UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Tiit Oja

Optimizing JVM profiling performance for Honest Profiler

Bachelor's Thesis (9 ECTS)

Supervisor: Vootele Rõtov, MSc
Supervisor: Vesal Vojdani, PhD

Tartu 2018

Optimizing JVM profiling performance for Honest Profiler

Abstract: Honest Profiler is a profiling tool which extracts performance information from applications running on the Java Virtual Machine. This information helps to locate the performance bottlenecks in the application observed. This thesis aims to provide solutions to increase the amount of useful information extracted by Honest Profiler. Achieving this would increase the accuracy of the performance information collected by Honest Profiler. Thesis will cover the basics of sampling profiling, the architecture of Honest Profiler and measures the performance of Honest Profiler's data collection logic. As the main result of this thesis, three different solutions for increasing the profiler information output are presented. Their performance and the extracted information amount is evaluated by a benchmark test.

Keywords: Profiling, optimization, Honest Profiler

CERCS: P170, Computer science, numerical analysis, systems, control

JVM profileerimise jõudluse optimeerimine Honest Profileri baasil

Lühikokkuvõte: Honest Profiler on tööriist, mis võimaldab mõõta Java virtuaalmasina peal jooksvate rakenduste jõudlust. Tööriista poolt kogutud informatsiooni põhjal on võimalik optimeerida vaadeldava rakenduse jõudlust. Käesoleva töö eesmärk on luua lahendusi, mis suurendaksid Honest Profileri tööriista poolt kogutud informatsiooni hulka. Suurem andmete hulk muudab jõudluse mõõtmise tulemused täpsemaks. Töö kirjeldab profiilide kogumise ning Honest Profileri arhitektuuri põhitõdesid. Ühtlasi mõõdetakse Honest Profileri informatsiooni kogumise loogika jõudlust. Töö põhitulem on kolm erinevat lähenemist, mis suurendavad kogutud informatsiooni hulka. Kirjeldatud lahenduste jõudlus ning kogutud informatsiooni hulk verifitseeritakse jõudlustesti abil.

Võtmesõnad: Jõudluse mõõtmine, optimeerimine, Honest Profiler

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Profiling is an activity which aims to identify performance issues in a program being observed. This task often relies on specific tools called profilers for extracting information from the program's execution. The information obtained by profilers helps to identify and locate the methods that have the largest effect on the program's execution time. Such methods are usually worth investigating as they affect the application's performance the most [1].

As the reliability of the obtained performance information relies on the accuracy of the profiling tool used, it is important to have sufficient and actionable information from the profiler to construct accurate profiles of the application's performance. This thesis investigates a specific profiler implementation named Honest Profiler. It is an open source sampling profiling tool which can be used to evaluate the performance of Java applications. The goal of this thesis is to increase the amount of information extracted from a Java application by Honest Profiler. Doing so potentially increases the accuracy and reliability of the information that is extracted from the observed application.

This thesis covers the basics of sampling profiling methods and the problems these methods have in the context of Honest Profiler. It explains the architecture of Honest Profiler and measures the performance of its profiling logic. The main result of this thesis is providing means to increase the amount of information that Honest Profiler can extract from the application under observation. The suggested solutions are then tested on a benchmark test to evaluate their performance and amount of useful information extracted.

2 Sampling profiling methodologies

Sampling profilers are a type of profilers which gather call traces from the observed program at varying intervals. A sample in the form of a call trace is a representation of a single thread's state at a particular moment in time. A simple call trace of a thread is presented in Listing 1.

```
"main" #1 prio=5 os_prio=0 tid=0x00007feccc224000 nid=0x1f2c
runnable [0x00007fecd5660000]
   java.lang.Thread.State: RUNNABLE
        at ee.ut.SimpleBenchmark.methodA(SimpleBenchmark.java:28)
        at ee.ut.SimpleBenchmark.doWork(SimpleBenchmark.java:22)
        at ee.ut.SimpleBenchmark.main(SimpleBenchmark.java:11)
```

Listing 1: Call trace of a thread

Gathered samples must then be processed in order to gain relevant information about the application's performance. Identical call trace samples can be collapsed into a single entity which shows how many samples represent a particular application's state. Distribution of the amount of identical samples highlights the hotspots in the program under observation. Higher occurrence of a sample suggests that more program's execution time was spent in that particular state.

Figure 1 visualizes the distribution of the gathered samples. This shows a general overview of the application's performance. The visualization was created by a tool called Flamegraphs which provides means to generate a comprehensive and intuitive visualization of the gathered call trace samples [2]. Visualization tools are necessary to produce meaningful representations of the collected data as unprocessed call trace samples *per se* are not informative without the context of other samples' frequency.

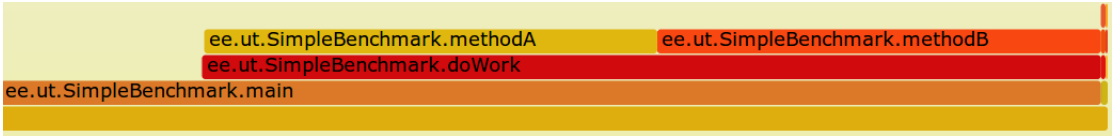

Figure 1: Visualization of sampling profiling output

The samples for the visualization in Figure 1 were gathered from a simple program that executed two identical methods, `methodA` and `methodB`, alternately in a loop. During its execution 481 usable samples were collected. 198 of those samples contained `methodA` in its top call frame and 194 contained `methodB` in its top frame. Samples with `ee.ut.SimpleBenchmark.main` in its top stack frame may

have been obtained during benchmark initialization or executing the loop's control flow instructions.

Although sampling profiling does not provide precise metrics for each method's execution time, it can provide a general overview of the time spent in the profileable application's context. Such information is often sufficient to make actionable optimizations in the observed application.

2.1 Sampling profiling challenges

For sampling profilers, the result is a statistical approximation of the program's performance. Thus, having more samples will provide a more accurate approximation.

Figure 2 illustrates the problem which is caused by the low sampling interval. Suppose that the Figure 2 resembles a program's execution in which the horizontal axis represents the current program's call trace state in that particular moment in time. It can be observed that the amount of time spent in method Y is significantly larger than the amount of time that was spent in methods X and Z. Suppose that the sampling profiler obtains a sample at each dotted vertical line. Such profiling results would represent each method X, Y and Z with a single sample. This implies that all the methods spent roughly the same amount of time during the execution of the program. The result is inaccurate as the visualization clearly shows that method Y spent roughly 4 times more time than it was spent for executing methods X and Z.

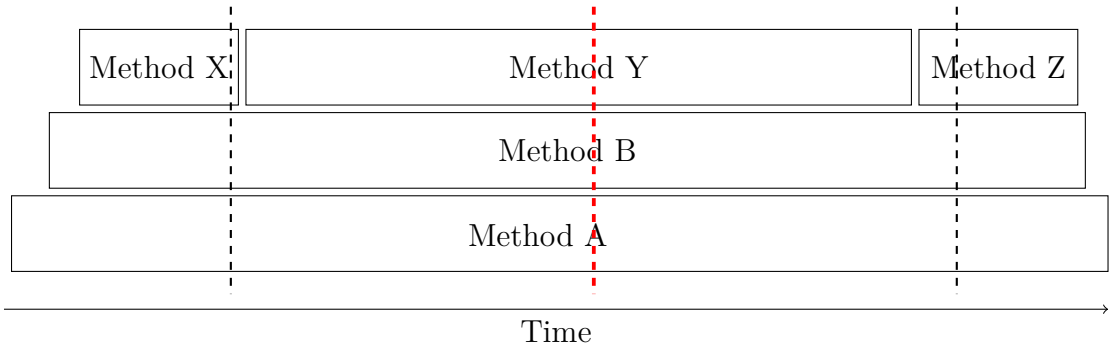

Figure 2: Scenario with low sampling interval

Increasing the sampling frequency would improve the situation as demonstrated on Figure 3 on page 7. Upon profiling with four times higher sampling frequency, it becomes apparent that method Y call trace samples have been proportionally

represented in the total call trace samples. The call trace samples containing the method Y in its top frame are colored red.

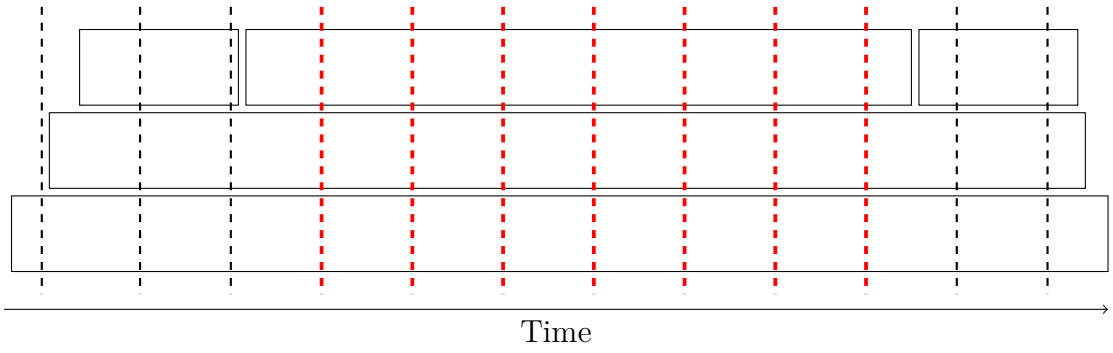

Figure 3: Scenario with sufficient sampling interval

Following subsections will provide examples of problematic scenarios in which the accuracy of a sampling profiler is negatively affected.

2.1.1 Periodicity bias

This problem occurs when the sampling interval catches on to some program's routine which executions match the sampling interval. Figure 4 illustrates the issue behind the bias. Suppose that the program to be observed runs methods X and Y alternately for some constant time period. If the dotted lines are the marks for the call trace samples taken during profiling, the results would be skewed as not a single sample represents method Y in the results.

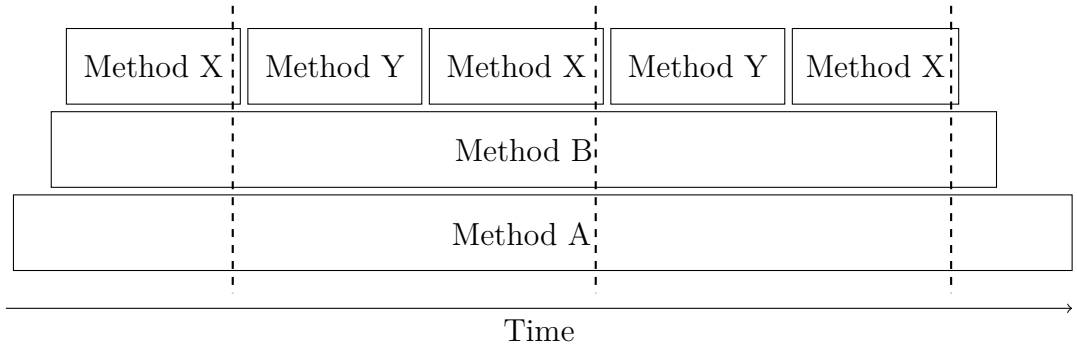

Figure 4: Illustration of periodicity bias

Possible way to tackle this problem would be to randomize the sampling interval by

having a random number of time units offseting the sampling interval. Increasing sample gathering frequency will also help to alleviate this problem.

2.1.2 Safepoint bias

Sampling profiling assumes that the obtained samples are acquired uniformly and randomly as bias in the samples could potentially yield inaccurate results. In light of this, when gathering samples from a program running on the Java Virtual Machine, one must take *safepoints* into consideration.

Safepoints in the Java Virtual Machine are defined as points during the program's execution during which all of the threads are in a consistent and well known state. Safepoints are necessary for various Java Virtual Machine's operations such as the garbage collection, method deoptimization and class redefinition [3]. It has been shown that many of the existing sampling Java profilers require the Java Virtual Machine to be stopped on a safepoint in order to obtain the call trace sample from the profileable thread [4]. However, this mechanism also casts a shadow on the profling results as this implies that the samples are not acquired randomly but rather require the Java Virtual Machine to be in a specific state [1].

The execution of the code sample in Listing 2 illustrates this issue well. The nested loop in this example is to avoid Java compiler specific optimizations.

```
int k = 0;
for (int i = 0; i < Integer.MAX_VALUE; i++) {
  for (int j = 0; j < 2; j++) {
      k++;
      if ((k % 2) == 1) k++;
  }
}
```

Listing 2: Counted loops do not contain safepoints

As various Java Virtual Machine routines are nondeterministic, it is impossible to predict when does the Java Virtual Machine signal its threads to be stopped on a safepoint. However, if such request should occur in the middle of the counted loop's execution in Listing 2, the thread executing the loop's instructions will not stop until the loop has finished. Thus, if an ordinary sampling profiler signals the thread for a call trace sample, it is delayed until the thread executing the loop's instructions finishes its task [5].

To measure the actual time that is spent waiting for the Java Virtual Machine to stop all its threads on a safepoint, `-XX:+PrintGCApplicationStoppedTime` flag can be used. Appending this flag to the Java Virtual Machine arguments enables

outputing the time it takes to stop all threads in order to execute a subroutine (e.g. garbage collection) and the time that this operation took in total. Due to nondeterminicity, the sample code in Listing 2 on page 8 was executed 100 times. The worst case among 100 program executions recorded a Java Virtual Machine operation that took 7.517 seconds and 99.999% of that time was spent on reaching a safepoint. It is worth mentioning that this operation took 65% of the whole program's execution time.

Previously described example in Listing 2 on page 8 is also relevant for sampling profilers. If a safepoint biased profiler wishes to obtain a sample during the loop's execution, the sample aquisition is delayed until the loop has finished and the JVM has reached a safepoint. Such bias during obtaining the call trace clearly contradicts the randomness prerequisite for sampling profiling as samples can be only obtained if the profiler is in a specific state.

It is worth mentioning that the provided example is rather artificial and such occurrences in real life programs are rare on such large scale. Despite the presence of safepoint bias, such profilers can still produce actionable profiling results but lessen the accuracy when high detail granularity of the results is important.

2.1.3 Observer effect

Attaching a profiler to an application alters the way how the application would normally execute. The *observer effect* describes how a presence of a profiler can change the profiling outcome.

Firstly, a profiler does introduce some performance overhead due to its nature. Profilers need to execute additional instructions for gathering information from the application under observation. To have meaningful profiling results, this overhead must not affect the application's performance by a large margin.

The presence of a profiler could also change the way how the Java Virtual Machine does its optimizations. Java Virtual Machine makes use of just-in-time (JIT) compilation techniques at run time. JVM identifies methods which meet some criterias (e.g. called frequently) and compiles these methods into native machine code, resulting in an highly optimized code which improves the performance of such methods [6]. Various information extracting functions might change the way how the bytecode on the Java Virtual Machine is executed [1]. The presence of a profiler could alter which methods are eligible for JIT compilation. Thus, resulting in a different performance profile when compared to an application's execution in which the profiler is not attached.

2.2 Sampling profiling compared to alternatives

An alternative approach for gathering performance information is using tracing profilers. Tracing profilers rely on instrumenting the bytecode of the profileable application. In the context of tracing profilers, instrumentation is the act of adding additional instructions to the application being observed in order to gather relevant profiling information. Tracing profilers instrument the observed application by wrapping method calls with instructions to measure method execution time. Result is a collection of call stacks with included precise method timings [7]. Simplified idea of tracing profiler instrumentation is shown on Figures 3 and 4. Figure 3 shows the source code of a simple method and Figure 4 shows how a tracing profiler could instrument the code shown on Figure 3 in order to obtain information about method execution time.

```
public void work() {
  doWork();
}
```

Listing 3: Original source

```
public void work() {
  long start = System.nanoTime();
  doWork();
  long end = System.nanoTime() -start;
  // Profiler then persists the ‘end‘ value
}
```

Listing 4: Instrumented source

Using tracing profilers usually requires the user or the profiler to know the application elements that are worth profiling as instrumenting all the methods in the application under observation is too expensive performance wise.

Downside of tracing profiling is its performance and its susceptibility to observer effect. Tracing profilers rely on bytecode instrumentation which directly affects the way how the Java Virtual Machine performs its optimizations. Some optimizations might not be possible due to the added bytecode instructions added by the tracing profiler. Additionally, depending on the level of information detail and the amount of instrumentation performed, requiring higher information granularity could potentially introduce impactful overhead to the application's performance which could result in inaccurate profiling results.

When comparing sampling profiling to tracing profiling, it excels with its performance. Sampling profilers can potentially obtain thousands of samples each second with negligible performance overhead. Additionally, sampling profilers do not require intrumentation of the observed application for obtaining its samples but rather uses the utilities of the JVM to do so.

2.3 Sampling profiling possible implementations for the JVM

2.3.1 `jstack`

`jstack` is an utility included in the Oracle's JDK and OpenJDK by default. This utility can be used to print out stack traces of all threads of a Java process [8]. The simplest implementation to show the concept of a sampling profiler would be to periodically call the `jstack` utility to obtain the snapshot of all threads. These snapshots can be persisted and processed to produce a visualization of the application's performance. This approach has a relatively high overhead when compared to other sampling profiling methods and is not a viable method for profiling real world applications but rather illustrates the concept of sampling profiling.

2.3.2 `GetAllStackTraces` profilers

`GetAllStackTraces` is a method in the official JVM Tooling Interface (JVMTI) which is a programming interface utilized by development and monitoring tools. This method enables the profiler to extract the stack traces of all the threads currently executing on the JVM [9]. Downside of this method is the fact that profilers utilizing this method are safepoint biased since calling this method requires the JVM to have reached a safepoint. For this reason, the performance is negatively affected due to frequent pauses for reaching safepoints. Due to safepoint induced performance problems, this method produces significantly less samples when compared to profilers based on `AsyncGetCallTrace` as described in Section 2.3.3. `GetAllStackTraces` method is used in most commercial profilers such as JVisualVM, YourKit and JProfiler [4, 10].

2.3.3 `AsyncGetCallTrace` profilers

These kind of profilers make use of an undocumented JVMTI method `AsyncGetCallTrace` which enables obtaining call traces from a thread without the safepoint bias [11]. The asynchronous nature of this method enables the profiler to safely call this method in a signal handler which is the key architectural element for these kind of profilers [12].

Due to safepoint bias free sample obtaining method, profiling samples tend to be more accurate since the JVM does not have to stop on a safepoint in order to

obtain the call trace of the running thread.

Notable examples of such profilers which utilize `AsyncGetCallTrace` method are Honest Profiler and Java Mission Control.

2.3.4 Native profilers

Native profilers are tools utilizing the low level abstractions of the operating system to profile the native binaries for the operating system. These profilers are superior to the previously described methods performance wise but fall short on the usefulness of the gathered information. The main downside of native profilers is the inability obtain sufficient information about the Java level stack frames interpreted within the Java Virtual Machine. Unlike other sampling techiniques introduced, native profilers are not much use to profile applications running on the Java Virtual Machine where most of the code has not been compiled into native machine code by JIT compilation yet.

3 Honest Profiler

Honest Profiler is a software written in Java and C++ which aims to gather honest and accurate samples from a running Java application to provide an overview of the application's performance [13]. Honest Profiler uses the `AsyncGetCallTrace` method for obtaining the samples. Moreover, Honest Profiler is licensed under an open source license which encourages the community to further investigate and develop this software solution. The fact that Honest Profiler is an open source software and utilizes a safepoint bias free method for obtaining the samples are the main reasons why this particular software is chosen for improvements and optimizations in this thesis.

3.1 Architecture

Honest Profiler consists of two larger components: a JVMTI native agent and a Java facade for result visualization, analysis and transformation. The profiling is almost completely done by the agent whereas the Java facade component aims to provide tools to ease the use of the profiler.

The JVMTI agent is a dynamic library which serves as a client of the JVM Tooling Interface [9]. JVMTI provides these agents means to investigate and control the applications running on the Java Virtual Machine. Since the agents are written in native languages such as C and C++, they can also utilize various operating system's tools.

Honest Profiler relies on UNIX operating system signals and timers for obtaining samples. It sets up a signal handler to handle `SIGPROF` signals sent to the Java process that the agent is attached to and an interval timer which sends the `SIGPROF` signal to that same process periodically.

Upon receiving a signal, an *arbitrary* thread currently executing on the CPU will handle the signal [14]. The signal handler proceeds to call `AsyncGetCallTrace` method from the JVMTI API to obtain the call trace of the current thread which is handling the signal. The obtained call trace is then persisted.

The thread handling the `SIGPROF` signal can be any thread of the Java process being profiled, even the threads responsible for garbage collection and just-in-time compilation. In such cases, the samples are discarded as such samples do not provide informative results. Samples which contain the observed application's stack frames are considered useful and provide relevant information about the application's performance.

3.2 Current limitations

Current implementation of Honest Profiler makes use of the standard CPU time based interval timer from GNU time library and UNIX operating system signals to periodically call the `AsyncGetCallTrace` method. CPU time based timer counts down time only when the process is consuming CPU time which means that the timer's value will only be incremented when the process is executing its instructions on the processor.

The lowest possible interval for the CPU based interval timer is the time between two ticks of the system's internal timer interrupt (*jiffy*). This also implies that the lowest possible `AsyncGetCallTrace` method call interval is also the duration of a jiffy. This duration depends on the interrupt frequency of the hardware platform being used. Default interrupt frequency for Linux kernel based operating systems (since kernel version 2.6.13) is 250 Hz which results in $\frac{1}{250} = 0.004$ seconds for the duration of a jiffy [15]. In such configuration, minimum sampling interval would be 4 milliseconds.

Therefore, increasing the sampling ratio will require significant changes in the architecture of how timer signals are sent to the profiler. The following sections will describe that different implementations not only perform differently in terms of overhead but also in terms of which thread ends up receiving timer signals. Hence, the ratio of usable stack traces versus those that have to be discarded are different for each implementation.

4 Honest Profiler performance analysis

Prior to increasing the sampling frequency, it is important to understand what are the performance limitations that calling the `AsyncGetCallTrace` would imply. It is necessary to understand how long does a call to the `AsyncGetCallTrace` method take to ensure that increasing the sampling frequency will not cause excessive performance overhead to the application being profiled. In order to measure the execution time, a custom Java Virtual Machine build with time measurement logic based on CPU time was added to the `AsyncGetCallTrace` method. The added timing functionality is presented in Listing 5.

```
void AsyncGetCallTrace(ASGCT_CallTrace *trace, jint depth, void* ucontext) {
    struct timespec start, stop;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start);

    // AsyncGetCallTrace method code

    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &stop);
    long accum = stop.tv_nsec -start.tv_nsec;
    printf("%li", accum);
}
```

Listing 5: CPU time based measurement in `AsyncGetCallTrace` method

Testing shows that a single `AsyncGetCallTrace` method call takes 4.6 microseconds on average.

Previously described limitations and `AsyncGetCallTrace` method performance measurements do not expose apparent problems or reasons on why `AsyncGetCallTrace` method can not be called more frequently during profiling. As the `AsyncGetCallTrace` method execution time is significantly smaller when compared to the default lowest achieveable sampling interval, the hypothesis is that that having smaller profiling interval will not affect the observed application's performance to a great extent. Thus, achieving higher sampling frequency is a goal worth pursuing.

5 Profiler optimization

This sections describes three different approaches for increasing the sampling frequency for Honest Profiler. To verify the performance overhead of the solution and its accuracy, a benchmark test was executed on each solution.

5.1 Benchmark

Benchmark test reads 10000000 random integers from an input file and then performs quicksort on the copy of the obtained list 10 times. For each solution, the benchmark is executed 10 times. Benchmark test code is included in Appendix A.

Baseline of the test using the default implementation of the Honest Profiler finished the test in 59.83 seconds on average. During profiling, 19025 samples were obtained on average from which 15130 (81%) contained useful and informative frames. Profiling results are visualized on the Figure 5 on page 17.

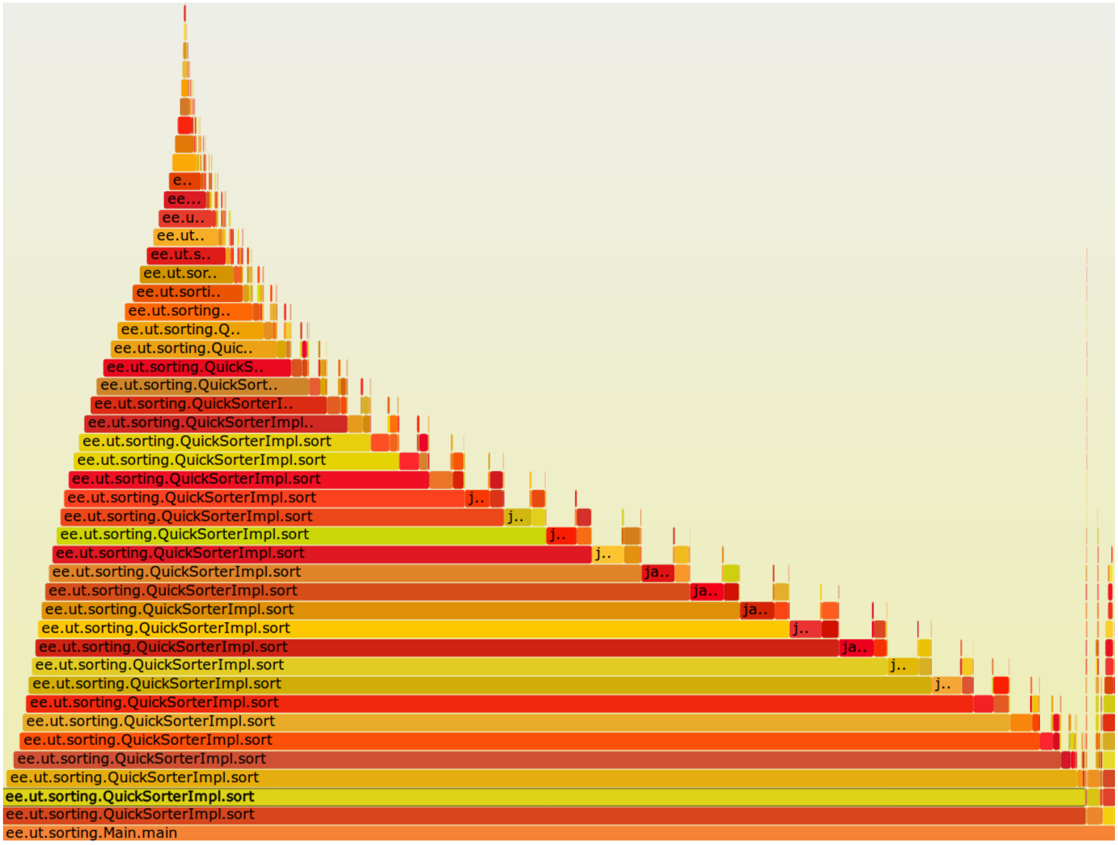

Figure 5: Visualization of baseline benchmark profiling results

5.2 Optimization implementations

Following sections will describe the implementations of the solutions more in-depth. The source code for the solutions can be found in the Appendix B.

5.2.1 Interval timer based on real time

Existing implementation of Honest Profiler uses CPU time based interval timer from GNU time library. Alternative approach would be to use an interval timer which counts time in real time instead. The same interval timer from GNU time library can be configured in such way that it counts down in real time by initilizing it with `ITIMER_REAL` flag. Upon timer expiration, a `SIGALRM` signal is sent to the process [16].

Using real time based timer means that its interval is not limited by the system's

interrupt frequency as it would be for CPU time based timers. Due to this fact, the timer can even be configured to send the `SIGALRM` signal every 1 microsecond.

Although this approach obtains considerably larger amount of samples due to higher profiling frequency, it also introduces a potential bias that could affect the profiling results. It is possible that between two consecutive timer expiration signals, the application under observation has not received any processor time to perform its instructions. This would imply that the same sample in the exact same state is persisted multiple times.

Similar issue occurs the other way around when the application's instructions are executed on multiple processor cores. There is a clear distinction in cases in which 4 processor cores are fully utilized when executing the application's instructions and cases in which a single processor core is used. The former scenario would get roughly four times as much work done when compared to the latter scenario. However, the real time based timer will not take these workload differences into account because it measures time based on the real world clock. This bias potentially introduces inaccuracy in the profiling results.

Average benchmark's execution time using this method finished in 70.78 seconds and obtained up to 12.7 million (12723206) samples during its execution from which only 73982 (0.58%) contained frames of interest. Whereas the 15% increase in execution times is acceptable for gaining significantly larger amount of samples, it is unknown why only 0.58% of the gathered samples contained frames of interest. This phenomenom is also further covered in the future research Section 6. Visualization of the profiling results utilizing this method is presented in Figure 6 on page 19.

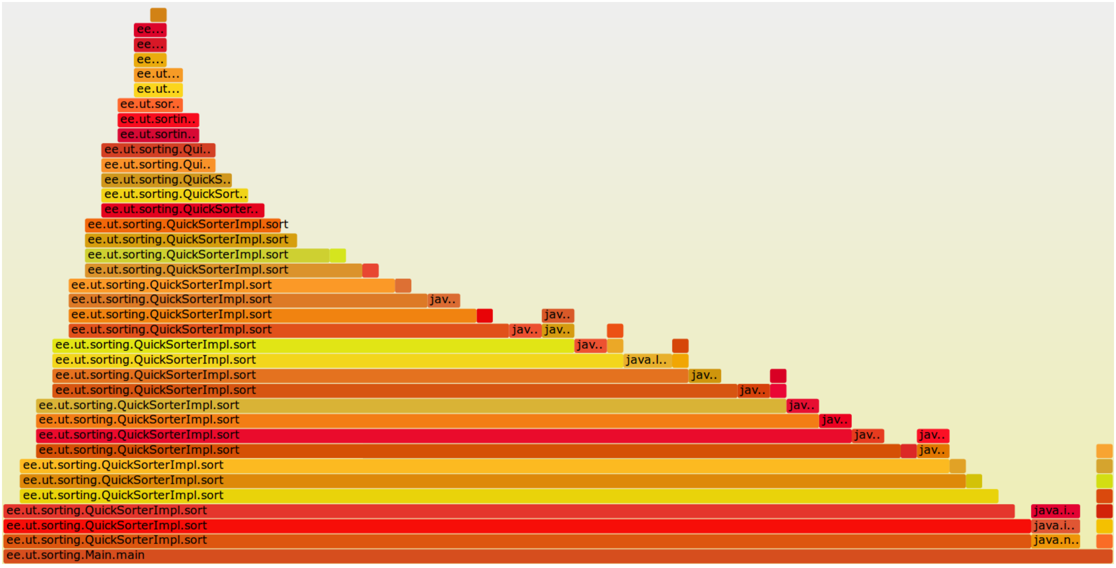

Figure 6: Visualization of real time based solution profiling results

5.2.2 Increasing the kernel clock frequency

As CPU time based interval timer from GNU timer library is directly dependent on the kernel's internal timer frequency, one approach to gather more samples would be to build a custom kernel with increased kernel clock frequency. By default this frequency is set to 250 Hz [17]. This value can be increased to achieve higher sampling frequency.

This can be accomplished by creating a custom configuration in the `Kconfig.hz` with a desired frequency value and building the kernel using the created configuration. An easier way to obtain a kernel with higher kernel timer frequency would be to install a `linux-lowlatency` kernel package which has set the kernel timer frequency to 1000 Hz.

Using the 1000 Hz kernel increases the sampling frequency as the CPU time based interval timer from GNU time library can now send an interrupt signal every $\frac{1}{1000} =$ 0.001 seconds. Experiments with such kernel has shown four times increase in the amount of samples gathered wihtout causing significant performance overhead to the application under observation.

Benchmark runs on such configuration had average execution time of 61.82 seconds during which 70525 samples were collected. 50937 of the samples (72%) were useful and contained the application's stack frames in them. When comparing to the baseline bechmark results, this solution resulted in 336% increase in useful

samples with negligible performance overhead. Visualization from the benchmark's execution using such profiling configuration is shown on Figure 7.

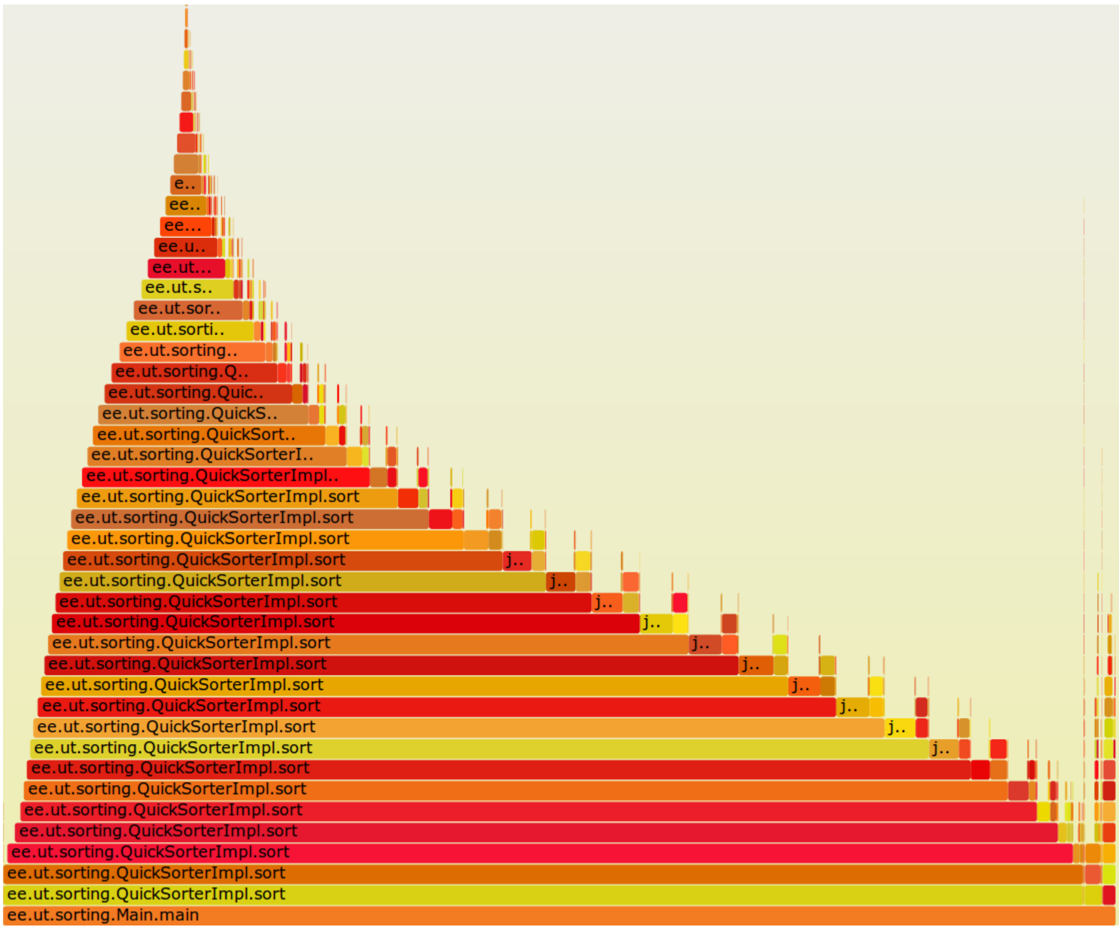

Figure 7: Visualization of the results from profiling with increased kernel timer frequency

5.2.3 External utility for timekeeping

This approach moves the time measurement logic to an external server process which uses UNIX operating system signals and shared memory to communicate with Honest Profiler and the profileable Java process. The solution is based on the fact that it is possible for a process to obtain the CPU time that the process has used without the limitations imposed by the kernel's internal timer frequency. The CPU time spent by a process can be obtained by calling the `clock_gettime` function from the GNU time library. Essentially, the external utility serves as a

CPU time based timer which enables calling `AsyncGetCallTrace` method more often than default implementation.

The external server process starts by creating a UNIX domain socket and waits for a connection. Upon a connection from a Java process with Honest Profiler's native agent attached, it obtains its process identifier (*PID*) for signal sending purposes. The server process then allocates a memory section which is shared with the profileable Java process. Server will then use two signals to request action from the Honest Profiler that is attached to the Java process:

- `SIGALRM` - write the used CPU time used by the Java process to the shared memory
- `SIGPROF` - call the `AsyncGetCallTrace` method to obtain a sample from the thread handling the signal

Shared memory for interprocess communication is used due to its superior performance over other communication approaches such as file I/O (input/output) or UNIX domain sockets.

Figure 8 summarizes the architecture of this implementation.

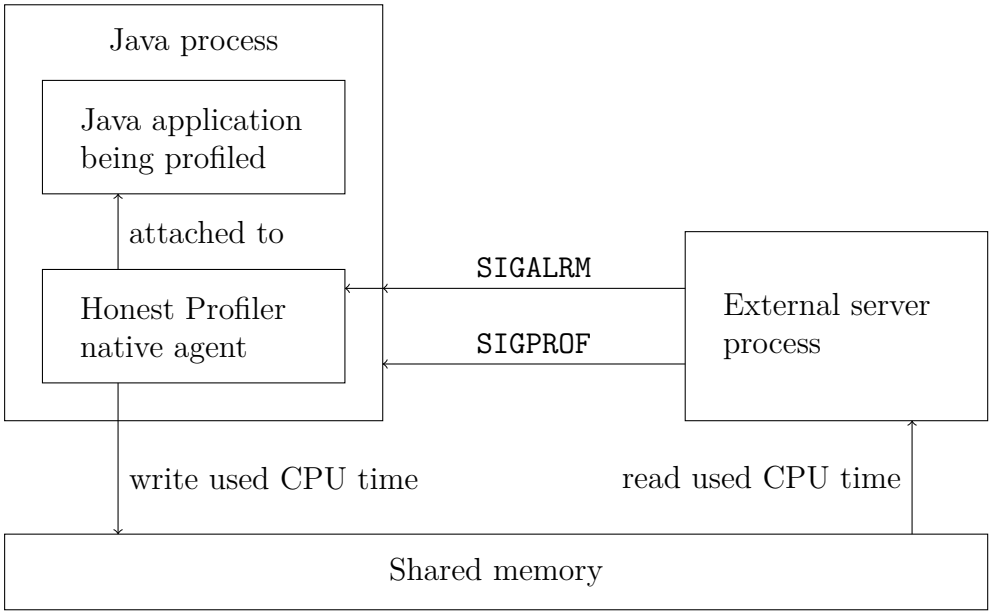

Figure 8: Architecture of the external timekeeping based solution

Testing shows that such modification for Honest Profiler runs the benchmark test in 63.85 seconds on average. From running the benchmark test, a sample was

obtained roughly after every 100 microseconds. 746868 samples were obtained during profiling and 38775 (5.2%) of them were useful. This approach obtained 256% more useful samples than the baseline benchmark.

Visualization of the profiling results from the benchmark run using this solution is presented in Figure 9.

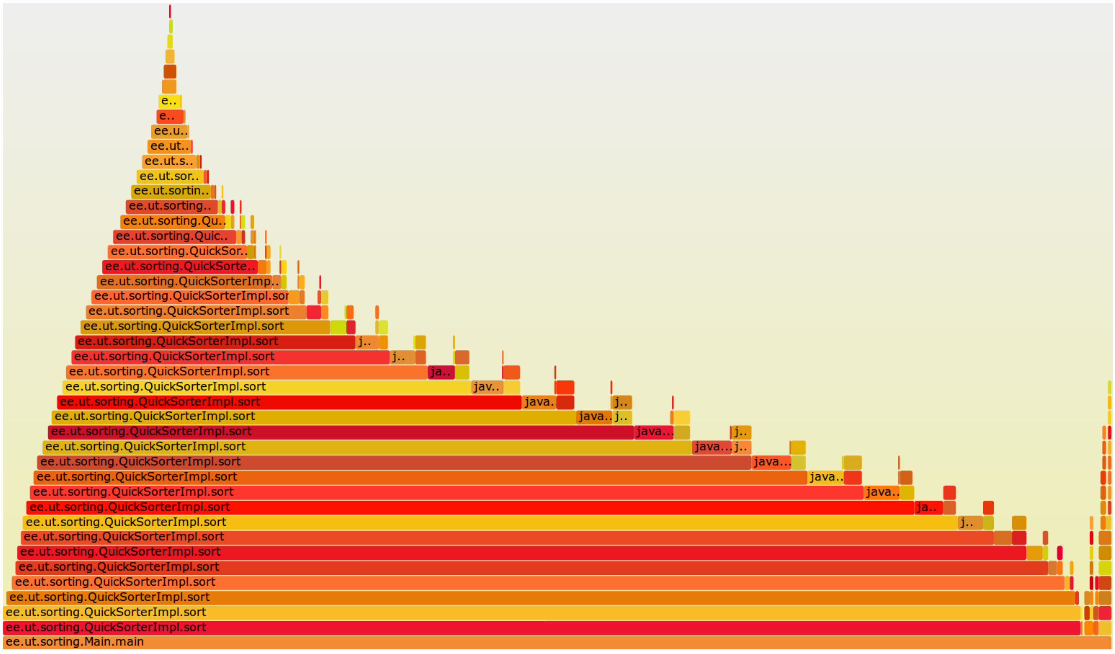

Figure 9: Visualization of the external timekeeping based solution profiling results

This approach is slighty less cumbersome to set up than installing a custom kernel as demonstrated in Section 5.2.2.

5.3 Solutions' comparison

All of the proposed solutions succeeded in extracting a larger amount of information from the benchmark test application when compared to the baseline benchmark run. The benchmark results for all the solutions are shown in Table 1 on page 23.

Table 1: Benchmark results of the solutions

	Benchmark time	Gathered samples	Useful samples
Baseline	59.83	19025	15130
Real time approach	70.78	12723206	73982
Frequency increase	61.82	70525	50937
External timekeeping	63.85	746868	38775

It can be seen that the real time based and the external timekeeping solutions have a different distribution of useful and non-useful samples compared to original implementation. Ideas describing this phenomenom are briefly covered in the Section 6.

6 Future research

This thesis focuses on increasing the useful information output from the observed application but leaves evaluating the accuracy of the provided solutions out of scope. For future studies, investigating the accuracy of the provided solutions based on the ideas brought out in the Mytkowicz's paper on profiler accuracy evaluation [1] could improve the practical value of these solutions.

Additionally, during testing the solutions, an anomaly with large amount of unusable samples occurred. Namely, since the `SIGPROF` signal sent to the profileable process is sent to an *arbitrary* thread then it is possible that the signal is handled by a thread which is not able to provide a call trace sample that could be used in results' visualization. Unusable sample is obtained when the `AsyncGetCallTrace` method returns a negative return value which mostly happens in the following cases:

1. `SIGPROF` signal is sent to a non-Java thread such as native threads for just-in-time compilation and garbage collection
2. `SIGPROF` signal is sent to a Java thread while garbage collection is active
3. `SIGPROF` signal is sent to a thread which is exiting
4. `SIGPROF` signal is sent to a thread whose stack trace is not walkable

Future studies could investigate the signals emitted and investigate the distribution of threads which are handling the signal and block the threads from handling the profiling signal as they will not provide usable stack traces for profiling results. As part of an experiment, patching the main thread loop in the Java Virtual Machine initialization was done. The experiment proved to decrease the amount of unusable samples obtained by a noticeable margin due to the fact that this thread can not handle the profiling signal sent by Honest Profiler. A promising idea would be to further investigate the initialization of other native threads within Java Virtual Machine and disable handling of the profiling signal.

7 Conclusion

The thesis thorougly explains the ideas behind sampling profiling and describes the problems that could affect such profiling method. It brings out the importance of sufficient samples' quantity and describes safepoint bias, periodicity bias and observer effect. The thesis then dissects the internals of Honest Profiler, describes its benefits over other profiling solutions and explains why it is a viable choice as a profiler when compared to other existing profiling solutions. The performance of Honest Profiler's information acquiring logic is measured. The performance measurements did not reveal any fundamental reasons on why acquiring a greater amount of samples is not possible. The thesis then investigates alternative methods to increase the amount of information that Honest Profiler can extract from the observed application.

The main result of this thesis are three solutions which increase the profiling information output wihtout causing noticeable performance overhead. The solutions' performance and extracted information amount was tested on a benchmark test to verify the results. The first approach used real time based time measurement instead of CPU time based time measurement. The second solution focused on increasing the system's interrupt frequency which, in consequence, increased the amount of samples extracted. As a part of the last solution, an external timekeeping utility was implemented which served as a high frequency CPU time based timer for the Honest Profiler. All of the solutions managed to increase the amount of useful samples extracted from the profileable application.

The provided solutions proved to produce reliable results with increased information output and negligible performance overhead. Two of the provided solutions have a different distribution of useful and non-useful samples when compared to the original implementation. Ideas for investigating the distribution differences are presented in the future research Section 6

References

[1] Mytkowicz T. et al. Evaluating the accuracy of Java profilers. ACM SIGPLAN Notices: ACM. 1–11, 2010. http://portal.acm.org/citation.cfm?doid=1809028.1806618 [23.03.2018].

[2] B. Gregg. Flame Graphs. http://www.brendangregg.com/flamegraphs.html [23.04.2018].

[3] Oracle Corporation. HotSpot Glossary of Terms. http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html [27.03.2018].

[4] Wakart N. Psychosomatic, Lobotomy, Saw: Why (Most) Sampling Java Profilers Are Fucking Terrible. https://psy-lob-saw.blogspot.com.ee/2016/02/why-most-sampling-java-profilers-are.html [23.04.2018].

[5] Wakart N. Psychosomatic, Lobotomy, Saw: Safepoints: Meaning, Side Effects and Overheads. https://psy-lob-saw.blogspot.com.ee/2015/12/safepoints.html [23.04.2018].

[6] Evans B. Understanding Java JIT Compilation with JITWatch, Part 1. http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html [09.05.2018].

[7] Whitham J. Profiling versus tracing, 2016. https://www.jwhitham.org/2016/02/profiling-versus-tracing.html [03.05.2018].

[8] Oracle Corporation. jstack documentation. https://docs.oracle.com/javase/7/docs/technotes/tools/share/jstack.html [24.04.2018].

[9] Oracle Corporation. JVM(TM) Tool Interface 1.2.3 documentation. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html [01.05.2018].

[10] VisualVM. Source code repository. http://visualvm.sourcearchive.com/documentation/0.20080728/Stacks_8c-source.html [30.04.2018].

[11] Oracle and its affiliates. OpenJDK, jdk8. Source code repository. http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/prims/forte.cpp#l513 [01.03.2018].

[12] signal-safety(7). Linux manual page. http://man7.org/linux/man-pages/man7/signal-safety.7.html [08.05.2018].

[13] Warburton R. Honest Profiler. GitHub repository. https://github.com/jvm-profiling-tools/honest-profiler [21.02.2018].

[14] Stevens W.R. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2013.

[15] time(7). Linux manual page. `http://man7.org/linux/man-pages/man7/time.7.html` [26.02.2018].

[16] getitimer(2). Linux manual page. `http://man7.org/linux/man-pages/man2/setitimer.2.html` [26.02.2018].

[17] Torvalds L. Linux kernel source tree. GitHub repository. `https://github.com/torvalds/linux` [03.05.2018].

[18] Wakart N. Java Profiling from the Ground Up. `https://zeroturnaround.com/rebellabs/java-profiling-from-the-ground-up-by-nitsan-wakart/` [29.04.2018].

Appendices

A Benchmark test code

All tests are performed on a machine running Intel i5-5200U processor with 8 GB of RAM.

```
package ee.ut.sorting;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.stream.Collectors;
import java.util.*;

public class Main {
  public static void main(String[] args) throws IOException {
    long totalStart = System.nanoTime();
    List<Integer> sortableList = Files.readAllLines(Paths.get("/tmp/integers.data"),
        StandardCharsets.UTF_8)
            .stream().mapToInt(x -> Integer.parseInt(x.trim())).boxed().collect(Collectors.
                toList());
    for (int i = 0; i < 10; i++) {
      QuickSorterImpl<Integer> integerQuickSorter = new QuickSorterImpl<>();
      long start = System.nanoTime();
      integerQuickSorter.sort(new ArrayList<>(sortableList));
      long end = System.nanoTime() -start;
      System.out.println(end);
    }
    long totalEnd = System.nanoTime() -totalStart;
    System.out.println(totalEnd);
  }
}
```

B Source code

All optimization implementations for Honest profiler are available at the following GitHub repository in separate branches:

`https://github.com/Oja95/honest-profiler`

External server implementation for the shared memory solution described in Section 5.2.3 is available at the following GitHub respository:

`https://github.com/Oja95/honest-profiler-shared-mem-server`

Non-exclusive licence to reproduce thesis and make thesis public

I, **Tiit Oja**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   Optimizing JVM profiling performance for Honest Profiler,

   supervised by **Vootele Rõtov** and **Vesal Vojdani**,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 13.05.2018