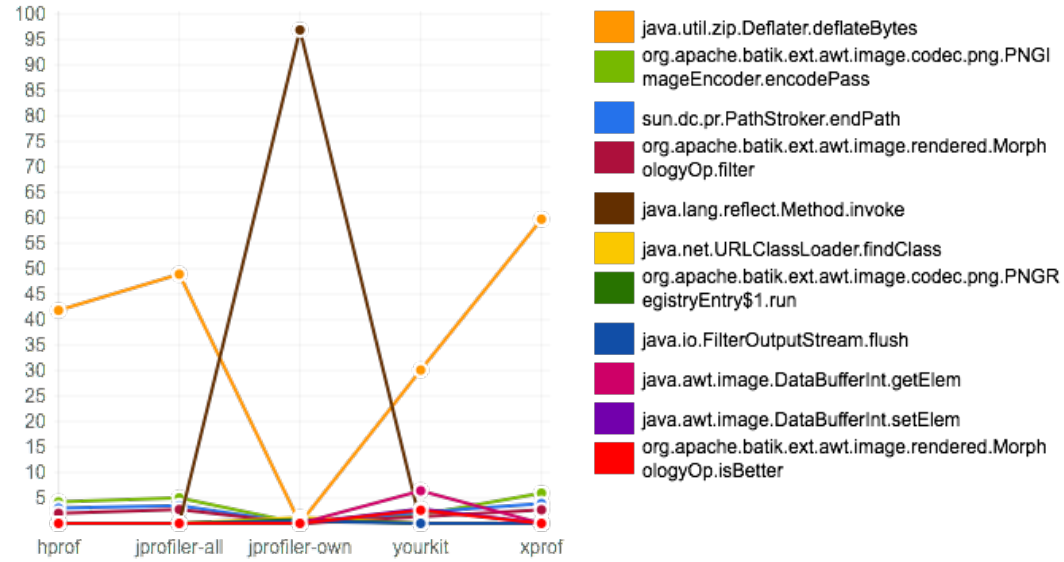



*Figure I.25: Top 4 hottest methods of Batik without inlining*

**Run with inlining disabled top 6 method comparison (without JProfiler with own method filter):**

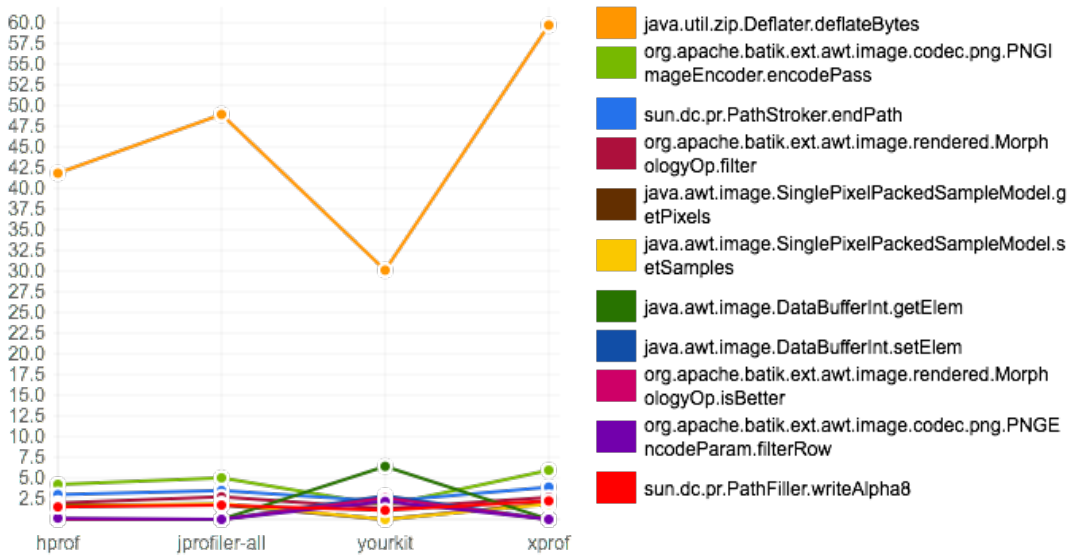



*Figure I.26: Top 6 hottest methods of Batik without inlining (filtered out JProfiler own method filter)*

## I.8 Eclipse (ran on single-core Ubuntu)

**Normal run top 4 method comparison:**

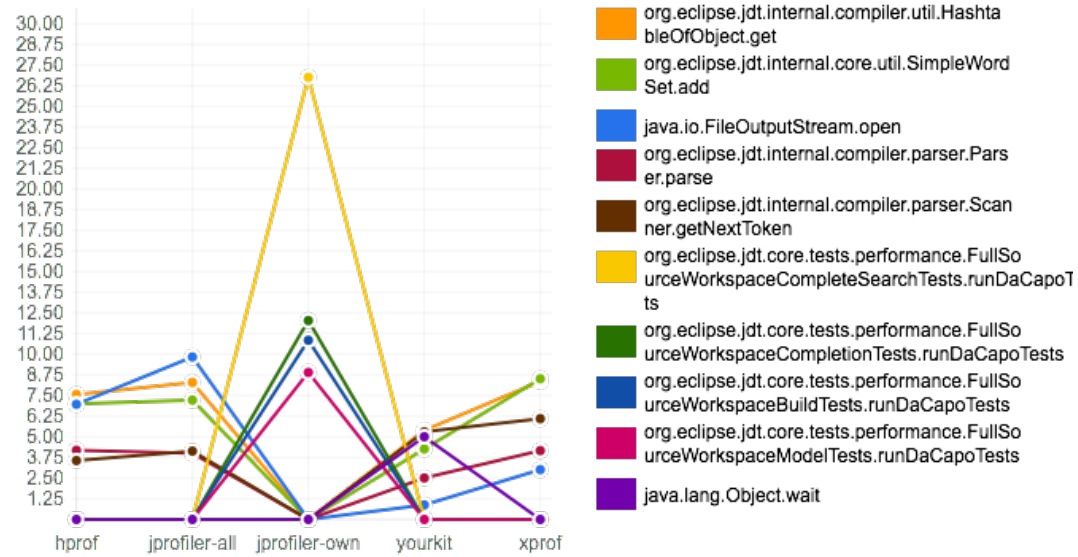



*Figure I.27: Top 4 hottest methods of Eclipse*

**Normal run top 7 method comparison (without JProfiler with own method filter):**

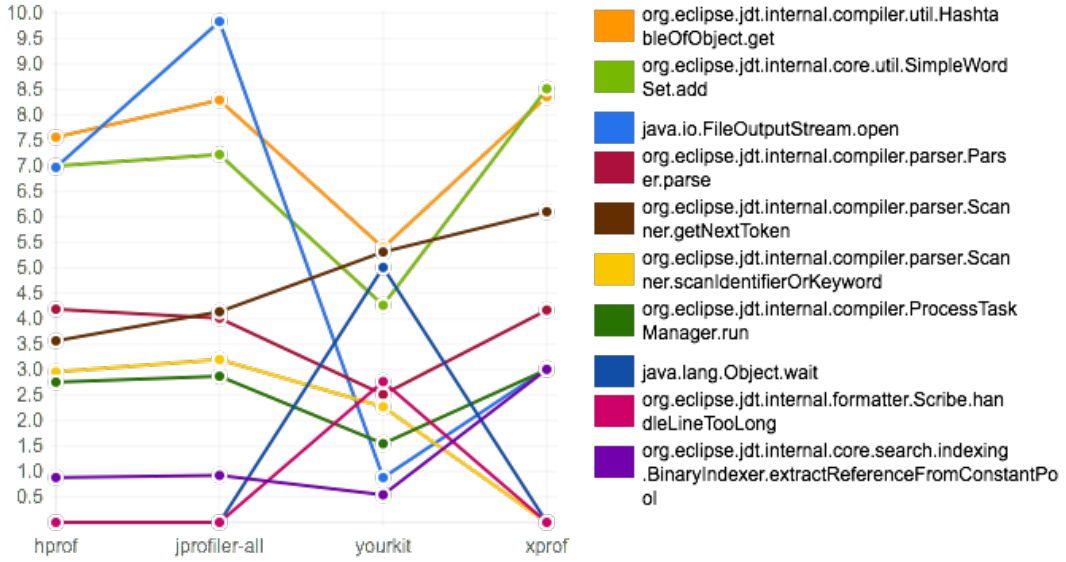



*Figure I.28: Top 7 hottest methods of Eclipse (filtered out JProfiler own method filter)*

**Run with inlining disabled top 3 method comparison:**

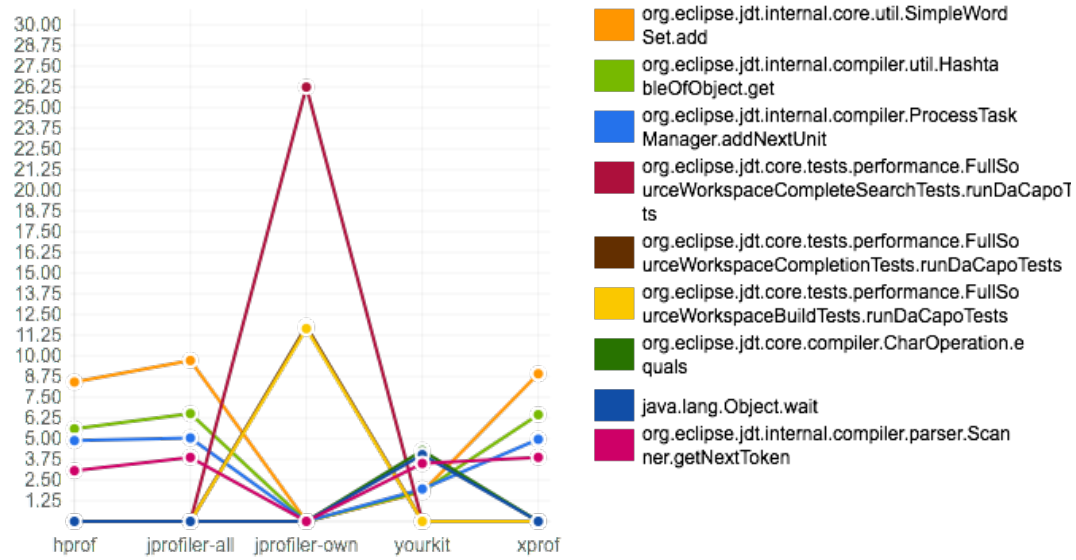



*Figure I.29: Top 3 hottest methods of Eclipse without inlining*

**Run with inlining disabled top 7 method comparison (without JProfiler with own method filter):**

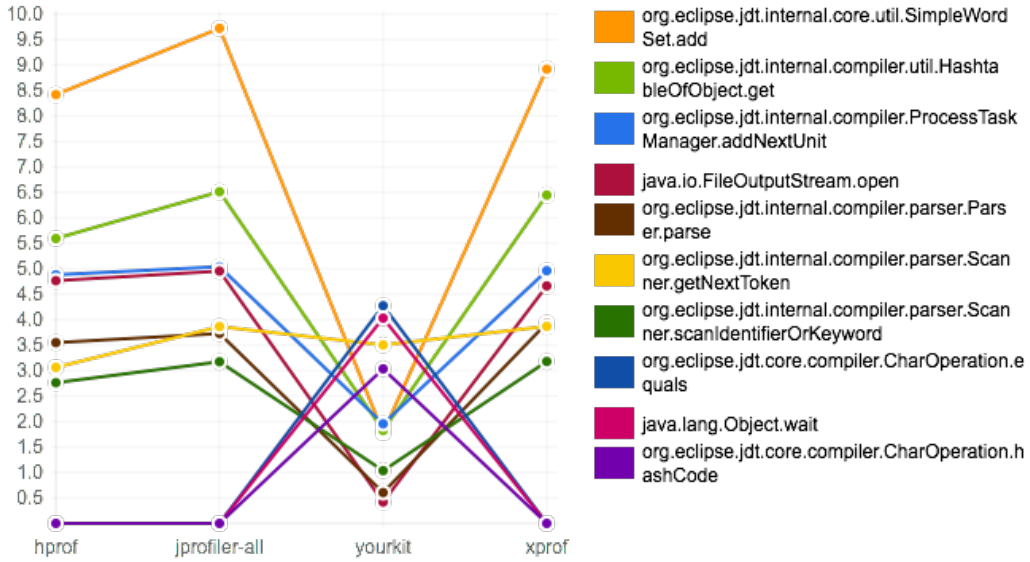



*Figure I.30: Top 7 hottest methods of Eclipse without inlining (filtered out JProfiler own method filter)*

## I.9 H2 (ran on single-core Ubuntu)

**Normal run top 3 method comparison:**

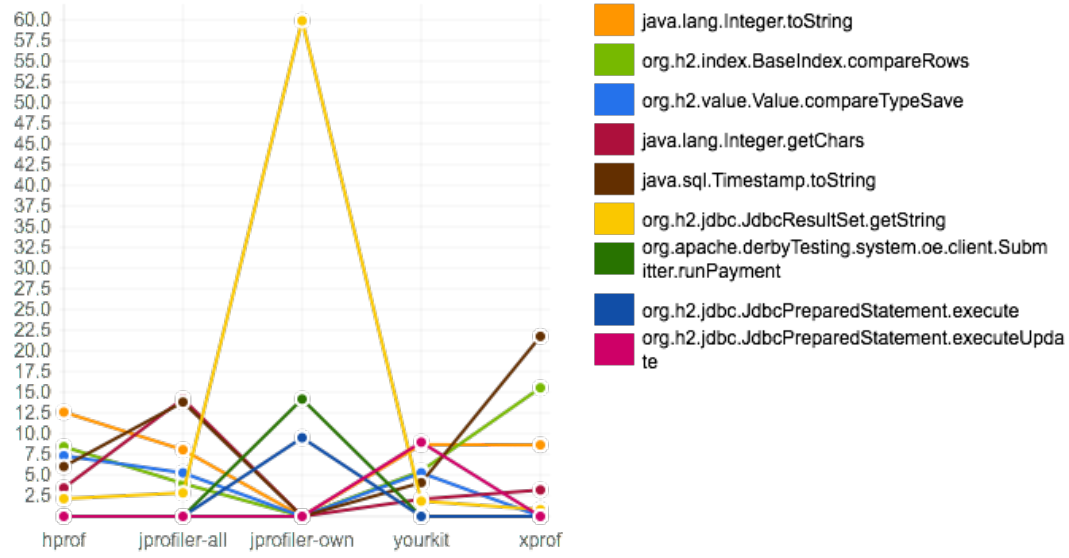



*Figure I.31: Top 3 hottest methods of H2*

**Normal run top 5 method comparison (without JProfiler with own method filter):**

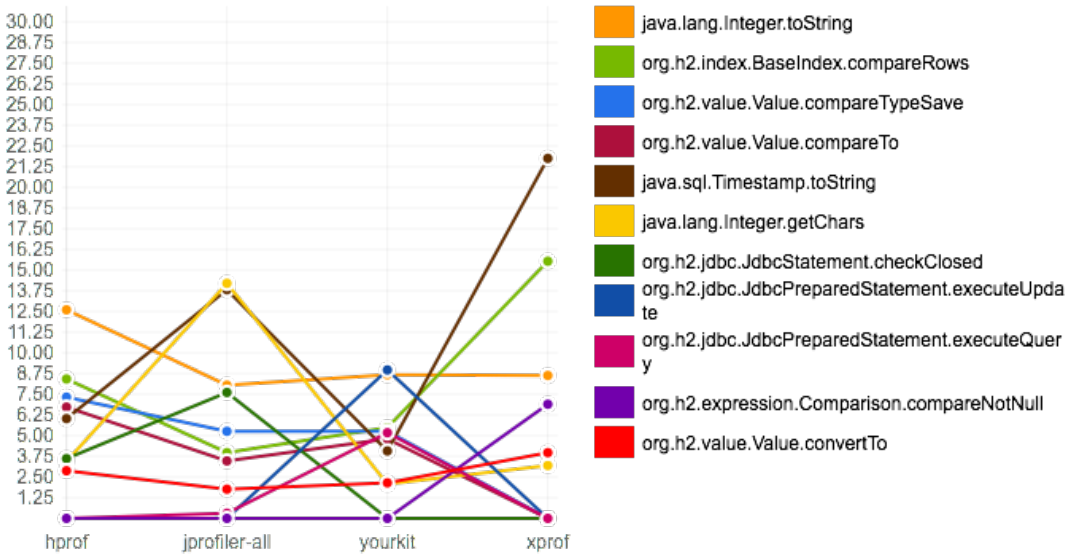



*Figure I.32: Top 5 hottest methods of H2 (filtered out JProfiler own method filter)*

**Run with inlining disabled top 3 method comparison:**

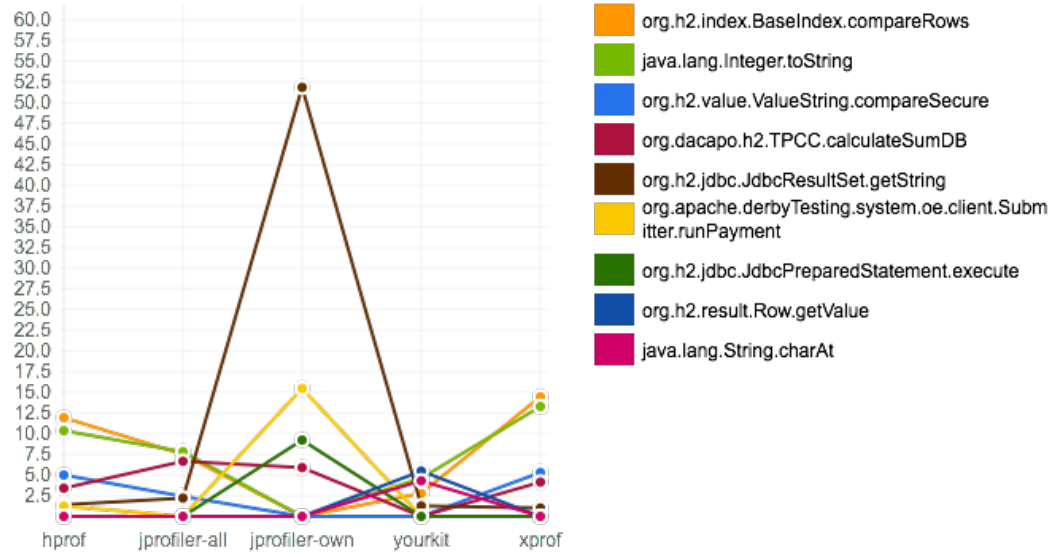



*Figure I.33: Top 3 hottest methods of H2 without inlining*

**Run with inlining disabled top 5 method comparison (without JProfiler with own method filter):**

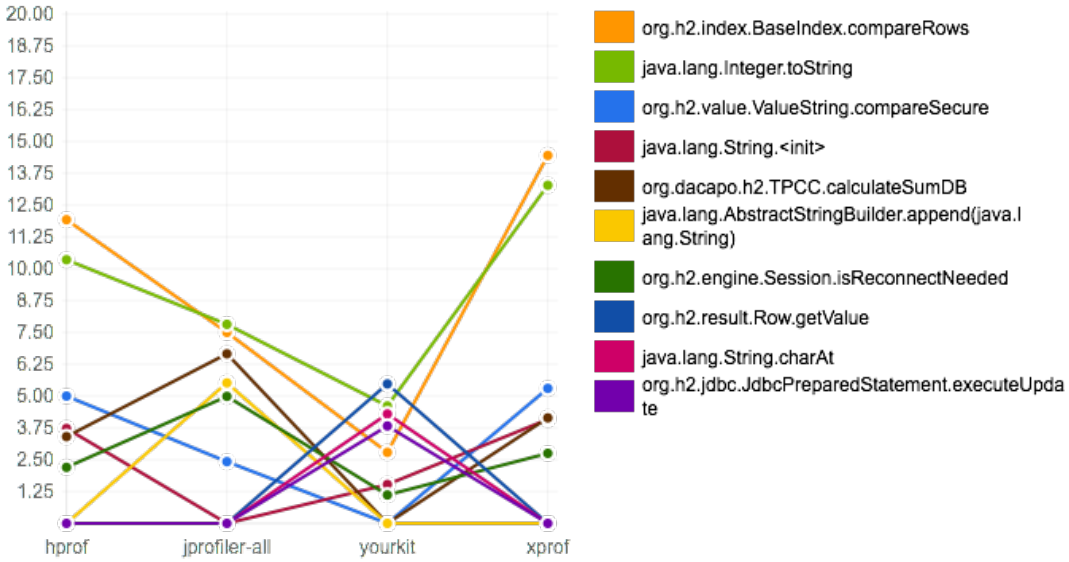



*Figure I.34: Top 5 hottest methods of H2 without inlining (filtered out JProfiler own method filter)*

## I.10 Jython (ran on single-core Ubuntu)

**Normal run top 4 method comparison:**

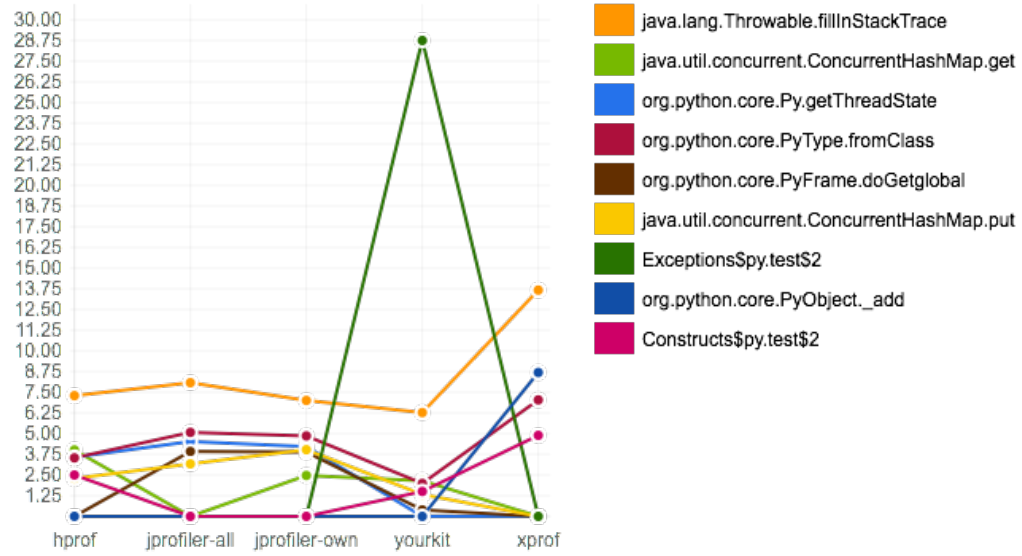



*Figure I.35: Top 4 hottest methods of Jython*

**Normal run top 5 method comparison (without YourKit):**

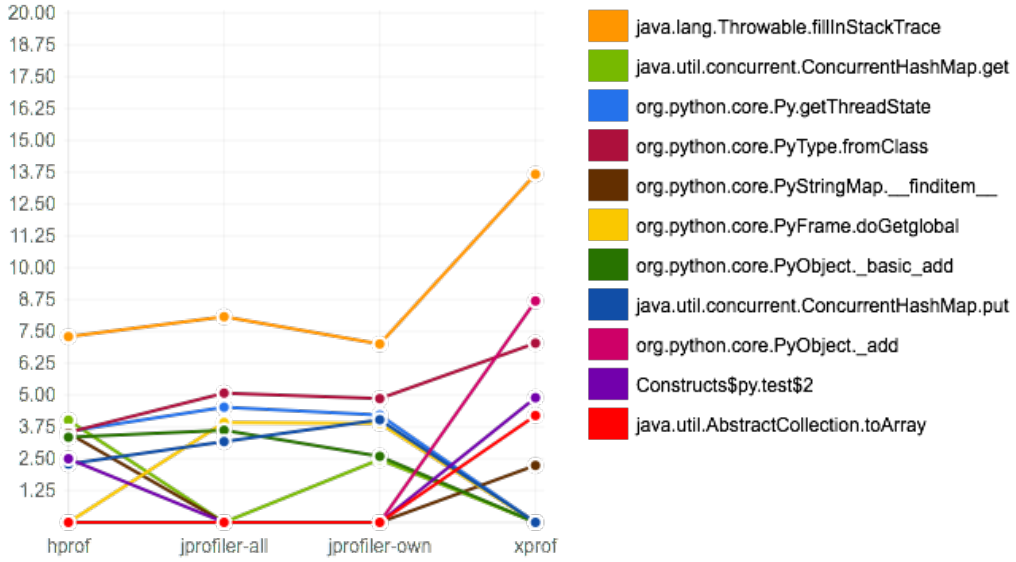



*Figure I.36: Top 5 hottest methods of Jython (filtered out YourKit)*

**Run with inlining disabled top 4 method comparison:**

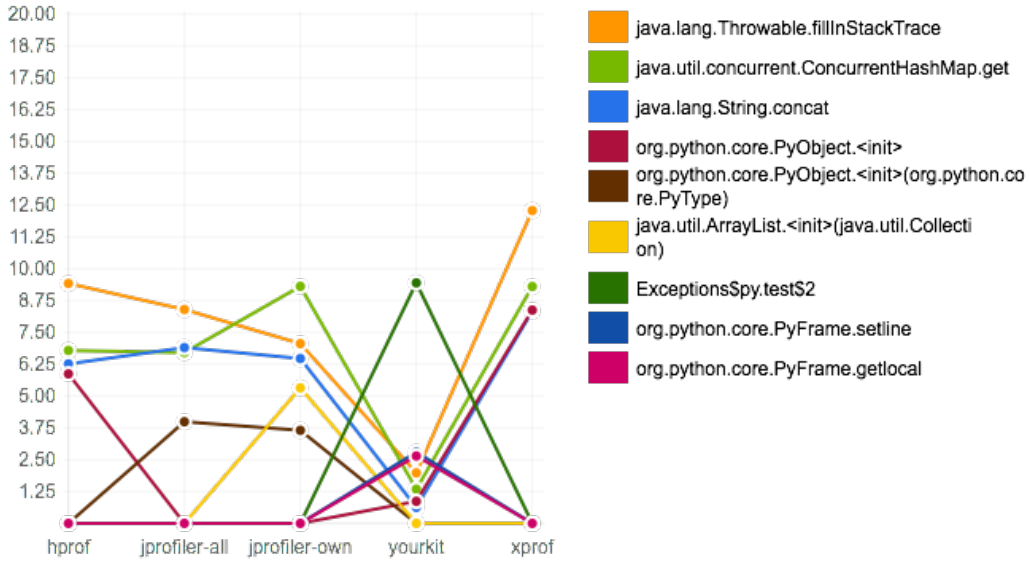



*Figure I.37: Top 4 hottest methods of Jython without inlining*

**Run with inlining disabled top 5 method comparison (without YourKit):**

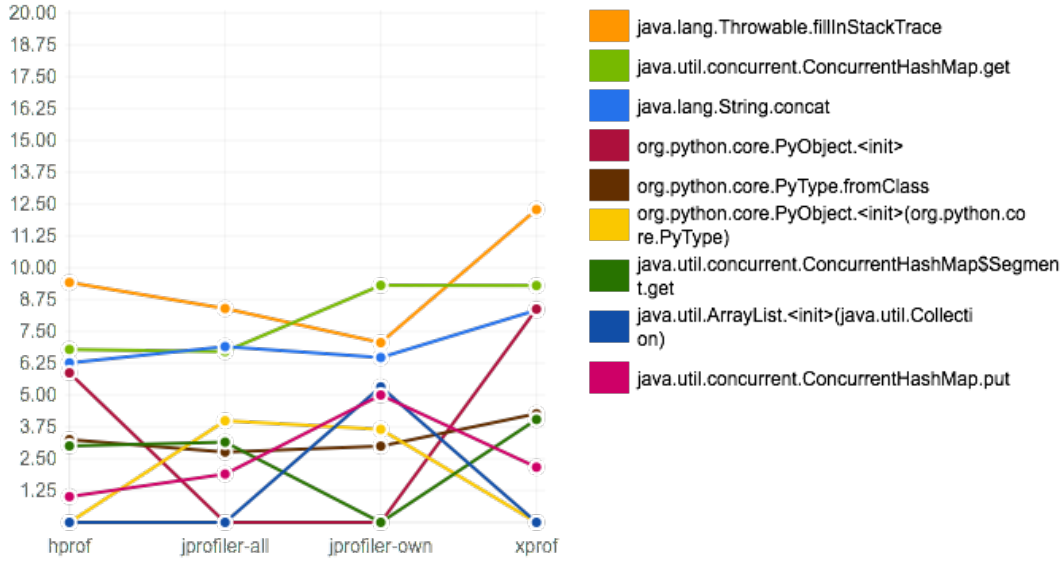



*Figure I.38: Top 5 hottest methods of Jython without inlining (filtered out YourKit)*

## I.11 Lucene Index (ran on single-core Ubuntu)

**Normal run top 4 method comparison:**

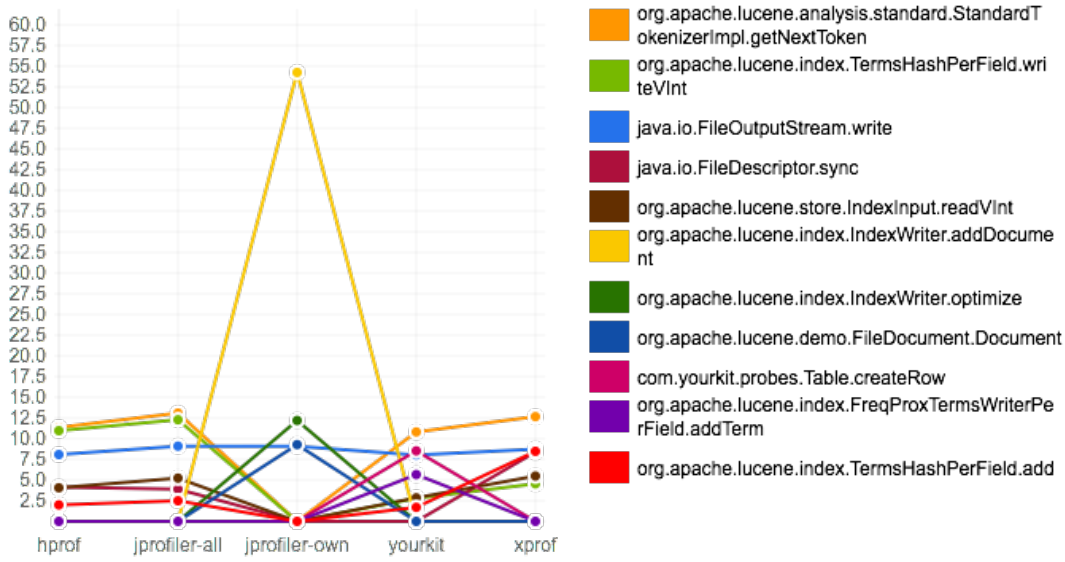



*Figure I.39: Top 4 hottest methods of Lucene Index*

**Normal run top 6 method comparison (without JProfiler with own method filter):**

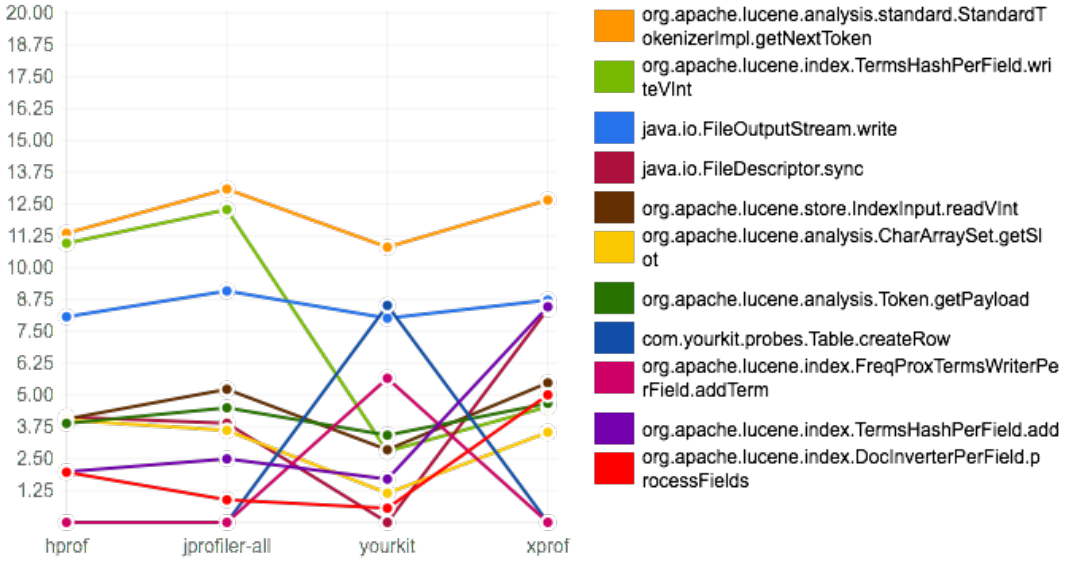



*Figure I.40: Top 6 hottest methods of Lucene Index (filtered out JProfiler own method filter)*

**Run with inlining disabled top 4 method comparison:**

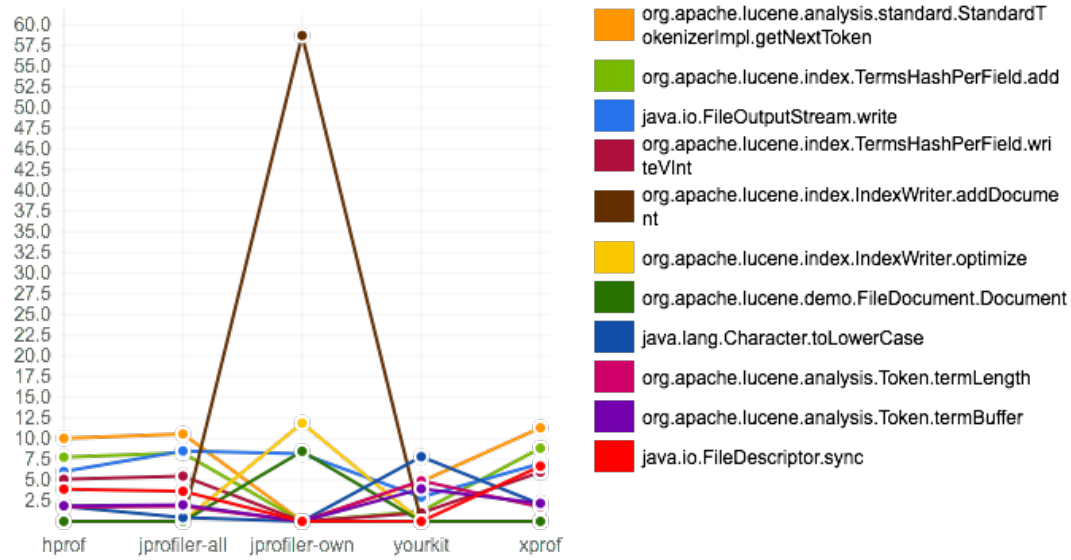



*Figure I.41: Top 4 hottest methods of Lucene Index without inlining*

**Run with inlining disabled top 5 method comparison (without JProfiler with own method filter):**

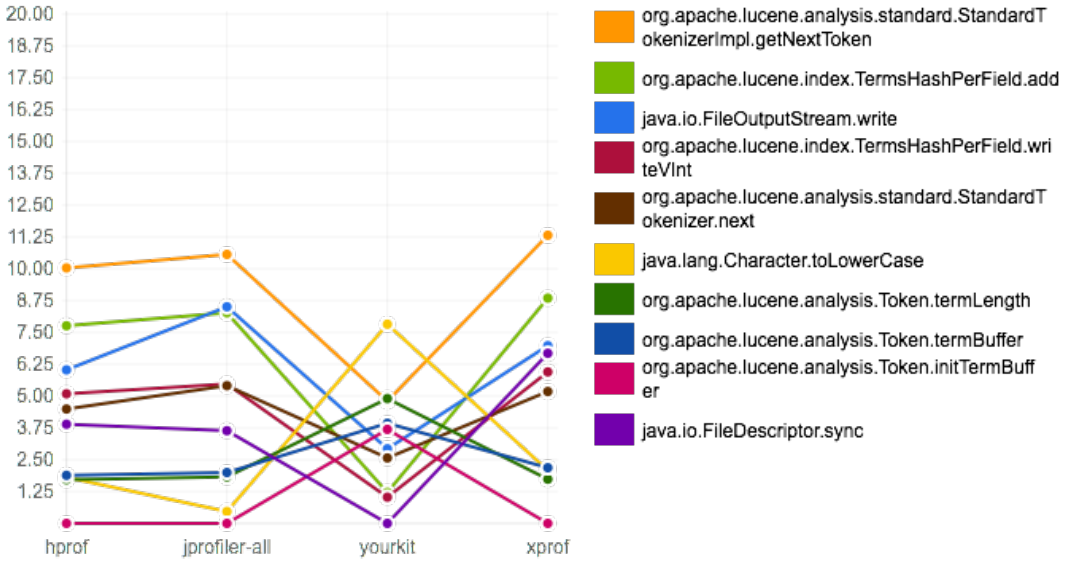



*Figure I.42: Top 5 hottest methods of Lucene Index without inlining (filtered out JProfiler own method filter)*

## I.12 Lucene Search (ran on single-core Ubuntu)

**Normal run top 3 method comparison:**

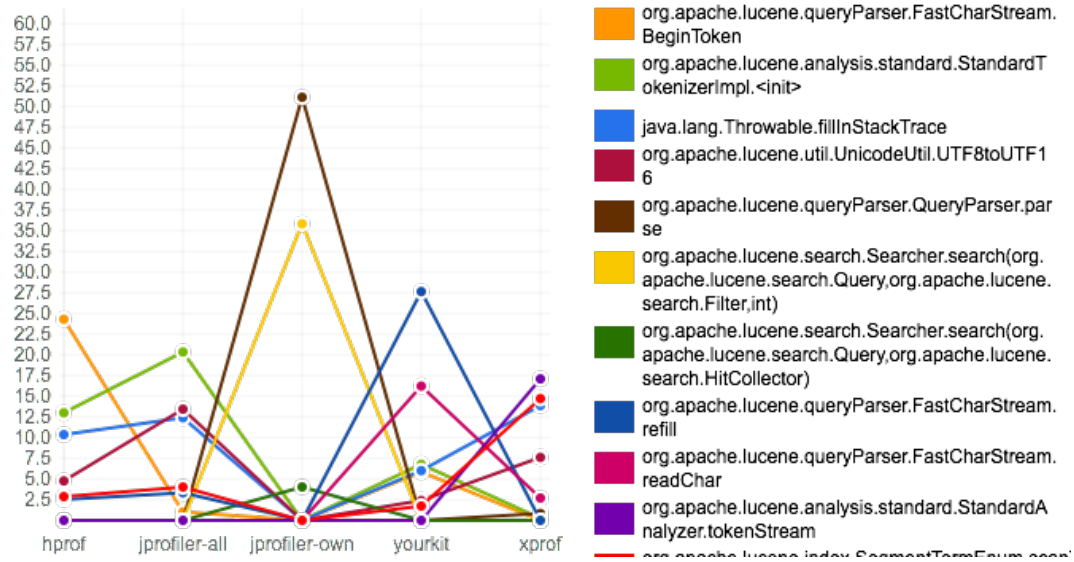



*Figure I.43: Top 3 hottest methods of Lucene Search*

**Normal run top 5 method comparison (without JProfiler with own method filter):**

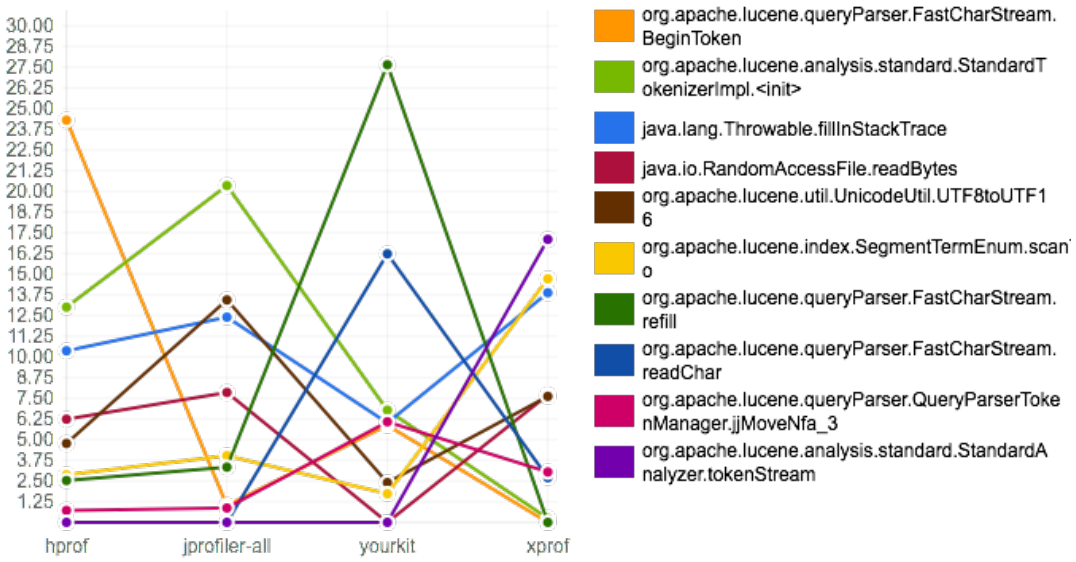



*Figure I.44: Top 5 hottest methods of Lucene Search (filtered out JProfiler own method filter)*

**Run with inlining disabled top 3 method comparison:**

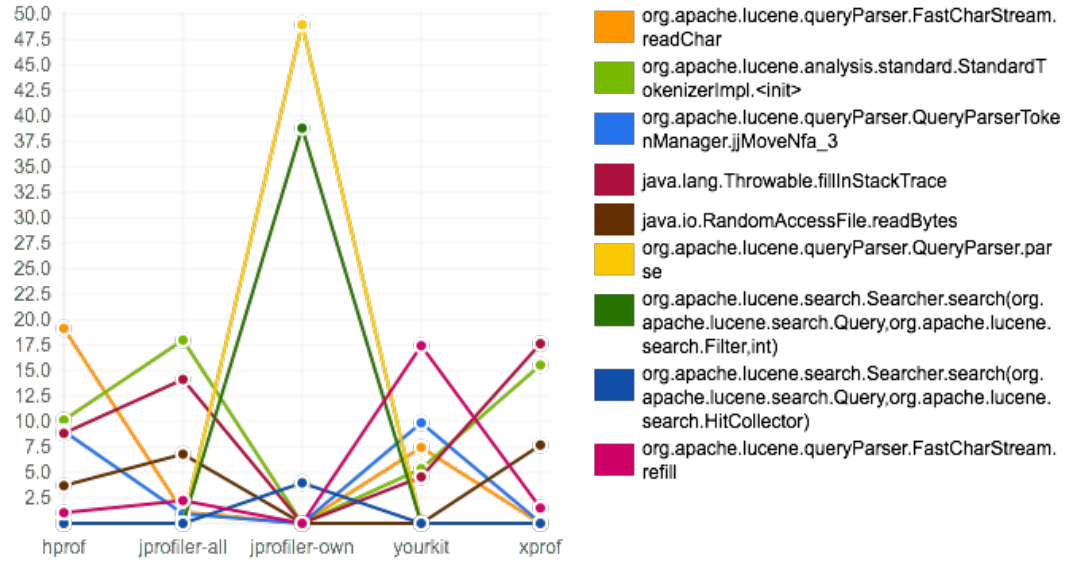



*Figure I.45: Top 3 hottest methods of Lucene Search without inlining*

**Run with inlining disabled top 5 method comparison (without JProfiler with own method filter):**

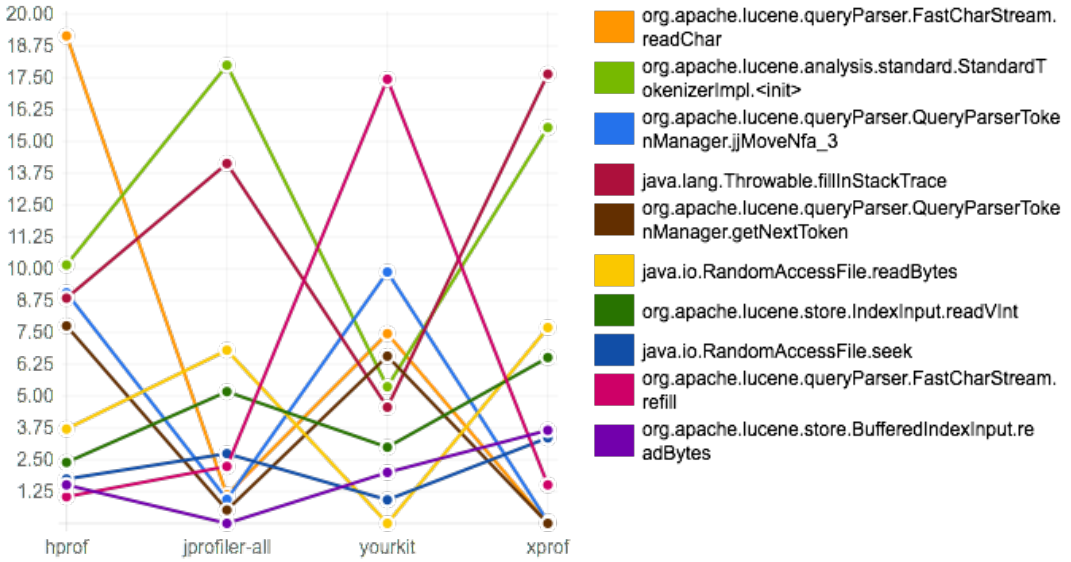



*Figure I.46: Top 5 hottest methods of Lucene Search without inlining (filtered out JProfiler own method filter)*

## I.13 Tomcat (ran on single-core Ubuntu)

**Normal run top 2 method comparison:**

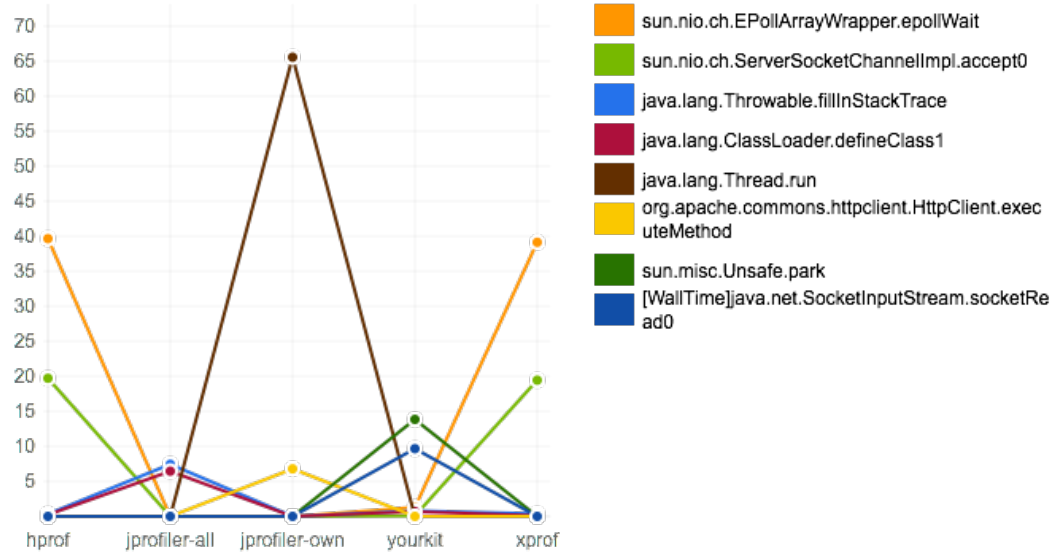



*Figure I.47: Top 2 hottest methods of Tomcat*

**Normal run top 3 method comparison (without JProfiler with own method filter):**

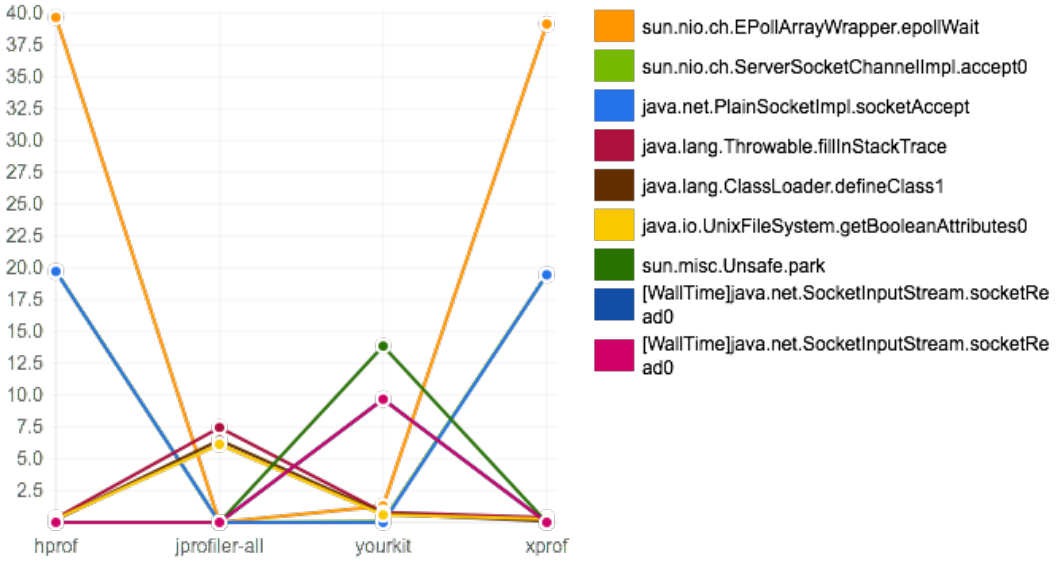



*Figure I.48: Top 3 hottest methods of Tomcat (filtered out JProfiler own method filter)*

**Run with inlining disabled top 2 method comparison:**

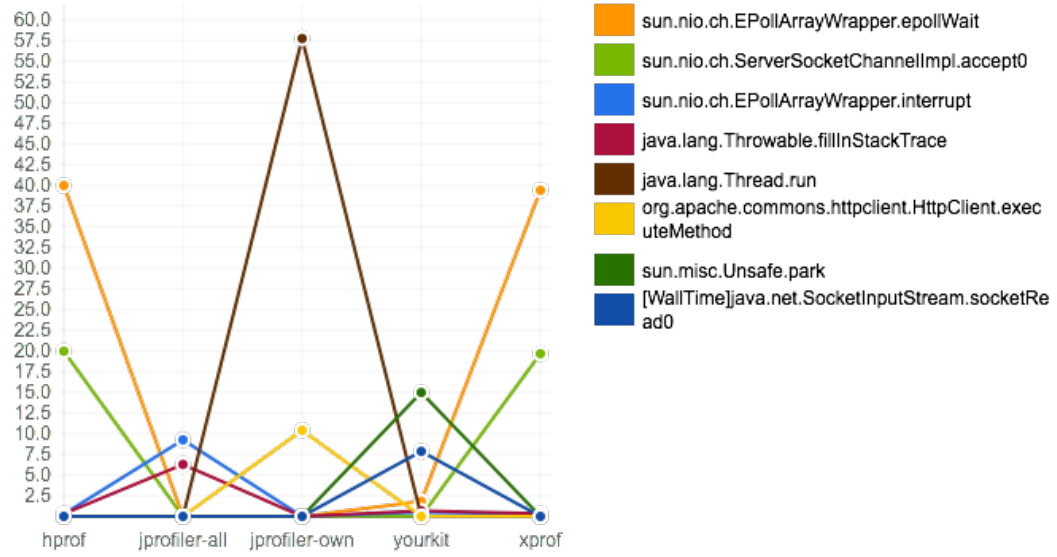



*Figure I.49: Top 2 hottest methods of Tomcat without inlining*

**Run with inlining disabled top 3 method comparison (without JProfiler with own method filter):**

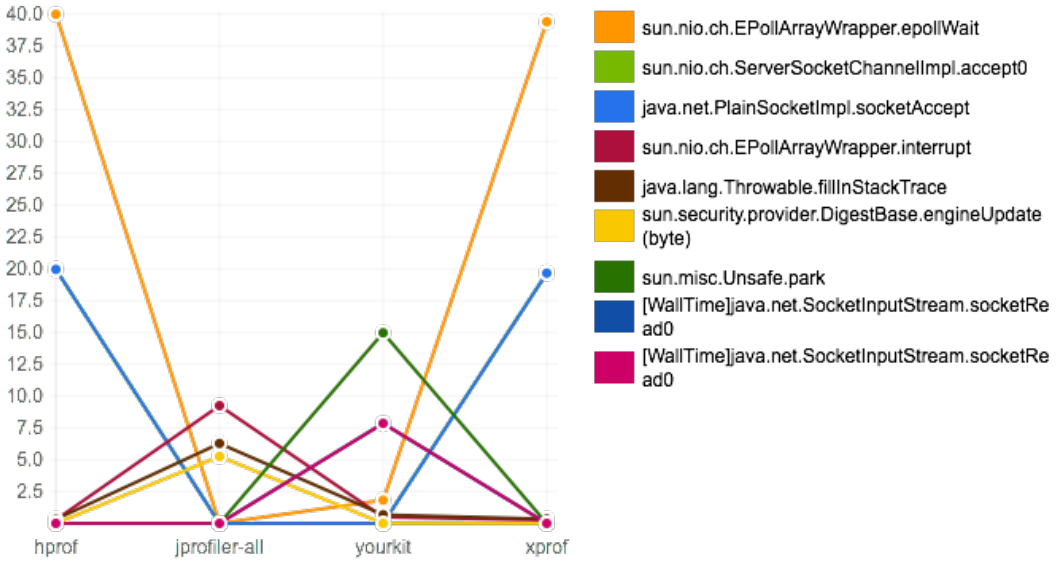



*Figure I.50: Top 3 hottest methods of Tomcat without inlining (filtered out JProfiler own method filter)*

## I.14 Xalan (ran on single-core Ubuntu)

**Normal run top 3 method comparison:**

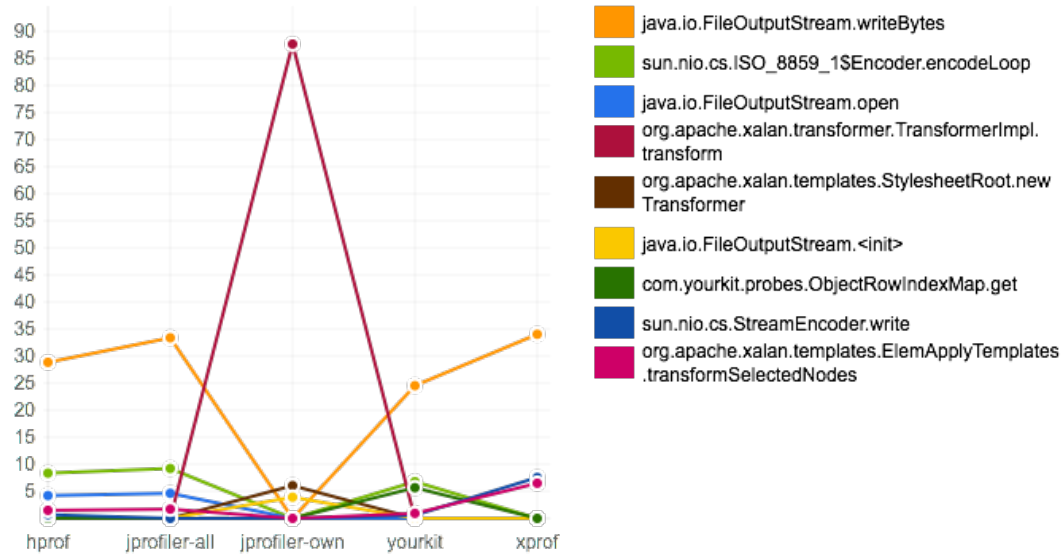



*Figure I.51: Top 3 hottest methods of Xalan*

**Normal run top 4 method comparison (without JProfiler with own method filter):**

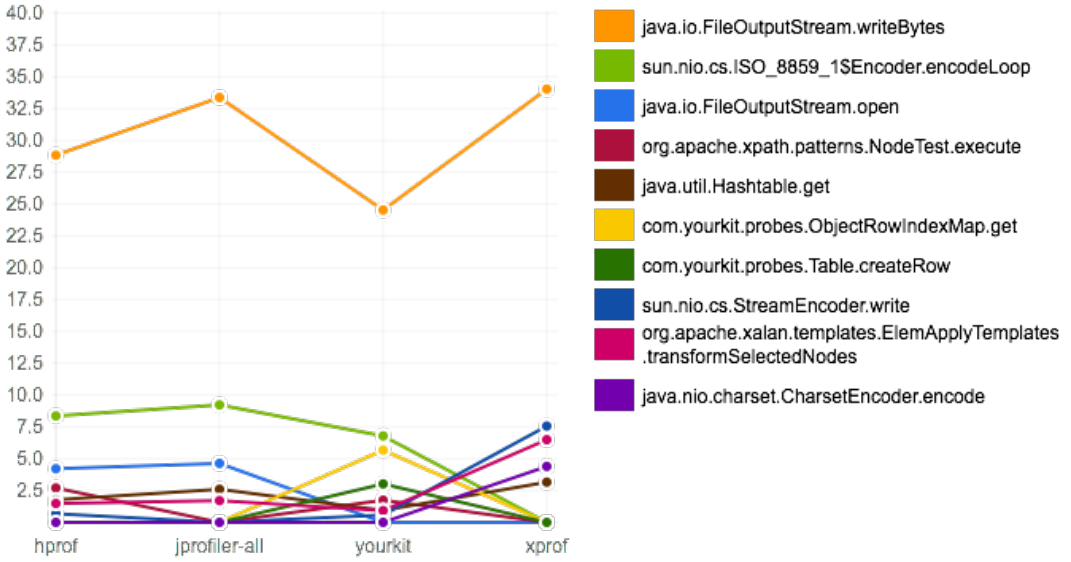



*Figure I.52: Top 4 hottest methods of Xalan (filtered out JProfiler own method filter)*

**Run with inlining disabled top 3 method comparison:**

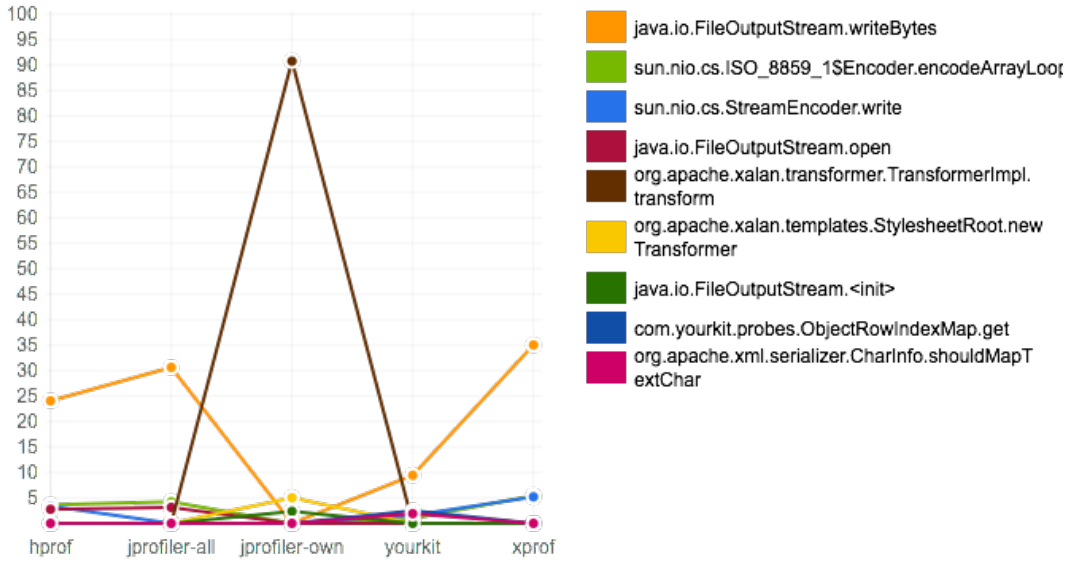



*Figure I.53: Top 3 hottest methods of Xalan without inlining*

**Run with inlining disabled top 5 method comparison (without JProfiler with own method filter):**

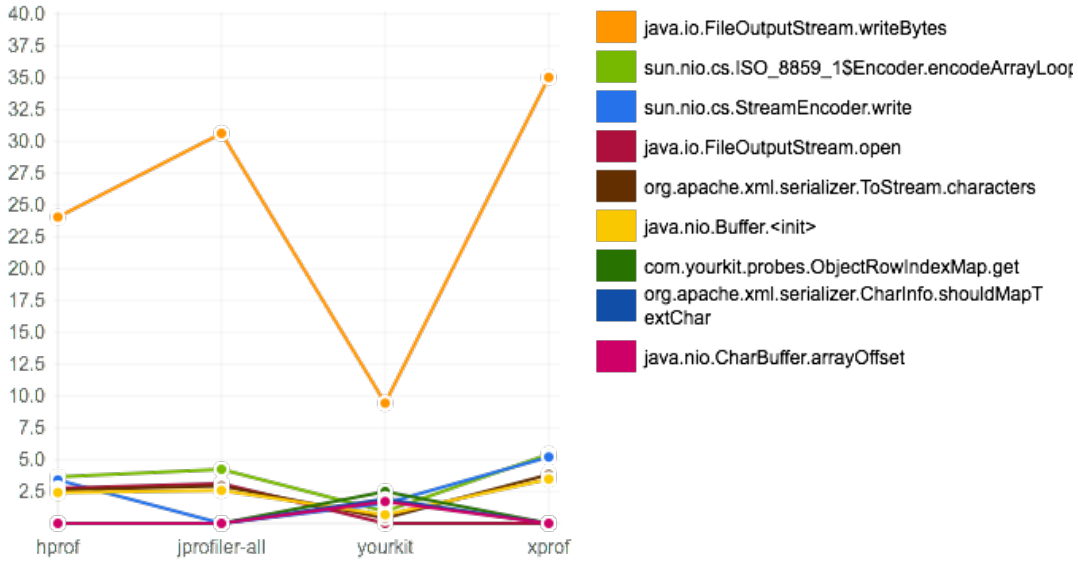



*Figure I.54: Top 5 hottest methods of Xalan without inlining (filtered out JProfiler own method filter)*

## I.15 Raw data number of methods found in top hottest methods across the 4 profilers

| normal run | Top 1 | Top 2 | Top 3 | Top 4 | Top 5 | Top 6 | Top 7 | Top 8 | Top 9 | Top 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| PMD (linux) | 2 | 4 | 6 | 7 | 10 | 13 | 14 | 16 | 18 | 22 |
| Sunflow | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 11 | 11 | 12 |
| Avrora | 2 | 5 | 5 | 8 | 9 | 11 | 15 | 16 | 19 | 20 |
| Batik | 1 | 2 | 3 | 6 | 6 | 8 | 11 | 14 | 15 | 16 |
| Eclipse | 3 | 4 | 5 | 6 | 7 | 9 | 10 | 14 | 16 | 17 |
| H2 | 4 | 5 | 6 | 9 | 11 | 14 | 14 | 16 | 19 | 21 |
| Jython | 2 | 5 | 6 | 8 | 11 | 14 | 16 | 20 | 22 | 22 |
| Lucene Index | 1 | 4 | 5 | 8 | 9 | 11 | 11 | 13 | 16 | 17 |
| Lucene Search | 4 | 7 | 8 | 10 | 10 | 13 | 14 | 16 | 16 | 17 |
| Tomcat | 3 | 6 | 9 | 12 | 15 | 17 | 20 | 22 | 24 | 27 |
| Xalan | 1 | 3 | 6 | 10 | 12 | 16 | 16 | 17 | 18 | 21 |
| Average | 2,3 | 4,4 | 5,7 | 8,2 | 9,7 | 12,2 | 13,6 | 15,9 | 17,6 | 19,3 |

| Inlining disabled | Top 1 | Top 2 | Top 3 | Top 4 | Top 5 | Top 6 | Top 7 | Top 8 | Top 9 | Top 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| PMD (linux) | 2 | 5 | 7 | 8 | 11 | 13 | 15 | 17 | 20 | 23 |
| Sunflow | 2 | 3 | 5 | 6 | 8 | 10 | 12 | 13 | 13 | 14 |
| Avrora | 2 | 3 | 7 | 9 | 10 | 12 | 14 | 16 | 18 | 20 |
| Batik | 1 | 2 | 3 | 4 | 8 | 9 | 9 | 11 | 14 | 16 |
| Eclipse | 2 | 3 | 5 | 6 | 8 | 8 | 9 | 12 | 14 | 15 |
| H2 | 3 | 3 | 5 | 8 | 10 | 12 | 16 | 17 | 20 | 21 |
| Jython | 2 | 4 | 5 | 7 | 9 | 11 | 13 | 16 | 17 | 19 |
| Lucene Index | 1 | 3 | 4 | 7 | 9 | 10 | 11 | 12 | 13 | 15 |
| Lucene Search | 4 | 4 | 7 | 8 | 11 | 13 | 14 | 15 | 18 | 19 |
| Tomcat | 3 | 6 | 9 | 12 | 15 | 18 | 20 | 22 | 25 | 28 |
| Xalan | 1 | 3 | 6 | 8 | 10 | 13 | 15 | 17 | 18 | 20 |
| Average | 2,1 | 3,5 | 5,7 | 7,5 | 9,9 | 11,7 | 13,5 | 15,3 | 17,3 | 19,1 |

# J DaCapo PMD bytecode injecting results

All injection tests of this chapter are run on Machine A.

## J.1 Xprof (inlining disabled)

### J.1.1 AbstractJavaRule.visit

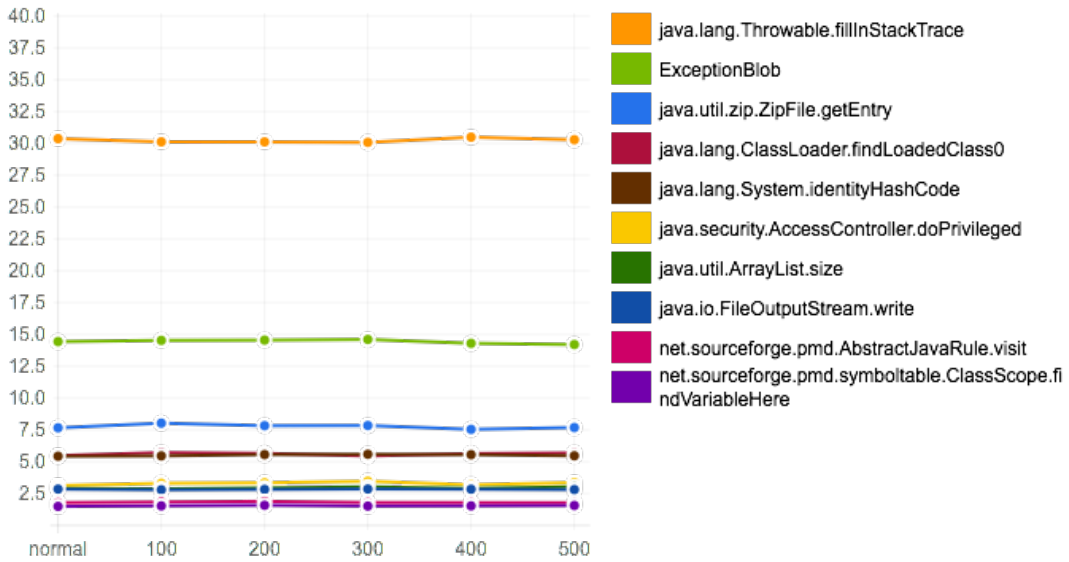



*Figure J.1: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

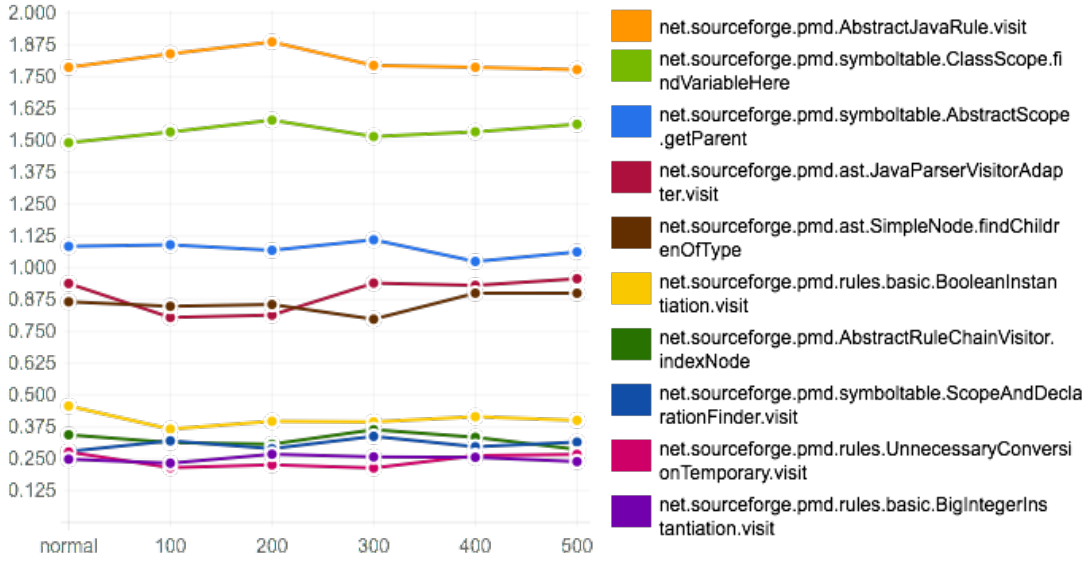



*Figure J.2: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining)*

## J.2 Hprof

### J.2.1 ast.JavaParser.jj_3R_91

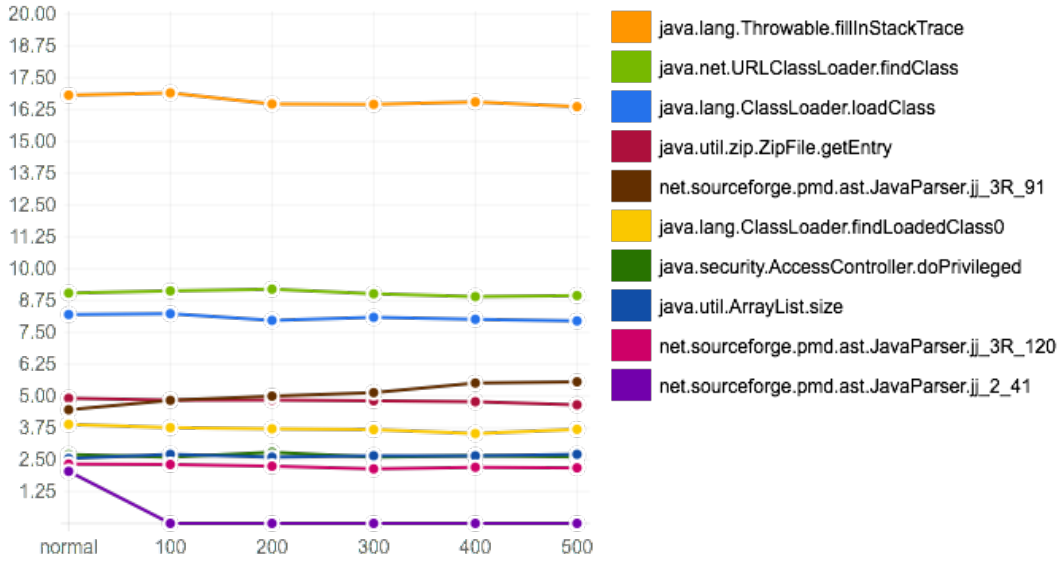



*Figure J.3: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

### J.2.2 ast.JavaParser.jj_3R_120

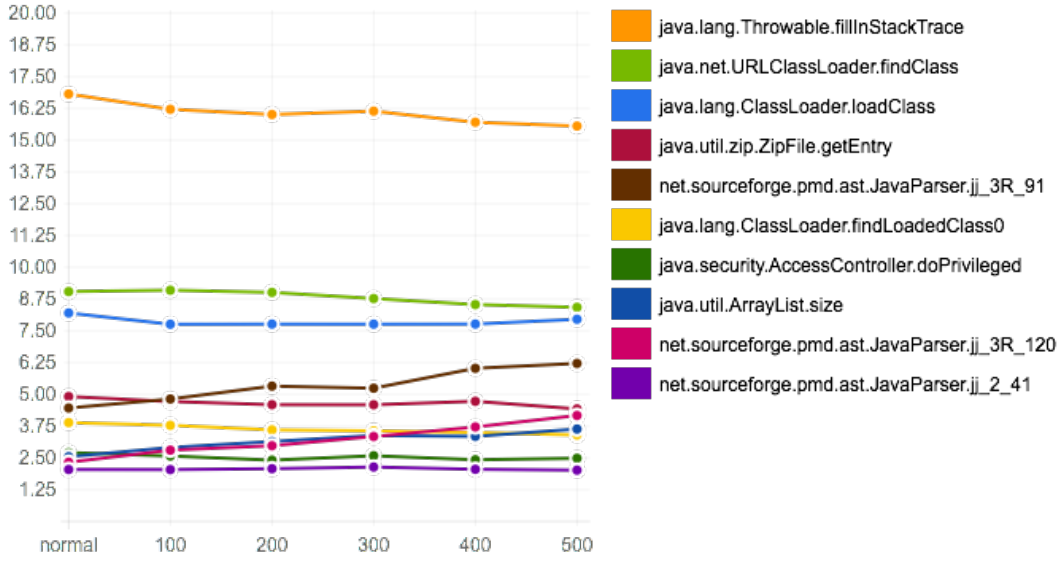



*Figure J.4: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

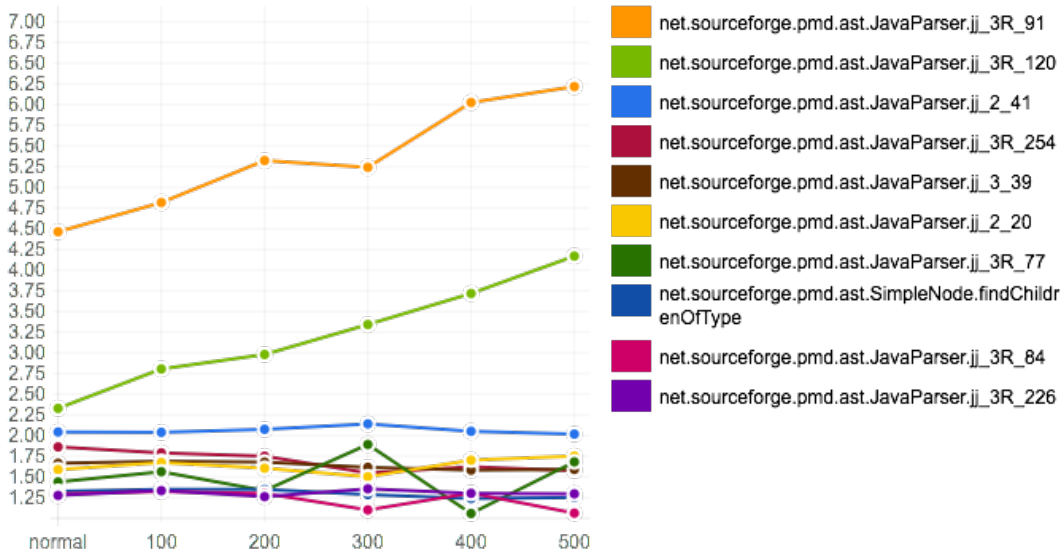



*Figure J.5: Effect on top 10 methods after being injected with the Fibonacci algorithm*

## J.3 Hprof (inlining disabled)

### J.3.1 ast.JavaParser.jj_3R_91

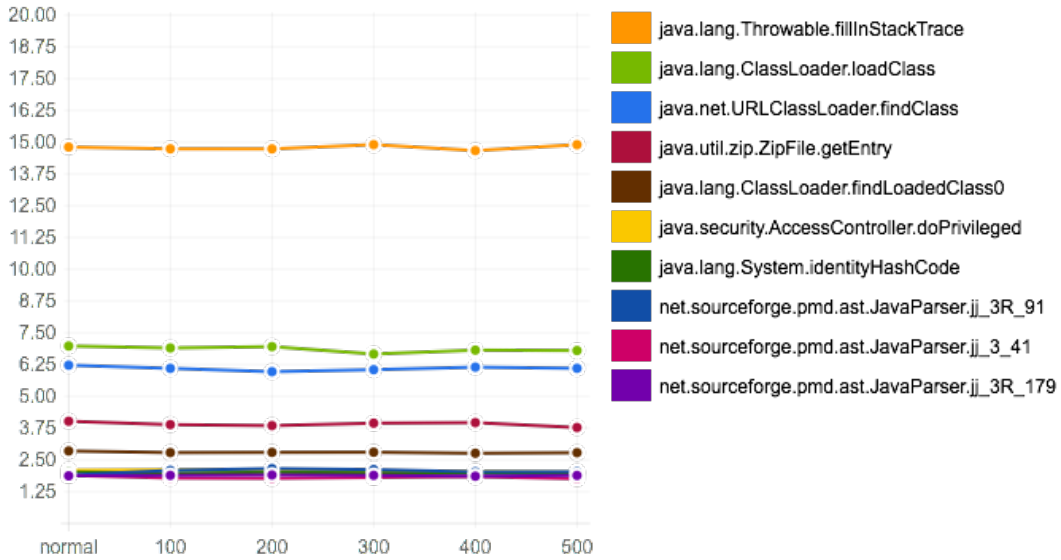



*Figure J.6: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

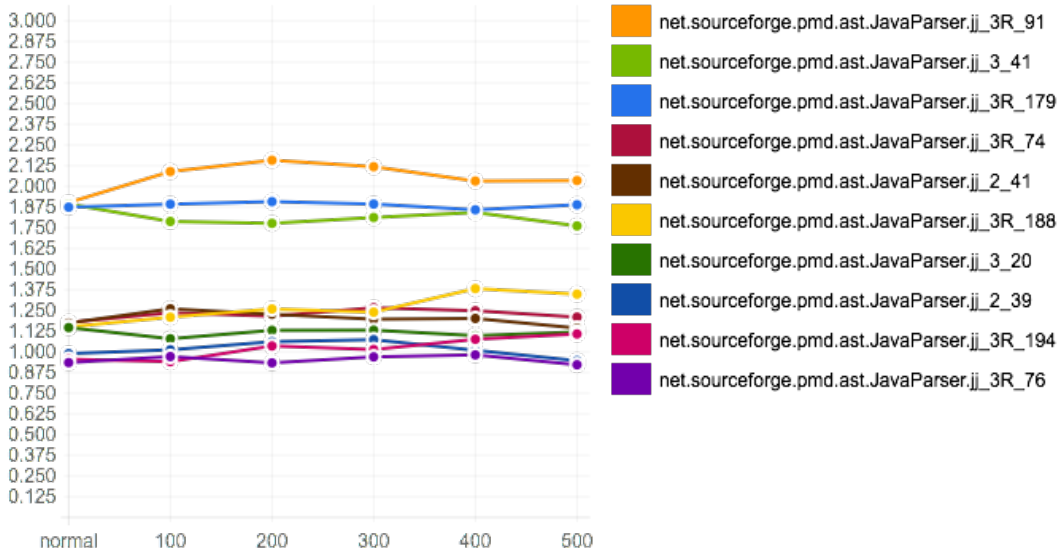



*Figure J.7: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining)*

### J.3.2 ast.JavaParser.jj_3_41

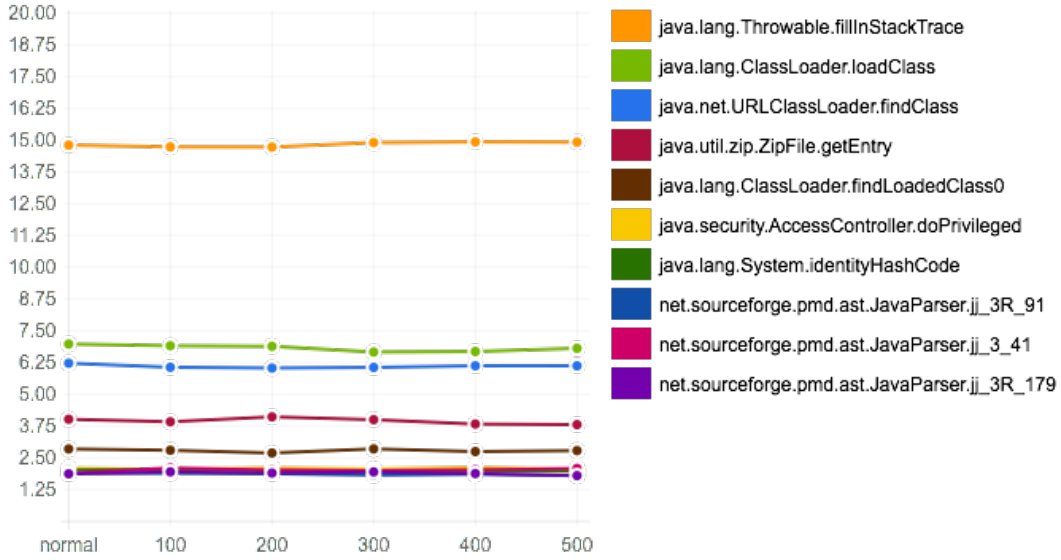



*Figure J.8: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

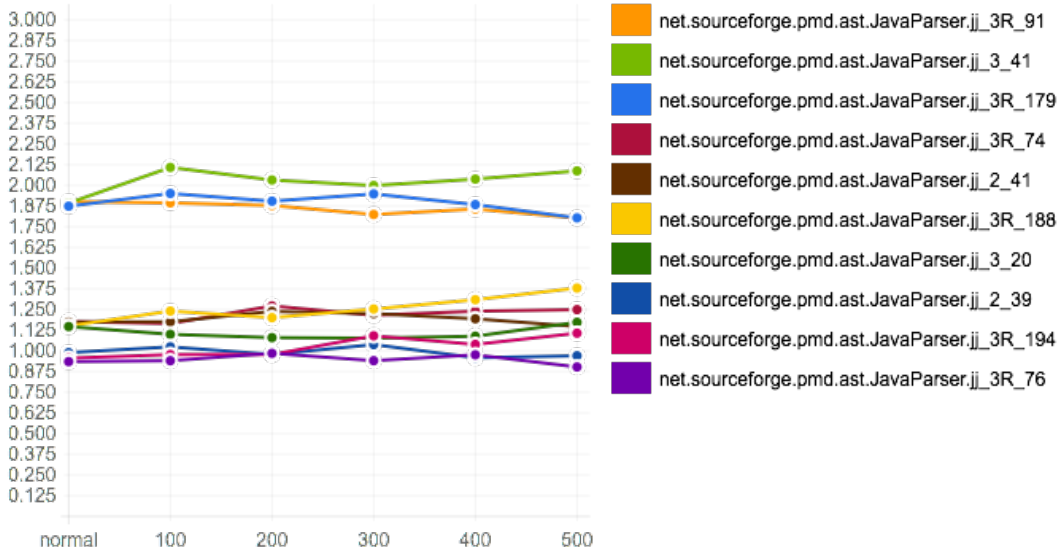



*Figure J.9: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining)*

## J.4 JProfiler

### J.4.1 ast.JavaParser.jj_3R_91

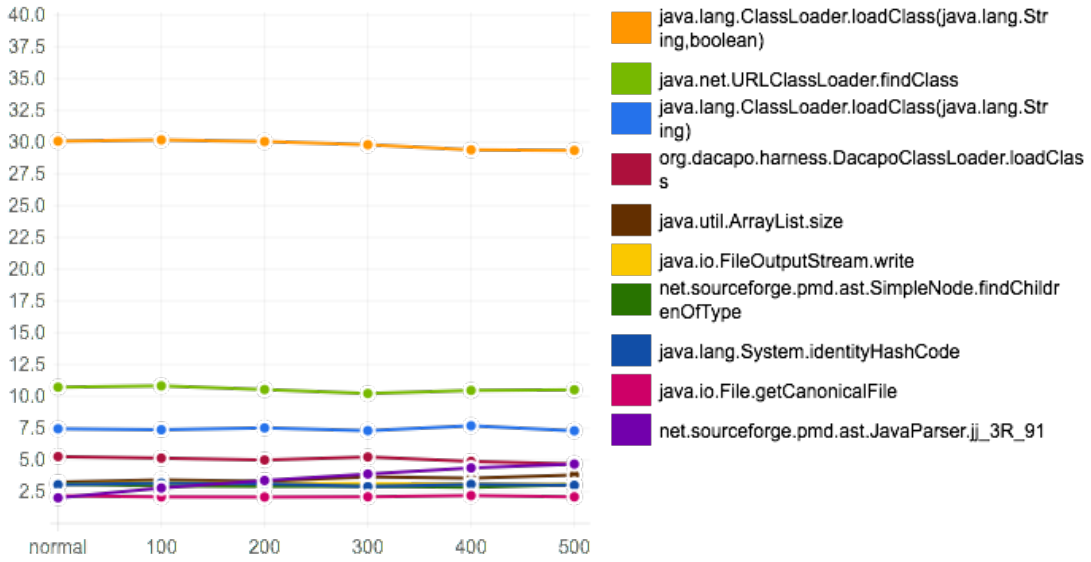



*Figure J.10: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

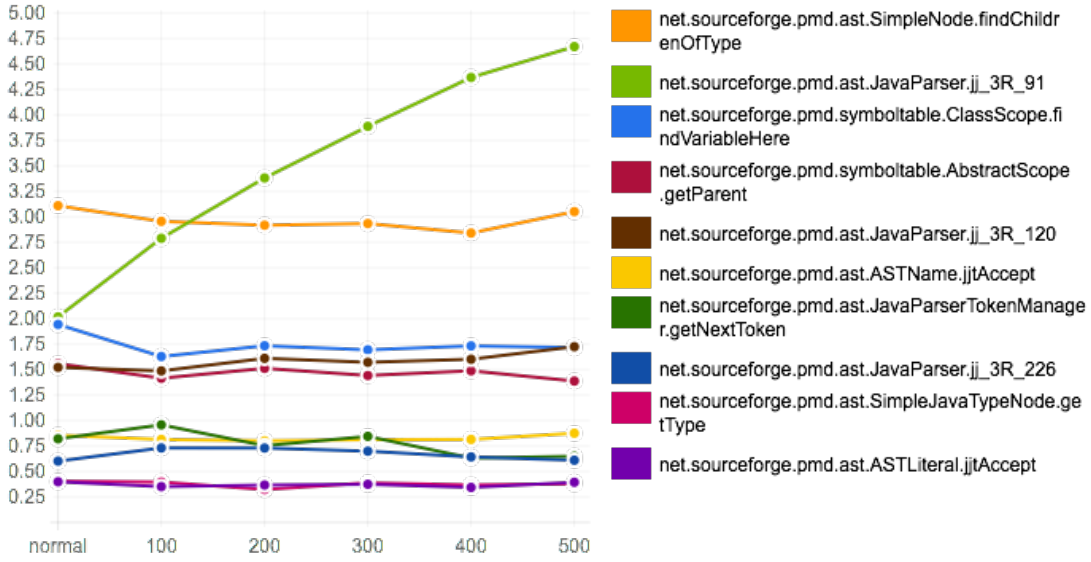



*Figure J.11: Effect on top 10 methods after being injected with the Fibonacci algorithm*

## J.5 JProfiler (inlining disabled)

### J.5.1 ast.JavaParser.jj_3R_179

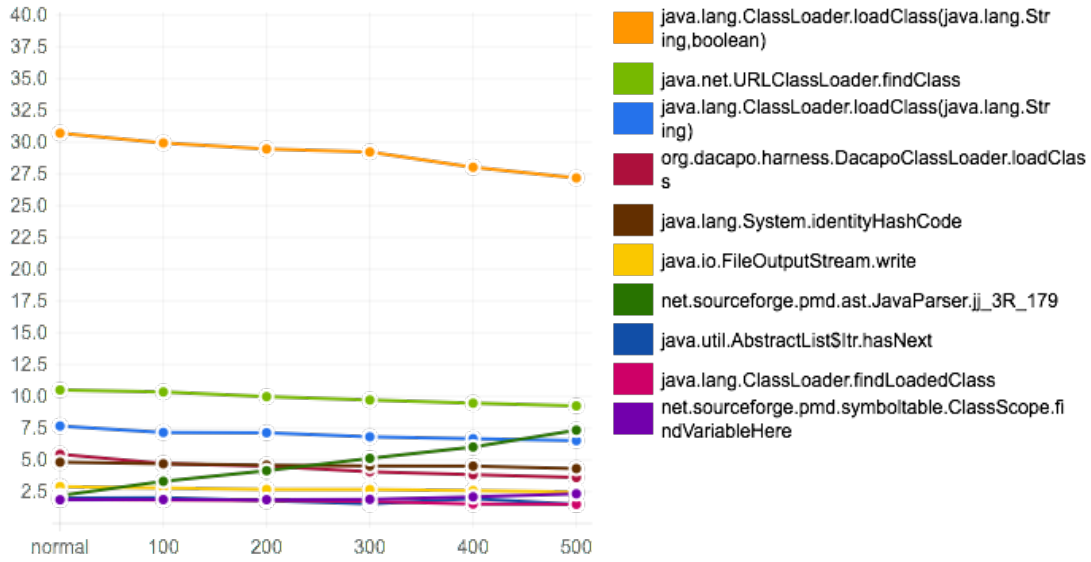



*Figure J.12: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

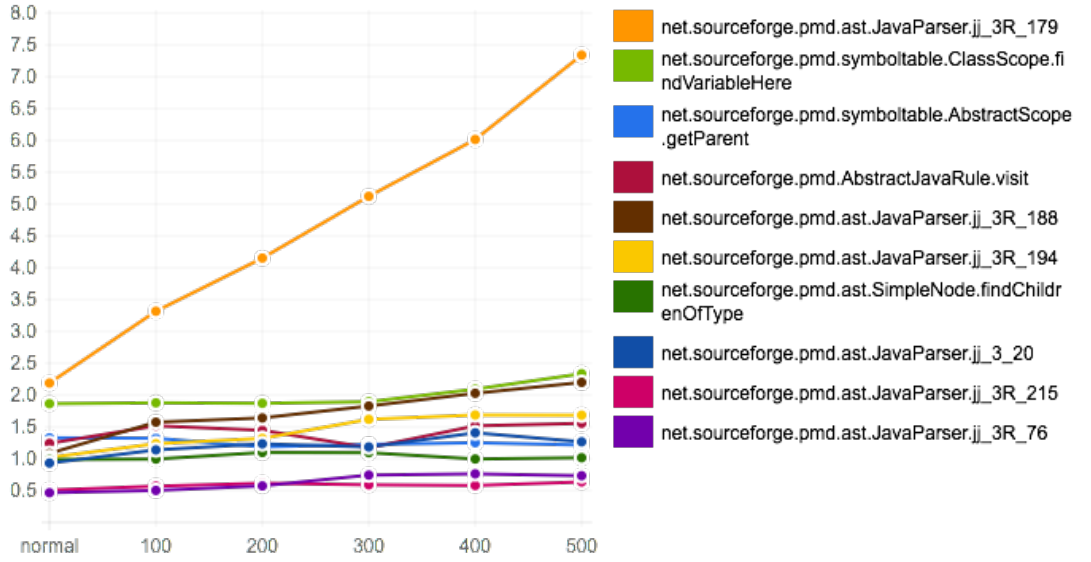



*Figure J.13: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining)*

## J.6 YourKit

### J.6.1 ast.JavaParser.jj_scan_token

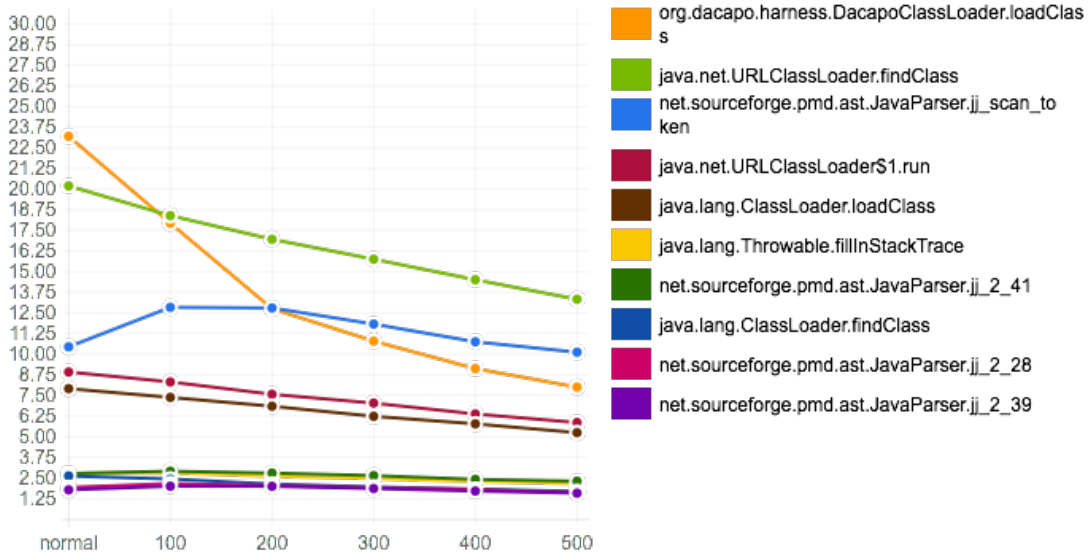



*Figure J.14: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

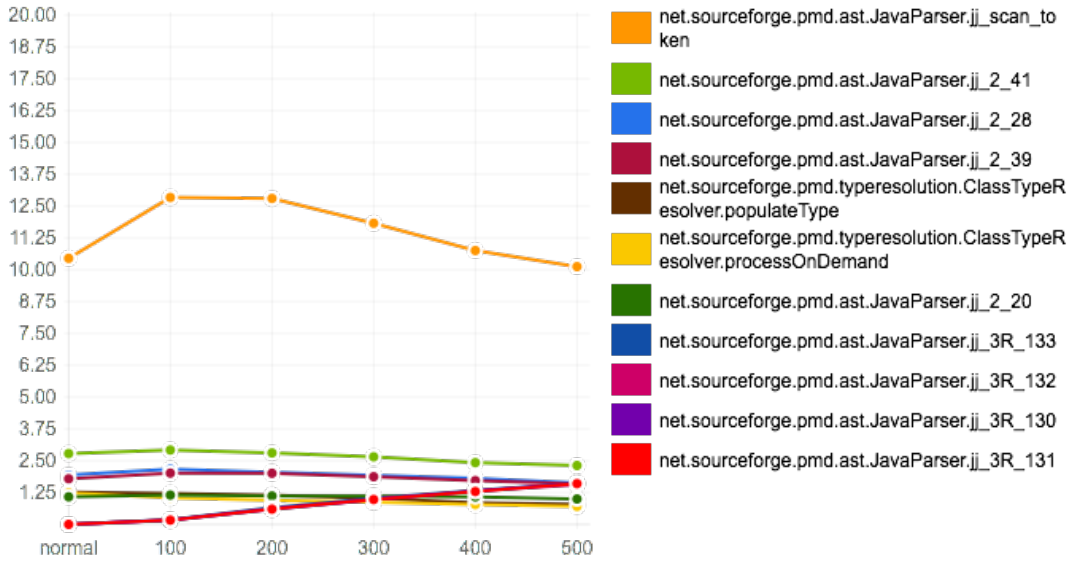



*Figure J.15: Effect on top 6 methods after being injected with the Fibonacci algorithm*

### J.6.2 ast.JavaParser.jj_2_41

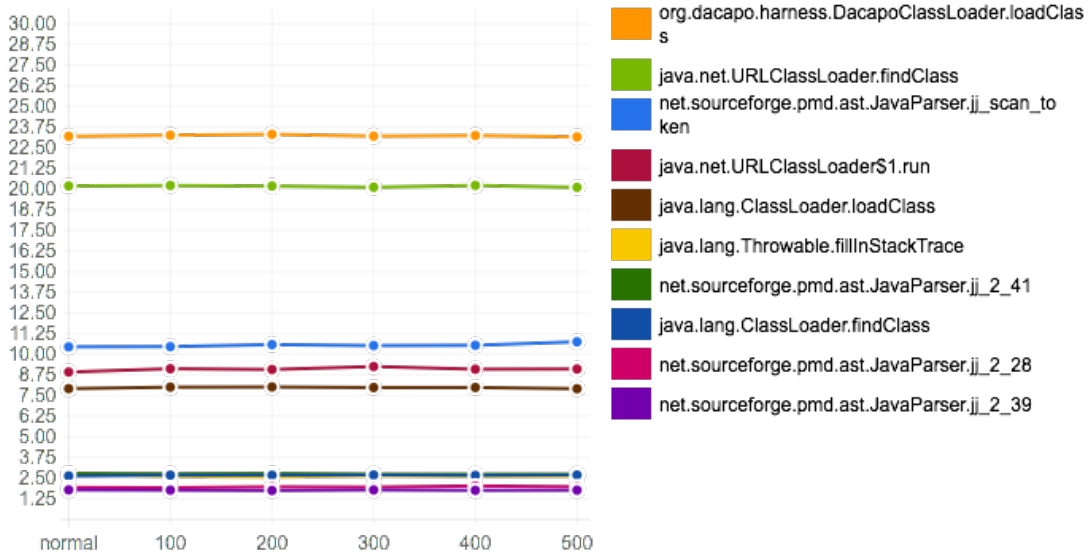



*Figure J.16: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

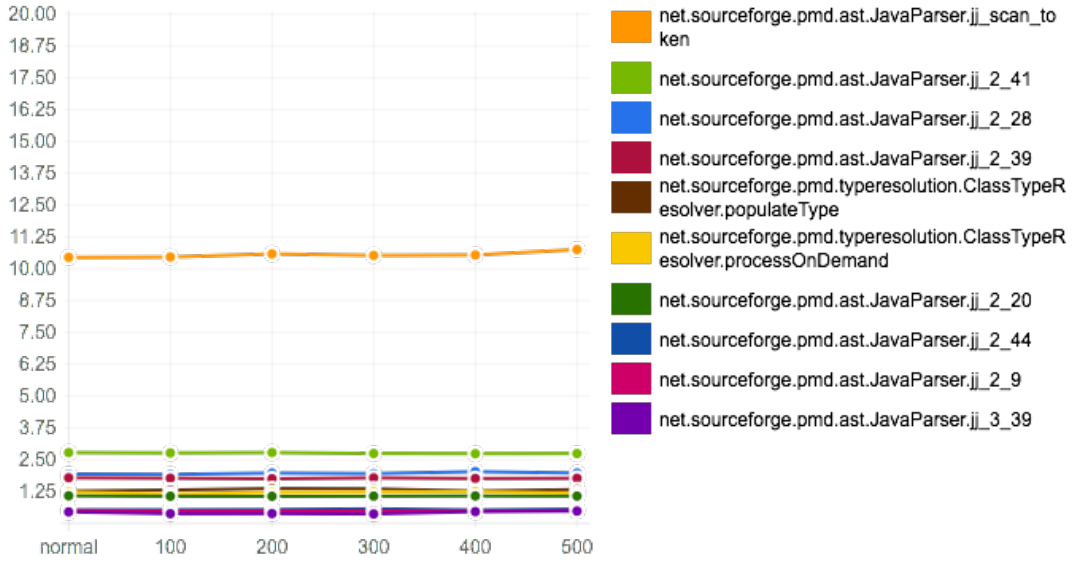



*Figure J.17: Effect on top 10 methods after being injected with the Fibonacci algorithm*

## J.7 YourKit (inlining disabled)

### J.7.1 ast.JavaParser.jj_scan_token

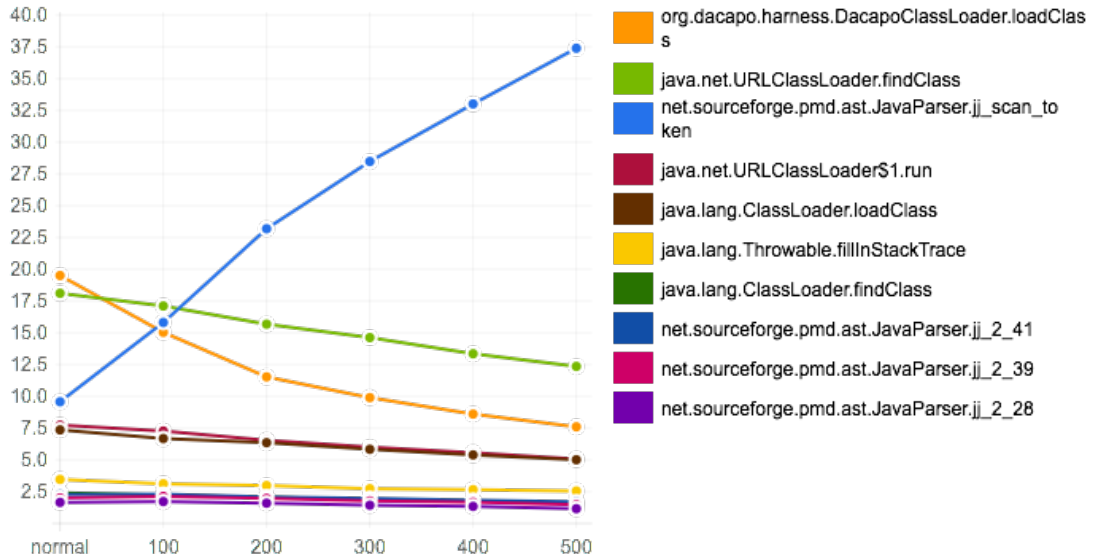



*Figure J.18: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

### J.7.2 ast.JavaParser.jj_2_41

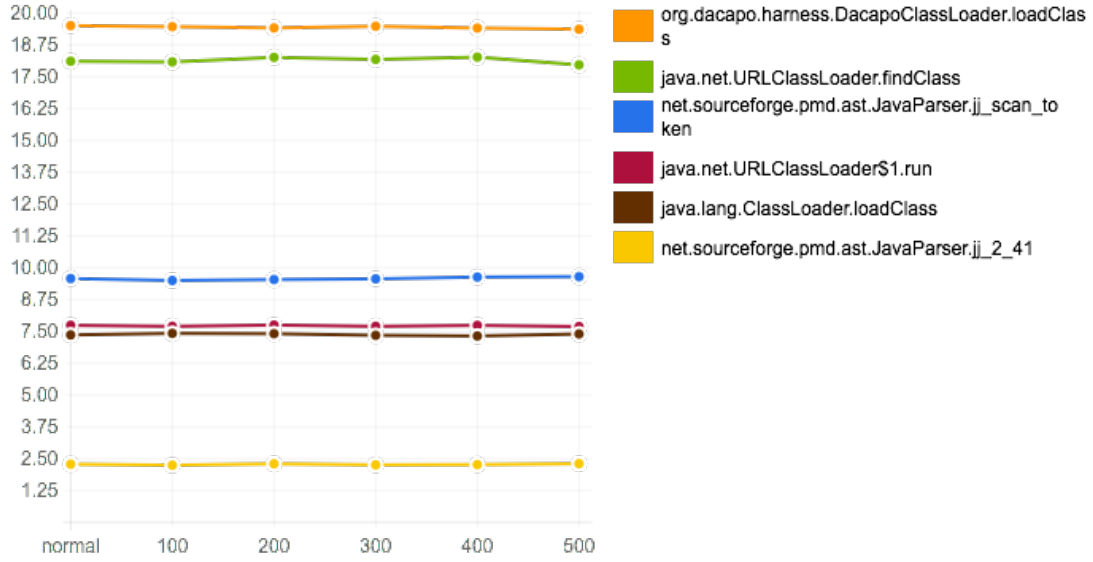



*Figure J.19: Effect on top 5 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

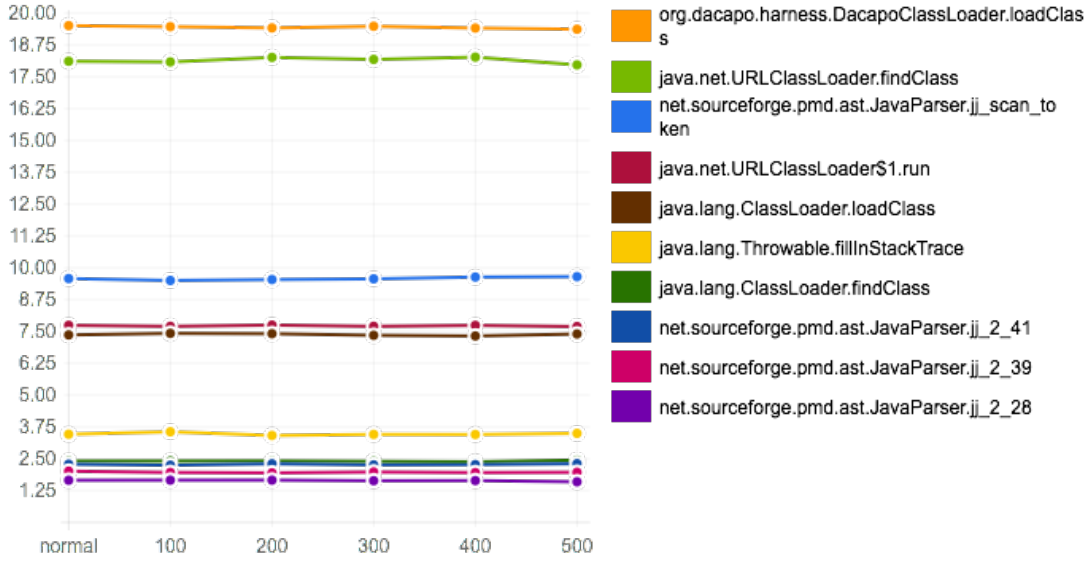



*Figure J.20: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

## J.8 LightWeight profiler

### J.8.1 ast.SimpleJavaNode.childrenAccept

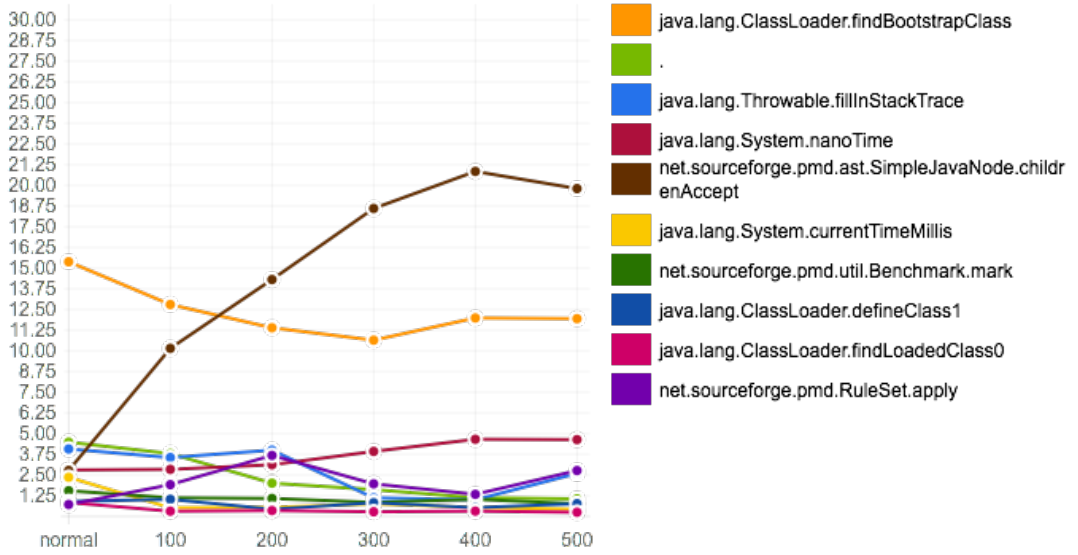



*Figure J.21: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

### J.8.2 util.Benchmark.mark

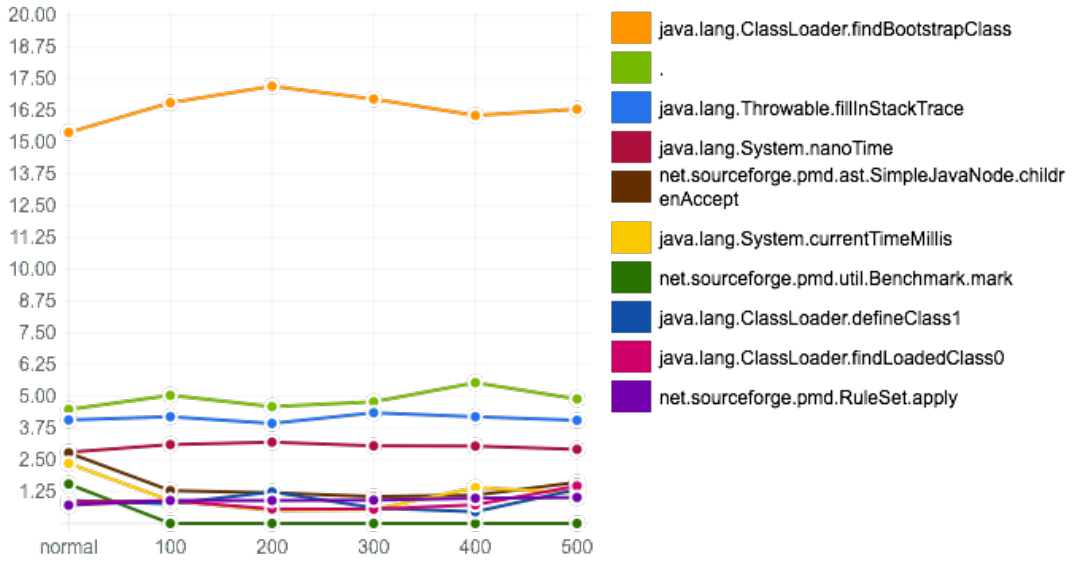



*Figure J.22: Effect on top 10 methods after being injected with the Fibonacci algorithm (only own methods shown)*

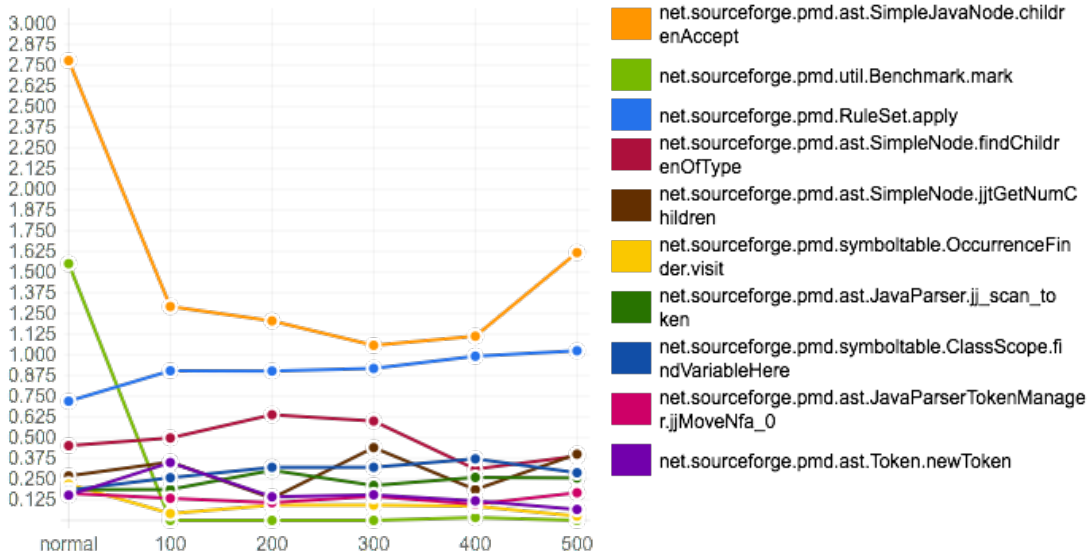



*Figure J.23: Effect on top 10 methods after being injected with the Fibonacci algorithm*

## J.9 LightWeight profiler (inlining disabled)

### J.9.1 AbstractRuleChainVisitor.visitAll

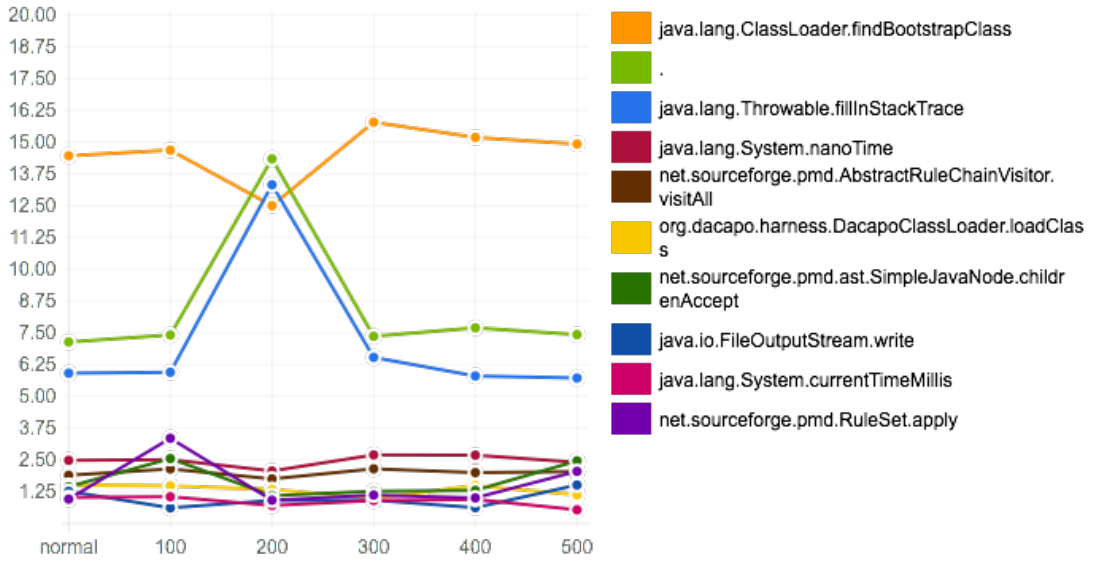



*Figure J.24: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

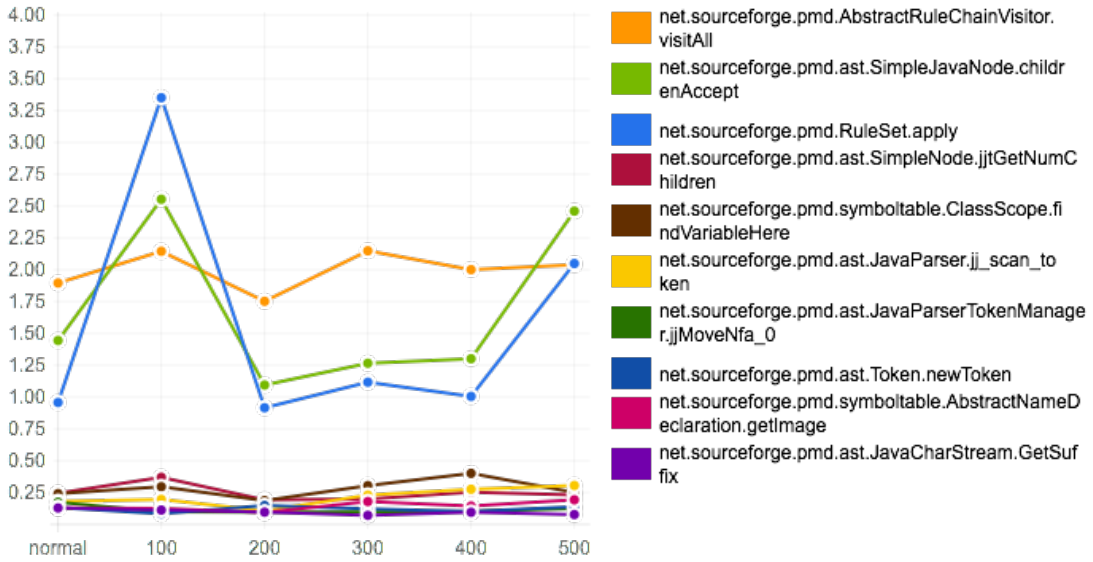



*Figure J.25: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining)*

### J.9.2 ast.SimpleJavaNode.childrenAccept

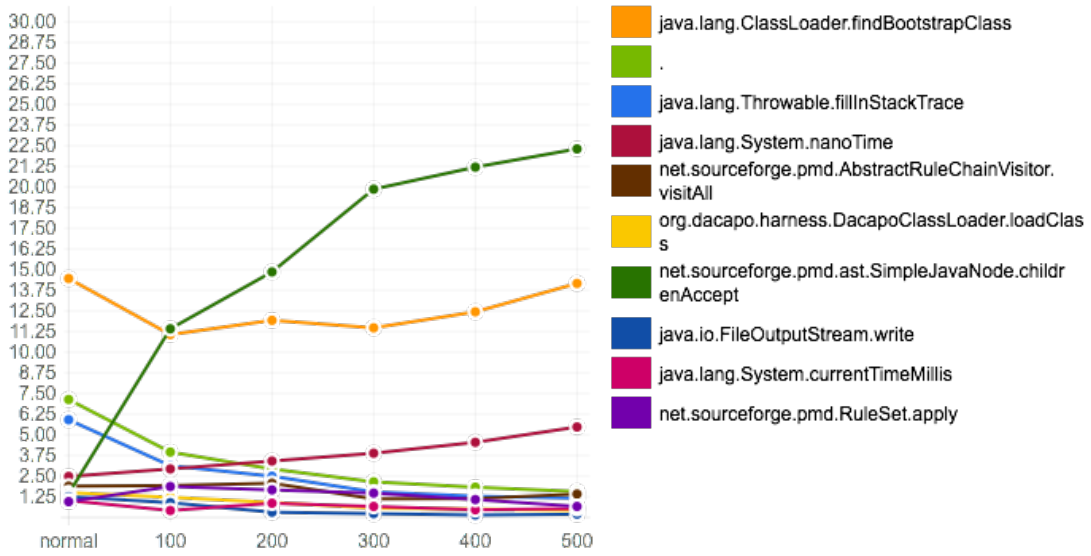



*Figure J.26: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining) (only own methods shown)*

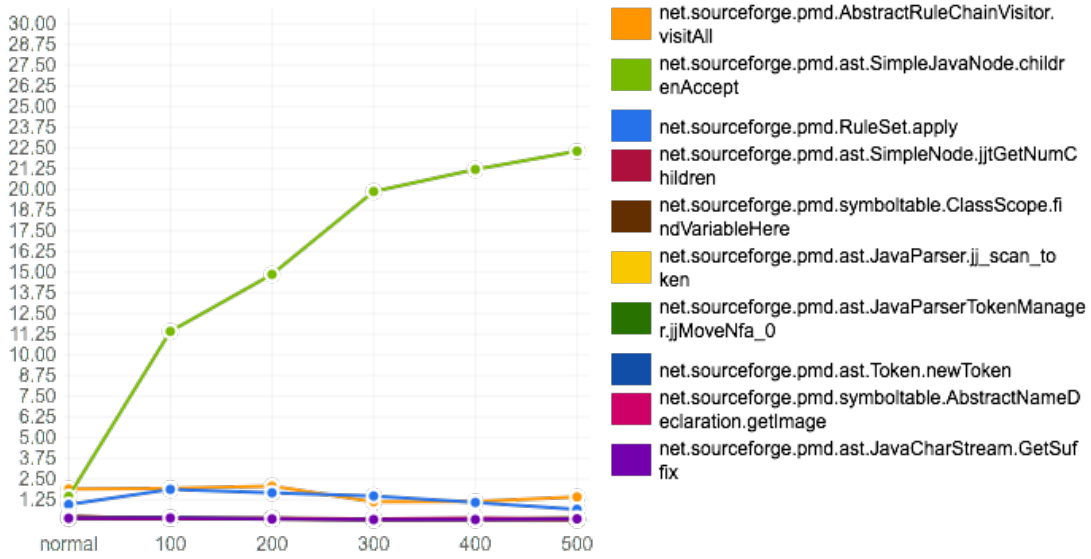



*Figure J.27: Effect on top 10 methods after being injected with the Fibonacci algorithm (without inlining)*

# K DaCapo Sunflow bytecode injecting results

All injection tests of this chapter are run on Machine C.

## K.1 Xprof

### K.1.1 accel.KDTree.intersect

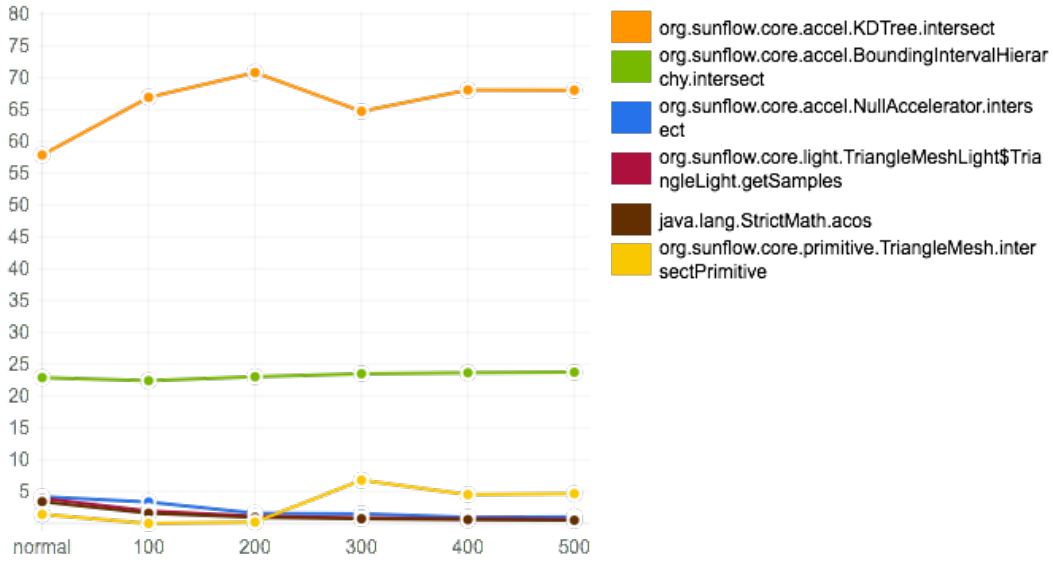



*Figure K.1: Effect on top 5 methods after being injected with the Fibonacci algorithm*

### K.1.2 accel.BoundingIntervalHierarchy.intersect

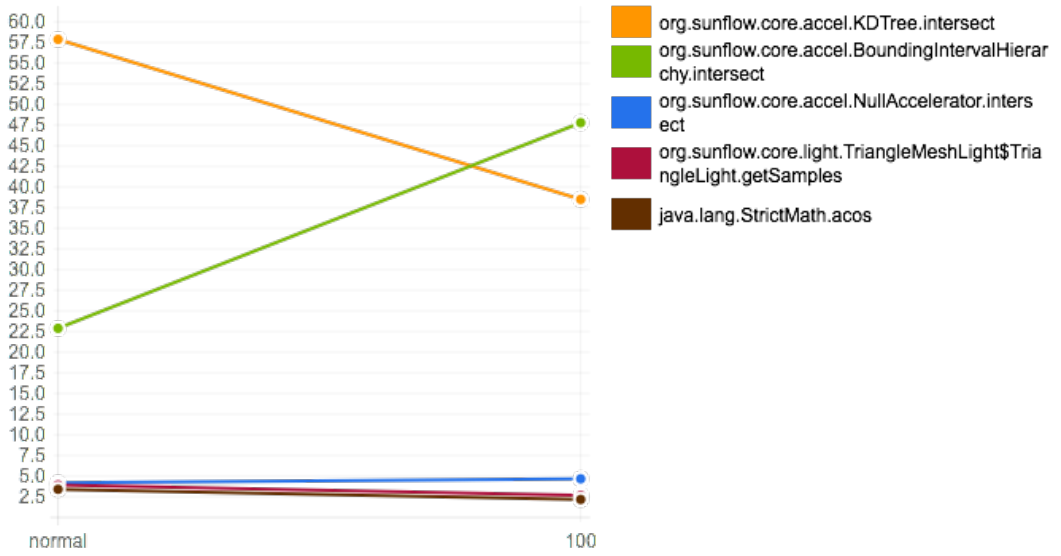



*Figure K.2: Effect on top 5 methods after being injected with the Fibonacci algorithm*

## K.2 Xprof (inlining disabled)

### K.2.1 accel.KDTree.intersect

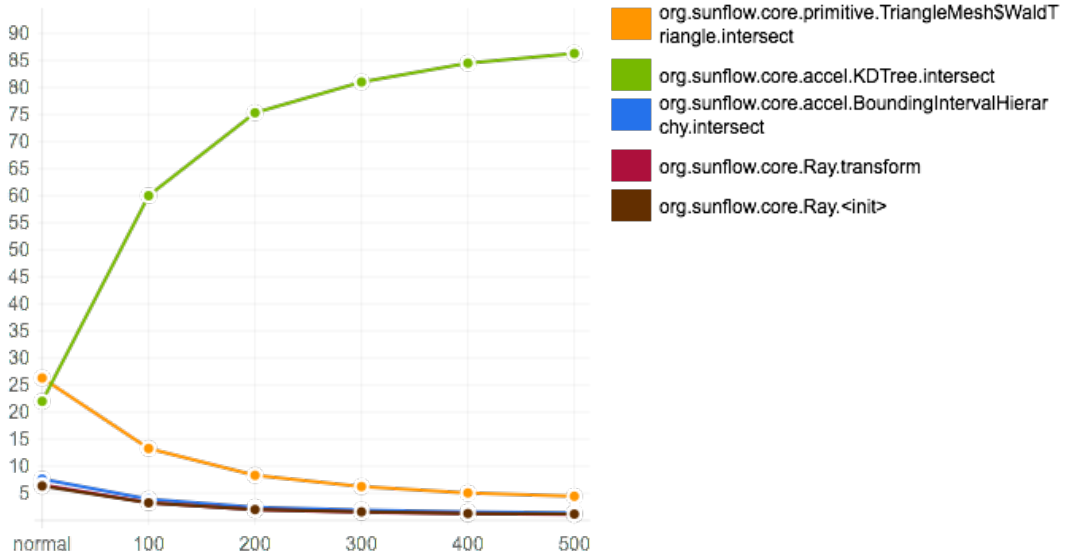



*Figure K.3: Effect on top 5 methods after being injected with the Fibonacci algorithm (without inlining)*

## K.3 Hprof

### K.3.1 primitive.TriangleMesh.intersectPrimitive

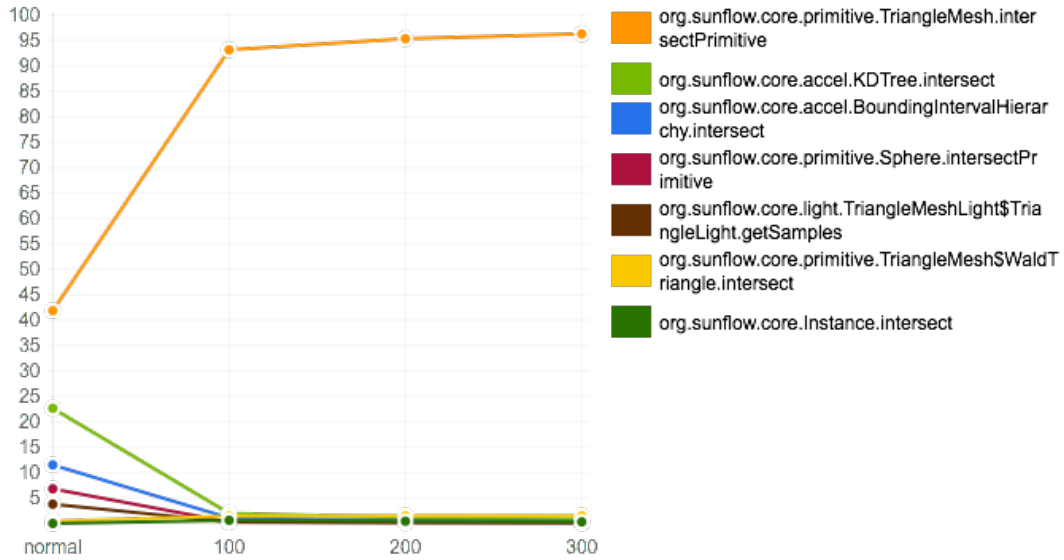



*Figure K.4: Effect on top 5 methods after being injected with the Fibonacci algorithm*

### K.3.2 accel.KDTree.intersect

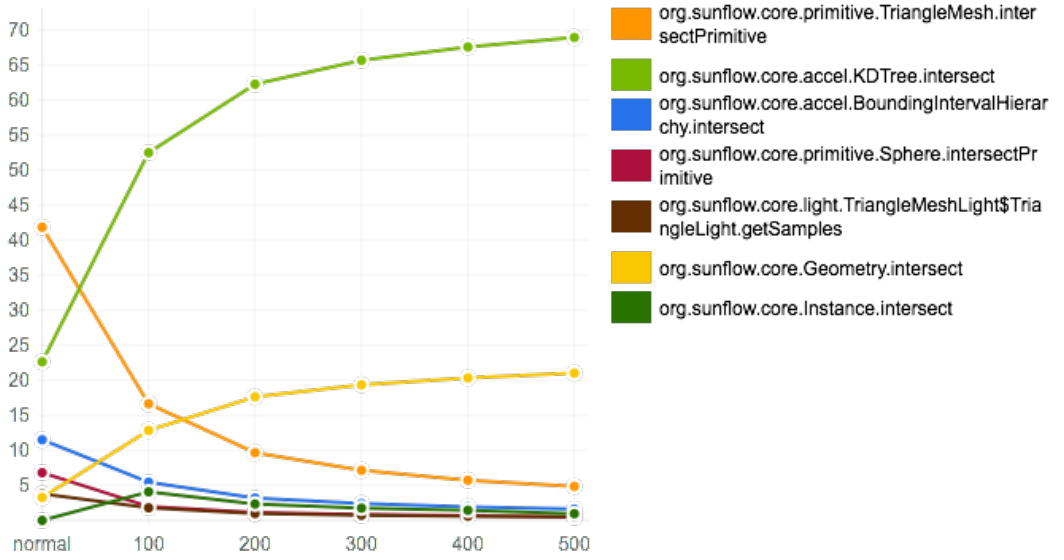



*Figure K.5: Effect on top 5 methods after being injected with the Fibonacci algorithm*

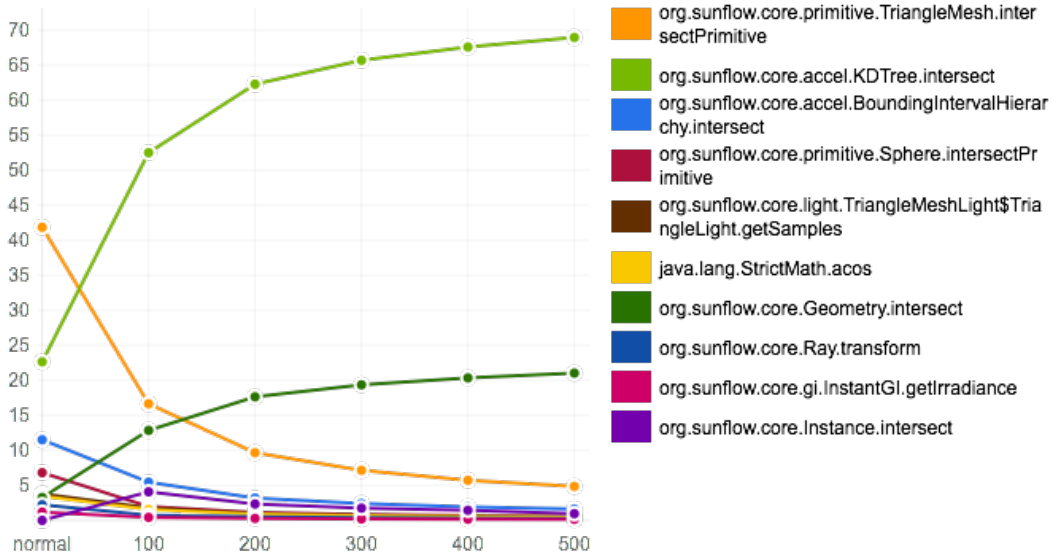



*Figure K.6: Effect on top 9 methods after being injected with the Fibonacci algorithm*

## K.4 Hprof (inlining disabled)

### K.4.1 accel.KDTree.intersect

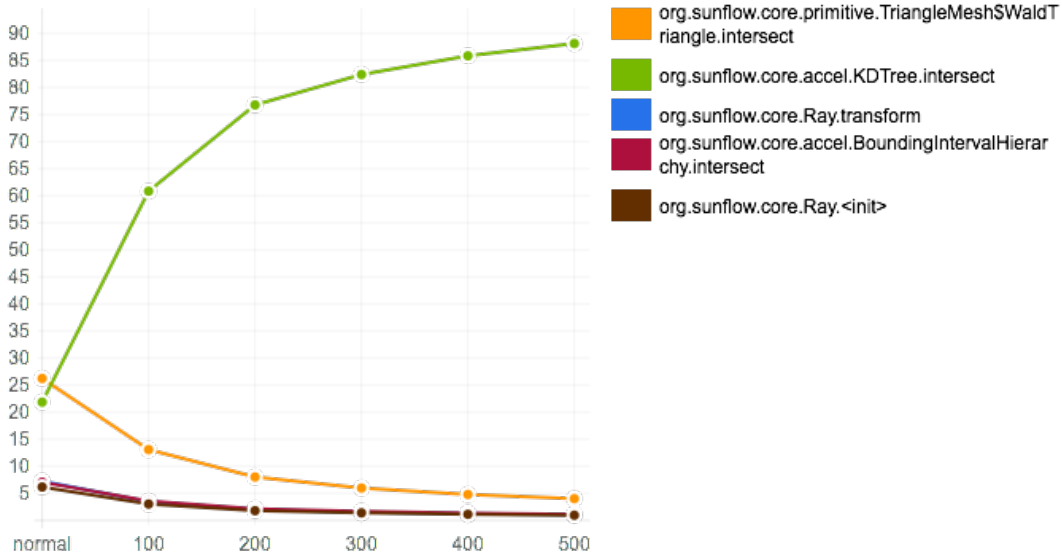



*Figure K.7: Effect on top 5 methods after being injected with the Fibonacci algorithm (without inlining)*

## K.5 JProfiler (all methods)

### K.5.1 primitive.TriangleMesh.intersectPrimitive

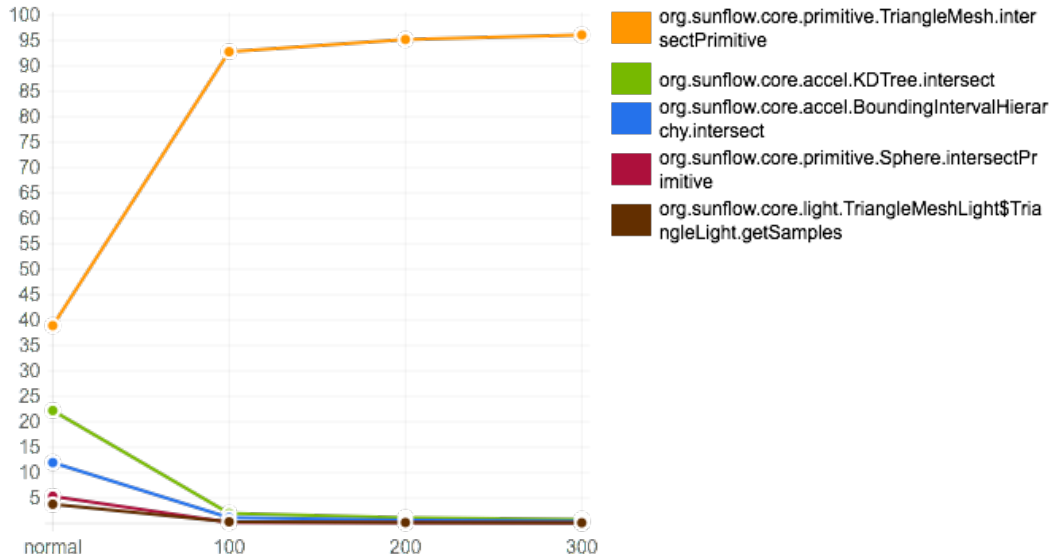



*Figure K.8: Effect on top 5 methods after being injected with the Fibonacci algorithm*

### K.5.2 accel.KDTree.intersect

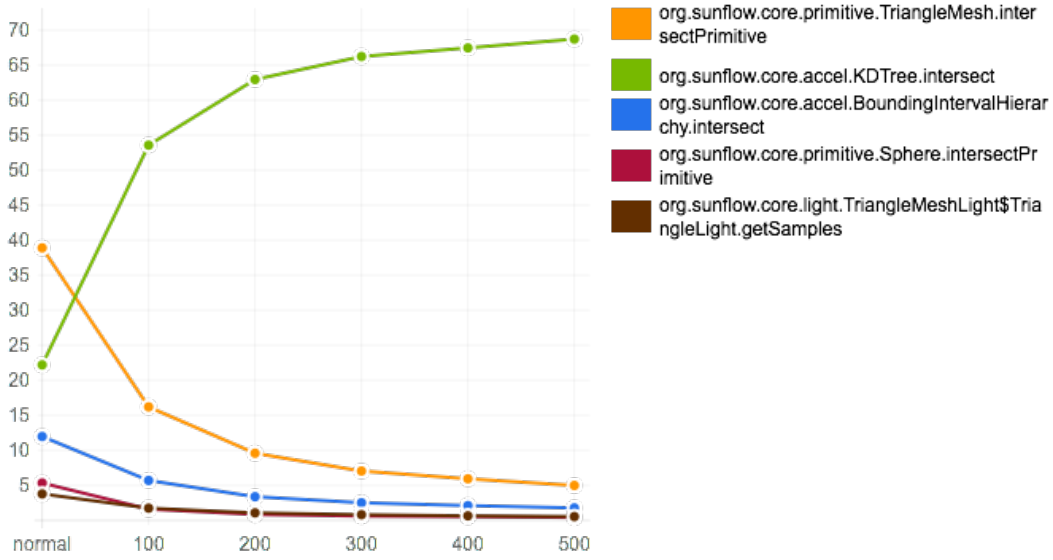



*Figure K.9: Effect on top 5 methods after being injected with the Fibonacci algorithm*

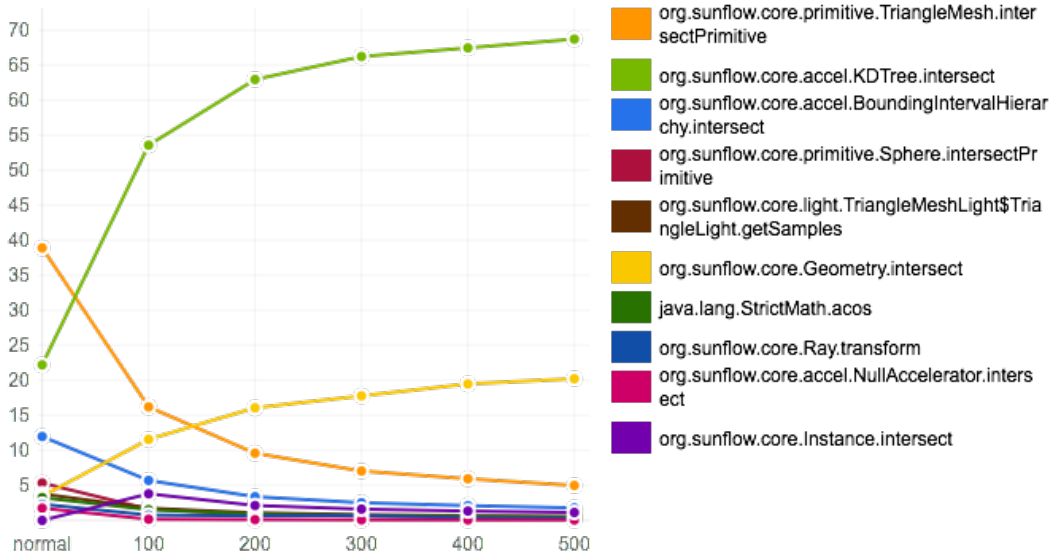



*Figure K.10: Effect on top 9 methods after being injected with the Fibonacci algorithm*

## K.6 JProfiler (all methods, inlining disabled)

### K.6.1 accel.KDTree.intersect

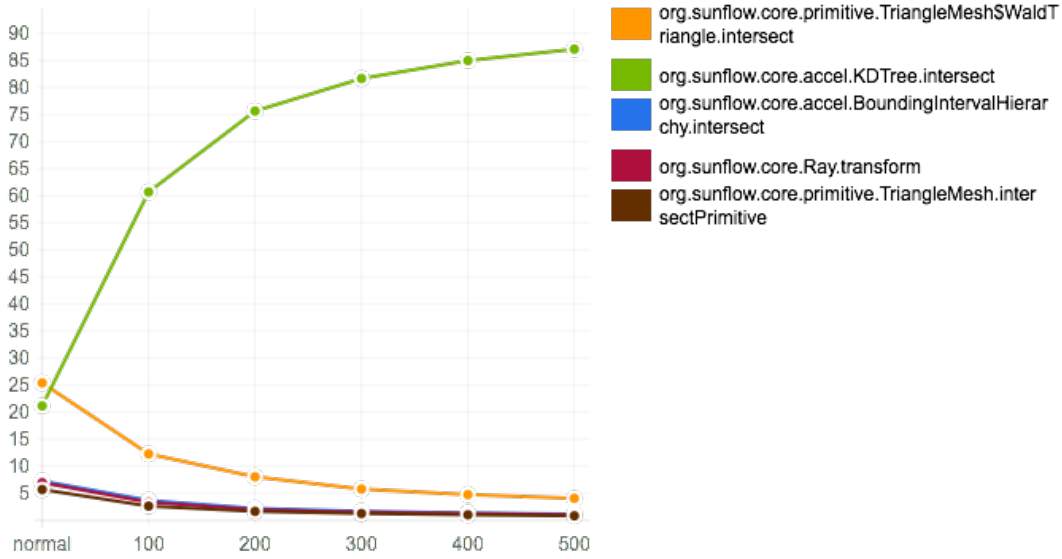



*Figure K.11: Effect on top 5 methods after being injected with the Fibonacci algorithm (without inlining)*

## K.7 YourKit

### K.7.1 primitive.TriangleMesh.intersectPrimitive

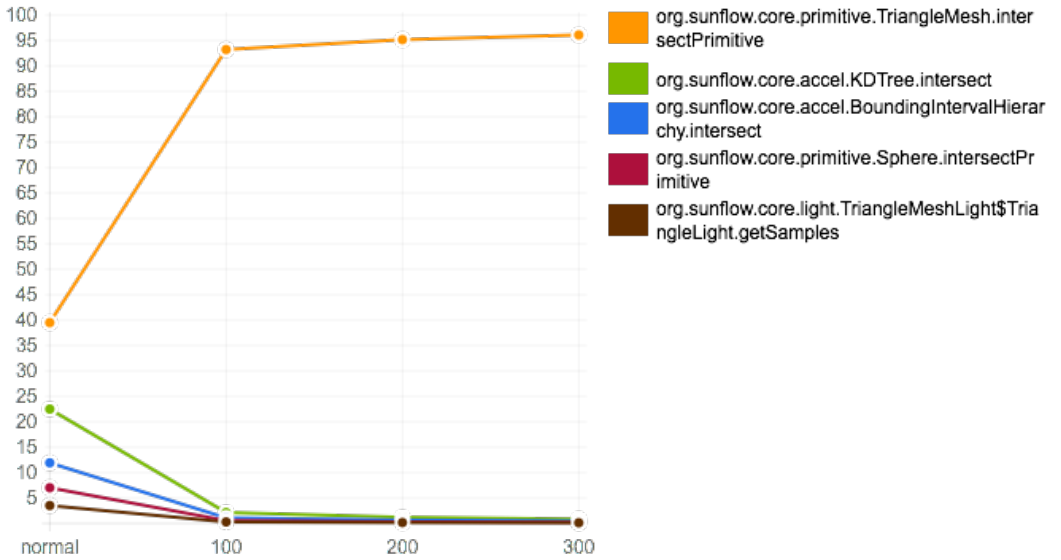



*Figure K.12: Effect on top 5 methods after being injected with the Fibonacci algorithm*

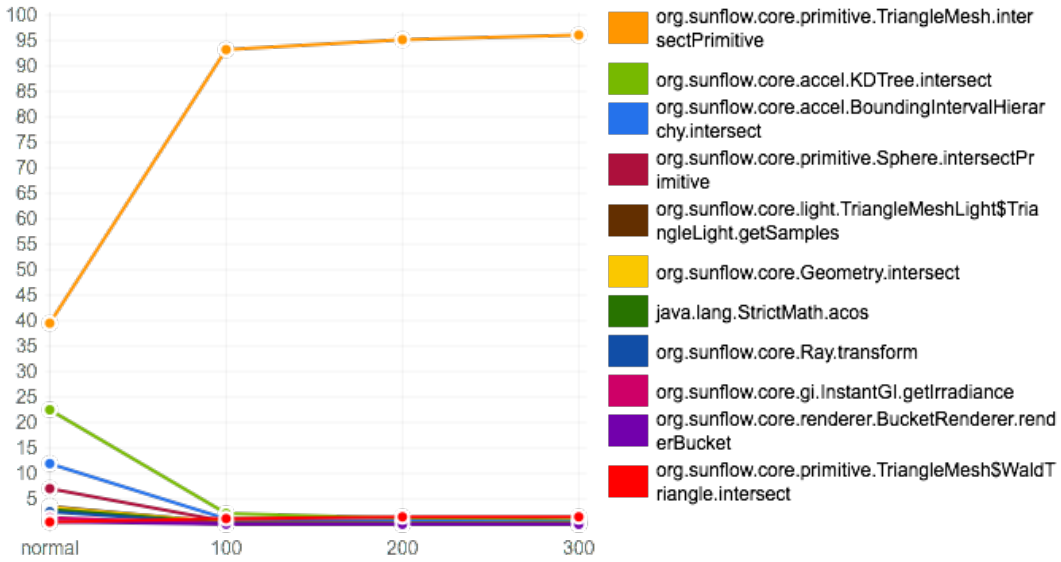



*Figure K.13: Effect on top 10 methods after being injected with the Fibonacci algorithm*

### K.7.2 accel.KDTree.intersect

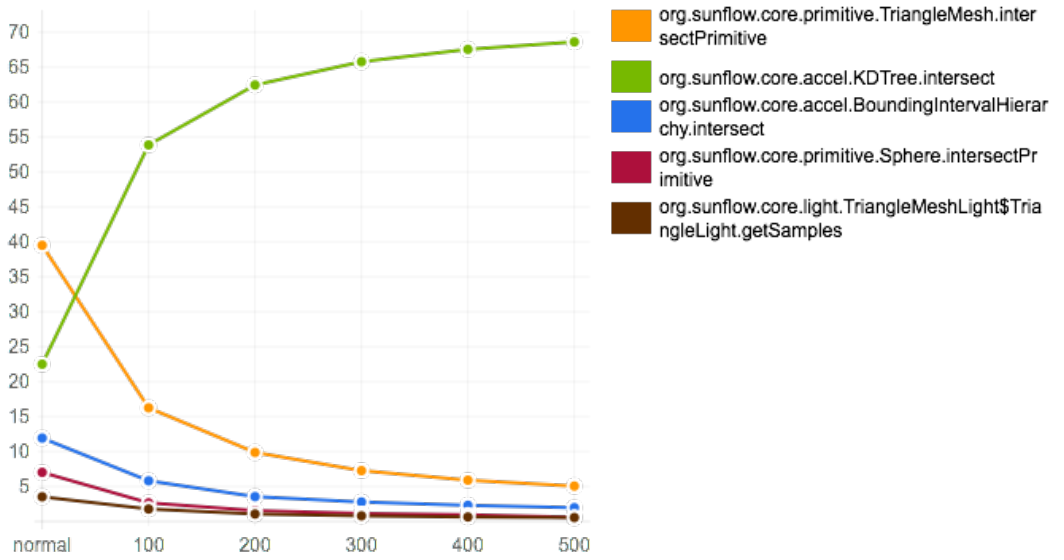



*Figure K.14: Effect on top 5 methods after being injected with the Fibonacci algorithm*

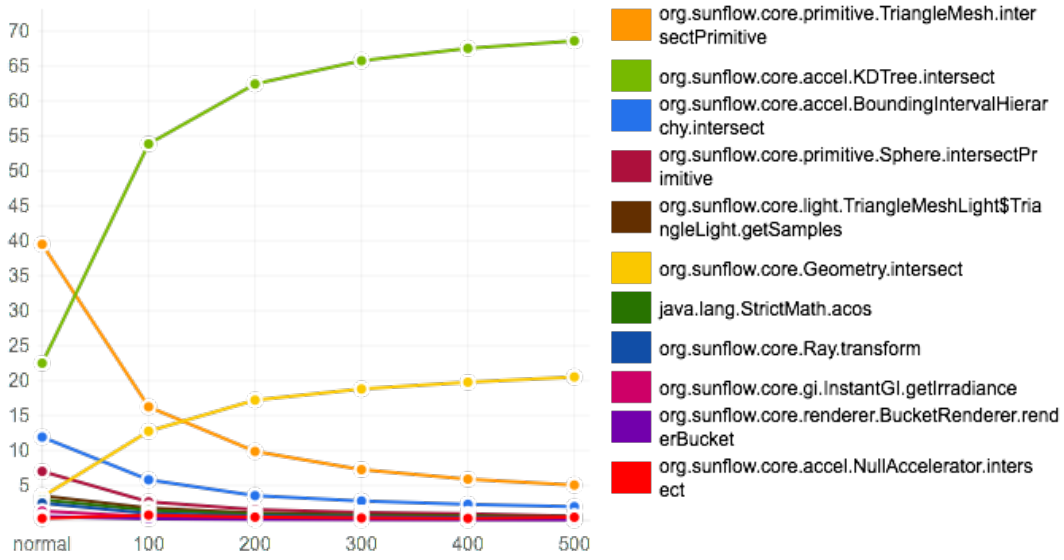



*Figure K.15: Effect on top 10 methods after being injected with the Fibonacci algorithm*